Ingeniería del software

Un enfoque práctico

Séptima edición

Roger S. Pressman

Mc Graw Hill

Ingeniería del software

UN ENFOQUE PRÁCTICO

Ingeniería del software

UN ENFOQUE PRÁCTICO

SÉPTIMA EDICIÓN

Roger S. Pressman, Ph.D.
University of Connecticut

MÉXICO • BOGOTÁ • BUENOS AIRES • CARACAS • GUATEMALA • MADRID
NUEVA YORK • SAN JUAN • SANTIAGO • SÃO PAULO • AUCKLAND • LONDRES • MILÁN
MONTREAL • NUEVA DELHI • SAN FRANCISCO • SINGAPUR • ST. LOUIS • SIDNEY • TORONTO

Director Higher Education: Miguel Ángel Toledo Castellanos
Editor sponsor: Pablo Roig Vázquez
Coordinadora editorial: Marcela I. Rocha Martínez
Editora de desarrollo: María Teresa Zapata Terrazas
Supervisor de producción: Zeferino García García

Traductores: Víctor Campos Olguín
Javier Enríquez Brito
Revisión técnica: Carlos Villegas Quezada
Bárbaro Jorge Ferro Castro

INGENIERÍA DEL SOFTWARE. UN ENFOQUE PRÁCTICO
Séptima edición

Prohibida la reproducción total o parcial de esta obra,
por cualquier medio, sin la autorización escrita del editor.

Educación

DERECHOS RESERVADOS © 2010, 2005, 2002 respecto a la tercera edición en español por
McGRAW-HILL INTERAMERICANA EDITORES, S.A. DE C.V.
A Subsidiary of ***The McGraw-Hill*** *Companies, Inc.*
Prolongación Paseo de la Reforma 1015, Torre A
Piso 17, Colonia Desarrollo Santa Fe,
Delegación Álvaro Obregón
C.P. 01376, México, D. F.
Miembro de la Cámara Nacional de la Industria Editorial Mexicana, Reg. Núm. 736

ISBN: 978-607-15-0314-5
(ISBN edición anterior: 970-10-5473-3)

Traducido de la séptima edición de SOFTWARE ENGINEERING. A PRACTITIONER'S APPROACH.
Published by McGraw-Hill, a business unit of The McGraw-Hill Companies, Inc., 1221 Avenue of the Americas, New York, NY 10020. Copyright © 2010 by The McGraw-Hill Companies, Inc. All rights reserved.
978-0-07-337597-7

1234567890 109876543210

Impreso en México *Printed in Mexico*

En recuerdo de mi querido padre,
quien vivió 94 años y me enseñó,
sobre todo, que la honestidad
y la integridad eran las mejores
guías para mi viaje por la vida.

Acerca del autor

Roger S. Pressman es una autoridad internacionalmente reconocida en el mejoramiento del proceso del software y en las tecnologías de la ingeniería del mismo. Durante casi cuatro décadas ha trabajado como ingeniero de software, gestor, profesor, escritor y consultor, especializado en temas de ingeniería del software.

Como profesional y gestor industrial, el doctor Pressman trabajó en el desarrollo de sistemas CAD/CAM para aplicaciones de ingeniería y fabricación avanzadas. También ha tenido posiciones de responsabilidad en la programación científica y de sistemas.

Después de recibir su doctorado en ingeniería por parte de la Universidad de Connecticut, Pressman se dedicó a la academia, donde se convirtió en profesor asociado de la cátedra Bullard en ingeniería de cómputo de la Universidad de Bridgeport, y en director del Centro de Diseño y Fabricación Asistidos por Computadora de dicha universidad.

En la actualidad, el doctor Pressman es presidente de R. S. Pressman & Associates, Inc., una empresa de consultoría especializada en métodos y capacitación en ingeniería del software. Trabaja como consultor principal y diseñó y desarrolló *Ingeniería del software esencial*, un video curricular completo acerca de ingeniería del software, y *Consultor de procesos*, un sistema autodirigido para el mejoramiento del proceso de software. Ambos productos los utilizan miles de compañías en todo el mundo. Más recientemente, trabajó en colaboración con *EdistaLearning*, en India, para desarrollar capacitación abarcadora basada en internet acerca de ingeniería del software.

El doctor Pressman ha escrito muchos artículos técnicos, es colaborador regular en revistas periódicas industriales y autor de siete libros técnicos. Además de *Ingeniería del software: un enfoque práctico*, es coautor de *Web Engineering* (McGraw-Hill), uno de los primeros libros en aplicar un conjunto personalizado de principios y prácticas de la ingeniería del software al desarrollo de sistemas y aplicaciones basados en web. También escribió el premiado *A Manager's Guide to Software Engineering* (McGraw-Hill); *Making Software Engineering Happen* (Prentice hall), el primer libro en abordar los problemas administrativos cruciales asociados con el mejoramiento del proceso de software; y *Software Shock* (Dorset House), un tratamiento que se enfoca en el software y su impacto en los negocios y la sociedad. Pressman ha formado parte de los consejos editoriales de varias publicaciones industriales y durante muchos años fue editor de la columna "Manager" en *IEEE Software*.

Además, es un orador bien conocido, y ha sido el orador principal en muchas conferencias industriales importantes. Es miembro de IEEE, y de Tau Beta Pi, Phi Kappa Phi, Eta Kappa Nu y Pi Tau Sigma.

En el lado personal, Pressman vive en el sur de Florida con su esposa, Bárbara. Atleta de toda la vida, sigue siendo un serio jugador de tenis (4.5 en el programa estadounidense de calificación de tenis, NTRP) y un golfista con un *handicap* de un solo dígito. En su tiempo libre escribió dos novelas, *Aymara Bridge* y *The Puppeteer*, y tiene planes para escribir una más.

Contenido breve

CAPÍTULO 1 El software y la ingeniería de software 1

PARTE UNO **EL PROCESO DEL SOFTWARE** 25

CAPÍTULO 2 Modelos del proceso 26
CAPÍTULO 3 Desarrollo ágil 55

PARTE DOS **MODELADO** 81

CAPÍTULO 4 Principios que guían la práctica 82
CAPÍTULO 5 Comprensión de los requerimientos 101
CAPÍTULO 6 Modelado de los requerimientos: escenarios, información y clases de análisis 126
CAPÍTULO 7 Modelado de los requerimientos: flujo, comportamiento, patrones y webapps 158
CAPÍTULO 8 Conceptos de diseño 183
CAPÍTULO 9 Diseño de la arquitectura 206
CAPÍTULO 10 Diseño en el nivel de componentes 234
CAPÍTULO 11 Diseño de la interfaz de usuario 265
CAPÍTULO 12 Diseño basado en patrones 295
CAPÍTULO 13 Diseño de webapps 317

PARTE TRES **ADMINISTRACIÓN DE LA CALIDAD** 337

CAPÍTULO 14 Conceptos de calidad 338
CAPÍTULO 15 Técnicas de revisión 354
CAPÍTULO 16 Aseguramiento de la calidad del software 368
CAPÍTULO 17 Estrategias de prueba de software 383
CAPÍTULO 18 Prueba de aplicaciones convencionales 411
CAPÍTULO 19 Prueba de aplicaciones orientadas a objetos 437
CAPÍTULO 20 Prueba de aplicaciones web 453
CAPÍTULO 21 Modelado y verificación formal 478
CAPÍTULO 22 Administración de la configuración del software 501
CAPÍTULO 23 Métricas de producto 526

PARTE CUATRO **ADMINISTRACIÓN DE PROYECTOS DE SOFTWARE** 553

CAPÍTULO 24 Conceptos de administración de proyecto 554
CAPÍTULO 25 Métricas de proceso y de proyecto 571
CAPÍTULO 26 Estimación para proyectos de software 593
CAPÍTULO 27 Calendarización del proyecto 620
CAPÍTULO 28 Administración del riesgo 640
CAPÍTULO 29 Mantenimiento y reingeniería 655

PARTE CINCO **TEMAS AVANZADOS** 675

CAPÍTULO 30 Mejoramiento del proceso de software 676
CAPÍTULO 31 Tendencias emergentes en ingeniería del software 695
CAPÍTULO 32 Comentarios finales 717

APÉNDICE 1 Introducción a UML 725
APÉNDICE 2 Conceptos orientados a objeto 743
REFERENCIAS 751
ÍNDICE ANALÍTICO 767

CONTENIDO

Prefacio xxv

CAPÍTULO 1 EL SOFTWARE Y LA INGENIERÍA DE SOFTWARE 1

1.1 La naturaleza del software 2
1.1.1 Definición de software 3
1.1.2 Dominios de aplicación del software 6
1.1.3 Software heredado 8
1.2 La naturaleza única de las webapps 9
1.3 Ingeniería de software 10
1.4 El proceso del software 12
1.5 La práctica de la ingeniería de software 15
1.5.1 La esencia de la práctica 15
1.5.2 Principios generales 16
1.6 Mitos del software 18
1.7 Cómo comienza todo 20
1.8 Resumen 21
PROBLEMAS Y PUNTOS POR EVALUAR 21
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 22

PARTE UNO EL PROCESO DEL SOFTWARE 25

CAPÍTULO 2 MODELOS DEL PROCESO 26

2.1 Un modelo general de proceso 27
2.1.1 Definición de actividad estructural 29
2.1.2 Identificación de un conjunto de tareas 29
2.1.3 Patrones del proceso 29
2.2 Evaluación y mejora del proceso 31
2.3 Modelos de proceso prescriptivo 33
2.3.1 Modelo de la cascada 33
2.3.2 Modelos de proceso incremental 35
2.3.3 Modelos de proceso evolutivo 36
2.3.4 Modelos concurrentes 40
2.3.5 Una última palabra acerca de los procesos evolutivos 42
2.4 Modelos de proceso especializado 43
2.4.1 Desarrollo basado en componentes 43
2.4.2 El modelo de métodos formales 44
2.4.3 Desarrollo de software orientado a aspectos 44
2.5 El proceso unificado 45
2.5.1 Breve historia 46
2.5.2 Fases del proceso unificado 46
2.6 Modelos del proceso personal y del equipo 48
2.6.1 Proceso personal del software (PPS) 48
2.6.2 Proceso del equipo de software (PES) 49
2.7 Tecnología del proceso 50
2.8 Producto y proceso 51
2.9 Resumen 52
PROBLEMAS Y PUNTOS POR EVALUAR 53
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 54

CAPÍTULO 3 DESARROLLO ÁGIL 55

3.1 ¿Qué es la agilidad? 56
3.2 La agilidad y el costo del cambio 57
3.3 ¿Qué es un proceso ágil? 58
3.3.1 Principios de agilidad 58
3.3.2 La política del desarrollo ágil 59
3.3.3 Factores humanos 60
3.4 Programación extrema (XP) 61
3.4.1 Valores XP 61
3.4.2 El proceso XP 62
3.4.3 XP industrial 65
3.4.4 El debate XP 66
3.5 Otros modelos ágiles de proceso 67
3.5.1 Desarrollo adaptativo de software (DAS) 68
3.5.2 Scrum 69
3.5.3 Método de desarrollo de sistemas dinámicos (MDSD) 71
3.5.4 Cristal 72
3.5.5 Desarrollo impulsado por las características (DIC) 72
3.5.6 Desarrollo esbelto de software (DES) 73
3.5.7 Modelado ágil (MA) 74
3.5.8 El proceso unificado ágil (PUA) 75
3.6 Conjunto de herramientas para el proceso ágil 76
3.7 Resumen 77
PROBLEMAS Y PUNTOS POR EVALUAR 78
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 79

PARTE DOS MODELADO 81

CAPÍTULO 4 PRINCIPIOS QUE GUÍAN LA PRÁCTICA 82

4.1 Conocimiento de la ingeniería de software 83
4.2 Principios fundamentales 83
4.2.1 Principios que guían el proceso 84
4.2.2 Principios que guían la práctica 84
4.3 Principios que guían toda actividad estructural 86
4.3.1 Principios de comunicación 86
4.3.2 Principios de planeación 88
4.3.3 Principios de modelado 90
4.3.4 Principios de construcción 94
4.3.5 Principios de despliegue 96
4.4 Resumen 97
PROBLEMAS Y PUNTOS POR EVALUAR 98
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 99

CAPÍTULO 5 COMPRENSIÓN DE LOS REQUERIMIENTOS 101

5.1 Ingeniería de requerimientos 102
5.2 Establecer las bases 106
5.2.1 Identificación de los participantes 106
5.2.2 Reconocer los múltiples puntos de vista 107
5.2.3 Trabajar hacia la colaboración 107
5.2.4 Hacer las primeras preguntas 108
5.3 Indagación de los requerimientos 108
5.3.1 Recabación de los requerimientos en forma colaborativa 109
5.3.2 Despliegue de la función de calidad 111
5.3.3 Escenarios de uso 112
5.3.4 Indagación de los productos del trabajo 112

5.4 Desarrollo de casos de uso 113
5.5 Elaboración del modelo de los requerimientos 117
5.5.1 Elementos del modelo de requerimientos 118
5.5.2 Patrones de análisis 120
5.6 Requerimientos de las negociaciones 121
5.7 Validación de los requerimientos 122
5.8 Resumen 123
PROBLEMAS Y PUNTOS POR EVALUAR 123
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 124

CAPÍTULO 6 MODELADO DE LOS REQUERIMIENTOS: ESCENARIOS, INFORMACIÓN Y CLASES DE ANÁLISIS 126

6.1 Análisis de los requerimientos 127
6.1.1 Objetivos y filosofía general 128
6.1.2 Reglas prácticas del análisis 128
6.1.3 Análisis del dominio 129
6.1.4 Enfoques del modelado de requerimientos 130
6.2 Modelado basado en escenarios 131
6.2.1 Creación de un caso preliminar de uso 132
6.2.2 Mejora de un caso de uso preliminar 134
6.2.3 Escritura de un caso de uso formal 135
6.3 Modelos UML que proporcionan el caso de uso 137
6.3.1 Desarrollo de un diagrama de actividades 137
6.3.2 Diagramas de canal (*swimlane*) 138
6.4 Conceptos de modelado de datos 139
6.4.1 Objetos de datos 139
6.4.2 Atributos de los datos 140
6.4.3 Relaciones 141
6.5 Modelado basado en clases 142
6.5.1 Identificación de las clases de análisis 143
6.5.2 Especificación de atributos 145
6.5.3 Definición de las operaciones 146
6.5.4 Modelado clase-responsabilidad-colaborador (CRC) 148
6.5.5 Asociaciones y dependencias 152
6.5.6 Paquetes de análisis 154
6.6 Resumen 155
PROBLEMAS Y PUNTOS POR EVALUAR 156
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 157

CAPÍTULO 7 MODELADO DE LOS REQUERIMIENTOS: FLUJO, COMPORTAMIENTO, PATRONES Y WEBAPPS 158

7.1 Requerimientos que modelan las estrategias 158
7.2 Modelado orientado al flujo 159
7.2.1 Creación de un modelo de flujo de datos 159
7.2.2 Creación de un modelo de flujo de control 162
7.2.3 La especificación de control 162
7.2.4 La especificación del proceso 163
7.3 Creación de un modelo de comportamiento 165
7.3.1 Identificar los eventos con el caso de uso 166
7.3.2 Representaciones de estado 166
7.4 Patrones para el modelado de requerimientos 169
7.4.1 Descubrimiento de patrones de análisis 169
7.4.2 Ejemplo de patrón de requerimientos: Actuador-Sensor 170
7.5 Modelado de requerimientos para webapps 174
7.5.1 ¿Cuánto análisis es suficiente? 174
7.5.2 Entrada del modelado de los requerimientos 174

7.5.3 Salida del modelado de los requerimientos 175
7.5.4 Modelo del contenido de las webapps 176
7.5.5 Modelo de la interacción para webapps 177
7.5.6 Modelo funcional para las webapps 178
7.5.7 Modelos de configuración para las webapps 179
7.5.8 Modelado de la navegación 180
7.6 Resumen 180
PROBLEMAS Y PUNTOS POR EVALUAR 181
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 182

CAPÍTULO 8 CONCEPTOS DE DISEÑO 183

8.1 Diseño en el contexto de la ingeniería de software 184
8.2 El proceso de diseño 186
8.2.1 Lineamientos y atributos de la calidad del software 186
8.2.2 La evolución del diseño del software 188
8.3 Conceptos de diseño 189
8.3.1 Abstracción 189
8.3.2 Arquitectura 190
8.3.3 Patrones 191
8.3.4 División de problemas 191
8.3.5 Modularidad 191
8.3.6 Ocultamiento de información 192
8.3.7 Independencia funcional 193
8.3.8 Refinamiento 194
8.3.9 Aspectos 194
8.3.10 Rediseño 195
8.3.11 Conceptos de diseño orientados a objeto 195
8.3.12 Clases de diseño 196
8.4 El modelo del diseño 197
8.4.1 Elementos del diseño de datos 199
8.4.2 Elementos del diseño arquitectónico 199
8.4.3 Elementos de diseño de la interfaz 199
8.4.4 Elementos del diseño en el nivel de los componentes 201
8.4.5 Elementos del diseño del despliegue 202
8.5 Resumen 203
PROBLEMAS Y PUNTOS POR EVALUAR 203
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 204

CAPÍTULO 9 DISEÑO DE LA ARQUITECTURA 206

9.1 Arquitectura del software 207
9.1.1 ¿Qué es la arquitectura? 207
9.1.2 ¿Por qué es importante la arquitectura? 208
9.1.3 Descripciones arquitectónicas 208
9.1.4 Decisiones arquitectónicas 209
9.2 Géneros arquitectónicos 209
9.3 Estilos arquitectónicos 211
9.3.1 Breve taxonomía de estilos de arquitectura 213
9.3.2 Patrones arquitectónicos 215
9.3.3 Organización y refinamiento 216
9.4 Diseño arquitectónico 217
9.4.1 Representación del sistema en contexto 217
9.4.2 Definición de arquetipos 218
9.4.3 Refinamiento de la arquitectura hacia los componentes 219
9.4.4 Descripción de las instancias del sistema 220

9.5 Evaluación de los diseños alternativos para la arquitectura 221
9.5.1 Método de la negociación para analizar la arquitectura 222
9.5.2 Complejidad arquitectónica 224
9.5.3 Lenguajes de descripción arquitectónica 224
9.6 Mapeo de la arquitectura con el uso del flujo de datos 225
9.6.1 Mapeo de transformación 225
9.6.2 Refinamiento del diseño arquitectónico 231
9.7 Resumen 232
PROBLEMAS Y PUNTOS POR EVALUAR 232
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 233

CAPÍTULO 10 DISEÑO EN EL NIVEL DE COMPONENTES 234

10.1 ¿Qué es un componente? 235
10.1.1 Una visión orientada a objetos 235
10.1.2 La visión tradicional 236
10.1.3 Visión relacionada con el proceso 239
10.2 Diseño de componentes basados en clase 239
10.2.1 Principios básicos del diseño 239
10.2.2 Lineamientos de diseño en el nivel de componentes 242
10.2.3 Cohesión 243
10.2.4 Acoplamiento 244
10.3 Realización del diseño en el nivel de componentes 246
10.4 Diseño en el nivel de componentes para webapps 251
10.4.1 Diseño del contenido en el nivel de componente 251
10.4.2 Diseño de las funciones en el nivel de componentes 252
10.5 Diseño de componentes tradicionales 252
10.5.1 Notación gráfica de diseño 253
10.5.2 Notación del diseño tabular 254
10.5.3 Lenguaje de diseño del programa 255
10.6 Desarrollo basado en componentes 256
10.6.1 Ingeniería del dominio 257
10.6.2 Calificación, adaptación y combinación de los componentes 257
10.6.3 Análisis y diseño para la reutilización 259
10.6.4 Clasificación y recuperación de componentes 260
10.7 Resumen 262
PROBLEMAS Y PUNTOS POR EVALUAR 263
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 263

CAPÍTULO 11 DISEÑO DE LA INTERFAZ DE USUARIO 265

11.1 Las reglas doradas 266
11.1.1 Dejar el control al usuario 266
11.1.2 Reducir la necesidad de que el usuario memorice 267
11.1.3 Hacer consistente la interfaz 268
11.2 Análisis y diseño de la interfaz de usuario 269
11.2.1 Análisis y modelos del diseño de la interfaz 269
11.2.2 El proceso 271
11.3 Análisis de la interfaz 272
11.3.1 Análisis del usuario 272
11.3.2 Análisis y modelado de la tarea 273
11.3.3 Análisis del contenido de la pantalla 277
11.3.4 Análisis del ambiente de trabajo 278
11.4 Etapas del diseño de la interfaz 278
11.4.1 Aplicación de las etapas de diseño de la interfaz 279
11.4.2 Patrones de diseño de la interfaz de usuario 280
11.4.3 Aspectos del diseño 281

11.5 Diseño de una interfaz para webapps 284
11.5.1 Principios y lineamientos del diseño de la interfaz 285
11.5.2 Flujo de trabajos para el diseño de la interfaz de webapp 289
11.6 Evaluación del diseño 290
11.7 Resumen 292
PROBLEMAS Y PUNTOS POR EVALUAR 293
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 293

CAPÍTULO 12 DISEÑO BASADO EN PATRONES 295

12.1 Patrones de diseño 296
12.1.1 Clases de patrones 297
12.1.2 Estructuras 299
12.1.3 Descripción de un patrón 299
12.1.4 Lenguajes y repositorios de patrones 300
12.2 Diseño de software basado en patrones 301
12.2.1 El diseño basado en patrones, en contexto 301
12.2.2 Pensar en patrones 302
12.2.3 Tareas de diseño 303
12.2.4 Construcción de una tabla para organizar el patrón 305
12.2.5 Errores comunes en el diseño 305
12.3 Patrones arquitectónicos 306
12.4 Patrones de diseño en el nivel de componentes 308
12.5 Patrones de diseño de la interfaz de usuario 310
12.6 Patrones de diseño de webapp 313
12.6.1 Centrarse en el diseño 313
12.6.2 Granularidad del diseño 314
12.7 Resumen 315
PROBLEMAS Y PUNTOS POR EVALUAR 315
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 316

CAPÍTULO 13 DISEÑO DE WEBAPPS 317

13.1 Calidad del diseño de webapps 318
13.2 Metas del diseño 320
13.3 Pirámide del diseño de webapps 321
13.4 Diseño de la interfaz de la webapp 321
13.5 Diseño de la estética 323
13.5.1 Aspectos de la distribución 323
13.5.2 Aspectos del diseño gráfico 324
13.6 Diseño del contenido 324
13.6.1 Objetos de contenido 324
13.6.2 Aspectos de diseño del contenido 325
13.7 Diseño arquitectónico 326
13.7.1 Arquitectura del contenido 326
13.7.2 Arquitectura de las webapps 328
13.8 Diseño de la navegación 329
13.8.1 Semántica de la navegación 329
13.8.2 Sintaxis de navegación 330
13.9 Diseño en el nivel de componentes 331
13.10 Método de diseño de hipermedios orientado a objetos (MDHOO) 332
13.10.1 Diseño conceptual del MDHOO 332
13.10.2 Diseño de la navegación para el MDHOO 333
13.10.3 Diseño abstracto de la interfaz y su implementación 333
13.11 Resumen 334
PROBLEMAS Y PUNTOS POR EVALUAR 335
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 335

PARTE TRES ADMINISTRACIÓN DE LA CALIDAD 337

CAPÍTULO 14 CONCEPTOS DE CALIDAD 338

14.1 ¿Qué es calidad? 339
14.2 Calidad del software 340
14.2.1 Dimensiones de la calidad de Garvin 341
14.2.2 Factores de la calidad de McCall 342
14.2.3 Factores de la calidad ISO 9126 343
14.2.4 Factores de calidad que se persiguen 343
14.2.5 Transición a un punto de vista cuantitativo 344
14.3 El dilema de la calidad del software 345
14.3.1 Software "suficientemente bueno" 345
14.3.2 El costo de la calidad 346
14.3.3 Riesgos 348
14.3.4 Negligencia y responsabilidad 348
14.3.5 Calidad y seguridad 349
14.3.6 El efecto de las acciones de la administración 349
14.4 Lograr la calidad del software 350
14.4.1 Métodos de la ingeniería de software 350
14.4.2 Técnicas de administración de proyectos 350
14.4.3 Control de calidad 351
14.4.4 Aseguramiento de la calidad 351
14.5 Resumen 351
PROBLEMAS Y PUNTOS POR EVALUAR 352
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 352

CAPÍTULO 15 TÉCNICAS DE REVISIÓN 354

15.1 Efecto de los defectos del software en el costo 355
15.2 Amplificación y eliminación del defecto 356
15.3 Métricas de revisión y su empleo 357
15.3.1 Análisis de las métricas 358
15.3.2 Eficacia del costo de las revisiones 358
15.4 Revisiones: espectro de formalidad 359
15.5 Revisiones informales 361
15.6 Revisiones técnicas formales 362
15.6.1 La reunión de revisión 363
15.6.2 Reporte y registro de la revisión 363
15.6.3 Lineamientos para la revisión 364
15.6.4 Revisiones orientadas al muestreo 365
15.7 Resumen 366
PROBLEMAS Y PUNTOS POR EVALUAR 367
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 367

CAPÍTULO 16 ASEGURAMIENTO DE LA CALIDAD DEL SOFTWARE 368

16.1 Antecedentes 369
16.2 Elementos de aseguramiento de la calidad del software 370
16.3 Tareas, metas y métricas del ACS 371
16.3.1 Tareas del ACS 371
16.3.2 Metas, atributos y métricas 372
16.4 Enfoques formales al ACS 373
16.5 Aseguramiento estadístico de la calidad del software 374
16.5.1 Ejemplo general 374
16.5.2 Seis Sigma para la ingeniería de software 375
16.6 Confiabilidad del software 376
16.6.1 Mediciones de la confiabilidad y disponibilidad 377
16.6.2 Seguridad del software 378

16.7 Las normas de calidad ISO 9000 378
16.8 El plan de ACS 379
16.9 Resumen 380
PROBLEMAS Y PUNTOS POR EVALUAR 381
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 381

CAPÍTULO 17 ESTRATEGIAS DE PRUEBA DE SOFTWARE 383

17.1 Un enfoque estratégico para la prueba de software 384
17.1.1 Verificación y validación 384
17.1.2 Organización de las pruebas del software 385
17.1.3 Estrategia de prueba del software. Visión general 386
17.1.4 Criterios para completar las pruebas 388
17.2 Aspectos estratégicos 388
17.3 Estrategias de prueba para software convencional 389
17.3.1 Prueba de unidad 389
17.3.2 Pruebas de integración 391
17.4 Estrategias de prueba para software orientado a objeto 397
17.4.1 Prueba de unidad en el contexto OO 397
17.4.2 Prueba de integración en el contexto OO 398
17.5 Estrategias de prueba para webapps 398
17.6 Pruebas de validación 399
17.6.1 Criterios de pruebas de validación 399
17.6.2 Revisión de la configuración 400
17.6.3 Pruebas alfa y beta 400
17.7 Pruebas del sistema 401
17.7.1 Pruebas de recuperación 401
17.7.2 Pruebas de seguridad 402
17.7.3 Pruebas de esfuerzo 402
17.7.4 Pruebas de rendimiento 403
17.7.5 Pruebas de despliegue 403
17.8 El arte de la depuración 404
17.8.1 El proceso de depuración 404
17.8.2 Consideraciones psicológicas 405
17.8.3 Estrategias de depuración 406
17.8.4 Corrección del error 408
17.9 Resumen 408
PROBLEMAS Y PUNTOS POR EVALUAR 409
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 409

CAPÍTULO 18 PRUEBA DE APLICACIONES CONVENCIONALES 411

18.1 Fundamentos de las pruebas del software 412
18.2 Visiones interna y externa de las pruebas 413
18.3 Prueba de caja blanca 414
18.4 Prueba de ruta básica 414
18.4.1 Notación de gráfico o grafo de flujo 415
18.4.2 Rutas de programa independientes 416
18.4.3 Derivación de casos de prueba 418
18.4.4 Matrices de grafo 420
18.5 Prueba de la estructura de control 420
18.5.1 Prueba de condición 421
18.5.2 Prueba de flujo de datos 421
18.5.3 Prueba de bucle 421
18.6 Pruebas de caja negra 423
18.6.1 Métodos de prueba basados en gráficos 423
18.6.2 Partición de equivalencia 425

18.6.3 Análisis de valor de frontera 425
18.6.4 Prueba de arreglo ortogonal 426
18.7 Prueba basada en modelo 429
18.8 Prueba para entornos, arquitecturas y aplicaciones especializados 429
18.8.1 Pruebas de interfaces gráficas de usuario 430
18.8.2 Prueba de arquitecturas cliente-servidor 430
18.8.3 Documentación de prueba y centros de ayuda 431
18.8.4 Prueba para sistemas de tiempo real 432
18.9 Patrones para pruebas de software 433
18.10 Resumen 434
PROBLEMAS Y PUNTOS POR EVALUAR 435
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 436

CAPÍTULO 19 PRUEBA DE APLICACIONES ORIENTADAS A OBJETOS 437

19.1 Ampliación de la definición de las pruebas 438
19.2 Modelos de prueba AOO y DOO 439
19.2.1 Exactitud de los modelos AOO y DOO 439
19.2.2 Consistencia de los modelos orientados a objetos 439
19.3 Estrategias de pruebas orientadas a objetos 441
19.3.1 Prueba de unidad en el contexto OO 441
19.3.2 Prueba de integración en el contexto OO 442
19.3.3 Prueba de validación en un contexto OO 442
19.4 Métodos de prueba orientada a objetos 442
19.4.1 Implicaciones del diseño de casos de prueba de los conceptos OO 443
19.4.2 Aplicabilidad de los métodos convencionales de diseño de casos de prueba 443
19.4.3 Prueba basada en fallo 444
19.4.4 Casos de prueba y jerarquía de clase 444
19.4.5 Diseño de pruebas basadas en escenario 445
19.4.6 Pruebas de las estructuras superficial y profunda 446
19.5 Métodos de prueba aplicables en el nivel clase 447
19.5.1 Prueba aleatoria para clases OO 447
19.5.2 Prueba de partición en el nivel de clase 448
19.6 Diseño de casos de prueba interclase 448
19.6.1 Prueba de clase múltiple 449
19.6.2 Pruebas derivadas a partir de modelos de comportamiento 450
19.7 Resumen 451
PROBLEMAS Y PUNTOS POR EVALUAR 451
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 452

CAPÍTULO 20 PRUEBA DE APLICACIONES WEB 453

20.1 Conceptos de pruebas para aplicaciones web 453
20.1.1 Dimensiones de calidad 454
20.1.2 Errores dentro de un entorno de *webapp* 455
20.1.3 Estrategia de las pruebas 455
20.1.4 Planificación de pruebas 456
20.2 Un panorama del proceso de prueba 456
20.3 Prueba de contenido 457
20.3.1 Objetivos de la prueba de contenido 457
20.3.2 Prueba de base de datos 458
20.4 Prueba de interfaz de usuario 460
20.4.1 Estrategia de prueba de interfaz 460
20.4.2 Prueba de mecanismos de interfaz 461
20.4.3 Prueba de la semántica de la interfaz 463
20.4.4 Pruebas de usabilidad 463
20.4.5 Pruebas de compatibilidad 465
20.5 Prueba en el nivel de componente 466

20.6 Prueba de navegación 467
20.6.1 Prueba de sintaxis de navegación 467
20.6.2 Prueba de la semántica de navegación 468
20.7 Prueba de configuración 469
20.7.1 Conflictos en el lado servidor 469
20.7.2 Conflictos en el lado cliente 470
20.8 Prueba de seguridad 470
20.9 Prueba de rendimiento 471
20.9.1 Objetivos de la prueba de rendimiento 472
20.9.2 Prueba de carga 472
20.9.3 Prueba de esfuerzo 473
20.10 Resumen 475
PROBLEMAS Y PUNTOS POR EVALUAR 475
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 476

CAPÍTULO 21 MODELADO Y VERIFICACIÓN FORMAL 478

21.1 Estrategia de cuarto limpio 479
21.2 Especificación funcional 480
21.2.1 Especificación de caja negra 482
21.2.2 Especificación de caja de estado 482
21.2.3 Especificación de caja clara 483
21.3 Diseño de cuarto limpio 483
21.3.1 Refinamiento de diseño 483
21.3.2 Verificación de diseño 484
21.4 Pruebas de cuarto limpio 485
21.4.1 Pruebas de uso estadístico 486
21.4.2 Certificación 487
21.5 Conceptos de métodos formales 487
21.6 Aplicación de notación matemática para especificación formal 490
21.7 Lenguajes de especificación formal 492
21.7.1 Lenguaje de restricción de objeto (OCL) 492
21.7.2 El lenguaje de especificación Z 495
21.8 Resumen 498
PROBLEMAS Y PUNTOS POR EVALUAR 499
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 500

CAPÍTULO 22 ADMINISTRACIÓN DE LA CONFIGURACIÓN DEL SOFTWARE 501

22.1 Administración de la configuración del software 502
22.1.1 Un escenario ACS 502
22.1.2 Elementos de un sistema de administración de la configuración 503
22.1.3 Líneas de referencia 504
22.1.4 Ítems de configuración del software 505
22.2 El repositorio ACS 506
22.2.1 El papel del repositorio 506
22.2.2 Características y contenido generales 507
22.2.3 Características ACS 507
22.3 El proceso ACS 508
22.3.1 Identificación de objetos en la configuración del software 509
22.3.2 Control de versión 510
22.3.3 Control de cambio 511
22.3.4 Auditoría de configuración 514
22.3.5 Reporte de estado 515
22.4 Administración de la configuración para *webapps* 515
22.4.1 Conflictos dominantes 516
22.4.2 Objetos de configuración de *webapps* 517
22.4.3 Administración de contenido 517

22.4.4 Administración del cambio 520
22.4.5 Control de versión 522
22.4.6 Auditoría y reporte 522
22.5 Resumen 523
PROBLEMAS Y PUNTOS POR EVALUAR 524
LECTURAS ADICIONALES Y FUENTES DE INFORMACIÓN 525

CAPÍTULO 23 MÉTRICAS DE PRODUCTO 526

23.1 Marco conceptual para las métricas de producto 527
23.1.1 Medidas, métricas e indicadores 527
23.1.2 El reto de la métrica de producto 527
23.1.3 Principios de medición 528
23.1.4 Medición de software orientado a meta 529
23.1.5 Atributos de las métricas de software efectivas 530
23.2 Métricas para el modelo de requerimientos 531
23.2.1 Métrica basada en funciones 531
23.2.2 Métricas para calidad de la especificación 534
23.3 Métricas para el modelo de diseño 535
23.3.1 Métricas del diseño arquitectónico 535
23.3.2 Métricas para diseño orientado a objetos 537
23.3.3 Métricas orientadas a clase: la suite de métricas CK 539
23.3.4 Métricas orientadas a clase: La suite de métricas MOOD 541
23.3.5 Métricas OO propuestas por Lorenz y Kidd 542
23.3.6 Métricas de diseño en el nivel de componente 542
23.3.7 Métricas orientadas a operación 544
23.3.8 Métricas de diseño de interfaz de usuario 545
23.4 Métricas de diseño para *webapps* 545
23.5 Métricas para código fuente 547
23.6 Métricas para pruebas 548
23.6.1 Métricas de Halstead aplicadas para probar 549
23.6.2 Métricas para pruebas orientadas a objetos 549
23.7 Métricas para mantenimiento 550
23.8 Resumen 551
PROBLEMAS Y PUNTOS POR EVALUAR 551
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 552

PARTE CUATRO ADMINISTRACIÓN DE PROYECTOS DE SOFTWARE 553

CAPÍTULO 24 CONCEPTOS DE ADMINISTRACIÓN DE PROYECTO 554

24.1 El espectro administrativo 555
24.1.1 El personal 555
24.1.2 El producto 555
24.1.3 El proceso 556
24.1.4 El proyecto 556
24.2 El personal 556
24.2.1 Los participantes 557
24.2.2 Líderes de equipo 557
24.2.3 El equipo de software 558
24.2.4 Equipos ágiles 561
24.2.5 Conflictos de coordinación y comunicación 561
24.3 El producto 562
24.3.1 Ámbito del software 562
24.3.2 Descomposición del problema 563
24.4 El proceso 563
24.4.1 Fusión de producto y proceso 564
24.4.2 Descomposición del proceso 564

24.5 El proyecto 566
24.6 El principio W5HH 567
24.7 Prácticas cruciales 567
24.8 Resumen 568
PROBLEMAS Y PUNTOS POR EVALUAR 569
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 569

CAPÍTULO 25 MÉTRICAS DE PROCESO Y DE PROYECTO 571

25.1 Métricas en los dominios de proceso y proyecto 572
25.1.1 Las métricas del proceso y la mejora del proceso de software 572
25.1.2 Métricas de proyecto 574
25.2 Medición del software 575
25.2.1 Métricas orientadas a tamaño 576
25.2.2 Métricas orientadas a función 577
25.2.3 Reconciliación de métricas LOC y PF 577
25.2.4 Métricas orientadas a objeto 579
25.2.5 Métricas orientadas a caso de uso 580
25.2.6 Métricas de proyecto *webapp* 580
25.3 Métricas para calidad de software 582
25.3.1 Medición de la calidad 583
25.3.2 Eficiencia en la remoción del defecto 584
25.4 Integración de métricas dentro del proceso de software 585
25.4.1 Argumentos para métricas de software 585
25.4.2 Establecimiento de una línea de referencia 586
25.4.3 Recolección, cálculo y evaluación de métricas 586
25.5 Métricas para organizaciones pequeñas 587
25.6 Establecimiento de un programa de métricas del software 588
25.7 Resumen 590
PROBLEMAS Y PUNTOS POR EVALUAR 590
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 591

CAPÍTULO 26 ESTIMACIÓN PARA PROYECTOS DE SOFTWARE 593

26.1 Observaciones acerca de las estimaciones 594
26.2 El proceso de planificación del proyecto 595
26.3 Ámbito y factibilidad del software 595
26.4 Recursos 596
26.4.1 Recursos humanos 596
26.4.2 Recursos de software reutilizables 597
26.4.3 Recursos ambientales 598
26.5 Estimación de proyectos de software 598
26.6 Técnicas de descomposición 599
26.6.1 Dimensionamiento del software 599
26.6.2 Estimación basada en problema 600
26.6.3 Un ejemplo de estimación basada en LOC 601
26.6.4 Un ejemplo de estimación basada en PF 602
26.6.5 Estimación basada en proceso 604
26.6.6 Un ejemplo de estimación basada en proceso 605
26.6.7 Estimación con casos de uso 605
26.6.8 Un ejemplo de estimación basada en caso de uso 606
26.6.9 Reconciliación de estimaciones 607
26.7 Modelos de estimación empíricos 608
26.7.1 La estructura de los modelos de estimación 608
26.7.2 El modelo COCOMO II 609
26.7.3 La ecuación del software 610

26.8 Estimación para proyectos orientados a objetos 611
26.9 Técnicas de estimación especializadas 612
26.9.1 Estimación para desarrollo ágil 612
26.9.2 Estimación para webapp 613
26.10 La decisión hacer/comprar 614
26.10.1 Creación de un árbol de decisión 615
26.10.2 *Outsourcing* 616
26.11 Resumen 617
PROBLEMAS Y PUNTOS POR EVALUAR 617
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 618

CAPÍTULO 27 CALENDARIZACIÓN DEL PROYECTO 620

27.1 Conceptos básicos 621
27.2 Calendarización del proyecto 622
27.2.1 Principios básicos 623
27.2.2 Relación entre personal y esfuerzo 624
27.2.3 Distribución de esfuerzo 625
27.3 Definición de un conjunto de tareas para el proyecto de software 626
27.3.1 Un ejemplo de conjunto de tareas 627
27.3.2 Refinamiento de acciones de ingeniería del software 627
27.4 Definición de una red de tareas 628
27.5 Calendarización 629
27.5.1 Cronogramas 629
27.5.2 Seguimiento del calendario 631
27.5.3 Seguimiento del progreso para un proyecto OO 632
27.5.4 Calendarización para proyectos webapp 633
27.6 Análisis de valor ganado 635
27.7 Resumen 637
PROBLEMAS Y PUNTOS POR EVALUAR 637
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 638

CAPÍTULO 28 ADMINISTRACIÓN DEL RIESGO 640

28.1 Estrategias reactivas de riesgo frente a estrategias proactivas de riesgo 641
28.2 Riesgos de software 641
28.3 Identificación de riesgos 642
28.3.1 Valoración del riesgo de proyecto global 643
28.3.2 Componentes y promotores de riesgo 644
28.4 Proyección del riesgo 644
28.4.1 Elaboración de una lista de riesgos 645
28.4.2 Valoración de impacto de riesgo 647
28.5 Refinamiento del riesgo 649
28.6 Mitigación, monitoreo y manejo de riesgo 649
28.7 El plan MMMR 651
28.8 Resumen 652
PROBLEMAS Y PUNTOS POR EVALUAR 653
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 653

CAPÍTULO 29 MANTENIMIENTO Y REINGENIERÍA 655

29.1 Mantenimiento de software 656
29.2 Soportabilidad del software 657
29.3 Reingeniería 658
29.4 Reingeniería de procesos de empresa 658
29.4.1 Procesos empresariales 659
29.4.2 Un modelo RPE 659

29.5 Reingeniería de software 661
29.5.1 Un modelo de proceso de reingeniería de software 661
29.5.2 Actividades de reingeniería de software 662
29.6 Ingeniería inversa 664
29.6.1 Ingeniería inversa para comprender datos 665
29.6.2 Ingeniería inversa para entender el procesamiento 666
29.6.3 Ingeniería inversa de interfaces de usuario 667
29.7 Reestructuración 668
29.7.1 Reestructuración de código 668
29.7.2 Reestructuración de datos 668
29.8 Ingeniería hacia adelante 669
29.8.1 Ingeniería hacia adelante para arquitecturas cliente-servidor 670
29.8.2 Ingeniería hacia adelante para arquitecturas orientadas a objetos 671
29.9 Economía de la reingeniería 671
29.10 Resumen 672
PROBLEMAS Y PUNTOS POR EVALUAR 673
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 674

PARTE CINCO TEMAS AVANZADOS 675

CAPÍTULO 30 MEJORAMIENTO DEL PROCESO DE SOFTWARE 676

30.1 ¿Qué es mps? 677
30.1.1 Enfoques del MPS 677
30.1.2 Modelos de madurez 679
30.1.3 ¿El MPS es para todos? 680
30.2 El proceso MPS 680
30.2.1 Valoración y análisis de la desviación 681
30.2.2 Educación y capacitación 682
30.2.3 Selección y justificación 682
30.2.4 Instalación/migración 683
30.2.5 Evaluación 683
30.2.6 Gestión del riesgo para MPS 684
30.2.7 Factores de éxito cruciales 685
30.3 El CMMI 685
30.4 El CMM de personal 688
30.5 Otros marcos conceptuales MPS 689
30.6 Rendimiento sobre inversión de MPS 691
30.7 Tendencias MPS 692
30.8 Resumen 693
PROBLEMAS Y PUNTOS POR EVALUAR 693
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 694

CAPÍTULO 31 TENDENCIAS EMERGENTES EN INGENIERÍA DEL SOFTWARE 695

31.1 Evolución tecnológica 696
31.2 Observación de las tendencias en ingeniería del software 697
31.3 Identificación de "tendencias blandas" 699
31.3.1 Administración de la complejidad 700
31.3.2 Software de mundo abierto 701
31.3.3 Requerimientos emergentes 701
31.3.4 La mezcla de talento 702
31.3.5 Bloques constructores de software 703
31.3.6 Cambio de percepciones de "valor" 703
31.3.7 Fuente abierta 703
31.4 Direcciones de la tecnología 704
31.4.1 Tendencias de proceso 705
31.4.2 El gran desafío 706

31.4.3 Desarrollo colaborativo 707
31.4.4 Ingeniería de requerimientos 708
31.4.5 Desarrollo de software impulsado por modelo 709
31.4.6 Diseño posmoderno 710
31.4.7 Desarrollo impulsado por pruebas 710
31.5 Tendencias relacionadas con herramientas 711
31.5.1 Herramientas que responden a tendencias blandas 712
31.5.2 Herramientas que abordan tendencias tecnológicas 714
31.6 Resumen 714
PROBLEMAS Y PUNTOS POR EVALUAR 715
LECTURAS Y FUENTES DE INFORMACIÓN ADICIONALES 715

CAPÍTULO 32 COMENTARIOS FINALES 717

32.1 La importancia del software-revisión 718
32.2 Las personas y la forma en la que construyen sistemas 718
32.3 Nuevos modos para representar la información 719
32.4 La vista larga 720
32.5 La responsabilidad del ingeniero de software 721
32.6 Un comentario final 722

APÉNDICE 1 Introducción a UML 725
APÉNDICE 2 Conceptos orientados a objeto 743
REFERENCIAS 751
ÍNDICE ANALÍTICO 767

Prefacio

Cuando el software de computadora triunfa (al satisfacer las necesidades de las personas que lo usan, trabajar sin fallos durante largos periodos, será fácil de modificar e incluso más fácil de usar) puede y debe cambiar las cosas a fin de mejorar. Pero cuando el software fracasa (cuando sus usuarios no están satisfechos, es proclive al error, es difícil de cambiar e incluso más difícil de usar) pueden ocurrir, y ocurren, cosas malas. Todo mundo quiere construir software que haga mejor las cosas y que evite las malas que acechan en la sombra de los esfuerzos fallidos. Para triunfar, se necesita disciplina al momento de diseñar y construir el software. Es necesario un enfoque de ingeniería.

Han pasado casi tres décadas desde que se escribió la primera edición de este libro. Durante ese tiempo, la ingeniería del software evolucionó desde una oscura idea practicada por un número relativamente pequeño de fanáticos hasta una legítima disciplina de la ingeniería. En la actualidad, se le reconoce como una materia merecedora de investigación seria, estudio concienzudo y debate turbulento. A lo largo de toda la industria, el ingeniero de software sustituyó al programador como el título laboral de preferencia. Los modelos de proceso de software, los métodos de ingeniería de software y las herramientas del software se adoptaron exitosamente a través de un amplio espectro de segmentos industriales.

Aunque los gestores y profesionales reconocen por igual la necesidad de un enfoque del software más disciplinado, continúan debatiendo la forma en la que la disciplina debe aplicarse. Muchos individuos y compañías todavía desarrollan el software de manera fortuita, incluso cuando construyen sistemas para atender las tecnologías más avanzadas de la actualidad. Muchos profesionales y estudiantes no están conscientes de los métodos modernos. Como resultado, la calidad del software que producen es deficiente y ocurren cosas malas. Además, continúa el debate y la controversia en torno de la verdadera naturaleza del enfoque de la ingeniería del software. El estatus de la ingeniería del software es un estudio en contrastes. Las actitudes han cambiado, se ha progresado, pero todavía falta mucho por hacer antes de que la disciplina alcance madurez plena.

La séptima edición de *Ingeniería del software: un enfoque práctico* tiene la intención de funcionar como guía hacia una disciplina de ingeniería que madura. Como las seis ediciones que la precedieron, la séptima se dirige a estudiantes y profesionales, y conserva su atractivo como guía para el profesional industrial y como introducción abarcadora para el estudiante en los niveles superiores de pregrado o en el primer año de graduado.

La séptima edición es considerablemente más que una simple actualización. El libro se revisó y reestructuró para mejorar el flujo pedagógico y enfatizar nuevos e importantes procesos y prácticas de la ingeniería del software. Además, este texto cuenta con un paquete de complementos, los cuales están disponibles para los profesores que lo adopten. Consulte con el representante de McGraw-Hill local.

La séptima edición. Los 32 capítulos de la séptima edición se reorganizaron en cinco partes. Esta organización, que difiere considerablemente de la sexta edición, se realizó para dividir mejor los temas y ayudar a los profesores que tal vez no tengan tiempo para completar todo el libro en un semestre.

La parte 1, *El proceso*, presenta varias visiones diferentes del proceso de software, considera todos los modelos de proceso importantes y aborda el debate entre las filosofías de proceso

prescriptivo y ágil. La parte 2, *Modelado*, presenta los métodos de análisis y diseño con énfasis en las técnicas orientadas a objeto y al modelado UML. También se considera el diseño basado en patrón y el diseño para aplicaciones web. La parte 3, *Gestión de la calidad*, presenta los conceptos, procedimientos, técnicas y métodos que permiten a un equipo de software valorar la calidad del software, revisar los productos de trabajo de la ingeniería del software, realizar procedimientos SQA y aplicar una estrategia y tácticas de prueba efectivas. Además, también se considera el modelado formal y los métodos de verificación. La parte 4, *Gestión de proyectos de software*, presenta temas que son relevantes a quienes planean, gestionan y controlan un proyecto de desarrollo de software. La parte 5, *Temas avanzados*, considera el mejoramiento del proceso de software y las tendencias en la ingeniería del software. Al continuar con la tradición de las ediciones pasadas, a lo largo del libro se usa una serie de recuadros para presentar las experiencias y tribulaciones de un equipo de software (ficticio) y para proporcionar materiales complementarios acerca de los métodos y herramientas que son relevantes para los temas del capítulo. Dos nuevos apéndices proporcionan breves tutoriales acerca del UML y del pensamiento orientado a objeto para quienes no estén familiarizados con estos importantes temas.

La organización en cinco partes de la séptima edición permite al profesor "englobar" los temas con base en el tiempo disponible y las necesidades del estudiante. Un curso de todo un semestre podría construirse en torno de uno o más de las cinco partes. Uno de evaluación de ingeniería del software seleccionaría capítulos de las cinco. Uno de ingeniería del software que enfatice el análisis y el diseño elegiría temas de las partes 1 y 2. Un curso de ingeniería del software orientado a pruebas seleccionaría temas de las partes 1 y 3, con una breve incursión en la parte 2. Un "curso administrativo" subrayaría las partes 1 y 4.

Reconocimientos. Mi trabajo en las siete ediciones de *Ingeniería del software: un enfoque práctico* ha sido el proyecto técnico continuo más largo de mi vida. Aun cuando la escritura cesó, la información extraída de la literatura técnica continúa asimilándose y organizándose, y las críticas y sugerencias de los lectores en todo el mundo se evalúan y catalogan. Por esta razón, agradezco a los muchos autores de libros, ponencias y artículos (tanto en copia dura como en medios electrónicos) que me han proporcionado comprensión, ideas y comentarios adicionales durante casi 30 años.

Agradezco especialmente a Tim Lethbridge, de la Universidad de Ottawa, quien me auxilió en el desarrollo de los ejemplos UML y OCL, y quien desarrolló el estudio de caso que acompaña a este libro, y a Dale Skrien, de Colby College, quien desarrolló el tutorial UML en el apéndice 1. Su asistencia y sus comentarios fueron invaluables. Un agradecimiento especial también para Bruce Maxim, de la Universidad de Michigan-Dearborn, quien me auxilió en el desarrollo de gran parte del contenido pedagógico en el sitio web que acompaña a este libro. Finalmente, quiero agradecer a los revisores de la séptima edición: sus comentarios a profundidad y críticas bien pensadas han sido invaluables.

Osman Balci,
Virginia Tech University
Max Fomitchev,
Penn State University
Jerry (Zeyu) Gao,
San Jose State University
Guillermo Garcia,
Universidad Alfonso X Madrid
Pablo Gervas,
Universidad Complutense de Madrid
SK Jain,
National Institute of Technology Hamirpur
Saeed Monemi,
Cal Poly Pomona
Ahmed Salem,
California State University
Vasudeva Varma,
IIIT Hyderabad

El contenido de la séptima edición de *Ingeniería del software: un enfoque práctico* fue conformado por profesionales de la industria, profesores universitarios y estudiantes, quienes usaron ediciones anteriores del libro y tomaron tiempo para comunicar sus sugerencias, críticas e ideas.

Mi agradecimiento a cada uno de ustedes. Además, mi reconocimiento personal a nuestros muchos clientes industriales en todo el mundo, quienes, ciertamente, me enseñaron tanto o más de lo que yo podría haberles enseñado en algún momento.

Conforme las ediciones de este libro evolucionaban, mis hijos, Mathew y Michael, crecieron de niños a hombres. Su madurez, carácter y éxito en el mundo real han sido una inspiración para mí. Nada me ha llenado más de orgullo. Y finalmente, a Bárbara, mi amor y agradecimiento por tolerar las muchísimas horas en la oficina y por alentar todavía otra edición de "el libro".

Roger S. Pressman

CAPÍTULO

1

EL SOFTWARE Y LA INGENIERÍA DE SOFTWARE

CONCEPTOS CLAVE

actividades estructurales 12
actividades sombrilla. 12
características del software . . . 3
dominios de aplicación 6
ingeniería de software 10
mitos del software 18
práctica 15
principios 16
proceso del software. 12
software heredado 8
***webapps* 9**

Tenía la apariencia clásica de un alto ejecutivo de una compañía importante de software —a la mitad de los 40, con las sienes comenzando a encanecer, esbelto y atlético, con ojos que penetraban al observador mientras hablaba—. Pero lo que dijo me dejó anonadado. "El software ha *muerto*".

Pestañeé con sorpresa y sonreí. "Bromeas, ¿verdad? El mundo es dirigido con software y tu empresa se ha beneficiado mucho de ello. ¡No ha muerto! Está vivo y en desarrollo."

Movió su cabeza de manera enfática. "No, está muerto... al menos como lo conocimos."

Me apoyé en el escritorio. "Continúa."

Habló al tiempo que golpeaba en la mesa con énfasis. "El concepto antiguo del software —lo compras, lo posees y tu trabajo consiste en administrarlo— está llegando a su fin. Hoy día, con Web 2.0 y la computación ubicua cada vez más fuerte, vamos a ver una generación de software por completo diferente. Se distribuirá por internet y se verá exactamente como si estuviera instalado en el equipo de cómputo de cada usuario... pero se encontrará en un servidor remoto."

Tuve que estar de acuerdo. "Entonces, tu vida será mucho más sencilla. Tus muchachos no tendrán que preocuparse por las cinco diferentes versiones de la misma App que utilizan decenas de miles de usuarios."

Sonrió. "Absolutamente. Sólo la versión más reciente estará en nuestros servidores. Cuando hagamos un cambio o corrección, actualizaremos funcionalidad y contenido a cada usuario. Todos lo tendrán en forma instantánea..."

Hice una mueca. "Pero si cometes un error, todos lo tendrán también instantáneamente".

Él se rió entre dientes. "Es verdad, por eso estamos redoblando nuestros esfuerzos para hacer una ingeniería de software aún mejor. El problema es que tenemos que hacerlo 'rápido' porque el mercado se ha acelerado en cada área de aplicación."

UNA MIRADA RÁPIDA

¿Qué es? El software de computadora es el producto que construyen los programadores profesionales y al que después le dan mantenimiento durante un largo tiempo. Incluye programas que se ejecutan en una computadora de cualquier tamaño y arquitectura, contenido que se presenta a medida de que se ejecutan los programas de cómputo e información descriptiva tanto en una copia dura como en formatos virtuales que engloban virtualmente a cualesquiera medios electrónicos. La ingeniería de software está formada por un proceso, un conjunto de métodos (prácticas) y un arreglo de herramientas que permite a los profesionales elaborar software de cómputo de alta calidad.

¿Quién lo hace? Los ingenieros de software elaboran y dan mantenimiento al software, y virtualmente cada persona lo emplea en el mundo industrializado, ya sea en forma directa o indirecta.

¿Por qué es importante? El software es importante porque afecta a casi todos los aspectos de nuestras vidas y ha invadido nuestro comercio, cultura y actividades cotidianas. La ingeniería de software es importante porque nos permite construir sistemas complejos en un tiempo razonable y con alta calidad.

¿Cuáles son los pasos? El software de computadora se construye del mismo modo que cualquier producto exitoso, con la aplicación de un proceso ágil y adaptable para obtener un resultado de mucha calidad, que satisfaga las necesidades de las personas que usarán el producto. En estos pasos se aplica el enfoque de la ingeniería de software.

¿Cuál es el producto final? Desde el punto de vista de un ingeniero de software, el producto final es el conjunto de programas, contenido (datos) y otros productos terminados que constituyen el software de computadora. Pero desde la perspectiva del usuario, el producto final es la información resultante que de algún modo hace mejor al mundo en el que vive.

¿Cómo me aseguro de que lo hice bien? Lea el resto de este libro, seleccione aquellas ideas que sean aplicables al software que usted hace y aplíquelas a su trabajo.

Me recargué en la espalda y coloqué mis manos en mi nuca. "Ya sabes lo que se dice... puedes tenerlo rápido o bien hecho o barato. Escoge dos de estas características..."

"Elijo rápido y bien hecho", dijo mientras comenzaba a levantarse.

También me incorporé. "Entonces realmente necesitas ingeniería de software."

"Ya lo sé", dijo mientras salía. "El problema es que tenemos que llegar a convencer a otra generación más de técnicos de que así es..."

¿Está muerto *realmente* el software? Si lo estuviera, usted no estaría leyendo este libro...

Cita:

"Las ideas y los descubrimientos tecnológicos son los motores que impulsan el crecimiento económico."

Wall Street Journal

El software de computadora sigue siendo la tecnología más importante en la escena mundial. Y también es un ejemplo magnífico de la ley de las consecuencias inesperadas. Hace 50 años, nadie hubiera podido predecir que el software se convertiría en una tecnología indispensable para los negocios, ciencias e ingeniería, ni que permitiría la creación de tecnologías nuevas (por ejemplo, ingeniería genética y nanotecnología), la ampliación de tecnologías ya existentes (telecomunicaciones) y el cambio radical de tecnologías antiguas (la industria de la impresión); tampoco que el software sería la fuerza que impulsaría la revolución de las computadoras personales, que productos de software empacados se comprarían en los supermercados, que el software evolucionaría poco a poco de un producto a un servicio cuando compañías de software "sobre pedido" proporcionaran funcionalidad justo a tiempo a través de un navegador web, que una compañía de software sería más grande y tendría más influencia que casi todas las empresas de la era industrial, que una vasta red llamada internet sería operada con software y evolucionaría y cambiaría todo, desde la investigación en bibliotecas y la compra de productos para el consumidor hasta el discurso político y los hábitos de encuentro de los adultos jóvenes (y no tan jóvenes).

Nadie pudo prever que habría software incrustado en sistemas de toda clase: de transporte, médicos, de telecomunicaciones, militares, industriales, de entretenimiento, en máquinas de oficina... la lista es casi infinita. Y si usted cree en la ley de las consecuencias inesperadas, hay muchos efectos que aún no podemos predecir.

Nadie podía anticipar que millones de programas de computadora tendrían que ser corregidos, adaptados y mejorados a medida que transcurriera el tiempo. Ni que la carga de ejecutar estas actividades de "mantenimiento" absorbería más personas y recursos que todo el trabajo aplicado a la creación de software nuevo.

Conforme ha aumentado la importancia del software, la comunidad de programadores ha tratado continuamente de desarrollar tecnologías que hagan más fácil, rápida y barata la elaboración de programas de cómputo de alta calidad. Algunas de estas tecnologías se dirigen a un dominio específico de aplicaciones (por ejemplo, diseño e implantación de un sitio web), otras se centran en un dominio tecnológico (sistemas orientados a objetos o programación orientada a aspectos), otros más tienen una base amplia (sistemas operativos, como Linux). Sin embargo, todavía falta por desarrollarse una tecnología de software que haga todo esto, y hay pocas probabilidades de que surja una en el futuro. A pesar de ello, las personas basan sus trabajos, confort, seguridad, diversiones, decisiones y sus propias vidas en software de computadora. Más vale que esté bien hecho.

Este libro presenta una estructura que puede ser utilizada por aquellos que hacen software de cómputo —personas que deben hacerlo bien—. La estructura incluye un proceso, un conjunto de métodos y unas herramientas que llamamos *ingeniería de software*.

1.1 La naturaleza del software

En la actualidad, el software tiene un papel dual. Es un producto y al mismo tiempo es el vehículo para entregar un producto. En su forma de producto, brinda el potencial de cómputo incorporado en el hardware de cómputo o, con más amplitud, en una red de computadoras a las

PUNTO CLAVE

El software es tanto un producto como un vehículo para entregar un producto.

que se accede por medio de un hardware local. Ya sea que resida en un teléfono móvil u opere en el interior de una computadora central, el software es un transformador de información —produce, administra, adquiere, modifica, despliega o transmite información que puede ser tan simple como un solo bit o tan compleja como una presentación con multimedios generada a partir de datos obtenidos de decenas de fuentes independientes—. Como vehículo utilizado para distribuir el producto, el software actúa como la base para el control de la computadora (sistemas operativos), para la comunicación de información (redes) y para la creación y control de otros programas (herramientas y ambientes de software).

El software distribuye el producto más importante de nuestro tiempo: *información*. Transforma los datos personales (por ejemplo, las transacciones financieras de un individuo) de modo que puedan ser más útiles en un contexto local, administra la información de negocios para mejorar la competitividad, provee una vía para las redes mundiales de información (la internet) y brinda los medios para obtener información en todas sus formas.

Cita:

"El software es un lugar donde se siembran sueños y se cosechan pesadillas, una ciénega abstracta y mística en la que terribles demonios luchan contra panaceas mágicas, un mundo de hombres lobo y balas de plata."

Brad J. Cox

En el último medio siglo, el papel del software de cómputo ha sufrido un cambio significativo. Las notables mejoras en el funcionamiento del hardware, los profundos cambios en las arquitecturas de computadora, el gran incremento en la memoria y capacidad de almacenamiento, y una amplia variedad de opciones de entradas y salidas exóticas han propiciado la existencia de sistemas basados en computadora más sofisticados y complejos. Cuando un sistema tiene éxito, la sofisticación y complejidad producen resultados deslumbrantes, pero también plantean problemas enormes para aquellos que deben construir sistemas complejos.

En la actualidad, la enorme industria del software se ha convertido en un factor dominante en las economías del mundo industrializado. Equipos de especialistas de software, cada uno centrado en una parte de la tecnología que se requiere para llegar a una aplicación compleja, han reemplazado al programador solitario de los primeros tiempos. A pesar de ello, las preguntas que se hacía aquel programador son las mismas que surgen cuando se construyen sistemas modernos basados en computadora:[1]

- ¿Por qué se requiere tanto tiempo para terminar el software?
- ¿Por qué son tan altos los costos de desarrollo?
- ¿Por qué no podemos detectar todos los errores antes de entregar el software a nuestros clientes?
- ¿Por qué dedicamos tanto tiempo y esfuerzo a mantener los programas existentes?
- ¿Por qué seguimos con dificultades para medir el avance mientras se desarrolla y mantiene el software?

Éstas y muchas otras preguntas, denotan la preocupación sobre el software y la manera en que se desarrolla, preocupación que ha llevado a la adopción de la práctica de la ingeniería del software.

1.1.1 Definición de software

En la actualidad, la mayoría de profesionales y muchos usuarios tienen la fuerte sensación de que entienden el software. Pero, ¿es así?

La descripción que daría un libro de texto sobre software sería más o menos así:

? ¿Cómo se define *software*?

El software es: 1) instrucciones (programas de cómputo) que cuando se ejecutan proporcionan las características, función y desempeño buscados; 2) estructuras de datos que permiten que los progra-

1 En un excelente libro de ensayos sobre el negocio del software, Tom DeMarco [DeM95] defiende el punto de vista contrario. Dice: "En lugar de preguntar por qué el software cuesta tanto, necesitamos comenzar a preguntar: ¿Qué hemos hecho para hacer posible que el software actual cueste tan poco? La respuesta a esa pregunta nos ayudará a continuar el extraordinario nivel de logro que siempre ha distinguido a la industria del software."

mas manipulen en forma adecuada la información, y 3) información descriptiva tanto en papel como en formas virtuales que describen la operación y uso de los programas.

No hay duda de que podrían darse definiciones más completas.

Pero es probable que una definición más formal no mejore de manera apreciable nuestra comprensión. Para asimilar lo anterior, es importante examinar las características del software que lo hacen diferente de otros objetos que construyen los seres humanos. El software es elemento de un sistema lógico y no de uno físico. Por tanto, tiene características que difieren considerablemente de las del hardware:

1. *El software se desarrolla o modifica con intelecto; no se manufactura en el sentido clásico.*

PUNTO CLAVE
El software se modifica con intelecto, no se manufactura.

Aunque hay algunas similitudes entre el desarrollo de software y la fabricación de hardware, las dos actividades son diferentes en lo fundamental. En ambas, la alta calidad se logra a través de un buen diseño, pero la fase de manufactura del hardware introduce problemas de calidad que no existen (o que se corrigen con facilidad) en el software. Ambas actividades dependen de personas, pero la relación entre los individuos dedicados y el trabajo logrado es diferente por completo (véase el capítulo 24). Las dos actividades requieren la construcción de un "producto", pero los enfoques son distintos. Los costos del software se concentran en la ingeniería. Esto significa que los proyectos de software no pueden administrarse como si fueran proyectos de manufactura.

2. *El software no se "desgasta".*

PUNTO CLAVE
El software no se desgasta, pero sí se deteriora.

La figura 1.1 ilustra la tasa de falla del hardware como función del tiempo. La relación, que es frecuente llamar "curva de tina", indica que el hardware presenta una tasa de fallas relativamente elevada en una etapa temprana de su vida (fallas que con frecuencia son atribuibles a defectos de diseño o manufactura); los defectos se corrigen y la tasa de fallas se abate a un nivel estable (muy bajo, por fortuna) durante cierto tiempo. No obstante, conforme pasa el tiempo, la tasa de fallas aumenta de nuevo a medida que los componentes del hardware resienten los efectos acumulativos de suciedad, vibración, abuso, temperaturas extremas y muchos otros inconvenientes ambientales. En pocas palabras, el hardware comienza a *desgastarse*.

El software no es susceptible a los problemas ambientales que hacen que el hardware se desgaste. Por tanto, en teoría, la curva de la tasa de fallas adopta la forma de la "curva idealizada" que se aprecia en la figura 1.2. Los defectos ocultos ocasionarán ta-

FIGURA 1.1
Curva de fallas del hardware

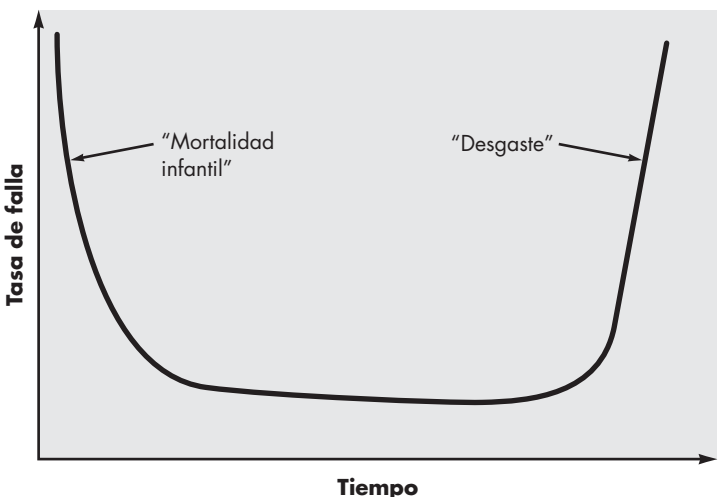
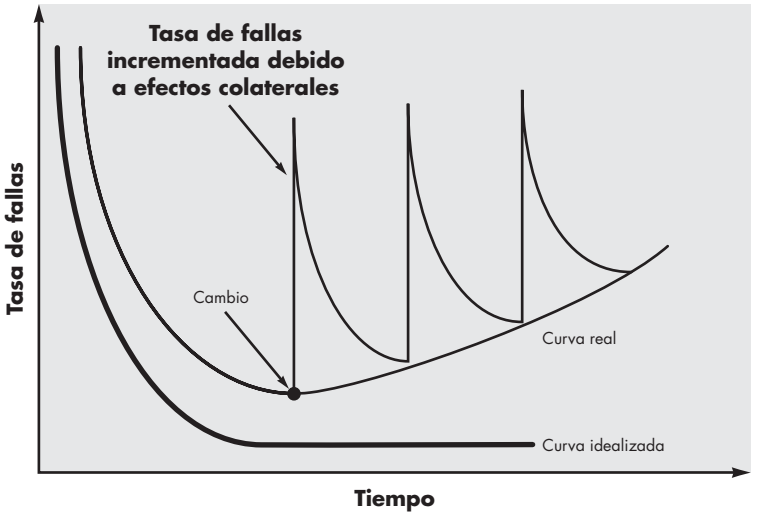


FIGURA 1.2
Curvas de falla del software



CONSEJO

Si quiere reducir el deterioro del software, tendrá que mejorar su diseño (capítulos 8 a 13).

PUNTO CLAVE

Los métodos de la ingeniería de software llevan a reducir la magnitud de los picos y de la pendiente de la curva real en la figura 1.2.

sas elevadas de fallas al comienzo de la vida de un programa. Sin embargo, éstas se corrigen y la curva se aplana, como se indica. La curva idealizada es una gran simplificación de los modelos reales de las fallas del software. Aun así, la implicación está clara: el software no se desgasta, ¡pero sí se *deteriora!*

Esta contradicción aparente se entiende mejor si se considera la curva real en la figura 1.2. Durante su vida,[2] el software sufrirá cambios. Es probable que cuando éstos se realicen, se introduzcan errores que ocasionen que la curva de tasa de fallas tenga aumentos súbitos, como se ilustra en la "curva real" (véase la figura 1.2). Antes de que la curva vuelva a su tasa de fallas original de estado estable, surge la solicitud de otro cambio que hace que la curva se dispare otra vez. Poco a poco, el nivel mínimo de la tasa de fallas comienza a aumentar: el software se está deteriorando como consecuencia del cambio.

Otro aspecto del desgaste ilustra la diferencia entre el hardware y el software. Cuando un componente del hardware se desgasta es sustituido por una refacción. En cambio, no hay refacciones para el software. Cada falla de éste indica un error en el diseño o en el proceso que tradujo el diseño a código ejecutable por la máquina. Entonces, las tareas de mantenimiento del software, que incluyen la satisfacción de peticiones de cambios, involucran una complejidad considerablemente mayor que el mantenimiento del hardware.

Cita:

"Las ideas son los ladrillos con los que se construyen las ideas."

Jason Zebehazy

3. *Aunque la industria se mueve hacia la construcción basada en componentes, la mayor parte del software se construye para un uso individualizado*.

A medida que evoluciona una disciplina de ingeniería, se crea un conjunto de componentes estandarizados para el diseño. Los tornillos estándar y los circuitos integrados preconstruidos son sólo dos de los miles de componentes estándar que utilizan los ingenieros mecánicos y eléctricos conforme diseñan nuevos sistemas. Los componentes reutilizables han sido creados para que el ingeniero pueda concentrarse en los elementos verdaderamente innovadores de un diseño; es decir, en las partes de éste que representan algo nuevo. En el mundo del hardware, volver a usar componentes es una parte

2 En realidad, los distintos participantes solicitan cambios desde el momento en que comienza el desarrollo y mucho antes de que se disponga de la primera versión.

natural del proceso de ingeniería. En el del software, es algo que apenas ha empezado a hacerse a gran escala.

Un componente de software debe diseñarse e implementarse de modo que pueda volverse a usar en muchos programas diferentes. Los modernos componentes reutilizables incorporan tanto los datos como el procesamiento que se les aplica, lo que permite que el ingeniero de software cree nuevas aplicaciones a partir de partes susceptibles de volverse a usar.[3] Por ejemplo, las actuales interfaces interactivas de usuario se construyen con componentes reutilizables que permiten la creación de ventanas gráficas, menús desplegables y una amplia variedad de mecanismos de interacción. Las estructuras de datos y el detalle de procesamiento que se requieren para construir la interfaz están contenidos en una librería de componentes reusables para tal fin.

1.1.2 Dominios de aplicación del software

Actualmente, hay siete grandes categorías de software de computadora que plantean retos continuos a los ingenieros de software:

Software de sistemas: conjunto de programas escritos para dar servicio a otros programas. Determinado software de sistemas (por ejemplo, compiladores, editores y herramientas para administrar archivos) procesa estructuras de información complejas pero deterministas.[4] Otras aplicaciones de sistemas (por ejemplo, componentes de sistemas operativos, manejadores, software de redes, procesadores de telecomunicaciones) procesan sobre todo datos indeterminados. En cualquier caso, el área de software de sistemas se caracteriza por: gran interacción con el hardware de la computadora, uso intensivo por parte de usuarios múltiples, operación concurrente que requiere la secuenciación, recursos compartidos y administración de un proceso sofisticado, estructuras complejas de datos e interfaces externas múltiples.

WebRef

En la dirección **shareware.cnet.com** se encuentra una de las librerías más completas de software compartido y libre.

Software de aplicación: programas aislados que resuelven una necesidad específica de negocios. Las aplicaciones en esta área procesan datos comerciales o técnicos en una forma que facilita las operaciones de negocios o la toma de decisiones administrativas o técnicas. Además de las aplicaciones convencionales de procesamiento de datos, el software de aplicación se usa para controlar funciones de negocios en tiempo real (por ejemplo, procesamiento de transacciones en punto de venta, control de procesos de manufactura en tiempo real).

Software de ingeniería y ciencias: se ha caracterizado por algoritmos "devoradores de números". Las aplicaciones van de la astronomía a la vulcanología, del análisis de tensiones en automóviles a la dinámica orbital del transbordador espacial, y de la biología molecular a la manufactura automatizada. Sin embargo, las aplicaciones modernas dentro del área de la ingeniería y las ciencias están abandonando los algoritmos numéricos convencionales. El diseño asistido por computadora, la simulación de sistemas y otras aplicaciones interactivas, han comenzado a hacerse en tiempo real e incluso han tomado características del software de sistemas.

Software incrustado: reside dentro de un producto o sistema y se usa para implementar y controlar características y funciones para el usuario final y para el sistema en sí. El software incrustado ejecuta funciones limitadas y particulares (por ejemplo, control del tablero de un horno de microondas) o provee una capacidad significativa de funcionamiento y control

3 El desarrollo basado en componentes se estudia en el capítulo 10.

4 El software es *determinista* si es posible predecir el orden y momento de las entradas, el procesamiento y las salidas. El software es *no determinista* si no pueden predecirse el orden y momento en que ocurren éstos.

(funciones digitales en un automóvil, como el control del combustible, del tablero de control y de los sistemas de frenado).

Software de línea de productos: es diseñado para proporcionar una capacidad específica para uso de muchos consumidores diferentes. El software de línea de productos se centra en algún mercado limitado y particular (por ejemplo, control del inventario de productos) o se dirige a mercados masivos de consumidores (procesamiento de textos, hojas de cálculo, gráficas por computadora, multimedios, entretenimiento, administración de base de datos y aplicaciones para finanzas personales o de negocios).

Aplicaciones web: llamadas "*webapps*", esta categoría de software centrado en redes agrupa una amplia gama de aplicaciones. En su forma más sencilla, las *webapps* son poco más que un conjunto de archivos de hipertexto vinculados que presentan información con uso de texto y gráficas limitadas. Sin embargo, desde que surgió Web 2.0, las *webapps* están evolucionando hacia ambientes de cómputo sofisticados que no sólo proveen características aisladas, funciones de cómputo y contenido para el usuario final, sino que también están integradas con bases de datos corporativas y aplicaciones de negocios.

Cita:

"No hay computadora que tenga sentido común."

Marvin Minsky

Software de inteligencia artificial: hace uso de algoritmos no numéricos para resolver problemas complejos que no son fáciles de tratar computacionalmente o con el análisis directo. Las aplicaciones en esta área incluyen robótica, sistemas expertos, reconocimiento de patrones (imagen y voz), redes neurales artificiales, demostración de teoremas y juegos.

Son millones de ingenieros de software en todo el mundo los que trabajan duro en proyectos de software en una o más de estas categorías. En ciertos casos se elaboran sistemas nuevos, pero en muchos otros se corrigen, adaptan y mejoran aplicaciones ya existentes. No es raro que una joven ingeniera de software trabaje en un programa de mayor edad que la de ella… Las generaciones pasadas de los trabajadores del software dejaron un legado en cada una de las categorías mencionadas. Por fortuna, la herencia que dejará la actual generación aligerará la carga de los futuros ingenieros de software. Aun así, nuevos desafíos (capítulo 31) han aparecido en el horizonte.

Computación en un mundo abierto: el rápido crecimiento de las redes inalámbricas quizá lleve pronto a la computación verdaderamente ubicua y distribuida. El reto para los ingenieros de software será desarrollar software de sistemas y aplicación que permita a dispositivos móviles, computadoras personales y sistemas empresariales comunicarse a través de redes enormes.

Construcción de redes: la red mundial (World Wide Web) se está convirtiendo con rapidez tanto en un motor de computación como en un proveedor de contenido. El desafío para los ingenieros de software es hacer arquitecturas sencillas (por ejemplo, planeación financiera personal y aplicaciones sofisticadas que proporcionen un beneficio a mercados objetivo de usuarios finales en todo el mundo).

Fuente abierta: tendencia creciente que da como resultado la distribución de código fuente para aplicaciones de sistemas (por ejemplo, sistemas operativos, bases de datos y ambientes de desarrollo) de modo que mucha gente pueda contribuir a su desarrollo. El desafío para los ingenieros de software es elaborar código fuente que sea autodescriptivo, y también, lo que es más importante, desarrollar técnicas que permitirán tanto a los consumidores como a los desarrolladores saber cuáles son los cambios hechos y cómo se manifiestan dentro del software.

Cita:

"No siempre puedes predecir, pero siempre puedes prepararte."

Anónimo

Es indudable que cada uno de estos nuevos retos obedecerá a la ley de las consecuencias imprevistas y tendrá efectos (para hombres de negocios, ingenieros de software y usuarios finales) que hoy no pueden predecirse. Sin embargo, los ingenieros de software pueden prepararse de-

sarrollando un proceso que sea ágil y suficientemente adaptable para que acepte los cambios profundos en la tecnología y las reglas de los negocios que seguramente surgirán en la década siguiente.

1.1.3 Software heredado

Cientos de miles de programas de cómputo caen en uno de los siete dominios amplios de aplicación que se estudiaron en la subsección anterior. Algunos de ellos son software muy nuevo, disponible para ciertos individuos, industria y gobierno. Pero otros programas son más viejos, en ciertos casos *muy* viejos.

Estos programas antiguos —que es frecuente denominar *software heredado*— han sido centro de atención y preocupación continuas desde la década de 1960. Dayani-Fard y sus colegas [Day99] describen el software heredado de la manera siguiente:

> Los sistemas de software heredado [...] fueron desarrollados hace varias décadas y han sido modificados de manera continua para que satisfagan los cambios en los requerimientos de los negocios y plataformas de computación. La proliferación de tales sistemas es causa de dolores de cabeza para las organizaciones grandes, a las que resulta costoso mantenerlos y riesgoso hacerlos evolucionar.

Liu y sus colegas [Liu98] amplían esta descripción al hacer notar que "muchos sistemas heredados continúan siendo un apoyo para las funciones básicas del negocio y son 'indispensables' para éste". Además, el software heredado se caracteriza por su longevidad e importancia crítica para el negocio.

¿Qué hago si encuentro un sistema heredado de mala calidad?

Desafortunadamente, en ocasiones hay otra característica presente en el software heredado: *mala calidad*.[5] Hay veces en las que los sistemas heredados tienen diseños que no son susceptibles de extenderse, código confuso, documentación mala o inexistente, casos y resultados de pruebas que nunca se archivaron, una historia de los cambios mal administrada... la lista es muy larga. A pesar de esto, dichos sistemas dan apoyo a las "funciones básicas del negocio y son indispensables para éste". ¿Qué hacer?

La única respuesta razonable es: *hacer nada*, al menos hasta que el sistema heredado tenga un cambio significativo. Si el software heredado satisface las necesidades de sus usuarios y corre de manera confiable, entonces no falla ni necesita repararse. Sin embargo, conforme pase el tiempo será frecuente que los sistemas de software evolucionen por una o varias de las siguientes razones:

¿Qué tipos de cambios se hacen a los sistemas heredados?

- El software debe adaptarse para que cumpla las necesidades de los nuevos ambientes del cómputo y de la tecnología.
- El software debe ser mejorado para implementar nuevos requerimientos del negocio.
- El software debe ampliarse para que sea operable con otros sistemas o bases de datos modernos.
- La arquitectura del software debe rediseñarse para hacerla viable dentro de un ambiente de redes.

Todo ingeniero de software debe reconocer que el cambio es natural. No trate de evitarlo.

Cuando ocurren estos modos de evolución, debe hacerse la reingeniería del sistema heredado (capítulo 29) para que sea viable en el futuro. La meta de la ingeniería de software moderna es "desarrollar metodologías que se basen en el concepto de evolución; es decir, el concepto de que los sistemas de software cambian continuamente, que los nuevos sistemas de software se

5 En este caso, la calidad se juzga con base en el pensamiento moderno de la ingeniería de software, criterio algo injusto, ya que algunos conceptos y principios de la ingeniería de software moderna tal vez no hayan sido bien entendidos en la época en que se desarrolló el software heredado.

desarrollan a partir de los antiguos y [...] que todo debe operar entre sí y cooperar con cada uno de los demás" [Day99].

1.2 LA NATURALEZA ÚNICA DE LAS *WEBAPPS*

Cita:

"Cuando veamos cualquier tipo de estabilización, la web se habrá convertido en algo completamente diferente."

Louis Monier

En los primeros días de la Red Mundial (entre 1990 y 1995), los *sitios web* consistían en poco más que un conjunto de archivos de hipertexto vinculados que presentaban la información con el empleo de texto y gráficas limitadas. Al pasar el tiempo, el aumento de HTML por medio de herramientas de desarrollo (XML, Java) permitió a los ingenieros de la web brindar capacidad de cómputo junto con contenido de información. Habían nacido los *sistemas y aplicaciones basados en la web*[6] (denominó a éstas en forma colectiva como *webapps*). En la actualidad, las *webapps* se han convertido en herramientas sofisticadas de cómputo que no sólo proporcionan funciones aisladas al usuario final, sino que también se han integrado con bases de datos corporativas y aplicaciones de negocios.

Como se dijo en la sección 1.1.2, las *webapps* son una de varias categorías distintas de software. No obstante, podría argumentarse que las *webapps* son diferentes. Powell [Pow98] sugiere que los sistemas y aplicaciones basados en web "involucran una mezcla entre las publicaciones impresas y el desarrollo de software, entre la mercadotecnia y la computación, entre las comunicaciones internas y las relaciones exteriores, y entre el arte y la tecnología". La gran mayoría de *webapps* presenta los siguientes atributos:

? ¿Qué característica diferencia las *webapps* de otro software?

Uso intensivo de redes. Una *webapp* reside en una red y debe atender las necesidades de una comunidad diversa de clientes. La red permite acceso y comunicación mundiales (por ejemplo, internet) o tiene acceso y comunicación limitados (por ejemplo, una intranet corporativa).

Concurrencia. A la *webapp* puede acceder un gran número de usuarios a la vez. En muchos casos, los patrones de uso entre los usuarios finales varían mucho.

Carga impredecible. El número de usuarios de la *webapp* cambia en varios órdenes de magnitud de un día a otro. El lunes tal vez la utilicen cien personas, el jueves quizá 10 000 usen el sistema.

Rendimiento. Si un usuario de la *webapp* debe esperar demasiado (para entrar, para el procesamiento por parte del servidor, para el formado y despliegue del lado del cliente), él o ella quizá decidan irse a otra parte.

Disponibilidad. Aunque no es razonable esperar una disponibilidad de 100%, es frecuente que los usuarios de *webapps* populares demanden acceso las 24 horas de los 365 días del año. Los usuarios en Australia o Asia quizá demanden acceso en horas en las que las aplicaciones internas de software tradicionales en Norteamérica no estén en línea por razones de mantenimiento.

Orientadas a los datos. La función principal de muchas *webapp* es el uso de hipermedios para presentar al usuario final contenido en forma de texto, gráficas, audio y video. Además, las *webapps* se utilizan en forma común para acceder a información que existe en bases de datos que no son parte integral del ambiente basado en web (por ejemplo, comercio electrónico o aplicaciones financieras).

---

6 En el contexto de este libro, el término *aplicación web* (*webapp*) agrupa todo, desde una simple página web que ayude al consumidor a calcular el pago del arrendamiento de un automóvil hasta un sitio web integral que proporcione servicios completos de viaje para gente de negocios y vacacionistas. En esta categoría se incluyen sitios web completos, funcionalidad especializada dentro de sitios web y aplicaciones de procesamiento de información que residen en internet o en una intranet o extranet.

Contenido sensible. La calidad y naturaleza estética del contenido constituye un rasgo importante de la calidad de una *webapp*.

Evolución continua. A diferencia del software de aplicación convencional que evoluciona a lo largo de una serie de etapas planeadas y separadas cronológicamente, las aplicaciones web evolucionan en forma continua. No es raro que ciertas *webapp* (específicamente su contenido) se actualicen minuto a minuto o que su contenido se calcule en cada solicitud.

Inmediatez. Aunque la *inmediatez* —necesidad apremiante de que el software llegue con rapidez al mercado— es una característica en muchos dominios de aplicación, es frecuente que las *webapps* tengan plazos de algunos días o semanas para llegar al mercado.[7]

Seguridad. Debido a que las *webapps* se encuentran disponibles con el acceso a una red, es difícil o imposible limitar la población de usuarios finales que pueden acceder a la aplicación. Con el fin de proteger el contenido sensible y brindar modos seguros de transmisión de los datos, deben implementarse medidas estrictas de seguridad a través de la infraestructura de apoyo de una *webapp* y dentro de la aplicación misma.

Estética. Parte innegable del atractivo de una *webapp* es su apariencia y percepción. Cuando se ha diseñado una aplicación para comercializar o vender productos o ideas, la estética tiene tanto que ver con el éxito como el diseño técnico.

Podría argumentarse que otras categorías de aplicaciones estudiadas en la sección 1.1.2 muestran algunos de los atributos mencionados. Sin embargo, las *webapps* casi siempre poseen todos ellos.

1.3 INGENIERÍA DE SOFTWARE

Con objeto de elaborar software listo para enfrentar los retos del siglo XXI, el lector debe aceptar algunas realidades sencillas:

PUNTO CLAVE
Entender el problema antes de dar una solución.

- El software se ha incrustado profundamente en casi todos los aspectos de nuestras vidas y, como consecuencia, el número de personas que tienen interés en las características y funciones que brinda una aplicación específica[8] ha crecido en forma notable. Cuando ha de construirse una aplicación nueva o sistema incrustado, deben escucharse muchas opiniones. Y en ocasiones parece que cada una de ellas tiene una idea un poco distinta de cuáles características y funciones debiera tener el software. *Se concluye que debe hacerse un esfuerzo concertado para entender el problema antes de desarrollar una aplicación de software.*

PUNTO CLAVE
El diseño es una actividad crucial de la ingeniería de software.

- Los requerimientos de la tecnología de la información que demandan los individuos, negocios y gobiernos se hacen más complejos con cada año que pasa. En la actualidad, grandes equipos de personas crean programas de cómputo que antes eran elaborados por un solo individuo. El software sofisticado, que alguna vez se implementó en un ambiente de cómputo predecible y autocontenido, hoy en día se halla incrustado en el interior de todo, desde la electrónica de consumo hasta dispositivos médicos o sistemas de armamento. La complejidad de estos nuevos sistemas y productos basados en computadora demanda atención cuidadosa a las interacciones de todos los elementos del sistema. *Se concluye que el diseño se ha vuelto una actividad crucial.*

7 Con las herramientas modernas es posible producir páginas web sofisticadas en unas cuantas horas.

8 En una parte posterior de este libro, llamaré a estas personas "participantes".

PUNTO CLAVE

Tanto la calidad como la facilidad de recibir mantenimiento son resultado de un buen diseño.

- Los individuos, negocios y gobiernos dependen cada vez más del software para tomar decisiones estratégicas y tácticas, así como para sus operaciones y control cotidianos. Si el software falla, las personas y empresas grandes pueden experimentar desde un inconveniente menor hasta fallas catastróficas. *Se concluye que el software debe tener alta calidad.*
- A medida que aumenta el valor percibido de una aplicación específica se incrementa la probabilidad de que su base de usuarios y longevidad también crezcan. Conforme se extienda su base de usuarios y el tiempo de uso, las demandas para adaptarla y mejorarla también crecerán. *Se concluye que el software debe tener facilidad para recibir mantenimiento.*

Estas realidades simples llevan a una conclusión: *debe hacerse ingeniería con el software en todas sus formas y a través de todos sus dominios de aplicación.* Y esto conduce al tema de este libro: *la ingeniería de software*.

Cita:

"Más que una disciplina o cuerpo de conocimientos, ingeniería es un verbo, una palabra de acción, una forma de abordar un problema."

Scott Whitmir

Aunque cientos de autores han desarrollado definiciones personales de la ingeniería de software, la propuesta por Fritz Bauer [Nau69] en la conferencia fundamental sobre el tema todavía sirve como base para el análisis:

> [La ingeniería de software es] el establecimiento y uso de principios fundamentales de la ingeniería con objeto de desarrollar en forma económica software que sea confiable y que trabaje con eficiencia en máquinas reales.

El lector se sentirá tentado de ampliar esta definición.[9] Dice poco sobre los aspectos técnicos de la calidad del software; no habla directamente de la necesidad de satisfacer a los consumidores ni de entregar el producto a tiempo; omite mencionar la importancia de la medición y la metrología; no establece la importancia de un proceso eficaz. No obstante, la definición de Bauer proporciona una base. ¿Cuáles son los "principios fundamentales de la ingeniería" que pueden aplicarse al desarrollo del software de computadora? ¿Cómo se desarrolla software "en forma económica" y que sea "confiable"? ¿Qué se requiere para crear programas de cómputo que trabajen con "eficiencia", no en una sino en muchas "máquinas reales" diferentes? Éstas son las preguntas que siguen siendo un reto para los ingenieros de software.

? ¿Cómo se define la ingeniería de software?

El IEEE [IEEE93a] ha desarrollado una definición más completa, como sigue:

> La ingeniería de software es: 1) La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software; es decir, la aplicación de la ingeniería al software. 2) El estudio de enfoques según el punto 1.

Aun así, el enfoque "sistemático, disciplinado y cuantificable" aplicado por un equipo de software podría ser algo burdo para otro. Se necesita disciplina, pero también adaptabilidad y agilidad.

PUNTO CLAVE

La ingeniería de software incluye un proceso, métodos y herramientas para administrar y hacer ingeniería con el software.

La ingeniería de software es una tecnología con varias capas. Como se aprecia en la figura 1.3, cualquier enfoque de ingeniería (incluso la de software) debe basarse en un compromiso organizacional con la calidad. La administración total de la calidad, Six Sigma y otras filosofías similares[10] alimentan la cultura de mejora continua, y es esta cultura la que lleva en última instancia al desarrollo de enfoques cada vez más eficaces de la ingeniería de software. El fundamento en el que se apoya la ingeniería de software es el compromiso con la calidad.

El fundamento para la ingeniería de software es la capa *proceso*. El proceso de ingeniería de software es el aglutinante que une las capas de la tecnología y permite el desarrollo racional y

9 Consulte muchas otras definiciones en www.answers.com/topic/software-engineering#wp-_note-13.

10 En el capítulo 14 y toda la parte 3 del libro se estudia la administración de la calidad y los enfoques relacionados con ésta.

FIGURA 1.3
Capas de la ingeniería de software

WebRef
CrossTalk es un periódico que da información práctica sobre procesos, métodos y herramientas. Se encuentra en **www.stsc.hill.af.mil**

oportuno del software de cómputo. El proceso define una estructura que debe establecerse para la obtención eficaz de tecnología de ingeniería de software. El proceso de software forma la base para el control de la administración de proyectos de software, y establece el contexto en el que se aplican métodos técnicos, se generan productos del trabajo (modelos, documentos, datos, reportes, formatos, etc.), se establecen puntos de referencia, se asegura la calidad y se administra el cambio de manera apropiada.

Los *métodos* de la ingeniería de software proporcionan la experiencia técnica para elaborar software. Incluyen un conjunto amplio de tareas, como comunicación, análisis de los requerimientos, modelación del diseño, construcción del programa, pruebas y apoyo. Los métodos de la ingeniería de software se basan en un conjunto de principios fundamentales que gobiernan cada área de la tecnología e incluyen actividades de modelación y otras técnicas descriptivas.

Las *herramientas* de la ingeniería de software proporcionan un apoyo automatizado o semiautomatizado para el proceso y los métodos. Cuando se integran las herramientas de modo que la información creada por una pueda ser utilizada por otra, queda establecido un sistema llamado *ingeniería de software asistido por computadora* que apoya el desarrollo de software.

1.4 EL PROCESO DEL SOFTWARE

? ¿Cuáles son los elementos de un proceso de software?

Cita:

"Un proceso define quién hace *qué*, *cuándo* y *cómo*, para alcanzar cierto objetivo."

Ivar Jacobson, Grady Booch y James Rumbaugh

Un *proceso* es un conjunto de actividades, acciones y tareas que se ejecutan cuando va a crearse algún producto del trabajo. Una *actividad* busca lograr un objetivo amplio (por ejemplo, comunicación con los participantes) y se desarrolla sin importar el dominio de la aplicación, tamaño del proyecto, complejidad del esfuerzo o grado de rigor con el que se usará la ingeniería de software. Una *acción* (diseño de la arquitectura) es un conjunto de tareas que producen un producto importante del trabajo (por ejemplo, un modelo del diseño de la arquitectura). Una *tarea* se centra en un objetivo pequeño pero bien definido (por ejemplo, realizar una prueba unitaria) que produce un resultado tangible.

En el contexto de la ingeniería de software, un proceso *no* es una prescripción rígida de cómo elaborar software de cómputo. Por el contrario, es un enfoque adaptable que permite que las personas que hacen el trabajo (el equipo de software) busquen y elijan el conjunto apropiado de acciones y tareas para el trabajo. Se busca siempre entregar el software en forma oportuna y con calidad suficiente para satisfacer a quienes patrocinaron su creación y a aquellos que lo usarán.

La *estructura del proceso* establece el fundamento para el proceso completo de la ingeniería de software por medio de la identificación de un número pequeño de *actividades estructurales* que sean aplicables a todos los proyectos de software, sin importar su tamaño o complejidad. Además, la estructura del proceso incluye un conjunto de *actividades sombrilla* que son aplicables a través de todo el proceso del software. Una estructura de proceso general para la ingeniería de software consta de cinco actividades:

¿Cuáles son las cinco actividades estructurales del proceso?

Comunicación. Antes de que comience cualquier trabajo técnico, tiene importancia crítica comunicarse y colaborar con el cliente (y con otros participantes).[11] Se busca entender los objetivos de los participantes respecto del proyecto, y reunir los requerimientos que ayuden a definir las características y funciones del software.

Planeación. Cualquier viaje complicado se simplifica si existe un mapa. Un proyecto de software es un viaje difícil, y la actividad de planeación crea un "mapa" que guía al equipo mientras viaja. El mapa —llamado *plan del proyecto de software*— define el trabajo de ingeniería de software al describir las tareas técnicas por realizar, los riesgos probables, los recursos que se requieren, los productos del trabajo que se obtendrán y una programación de las actividades.

Cita:

"Einstein afirmaba que debía haber una explicación sencilla de la naturaleza porque Dios no es caprichoso o arbitrario. Al ingeniero de software no lo conforta una fe parecida. Gran parte de la complejidad que debe doblegar es de origen arbitrario."

Fred Brooks

Modelado. Ya sea usted diseñador de paisaje, constructor de puentes, ingeniero aeronáutico, carpintero o arquitecto, a diario trabaja con modelos. Crea un "bosquejo" del objeto por hacer a fin de entender el panorama general —cómo se verá arquitectónicamente, cómo ajustan entre sí las partes constituyentes y muchas características más—. Si se requiere, refina el bosquejo con más y más detalles en un esfuerzo por comprender mejor el problema y cómo resolverlo. Un ingeniero de software hace lo mismo al crear modelos a fin de entender mejor los requerimientos del software y el diseño que los satisfará.

Construcción. Esta actividad combina la generación de código (ya sea manual o automatizada) y las pruebas que se requieren para descubrir errores en éste.

Despliegue. El software (como entidad completa o como un incremento parcialmente terminado) se entrega al consumidor que lo evalúa y que le da retroalimentación, misma que se basa en dicha evaluación.

Estas cinco actividades estructurales genéricas se usan durante el desarrollo de programas pequeños y sencillos, en la creación de aplicaciones web grandes y en la ingeniería de sistemas enormes y complejos basados en computadoras. Los detalles del proceso de software serán distintos en cada caso, pero las actividades estructurales son las mismas.

Para muchos proyectos de software, las actividades estructurales se aplican en forma iterativa a medida que avanza el proyecto. Es decir, la **comunicación,** la **planeación,** el **modelado,** la **construcción** y el **despliegue** se ejecutan a través de cierto número de repeticiones del proyecto. Cada iteración produce un *incremento del software* que da a los participantes un subconjunto de características y funcionalidad generales del software. Conforme se produce cada incremento, el software se hace más y más completo.

Las actividades estructurales del proceso de ingeniería de software son complementadas por cierto número de *actividades sombrilla*. En general, las actividades sombrilla se aplican a lo largo de un proyecto de software y ayudan al equipo que lo lleva a cabo a administrar y controlar el avance, la calidad, el cambio y el riesgo. Es común que las actividades sombrilla sean las siguientes:

PUNTO CLAVE

Las actividades sombrilla ocurren a lo largo del proceso de software y se centran sobre todo en la administración, el seguimiento y el control del proyecto.

Seguimiento y control del proyecto de software: permite que el equipo de software evalúe el progreso comparándolo con el plan del proyecto y tome cualquier acción necesaria para apegarse a la programación de actividades.

Administración del riesgo: evalúa los riesgos que puedan afectar el resultado del proyecto o la calidad del producto.

---

11 Un *participante* es cualquier persona que tenga algo que ver en el resultado exitoso del proyecto —gerentes del negocio, usuarios finales, ingenieros de software, personal de apoyo, etc.—. Rob Thomset dice en broma que "un participante es una persona que blande una estaca grande y aguda [...] Si no vez más lejos que los participantes, ya sabes dónde terminará la estaca". (N. del T.: Esta nota es un juego de palabras: *stake* significa estaca y también *parte*, y *stakeholder* es el que blande una estaca, pero también un *participante*.)

Aseguramiento de la calidad del software: define y ejecuta las actividades requeridas para garantizar la calidad del software.

Revisiones técnicas: evalúa los productos del trabajo de la ingeniería de software a fin de descubrir y eliminar errores antes de que se propaguen a la siguiente actividad.

Medición: define y reúne mediciones del proceso, proyecto y producto para ayudar al equipo a entregar el software que satisfaga las necesidades de los participantes; puede usarse junto con todas las demás actividades estructurales y sombrilla.

Administración de la configuración del software: administra los efectos del cambio a lo largo del proceso del software.

Administración de la reutilización: define criterios para volver a usar el producto del trabajo (incluso los componentes del software) y establece mecanismos para obtener componentes reutilizables.

Preparación y producción del producto del trabajo: agrupa las actividades requeridas para crear productos del trabajo, tales como modelos, documentos, registros, formatos y listas.

Cada una de estas actividades sombrilla se analiza en detalle más adelante.

PUNTO CLAVE
La adaptación del proceso de software es esencial para el éxito del proyecto.

Ya se dijo en esta sección que el proceso de ingeniería de software no es una prescripción rígida que deba seguir en forma dogmática el equipo que lo crea. Al contrario, debe ser ágil y adaptable (al problema, al proyecto, al equipo y a la cultura organizacional). Por tanto, un proceso adoptado para un proyecto puede ser significativamente distinto de otro adoptado para otro proyecto. Entre las diferencias se encuentran las siguientes:

? ¿Qué diferencias existen entre los modelos del proceso?

- Flujo general de las actividades, acciones y tareas, así como de las interdependencias entre ellas
- Grado en el que las acciones y tareas están definidas dentro de cada actividad estructural
- Grado en el que se identifican y requieren los productos del trabajo
- Forma en la que se aplican las actividades de aseguramiento de la calidad
- Manera en la que se realizan las actividades de seguimiento y control del proyecto
- Grado general de detalle y rigor con el que se describe el proceso
- Grado con el que el cliente y otros participantes se involucran con el proyecto
- Nivel de autonomía que se da al equipo de software
- Grado con el que son prescritos la organización y los roles del equipo

Cita:

"Siento que una receta es sólo un tema que una cocinera inteligente ejecuta con una variación en cada ocasión."

Madame Benoit

En la parte 1 de este libro, se examinará el proceso de software con mucho detalle. Los *modelos de proceso prescriptivo* (capítulo 2) enfatizan la definición, la identificación y la aplicación detalladas de las actividades y tareas del proceso. Su objetivo es mejorar la calidad del sistema, desarrollar proyectos más manejables, hacer más predecibles las fechas de entrega y los costos, y guiar a los equipos de ingenieros de software cuando realizan el trabajo que se requiere para construir un sistema. Desafortunadamente, ha habido casos en los que estos objetivos no se han logrado. Si los modelos prescriptivos se aplican en forma dogmática y sin adaptación, pueden incrementar el nivel de burocracia asociada con el desarrollo de sistemas basados en computadora y crear inadvertidamente dificultades para todos los participantes.

? ¿Qué caracteriza a un proceso "ágil"?

Los *modelos de proceso ágil* (capítulo 3) ponen el énfasis en la "agilidad" del proyecto y siguen un conjunto de principios que conducen a un enfoque más informal (pero no menos efectivo, dicen sus defensores) del proceso de software. Por lo general, se dice que estos modelos del proceso son "ágiles" porque acentúan la maniobrabilidad y la adaptabilidad. Son apropiados para muchos tipos de proyectos y son útiles en particular cuando se hace ingeniería sobre aplicaciones web.

1.5 LA PRÁCTICA DE LA INGENIERÍA DE SOFTWARE

WebRef

En la dirección **www.literateprogramming.com** se encuentran varias citas provocativas sobre la práctica de la ingeniería de software.

En la sección 1.4 se introdujo un modelo general de proceso de software compuesto de un conjunto de actividades que establecen una estructura para la práctica de la ingeniería de software. Las actividades estructurales generales —**comunicación, planeación, modelado, construcción** y **despliegue**— y las actividades sombrilla establecen el esqueleto de la arquitectura para el trabajo de ingeniería de software. Pero, ¿cómo entra aquí la práctica de la ingeniería de software? En las secciones que siguen, el lector obtendrá la comprensión básica de los conceptos y principios generales que se aplican a las actividades estructurales.[12]

1.5.1 La esencia de la práctica

CONSEJO

Podría decirse que el enfoque de Polya es simple sentido común. Es verdad. Pero es sorprendente la frecuencia con la que el sentido común es poco común en el mundo del software.

En un libro clásico, *How to Solve It*, escrito antes de que existieran las computadoras modernas, George Polya [Pol45] describió la esencia de la solución de problemas y, en consecuencia, la esencia de la práctica de la ingeniería de software:

1. *Entender el problema* (comunicación y análisis).
2. *Planear la solución* (modelado y diseño del software).
3. *Ejecutar el plan* (generación del código).
4. *Examinar la exactitud del resultado* (probar y asegurar la calidad).

En el contexto de la ingeniería de software, estas etapas de sentido común conducen a una serie de preguntas esenciales [adaptado de Pol45]:

Entender el problema. En ocasiones es difícil de admitir, pero la mayor parte de nosotros adoptamos una actitud de orgullo desmedido cuando se nos presenta un problema. Escuchamos por unos segundos y después pensamos: *Claro, sí, entiendo, resolvamos esto.* Desafortunadamente, entender no siempre es fácil. Es conveniente dedicar un poco de tiempo a responder algunas preguntas sencillas:

- *¿Quiénes tienen que ver con la solución del problema?* Es decir, ¿quiénes son los participantes?
- *¿Cuáles son las incógnitas?* ¿Cuáles datos, funciones y características se requieren para resolver el problema en forma apropiada?
- *¿Puede fraccionarse el problema?* ¿Es posible representarlo con problemas más pequeños que sean más fáciles de entender?
- *¿Es posible representar gráficamente el problema?* ¿Puede crearse un modelo de análisis?

Cita:

"En la solución de cualquier problema hay un grano de descubrimiento."

George Polya

Planear la solución. Ahora entiende el problema (o es lo que piensa) y no puede esperar para escribir el código. Antes de hacerlo, cálmese un poco y haga un pequeño diseño:

- *¿Ha visto antes problemas similares?* ¿Hay patrones reconocibles en una solución potencial? ¿Hay algún software existente que implemente los datos, funciones y características que se requieren?
- *¿Ha resuelto un problema similar?* Si es así, ¿son reutilizables los elementos de la solución?
- *¿Pueden definirse problemas más pequeños?* Si así fuera, ¿hay soluciones evidentes para éstos?

12 El lector debería volver a consultar las secciones de este capítulo a medida que en el libro se describan en específico los métodos y las actividades sombrilla de la ingeniería de software.

- *¿Es capaz de representar una solución en una forma que lleve a su implementación eficaz?* ¿Es posible crear un modelo del diseño?

Ejecutar el plan. El diseño que creó sirve como un mapa de carreteras para el sistema que quiere construir. Puede haber desviaciones inesperadas y es posible que descubra un camino mejor a medida que avanza, pero el "plan" le permitirá proceder sin que se pierda.

- *¿Se ajusta la solución al plan?* ¿El código fuente puede apegarse al modelo del diseño?
- *¿Es probable que cada parte componente de la solución sea correcta?* ¿El diseño y código se han revisado o, mejor aún, se han hecho pruebas respecto de la corrección del algoritmo?

Examinar el resultado. No se puede estar seguro de que la solución sea perfecta, pero sí de que se ha diseñado un número suficiente de pruebas para descubrir tantos errores como sea posible.

- *¿Puede probarse cada parte componente de la solución?* ¿Se ha implementado una estrategia razonable para hacer pruebas?
- *¿La solución produce resultados que se apegan a los datos, funciones y características que se requieren?* ¿El software se ha validado contra todos los requerimientos de los participantes?

No debiera sorprender que gran parte de este enfoque tenga que ver con el sentido común. En realidad, es razonable afirmar que un enfoque de sentido común para la ingeniería de software hará que nunca se extravíe.

1.5.2 Principios generales

El diccionario define la palabra *principio* como "una ley importante o suposición que subyace y se requiere en un sistema de pensamiento". En este libro se analizarán principios en muchos niveles distintos de abstracción. Algunos se centran en la ingeniería de software como un todo, otros consideran una actividad estructural general específica (por ejemplo, **comunicación**), y otros más se centran en acciones de la ingeniería de software (por ejemplo, diseño de la arquitectura) o en tareas técnicas (escribir un escenario para el uso). Sin importar su nivel de enfoque, los principios lo ayudarán a establecer un conjunto de herramientas mentales para una práctica sólida de la ingeniería de software. Ésa es la razón de que sean importantes.

David Hooker [Hoo96] propuso siete principios que se centran en la práctica de la ingeniería de software como un todo. Se reproducen en los párrafos siguientes:[13]

Antes de comenzar un proyecto de software, asegúrese de que el software tenga un propósito para el negocio y que los usuarios perciben valor en él.

Primer principio: *La razón de que exista todo*

Un sistema de software existe por una razón: *dar valor a sus usuarios.* Todas las decisiones deben tomarse teniendo esto en mente. Antes de especificar un requerimiento del sistema, antes de notar la funcionalidad de una parte de él, antes de determinar las plataformas del hardware o desarrollar procesos, plantéese preguntas tales como: "¿Esto agrega valor real al sistema?" Si la respuesta es "no", entonces no lo haga. Todos los demás principios apoyan a éste.

Segundo principio: *MSE (Mantenlo sencillo, estúpido...)*

El diseño de software no es un proceso caprichoso. Hay muchos factores por considerar en cualquier actividad de diseño. *Todo diseño debe ser tan simple como sea posible, pero no más.*

13 Reproducido con permiso del autor [Hoo96]. Hooker define algunos patrones para estos principios en http://c2.com/cgi/wiki?SevenPrinciplesOfSoftwareDevelopment.

Cita:

"Hay cierta majestad en la sencillez, que es con mucho todo lo que adorna al ingenio."

Papa Alejandro (1688-1744)

Esto facilita conseguir un sistema que sea comprendido más fácilmente y que sea susceptible de recibir mantenimiento, lo que no quiere decir que en nombre de la simplicidad deban descartarse características o hasta rasgos internos. En realidad, los diseños más elegantes por lo general son los más simples. Simple tampoco significa "rápido y sucio". La verdad es que con frecuencia se requiere mucha reflexión y trabajo con iteraciones múltiples para poder simplificar. La recompensa es un software más fácil de mantener y menos propenso al error.

Tercer principio: *Mantener la visión*

Una *visión clara es esencial para el éxito de un proyecto de software*. Sin ella, casi infaliblemente el proyecto terminará siendo un ser "con dos [o más mentes]". Sin integridad conceptual, un sistema está amenazado de convertirse en una urdimbre de diseños incompatibles unidos por tornillos del tipo equivocado [...] Comprometer la visión de la arquitectura de un sistema de software debilita y, finalmente hará que colapsen incluso los sistemas bien diseñados. Tener un arquitecto que pueda para mantener la visión y que obligue a su cumplimiento garantiza un proyecto de software muy exitoso.

Cuarto principio: *Otros consumirán lo que usted produce*

PUNTO CLAVE

Si el software tiene valor, cambiará durante su vida útil. Por esa razón, debe construirse de forma que sea fácil darle mantenimiento.

Rara vez se construye en el vacío un sistema de software con fortaleza industrial. En un modo u otro, alguien más lo usará, mantendrá, documentará o, de alguna forma, dependerá de su capacidad para entender el sistema. Así que *siempre establezca especificaciones, diseñe e implemente con la seguridad de que alguien más tendrá que entender lo que usted haga*. La audiencia para cualquier producto de desarrollo de software es potencialmente grande. Elabore especificaciones con la mirada puesta en los usuarios. Diseñe con los implementadores en mente. Codifique pensando en aquellos que deben dar mantenimiento y ampliar el sistema. Alguien debe depurar el código que usted escriba, y eso lo hace usuario de su código. Hacer su trabajo más fácil agrega valor al sistema.

Quinto principio: *Ábrase al futuro*

Un sistema con larga vida útil tiene más valor. En los ambientes de cómputo actuales, donde las especificaciones cambian de un momento a otro y las plataformas de hardware quedan obsoletas con sólo unos meses de edad, es común que la vida útil del software se mida en meses y no en años. Sin embargo, los sistemas de software con verdadera "fortaleza industrial" deben durar mucho más tiempo. Para tener éxito en esto, los sistemas deben ser fáciles de adaptar a ésos y otros cambios. Los sistemas que lo logran son los que se diseñaron para ello desde el principio. *Nunca diseñe sobre algo iniciado.* Siempre pregunte: "¿qué pasa si...?" y prepárese para todas las respuestas posibles mediante la creación de sistemas que resuelvan el problema general, no sólo uno específico.[14] Es muy posible que esto lleve a volver a usar un sistema completo.

Sexto principio: *Planee por anticipado la reutilización*

La reutilización ahorra tiempo y esfuerzo.[15] Al desarrollar un sistema de software, lograr un alto nivel de reutilización es quizá la meta más difícil de lograr. La reutilización del código y de los diseños se ha reconocido como uno de los mayores beneficios de usar tecnologías orientadas a objetos. Sin embargo, la recuperación de esta inversión no es automática. Para reforzar las posibilidades de la reutilización que da la programación orientada a objetos [o la

14 Es peligroso llevar este consejo a los extremos. Diseñar para resolver "el problema general" en ocasiones requiere compromisos de rendimiento y puede volver ineficientes las soluciones específicas.

15 Aunque esto es verdad para aquellos que reutilizan software en proyectos futuros, volver a usar puede ser caro para quienes deben diseñar y elaborar componentes reutilizables. Los estudios indican que diseñar y construir componentes reutilizables llega a costar entre 25 y 200% más que el software buscado. En ciertos casos no se justifica la diferencia de costos.

convencional], se requiere reflexión y planeación. Hay muchas técnicas para incluir la reutilización en cada nivel del proceso de desarrollo del sistema... *La planeación anticipada en busca de la reutilización disminuye el costo e incrementa el valor tanto de los componentes reutilizables como de los sistemas en los que se incorpora.*

Séptimo principio: *¡Piense!*

Este último principio es tal vez el que más se pasa por alto. *Pensar en todo con claridad antes de emprender la acción casi siempre produce mejores resultados*. Cuando se piensa en algo es más probable que se haga bien. Asimismo, también se gana conocimiento al pensar cómo volver a hacerlo bien. Si usted piensa en algo y aun así lo hace mal, eso se convierte en una experiencia valiosa. Un efecto colateral de pensar es aprender a reconocer cuando no se sabe algo, punto en el que se puede investigar la respuesta. Cuando en un sistema se han puesto pensamientos claros, el valor se manifiesta. La aplicación de los primeros seis principios requiere pensar con intensidad, por lo que las recompensas potenciales son enormes.

Si todo ingeniero y equipo de software tan sólo siguiera los siete principios de Hooker, se eliminarían muchas de las dificultades que se experimentan al construir sistemas complejos basados en computadora.

1.6 MITOS DEL SOFTWARE

Cita:

"En ausencia de estándares significativos, una industria nueva como la del software depende sólo del folklore."

Tom DeMarco

Los mitos del software —creencias erróneas sobre éste y sobre el proceso que se utiliza para obtenerlo— se remontan a los primeros días de la computación. Los mitos tienen cierto número de atributos que los hacen insidiosos. Por ejemplo, parecen enunciados razonables de hechos (a veces contienen elementos de verdad), tienen una sensación intuitiva y es frecuente que los manifiesten profesionales experimentados que "conocen la historia".

En la actualidad, la mayoría de profesionales de la ingeniería de software reconocen los mitos como lo que son: actitudes equivocadas que han ocasionado serios problemas a los administradores y a los trabajadores por igual. Sin embargo, las actitudes y hábitos antiguos son difíciles de modificar, y persisten algunos remanentes de los mitos del software.

WebRef

La Software Project Managers Network (Red de Gerentes de Proyectos de Software), en **www.spmn.com**, lo ayuda a eliminar éstos y otros mitos.

Mitos de la administración. Los gerentes que tienen responsabilidades en el software, como los de otras disciplinas, con frecuencia se hallan bajo presión para cumplir el presupuesto, mantener la programación de actividades sin desvíos y mejorar la calidad. Así como la persona que se ahoga se agarra de un clavo ardiente, no es raro que un gerente de software sostenga la creencia en un mito del software si eso disminuye la presión a que está sujeto (incluso de manera temporal).

Mito: *Tenemos un libro lleno de estándares y procedimientos para elaborar software. ¿No le dará a mi personal todo lo que necesita saber?*

Realidad: Tal vez exista el libro de estándares, pero ¿se utiliza? ¿Saben de su existencia los trabajadores del software? ¿Refleja la práctica moderna de la ingeniería de software? ¿Es completo? ¿Es adaptable? ¿Está dirigido a mejorar la entrega a tiempo y también se centra en la calidad? En muchos casos, la respuesta a todas estas preguntas es "no".

Mito: *Si nos atrasamos, podemos agregar más programadores y ponernos al corriente (en ocasiones, a esto se le llama "concepto de la horda de mongoles").*

Realidad: El desarrollo del software no es un proceso mecánico similar a la manufactura. En palabras de Brooks [Bro95]: "agregar personal a un proyecto de software atrasado lo atrasará más". Al principio, esta afirmación parece ir contra la intuición. Sin embargo, a medida que se agregan personas, las que ya se

encontraban trabajando deben dedicar tiempo para enseñar a los recién llegados, lo que disminuye la cantidad de tiempo dedicada al esfuerzo de desarrollo productivo. Pueden agregarse individuos, pero sólo en forma planeada y bien coordinada.

Mito: *Si decido subcontratar el proyecto de software a un tercero, puedo descansar y dejar que esa compañía lo elabore.*

Realidad: Si una organización no comprende cómo administrar y controlar proyectos de software internamente, de manera invariable tendrá dificultades cuando subcontrate proyectos de software.

Mitos del cliente. El cliente que requiere software de computadora puede ser la persona en el escritorio de al lado, un grupo técnico en el piso inferior, el departamento de mercadotecnia y ventas, o una compañía externa que solicita software mediante un contrato. En muchos casos, el cliente sostiene mitos sobre el software porque los gerentes y profesionales de éste hacen poco para corregir la mala información. Los mitos generan falsas expectativas (por parte del cliente) y, en última instancia, la insatisfacción con el desarrollador.

Trabaje muy duro para entender qué es lo que tiene que hacer antes de empezar. Quizás no pueda desarrollarlo a detalle, pero entre más sepa, menor será el riesgo que tome.

Mito: *Para comenzar a escribir programas, es suficiente el enunciado general de los objetivos —podremos entrar en detalles más adelante.*

Realidad: Aunque no siempre es posible tener el enunciado exhaustivo y estable de los requerimientos, un "planteamiento de objetivos" ambiguo es una receta para el desastre. Los requerimientos que no son ambiguos (que por lo general se obtienen en forma iterativa) se desarrollan sólo por medio de una comunicación eficaz y continua entre el cliente y el desarrollador.

Mito: *Los requerimientos del software cambian continuamente, pero el cambio se asimila con facilidad debido a que el software es flexible.*

Realidad: Es verdad que los requerimientos del software cambian, pero el efecto que los cambios tienen varía según la época en la que se introducen. Cuando se solicitan al principio cambios en los requerimientos (antes de que haya comenzado el diseño o la elaboración de código), el efecto sobre el costo es relativamente pequeño.[16] Sin embargo, conforme pasa el tiempo, el costo aumenta con rapidez: los recursos ya se han comprometido, se ha establecido la estructura del diseño y el cambio ocasiona perturbaciones que exigen recursos adicionales y modificaciones importantes del diseño.

Siempre que piense que no hay tiempo para la ingeniería de software, pregúntese: "¿tendremos tiempo de hacerlo otra vez?".

Mitos del profesional. Los mitos que aún sostienen los trabajadores del software han sido alimentados por más de 50 años de cultura de programación. Durante los primeros días, la programación se veía como una forma del arte. Es difícil que mueran los hábitos y actitudes arraigados.

Mito: *Una vez que escribimos el programa y hacemos que funcione, nuestro trabajo ha terminado.*

Realidad: Alguien dijo alguna vez que "entre más pronto se comience a 'escribir el código', más tiempo tomará hacer que funcione". Los datos de la industria indican que entre 60 y 80% de todo el esfuerzo dedicado al software ocurrirá después de entregarlo al cliente por primera vez.

Mito: *Hasta que no se haga "correr" el programa, no hay manera de evaluar su calidad.*

16 Muchos ingenieros de software han adoptado un enfoque "ágil" que asimila los cambios en forma gradual y creciente, con lo que controlan su efecto y costo. Los métodos ágiles se estudian en el capítulo 3.

Realidad: Uno de los mecanismos más eficaces de asegurar la calidad del software puede aplicarse desde la concepción del proyecto: *la revisión técnica*. Las revisiones del software (descritas en el capítulo 15) son un "filtro de la calidad" que se ha revelado más eficaz que las pruebas para encontrar ciertas clases de defectos de software.

Mito: *El único producto del trabajo que se entrega en un proyecto exitoso es el programa que funciona.*

Realidad: Un programa que funciona sólo es una parte de una configuración de software que incluye muchos elementos. Son varios los productos terminados (modelos, documentos, planes) que proporcionan la base de la ingeniería exitosa y, lo más importante, que guían el apoyo para el software.

Mito: *La ingeniería de software hará que generemos documentación voluminosa e innecesaria, e invariablemente nos retrasará.*

Realidad: La ingeniería de software no consiste en producir documentos. Se trata de crear un producto de calidad. La mejor calidad conduce a menos repeticiones, lo que da como resultado tiempos de entrega más cortos.

Muchos profesionales del software reconocen la falacia de los mitos mencionados. Es lamentable que las actitudes y métodos habituales nutran la administración y las prácticas técnicas deficientes, aun cuando la realidad dicta un enfoque mejor. El primer paso hacia la formulación de soluciones prácticas para la ingeniería de software es el reconocimiento de las realidades en este campo.

1.7 CÓMO COMIENZA TODO

Todo proyecto de software se desencadena por alguna necesidad de negocios: la de corregir un defecto en una aplicación existente, la de adaptar un "sistema heredado" a un ambiente de negocios cambiante, la de ampliar las funciones y características de una aplicación ya existente o la necesidad de crear un producto, servicio o sistema nuevo.

Al comenzar un proyecto de software, es frecuente que las necesidades del negocio se expresen de manera informal como parte de una simple conversación. La plática que se presenta en el recuadro que sigue es muy común.

CASASEGURA[17]

Cómo se inicia un proyecto

La escena: Sala de juntas en CPI Corporation, empresa (ficticia) que manufactura productos de consumo para uso doméstico y comercial.

Participantes: Mal Golden, alto directivo de desarrollo de productos; Lisa Pérez, gerente comercial; Lee Warren, gerente de ingeniería; Joe Camalleri, VP ejecutivo, desarrollo de negocios.

La conversación:

Joe: Oye, Lee, ¿qué es eso que oí acerca de que tu gente va a desarrollar no sé qué? ¿Una caja inalámbrica universal general?

Lee: Es sensacional… más o menos del tamaño de una caja de cerillos pequeña… podemos conectarla a sensores de todo tipo, una cámara digital… a cualquier cosa. Usa el protocolo 802.11g inalámbrico. Permite el acceso a la salida de dispositivos sin cables. Pensamos que llevará a toda una nueva generación de productos.

Joe: ¿Estás de acuerdo, Mal?

Mal: Sí. En realidad, con las ventas tan planas que hemos tenido este año necesitamos algo nuevo. Lisa y yo hemos hecho algo de investigación del mercado y pensamos que tenemos una línea de productos que podría ser algo grande.

17 El proyecto *CasaSegura* se usará en todo el libro para ilustrar los entretelones de un equipo de proyecto que elabora un producto de software. La compañía, el proyecto y las personas son ficticias, pero las situaciones y problemas son reales.

Joe: ¿Cuán grande… tanto como el renglón de utilidades?

Mal (que evita el compromiso directo): Cuéntale nuestra idea, Lisa.

Lisa: Es toda una nueva generación que hemos llamado "productos para la administración del hogar". Le dimos el nombre de *CasaSegura*. Usan la nueva interfaz inalámbrica, proporcionan a los dueños de viviendas o pequeños negocios un sistema controlado por su PC —seguridad del hogar, vigilancia, control de aparatos y equipos—, tú sabes, apaga el aire acondicionado cuando sales de casa, esa clase de cosas.

Lee (dando un brinco): La oficina de ingeniería hizo un estudio de factibilidad técnica de esta idea, Joe. Es algo realizable con un costo bajo de manufactura. La mayor parte del hardware es de línea. Queda pendiente el software, pero no es algo que no podamos hacer.

Joe: Interesante. Pero pregunté sobre las utilidades.

Mal: Las PC han penetrado a 70 por ciento de los hogares de Estados Unidos. Si lo vendemos en el precio correcto, podría ser una aplicación sensacional. Nadie tiene nuestra caja inalámbrica… somos dueños. Nos adelantaremos dos años a la competencia. ¿Las ganancias? Quizá tanto como 30 a 40 millones de dólares en el segundo año.

Joe (sonriente): Llevemos esto al siguiente nivel. Estoy interesado.

Con excepción de una referencia casual, el software no se mencionó en la conversación. Y, sin embargo, es lo que hará triunfar o fracasar la línea de productos *CasaSegura*. El esfuerzo de ingeniería tendrá éxito sólo si también lo tiene el software de *CasaSegura*. El mercado aceptará el producto sólo si el software incrustado en éste satisface las necesidades del cliente (aún no establecidas). En muchos de los capítulos siguientes continuaremos el avance de la ingeniería del software en *CasaSegura*.

1.8 Resumen

El software es un elemento clave en la evolución de sistemas y productos basados en computadoras, y una de las tecnologías más importantes en todo el mundo. En los últimos 50 años, el software ha pasado de ser la solución de un problema especializado y herramienta de análisis de la información a una industria en sí misma. No obstante, aún hay problemas para desarrollar software de alta calidad a tiempo y dentro del presupuesto asignado.

El software —programas, datos e información descriptiva— se dirige a una gama amplia de tecnología y campos de aplicación. El software heredado sigue planteando retos especiales a quienes deben darle mantenimiento.

Los sistemas y aplicaciones basados en web han evolucionado de simples conjuntos de contenido de información a sistemas sofisticados que presentan una funcionalidad compleja y contenido en multimedios. Aunque dichas *webapps* tienen características y requerimientos únicos, son software.

La ingeniería de software incluye procesos, métodos y herramientas que permiten elaborar a tiempo y con calidad sistemas complejos basados en computadoras. El proceso de software incorpora cinco actividades estructurales: comunicación, planeación, modelado, construcción y despliegue que son aplicables a todos los proyectos de software. La práctica de la ingeniería de software es una actividad para resolver problemas, que sigue un conjunto de principios fundamentales.

Muchos mitos del software todavía hacen que administradores y trabajadores se equivoquen, aun cuando ha aumentado nuestro conocimiento colectivo del software y las tecnologías requeridas para elaborarlo. Conforme el lector aprenda más sobre ingeniería de software, comenzará a entender por qué deben rebatirse estos mitos cada vez que surjan.

Problemas y puntos por evaluar

1.1. Dé al menos cinco ejemplos de la forma en que se aplica la ley de las consecuencias imprevistas al software de cómputo.

1.2. Diga algunos ejemplos (tanto positivos como negativos) que indiquen el efecto del software en nuestra sociedad.

1.3. Desarrolle sus propias respuestas a las cinco preguntas planteadas al principio de la sección 1.1. Analícelas con sus compañeros estudiantes.

1.4. Muchas aplicaciones modernas cambian con frecuencia, antes de que se presenten al usuario final y después de que la primera versión ha entrado en uso. Sugiera algunos modos de elaborar software para detener el deterioro que produce el cambio.

1.5. Considere las siete categorías de software presentadas en la sección 1.1.2. ¿Piensa que puede aplicarse a cada una el mismo enfoque de ingeniería de software? Explique su respuesta.

1.6. La figura 1.3 muestra las tres capas de la ingeniería de software arriba de otra llamada "compromiso con la calidad". Esto implica un programa de calidad organizacional como el enfoque de la administración total de la calidad. Haga un poco de investigación y desarrolle los lineamientos de los elementos clave de un programa para la administración de la calidad.

1.7. ¿Es aplicable la ingeniería de software cuando se elaboran *webapps*? Si es así, ¿cómo puede modificarse para que asimile las características únicas de éstas?

1.8. A medida que el software gana ubicuidad, los riesgos para el público (debidos a programas defectuosos) se convierten en motivo de preocupación significativa. Desarrolle un escenario catastrófico pero realista en el que la falla de un programa de cómputo pudiera ocasionar un gran daño (económico o humano).

1.9. Describa con sus propias palabras una estructura de proceso. Cuando se dice que las actividades estructurales son aplicables a todos los proyectos, ¿significa que se realizan las mismas tareas en todos los proyectos sin que importe su tamaño y complejidad? Explique su respuesta.

1.10. Las actividades sombrilla ocurren a través de todo el proceso del software. ¿Piensa usted que son aplicables por igual a través del proceso, o que algunas se concentran en una o más actividades estructurales?

1.11. Agregue dos mitos adicionales a la lista presentada en la sección 1.6. También diga la realidad que acompaña al mito.

Lecturas adicionales y fuentes de información[18]

Hay literalmente miles de libros escritos sobre software de cómputo. La gran mayoría analiza lenguajes de programación o aplicaciones de software, pero algunos estudian al software en sí mismo. Pressman y Herron (*Software Shock*, Dorset House, 1991) presentaron un estudio temprano (dirigido a las personas comunes) sobre el software y la forma en la que lo elaboran los profesionales. El libro de Negroponte que se convirtió en un éxito de ventas (*Being Digital*, Alfred A. Knopf, Inc., 1995) describe el panorama de la computación y su efecto general en el siglo XXI. DeMarco (*Why Does Software Cost So Much?*, Dorset House, 1995) ha producido varios ensayos amenos y profundos sobre el software y el proceso con el que se elabora.

Minasi (*The Software Conspiracy: Why Software Companies Put out Faulty Products, How They Can Hurt You, and What You Can Do*, McGraw-Hill, 2000) afirma que el "flagelo moderno" de los errores en el software puede eliminarse y sugiere formas de lograrlo. Compaine (*Digital Divide: Facing A Crisis or Creating a Myth*, MIT Press, 2001) asegura que la "división" entre aquellos que tienen acceso a recursos de la información (por ejemplo, la web) y los que no lo tienen se está estrechando conforme avanzamos en la primera década de este siglo. Los libros escritos por Greenfield (*Everyware: The Dawning Age of Ubiquitous Computing*, New Riders Publishing, 2006) y Loke (*Context-Aware Pervasive Systems: Architectures for a New Breed of Applications*, Auerbach, 2006) introducen el concepto de software de "mundo abierto" y predicen un ambiente inalámbrico en el que el software deba adaptarse a los requerimientos que surjan en tiempo real.

---

18 La sección de "Lecturas adicionales y fuentes de información" que se presenta al final de cada capítulo expone un panorama breve de fuentes impresas que ayudan a aumentar la comprensión de los principales temas presentados. El autor ha creado un sitio web para apoyar al libro *Ingeniería de software: enfoque del profesional* en www.mhhe.com/compsci/pressman. Entre los muchos temas que se abordan en dicho sitio, se encuentran desde los recursos de la ingeniería de software capítulo por capítulo hasta información basada en web que complementa el material presentado. Como parte de esos recursos se halla un vínculo hacia Amazon.com para cada libro citado en esta sección.

El estado actual de la ingeniería y del proceso de software se determina mejor a partir de publicaciones tales como *IEEE Software, IEEE Computer, CrossTalk* y *IEEE Transactions on Software Engineering*. Publicaciones periódicas como *Application Development Trends* y *Cutter IT Journal* con frecuencia contienen artículos sobre temas de ingeniería de software. La disciplina se "resume" cada año en *Proceeding of the International Conference on Software Engineering*, patrocinada por IEEE y ACM, y se analiza a profundidad en revistas tales como *ACM Transactions on Software Engineering and Methodology, ACM Software Engineering Notes* y *Annals of Software Engineering*. Hay decenas de miles de sitios web dedicados a la ingeniería y al proceso de software.

En años recientes se han publicado muchos libros que abordan el proceso y la ingeniería de software. Algunos presentan un panorama de todo el proceso, mientras otros profundizan en algunos temas importantes y omiten otros. Entre los más populares (¡además del que tiene usted en sus manos!) se encuentran los siguientes:

Abran, A., and J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge,* IEEE, 2002.

Andersson, E., et al., *Software Engineering for Internet Applications,* The MIT Press, 2006.

Christensen, M., and R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices,* IEEE-CS Press (Wiley), 2002.

Glass, R., *Fact and Fallacies of Software Engineering,* Addison-Wesley, 2002.

Jacobson, I., *Object-Oriented Software Engineering: A Use Case Driven Approach,* 2d ed., Addison-Wesley, 2008.

Jalote, P., *An Integrated Approach to Software Engineering,* Springer, 2006.

Pfleeger, S., *Software Engineering: Theory and Practice,* 3d ed., Prentice-Hall, 2005.

Schach, S., *Object-Oriented and Classical Software Engineering,* 7th ed., McGraw-Hill, 2006.

Sommerville, I., *Software Engineering,* 8th ed., Addison-Wesley, 2006.

Tsui, F., and O. Karam, *Essentials of Software Engineering,* Jones & Bartlett Publishers, 2006.

En las últimas décadas, son muchos los estándares para la ingeniería de software que han sido publicados por IEEE, ISO y sus organizaciones. Moore (*The Road Map to Software Engineering: A Standards-Based Guide*, Wiley-IEEE Computer Society Press, 2006) proporciona una revisión útil de los estándares relevantes y la forma en la que se aplican a proyectos reales.

En internet se encuentra disponible una amplia variedad de fuentes acerca de la ingeniería y el proceso de software. Una lista actualizada de referencias en la Red Mundial que son útiles para el proceso de software se encuentra en el sitio web del libro, en la dirección **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

PARTE

Uno

El proceso del software

En esta parte de la obra, aprenderá sobre el proceso que genera una estructura para la práctica de la ingeniería de software. En los capítulos que siguen se abordan preguntas como las siguientes:

- ¿Qué es el proceso del software?
- ¿Cuáles son las actividades estructurales generales que están presentes en todo proceso del software?
- ¿Cómo se modelan los procesos y cuáles son los patrones del proceso?
- ¿Cuáles son los modelos prescriptivos del proceso y cuáles son sus fortalezas y debilidades?
- ¿Por qué la *agilidad* es un imperativo en la ingeniería de software moderna?
- ¿Qué es un desarrollo ágil del software y en qué se diferencia de los modelos más tradicionales del proceso?

Una vez respondidas estas preguntas, el lector estará mejor preparado para entender el contexto en el que se aplica la práctica de la ingeniería de software.

CAPÍTULO

2 MODELOS DEL PROCESO

CONCEPTOS CLAVE

conjunto de tareas..... 29
desarrollo basado en componentes 43
modelo de métodos formales..... 44
modelo general de proceso... 27
modelos concurrentes 40
modelos de proceso evolutivo 36
modelos de proceso incremental..... 35
modelos de proceso prescriptivo 33
patrones del proceso..... 29
proceso del equipo de software 49
proceso personal del software..... 48
proceso unificado 45

En un libro fascinante que expone el punto de vista de un economista sobre el software y su ingeniería, Howard Baetjer, Jr. [Bae98] comenta acerca del proceso del software.

> Debido a que el software, como todo capital, es conocimiento incorporado y a que el conocimiento originalmente se halla disperso, tácito, latente e incompleto en gran medida, el desarrollo de software es un proceso de aprendizaje social. El proceso es un diálogo en el que el conocimiento que debe convertirse en software se reúne e incorpora en éste. El proceso genera interacción entre usuarios y diseñadores, entre usuarios y herramientas cambiantes, y entre diseñadores y herramientas en evolución [tecnología]. Es un proceso que se repite y en el que la herramienta que evoluciona sirve por sí misma como medio para la comunicación: con cada nueva ronda del diálogo se genera más conocimiento útil a partir de las personas involucradas.

En realidad, la elaboración de software de computadora es un proceso reiterativo de aprendizaje social, y el resultado, algo que Baetjer llamaría "capital de software", es la reunión de conocimiento recabado, depurado y organizado a medida que se realiza el proceso.

Pero desde el punto de vista técnico, ¿qué es exactamente un proceso del software? En el contexto de este libro, se define *proceso del software* como una estructura para las actividades, acciones y tareas que se requieren a fin de construir software de alta calidad. ¿"Proceso" es sinónimo de "ingeniería de software"? La respuesta es "sí y no". Un proceso del software define el enfoque adoptado mientras se hace ingeniería sobre el software. Pero la ingeniería de software también incluye tecnologías que pueblan el proceso: métodos técnicos y herramientas automatizadas.

Más importante aún, la ingeniería de software es llevada a cabo por personas creativas y preparadas que deben adaptar un proceso maduro de software a fin de que resulte apropiado para los productos que construyen y para las demandas de su mercado.

UNA MIRADA RÁPIDA

¿Qué es? Cuando se trabaja en la construcción de un producto o sistema, es importante ejecutar una serie de pasos predecibles —el mapa de carreteras que lo ayuda a obtener a tiempo un resultado de alta calidad—. El mapa que se sigue se llama "proceso del software".

¿Quién lo hace? Los ingenieros de software y sus gerentes adaptan el proceso a sus necesidades y luego lo siguen. Además, las personas que solicitaron el software tienen un papel en el proceso de definición, elaboración y prueba.

¿Por qué es importante? Porque da estabilidad, control y organización a una actividad que puede volverse caótica si se descontrola. Sin embargo, un enfoque moderno de ingeniería de software debe ser "ágil". Debe incluir sólo aquellas actividades, controles y productos del trabajo que sean apropiados para el equipo del proyecto y para el producto que se busca obtener.

¿Cuáles son los pasos? En un nivel detallado, el proceso que se adopte depende del software que se esté elaborando. Un proceso puede ser apropiado para crear software destinado a un sistema de control electrónico de un aeroplano, mientras que para la creación de un sitio web será necesario un proceso completamente distinto.

¿Cuál es el producto final? Desde el punto de vista de un ingeniero de software, los productos del trabajo son los programas, documentos y datos que se producen como consecuencia de las actividades y tareas definidas por el proceso.

¿Cómo me aseguro de que lo hice bien? Hay cierto número de mecanismos de evaluación del proceso del software que permiten que las organizaciones determinen la "madurez" de su proceso. Sin embargo, la calidad, oportunidad y viabilidad a largo plazo del producto que se elabora son los mejores indicadores de la eficacia del proceso que se utiliza.

2.1 UN MODELO GENERAL DE PROCESO

En el capítulo 1 se definió un proceso como la colección de actividades de trabajo, acciones y tareas que se realizan cuando va a crearse algún producto terminado. Cada una de las actividades, acciones y tareas se encuentra dentro de una estructura o modelo que define su relación tanto con el proceso como entre sí.

En la figura 2.1 se representa el proceso del software de manera esquemática. En dicha figura, cada actividad estructural está formada por un conjunto de acciones de ingeniería de software y cada una de éstas se encuentra definida por un *conjunto de tareas* que identifica las tareas del trabajo que deben realizarse, los productos del trabajo que se producirán, los puntos de aseguramiento de la calidad que se requieren y los puntos de referencia que se utilizarán para evaluar el avance.

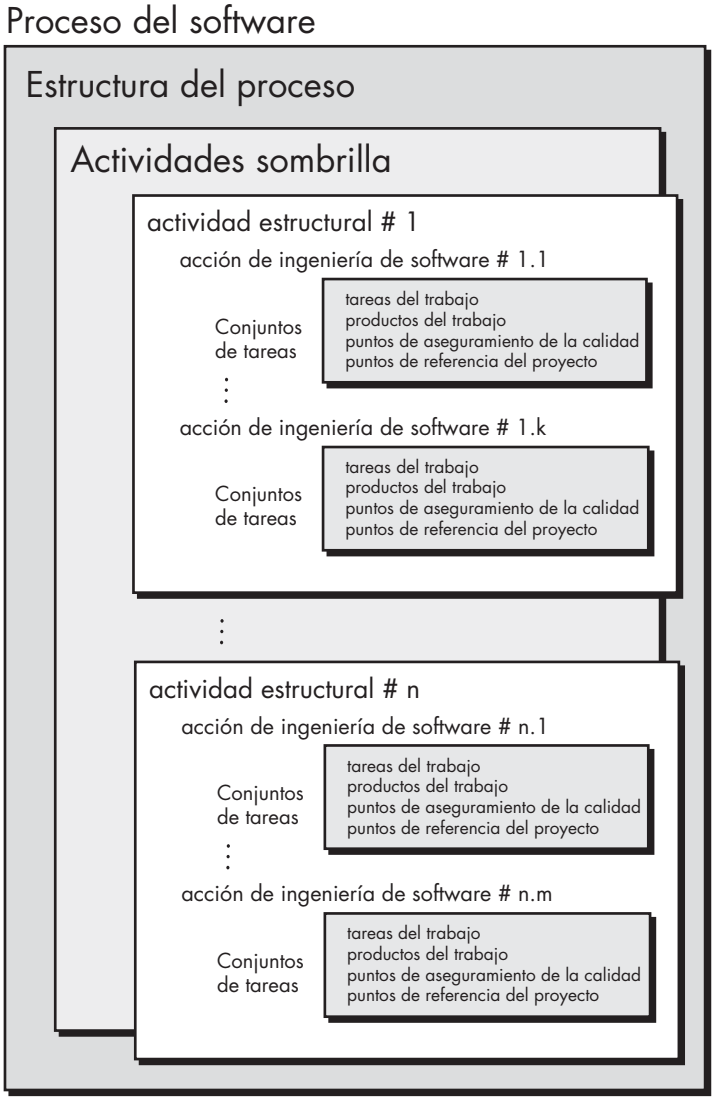
PUNTO CLAVE

La jerarquía del trabajo técnico dentro del proceso del software es: actividades, acciones que contiene y tareas constituyentes.

Como se dijo en el capítulo 1, una estructura general para la ingeniería de software define cinco actividades estructurales: **comunicación, planeación, modelado, construcción** y **despliegue**. Además, a lo largo de todo el proceso se aplica un conjunto de actividades som-

FIGURA 2.1

Estructura de un proceso del software



brilla: seguimiento y control del proyecto, administración de riesgos, aseguramiento de la calidad, administración de la configuración, revisiones técnicas, entre otras.

Cita:

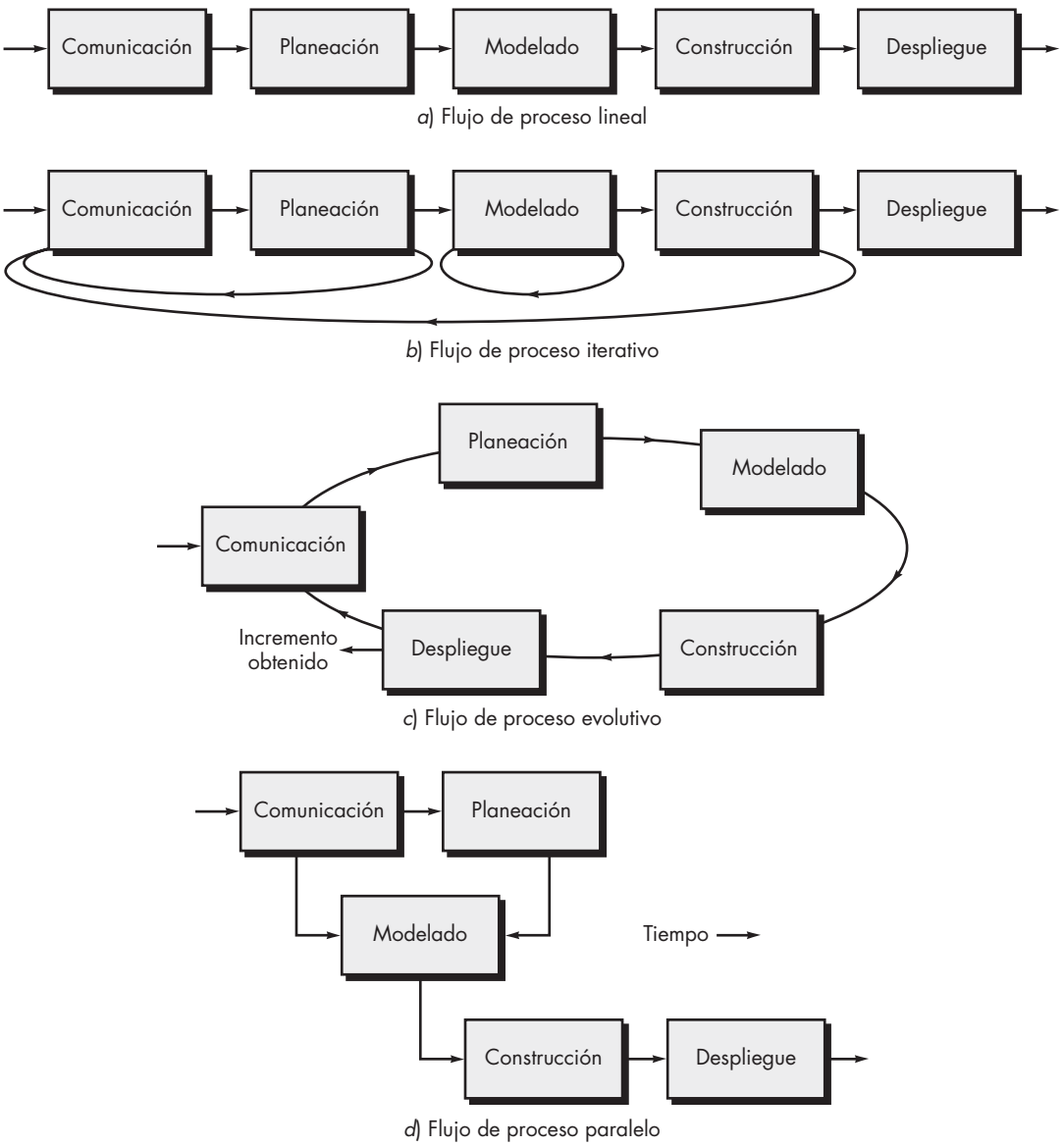
"Pensamos que los desarrolladores de software pierden de vista una verdad fundamental: la mayor parte de organizaciones no saben lo que hacen. Piensan que lo saben, pero no es así."

Tom DeMarco

El lector debe observar que aún no se menciona un aspecto importante del proceso del software. En la figura 2.2 se ilustra dicho aspecto —llamado *flujo del proceso*— y se describe la manera en que están organizadas las actividades estructurales y las acciones y tareas que ocurren dentro de cada una con respecto de la secuencia y el tiempo.

Un *flujo de proceso lineal* ejecuta cada una de las cinco actividades estructurales en secuencia, comenzando por la comunicación y terminando con el despliegue (véase la figura 2.2*a*). Un *flujo de proceso iterativo* repite una o más de las actividades antes de pasar a la siguiente (véase la figura 2.2*b*). Un *flujo de proceso evolutivo* realiza las actividades en forma "circular". A través de las cinco actividades, cada circuito lleva a una versión más completa del software (véase la figura 2.2*c*). Un *flujo de proceso paralelo* (véase la figura 2.2*d*) ejecuta una o más actividades en

FIGURA 2.2 Flujo del proceso



paralelo con otras (por ejemplo, el modelado de un aspecto del software tal vez se ejecute en paralelo con la construcción de otro aspecto del software).

2.1.1 Definición de actividad estructural

Aunque en el capítulo 1 se describieron cinco actividades estructurales y se dio una definición básica de cada una, un equipo de software necesitará mucha más información antes de poder ejecutar de manera apropiada cualquiera de ellas como parte del proceso del software. Por tanto, surge una pregunta clave: *¿qué acciones son apropiadas para una actividad estructural, dados la naturaleza del problema por resolver, las características de las personas que hacen el trabajo y los participantes que patrocinan el proyecto?*

¿Cómo se transforma una actividad estructural cuando cambia la naturaleza del proyecto?

Para un proyecto de software pequeño solicitado por una persona (en una ubicación remota) con requerimientos sencillos y directos, la actividad de comunicación tal vez no incluya algo más que una llamada telefónica con el participante apropiado. Entonces, la única acción necesaria es una *conversación telefónica*, y las tareas del trabajo (el *conjunto de tareas*) que engloba son las siguientes:

1. Hacer contacto con el participante por vía telefónica.
2. Analizar los requerimientos y tomar notas.
3. Organizar las notas por escrito en una formulación breve de los requerimientos.
4. Enviar correo electrónico al participante para que revise y apruebe.

Si el proyecto fuera considerablemente más complejo, con muchos participantes y cada uno con un distinto conjunto de requerimientos (a veces en conflicto), la actividad de comunicación puede tener seis acciones distintas (descritas en el capítulo 5): *concepción, indagación, elaboración, negociación, especificación* y *validación*. Cada una de estas acciones de la ingeniería del software tendrá muchas tareas de trabajo y un número grande de diferentes productos finales.

PUNTO CLAVE

Diferentes proyectos demandan diferentes conjuntos de tareas. El equipo de software elige el conjunto de tareas con base en las características del problema y el proyecto.

2.1.2 Identificación de un conjunto de tareas

En relación con la figura 2.1, cada acción de la ingeniería de software (por ejemplo, *obtención*, asociada a la actividad de comunicación) se representa por cierto número de distintos *conjuntos de tareas*, cada uno de los cuales es una colección de tareas de trabajo de la ingeniería de software, relacionadas con productos del trabajo, puntos de aseguramiento de la calidad y puntos de referencia del proyecto. Debe escogerse el conjunto de tareas que se adapte mejor a las necesidades del proyecto y a las características del equipo. Esto implica que una acción de la ingeniería de software puede adaptarse a las necesidades específicas del proyecto de software y a las características del equipo del proyecto.

2.1.3 Patrones del proceso

¿Qué es un patrón del proceso?

Cada equipo de software se enfrenta a problemas conforme avanza en el proceso del software. Si se demostrara que existen soluciones fáciles para dichos problemas, sería útil para el equipo abordarlos y resolverlos rápidamente. Un *patrón del proceso*[1] describe un problema relacionado con el proceso que se encuentra durante el trabajo de ingeniería de software, identifica el ambiente en el que surge el problema y sugiere una o más soluciones para el mismo. Dicho de manera general, un patrón de proceso da un formato [Amb98]: un método consistente para describir soluciones del problema en el contexto del proceso del software. Al combinar patrones, un equipo de software resuelve problemas y construye el proceso que mejor satisfaga las necesidades de un proyecto.

1 En el capítulo 12 se hace el análisis detallado de los patrones.

Información

Conjunto de tareas

Un conjunto de tareas define el trabajo real por efectuar a fin de cumplir los objetivos de una acción de ingeniería de software. Por ejemplo, la *indagación* (mejor conocida como "recabar los requerimientos") es una acción importante de la ingeniería de software que ocurre durante la actividad de comunicación. La meta al recabar los requerimientos es entender lo que los distintos participantes desean del software que se va a elaborar.

Para un proyecto pequeño y relativamente sencillo, el conjunto de tareas para la indagación de requerimientos tendrá un aspecto parecido al siguiente:

1. Elaborar la lista de participantes del proyecto.
2. Invitar a todos los participantes a una reunión informal.
3. Pedir a cada participante que haga una relación de las características y funciones que requiere.
4. Analizar los requerimientos y construir la lista definitiva.
5. Ordenar los requerimientos según su prioridad.
6. Identificar las áreas de incertidumbre.

Para un proyecto de software más grande y complejo se requerirá de un conjunto de tareas diferente que quizá esté constituido por las siguientes tareas de trabajo:

1. Hacer la lista de participantes del proyecto.
2. Entrevistar a cada participante por separado a fin de determinar los deseos y necesidades generales.
3. Formar la lista preliminar de las funciones y características con base en las aportaciones del participante.
4. Programar una serie de reuniones para facilitar la elaboración de las especificaciones de la aplicación.
5. Celebrar las reuniones.
6. Producir en cada reunión escenarios informales de usuario.
7. Afinar los escenarios del usuario con base en la retroalimentación de los participantes.
8. Formar una lista revisada de los requerimientos de los participantes.
9. Usar técnicas de despliegue de la función de calidad para asignar prioridades a los requerimientos.
10. Agrupar los requerimientos de modo que puedan entregarse en forma paulatina y creciente.
11. Resaltar las limitantes y restricciones que se introducirán al sistema.
12. Analizar métodos para validar el sistema.

Los dos conjuntos de tareas mencionados sirven para "recabar los requerimientos", pero son muy distintos en profundidad y formalidad. El equipo de software elige el conjunto de tareas que le permita alcanzar la meta de cada acción con calidad y agilidad.

Cita:

"La repetición de patrones es algo muy diferente de la repetición de las partes. En realidad, las distintas partes serán únicas porque los patrones son los mismos."

Christopher Alexander

Un formato de patrón proporciona un medio consistente para describir al patrón.

Los patrones se definen en cualquier nivel de abstracción.[2] En ciertos casos, un patrón puede usarse para describir un problema (y su solución) asociado con un modelo completo del proceso (por ejemplo, hacer prototipos). En otras situaciones, los patrones se utilizan para describir un problema (y su solución) asociado con una actividad estructural (por ejemplo, **planeación**) o una acción dentro de una actividad estructural (estimación de proyectos).

Ambler [Amb98] ha propuesto un formato para describir un patrón del proceso:

Nombre del patrón. El patrón recibe un nombre significativo que lo describe en el contexto del proceso del software (por ejemplo, **RevisionesTécnicas**).

Fuerzas. El ambiente en el que se encuentra el patrón y los aspectos que hacen visible el problema y afectan su solución.

Tipo. Se especifica el tipo de patrón. Ambler [Amb98] sugiere tres tipos:

1. *Patrón de etapa*: define un problema asociado con una actividad estructural para el proceso. Como una actividad estructural incluye múltiples acciones y tareas del trabajo, un patrón de la etapa incorpora múltiples patrones de la tarea (véase a continuación) que son relevantes para la etapa (actividad estructural). Un ejemplo de patrón de etapa sería **EstablecerComunicación**. Este patrón incorporaría el patrón de tarea **RecabarRequerimientos** y otros más.
2. *Patrón de tarea*: define un problema asociado con una acción o tarea de trabajo de la ingeniería de software y que es relevante para el éxito de la práctica de ingeniería de software (por ejemplo, **RecabarRequerimientos** es un patrón de tarea).

2 Los patrones son aplicables a muchas actividades de la ingeniería de software. El análisis, el diseño y la prueba de patrones se estudian en los capítulos 7, 9, 10, 12 y 14. Los patrones y "antipatrones" para las actividades de administración de proyectos se analizan en la parte 4 del libro.

3. *Patrón de fase*: define la secuencia de las actividades estructurales que ocurren dentro del proceso, aun cuando el flujo general de las actividades sea de naturaleza iterativa. Un ejemplo de patrón de fase es **ModeloEspiral** o **Prototipos**.[3]

Contexto inicial. Describe las condiciones en las que se aplica el patrón. Antes de iniciar el patrón: 1) ¿Qué actividades organizacionales o relacionadas con el equipo han ocurrido? 2) ¿Cuál es el estado de entrada para el proceso? 3) ¿Qué información de ingeniería de software o del proyecto ya existe?

Por ejemplo, el patrón **Planeación** (patrón de etapa) requiere que: 1) los clientes y los ingenieros de software hayan establecido una comunicación colaboradora; 2) haya terminado con éxito cierto número de patrones de tarea [especificado] para el patrón **Comunicación**; y 3) se conozcan el alcance del proyecto, los requerimientos básicos del negocio y las restricciones del proyecto.

Problema. El problema específico que debe resolver el patrón.

Solución. Describe cómo implementar con éxito el patrón. Esta sección describe la forma en la que se modifica el estado inicial del proceso (que existe antes de implementar el patrón) como consecuencia de la iniciación del patrón. También describe cómo se transforma la información sobre la ingeniería de software o sobre el proyecto, disponible antes de que inicie el patrón, como consecuencia de la ejecución exitosa del patrón.

Contexto resultante. Describe las condiciones que resultarán una vez que se haya implementado con éxito el patrón: 1) ¿Qué actividades organizacionales o relacionadas con el equipo deben haber ocurrido? 2) ¿Cuál es el estado de salida del proceso? 3) ¿Qué información sobre la ingeniería de software o sobre el proyecto se ha desarrollado?

Patrones relacionados. Proporciona una lista de todos los patrones de proceso directamente relacionados con éste. Puede representarse como una jerarquía o en alguna forma de diagrama. Por ejemplo, el patrón de etapa **Comunicación** incluye los patrones de tarea: **EquipoDelProyecto, LineamientosDeColaboración, DefiniciónDeAlcances, RecabarRequerimientos, DescripciónDeRestricciones** y **CreaciónDeEscenarios**.

Usos y ejemplos conocidos. Indica las instancias específicas en las que es aplicable el patrón. Por ejemplo, **Comunicación** es obligatoria al principio de todo proyecto de software, es recomendable a lo largo del proyecto y de nuevo obligatoria una vez alcanzada la actividad de despliegue.

WebRef

En la dirección **www.ambysoft.com/processPatternsPage.html** se encuentran recursos amplios sobre los patrones de proceso.

Los patrones de proceso dan un mecanismo efectivo para enfrentar problemas asociados con cualquier proceso del software. Los patrones permiten desarrollar una descripción jerárquica del proceso, que comienza en un nivel alto de abstracción (un patrón de fase). Después se mejora la descripción como un conjunto de patrones de etapa que describe las actividades estructurales y se mejora aún más en forma jerárquica en patrones de tarea más detallados para cada patrón de etapa. Una vez desarrollados los patrones de proceso, pueden reutilizarse para la definición de variantes del proceso, es decir, un equipo de software puede definir un modelo de proceso específico con el empleo de los patrones como bloques constituyentes del modelo del proceso.

2.2 Evaluación y mejora del proceso

La existencia de un proceso del software no es garantía de que el software se entregue a tiempo, que satisfaga las necesidades de los consumidores o que tenga las características técnicas que

3 Estos patrones de fase se estudian en la sección 2.3.3.

INFORMACIÓN

Ejemplo de patrón de proceso

El siguiente patrón de proceso abreviado describe un enfoque aplicable en el caso en el que los participantes tienen una idea general de lo que debe hacerse, pero no están seguros de los requerimientos específicos de software.

Nombre del patrón. RequerimientosPocoClaros

Intención. Este patrón describe un enfoque para construir un modelo (un prototipo) que los participantes pueden evaluar en forma iterativa, en un esfuerzo por identificar o solidificar los requerimientos de software.

Tipo. Patrón de fase.

Contexto inicial. Antes de iniciar este patrón deben cumplirse las siguientes condiciones: 1) se ha identificado a los participantes; 2) se ha establecido un modo de comunicación entre los participantes y el equipo de software; 3) los participantes han identificado el problema general de software que se va a resolver; 4) se ha obtenido el entendimiento inicial del alcance del proyecto, los requerimientos básicos del negocio y las restricciones del proyecto.

Problema. Los requerimientos son confusos o inexistentes, pero hay un reconocimiento claro de que existe un problema por resolver y que debe hacerse con una solución de software. Los participantes no están seguros de lo que quieren, es decir, no pueden describir con detalle los requerimientos del software.

Solución. Aquí se presentaría una descripción del proceso prototipo, que se describirá más adelante, en la sección 2.3.3.

Contexto resultante. Los participantes aprueban un prototipo de software que identifica los requerimientos básicos (por ejemplo, modos de interacción, características computacionales, funciones de procesamiento). Después de esto, 1) el prototipo quizá evolucione a través de una serie de incrementos para convertirse en el software de producción, o 2) tal vez se descarte el prototipo y el software de producción se elabore con el empleo de otro proceso de patrón.

Patrones relacionados. Los patrones siguientes están relacionados con este patrón: **ComunicaciónConClientes, DiseñoIterativo, DesarrolloIterativo, EvaluaciónDelCliente, ObtenciónDeRequerimientos.**

Usos y ejemplos conocidos. Cuando los requerimientos sean inciertos, es recomendable hacer prototipos.

PUNTO CLAVE

La evaluación busca entender el estado actual del proceso del software con el objeto de mejorarlo.

conducirán a características de calidad de largo plazo (véanse los capítulos 14 y 16). Los patrones de proceso deben acoplarse con una práctica sólida de ingeniería de software (véase la parte 2 del libro). Además, el proceso en sí puede evaluarse para garantizar que cumple con ciertos criterios de proceso básicos que se haya demostrado que son esenciales para el éxito de la ingeniería de software.[4]

En las últimas décadas se han propuesto numerosos enfoques para la evaluación y mejora de un proceso del software:

¿De qué técnicas formales se dispone para evaluar el proceso del software?

Método de evaluación del estándar CMMI para el proceso de mejora (**SCAMPI,** por sus siglas en inglés): proporciona un modelo de cinco fases para evaluar el proceso: inicio, diagnóstico, establecimiento, actuación y aprendizaje. El método SCAMPI emplea el SEI CMMI como la base de la evaluación [SEI00].

Evaluación basada en CMM para la mejora del proceso interno (**CBA IPI,** por sus siglas en inglés): proporciona una técnica de diagnóstico para evaluar la madurez relativa de una organización de software; usa el SEI CMM como la base de la evaluación [Dun01].

SPICE (ISO/IEC 15504): estándar que define un conjunto de requerimientos para la evaluación del proceso del software. El objetivo del estándar es ayudar a las organizaciones a desarrollar una evaluación objetiva de cualquier proceso del software definido [ISO08].

ISO9001:2000 para software: estándar genérico que se aplica a cualquier organización que desee mejorar la calidad general de los productos, sistemas o servicios que proporciona. Por tanto, el estándar es directamente aplicable a las organizaciones y compañías de software [Ant06].

En el capítulo 30 se presenta un análisis detallado de los métodos de evaluación del software y del proceso de mejora.

Cita:

"Las organizaciones de software tienen deficiencias significativas en su capacidad de capitalizar las experiencias obtenidas de los proyectos terminados."

NASA

4 En la publicación CMMI [CMM07] del SEI, se describen con muchos detalles las características de un proceso del software y los criterios para un proceso exitoso.

2.3 Modelos de proceso prescriptivo

Los modelos de proceso prescriptivo fueron propuestos originalmente para poner orden en el caos del desarrollo de software. La historia indica que estos modelos tradicionales han dado cierta estructura útil al trabajo de ingeniería de software y que constituyen un mapa razonablemente eficaz para los equipos de software. Sin embargo, el trabajo de ingeniería de software y el producto que genera siguen "al borde del caos".

Cita:

"Si el proceso está bien, los resultados cuidarán de sí mismos."

Takashi Osada

En un artículo intrigante sobre la extraña relación entre el orden y el caos en el mundo del software, Nogueira y sus colegas [Nog00] afirman lo siguiente:

> El borde del caos se define como "el estado natural entre el orden y el caos, un compromiso grande entre la estructura y la sorpresa" [Kau95]. El borde del caos se visualiza como un estado inestable y parcialmente estructurado [...] Es inestable debido a que se ve atraído constantemente hacia el caos o hacia el orden absoluto.
>
> Tenemos la tendencia de pensar que el orden es el estado ideal de la naturaleza. Esto podría ser un error [...] las investigaciones apoyan la teoría de que la operación que se aleja del equilibrio genera creatividad, procesos autoorganizados y rendimientos crecientes [Roo96]. El orden absoluto significa ausencia de variabilidad, que podría ser una ventaja en los ambientes impredecibles. El cambio ocurre cuando hay cierta estructura que permita que el cambio pueda organizarse, pero que no sea tan rígida como para que no pueda suceder. Por otro lado, demasiado caos hace imposible la coordinación y la coherencia. La falta de estructura no siempre significa desorden.

PUNTO CLAVE

Los modelos de proceso prescriptivo definen un conjunto prescrito de elementos del proceso y un flujo predecible para el trabajo del proceso.

Las implicaciones filosóficas de este argumento son significativas para la ingeniería de software. Si los modelos de proceso prescriptivo[5] buscan generar estructura y orden, ¿son inapropiados para el mundo del software, que se basa en el cambio? Pero si rechazamos los modelos de proceso tradicional (y el orden que implican) y los reemplazamos con algo menos estructurado, ¿hacemos imposible la coordinación y coherencia en el trabajo de software?

No hay respuestas fáciles para estas preguntas, pero existen alternativas disponibles para los ingenieros de software. En las secciones que siguen se estudia el enfoque de proceso prescriptivo en el que los temas dominantes son el orden y la consistencia del proyecto. El autor los llama "prescriptivos" porque prescriben un conjunto de elementos del proceso: actividades estructurales, acciones de ingeniería de software, tareas, productos del trabajo, aseguramiento de la calidad y mecanismos de control del cambio para cada proyecto. Cada modelo del proceso también prescribe un flujo del proceso (también llamado *flujo de trabajo*), es decir, la manera en la que los elementos del proceso se relacionan entre sí.

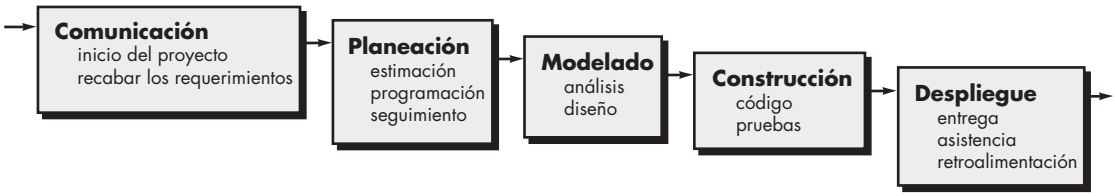
Todos los modelos del proceso del software pueden incluir las actividades estructurales generales descritas en el capítulo 1, pero cada una pone distinto énfasis en ellas y define en forma diferente el flujo de proceso que invoca cada actividad estructural (así como acciones y tareas de ingeniería de software).

2.3.1 Modelo de la cascada

Hay veces en las que los requerimientos para cierto problema se comprenden bien: cuando el trabajo desde la **comunicación** hasta el **despliegue** fluye en forma razonablemente lineal. Esta situación se encuentra en ocasiones cuando deben hacerse adaptaciones o mejoras bien definidas a un sistema ya existente (por ejemplo, una adaptación para software de contabilidad que es obligatorio hacer debido a cambios en las regulaciones gubernamentales). También ocurre en cierto número limitado de nuevos esfuerzos de desarrollo, pero sólo cuando los requerimientos están bien definidos y tienen una estabilidad razonable.

5 Los modelos de proceso prescriptivo en ocasiones son denominados modelos de proceso "tradicional".

FIGURA 2.3 Modelo de la cascada



El *modelo de la cascada*, a veces llamado *ciclo de vida clásico*, sugiere un enfoque sistemático y secuencial[6] para el desarrollo del software, que comienza con la especificación de los requerimientos por parte del cliente y avanza a través de planeación, modelado, construcción y despliegue, para concluir con el apoyo del software terminado (véase la figura 2.3).

PUNTO CLAVE

El modelo en V ilustra la forma en la que se asocian las acciones de verificación y validación con las primeras acciones de ingeniería.

Una variante de la representación del modelo de la cascada se denomina *modelo en V*. En la figura 2.4 se ilustra el modelo en V [Buc99], donde se aprecia la relación entre las acciones para el aseguramiento de la calidad y aquellas asociadas con la comunicación, modelado y construcción temprana. A medida que el equipo de software avanza hacia abajo desde el lado izquierdo de la V, los requerimientos básicos del problema mejoran hacia representaciones técnicas cada vez más detalladas del problema y de su solución. Una vez que se ha generado el código, el equipo sube por el lado derecho de la V, y en esencia ejecuta una serie de pruebas (acciones para asegurar la calidad) que validan cada uno de los modelos creados cuando el equipo fue hacia abajo por el lado izquierdo.[7] En realidad, no hay diferencias fundamentales entre el ciclo de vida clásico y el modelo en V. Este último proporciona una forma de visualizar el modo de aplicación de las acciones de verificación y validación al trabajo de ingeniería inicial.

El modelo de la cascada es el paradigma más antiguo de la ingeniería de software. Sin embargo, en las últimas tres décadas, las críticas hechas al modelo han ocasionado que incluso sus defensores más obstinados cuestionen su eficacia [Han95]. Entre los problemas que en ocasiones surgen al aplicar el modelo de la cascada se encuentran los siguientes:

¿Por qué a veces falla el modelo de la cascada?

1. Es raro que los proyectos reales sigan el flujo secuencial propuesto por el modelo. Aunque el modelo lineal acepta repeticiones, lo hace en forma indirecta. Como resultado, los cambios generan confusión conforme el equipo del proyecto avanza.
2. A menudo, es difícil para el cliente enunciar en forma explícita todos los requerimientos. El modelo de la cascada necesita que se haga y tiene dificultades para aceptar la incertidumbre natural que existe al principio de muchos proyectos.
3. El cliente debe tener paciencia. No se dispondrá de una versión funcional del(de los) programa(s) hasta que el proyecto esté muy avanzado. Un error grande sería desastroso si se detectara hasta revisar el programa en funcionamiento.

Cita:

"Con demasiada frecuencia, el trabajo de software sigue la primera ley del ciclismo: no importa hacia dónde te dirijas, vas hacia arriba y contra el viento."

Anónimo

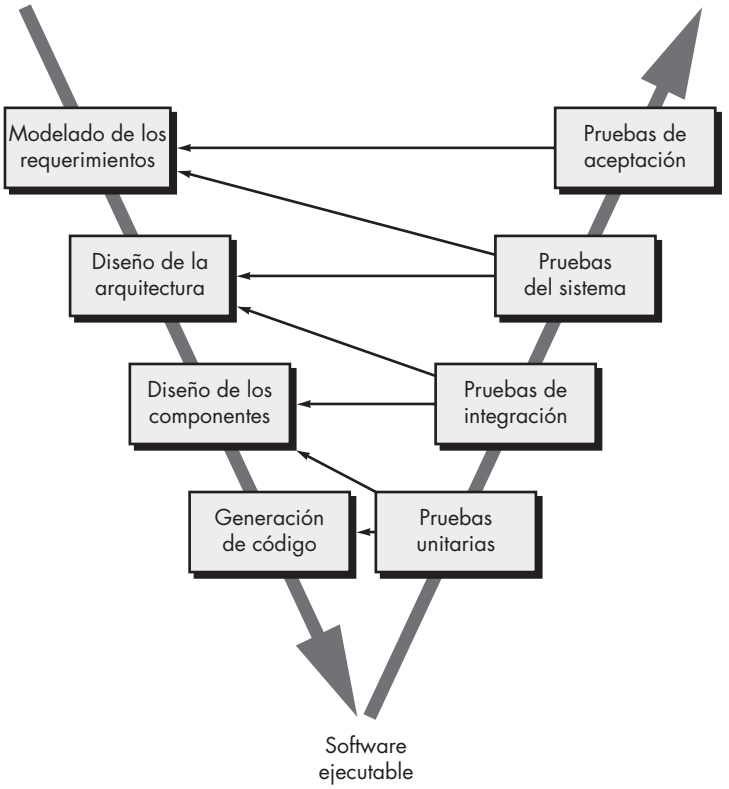
En un análisis interesante de proyectos reales, Bradac [Bra94] encontró que la naturaleza lineal del ciclo de vida clásico llega a "estados de bloqueo" en los que ciertos miembros del equipo de proyecto deben esperar a otros a fin de terminar tareas interdependientes. En realidad, ¡el tiempo de espera llega a superar al dedicado al trabajo productivo! Los estados de bloqueo tienden a ocurrir más al principio y al final de un proceso secuencial lineal.

Hoy en día, el trabajo de software es acelerado y está sujeto a una corriente sin fin de cambios (en las características, funciones y contenido de información). El modelo de la cascada suele ser

6 Aunque el modelo de la cascada propuesto originalmente por Winston Royce [Roy70] prevé los "bucles de retroalimentación", la gran mayoría de organizaciones que aplican este modelo de proceso lo tratan como si fuera estrictamente lineal.

7 En la parte 3 del libro se estudian con detalle las acciones de aseguramiento de la calidad.

FIGURA 2.4
El modelo en V



inapropiado para ese tipo de labor. No obstante, sirve como un modelo de proceso útil en situaciones en las que los requerimientos son fijos y el trabajo avanza en forma lineal hacia el final.

2.3.2 Modelos de proceso incremental

PUNTO CLAVE

El modelo incremental ejecuta una serie de avances, llamados incrementos, que en forma progresiva dan más funcionalidad al cliente conforme se le entrega cada incremento.

Hay muchas situaciones en las que los requerimientos iniciales del software están razonablemente bien definidos, pero el alcance general del esfuerzo de desarrollo imposibilita un proceso lineal. Además, tal vez haya una necesidad imperiosa de dar rápidamente cierta funcionalidad limitada de software a los usuarios y aumentarla en las entregas posteriores de software. En tales casos, se elige un modelo de proceso diseñado para producir el software en incrementos.

El modelo *incremental* combina elementos de los flujos de proceso lineal y paralelo estudiados en la sección 2.1. En relación con la figura 2.5, el modelo incremental aplica secuencias lineales en forma escalonada a medida que avanza el calendario de actividades. Cada secuencia lineal produce "incrementos" de software susceptibles de entregarse [McD93] de manera parecida a los incrementos producidos en un flujo de proceso evolutivo (sección 2.3.3).

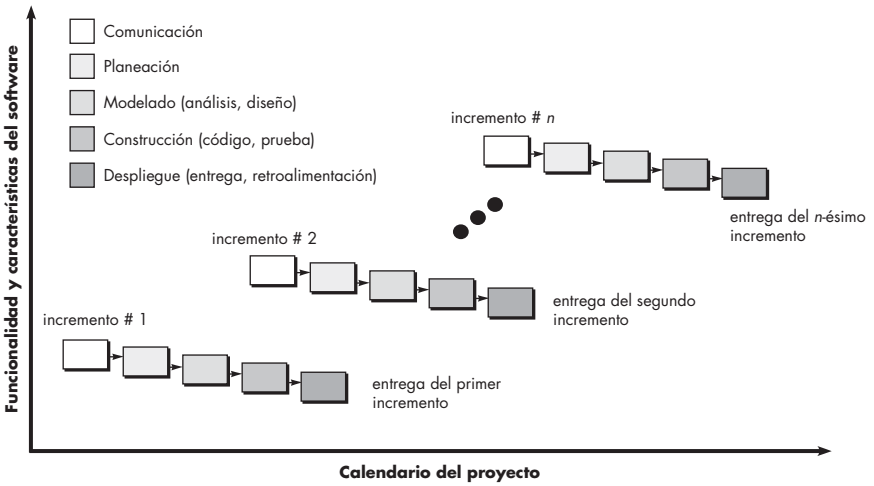
Por ejemplo, un software para procesar textos que se elabore con el paradigma incremental quizá entregue en el primer incremento las funciones básicas de administración de archivos, edición y producción del documento; en el segundo dará herramientas más sofisticadas de edición y producción de documentos; en el tercero habrá separación de palabras y revisión de la ortografía; y en el cuarto se proporcionará la capacidad para dar formato avanzado a las páginas. Debe observarse que el flujo de proceso para cualquier incremento puede incorporar el paradigma del prototipo.

CONSEJO

Su cliente solicita la entrega para una fecha que es imposible de cumplir. Sugiera entregar uno o más incrementos en la fecha que pide, y el resto del software (incrementos adicionales) en un momento posterior.

Cuando se utiliza un modelo incremental, es frecuente que el primer incremento sea el *producto fundamental*. Es decir, se abordan los requerimientos básicos, pero no se proporcionan muchas características suplementarias (algunas conocidas y otras no). El cliente usa el producto fundamental (o lo somete a una evaluación detallada). Como resultado del uso y/o evaluación,

FIGURA 2.5
El modelo incremental



se desarrolla un plan para el incremento que sigue. El plan incluye la modificación del producto fundamental para cumplir mejor las necesidades del cliente, así como la entrega de características adicionales y más funcionalidad. Este proceso se repite después de entregar cada incremento, hasta terminar el producto final.

El modelo de proceso incremental se centra en que en cada incremento se entrega un producto que ya opera. Los primeros incrementos son versiones desnudas del producto final, pero proporcionan capacidad que sirve al usuario y también le dan una plataforma de evaluación.[8]

El desarrollo incremental es útil en particular cuando no se dispone de personal para la implementación completa del proyecto en el plazo establecido por el negocio. Los primeros incrementos se desarrollan con pocos trabajadores. Si el producto básico es bien recibido, entonces se agrega más personal (si se requiere) para que labore en el siguiente incremento. Además, los incrementos se planean para administrar riesgos técnicos. Por ejemplo, un sistema grande tal vez requiera que se disponga de hardware nuevo que se encuentre en desarrollo y cuya fecha de entrega sea incierta. En este caso, tal vez sea posible planear los primeros incrementos de forma que eviten el uso de dicho hardware, y así proporcionar una funcionalidad parcial a los usuarios finales sin un retraso importante.

2.3.3 Modelos de proceso evolutivo

PUNTO CLAVE
El modelo del proceso evolutivo genera en cada iteración una versión final cada vez más completa del software.

El software, como todos los sistemas complejos, evoluciona en el tiempo. Es frecuente que los requerimientos del negocio y del producto cambien conforme avanza el desarrollo, lo que hace que no sea realista trazar una trayectoria rectilínea hacia el producto final; los plazos apretados del mercado hacen que sea imposible la terminación de un software perfecto, pero debe lanzarse una versión limitada a fin de aliviar la presión de la competencia o del negocio; se comprende bien el conjunto de requerimientos o el producto básico, pero los detalles del producto o extensiones del sistema aún están por definirse. En estas situaciones y otras parecidas se necesita un modelo de proceso diseñado explícitamente para adaptarse a un producto que evoluciona con el tiempo.

Los modelos evolutivos son iterativos. Se caracterizan por la manera en la que permiten desarrollar versiones cada vez más completas del software. En los párrafos que siguen se presentan dos modelos comunes de proceso evolutivo.

8 Es importante observar que para todos los modelos de proceso "ágiles" que se estudian en el capítulo 3 también se usa la filosofía incremental.

Cita:

"Planee para lanzar uno. De todos modos hará eso. Su única elección es si tratará de vender a sus clientes lo que lanzó."

Frederick P. Brooks

Hacer prototipos. Es frecuente que un cliente defina un conjunto de objetivos generales para el software, pero que no identifique los requerimientos detallados para las funciones y características. En otros casos, el desarrollador tal vez no esté seguro de la eficiencia de un algoritmo, de la adaptabilidad de un sistema operativo o de la forma que debe adoptar la interacción entre el humano y la máquina. En estas situaciones, y muchas otras, el *paradigma de hacer prototipos* tal vez ofrezca el mejor enfoque.

Aunque es posible hacer prototipos como un modelo de proceso aislado, es más común usarlo como una técnica que puede implementarse en el contexto de cualquiera de los modelos de proceso descritos en este capítulo. Sin importar la manera en la que se aplique, el paradigma de hacer prototipos le ayudará a usted y a otros participantes a mejorar la comprensión de lo que hay que elaborar cuando los requerimientos no están claros.

CONSEJO

Cuando su cliente tiene una necesidad legítima, pero ignora los detalles, como primer paso desarrolle un prototipo.

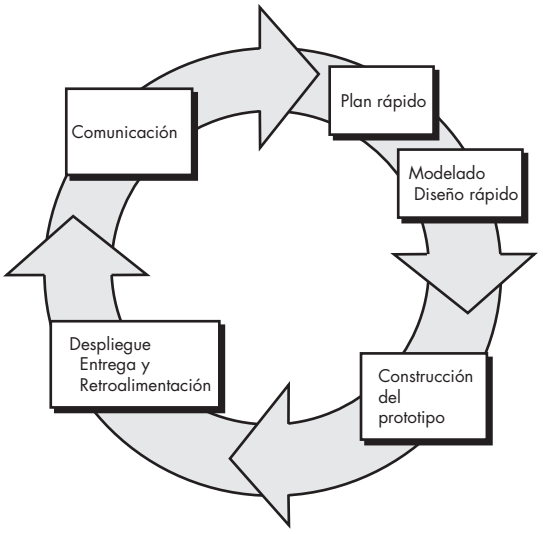
El paradigma de hacer prototipos (véase la figura 2.6) comienza con comunicación. Usted se reúne con otros participantes para definir los objetivos generales del software, identifica cualesquiera requerimientos que conozca y detecta las áreas en las que es imprescindible una mayor definición. Se planea rápidamente una iteración para hacer el prototipo, y se lleva a cabo el modelado (en forma de un "diseño rápido"). Éste se centra en la representación de aquellos aspectos del software que serán visibles para los usuarios finales (por ejemplo, disposición de la interfaz humana o formatos de la pantalla de salida). El diseño rápido lleva a la construcción de un prototipo. Éste se entrega y es evaluado por los participantes, que dan retroalimentación para mejorar los requerimientos. La iteración ocurre a medida de que el prototipo es afinado para satisfacer las necesidades de distintos participantes, y al mismo tiempo le permite a usted entender mejor lo que se necesita hacer.

El ideal es que el prototipo sirva como mecanismo para identificar los requerimientos del software. Si va a construirse un prototipo, pueden utilizarse fragmentos de programas existentes o aplicar herramientas (por ejemplo, generadores de reportes y administradores de ventanas) que permitan generar rápidamente programas que funcionen.

Pero, ¿qué hacer con el prototipo cuando ya sirvió para el propósito descrito? Brooks [Bro95] da una respuesta:

> En la mayoría de proyectos es raro que el primer sistema elaborado sea utilizable. Tal vez sea muy lento, muy grande, difícil de usar o todo a la vez. No hay más alternativa que comenzar de nuevo, con más inteligencia, y construir una versión rediseñada en la que se resuelvan los problemas.

FIGURA 2.6
El paradigma de hacer prototipos



El prototipo sirve como "el primer sistema". Lo que Brooks recomienda es desecharlo. Pero esto quizá sea un punto de vista idealizado. Aunque algunos prototipos se construyen para ser "desechables", otros son evolutivos; es decir, poco a poco se transforman en el sistema real.

Tanto a los participantes como a los ingenieros de software les gusta el paradigma de hacer prototipos. Los usuarios adquieren la sensación del sistema real, y los desarrolladores logran construir algo de inmediato. No obstante, hacer prototipos llega a ser problemático por las siguientes razones:

Resista la presión para convertir un prototipo burdo en un producto terminado. Como resultado de ello, casi siempre disminuye la calidad.

1. Los participantes ven lo que parece ser una versión funcional del software, sin darse cuenta de que el prototipo se obtuvo de manera caprichosa; no perciben que en la prisa por hacer que funcionara, usted no consideró la calidad general del software o la facilidad de darle mantenimiento a largo plazo. Cuando se les informa que el producto debe rehacerse a fin de obtener altos niveles de calidad, los participantes gritan que es usted un tonto y piden "unos cuantos arreglos" para hacer del prototipo un producto funcional. Con demasiada frecuencia, el gerente de desarrollo del software cede.
2. Como ingeniero de software, es frecuente que llegue a compromisos respecto de la implementación a fin de hacer que el prototipo funcione rápido. Quizá utilice un sistema operativo inapropiado, o un lenguaje de programación tan sólo porque cuenta con él y lo conoce; tal vez implementó un algoritmo ineficiente sólo para demostrar capacidad. Después de un tiempo, quizá se sienta cómodo con esas elecciones y olvide todas las razones por las que eran inadecuadas. La elección de algo menos que lo ideal ahora ha pasado a formar parte del sistema.

Aunque puede haber problemas, hacer prototipos es un paradigma eficaz para la ingeniería de software. La clave es definir desde el principio las reglas del juego; es decir, todos los participantes deben estar de acuerdo en que el prototipo sirva como el mecanismo para definir los requerimientos. Después se descartará (al menos en parte) y se hará la ingeniería del software real con la mirada puesta en la calidad.

CasaSegura

Selección de un modelo de proceso, parte 1

La escena: Sala de juntas del grupo de ingeniería de software de CPI Corporation, compañía (ficticia) que manufactura artículos de consumo para el hogar y para uso comercial.

Participantes: Lee Warren, gerente de ingeniería; Doug Miller, gerente de ingeniería de software; Jamie Lazar, miembro del equipo de software; Vinod Raman, miembro del equipo de software; y Ed Robbins, miembro del equipo de software.

La conversación:

Lee: Recapitulemos. He dedicado algún tiempo al análisis de la línea de productos *CasaSegura*, según la vemos hasta el momento. No hay duda de que hemos efectuado mucho trabajo tan sólo para definir el concepto, pero me gustaría que ustedes comenzaran a pensar en cómo van a enfocar la parte del software de este proyecto.

Doug: Pareciera que en el pasado hemos estado muy desorganizados en nuestro enfoque del software.

Ed: No sé, Doug, siempre sacamos el producto.

Doug: Es cierto, pero no sin muchos sobresaltos, y este proyecto parece más grande y complejo que cualquier cosa que hayamos hecho antes.

Jamie: No parece tan mal, pero estoy de acuerdo… nuestro enfoque *ad hoc* de los proyectos anteriores no funcionará en éste, en particular si tenemos una fecha de entrega muy apretada.

Doug (sonríe): Quiero ser un poco más profesional en nuestro enfoque. La semana pasada asistí a un curso breve y aprendí mucho sobre ingeniería de software… algo bueno. Aquí necesitamos un proceso.

Jamie (con el ceño fruncido): Mi trabajo es producir programas de computadora, no papel.

Doug: Den una oportunidad antes de ser tan negativos conmigo. Lo que quiero decir es esto: [Doug pasa a describir la estructura del proceso vista en este capítulo y los modelos de proceso prescriptivo presentados hasta el momento.]

Doug: De cualquier forma, parece que un modelo lineal no es para nosotros… pues supone que conocemos todos los requerimientos y, conociendo esta empresa, eso no parece probable.

Vinod: Sí, y parece demasiado orientado a las tecnologías de información… tal vez sea bueno para hacer un sistema de control de inventarios o algo así, pero no parece bueno para *CasaSegura*.

Doug: Estoy de acuerdo.

Ed: Ese enfoque de hacer prototipos parece bueno. En todo caso, se asemeja mucho a lo que hacemos aquí.

Vinod: Eso es un problema. Me preocupa que no nos dé suficiente estructura.

Doug: No te preocupes. Tenemos muchas opciones más, y quisiera que ustedes, muchachos, elijan la que sea mejor para el equipo y para el proyecto.

El modelo espiral. Propuesto en primer lugar por Barry Boehm [Boe88], el *modelo espiral* es un modelo evolutivo del proceso del software y se acopla con la naturaleza iterativa de hacer prototipos con los aspectos controlados y sistémicos del modelo de cascada. Tiene el potencial para hacer un desarrollo rápido de versiones cada vez más completas. Boehm [Boe01a] describe el modelo del modo siguiente:

> El modelo de desarrollo espiral es un generador de *modelo de proceso* impulsado por el *riesgo*, que se usa para guiar la ingeniería concurrente con participantes múltiples de sistemas intensivos en software. Tiene dos características distintivas principales. La primera es el enfoque *cíclico* para el crecimiento incremental del grado de definición de un sistema y su implementación, mientras que disminuye su grado de riesgo. La otra es un conjunto de *puntos de referencia de anclaje puntual* para asegurar el compromiso del participante con soluciones factibles y mutuamente satisfactorias.

PUNTO CLAVE

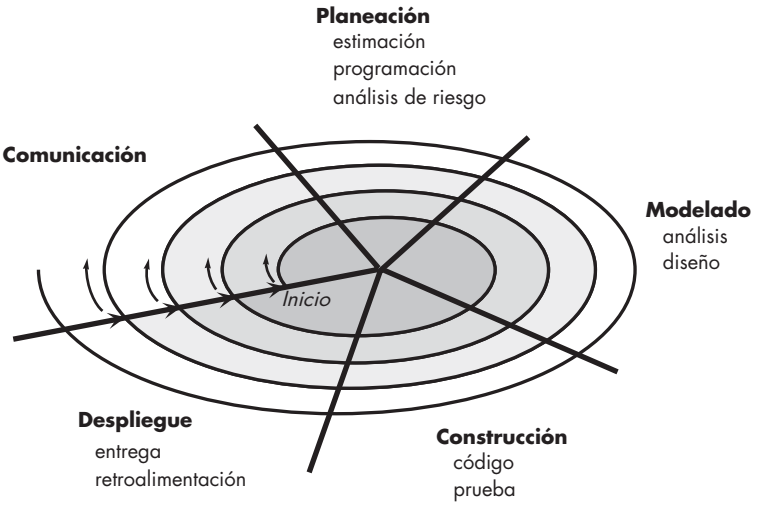
El modelo en espiral se adapta para emplearse a lo largo de todo el ciclo de vida de una aplicación, desde el desarrollo del concepto hasta el mantenimiento.

Con el empleo del modelo espiral, el software se desarrolla en una serie de entregas evolutivas. Durante las primeras iteraciones, lo que se entrega puede ser un modelo o prototipo. En las iteraciones posteriores se producen versiones cada vez más completas del sistema cuya ingeniería se está haciendo.

Un modelo en espiral es dividido por el equipo de software en un conjunto de actividades estructurales. Para fines ilustrativos, se utilizan las actividades estructurales generales ya analizadas.[9] Cada una de ellas representa un segmento de la trayectoria espiral ilustrada en la figura 2.7. Al comenzar el proceso evolutivo, el equipo de software realiza actividades implícitas en un

FIGURA 2.7

Modelo de espiral común



9 El modelo espiral estudiado en esta sección es una variante del propuesto por Boehm. Para más información acerca del modelo espiral original, consulte [Boe88]. En [Boe98] se encuentra un análisis más reciente del modelo espiral del mismo autor.

circuito alrededor de la espiral en el sentido horario, partiendo del centro. El riesgo se considera conforme se desarrolla cada revolución (capítulo 28). En cada paso evolutivo se marcan *puntos de referencia puntuales*: combinación de productos del trabajo y condiciones que se encuentran a lo largo de la trayectoria de la espiral.

El primer circuito alrededor de la espiral da como resultado el desarrollo de una especificación del producto; las vueltas sucesivas se usan para desarrollar un prototipo y, luego, versiones cada vez más sofisticadas del software. Cada paso por la región de planeación da como resultado ajustes en el plan del proyecto. El costo y la programación de actividades se ajustan con base en la retroalimentación obtenida del cliente después de la entrega. Además, el gerente del proyecto ajusta el número planeado de iteraciones que se requieren para terminar el software.

WebRef

En la dirección **www.sei.cmu.edu/publications/documents/00.reports/00sr008.html** se encuentra información útil sobre el modelo espiral.

A diferencia de otros modelos del proceso que finalizan cuando se entrega el software, el modelo espiral puede adaptarse para aplicarse a lo largo de toda la vida del software de cómputo. Entonces, el primer circuito alrededor de la espiral quizá represente un "proyecto de desarrollo del concepto" que comienza en el centro de la espiral y continúa por iteraciones múltiples[10] hasta que queda terminado el desarrollo del concepto. Si el concepto va a desarrollarse en un producto real, el proceso sigue hacia fuera de la espiral y comienza un "proyecto de desarrollo de producto nuevo". El nuevo producto evolucionará a través de cierto número de iteraciones alrededor de la espiral. Más adelante puede usarse un circuito alrededor de la espiral para que represente un "proyecto de mejora del producto". En esencia, la espiral, cuando se caracteriza de este modo, sigue operativa hasta que el software se retira. Hay ocasiones en las que el proceso está inmóvil, pero siempre que se inicia un cambio comienza en el punto de entrada apropiado (por ejemplo, mejora del producto).

Si su administración pide un desarrollo apegado al presupuesto (mala idea, por lo general), la espiral se convierte en un problema. El costo se revisa y modifica cada vez que se termina un circuito.

El modelo espiral es un enfoque realista para el desarrollo de sistemas y de software a gran escala. Como el software evoluciona a medida que el proceso avanza, el desarrollador y cliente comprenden y reaccionan mejor ante los riesgos en cada nivel de evolución. El modelo espiral usa los prototipos como mecanismo de reducción de riesgos, pero, más importante, permite aplicar el enfoque de hacer prototipos en cualquier etapa de la evolución del producto. Mantiene el enfoque de escalón sistemático sugerido por el ciclo de vida clásico, pero lo incorpora en una estructura iterativa que refleja al mundo real en una forma más realista. El modelo espiral demanda una consideración directa de los riesgos técnicos en todas las etapas del proyecto y, si se aplica de manera apropiada, debe reducir los riesgos antes de que se vuelvan un problema.

Cita:

"Sólo voy aquí y sólo el mañana me guía."

Dave Matthews Band

Pero, como otros paradigmas, el modelo espiral no es una panacea. Es difícil convencer a los clientes (en particular en situaciones bajo contrato) de que el enfoque evolutivo es controlable. Demanda mucha experiencia en la evaluación del riesgo y se basa en ella para llegar al éxito. No hay duda de que habrá problemas si algún riesgo importante no se descubre y administra.

2.3.4 Modelos concurrentes

Con frecuencia, el modelo concurrente es más apropiado para proyectos de ingeniería de productos en los que se involucran varios equipos de trabajo.

El *modelo de desarrollo concurrente*, en ocasiones llamado *ingeniería concurrente*, permite que un equipo de software represente elementos iterativos y concurrentes de cualquiera de los modelos de proceso descritos en este capítulo. Por ejemplo, la actividad de modelado definida para el modelo espiral se logra por medio de invocar una o más de las siguientes acciones de software: hacer prototipos, análisis y diseño.[11]

La figura 2.8 muestra la representación esquemática de una actividad de ingeniería de software dentro de la actividad de modelado con el uso del enfoque de modelado concurrente. La

10 Las flechas que apuntan hacia dentro a lo largo del eje que separa la región del **despliegue** de la de **comunicación** indican un potencial para la iteración local a lo largo de la misma trayectoria espiral.

11 Debe observarse que el análisis y diseño son tareas complejas que requieren mucho análisis. La parte 2 de este libro considera en detalle dichos temas.

CasaSegura

Selección de un modelo de proceso, parte 2

La escena: Sala de juntas del grupo de ingeniería de software de CPI Corporation, compañía que manufactura productos de consumo para uso doméstico y comercial.

Participantes: Lee Warren, gerente de ingeniería; Doug Miller, gerente de ingeniería de software; Vinod y Jamie, miembros del equipo de ingeniería de software.

La conversación: [Doug describe las opciones de proceso evolutivo.]

Jamie: Ahora me doy cuenta de algo. El enfoque incremental tiene sentido, y en verdad me gusta el flujo del modelo en espiral. Es bastante realista.

Vinod: De acuerdo. Entregamos un incremento, aprendemos de la retroalimentación del cliente, volvemos a planear y luego entregamos otro incremento. También se ajusta a la naturaleza del producto. Podemos lanzar con rapidez algo al mercado y luego agregar funcionalidad con cada versión, digo… con cada incremento.

Lee: Un momento. Doug, ¿dijiste que volveríamos a hacer el plan a cada vuelta de la espiral? Eso no es nada bueno; necesitamos un plan, un programa de actividades y apegarnos a ellos.

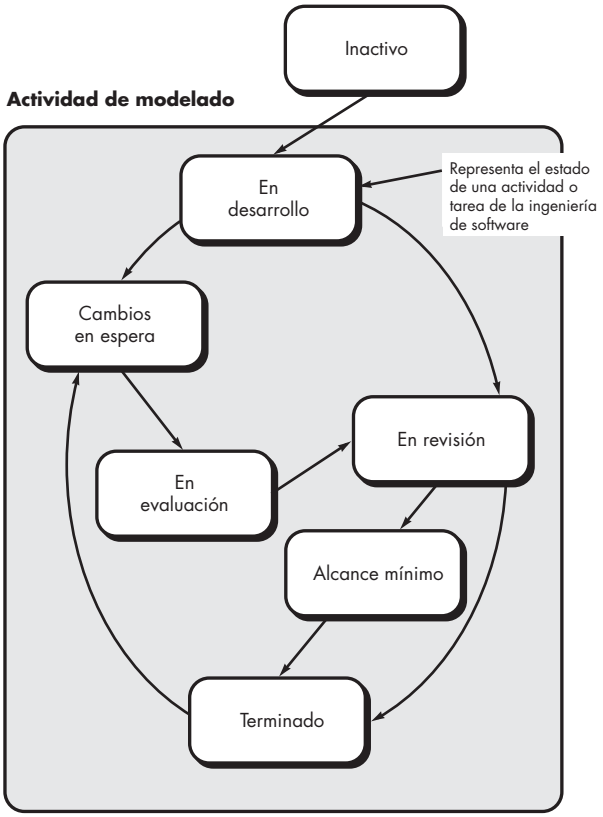
Doug: Ésa es la vieja escuela, Lee. Como dijeron los chicos, tenemos que hacerlo apegado a la realidad. Afirmo que es mejor afinar el plan a medida de que aprendamos más y conforme se soliciten cambios. Eso es más realista. ¿Qué sentido tiene un plan si no refleja la realidad?

Lee (con el ceño fruncido): Supongo, pero… a la alta dirección no le va a gustar… quieren un plan fijo.

Doug (sonriente): Entonces tendrás que reeducarlos, amigo.

actividad **—modelado—** puede estar en cualquiera de los estados[12] mencionados en un momento dado. En forma similar, es posible representar de manera análoga otras actividades, acciones o tareas (por ejemplo, **comunicación** o **construcción**). Todas las actividades de ingeniería de software existen de manera concurrente, pero se hallan en diferentes estados.

Figura 2.8 Un elemento del modelo de proceso concurrente



12 Un *estado* es algún modo de comportamiento observable externamente.

Por ejemplo, la actividad de comunicación (no se muestra en la figura) termina su primera iteración al principio de un proyecto y existe en el estado de **cambios en espera**. La actividad de modelado (que existía en estado **inactivo** mientras concluía la comunicación inicial, ahora hace una transición al estado **en desarrollo**. Sin embargo, si el cliente indica que deben hacerse cambios en los requerimientos, la actividad de modelado pasa del estado **en desarrollo** al de **cambios en espera**.

El modelado concurrente define una serie de eventos que desencadenan transiciones de un estado a otro para cada una de las actividades, acciones o tareas de la ingeniería de software. Por ejemplo, durante las primeras etapas del diseño (acción importante de la ingeniería de software que ocurre durante la actividad de modelado), no se detecta una inconsistencia en el modelo de requerimientos. Esto genera el evento *corrección del modelo de análisis*, que disparará la acción de análisis de requerimientos del estado **terminado** al de **cambios en espera**.

Cita:

"Todo proceso en su organización tiene un cliente, y un proceso sin cliente no tiene propósito."

V. Daniel Hunt

El modelado concurrente es aplicable a todos los tipos de desarrollo de software y proporciona un panorama apropiado del estado actual del proyecto. En lugar de confinar las actividades, acciones y tareas de la ingeniería de software a una secuencia de eventos, define una red del proceso. Cada actividad, acción o tarea de la red existe simultáneamente con otras actividades, acciones o tareas. Los eventos generados en cierto punto de la red del proceso desencadenan transiciones entre los estados.

2.3.5 Una última palabra acerca de los procesos evolutivos

Ya se dijo que el software de cómputo moderno se caracteriza por el cambio continuo, por tiempos de entrega muy apretados y por una necesidad apremiante de la satisfacción del cliente o usuario. En muchos casos, el tiempo para llegar al mercado es el requerimiento administrativo más importante. Si se pierde un nicho de mercado, todo el proyecto de software podría carecer de sentido.[13]

Los modelos de proceso evolutivo fueron concebidos para cumplir esos requisitos, pero, aun así, como clase general de modelos de proceso tienen demasiadas debilidades, que fueron resumidas por Nogueira y sus colegas [Nog00]:

> A pesar de los beneficios incuestionables de los procesos evolutivos de software, existen algunas preocupaciones. La primera es que hacer prototipos (y otros procesos evolutivos más sofisticados) plantea un problema para la planeación del proyecto debido a la incertidumbre en el número de ciclos que se requieren para elaborar el producto. La mayor parte de técnicas de administración y estimación de proyectos se basa en un planteamiento lineal de las actividades, por lo que no se ajustan por completo.
>
> En segundo lugar, los procesos evolutivos de software no establecen la velocidad máxima de la evolución. Si las evoluciones ocurren demasiado rápido, sin un periodo de relajamiento, es seguro que el proceso se volverá un caos. Por otro lado, si la velocidad es muy lenta, se verá perjudicada la productividad…
>
> En tercer lugar, los procesos de software deben centrarse en la flexibilidad y capacidad de extensión en lugar de en la alta calidad. Esto suena preocupante. Sin embargo, debe darse prioridad a la velocidad del desarrollo con el enfoque de cero defectos. Extender el desarrollo a fin de lograr alta calidad podría dar como resultado la entrega tardía del producto, cuando haya desaparecido el nicho de oportunidad. Este cambio de paradigma es impuesto por la competencia al borde del caos.

En realidad, sí parece preocupante un proceso del software que se centre en la flexibilidad, expansión y velocidad del desarrollo por encima de la calidad. No obstante, esta idea ha sido propuesta por varios expertos en ingeniería de software muy respetados ([You95], [Bac97]).

---

13 Sin embargo, es importante notar que ser el primero en llegar al mercado no es garantía de éxito. En realidad, muchos productos de software muy exitosos han llegado en segundo o hasta en tercer lugar al mercado (aprenden de los errores de sus antecesores).

El objetivo de los modelos evolutivos es desarrollar software de alta calidad[14] en forma iterativa o incremental. Sin embargo, es posible usar un proceso evolutivo para hacer énfasis en la flexibilidad, expansibilidad y velocidad del desarrollo. El reto para los equipos de software y sus administradores es establecer un balance apropiado entre estos parámetros críticos del proyecto y el producto, y la satisfacción del cliente (árbitro definitivo de la calidad del software).

2.4 MODELOS DE PROCESO ESPECIALIZADO

Los modelos de proceso especializado tienen muchas de las características de uno o más de los modelos tradicionales que se presentaron en las secciones anteriores. Sin embargo, dichos modelos tienden a aplicarse cuando se elige un enfoque de ingeniería de software especializado o definido muy específicamente.[15]

2.4.1 Desarrollo basado en componentes

WebRef

En la dirección **www.cbd-hq.com** hay información útil sobre el desarrollo basado en componentes.

Los componentes comerciales de software general (COTS, por sus siglas en inglés), desarrollados por vendedores que los ofrecen como productos, brindan una funcionalidad que se persigue con interfaces bien definidas que permiten que el componente se integre en el software que se va a construir. El *modelo de desarrollo basado en componentes* incorpora muchas de las características del modelo espiral. Es de naturaleza evolutiva [Nie92] y demanda un enfoque iterativo para la creación de software. Sin embargo, el modelo de desarrollo basado en componentes construye aplicaciones a partir de fragmentos de software prefabricados.

Las actividades de modelado y construcción comienzan con la identificación de candidatos de componentes. Éstos pueden diseñarse como módulos de software convencional o clases orientadas a objetos o paquetes[16] de clases. Sin importar la tecnología usada para crear los componentes, el modelo de desarrollo basado en componentes incorpora las etapas siguientes (se implementan con el uso de un enfoque evolutivo):

1. Se investigan y evalúan, para el tipo de aplicación de que se trate, productos disponibles basados en componentes.
2. Se consideran los aspectos de integración de los componentes.
3. Se diseña una arquitectura del software para que reciba los componentes.
4. Se integran los componentes en la arquitectura.
5. Se efectúan pruebas exhaustivas para asegurar la funcionalidad apropiada.

El modelo del desarrollo basado en componentes lleva a la reutilización del software, y eso da a los ingenieros de software varios beneficios en cuanto a la mensurabilidad. Si la reutilización de componentes se vuelve parte de la cultura, el equipo de ingeniería de software tiene la posibilidad tanto de reducir el ciclo de tiempo del desarrollo como el costo del proyecto. En el capítulo 10 se analiza con más detalle el desarrollo basado en componentes.

---

14 En este contexto, la calidad del software se define con mucha amplitud para que agrupe no sólo la satisfacción del cliente sino también varios criterios técnicos que se estudian en los capítulos 14 y 16.

15 En ciertos casos, los modelos de proceso especializado pueden caracterizarse mejor como un conjunto de técnicas o "metodología" para alcanzar una meta específica de desarrollo de software. No obstante, implican un proceso.

16 En el apéndice 2 se estudian los conceptos orientados a objetos, y se utilizan en toda la parte 2 del libro. En este contexto, una clase agrupa un conjunto de datos y los procedimientos para procesarlos. Un paquete de clases es un conjunto de clases relacionadas que funcionan juntas para alcanzar cierto resultado final.

2.4.2 El modelo de métodos formales

El *modelo de métodos formales* agrupa actividades que llevan a la especificación matemática formal del software de cómputo. Los métodos formales permiten especificar, desarrollar y verificar un sistema basado en computadora por medio del empleo de una notación matemática rigurosa. Ciertas organizaciones de desarrollo de software aplican una variante de este enfoque, que se denomina *ingeniería de software de quirófano* [Mil87, Dye92].

Cuando durante el desarrollo se usan métodos formales (capítulo 21), se obtiene un mecanismo para eliminar muchos de los problemas difíciles de vencer con otros paradigmas de la ingeniería de software. Lo ambiguo, incompleto e inconsistente se descubre y corrige con más facilidad, no a través de una revisión *ad hoc* sino con la aplicación de análisis matemático. Si durante el diseño se emplean métodos formales, éstos sirven como base para la verificación del programa, y así permiten descubrir y corregir errores que de otro modo no serían detectados.

Aunque el modelo de los métodos formales no es el más seguido, promete un software libre de defectos. Sin embargo, se han expresado preocupaciones acerca de su aplicabilidad en un ambiente de negocios:

? Si con los métodos formales puede demostrarse lo correcto de un software, ¿por qué no son ampliamente utilizados?

- El desarrollo de modelos formales consume mucho tiempo y es caro.
- Debido a que pocos desarrolladores de software tienen la formación necesaria para aplicar métodos formales, se requiere mucha capacitación.
- Es difícil utilizar los modelos como mecanismo de comunicación para clientes sin complejidad técnica.

A pesar de estas preocupaciones, el enfoque de los métodos formales ha ganado partidarios entre los desarrolladores que deben construir software de primera calidad en seguridad (por ejemplo, control electrónico de aeronaves y equipos médicos), y entre los desarrolladores que sufrirían graves pérdidas económicas si ocurrieran errores en su software.

2.4.3 Desarrollo de software orientado a aspectos

WebRef
Existen muchos recursos e información sobre SOA en la dirección: **aosd.net**

Sin importar el proceso del software que se elija, los constructores de software complejo implementan de manera invariable un conjunto de características, funciones y contenido de información localizados. Estas características localizadas del software se modelan como componentes (clases orientadas a objetos) y luego se construyen dentro del contexto de una arquitectura de sistemas. A medida que los sistemas modernos basados en computadora se hacen más sofisticados (y complejos), ciertas *preocupaciones* —propiedades que requiere el cliente o áreas de interés técnico— se extienden a toda la arquitectura. Algunas de ellas son las propiedades de alto nivel de un sistema (por ejemplo, seguridad y tolerancia a fallas). Otras afectan a funciones (aplicación de las reglas de negocios), mientras que otras más son sistémicas (sincronización de la tarea o administración de la memoria).

PUNTO CLAVE
El DSOA define "aspectos" que expresan preocupaciones del cliente que afectan múltiples funciones, características e información del sistema.

Cuando las preocupaciones afectan múltiples funciones, características e información del sistema, es frecuente que se les llame *preocupaciones globales*. Los *requerimientos del aspecto* definen aquellas preocupaciones globales que tienen algún efecto a través de la arquitectura del software. El *desarrollo de software orientado a aspectos* (DSOA), conocido también como *programación orientada a aspectos* (POA), es un paradigma de ingeniería de software relativamente nuevo que proporciona un proceso y enfoque metodológico para definir, especificar, diseñar y construir *aspectos*: "mecanismos más allá de subrutinas y herencia para localizar la expresión de una preocupación global" [Elr01].

Grundy [Gru02] analiza con más profundidad los aspectos en el contexto de lo que denomina *ingeniería de componentes orientada a aspectos* (ICOA):

> La ICOA usa el concepto de rebanadas horizontales a través de componentes de software descompuestos verticalmente, llamados "aspectos", para caracterizar las propiedades globales funcionales y

no funcionales de los componentes. Los aspectos comunes y sistémicos incluyen interfaces de usuario, trabajo en colaboración, distribución, persistencia, administración de la memoria, procesamiento de las transacciones, seguridad, integridad, etc. Los componentes pueden proveer o requerir uno o más "detalles de aspectos" en relación con un aspecto particular, como un mecanismo de visión, alcance extensible y clase de interfaz (aspectos de la interfaz de usuario); generación de eventos, transporte y recepción (aspectos de distribución); almacenamiento, recuperación e indización de datos (aspectos de persistencia); autenticación, encriptación y derechos de acceso (aspectos de seguridad); descomposición de las transacciones, control de concurrencia y estrategia de registro (aspectos de las transacciones), entre otros. Cada detalle del aspecto tiene cierto número de propiedades relacionadas con las características funcionales o no del detalle del aspecto.

Aún no madura un proceso distinto orientado a aspectos. Sin embargo, es probable que un proceso así adopte características tanto de los modelos de proceso evolutivo como concurrente. El modelo evolutivo es apropiado en tanto los aspectos se identifican y después se construyen. La naturaleza paralela del desarrollo concurrente es esencial porque la ingeniería de aspectos se hace en forma independiente de los componentes de software localizados; aun así, los aspectos tienen un efecto directo sobre éstos. De esta forma, es esencial disponer de comunicación asincrónica entre las actividades de proceso del software aplicadas a la ingeniería, y la construcción de los aspectos y componentes.

El análisis detallado del desarrollo de software orientado al aspecto se deja a libros especializados en el tema. Si el lector tiene interés en profundizar, se le invita a consultar [Saf08], [Cla05], [Jac04] y [Gra03].

HERRAMIENTAS DE SOFTWARE

Administración del proceso

Objetivo: Ayudar a la definición, ejecución y administración de modelos de proceso prescriptivo.

Mecánica: Las herramientas de administración del proceso permiten que una organización o equipo de software defina un modelo completo del proceso (actividades estructurales, acciones, tareas, aseguramiento de la calidad, puntos de revisión, referencias y productos del trabajo). Además, las herramientas proporcionan un mapa conforme los ingenieros de software realizan el trabajo técnico, y una plantilla para los gerentes que deben dar seguimiento y controlar el proceso del software.

Herramientas representativas:[17]

GDPA, grupo de herramientas de investigación de definición del proceso, desarrollada por la Universidad de Bremen, en Alemania (**www.informatik.uni-bremen.de/uniform/gdpa/home.htm**), proporciona una amplia variedad de funciones para modelar y administrar procesos.

SpeeDev, desarrollada por SpeeDev Corporation (**www.speedev.com**), incluye un conjunto de herramientas para la definición del proceso, administración de los requerimientos, resolución de problemas, y planeación y seguimiento del proyecto.

ProVision BPMx, desarrollado por Proforma (**www.proformacorp.com**), es representativo de muchas herramientas que ayudan a definir el proceso y que automatizan el flujo del trabajo.

En la dirección **www.processwave.net/Links/tool_links.htm**, se encuentra una lista extensa de muchas herramientas diferentes asociadas con el proceso del software.

2.5 EL PROCESO UNIFICADO

En su libro fundamental, *Unified Process*, Ivar Jacobson, Grady Booch y James Rumbaugh [Jac99] analizan la necesidad de un proceso del software "impulsado por el caso de uso, centrado en la arquitectura, iterativo e incremental", con la afirmación siguiente:

17 Las herramientas mencionadas aquí no representan una obligación; sólo son una muestra de las de esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus desarrolladores respectivos.

> En la actualidad, la tendencia en el software es hacia sistemas más grandes y complejos. Eso se debe en parte al hecho de que año tras año las computadoras son más poderosas, lo que hace que los usuarios esperen más de ellas. Esta tendencia también se ha visto influida por el uso creciente de internet para intercambiar toda clase de información [...] Nuestro apetito por software cada vez más sofisticado aumenta conforme aprendemos, entre un lanzamiento y otro de un producto, cómo mejorar éste. Queremos software que se adapte mejor a nuestras necesidades, pero eso a su vez lo hace más complejo. En pocas palabras, queremos más.

En cierto modo, el proceso unificado es un intento por obtener los mejores rasgos y características de los modelos tradicionales del proceso del software, pero en forma que implemente muchos de los mejores principios del desarrollo ágil de software (véase el capítulo 3). El proceso unificado reconoce la importancia de la comunicación con el cliente y los métodos directos para describir su punto de vista respecto de un sistema (el caso de uso).[18] Hace énfasis en la importancia de la arquitectura del software y "ayuda a que el arquitecto se centre en las metas correctas, tales como que sea comprensible, permita cambios futuros y la reutilización" [Jac99]: Sugiere un flujo del proceso iterativo e incremental, lo que da la sensación evolutiva que resulta esencial en el desarrollo moderno del software.

2.5.1 Breve historia

Al principio de la década de 1990, James Rumbaugh [Rum91], Grady Booch [Boo94] e Ivar Jacobson [Jac92] comenzaron a trabajar en un "método unificado" que combinaría lo mejor de cada uno de sus métodos individuales de análisis y diseño orientado a objetos. El resultado fue un UML, *lenguaje de modelado unificado*, que contiene una notación robusta para el modelado y desarrollo de los sistemas orientados a objetos.

El UML se utiliza en toda la parte 2 del libro para representar tanto los modelos de requerimientos como el diseño. En el apéndice 1 se presenta un método introductorio a la enseñanza para quienes no están familiarizados con las reglas básicas de notación y modelado con el UML. El estudio exhaustivo del UML se deja a libros dedicados al tema. En el apéndice 1 se enlistan los textos recomendables.

El UML brinda la tecnología necesaria para apoyar la práctica de la ingeniería de software orientada a objetos, pero no da la estructura del proceso que guíe a los equipos del proyecto cuando aplican la tecnología. En los siguientes años, Jacobson, Rumbaugh y Booch desarrollaron el *proceso unificado*, estructura para la ingeniería de software orientado a objetos que utiliza UML. Actualmente, el proceso unificado (PU) y el UML se usan mucho en proyectos de toda clase orientados a objetos. El modelo iterativo e incremental propuesto por el PU puede y debe adaptarse para que satisfaga necesidades específicas del proyecto.

2.5.2 Fases del proceso unificado[19]

PUNTO CLAVE

Las fases del PU tienen un objetivo similar al de las actividades estructurales generales definidas en este libro.

Al principio de este capítulo se estudiaron cinco actividades estructurales generales y se dijo que podían usarse para describir cualquier modelo de proceso del software. El proceso unificado no es la excepción. La figura 2.9 ilustra las "fases" del PU y las relaciona con las actividades generales estudiadas en el capítulo 1 y al inicio de éste.

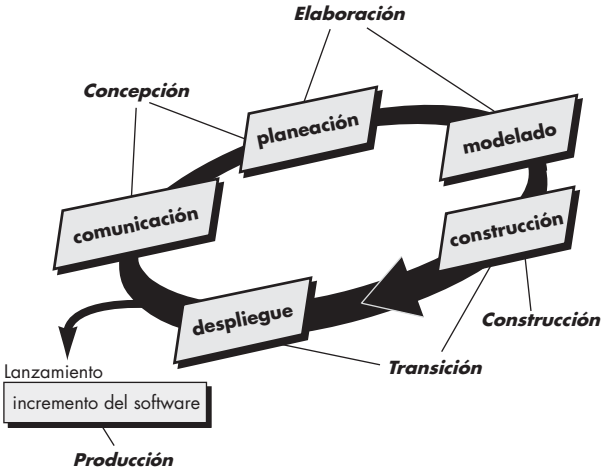
La *fase de concepción* del PU agrupa actividades tanto de comunicación con el cliente como de planeación. Al colaborar con los participantes, se identifican los requerimientos del negocio,

---

18 El *caso de uso* (véase el capítulo 5) es la narración o plantilla que describe una función o rasgo de un sistema desde el punto de vista del usuario. Éste escribe un caso en uso que sirve como base para la creación de un modelo de requerimientos más completos.

19 El proceso unificado en ocasiones recibe el nombre de *Proceso Racional Unificado* (PRU), acuñado por Rational Corporation (adquirida posteriormente por IBM), que contribuyó desde el principio al desarrollo y mejora del PU y a la elaboración de ambientes completos (herramientas y tecnología) que apoyan el proceso.

FIGURA 2.9
El proceso unificado



se propone una arquitectura aproximada para el sistema y se desarrolla un plan para la naturaleza iterativa e incremental del proyecto en cuestión. Los requerimientos fundamentales del negocio se describen por medio de un conjunto de casos de uso preliminares (véase el capítulo 5) que detallan las características y funciones que desea cada clase principal de usuarios. En este punto, la arquitectura no es más que un lineamiento tentativo de subsistemas principales y la función y rasgos que tienen. La arquitectura se mejorará después y se expandirá en un conjunto de modelos que representarán distintos puntos de vista del sistema. La planeación identifica los recursos, evalúa los riesgos principales, define un programa de actividades y establece una base para las fases que se van a aplicar a medida que avanza el incremento del software.

La *fase de elaboración* incluye las actividades de comunicación y modelado del modelo general del proceso (véase la figura 2.9). La elaboración mejora y amplía los casos de uso preliminares desarrollados como parte de la fase de concepción y aumenta la representación de la arquitectura para incluir cinco puntos de vista distintos del software: los modelos del caso de uso, de requerimientos, del diseño, de la implementación y del despliegue. En ciertos casos, la elaboración crea una "línea de base de la arquitectura ejecutable" [Arl02] que representa un sistema ejecutable de "primer corte".[20] La línea de base de la arquitectura demuestra la viabilidad de ésta, pero no proporciona todas las características y funciones que se requieren para usar el sistema. Además, al terminar la fase de elaboración se revisa con cuidado el plan a fin de asegurar que el alcance, riesgos y fechas de entrega siguen siendo razonables. Es frecuente que en este momento se hagan modificaciones al plan.

WebRef

En la dirección **www.ambysoft.com/unifiedprocess/agileUP.html**, se encuentra un análisis interesante del PU en el contexto del desarrollo ágil.

La *fase de construcción* del PU es idéntica a la actividad de construcción definida para el proceso general del software. Con el uso del modelo de arquitectura como entrada, la fase de construcción desarrolla o adquiere los componentes del software que harán que cada caso de uso sea operativo para los usuarios finales. Para lograrlo, se completan los modelos de requerimientos y diseño que se comenzaron durante la fase de elaboración, a fin de que reflejen la versión final del incremento de software. Después se implementan en código fuente todas las características y funciones necesarias para el incremento de software (por ejemplo, el lanzamiento). A medida de que se implementan los componentes, se diseñan y efectúan pruebas unitarias[21] para cada uno. Además, se realizan actividades de integración (ensamble de compo-

20 Es importante darse cuenta de que la línea de base de la arquitectura no es un prototipo y que no se desecha. Por el contrario, es revestida durante la fase siguiente del PU.

21 En los capítulos 17 a 20 se presenta el análisis exhaustivo de las pruebas del software (incluso las *pruebas unitarias*).

nentes y pruebas de integración). Se emplean casos de uso para obtener un grupo de pruebas de aceptación que se ejecutan antes de comenzar la siguiente fase del PU.

La *fase de transición* del PU incluye las últimas etapas de la actividad general de construcción y la primera parte de la actividad de despliegue general (entrega y retroalimentación). Se da el software a los usuarios finales para las pruebas beta, quienes reportan tanto los defectos como los cambios necesarios. Además, el equipo de software genera la información de apoyo necesaria (por ejemplo, manuales de usuario, guías de solución de problemas, procedimientos de instalación, etc.) que se requiere para el lanzamiento. Al finalizar la fase de transición, el software incrementado se convierte en un producto utilizable que se lanza.

La *fase de producción* del PU coincide con la actividad de despliegue del proceso general. Durante esta fase, se vigila el uso que se da al software, se brinda apoyo para el ambiente de operación (infraestructura) y se reportan defectos y solicitudes de cambio para su evaluación.

Es probable que al mismo tiempo que se llevan a cabo las fases de construcción, transición y producción, comience el trabajo sobre el siguiente incremento del software. Esto significa que las cinco fases del PU no ocurren en secuencia sino que concurren en forma escalonada.

El flujo de trabajo de la ingeniería de software está distribuido a través de todas las fases del PU. En el contexto de éste, un *flujo de trabajo* es análogo al conjunto de tareas (que ya se describió en este capítulo). Es decir, un flujo de trabajo identifica las tareas necesarias para completar una acción importante de la ingeniería de software y los productos de trabajo que se generan como consecuencia de la terminación exitosa de aquéllas. Debe notarse que no toda tarea identificada para el flujo de trabajo del PU es realizada en todos los proyectos de software. El equipo adapta el proceso (acciones, tareas, subtareas y productos del trabajo) a fin de que cumpla sus necesidades.

2.6 Modelos del proceso personal y del equipo

El mejor proceso del software es el que está cerca de las personas que harán el trabajo. Si un modelo del proceso del software se ha desarrollado en un nivel corporativo u organizacional, será eficaz sólo si acepta una adaptación significativa para que cubra las necesidades del equipo de proyecto que en realidad hace el trabajo de ingeniería de software. En la situación ideal se crearía un proceso que se ajustara del mejor modo a los requerimientos, y al mismo tiempo cubriera las más amplias necesidades del equipo y de la organización. En forma alternativa, el equipo crearía un proceso propio que satisficiera las necesidades más estrechas de los individuos y las más generales de la organización. Watts Humphrey ([Hum97] y [Hum00]) afirma que es posible crear un "proceso personal de software" y/o un "proceso del equipo de software". Ambos requieren trabajo duro, capacitación y coordinación, pero los dos son asequibles.[22]

Cita:

"La persona que es exitosa tan sólo se ha hecho el hábito de hacer las cosas que no hacen las personas que no tienen éxito."

Dexter Yager

2.6.1 Proceso personal del software (PPS)

Todo desarrollador utiliza algún proceso para elaborar software de cómputo. El proceso puede ser caprichoso o *ad hoc*; quizá cambie a diario; tal vez no sea eficiente, eficaz o incluso no sirva; pero sí existe un "proceso". Watts Humphrey [Hum97] sugiere que a fin de cambiar un proceso personal ineficaz, un individuo debe pasar por las cuatro fases, cada una de las cuales requiere capacitación e instrumentación cuidadosa. El *proceso personal del software* (PPS) pone el énfasis en la medición personal tanto del producto del trabajo que se genera como de su calidad. Además, el PPS responsabiliza al profesional acerca de la planeación del proyecto (por ejemplo,

WebRef

En la dirección **www.ipd.uka.de/PSP**, se hallan muchos recursos para el PPS.

22 Es útil notar que quienes proponen un desarrollo ágil del software (véase el capítulo 3) también plantean que el proceso debe ser cercano al equipo. Para lograr esto sugieren un método alternativo.

estimación y programación de actividades) y delega en el practicante el poder de controlar la calidad de todos los productos del trabajo de software que se desarrollen. El modelo del PPS define cinco actividades estructurales:

Planeación. Esta actividad aísla los requerimientos y desarrolla las estimaciones tanto del tamaño como de los recursos. Además, realiza la estimación de los defectos (el número de defectos proyectados para el trabajo). Todas las mediciones se registran en hojas de trabajo o plantillas. Por último, se identifican las tareas de desarrollo y se crea un programa para el proyecto.

¿Qué actividades estructurales se usan durante el PPS?

Diseño de alto nivel. Se desarrollan las especificaciones externas para cada componente que se va a construir y se crea el diseño de componentes. Si hay incertidumbre, se elaboran prototipos. Se registran todos los aspectos relevantes y se les da seguimiento.

Revisión del diseño de alto nivel. Se aplican métodos de verificación formal (véase el capítulo 21) para descubrir errores en el diseño. Se mantienen las mediciones para todas las tareas y resultados del trabajo importantes.

Desarrollo. Se mejora y revisa el diseño del componente. El código se genera, revisa, compila y prueba. Las mediciones se mantienen para todas las tareas y resultados de trabajo de importancia.

Post mórtem. Se determina la eficacia del proceso por medio de medidas y mediciones obtenidas (ésta es una cantidad sustancial de datos que deben analizarse con métodos estadísticos). Las medidas y mediciones deben dar la guía para modificar el proceso a fin de mejorar su eficacia.

El PPS enfatiza la necesidad de detectar pronto los errores; de igual importancia es entender los tipos de ellos que es probable cometer. Esto se logra a través de una actividad de evaluación rigurosa ejecutada para todos los productos del trabajo que se generen.

PUNTO CLAVE

El PPS pone el énfasis en la necesidad de registrar y analizar los tipos de errores que se cometen, de modo que se desarrollen estrategias para eliminarlos.

El PPS representa un enfoque disciplinado basado en la medición para la ingeniería de software que quizá sea un choque cultural para muchos de sus practicantes. Sin embargo, cuando se introduce el PPS en forma apropiada en los ingenieros de software [Hum96], es significativa la mejora resultante en la productividad de la ingeniería respectiva y en la calidad del software [Fer97]. No obstante, el PPS no ha sido adoptado con amplitud por la industria. Es triste reconocer que las razones de esto tienen que ver más con la naturaleza humana y la inercia organizacional que con las fortalezas y debilidades del enfoque del PPS. Dicho enfoque plantea desafíos intelectuales y demanda un nivel de compromiso (por parte de los practicantes y sus administradores) que no siempre es posible obtener. La capacitación es relativamente larga y sus costos elevados. El nivel requerido de las mediciones es culturalmente difícil para muchas personas de la comunidad del software.

¿Es posible usar el PPS como un proceso eficaz de software a nivel personal? La respuesta es un rotundo "sí". Pero aun si no se adoptara por completo el PPS, muchos de los conceptos del proceso de mejora personal que introduce constituyen un aprendizaje provechoso.

2.6.2 Proceso del equipo de software (PES)

WebRef

En la dirección **www.sei.cmu.edu/tsp/**, hay información sobre la formación de equipos de alto rendimiento que usan PES y PPS.

Debido a que muchos proyectos de software industrial son elaborados por un equipo de profesionales, Watts Humphrey extendió las lecciones aprendidas de la introducción del PPS y propuso un *proceso del equipo de software* (PES). El objetivo de éste es construir un equipo "autodirigido" para el proyecto, que se organice para producir software de alta calidad. Humphrey [Hum98] define los objetivos siguientes para el PES:

- Formar equipos autodirigidos que planeen y den seguimiento a su trabajo, que establezcan metas y que sean dueños de sus procesos y planes. Éstos pueden ser equipos de software puros o de productos integrados (EPI) constituidos por 3 a 20 ingenieros.

- Mostrar a los gerentes cómo dirigir y motivar a sus equipos y cómo ayudarlos a mantener un rendimiento máximo.
- Acelerar la mejora del proceso del software, haciendo del modelo de madurez de la capacidad, CMM,[23] nivel 5, el comportamiento normal y esperado.
- Brindar a las organizaciones muy maduras una guía para la mejora.
- Facilitar la enseñanza universitaria de aptitudes de equipo con grado industrial.

Para formar un equipo autodirigido, usted debe colaborar bien en lo interno y comunicarse bien en lo externo.

Un equipo autodirigido tiene la comprensión consistente de sus metas y objetivos generales; define el papel y responsabilidad de cada miembro del equipo; da seguimiento cuantitativo a los datos del proyecto (sobre la productividad y calidad); identifica un proceso de equipo que sea apropiado para el proyecto y una estrategia para implementarlo; define estándares locales aplicables al trabajo de ingeniería de software del equipo; evalúa en forma continua el riesgo y reacciona en consecuencia; y da seguimiento, administra y reporta el estado del proyecto.

El PES define las siguientes actividades estructurales: **inicio del proyecto, diseño de alto nivel, implementación, integración y pruebas,** y **post mórtem.** Como sus contrapartes del PPS (observe que la terminología es algo diferente), estas actividades permiten que el equipo planee, diseñe y construya software en forma disciplinada, al mismo tiempo que mide cuantitativamente el proceso y el producto. La etapa post mórtem es el escenario de las mejoras del proceso.

El PES utiliza una variedad amplia de *scripts*, formatos y estándares que guían a los miembros del equipo en su trabajo. Los *scripts* definen actividades específicas del proceso (por ejemplo, inicio del proyecto, diseño, implementación, integración y pruebas del sistema, y post mórtem), así como otras funciones más detalladas del trabajo (planeación del desarrollo, desarrollo de requerimientos, administración de la configuración del software y prueba unitaria) que forman parte del proceso de equipo.

Los scripts del PES definen elementos del proceso del equipo y de las actividades que ocurren dentro del proceso.

El PES reconoce que los mejores equipos de software son los autodirigidos.[24] Los miembros del equipo establecen los objetivos del proyecto, adaptan el proceso para que cubra las necesidades, controlan la programación de actividades del proyecto y, con la medida y análisis de las mediciones efectuadas, trabajan de manera continua en la mejora del enfoque de ingeniería de software que tiene el equipo.

Igual que el PPS, el PES es un enfoque riguroso para la ingeniería de software y proporciona beneficios distintivos y cuantificables en productividad y calidad. El equipo debe tener un compromiso total con el proceso y recibir capacitación completa para asegurar que el enfoque se aplique en forma apropiada.

2.7 Tecnología del proceso

El equipo del software debe adaptar uno o más de los modelos del proceso estudiados en las secciones precedentes. Para ello, se han desarrollado *herramientas de tecnología del proceso* que ayudan a las organizaciones de software a analizar su proceso actual, organizar las tareas de trabajo, controlar y vigilar el avance, y administrar la calidad técnica.

Las herramientas de tecnología del proceso permiten que una organización de software construya un modelo automatizado de la estructura del proceso, conjuntos de tareas y actividades sombrilla, estudiados en la sección 2.1. El modelo, que normalmente se representa como

23 El modelo de madurez de la capacidad (CMM), que es una medida de la eficacia de un proceso del software, se estudia en el capítulo 30.

24 En el capítulo 31 se analiza la importancia de los equipos "autoorganizados" como elemento clave del desarrollo ágil del software.

una red, se analiza para determinar el flujo de trabajo normal y se examinan estructuras alternativas del proceso que podrían llevar a disminuir el tiempo o costo del desarrollo.

Una vez creado un proceso aceptable, se emplean otras herramientas de tecnología para asignar, vigilar e incluso controlar todas las actividades, acciones y tareas de la ingeniería de software definidas como parte del modelo del proceso. Cada miembro de un equipo de software utiliza dichas herramientas para desarrollar una lista de verificación de las tareas de trabajo que deben realizarse. La herramienta de tecnología del proceso también se usa para coordinar el empleo de otras herramientas de la ingeniería de software que sean apropiadas para una tarea particular del trabajo.

HERRAMIENTAS DE SOFTWARE

Herramientas de modelado del proceso

Objetivo: Si una organización trabaja para mejorar un proceso (o software) de negocios, primero debe entenderlo. Las herramientas de modelado del proceso (también llamadas herramientas de *tecnología del proceso* o *de administración del proceso*) se usan para representar los elementos clave de un proceso, de modo que se entienda mejor. Dichas herramientas también se relacionan con descripciones del proceso que ayudan a los involucrados a entender las acciones y tareas del trabajo que se requieren para llevarlo a cabo. Las herramientas de modelado del proceso tienen vínculos con otras que dan apoyo a las actividades del proceso definido.

Mecánica: Las herramientas en esta categoría permiten que un equipo defina los elementos de un modelo de proceso único (acciones, tareas, productos del trabajo, puntos de aseguramiento de la calidad, etc.), dan una guía detallada acerca del contenido o descripción de cada elemento del proceso, y después administran el proceso conforme se realiza. En ciertos casos, las herramientas de tecnología del proceso incorporan tareas estándar de administración de proyectos, tales como estimación, programación, seguimiento y control.

Herramientas representativas:[25]

Igrafx Process Tools: herramientas que permiten que un equipo mapee, mida y modele el proceso del software (**www.micrografx.com**)

Adeptia BPM Server: diseñado para administrar, automatizar y optimizar procesos de negocios (**www.adptia.com**)

SpeedDev Suite: conjunto de seis herramientas con mucho énfasis en las actividades de administración de la comunicación y modelado (**www.speedev.com**)

2.8 PRODUCTO Y PROCESO

Si el proceso es deficiente, no cabe duda de que el producto final sufrirá. Pero también es peligrosa la dependencia excesiva del proceso. En un ensayo corto escrito hace muchos años, Margaret Davis [Dav95a] hace comentarios atemporales sobre la dualidad del producto y del proceso:

> Cada diez años, más o menos, la comunidad del software redefine "el problema" por medio de cambiar su atención de aspectos del producto a aspectos del proceso. Así, hemos adoptado lenguajes de programación estructurada (producto) seguidos de métodos de análisis estructurados (proceso) que van seguidos por el encapsulamiento de datos (producto) a los que siguieron el énfasis actual en el modelo de madurez de la capacidad, del Instituto de Ingeniería de Software para el Desarrollo de Software (proceso) (seguido por métodos orientados a objetos, a los que sigue el desarrollo ágil de software).
>
> En tanto que la tendencia natural de un péndulo es alcanzar el estado de reposo en el punto medio entre dos extremos, la atención de la comunidad del software cambia constantemente porque se aplica una nueva fuerza al fallar la última oscilación. Estos vaivenes son dañinos en sí mismos porque confunden al profesional promedio del software al cambiar en forma radical lo que significa hacer el trabajo bien. Los cambios periódicos no resuelven "el problema" porque están predestinados a fallar toda vez que el producto y el proceso son tratados como si fueran una dicotomía en lugar de una dualidad.

25 Las herramientas mencionadas aquí no son obligatorias, sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

> En la comunidad científica existe el precedente de adoptar nociones de dualidad cuando las contradicciones en las observaciones no pueden ser explicadas por alguna teoría alternativa. La naturaleza dual de la luz, que parece ser al mismo tiempo onda y partícula, ha sido aceptada desde la década de 1920, cuando la propuso Louis de Broglie. Pienso que las observaciones que podemos hacer sobre el conjunto del software y su desarrollo demuestran una dualidad fundamental entre el producto y el proceso. Nunca es posible derivar u obtener todo el conjunto, su contexto, uso, significado y beneficios si se le ve sólo como proceso o sólo como producto...
>
> Toda la actividad humana es un proceso, pero cada uno de nosotros obtiene un sentido de beneficio propio gracias a aquellas actividades que dan como resultado una representación o instancia que puede usar o apreciar más de una persona, utilizarla una y otra vez, o emplearla en algún otro contexto no considerado. Es decir, obtenemos sentimientos de satisfacción por la reutilización de nuestros productos, ya sea que lo hagamos nosotros u otras personas.
>
> Entonces, si bien la rápida asimilación de las metas de reutilización en el desarrollo del software incrementa potencialmente la satisfacción que obtienen los profesionales del software en su trabajo, también aumenta la urgencia de la aceptación de la dualidad de producto y proceso. Pensar en un artefacto reutilizable como si fuera sólo un producto o sólo un proceso oscurece el contexto y las formas de emplearlo, o bien oculta el hecho de que cada uso da como resultado un producto que a su vez será utilizado como entrada para alguna otra actividad de desarrollo de software. Privilegiar un punto de vista sobre el otro reduce mucho las oportunidades para la reutilización y, por tanto, se pierde la oportunidad de aumentar la satisfacción por el trabajo.

La gente obtiene tanta (o más) satisfacción del proceso creativo como del producto final. Un artista disfruta las pinceladas tanto como el resultado que enmarca. Un escritor goza de la búsqueda de la metáfora apropiada tanto como del libro terminado. Como profesional creativo del software, usted también debe obtener tanta satisfacción del proceso como del producto final. La dualidad de producto y proceso es un elemento importante para hacer que personas creativas se involucren conforme la ingeniería de software evoluciona.

2.9 Resumen

Un modelo general del proceso para la ingeniería de software incluye un conjunto de actividades estructurales y sombrilla, acciones y tareas de trabajo. Cada uno de los modelos de proceso puede describirse por un flujo distinto del proceso: descripción de cómo se organizan secuencial y cronológicamente las actividades estructurales, acciones y tareas. Los patrones del proceso pueden utilizarse para resolver los problemas comunes que surgen como parte del proceso del software.

Los modelos de proceso prescriptivo se han aplicado durante muchos años en un esfuerzo por introducir orden y estructura al desarrollo de software. Cada uno de dichos modelos sugiere un flujo de proceso algo distinto, pero todos llevan a cabo el mismo conjunto de actividades estructurales generales: comunicación, planeación, modelado, construcción y desarrollo.

Los modelos de proceso secuencial, como el de la cascada y en V, son los paradigmas más antiguos del software. Sugieren un flujo lineal del proceso que con frecuencia no es congruente con las realidades modernas (cambio continuo, sistemas en evolución, plazos ajustados, etc.) del mundo del software. Sin embargo, tienen aplicación en situaciones en las que los requerimientos están bien definidos y son estables.

Los modelos de proceso incremental son de naturaleza iterativa y producen con mucha rapidez versiones funcionales del software. Los modelos de proceso evolutivo reconocen la naturaleza iterativa e incremental de la mayoría de proyectos de ingeniería de software y están diseñados para aceptar los cambios. Los modelos evolutivos, tales como el de hacer prototipos y el espiral, generan rápido productos de trabajo incremental (o versiones funcionales del software). Estos modelos se adoptan para aplicarse a lo largo de todas las actividades de la inge-

niería de software, desde el desarrollo del concepto hasta el mantenimiento del sistema a largo plazo.

El modelo de proceso concurrente permite que un equipo de software represente los elementos iterativos y concurrentes de cualquier modelo de proceso. Los modelos especializados incluyen el basado en componentes, que pone el énfasis en la reutilización y ensamble de los componentes; el modelo de métodos formales consiste en un enfoque basado en matemáticas para desarrollar y verificar el software; y el modelo orientado a aspectos implica preocupaciones globales que afectan toda la arquitectura del sistema. El proceso unificado es un proceso del software diseñado como estructura para los métodos y herramientas del UML, y está "impulsado por el caso de uso, centrado en la arquitectura, y es iterativo e incremental".

Se han propuesto modelos personal y del equipo para el proceso del software. Ambos enfatizan la medición, planeación y autodirección como los ingredientes clave para un proceso exitoso del software.

Problemas y puntos por evaluar

2.1. En la introducción de este capítulo, Baetjer afirma que: "El proceso genera interacción entre usuarios y diseñadores, entre usuarios y herramientas cambiantes [tecnología]." Enliste cinco preguntas que *a*) los diseñadores deben responder a los usuarios, *b*) los usuarios deben plantear a los diseñadores, *c*) los usuarios deben hacerse a sí mismos sobre el producto de software que ha de elaborarse, *d*) los diseñadores deben plantearse acerca del producto de software que va a construirse y del proceso que se usará para ello.

2.2. Trate de desarrollar un conjunto de acciones para la actividad de comunicación. Seleccione una acción y defina un conjunto de tareas para ella.

2.3. Un problema común durante la **comunicación** ocurre cuando se encuentra a dos participantes que tienen ideas en conflicto sobre lo que debe ser el software, es decir, que tienen requerimientos mutuamente conflictivos. Desarrolle un patrón del proceso (esto sería un patrón de la etapa) con el empleo de la plantilla presentada en la sección 2.1.3 que aborda este problema y sugiera un enfoque eficaz para él.

2.4. Investigue un poco sobre el PPS y haga una breve presentación que describa los tipos de mediciones que se pide hacer a un ingeniero individual de software y la forma en la que pueden usarse para mejorar la eficacia personal.

2.5. El uso de scripts (mecanismo requerido en el PES) no es apreciado de manera universal en la comunidad del software. Haga una lista de pros y contras en relación con los scripts y sugiera al menos dos situaciones en las que serían útiles, y otras dos en las que generarían menos beneficios.

2.6. Lea a [Nog00] y escriba un ensayo de dos o tres páginas donde analice el efecto que tiene el "caos" en la ingeniería de software.

2.7. Dé tres ejemplos de proyectos de software que podrían efectuarse con el modelo de cascada. Sea específico.

2.8. Proporcione tres ejemplos de proyectos de software que podrían abordarse con el modelo de hacer prototipos. Sea específico.

2.9. ¿Qué adaptaciones del proceso se requerirían si el proyecto evolucionara en un sistema o producto que se entregase?

2.10. Diga tres ejemplos de proyectos de software que podrían realizarse con el modelo incremental. Sea específico.

2.11. Conforme avanza hacia fuera por el flujo de proceso en espiral, ¿qué puede decirse sobre el software que se está desarrollando o que está en mantenimiento?

2.12. ¿Es posible combinar modelos de proceso? Si es así, diga un ejemplo.

2.13. El modelo de proceso concurrente define un conjunto de "estados". Describa con sus propias palabras qué es lo que representan, y después indique cómo entran en juego dentro del modelo de proceso concurrente.

2.14. ¿Cuáles son las ventajas y desventajas de desarrollar software en el que la calidad no es "suficientemente buena"? Es decir, ¿qué pasa cuando se pone el énfasis en la velocidad de desarrollo sobre la calidad del producto?

2.15. Dé tres ejemplos de proyectos de software que serían abordables con el modelo basado en componentes. Sea específico.

2.16. ¿Es posible demostrar que un componente de software, o incluso un programa completo, es correcto? Entonces, ¿por qué no todos lo hacen?

2.17. ¿Son lo mismo el proceso unificado y el UML? Explique su respuesta.

Lecturas adicionales y fuentes de información

La mayor parte de los libros de ingeniería de software consideran en detalle los modelos de proceso tradicionales. Libros como el de Sommerville (*Software Engineering*, 8a. ed., Addison-Wesley, 2006), Pfleeger y Atlee (*Software Engineering*, 3a. ed., Prentice-Hall, 2005), y Schach (*Object-Oriented and Classical Software Engineering*, 7a. ed., McGraw-Hill, 2006) consideran los paradigmas tradicionales y estudian sus fortalezas y debilidades. Glass (*Facts and Fallacies of Software Engineering*, Prentice-Hall, 2002) da un punto de vista pragmático y crudo del proceso de ingeniería de software. Aunque no se dedica específicamente al proceso, Brooks (*The Mythical Man-Month*, 2a. ed., Addison-Wesley, 1995) presenta la sabiduría antigua sobre los proyectos y plantea que todo tiene que ver con el proceso.

Firesmith y Henderson-Sellers (*The OPEN Process Framework: An Introduction*, Addison-Wesley, 2001) presenta una plantilla general para crear "procesos de software flexibles pero con disciplina" y analiza los atributos y objetivos del proceso. Madachy (*Software Process Dynamics*, Wiley-IEEE, 2008) estudia técnicas de modelado que permiten analizar los elementos técnicos y sociales interrelacionados del proceso del software. Sharpe y McDermott (*Workflow Modeling: Tools for Process Improvement and Application Development*, Artech House, 2001) presentan herramientas para modelar procesos tanto de software como de negocios.

Lim (*Managing Software Reuse*, Prentice-Hall, 2004) estudia la reutilización desde la perspectiva del gerente. Ezran, Morisio y Tully (*Practical Software Reuse*, Springer, 2002) y Jacobson, Griss y Jonsson (*Software Reuse*, Addison-Wesley, 1997) presentan mucha información útil sobre el desarrollo basado en componentes. Heineman y Council (*Component-Based Software Engineering*, Addison-Wesley, 2001) describen el proceso requerido para implementar sistemas basados en componentes. Kenett y Baker (*Software Process Quality: Management and Control*, Marcel Dekker, 1999) analizan la manera en la que se conectan íntimamente la administración de la calidad y el diseño del proceso.

Nygard (*Release It!: Design and Deploy Production-Ready Software,* Pragmatic Bookshelf, 2007) y Richardson y Gwaltney (*Ship it! A Practical Guide to Successful Software Projects*, Pragmatic Bookshelf, 2005) presentan una amplia colección de lineamientos útiles aplicables a la actividad de despliegue.

Además del libro fundamental de Jacobson, Rumbaugh y Booch acerca del proceso unificado [Jac99], los libros de Arlow y Neustadt (*UML 2 and the Unified Process,* Addison-Wesley, 2005), Kroll y Kruchten (*The Rational Unified Process Made Easy*, Addison-Wesley, 2003) y Farve (*UML and the Unified Process*, IRM Press, 2003) proveen información complementaria excelente. Gibbs (*Project Management with the IBM Rational Unified Process*, IBM Press, 2006) analiza la administración de proyectos dentro del contexto del PU.

En internet existe una amplia variedad de fuentes de información sobre la ingeniería de software y el proceso del software. En el sitio web del libro, **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, hay una lista actualizada de referencias en la Red Mundial que son relevantes para el proceso del software.

CAPÍTULO

3

DESARROLLO ÁGIL

CONCEPTOS CLAVE

agilidad 56
Cristal 72
Desarrollo adaptativo de software 68
Desarrollo esbelto de software 73
DIC. 72
historias 62
MDSD 71
proceso ágil 58
Proceso unificado ágil 75
proceso XP 62
programación extrema. 61
programación por parejas . . . 64
rediseño 63
Scrum. 69
velocidad del proyecto. 63
XP industrial. 65

En 2001, Kent Beck y otros 16 notables desarrolladores de software, escritores y consultores [Bec01a] (grupo conocido como la "Alianza Ágil") firmaron el "Manifiesto por el desarrollo ágil de software". En él se establecía lo siguiente:

> Estamos descubriendo formas mejores de desarrollar software, por medio de hacerlo y de dar ayuda a otros para que lo hagan. Ese trabajo nos ha hecho valorar:
>
> *Los individuos y sus interacciones*, sobre los procesos y las herramientas
>
> *El software que funciona*, más que la documentación exhaustiva
>
> *La colaboración con el cliente*, y no tanto la negociación del contrato
>
> *Responder al cambio*, mejor que apegarse a un plan
>
> Es decir, si bien son valiosos los conceptos que aparecen en segundo lugar, valoramos más los que aparecen en primer sitio.

Un manifiesto normalmente se asocia con un movimiento político emergente: ataca a la vieja guardia y sugiere un cambio revolucionario (se espera que para mejorar). En cierta forma, de eso es de lo que trata el desarrollo ágil.

Aunque las ideas subyacentes que lo guían han estado durante muchos años entre nosotros, ha sido en menos de dos décadas que cristalizaron en un "movimiento". Los métodos ágiles[1] se desarrollaron como un esfuerzo por superar las debilidades reales y percibidas de la ingeniería de software convencional. El desarrollo ágil proporciona beneficios importantes, pero no es

UNA MIRADA RÁPIDA

¿Qué es? La ingeniería de software ágil combina una filosofía con un conjunto de lineamientos de desarrollo. La filosofía pone el énfasis en: la satisfacción del cliente y en la entrega rápida de software incremental, los equipos pequeños y muy motivados para efectuar el proyecto, los métodos informales, los productos del trabajo con mínima ingeniería de software y la sencillez general en el desarrollo. Los lineamientos de desarrollo enfatizan la entrega sobre el análisis y el diseño (aunque estas actividades no se desalientan) y la comunicación activa y continua entre desarrolladores y clientes.

¿Quién lo hace? Los ingenieros de software y otros participantes en el proyecto (gerentes, clientes, usuarios finales, etc.) trabajan juntos en un proyecto ágil, formando un equipo con organización propia y que controla su propio destino. Un equipo ágil facilita la comunicación y colaboración entre aquellos a quienes sirve.

¿Por qué es importante? El ambiente moderno de negocios que genera sistemas basados en computadora y productos de software evoluciona rápida y constantemente. La ingeniería de software ágil representa una alternativa razonable a la ingeniería de software convencional para ciertas clases de software y en algunos tipos de proyectos. Asimismo, se ha demostrado que concluye con rapidez sistemas exitosos.

¿Cuáles son los pasos? Un nombre más apropiado para el desarrollo ágil sería "ingeniería de software ligero". Permanecen las actividades estructurales fundamentales: comunicación, planeación, modelado, construcción y despliegue. Pero se transforman en un conjunto mínimo de tareas que lleva al equipo del proyecto hacia la construcción y entrega (algunas personas dirían que esto se hace a costa del análisis del problema y del diseño de la solución).

¿Cuál es el producto final? Tanto el cliente como el ingeniero de software tienen la misma perspectiva: el único producto del trabajo realmente importante es un "incremento de software" operativo que se entrega al cliente exactamente en la fecha acordada.

¿Cómo me aseguro de que lo hice bien? El trabajo estará bien hecho si el equipo ágil concuerda en que el proceso funciona y en que produce incrementos de software utilizables que satisfagan al cliente.

1 En ocasiones se conoce a los métodos ágiles como *métodos ligeros* o *métodos esbeltos*.

aplicable a todos los proyectos, productos, personas y situaciones. *No* es la antítesis de la práctica de la ingeniería de software sólida y puede aplicarse como filosofía general para todo el trabajo de software.

Es frecuente que en la economía moderna sea difícil o imposible predecir la forma en la que evolucionará un sistema basado en computadora (por ejemplo, una aplicación con base en web). Las condiciones del mercado cambian con rapidez, las necesidades de los usuarios finales se transforman y emergen nuevas amenazas competitivas sin previo aviso. En muchas situaciones no será posible definir los requerimientos por completo antes de que el proyecto comience. Se debe ser suficientemente ágil para responder a lo fluido que se presenta el ambiente de negocios.

La fluidez implica cambio, y el cambio es caro, en particular si es descontrolado o si se administra mal. Una de las características más atractivas del enfoque ágil es su capacidad de reducir los costos del cambio durante el proceso del software.

¿Significa esto que el reconocimiento de los retos planteados por las realidades modernas hace que sean descartables los valiosos principios, conceptos, métodos y herramientas de la ingeniería del software? No, en absoluto… Igual que todas las disciplinas de la ingeniería, la del software evoluciona en forma continua. Puede adaptarse con facilidad para que satisfaga los desafíos que surgen de la demanda de agilidad.

Cita:

"Agilidad: 1, todo lo demás: 0."

Tom DeMarco

En un libro que suscita la reflexión sobre el desarrollo de software ágil, Alistair Cockburn [CocO2] argumenta que los modelos de proceso prescriptivo, introducidos en el capítulo 2, tienen una falla grande: *olvidan las flaquezas de las personas cuando construyen software*. Los ingenieros de software no son robots. Sus estilos de trabajo varían mucho; tienen diferencias significativas en habilidad, creatividad, orden, consistencia y espontaneidad. Algunos se comunican bien por escrito, pero otros no. Cockburn afirma que los modelos de proceso pueden "manejar las carencias de disciplina o tolerancia de las personas comunes" y que los modelos de proceso más prescriptivo eligen la disciplina. Dice: "Como la consistencia de las acciones es una debilidad humana, las metodología que requieren mucha disciplina son frágiles."

Para funcionar, los modelos de proceso deben proveer un mecanismo realista que estimule la disciplina necesaria, o deben caracterizarse por la "tolerancia" con las personas que hacen el trabajo de ingeniería de software. Invariablemente, las prácticas tolerantes son más fáciles de adoptar y sostener por parte de la comunidad del software, pero son menos productivas (como admite Cockburn). Debe considerarse la negociación entre ellas, como en todas las cosas de la vida.

3.1 ¿Qué es la agilidad?

Pero, ¿qué es la agilidad en el contexto del trabajo de la ingeniería de software? Ivar Jacobson [Jac02a] hace un análisis útil:

> La *agilidad* se ha convertido en la palabra mágica de hoy para describir un proceso del software moderno. Todos son ágiles. Un equipo ágil es diestro y capaz de responder de manera apropiada a los cambios. El cambio es de lo que trata el software en gran medida. Hay cambios en el software que se construye, en los miembros del equipo, debidos a las nuevas tecnologías, de todas clases y que tienen un efecto en el producto que se elabora o en el proyecto que lo crea. Deben introducirse apoyos para el cambio en todo lo que se haga en el software; en ocasiones se hace porque es el alma y corazón de éste. Un equipo ágil reconoce que el software es desarrollado por individuos que trabajan en equipo, y que su capacidad, su habilidad para colaborar, es el fundamento para el éxito del proyecto.

Desde el punto de vista de Jacobson, la ubicuidad del cambio es el motor principal de la agilidad. Los ingenieros de software deben ir rápido si han de adaptarse a los cambios veloces que describe Jacobson.

CONSEJO

No cometa el error de suponer que la agilidad le da permiso para improvisar soluciones. Se requiere de un proceso, y la disciplina es esencial.

Pero la agilidad es algo más que una respuesta efectiva al cambio. También incluye la filosofía expuesta en el manifiesto citado al principio de este capítulo. Ésta recomienda las estructuras de equipo y las actitudes que hacen más fácil la comunicación (entre los miembros del equipo, tecnólogos y gente de negocios, entre los ingenieros de software y sus gerentes, etc.); pone el énfasis en la entrega rápida de software funcional y resta importancia a los productos intermedios del trabajo (lo que no siempre es bueno); adopta al cliente como parte del equipo de desarrollo y trabaja para eliminar la actitud de "nosotros y ellos" que todavía invade muchos proyectos de software; reconoce que la planeación en un mundo incierto tiene sus límites y que un plan de proyecto debe ser flexible.

La agilidad puede aplicarse a cualquier proceso del software. Sin embargo, para lograrlo es esencial que éste se diseñe en forma que permita al equipo del proyecto adaptar las tareas y hacerlas directas, ejecutar la planeación de manera que entienda la fluidez de un enfoque ágil del desarrollo, eliminar todos los productos del trabajo excepto los más esenciales y mantenerlos esbeltos, y poner el énfasis en una estrategia de entrega incremental que haga trabajar al software tan rápido como sea posible para el cliente, según el tipo de producto y el ambiente de operación.

3.2 LA AGILIDAD Y EL COSTO DEL CAMBIO

La sabiduría convencional del desarrollo de software (apoyada por décadas de experiencia) señala que el costo se incrementa en forma no lineal a medida que el proyecto avanza (véase la figura 3.1, curva continua negra). Es relativamente fácil efectuar un cambio cuando el equipo de software reúne los requerimientos (al principio de un proyecto). El escenario de uso tal vez tenga que modificarse, la lista de funciones puede aumentar, o editarse una especificación escrita. Los costos de hacer que esto funcione son mínimos, y el tiempo requerido no perjudicará el resultado del proyecto. Pero, ¿qué pasa una vez transcurridos algunos meses? El equipo está a la mitad de las pruebas de validación (algo que ocurre cuando el proyecto está relativamente avanzado) y un participante de importancia solicita que se haga un cambio funcional grande. El cambio requiere modificar el diseño de la arquitectura del software, el diseño y construcción de tres componentes nuevos, hacer cambios en otros cinco componentes, diseñar nuevas pruebas, etc. Los costos aumentan con rapidez, y no son pocos el tiempo y el dinero requeridos para asegurar que se haga el cambio sin efectos colaterales no intencionados.

Cita:

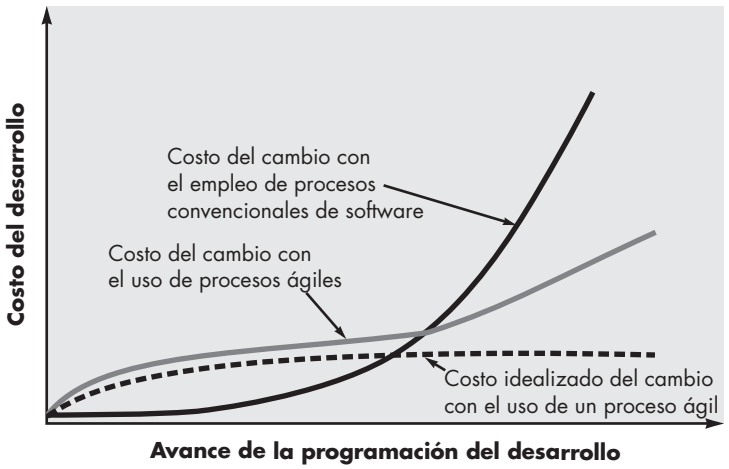
"La agilidad es dinámica, específica en el contenido, acepta con entusiasmo el cambio y se orienta al crecimiento."

Steven Goldman et al.

Los defensores de la agilidad (por ejemplo [Bec001] y [Amb04]) afirman que un proceso ágil bien diseñado "aplana" el costo de la curva de cambio (véase la figura 3.1, curva continua y

FIGURA 3.1

Cambio de los costos como función del tiempo transcurrido en el desarrollo



sombreada), lo que permite que el equipo de software haga cambios en una fase tardía de un proyecto de software sin que haya un efecto notable en el costo y en el tiempo. El lector ya sabe que el proceso ágil incluye la entrega incremental. Cuando ésta se acopla con otras prácticas ágiles, como las pruebas unitarias continuas y la programación por parejas (que se estudia más adelante, en este capítulo), el costo de hacer un cambio disminuye. Aunque hay debate sobre el grado en el que se aplana la curva de costo, existen evidencias [Coc01a] que sugieren que es posible lograr una reducción significativa del costo.

PUNTO CLAVE

Un proceso ágil reduce el costo del cambio porque el software se entrega en incrementos y en esta forma el cambio se controla mejor.

3.3 ¿Qué es un proceso ágil?

Cualquier proceso del software ágil se caracteriza por la forma en la que aborda cierto número de suposiciones clave [Fow02] acerca de la mayoría de proyectos de software:

1. Es difícil predecir qué requerimientos de software persistirán y cuáles cambiarán. También es difícil pronosticar cómo cambiarán las prioridades del cliente a medida que avanza el proyecto.
2. Para muchos tipos de software, el diseño y la construcción están imbricados. Es decir, ambas actividades deben ejecutarse en forma simultánea, de modo que los modelos de diseño se prueben a medida que se crean. Es difícil predecir cuánto diseño se necesita antes de que se use la construcción para probar el diseño.
3. El análisis, el diseño, la construcción y las pruebas no son tan predecibles como nos gustaría (desde un punto de vista de planeación).

WebRef

En la dirección **www.aanpo.org/articles/index** hay una colección completa de artículos sobre el proceso ágil.

Dadas estas tres suposiciones, surge una pregunta importante: ¿cómo crear un proceso que pueda manejar lo *impredecible*? La respuesta, como ya se dijo, está en la adaptabilidad del proceso (al cambio rápido del proyecto y a las condiciones técnicas). Por tanto, un proceso ágil debe ser *adaptable*.

Pero la adaptación continua logra muy poco si no hay avance. Entonces, un proceso de software ágil debe adaptarse *incrementalmente*. Para lograr la adaptación incremental, un equipo ágil requiere retroalimentación con el cliente (de modo que sea posible hacer las adaptaciones apropiadas). Un catalizador eficaz para la retroalimentación con el cliente es un prototipo operativo o una porción de un sistema operativo. Así, debe instituirse una *estrategia de desarrollo incremental*. Deben entregarse *incrementos de software* (prototipos ejecutables o porciones de un sistema operativo) en periodos cortos de tiempo, de modo que la adaptación vaya a ritmo con el cambio (impredecible). Este enfoque iterativo permite que el cliente evalúe en forma regular el incremento de software, dé la retroalimentación necesaria al equipo de software e influya en las adaptaciones del proceso que se realicen para aprovechar la retroalimentación.

3.3.1 Principios de agilidad

PUNTO CLAVE

Aunque los procesos ágiles aceptan el cambio, es importante examinar las razones de éste.

La Alianza Ágil (véase [Agi03]), [Fow01]) define 12 principios de agilidad para aquellos que la quieran alcanzar:

1. La prioridad más alta es satisfacer al cliente a través de la entrega pronta y continua de software valioso.
2. Son bienvenidos los requerimientos cambiantes, aun en una etapa avanzada del desarrollo. Los procesos ágiles dominan el cambio para provecho de la ventaja competitiva del cliente.
3. Entregar con frecuencia software que funcione, de dos semanas a un par de meses, de preferencia lo más pronto que se pueda.

4. Las personas de negocios y los desarrolladores deben trabajar juntos, a diario y durante todo el proyecto.
5. Hay que desarrollar los proyectos con individuos motivados. Debe darse a éstos el ambiente y el apoyo que necesiten, y confiar en que harán el trabajo.
6. El método más eficiente y eficaz para transmitir información a los integrantes de un equipo de desarrollo, y entre éstos, es la conversación cara a cara.
7. La medida principal de avance es el software que funciona.
8. Los procesos ágiles promueven el desarrollo sostenible. Los patrocinadores, desarrolladores y usuarios deben poder mantener un ritmo constante en forma indefinida.
9. La atención continua a la excelencia técnica y el buen diseño mejora la agilidad.
10. Es esencial la simplicidad: el arte de maximizar la cantidad de trabajo no realizado.
11. Las mejores arquitecturas, requerimientos y diseños surgen de los equipos con organización propia.
12. El equipo reflexiona a intervalos regulares sobre cómo ser más eficaz, para después afinar y ajustar su comportamiento en consecuencia.

El software que funciona es importante, pero no olvide que también debe poseer varios atributos de calidad, como ser confiable, utilizable y susceptible de recibir mantenimiento.

No todo modelo de proceso ágil aplica estos 12 principios con igual intensidad y algunos eligen ignorar (o al menos soslayar) la importancia de uno o más de ellos. Sin embargo, los principios definen un *espíritu ágil* que se mantiene en cada uno de los modelos de proceso que se presentan en este capítulo.

3.3.2 La política del desarrollo ágil

Hay mucho debate (a veces estridente) sobre los beneficios y aplicabilidad del desarrollo de software ágil como oposición a los procesos más convencionales. Jim Highsmith [Hig02a] señala (en tono de burla) los extremos cuando caracteriza la posición del campo a favor de la agilidad ("agilistas"). "Los metodólogos tradicionales están atrapados en un pantano y producirán una documentación sin defectos en vez de un sistema funcional que satisfaga las necesidades del negocio." Como contrapunto, plantea (de nuevo como burla) la posición del campo de la ingeniería de software tradicional: "Los metodólogos ligeros, perdón, 'ágiles', son un grupo de remendones famosos que se van a llevar una sorpresa cuando intenten convertir sus juguetes en software a la medida de la empresa."

No tiene que elegirse entre la agilidad y la ingeniería de software. En vez de ello, hay que definir un enfoque de ingeniería de software que sea ágil.

Como todos los argumentos sobre la tecnología de software, este debate sobre la metodología corre el riesgo de degenerar en una guerra religiosa. Si estalla, desaparece el pensamiento racional y lo que guía la toma de decisiones son las creencias y no los hechos.

Nadie está contra la agilidad. La pregunta real es: ¿cuál es la mejor forma de lograrla? De igual importancia: ¿cómo construir software que satisfaga en el momento las necesidades de los clientes y que tenga características de calidad que permitan ampliarlo y escalarlo para que también las satisfaga en el largo plazo?

No hay respuestas absolutas a ninguna de estas preguntas. Aun dentro de la escuela ágil hay muchos modelos de proceso propuestos (véase la sección 3.4), cada uno con un enfoque algo diferente para el problema de la agilidad. Dentro de cada modelo hay un conjunto de "ideas" (los agilistas las llaman "tareas del trabajo") que representan un alejamiento significativo de la ingeniería de software tradicional. No obstante, muchos conceptos ágiles sólo son adaptaciones de algunos que provienen de la buena ingeniería de software. En resumen: hay mucho por ganar si se considera lo mejor de ambas escuelas, y virtualmente no se gana nada si se denigra cualquiera de los enfoques.

Si el lector está interesado, consulte [Hig01], [Hig02a] y [DeM02] para ver un resumen ameno de otros aspectos técnicos y políticos importantes.

3.3.3 Factores humanos

Cita:

"Los métodos ágiles obtienen gran parte de su agilidad por basarse en el conocimiento tácito incorporado en el equipo, más que en escribir el conocimiento en planes."

Barry Boehm

Los defensores del desarrollo de software ágil se toman muchas molestias para enfatizar la importancia de los "factores personales". Como dicen Cockburn y Highsmith [Coc01a]: "El desarrollo ágil se centra en los talentos y habilidades de los individuos, y adapta el proceso a personas y equipos específicos." El punto clave de esta afirmación es que *el proceso se adapta a las necesidades de las personas y del equipo*, no al revés.[2]

Si los miembros del equipo de software son los que van a generar las características del proceso que van a aplicarse a la elaboración de software, entre ellos debe existir cierto número de características clave, mismas que debe compartir el equipo ágil como tal:

¿Qué características clave deben existir entre los integrantes de un equipo eficaz de software?

Competencia. En un contexto de desarrollo ágil (así como en la ingeniería de software), la "competencia" incluye el talento innato, las habilidades específicas relacionadas con el software y el conocimiento general del proceso que el equipo haya elegido aplicar. La habilidad y el conocimiento del proceso pueden y deben considerarse para todas las personas que sean miembros ágiles del equipo.

Enfoque común. Aunque los miembros del equipo ágil realicen diferentes tareas y aporten habilidades distintas al proyecto, todos deben centrarse en una meta: entregar al cliente en la fecha prometida un incremento de software que funcione. Para lograrlo, el equipo también se centrará en adaptaciones continuas (pequeñas y grandes) que hagan que el proceso se ajuste a las necesidades del equipo.

Colaboración. La ingeniería de software (sin importar el proceso) trata de evaluar, analizar y usar la información que se comunica al equipo de software; crear información que ayudará a todos los participantes a entender el trabajo del equipo; y generar información (software de cómputo y bases de datos relevantes) que aporten al cliente valor del negocio. Para efectuar estas tareas, los miembros del equipo deben colaborar, entre sí y con todos los participantes.

Cita:

"Lo que para un equipo es apenas suficiente, para otro es más que suficiente y para otro más resulta insuficiente."

Alistair Cockburn

Habilidad para tomar decisiones. Cualquier equipo bueno de software (incluso los equipos ágiles) debe tener libertad para controlar su destino. Esto implica que se dé autonomía al equipo: autoridad para tomar decisiones sobre asuntos tanto técnicos como del proyecto.

Capacidad para resolver problemas difusos. Los gerentes de software deben reconocer que el equipo ágil tendrá que tratar en forma continua con la ambigüedad y que será sacudido de manera permanente por el cambio. En ciertos casos, el equipo debe aceptar el hecho de que el problema que resuelven ahora tal vez no sea el que se necesite resolver mañana. Sin embargo, las lecciones aprendidas de cualquier actividad relacionada con la solución de problemas (incluso aquellas que resuelven el problema equivocado) serán benéficas para el equipo en una etapa posterior del proyecto.

Confianza y respeto mutuos. El equipo ágil debe convertirse en lo que DeMarco y Lister [DeM98] llaman "pegado" (véase el capítulo 24). Un equipo pegado tiene la confianza y respeto que son necesarios para hacer "su tejido tan fuerte que el todo es más que la suma de sus partes" [DeM98].

PUNTO CLAVE

Un equipo con organización propia tiene el control del trabajo que realiza. Establece sus propios compromisos y define los planes para lograrlo.

Organización propia. En el contexto del desarrollo ágil, la organización propia implica tres cosas: 1) el equipo ágil se organiza a sí mismo para hacer el trabajo, 2) el equipo organiza el proceso que se adapte mejor a su ambiente local, 3) el equipo organiza la programación del trabajo a fin de que se logre del mejor modo posible la entrega del incremento

2 Las organizaciones exitosas de ingeniería de software reconocen esta realidad sin importar el modelo de proceso que elijan.

de software. La organización propia tiene cierto número de beneficios técnicos, pero, lo que es más importante, sirve para mejorar la colaboración y elevar la moral del equipo. En esencia, el equipo sirve como su propio gerente. Ken Schwaber [Sch02] aborda estos aspectos cuando escribe: "El equipo selecciona cuánto trabajo cree que puede realizar en cada iteración, y se compromete con la labor. Nada desmotiva tanto a un equipo como que alguien establezca compromisos por él. Nada motiva más a un equipo como aceptar la responsabilidad de cumplir los compromisos que haya hecho él mismo."

3.4 Programación extrema (XP)

A fin de ilustrar un proceso ágil con más detalle, daremos un panorama de la *programación extrema* (XP), el enfoque más utilizado del desarrollo de software ágil. Aunque las primeras actividades con las ideas y los métodos asociados a XP ocurrieron al final de la década de 1980, el trabajo fundamental sobre la materia había sido escrito por Kent Beck [Bec04a]. Una variante de XP llamada *XP industrial* [IXP] se propuso en una época más reciente [Ker05]. IXP mejora la XP y tiene como objetivo el proceso ágil para ser usado específicamente en organizaciones grandes.

3.4.1 Valores XP

Beck [Bec04a] define un conjunto de cinco *valores* que establecen el fundamento para todo trabajo realizado como parte de XP: comunicación, simplicidad, retroalimentación, valentía y respeto. Cada uno de estos valores se usa como un motor para actividades, acciones y tareas específicas de XP.

A fin de lograr la *comunicación* eficaz entre los ingenieros de software y otros participantes (por ejemplo, para establecer las características y funciones requeridas para el software), XP pone el énfasis en la colaboración estrecha pero informal (verbal) entre los clientes y los desarrolladores, en el establecimiento de metáforas[3] para comunicar conceptos importantes, en la retroalimentación continua y en evitar la documentación voluminosa como medio de comunicación.

Mantenlo sencillo siempre que se pueda, pero reconoce que el "rediseño" continuo consume mucho tiempo y recursos.

Para alcanzar la *simplicidad*, XP restringe a los desarrolladores para que diseñen sólo para las necesidades inmediatas, en lugar de considerar las del futuro. El objetivo es crear un diseño sencillo que se implemente con facilidad en forma de código. Si hay que mejorar el diseño, se rediseñará[4] en un momento posterior.

La *retroalimentación* se obtiene de tres fuentes: el software implementado, el cliente y otros miembros del equipo de software. Al diseñar e implementar una estrategia de pruebas eficaz (capítulos 17 a 20), el software (por medio de los resultados de las pruebas) da retroalimentación al equipo ágil. XP usa la *prueba unitaria* como su táctica principal de pruebas. A medida que se desarrolla cada clase, el equipo implementa una prueba unitaria para ejecutar cada operación de acuerdo con su funcionalidad especificada. Cuando se entrega un incremento a un cliente, las *historias del usuario* o *casos de uso* (véase el capítulo 5) que se implementan con el incremento se utilizan como base para las pruebas de aceptación. El grado en el que el software implementa la salida, función y comportamiento del caso de uso es una forma de retroalimentación. Por último, conforme se obtienen nuevos requerimientos como parte de la planeación iterativa, el equipo da al cliente una retroalimentación rápida con miras al costo y al efecto en la programación de actividades.

---

3 En el contexto de XP, una *metáfora* es "una historia que cada quien —clientes, programadores y gerentes— narra, acerca de cómo funciona el sistema" [Bec04a].

4 El rediseño permite que un ingeniero mejore la estructura interna de un diseño (o código fuente) sin cambiar su funcionalidad o comportamiento externos. En esencia, el rediseño puede utilizarse para mejorar la eficiencia, disponibilidad o rendimiento de un diseño o del código que lo implementa.

Cita:

"XP es la respuesta a la pregunta: '¿Cuán pequeño podemos hacer un gran software?'."

Anónimo

Beck [Bec04a] afirma que la adhesión estricta a ciertas prácticas de XP requiere *valentía*. Un término más apropiado sería *disciplina*. Por ejemplo, es frecuente que haya mucha presión para diseñar requerimientos futuros. La mayor parte de equipos de software sucumben a ella y se justifican porque "diseñar para el mañana" ahorrará tiempo y esfuerzo en el largo plazo. Un equipo XP ágil debe tener la disciplina (valentía) para diseñar para hoy y reconocer que los requerimientos futuros tal vez cambien mucho, por lo que demandarán repeticiones sustanciales del diseño y del código implementado.

Al apegarse a cada uno de estos valores, el equipo ágil inculca *respeto* entre sus miembros, entre otros participantes y los integrantes del equipo, e indirectamente para el software en sí mismo. Conforme logra la entrega exitosa de incrementos de software, el equipo desarrolla más respeto para el proceso XP.

3.4.2 El proceso XP

WebRef

En la dirección **www.extremeprogramming.org/rules.html**, se encuentra un panorama excelente de las "reglas" de XP.

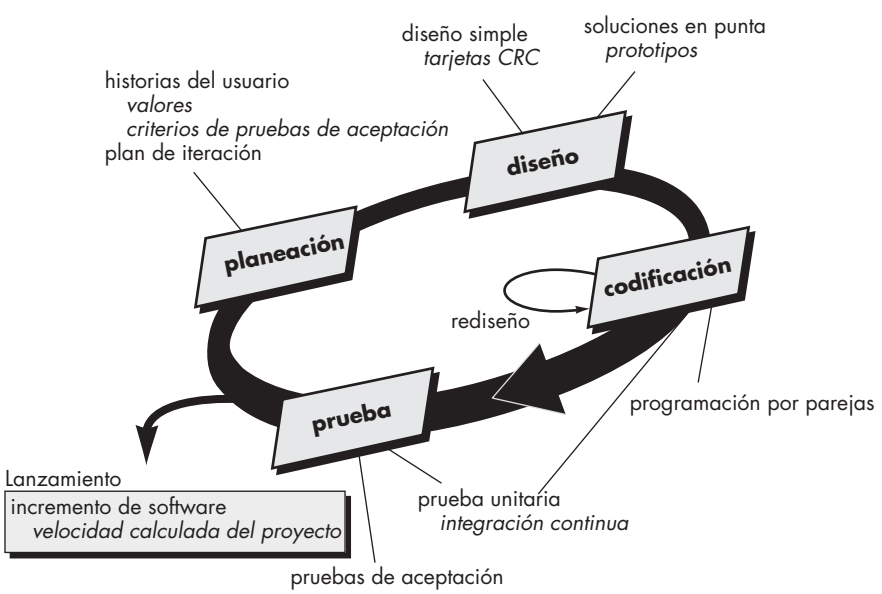
La programación extrema usa un enfoque orientado a objetos (véase el apéndice 2) como paradigma preferido de desarrollo, y engloba un conjunto de reglas y prácticas que ocurren en el contexto de cuatro actividades estructurales: planeación, diseño, codificación y pruebas. La figura 3.2 ilustra el proceso XP y resalta algunas de las ideas y tareas clave que se asocian con cada actividad estructural. En los párrafos que siguen se resumen las actividades de XP clave.

Planeación. La actividad de planeación (también llamada *juego de planeación*) comienza *escuchando* —actividad para recabar requerimientos que permite que los miembros técnicos del equipo XP entiendan el contexto del negocio para el software y adquieran la sensibilidad de la salida y características principales y funcionalidad que se requieren—. Escuchar lleva a la creación de algunas "historias" (también llamadas *historias del usuario*) que describen la salida necesaria, características y funcionalidad del software que se va a elaborar. Cada *historia* (similar a los casos de uso descritos en el capítulo 5) es escrita por el cliente y colocada en una tarjeta indizada. El cliente asigna un *valor* (es decir, una prioridad) a la historia con base en el valor general de la característica o función para el negocio.[5] Después, los miembros del equipo XP

? ¿Qué es una "historia" XP?

FIGURA 3.2

El proceso de la programación extrema



5 El valor de una historia también puede depender de la presencia de otra historia.

evalúan cada historia y le asignan un costo, medido en semanas de desarrollo. Si se estima que la historia requiere más de tres semanas de desarrollo, se pide al cliente que la descomponga en historias más chicas y de nuevo se asigna un valor y costo. Es importante observar que en cualquier momento es posible escribir nuevas historias.

WebRef
En la dirección **c2.com/cgi/wiki?planningGame**, se halla un "juego de planeación" XP provechoso.

Los clientes y desarrolladores trabajan juntos para decidir cómo agrupar las historias en la siguiente entrega (el siguiente incremento de software) que desarrollará el equipo XP. Una vez que se llega a un *compromiso* sobre la entrega (acuerdo sobre las historias por incluir, la fecha de entrega y otros aspectos del proyecto), el equipo XP ordena las historias que serán desarrolladas en una de tres formas: 1) todas las historias se implementarán de inmediato (en pocas semanas), 2) las historias con más valor entrarán a la programación de actividades y se implementarán en primer lugar o 3) las historias más riesgosas formarán parte de la programación de actividades y se implementarán primero.

La velocidad del proyecto es una medición sutil de la productividad del equipo.

Después de la primera entrega del proyecto (también llamada incremento de software), el equipo XP calcula la velocidad de éste. En pocas palabras, la *velocidad del proyecto* es el número de historias de los clientes implementadas durante la primera entrega. La velocidad del proyecto se usa para: 1) ayudar a estimar las fechas de entrega y programar las actividades para las entregas posteriores, y 2) determinar si se ha hecho un gran compromiso para todas las historias durante todo el desarrollo del proyecto. Si esto ocurre, se modifica el contenido de las entregas o se cambian las fechas de entrega final.

A medida que avanza el trabajo, el cliente puede agregar historias, cambiar el valor de una ya existente, descomponerlas o eliminarlas. Entonces, el equipo XP reconsidera todas las entregas faltantes y modifica sus planes en consecuencia.

Diseño. El diseño XP sigue rigurosamente el principio MS (mantenlo sencillo). Un diseño sencillo siempre se prefiere sobre una representación más compleja. Además, el diseño guía la implementación de una historia conforme se escribe: nada más y nada menos. Se desalienta el diseño de funcionalidad adicional porque el desarrollador supone que se requerirá después.[6]

XP desalienta la importancia del diseño, opinión con la que no todos están de acuerdo. En realidad, hay veces en las que debe hacerse énfasis en el diseño.

XP estimula el uso de las tarjetas CRC (véase el capítulo 7) como un mecanismo eficaz para pensar en el software en un contexto orientado a objetos. Las tarjetas CRC (clase-responsabilidad-colaborador) identifican y organizan las clases orientadas a objetos[7] que son relevantes para el incremento actual de software. El equipo XP dirige el ejercicio de diseño con el uso de un proceso similar al que se describe en el capítulo 8. Las tarjetas CRC son el único producto del trabajo de diseño que se genera como parte del proceso XP.

Si en el diseño de una historia se encuentra un problema de diseño difícil, XP recomienda la creación inmediata de un prototipo operativo de esa porción del diseño. Entonces, se implementa y evalúa el prototipo del diseño, llamado *solución en punta*. El objetivo es disminuir el riesgo cuando comience la implementación verdadera y validar las estimaciones originales para la historia que contiene el problema de diseño.

WebRef
En la dirección **www.refactoring.com** se encuentran técnicas y herramientas de rediseño.

En la sección anterior se dijo que XP estimula el *rediseño*, técnica de construcción que también es un método para la optimización del diseño. Fowler [Fow00] describe el rediseño del modo siguiente:

> *Rediseño* es el proceso mediante el cual se cambia un sistema de software en forma tal que no altere el comportamiento externo del código, pero sí mejore la estructura interna. Es una manera disciplinada de limpiar el código [y modificar o simplificar el diseño interno] que minimiza la probabilidad de introducir errores. En esencia, cuando se rediseña, se mejora el diseño del código después de haber sido escrito.

---

6 Estos lineamientos de diseño deben seguirse en todo método de ingeniería de software, aunque hay ocasiones en los que la notación y terminología sofisticadas del diseño son un camino hacia la simplicidad.

7 Las clases orientadas a objetos se estudian en el apéndice 2, en el capítulo 8 y en toda la parte 2 de este libro.

PUNTO CLAVE

El rediseño mejora la estructura interna de un diseño (o código fuente) sin cambiar su funcionalidad o comportamiento externo.

Como el diseño XP virtualmente no utiliza notación y genera pocos, si alguno, productos del trabajo que no sean tarjetas CRC y soluciones en punta, el diseño es visto como un artefacto en transición que puede y debe modificarse continuamente a medida que avanza la construcción. El objetivo del rediseño es controlar dichas modificaciones, sugiriendo pequeños cambios en el diseño que "son capaces de mejorarlo en forma radical" [Fow00]. Sin embargo, debe notarse que el esfuerzo que requiere el rediseño aumenta en forma notable con el tamaño de la aplicación.

Un concepto central en XP es que el diseño ocurre tanto antes *como después* de que comienza la codificación. Rediseñar significa que el diseño se hace de manera continua conforme se construye el sistema. En realidad, la actividad de construcción en sí misma dará al equipo XP una guía para mejorar el diseño.

Codificación. Después de que las historias han sido desarrolladas y de que se ha hecho el trabajo de diseño preliminar, el equipo *no* inicia la codificación, sino que desarrolla una serie de pruebas unitarias a cada una de las historias que se van a incluir en la entrega en curso (incremento de software).[8] Una vez creada la prueba unitaria,[9] el desarrollador está mejor capacitado para centrarse en lo que debe implementarse para pasar la prueba. No se agrega nada extraño (MS). Una vez que el código está terminado, se le aplica de inmediato una prueba unitaria, con lo que se obtiene retroalimentación instantánea para los desarrolladores.

WebRef

Hay información útil acerca de XP en la dirección **www.xprogramming.com.**

¿Qué es la programación por parejas?

Un concepto clave durante la actividad de codificación (y uno de los aspectos del que más se habla en la XP) es la *programación por parejas*. XP recomienda que dos personas trabajen juntas en una estación de trabajo con el objeto de crear código para una historia. Esto da un mecanismo para la solución de problemas en tiempo real (es frecuente que dos cabezas piensen más que una) y para el aseguramiento de la calidad también en tiempo real (el código se revisa conforme se crea). También mantiene a los desarrolladores centrados en el problema de que se trate. En la práctica, cada persona adopta un papel un poco diferente. Por ejemplo, una de ellas tal vez piense en los detalles del código de una porción particular del diseño, mientras la otra se asegura de que se siguen los estándares de codificación (parte necesaria de XP) o de que el código para la historia satisfará la prueba unitaria desarrollada a fin de validar el código confrontándolo con la historia.

CONSEJO

Muchos equipos de software están llenos de individualistas. Si la programación por parejas ha de funcionar con eficacia, tendrá que trabajar para cambiar esa cultura.

A medida que las parejas de programadores terminan su trabajo, el código que desarrollan se integra con el trabajo de los demás. En ciertos casos, esto lo lleva a cabo a diario un equipo de integración. En otros, las parejas de programadores tienen la responsabilidad de la integración. Esta estrategia de "integración continua" ayuda a evitar los problemas de compatibilidad e interfaces y brinda un ambiente de "prueba de humo" (véase el capítulo 17) que ayuda a descubrir a tiempo los errores.

¿Cómo se usan las pruebas unitarias en XP?

Pruebas. Ya se dijo que la creación de pruebas unitarias antes de que comience la codificación es un elemento clave del enfoque de XP. Las pruebas unitarias que se crean deben implementarse con el uso de una estructura que permita automatizarlas (de modo que puedan ejecutarse en repetidas veces y con facilidad). Esto estimula una estrategia de pruebas de regresión (véase el capítulo 17) siempre que se modifique el código (lo que ocurre con frecuencia, dada la filosofía del rediseño en XP).

A medida que se organizan las pruebas unitarias individuales en un "grupo de prueba universal" [Wel99], las pruebas de la integración y validación del sistema pueden efectuarse a diario. Esto da al equipo XP una indicación continua del avance y también lanza señales de alerta si las

8 Este enfoque es equivalente a saber las preguntas del examen antes de comenzar a estudiar. Vuelve mucho más fácil el estudio porque centra la atención sólo en las preguntas que se van a responder.

9 La prueba unitaria, que se estudia en detalle en el capítulo 17, se centra en un componente de software individual sobre interfaz, estructuras de datos y funcionalidad del componente, en un esfuerzo por descubrir errores locales del componente.

cosas marchan mal. Wells [Wel99] dice: "Corregir pequeños problemas cada cierto número de horas toma menos tiempo que resolver problemas enormes justo antes del plazo final."

PUNTO CLAVE
Las pruebas de aceptación se derivan de las historias de los usuarios.

Las *pruebas de aceptación* XP, también llamadas *pruebas del cliente*, son especificadas por el cliente y se centran en las características y funcionalidad generales del sistema que son visibles y revisables por parte del cliente. Las pruebas de aceptación se derivan de las historias de los usuarios que se han implementado como parte de la liberación del software.

3.4.3 XP industrial

Joshua Kerievsky [Ker05] describe la *programación extrema industrial* [IXP, por sus siglas en inglés] en la forma siguiente: "IXP es la evolución orgánica de XP. Está imbuida del espíritu minimalista, centrado en el cliente y orientado a las pruebas que tiene XP. IXP difiere sobre todo de la XP original en su mayor inclusión de la gerencia, el papel más amplio de los clientes y en sus prácticas técnicas actualizadas". IXP incorpora seis prácticas nuevas diseñadas para ayudar a garantizar que un proyecto XP funciona con éxito para proyectos significativos dentro de una organización grande.

? ¿Qué nuevas prácticas se agregan a XP para crear IXP?

Evaluación de la factibilidad. Antes de iniciar un proyecto IXP, la organización debe efectuar una *evaluación de la factibilidad*. Ésta deja en claro si: 1) existe un ambiente apropiado de desarrollo que acepte IXP, 2) el equipo estará constituido por los participantes adecuados, 3) la organización tiene un programa de calidad distintivo y apoya la mejora continua, 4) la cultura organizacional apoyará los nuevos valores de un equipo ágil, y 5) la comunidad extendida del proyecto estará constituida de modo apropiado.

Cita:

"Habilidad es aquello que eres capaz de hacer. La motivación determina lo que haces. La actitud determina cuán bien lo haces."

Lou Holtz

Comunidad del proyecto. La XP clásica sugiere que se utilice personal apropiado para formar el equipo ágil a fin de asegurar el éxito. La implicación es que las personas en el equipo deben estar bien capacitadas, ser adaptables y hábiles, y tener el temperamento apropiado para contribuir al equipo con organización propia. Cuando se aplica XP a un proyecto significativo en una organización grande, el concepto de "equipo" debe adoptar la forma de *comunidad*. Una comunidad puede tener un tecnólogo y clientes que son fundamentales para el éxito del proyecto, así como muchos otros participantes (equipo jurídico; auditores de calidad, de tipos de manufactura o de ventas, etc.) que "con frecuencia se encuentran en la periferia en un proyecto IXP, pero que desempeñan en éste papeles importantes" [Ker05]. En IXP, los miembros de la comunidad y sus papeles deben definirse de modo explícito, así como establecer los mecanismos para la comunicación y coordinación entre los integrantes de la comunidad.

Calificación del proyecto. El equipo de IXP evalúa el proyecto para determinar si existe una justificación apropiada de negocios y si el proyecto cumplirá las metas y objetivos generales de la organización. La calificación también analiza el contexto del proyecto a fin de determinar cómo complementa, extiende o reemplaza sistemas o procesos existentes.

Administración orientada a pruebas. Un proyecto IXP requiere criterios medibles para evaluar el estado del proyecto y el avance realizado. La administración orientada a pruebas establece una serie de "destinos" medibles [Ker05] y luego define los mecanismos para determinar si se han alcanzado o no éstos.

Retrospectivas. Después de entregar un incremento de software, el equipo XP realiza una revisión técnica especializada que se llama *retrospectiva* y que examina "los temas, eventos y lecciones aprendidas" [Ker05] a lo largo del incremento de software y/o de la liberación de todo el software. El objetivo es mejorar el proceso IXP.

Aprendizaje continuo. Como el aprendizaje es una parte vital del proceso de mejora continua, los miembros del equipo XP son invitados (y tal vez incentivados) a aprender nuevos métodos y técnicas que conduzcan a una calidad más alta del producto.

Además de las seis nuevas prácticas analizadas, IXP modifica algunas de las existentes en XP. El *desarrollo impulsado por la historia* (DIH) insiste en que las historias de las pruebas de aceptación se escriban antes de generar una sola línea de código. El *diseño impulsado por el dominio* (DID) es una mejora sobre el concepto de la "metáfora del sistema" usado en XP. El DID [Eva03] sugiere la creación evolutiva de un modelo de dominio que "represente con exactitud cómo piensan los expertos del dominio en su materia" [Ker05]. La *formación de parejas* amplía el concepto de programación en pareja para que incluya a los gerentes y a otros participantes. El objetivo es mejorar la manera de compartir conocimientos entre los integrantes del equipo XP que no estén directamente involucrados en el desarrollo técnico. La *usabilidad iterativa* desalienta el diseño de una interfaz cargada al frente y estimula un diseño que evoluciona a medida que se liberan los incrementos de software y que se estudia la interacción de los usuarios con el software.

La IXP hace modificaciones más pequeñas a otras prácticas XP y redefine ciertos roles y responsabilidades para hacerlos más asequibles a proyectos significativos de las organizaciones grandes. Para mayores detalles de IXP, visite el sitio **http://industrialxp.org**.

3.4.4 El debate XP

Los nuevos modelos y métodos de proceso han motivado análisis provechosos y en ciertas instancias debates acalorados. La programación extrema desencadena ambos. En un libro interesante que examina la eficacia de XP, Stephens y Rosenberg [Ste03] afirman que muchas prácticas de XP son benéficas, pero que otras están sobreestimadas y unas más son problemáticas. Los autores sugieren que la naturaleza codependiente de las prácticas de XP constituye tanto su fortaleza como su debilidad. Debido a que muchas organizaciones adoptan sólo un subconjunto de prácticas XP, debilitan la eficacia de todo el proceso. Los defensores contradicen esto al afirmar que la XP está en evolución continua y que muchas de las críticas que se le hacen han llevado a correcciones conforme maduran sus prácticas. Entre los aspectos que destacan algunos críticos de la XP están los siguientes:[10]

? ¿Cuáles son algunos de los aspectos que llevan al debate de XP?

- *Volatilidad de los requerimientos.* Como el cliente es un miembro activo del equipo XP, los cambios a los requerimientos se solicitan de manera informal. En consecuencia, el alcance del proyecto cambia y el trabajo inicial tiene que modificarse para dar acomodo a las nuevas necesidades. Los defensores afirman que esto pasa sin importar el proceso que se aplique y que la XP proporciona mecanismos para controlar los vaivenes del alcance.
- *Necesidades conflictivas del cliente.* Muchos proyectos tienen clientes múltiples, cada uno con sus propias necesidades. En XP, el equipo mismo tiene la tarea de asimilar las necesidades de distintos clientes, trabajo que tal vez esté más allá del alcance de su autoridad.
- *Los requerimientos se expresan informalmente.* Las historias de usuario y las pruebas de aceptación son la única manifestación explícita de los requerimientos en XP. Los críticos afirman que es frecuente que se necesite un modelo o especificación más formal para garantizar que se descubran las omisiones, inconsistencias y errores antes de que se construya el sistema. Los defensores contraatacan diciendo que la naturaleza cambiante de los requerimientos vuelve obsoletos esos modelos y especificaciones casi tan pronto como se desarrollan.
- *Falta de un diseño formal.* XP desalienta la necesidad del diseño de la arquitectura y, en muchas instancias, sugiere que el diseño de todas las clases debe ser relativamente informal. Los críticos argumentan que cuando se construyen sistemas complejos, debe ponerse el énfasis en el diseño con el objeto de garantizar que la estructura general del software tendrá calidad y que será susceptible de recibir mantenimiento. Los defensores

10 Para un estudio más detallado de ciertas críticas profundas hechas a XP, visite www.softwarereality.com/ExtremeProgramming.jsp.

de XP sugieren que la naturaleza incremental del proceso XP limita la complejidad (la sencillez es un valor fundamental), lo que reduce la necesidad de un diseño extenso.

El lector debe observar que todo proceso del software tiene sus desventajas, y que muchas organizaciones de software han utilizado con éxito la XP. La clave es identificar dónde tiene sus debilidades un proceso y adaptarlo a las necesidades de la organización.

3.5 Otros modelos ágiles de proceso

Cita:

"Nuestra profesión pasa por las metodologías como un chico de 14 años pasa por la ropa."

Stephen Hawrysh y Jim Ruprecht

La historia de la ingeniería de software está salpicada de decenas de descripciones y metodologías de proceso, métodos de modelado y notaciones, herramientas y tecnología, todos ellos obsoletos. Cada uno tuvo notoriedad y luego fue eclipsado por algo nuevo y (supuestamente) mejor. Con la introducción de una amplia variedad de modelos ágiles del proceso —cada uno en lucha por la aceptación de la comunidad de desarrollo de software— el movimiento ágil está siguiendo la misma ruta histórica.[11]

CasaSegura

Consideración del desarrollo ágil de software

La escena: Oficina de Doug Miller.

Participantes: Doug Miller, gerente de ingeniería de software; Jamie Lazar, miembro del equipo de software; Vinod Raman, integrante del equipo de software.

La conversación:

(Tocan a la puerta; Jamie y Vinod entran a la oficina de Doug.)

Jamie: Doug, ¿tienes un minuto?

Doug: Seguro, Jamie, ¿qué pasa?

Jamie: Hemos estado pensando en nuestra conversación de ayer sobre el proceso… ya sabes, el que vamos a elegir para este nuevo proyecto de *CasaSegura*.

Doug: ¿Y?

Vinod: Hablé con un amigo de otra compañía, y me contó sobre la programación extrema. Es un modelo de proceso ágil… ¿has oído de él?

Doug: Sí, algunas cosas buenas y otras malas.

Jamie: Bueno, a nosotros nos pareció bien. Permite el desarrollo de software realmente rápido, usa algo llamado programación en parejas para revisar la calidad en tiempo real… Pienso que es muy bueno.

Doug: Tiene muchas ideas realmente buenas. Por ejemplo, me gusta el concepto de programación en parejas y la idea de que los participantes deben formar parte del equipo.

Jamie: ¿Qué? ¿Quieren decir que mercadotecnia trabajará con nosotros en el proyecto?

Doug (afirmando con la cabeza): Ellos son uno de los participantes, ¿o no?

Jamie: Sí… Pedirán cambios cada cinco minutos.

Vinod: No necesariamente. Mi amigo dijo que hay formas de "adoptar" los cambios durante un proyecto de XP.

Doug: Entonces, chicos, ¿piensan que debemos usar XP?

Jamie: Definitivamente, sería bueno considerarlo.

Doug: Estoy de acuerdo. E incluso si elegimos un modelo incremental como nuestro enfoque, no hay razón para no incorporar mucho de lo que XP tiene que ofrecer.

Vinod: Doug, dijiste hace un rato "cosas buenas y malas". ¿Cuáles son las malas?

Doug: Lo que no me gusta es la forma en la que XP desprecia el análisis y el diseño… algo así como decir que la escritura del código está donde hay acción…

(Los miembros del equipo se miran entre sí y sonríen.)

Doug: Entonces, ¿están de acuerdo con el enfoque XP?

Jamie (habla por ambos): ¡Escribir código es lo que hacemos, jefe!

Doug (ríe): Es cierto, pero me gustaría ver que dediquen un poco menos de tiempo a escribir código y luego a repetirlo, y que pasen algo más de tiempo en el análisis de lo que debe hacerse para diseñar una solución que funcione.

Vinod: Tal vez tengamos las dos cosas, agilidad con un poco de disciplina.

Doug: Creo que podemos, Vinod. En realidad, estoy seguro de que se puede.

11 Esto no es algo malo. Antes de que uno o varios modelos se acepten como el estándar *de facto*, todos deben luchar por conquistar las mentes y corazones de los ingenieros de software. Los "ganadores" evolucionan hacia las mejores prácticas, mientras que los "perdedores" desaparecen o se funden con los modelos ganadores.

Como se dijo en la sección anterior, el más usado de todos los modelos ágiles de proceso es la programación extrema (XP). Pero se han propuesto muchos otros y están en uso en toda la industria. Entre ellos se encuentran los siguientes:

- Desarrollo adaptativo de software (DAS)
- Scrum
- Método de desarrollo de sistemas dinámicos (MDSD)
- Cristal
- Desarrollo impulsado por las características (DIC)
- Desarrollo esbelto de software (DES)
- Modelado ágil (MA)
- Proceso unificado ágil (PUA)

En las secciones que siguen se presenta un panorama muy breve de cada uno de estos modelos ágiles del proceso. Es importante notar que *todos* los modelos de proceso ágil se apegan (en mayor o menor grado) al *Manifiesto para el desarrollo ágil de software* y a los principios descritos en la sección 3.3.1. Para mayores detalles, consulte las referencias mencionadas en cada subsección o ingrese en la entrada "desarrollo de software ágil" de Wikipedia.[12]

3.5.1 Desarrollo adaptativo de software (DAS)

WebRef

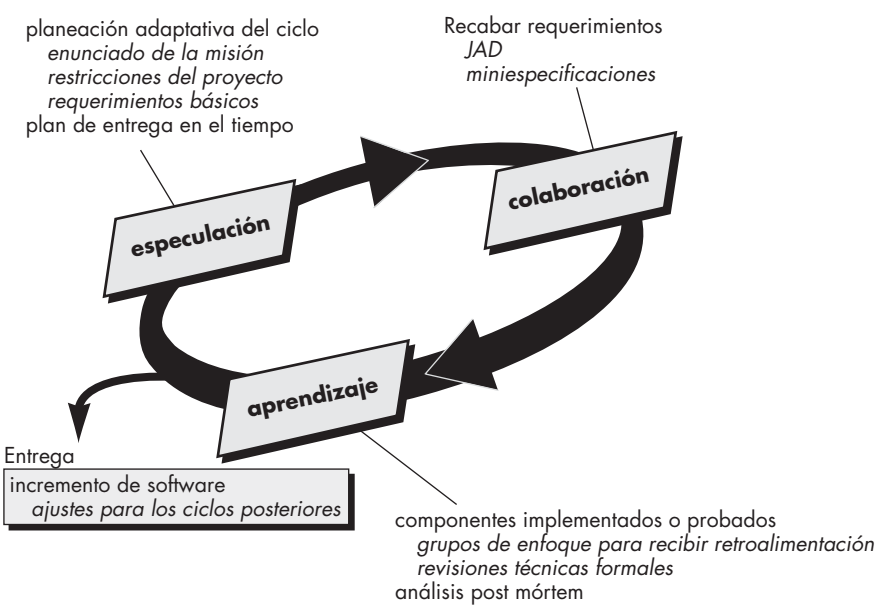
En la dirección **www.adaptivesd.com** hay referencias útiles sobre el DAS.

El *desarrollo adaptativo de software* (DAS) fue propuesto por Jim Highsmith [Hig00] como una técnica para elaborar software y sistemas complejos. Los fundamentos filosóficos del DAS se centran en la colaboración humana y en la organización propia del equipo.

Highsmith argumenta que un enfoque de desarrollo adaptativo basado en la colaboración es "tanto una fuente de *orden* en nuestras complejas interacciones, como de disciplina e ingeniería". Él define un "ciclo de vida" del DAS (véase la figura 3.3) que incorpora tres fases: especulación, colaboración y aprendizaje.

FIGURA 3.3

Desarrollo adaptativo de software



12 Consulte http://en.wikipedia.org/wiki/Agile_software_development#Agile_methods.

Durante la *especulación*, se inicia el proyecto y se lleva a cabo la *planeación adaptativa del ciclo*. La especulación emplea la información de inicio del proyecto —enunciado de misión de los clientes, restricciones del proyecto (por ejemplo, fechas de entrega o descripciones de usuario) y requerimientos básicos— para definir el conjunto de ciclos de entrega (incrementos de software) que se requerirán para el proyecto.

La colaboración eficaz con el cliente sólo ocurrirá si evita cualquier actitud del tipo "nosotros y ellos".

No importa lo completo y previsor que sea el plan del ciclo, será inevitable que cambie. Con base en la información obtenida al terminar el primer ciclo, el plan se revisa y se ajusta, de modo que el trabajo planeado se acomode mejor a la realidad en la que trabaja el equipo DAS.

Las personas motivadas usan la *colaboración* de manera que multiplica su talento y producción creativa más allá de sus números absolutos. Este enfoque es un tema recurrente en todos los métodos ágiles. Sin embargo, la colaboración no es fácil. Incluye la comunicación y el trabajo en equipo, pero también resalta el individualismo porque la creatividad individual desempeña un papel importante en el pensamiento colaborativo. Es cuestión, sobre todo, de confianza. Las personas que trabajan juntas deben confiar una en otra a fin de: 1) criticarse sin enojo, 2) ayudarse sin resentimiento, 3) trabajar tan duro, o más, que como de costumbre, 4) tener el conjunto de aptitudes para contribuir al trabajo, y 5) comunicar los problemas o preocupaciones de manera que conduzcan a la acción efectiva.

El DAS pone el énfasis en el aprendizaje como elemento clave para lograr un equipo con "organización propia".

Conforme los miembros de un equipo DAS comienzan a desarrollar los componentes que forman parte de un ciclo adaptativo, el énfasis se traslada al "aprendizaje" de todo lo que hay en el avance hacia la terminación del ciclo. En realidad, Highsmith [Hig00] afirma que los desarrolladores de software sobreestiman con frecuencia su propia comprensión (de la tecnología, del proceso y del proyecto) y que el aprendizaje los ayudará a mejorar su nivel de entendimiento real. Los equipos DAS aprenden de tres maneras: grupos de enfoque (véase el capítulo 5), revisiones técnicas (véase el capítulo 14) y análisis post mórtem del proyecto.

La filosofía DAS tiene un mérito, sin importar el modelo de proceso que se use. El énfasis general que hace el DAS en la dinámica de los equipos con organización propia, la colaboración interpersonal y el aprendizaje individual y del equipo generan equipos para proyectos de software que tienen una probabilidad de éxito mucho mayor.

3.5.2 Scrum

Scrum (nombre que proviene de cierta jugada que tiene lugar durante un partido de rugby)[13] es un método de desarrollo ágil de software concebido por Jeff Sutherland y su equipo de desarrollo a principios de la década de 1990. En años recientes, Schwaber y Beedle [Sch01a] han desarrollado más los métodos Scrum.

WebRef

En la dirección **www.controlchaos.com** hay información útil sobre Scrum.

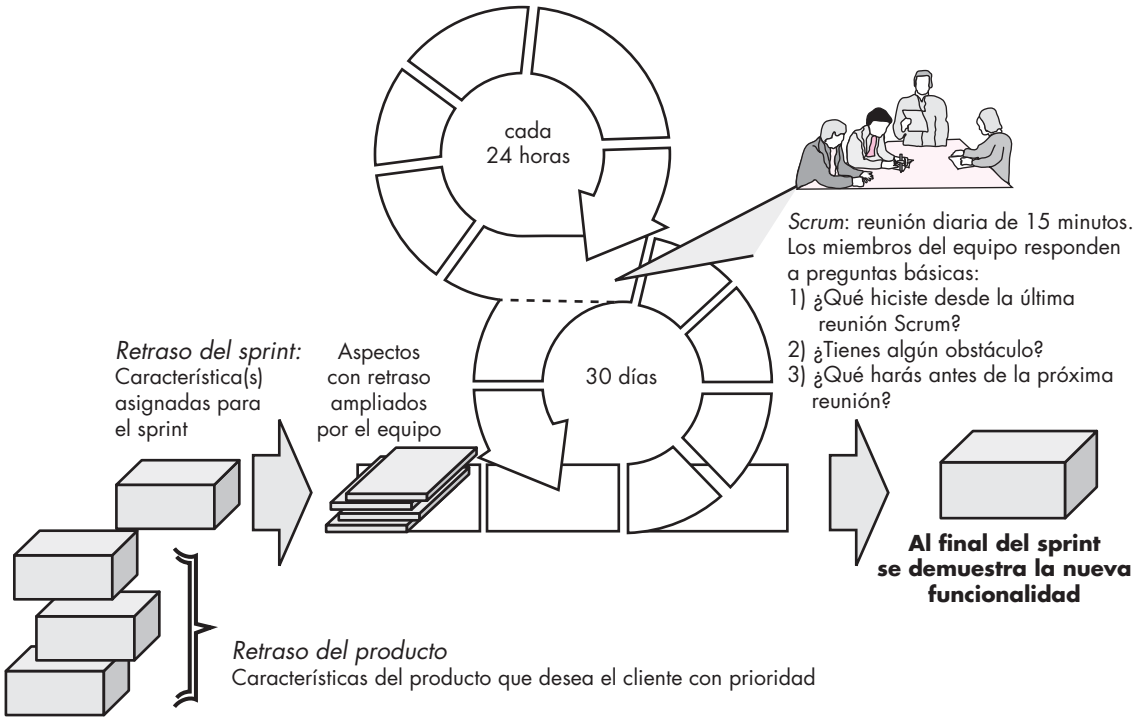
Los principios Scrum son congruentes con el manifiesto ágil y se utilizan para guiar actividades de desarrollo dentro de un proceso de análisis que incorpora las siguientes actividades estructurales: requerimientos, análisis, diseño, evolución y entrega. Dentro de cada actividad estructural, las tareas del trabajo ocurren con un patrón del proceso (que se estudia en el párrafo siguiente) llamado *sprint*. El trabajo realizado dentro de un sprint (el número de éstos que requiere cada actividad estructural variará en función de la complejidad y tamaño del producto) se adapta al problema en cuestión y se define —y con frecuencia se modifica— en tiempo real por parte del equipo Scrum. El flujo general del proceso Scrum se ilustra en la figura 3.4.

Scrum incorpora un conjunto de patrones del proceso que ponen el énfasis en las prioridades del proyecto, las unidades de trabajo agrupadas, la comunicación y la retroalimentación frecuente con el cliente.

Scrum acentúa el uso de un conjunto de patrones de proceso del software [Noy02] que han demostrado ser eficaces para proyectos con plazos de entrega muy apretados, requerimientos cambiantes y negocios críticos. Cada uno de estos patrones de proceso define un grupo de acciones de desarrollo:

13 Se forma un grupo de jugadores alrededor del balón y todos trabajan juntos (a veces con violencia) para moverlo a través del campo.

FIGURA 3.4
Flujo del proceso Scrum



Retraso: lista de prioridades de los requerimientos o características del proyecto que dan al cliente un valor del negocio. Es posible agregar en cualquier momento otros aspectos al retraso (ésta es la forma en la que se introducen los cambios). El gerente del proyecto evalúa el retraso y actualiza las prioridades según se requiera.

Sprints: consiste en unidades de trabajo que se necesitan para alcanzar un requerimiento definido en el retraso que debe ajustarse en una caja de tiempo[14] predefinida (lo común son 30 días). Durante el sprint no se introducen cambios (por ejemplo, aspectos del trabajo retrasado). Así, el sprint permite a los miembros del equipo trabajar en un ambiente de corto plazo pero estable.

Reuniones Scrum: son reuniones breves (de 15 minutos, por lo general) que el equipo Scrum efectúa a diario. Hay tres preguntas clave que se pide que respondan todos los miembros del equipo [Noy02]:

- ¿Qué hiciste desde la última reunión del equipo?
- ¿Qué obstáculos estás encontrando?
- ¿Qué planeas hacer mientras llega la siguiente reunión del equipo?

Un líder del equipo, llamado *maestro Scrum*, dirige la junta y evalúa las respuestas de cada persona. La junta Scrum ayuda al equipo a descubrir los problemas potenciales tan pronto como sea posible. Asimismo, estas juntas diarias llevan a la "socialización del conocimiento" [Bee99], con lo que se promueve una estructura de equipo con organización propia.

Demostraciones preliminares: entregar el incremento de software al cliente de modo que la funcionalidad que se haya implementado pueda demostrarse al cliente y éste pueda evaluarla.

14 Una *caja de tiempo* es un término de la administración de proyectos (véase la parte 4 de este libro) que indica el tiempo que se ha asignado para cumplir alguna tarea.

Es importante notar que las demostraciones preliminares no contienen toda la funcionalidad planeada, sino que éstas se entregarán dentro de la caja de tiempo establecida.

Beedle y sus colegas [Bee99] presentan un análisis exhaustivo de estos patrones en el que dicen: "Scrum supone de entrada la existencia de caos…" Los patrones de proceso Scrum permiten que un equipo de software trabaje con éxito en un mundo en el que es imposible eliminar la incertidumbre.

3.5.3 Método de desarrollo de sistemas dinámicos (MDSD)

WebRef

En la dirección **www.dsdm.org** hay recursos útiles para el MDSD.

El *método de desarrollo de sistemas dinámicos* (MDSD) [Sta97] es un enfoque de desarrollo ágil de software que "proporciona una estructura para construir y dar mantenimiento a sistemas que cumplan restricciones apretadas de tiempo mediante la realización de prototipos incrementales en un ambiente controlado de proyectos" [CCS02]. La filosofía MDSD está tomada de una versión modificada de la regla de Pareto: 80 por ciento de una aplicación puede entregarse en 20 por ciento del tiempo que tomaría entregarla completa (100 por ciento).

El MDSD es un proceso iterativo de software en el que cada iteración sigue la regla de 80 por ciento. Es decir, se requiere sólo suficiente trabajo para cada incremento con objeto de facilitar el paso al siguiente. Los detalles restantes se terminan más tarde, cuando se conocen los requerimientos del negocio y se han pedido y efectuado cambios.

El grupo DSDM Consortium (**www.dsdm.org**) es un conglomerado mundial de compañías que adoptan colectivamente el papel de "custodios" del método. El consorcio ha definido un modelo de proceso ágil, llamado *ciclo de vida MDSD*, que define tres ciclos iterativos distintos, precedidos de dos actividades adicionales al ciclo de vida:

Estudio de factibilidad: establece los requerimientos y restricciones básicas del negocio, asociados con la aplicación que se va a construir, para luego evaluar si la aplicación es un candidato viable para aplicarle el proceso MDSD.

El MDSD es una estructura de proceso que adopta las tácticas de otro enfoque ágil, como XP.

Estudio del negocio: establece los requerimientos e información funcionales que permitirán la aplicación para dar valor al negocio; asimismo, define la arquitectura básica de la aplicación e identifica los requerimientos para darle mantenimiento.

Iteración del modelo funcional: produce un conjunto de prototipos incrementales que demuestran al cliente la funcionalidad. (Nota: todos los prototipos de MDSD están pensados para que evolucionen hacia la aplicación que se entrega.) El objetivo de este ciclo iterativo es recabar requerimientos adicionales por medio de la obtención de retroalimentación de los usuarios cuando practican con el prototipo.

Diseño e iteración de la construcción: revisita los prototipos construidos durante la *iteración del modelo funcional* a fin de garantizar que en cada iteración se ha hecho ingeniería en forma que permita dar valor operativo del negocio a los usuarios finales; la *iteración del modelo funcional* y el *diseño e iteración de la construcción* ocurren de manera concurrente.

Implementación: coloca el incremento más reciente del software (un prototipo "operacional") en el ambiente de operación. Debe notarse que: 1) el incremento tal vez no sea el de 100% final, o 2) quizá se pidan cambios cuando el incremento se ponga en su lugar. En cualquier caso, el trabajo de desarrollo MDSD continúa y vuelve a la actividad de iteración del modelo funcional.

El MDSD se combina con XP (véase la sección 3.4) para dar un enfoque de combinación que define un modelo sólido del proceso (ciclo de vida MDSD) con las prácticas detalladas (XP) que se requieren para elaborar incrementos de software. Además, los conceptos DAS se adaptan a un modelo combinado del proceso.

3.5.4 Cristal

Alistar Cockburn [Coc05] creó la *familia Cristal de métodos ágiles*[15] a fin de obtener un enfoque de desarrollo de software que premia la "maniobrabilidad" durante lo que Cockburn caracteriza como "un juego cooperativo con recursos limitados, de invención y comunicación, con el objetivo primario de entregar software útil que funcione y con la meta secundaria de plantear el siguiente juego" [Coc02].

PUNTO CLAVE

Cristal es una familia de modelos de proceso con el mismo "código genético" pero diferentes métodos para adaptarse a las características del proyecto.

Para lograr la maniobrabilidad, Cockburn y Highsmith definieron un conjunto de metodologías, cada una con elementos fundamentales comunes a todos, y roles, patrones de proceso, producto del trabajo y prácticas que son únicas para cada uno. La familia Cristal en realidad es un conjunto de ejemplos de procesos ágiles que han demostrado ser efectivos para diferentes tipos de proyectos. El objetivo es permitir que equipos ágiles seleccionen al miembro de la familia Cristal más apropiado para su proyecto y ambiente.

3.5.5 Desarrollo impulsado por las características (DIC)

El *desarrollo impulsado por las características* (DIC) lo concibió originalmente Peter Coad y sus colegas [Coa99] como modelo práctico de proceso para la ingeniería de software orientada a objetos. Stephen Palmer y John Felsing [Pal02] ampliaron y mejoraron el trabajo de Coad con la descripción de un proceso adaptativo y ágil aplicable a proyectos de software de tamaño moderado y grande.

WebRef

En la dirección **www.featuredrivendevelopment.com/** se encuentra una amplia variedad de artículos y presentaciones sobre el DIC.

Igual que otros proyectos ágiles, DIC adopta una filosofía que: 1) pone el énfasis en la colaboración entre los integrantes de un equipo DIC; 2) administra la complejidad de los problemas y del proyecto con el uso de la descomposición basada en las características, seguida de la integración de incrementos de software, y 3) comunica los detalles técnicos en forma verbal, gráfica y con medios basados en texto. El DIC pone el énfasis en las actividades de aseguramiento de la calidad del software mediante el estímulo de la estrategia de desarrollo incremental, el uso de inspecciones del diseño y del código, la aplicación de auditorías de aseguramiento de la calidad del software (véase el capítulo 16), el conjunto de mediciones y el uso de patrones (para el análisis, diseño y construcción).

En el contexto del DIC, una *característica* "es una función valiosa para el cliente que puede implementarse en dos semanas o menos" [Coa99]. El énfasis en la definición de características proporciona los beneficios siguientes:

- Debido a que las características son bloques pequeños de funcionalidad que se entrega, los usuarios las describen con más facilidad, entienden cómo se relacionan entre sí y las revisan mejor en busca de ambigüedades, errores u omisiones.
- Las características se organizan por jerarquía de grupos relacionados con el negocio.
- Como una característica es el incremento de software DIC que se entrega, el equipo desarrolla características operativas cada dos semanas.
- El diseño y representación en código de las características son más fáciles de inspeccionar con eficacia porque éstas son pequeñas.
- La planeación, programación de actividades y seguimiento son determinadas por la jerarquía de características, y no por un conjunto de tareas de ingeniería de software adoptadas en forma arbitraria.

Coad y sus colegas [Coa99] sugieren el esquema siguiente para definir una característica:

<acción> el **<resultado>** **<a|por|de|para>** un **<objeto>**

15 El nombre "cristal" se deriva de los cristales de minerales, cada uno de los cuales tiene propiedades específicas de color, forma y dureza.

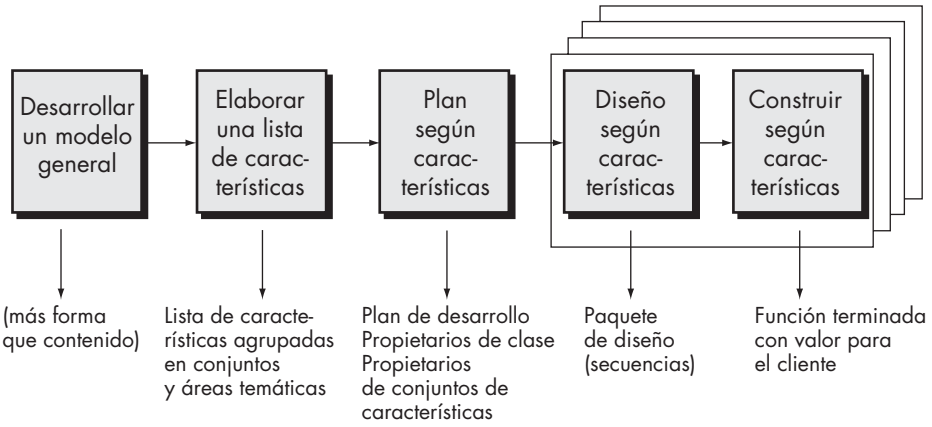


FIGURA 3.5
Desarrollo impulsado por las características [Coa99] (con permiso)

donde **<objeto>** es "una persona, lugar o cosa (incluso roles, momentos del tiempo o intervalos temporales, o descripciones parecidas a las entradas de un catálogo)". Algunos ejemplos de características para una aplicación de comercio electrónico son los siguientes:

Agregar el producto al carrito de compras

Mostrar las especificaciones técnicas del producto

Guardar la información de envío para el cliente

Un conjunto de características agrupa las que son similares en categorías relacionadas con el negocio y se define así:

<acción><ndo> un **<objeto>**

Por ejemplo: *Haciendo una venta del producto* es un conjunto de características que agruparía las que ya se mencionaron y otras más.

El enfoque DIC define cinco actividades estructurales "colaborativas" [Coa99] (en el enfoque DIC se llaman "procesos"), como se muestra en la figura 3.5.

El DIC pone más énfasis que otros métodos ágiles en los lineamientos y técnicas para la administración de proyectos. A medida que éstos aumentan su tamaño y complejidad, no es raro que la administración de proyectos *ad hoc* sea inadecuada. Para los desarrolladores, sus gerentes y otros participantes, es esencial entender el estado del proyecto, es decir, los avances realizados y los problemas que han surgido. Si la presión por cumplir el plazo de entrega es mucha, tiene importancia crítica determinar si la entrega de los incrementos del software está programada en forma adecuada. Para lograr esto, el DIC define seis puntos de referencia durante el diseño e implementación de una característica: "recorrido por el diseño, diseño, inspección del diseño, código, inspección del código, decisión de construir" [Coa99].

3.5.6 Desarrollo esbelto de software (DES)

El *desarrollo esbelto de software* (DES) adapta los principios de la manufactura esbelta al mundo de la ingeniería de software. Los principios de esbeltez que inspiran al proceso DES se resumen como sigue ([Pop03], [Pop06a]): *eliminar el desperdicio, generar calidad, crear conocimiento, aplazar el compromiso, entregar rápido, respetar a las personas* y *optimizar al todo*.

Es posible adaptar cada uno de estos principios al proceso del software. Por ejemplo, *eliminar el desperdicio* en el contexto de un proyecto de software ágil significa [Das05]: 1) no agregar características o funciones extrañas, 2) evaluar el costo y el efecto que tendrá en la programación de actividades cualquier nuevo requerimiento solicitado, 3) eliminar cualesquiera etapas superfluas del proceso, 4) establecer mecanismos para mejorar la forma en la que los miembros

del equipo obtienen información, 5) asegurar que las pruebas detecten tantos errores como sea posible, 6) reducir el tiempo requerido para pedir y obtener una decisión que afecta al software o al proceso que se aplica para crearlo, y 7) simplificar la manera en la que se transmite la información a todos los participantes involucrados en el proceso.

Para un análisis detallado del DES y para conocer lineamientos prácticos a fin de implementar el proceso, debe consultarse [Pop06a] y [Pop06b].

3.5.7 Modelado ágil (MA)

WebRef

En la dirección **www.agilemodeling.com** hay información amplia sobre el modelado ágil.

Hay muchas situaciones en las que los ingenieros de software deben construir sistemas grandes de importancia crítica para el negocio. El alcance y complejidad de tales sistemas debe modelarse de modo que: 1) todos los actores entiendan mejor cuáles son las necesidades que deben satisfacerse, 2) el problema pueda dividirse con eficacia entre las personas que deben resolverlo, y 3) se asegure la calidad a medida que se hace la ingeniería y se construye el sistema.

En los últimos 30 años se ha propuesto una gran variedad de métodos de modelado y notación para la ingeniería de software con objeto de hacer el análisis y el diseño (tanto en la arquitectura como en los componentes). Estos métodos tienen su mérito, pero se ha demostrado que son difíciles de aplicar y sostener (en muchos proyectos). Parte del problema es el "peso" de dichos métodos de modelación. Con esto se hace referencia al volumen de la notación que se requiere, al grado de formalismo sugerido, al tamaño absoluto de los modelos para proyectos grandes y a la dificultad de mantener el(los) modelo(s) conforme suceden los cambios. Sin embargo, el análisis y el modelado del diseño tienen muchos beneficios para los proyectos grandes, aunque no fuera más que porque hacen a esos proyectos intelectualmente más manejables. ¿Hay algún enfoque ágil para el modelado de la ingeniería de software que brinde una alternativa?

Cita:

"El otro día fui a la farmacia por una medicina para el resfriado… no fue fácil. Había toda una pared cubierta de productos. Al recorrerla vi uno que era de acción rápida, pero otro que era de larga duración… ¿Qué es más importante, el presente o el futuro?"

Jerry Seinfeld

En el "sitio oficial de modelado ágil", Scott Ambler [Amb02a] describe el *modelado ágil* (MA) del modo siguiente:

> El modelado ágil (MA) es una metodología basada en la práctica para modelar y documentar con eficacia los sistemas basados en software. En pocas palabras, es un conjunto de valores, principios y prácticas para hacer modelos de software aplicables de manera eficaz y ligera a un proyecto de desarrollo de software. Los modelos ágiles son más eficaces que los tradicionales porque son sólo buenos, sin pretender ser perfectos.

El modelado ágil adopta todos los valores del manifiesto ágil. La filosofía de modelado ágil afirma que un equipo ágil debe tener la valentía para tomar decisiones que impliquen rechazar un diseño y reconstruirlo. El equipo también debe tener la humildad de reconocer que los tecnólogos no tienen todas las respuestas y que los expertos en el negocio y otros participantes deben ser respetados e incluidos.

Aunque el MA sugiere una amplia variedad de principios de modelado "fundamentales" y "suplementarios", aquellos que son exclusivos del MA son los siguientes [Amb02a]:

Modelo con un propósito. Un desarrollador que use el MA debe tener en mente una meta específica (por ejemplo, comunicar información al cliente o ayudarlo a entender mejor algún aspecto del software) antes de crear el modelo. Una vez identificada la meta para el modelo, el tipo y nivel de detalle de la notación por usar serán más obvios.

Uso de modelos múltiples. Hay muchos modelos y notaciones diferentes que pueden usarse para describir el software. Para la mayoría de proyectos sólo es esencial un pequeño subconjunto. El MA sugiere que para dar la perspectiva necesaria, cada modelo debe presentar un diferente aspecto del sistema y que sólo deben utilizarse aquellos modelos que den valor al público al que se dirigen.

"Viajar ligero" es una filosofía apropiada para todo el trabajo de ingeniería de software. Construir sólo aquellos modelos que agreguen valor… ni más ni menos.

Viajar ligero. Conforme avanza el trabajo de ingeniería de software, conserve sólo aquellos modelos que agreguen valor a largo plazo y elimine los demás. Todo producto del trabajo que se conserve debe recibir mantenimiento cuando haya cambios. Esto representa una labor que hace lento al equipo. Ambler [Amb02a] afirma que "cada vez que se decide conservar un modelo, se pierde agilidad en nombre de la conveniencia de tener disponible esa información en forma abstracta para el equipo (y de ese modo mejorar potencialmente la comunicación dentro del equipo, así como con los participantes)".

El contenido es más importante que la representación. El modelado debe transmitir información al público al que se dirige. Un modelo con sintaxis perfecta que transmita poco contenido útil no es tan valioso como otro que tenga notación defectuosa, pero que, no obstante, provea contenido de valor para los usuarios.

Conocer los modelos y herramientas que se utilizan en su creación. Entender las fortalezas y debilidades de cada modelo y las herramientas que se emplean para crearlos.

Adaptación local. El enfoque de modelado debe adaptarse a las necesidades del equipo ágil.

Un segmento importante de la comunidad de ingeniería de software ha adoptado el lenguaje de unificado de modelado (UML, por sus siglas en inglés)[16] como el método preferido para representar modelos del análisis y del diseño. El proceso unificado (véase el capítulo 2) fue desarrollado para proveer una estructura para la aplicación del UML. Scott Ambler [Amb06] desarrolló una versión simplificada del PU que integra su filosofía de modelado ágil.

3.5.8 El proceso unificado ágil (PUA)

El *proceso unificado ágil* (PUA) adopta una filosofía "en serie para lo grande" e "iterativa para lo pequeño" [Amb06] a fin de construir sistemas basados en computadora. Al adoptar las actividades en fase clásicas del PU —*concepción, elaboración, construcción* y *transición*—, el PUA brinda un revestimiento en serie (por ejemplo, una secuencia lineal de actividades de ingeniería de software) que permite que el equipo visualice el flujo general del proceso de un proyecto de software. Sin embargo, dentro de cada actividad, el equipo repite con objeto de alcanzar la agilidad y entregar tan rápido como sea posible incrementos de software significativos a los usuarios finales. Cada iteración del PUA aborda las actividades siguientes [Amb06]:

- *Modelado.* Se crean representaciones de UML de los dominios del negocio y el problema. No obstante, para conservar la agilidad, estos modelos deben ser "sólo suficientemente buenos" [Amb06] para permitir que el equipo avance.
- *Implementación.* Los modelos se traducen a código fuente.
- *Pruebas*. Igual que con la XP, el equipo diseña y ejecuta una serie de pruebas para detectar errores y garantizar que el código fuente cumple sus requerimientos.
- *Despliegue.* Como en la actividad general del proceso que se estudió en los capítulos 1 y 2, el despliegue en este contexto se centra en la entrega de un incremento de software y en la obtención de retroalimentación de los usuarios finales.
- *Configuración y administración del proyecto*. En el contexto del PUA, la administración de la configuración (véase el capítulo 22) incluye la administración del cambio y el riesgo, y el control de cualesquiera productos del trabajo persistentes[17] que produzca el equipo.

16 En el apéndice 1 se presenta un método breve para aprender UML.

17 Un *producto del trabajo persistente* es un modelo o documento o caso de prueba producido por el equipo y que se conservará durante un periodo indeterminado. *No* se eliminará una vez entregado el incremento de software.

La administración del proyecto da seguimiento y controla el avance del equipo y coordina sus actividades.

- *Administración del ambiente.* La administración del ambiente coordina una infraestructura del proceso que incluye estándares, herramientas y otra tecnología de apoyo de la que dispone el equipo.

Aunque el PUA tiene conexiones históricas y técnicas con el lenguaje unificado de modelado, es importante observar que el modelado UML puede usarse junto con cualesquiera de los modelos de proceso ágil descritos en la sección 3.5.

HERRAMIENTAS DE SOFTWARE

Desarrollo ágil

Objetivo: El objetivo de las herramientas de desarrollo ágil es ayudar en uno o más aspectos de éste, con énfasis en facilitar la elaboración rápida de software funcional. Estas herramientas también pueden emplearse cuando se aplican modelos de proceso prescriptivo (véase el capítulo 2).

Mecánica: Las herramientas de mecánica varían. En general, las herramientas ágiles incluyen el apoyo automatizado para la planeación del proyecto, el desarrollo de casos y la obtención de requerimientos, el diseño rápido, la generación de código y la realización de pruebas.

Herramientas representativas:[18]

Nota: Debido a que el desarrollo ágil es un tema de moda, la mayoría de los vendedores de herramientas de software tratan de colocar herramientas que lo apoyan. Las que se mencionan a continuación tienen características que las hacen particularmente útiles para los proyectos ágiles.

OnTime, desarrollada por Axosoft (**www.axosoft.com**), presta apoyo a la administración de un proceso ágil para distintas actividades técnicas dentro del proceso.

Ideogramic UML, desarrollada por Ideogramic (**www.ideogramic.com**), es un conjunto de herramientas UML desarrolladas específicamente para usarlas dentro de un proceso ágil.

Together Tool Set, distribuido por Borland (**www.borland.com**), proporciona un grupo de herramientas para apoyar muchas actividades técnicas dentro de XP y otros procesos ágiles.

3.6 CONJUNTO DE HERRAMIENTAS PARA EL PROCESO ÁGIL

El "conjunto de herramientas" que da apoyo a los procesos ágiles se centra más en aspectos de la persona que en los de la tecnología.

Algunos defensores de la filosofía ágil afirman que las herramientas automatizadas de software (por ejemplo, las de diseño) deben verse como un complemento menor de las actividades del equipo, y no como algo fundamental para el éxito. Sin embargo, Alistair Cockburn [Coc04] sugiere que las herramientas tienen un beneficio y que "los equipos ágiles favorecen el uso de herramientas que permiten el flujo rápido de entendimiento. Algunas de estas herramientas son sociales y comienzan incluso en la etapa de reclutamiento. Otras son tecnológicas y ayudan a que los equipos distribuidos simulen su presencia física. Muchas herramientas son físicas y permiten que las personas las manipulen en talleres".

Prácticamente todos los modelos de proceso ágil son elementos clave en la contratación del personal adecuado (reclutamiento), la colaboración en equipo, la comunicación con los participantes y la administración indirecta; por eso, Cockburn afirma que las "herramientas" que se abocan a dichos aspectos son factores críticos para el éxito de la agilidad. Por ejemplo, una "herramienta" de reclutamiento tal vez sea el requerimiento de que un prospecto a miembro del equipo pase algunas horas programando en pareja con alguien que ya es integrante del equipo. El "ajuste" se evalúa de inmediato.

Las "herramientas" de colaboración y comunicación por lo general son de baja tecnología e incorporan cualquier mecanismo ("proximidad física, pizarrones, tableros, tarjetas y notas ad-

18 Las herramientas mencionadas aquí no son obligatorias, sólo son una muestra en esta categoría. En la mayoría de casos, sus nombres son marcas registradas por sus respectivos desarrolladores.

heribles" [Coc04] que provea información y coordinación entre los desarrolladores ágiles. La comunicación activa se logra por medio de la dinámica del equipo (por ejemplo, la programación en parejas), mientras que la comunicación pasiva se consigue con "radiadores de información" (un tablero que muestre el estado general de de los distintos componentes de un incremento). Las herramientas de administración de proyectos no ponen el énfasis en la gráfica de Gantt y la sustituyen con otras de valor agregado o "gráficas de pruebas creadas *versus* pasadas; otras herramientas ágiles se utilizan para optimizar el ambiente en el que trabaja el equipo ágil (por ejemplo, áreas más eficientes para reunirse), mejoran la cultura del equipo por medio de cultivar las interacciones sociales (equipos con algo en común), dispositivos físicos (pizarrones electrónicos) y el mejoramiento del proceso (por ejemplo, la programación por parejas o la caja de tiempo)" [Coc04].

¿Algunas de las mencionadas son en verdad herramientas? Sí, lo son, si facilitan el trabajo efectuado por un miembro del equipo ágil y mejoran la calidad del producto final.

3.7 Resumen

En una economía moderna, las condiciones del mercado cambian con rapidez, los clientes y usuarios finales necesitan evolucionar y surgen nuevas amenazas competitivas sin aviso previo. Los profesionales deben enfocar la ingeniería de software en forma que les permita mantenerse ágiles para definir procesos maniobrables, adaptativos y esbeltos que satisfagan las necesidades de los negocios modernos.

Una filosofía ágil para la ingeniería de software pone el énfasis en cuatro aspectos clave: la importancia de los equipos con organización propia que tienen el control sobre el trabajo que realizan, la comunicación y colaboración entre los miembros del equipo y entre los profesionales y sus clientes, el reconocimiento de que el cambio representa una oportunidad y la insistencia en la entrega rápida de software que satisfaga al consumidor. Los modelos de proceso ágil han sido diseñados para abordar cada uno de estos aspectos.

La programación extrema (XP) es el proceso ágil de más uso. Organizada con cuatro actividades estructurales: planeación, diseño, codificación y pruebas, la XP sugiere cierto número de técnicas innovadoras y poderosas que permiten a un equipo ágil generar entregas frecuentes de software que posee características y funcionalidad que han sido descritas y clasificadas según su prioridad por los participantes.

Otros modelos de proceso ágil también insisten en la colaboración humana y en la organización propia del equipo, pero definen sus actividades estructurales y seleccionan diferentes puntos de importancia. Por ejemplo, el DAS utiliza un proceso iterativo que incluye un ciclo de planeación adaptativa, métodos relativamente rigurosos para recabar requerimientos, y un ciclo de desarrollo iterativo que incorpora grupos de consumidores y revisiones técnicas formales como mecanismos de retroalimentación en tiempo real. El Scrum pone el énfasis en el uso de un conjunto de patrones de software que han demostrado ser eficaces para proyectos que tienen plazos de entrega apretados, requerimientos cambiantes o que se emplean en negocios críticos. Cada patrón de proceso define un conjunto de tareas de desarrollo y permite al equipo Scrum construir un proceso que se adapte a las necesidades del proyecto. El método de desarrollo de sistemas dinámicos (MDSD) resalta el uso de la programación con caja de tiempo y sugiere que en cada incremento de software sólo se requiere el trabajo suficiente que facilite el paso al incremento que sigue. Cristal es una familia de modelos de proceso ágil que se adaptan a las características específicas del proyecto.

El desarrollo impulsado por las características (DIC) es algo más "formal" que otros métodos ágiles, pero conserva su agilidad al centrar al equipo del proyecto en el desarrollo de características, funciones valiosas para el cliente que pueden implementarse en dos semanas o menos. El

desarrollo esbelto de software (DES) ha adaptado los principios de la manufactura esbelta al mundo de la ingeniería de software. El modelado ágil (MA) sugiere que el modelado es esencial para todos los sistemas, pero que la complejidad, tipo y tamaño del modelo deben ajustarse al software que se va a elaborar. El proceso unificado ágil (PUA) adopta una filosofía "serial en lo grande" e "iterativo en lo pequeño" para la elaboración de software.

Problemas y puntos por evaluar

3.1. Vuelva a leer el "Manifiesto para el desarrollo ágil de software" al principio de este capítulo. ¿Puede pensar en una situación en la que uno o más de los cuatro "valores" pudieran causar problemas al equipo de software?

3.2. Describa con sus propias palabras la *agilidad* (para proyectos de software).

3.3. ¿Por qué un proceso iterativo hace más fácil administrar el cambio? ¿Es iterativo todo proceso ágil analizado en este capítulo? ¿Es posible terminar un proyecto en sólo una iteración y aún así conseguir que sea ágil? Explique sus respuestas.

3.4. ¿Podría describirse cada uno de los procesos ágiles con el uso de las actividades estructurales generales mencionadas en el capítulo 2? Construya una tabla que mapee las actividades generales en las actividades definidas para cada proceso ágil.

3.5. Proponga un "principio de agilidad" más que ayudaría al equipo de ingeniería de software a ser aún más maniobrable.

3.6. Seleccione un principio de agilidad mencionado en la sección 3.3.1 y trate de determinar si está incluido en cada uno de los modelos de proceso presentados en este capítulo. [Nota: sólo se presentó el panorama general de estos modelos de proceso, por lo que tal vez no fuera posible determinar si un principio está incluido en uno o más de ellos, a menos que el lector hiciera una investigación (lo que no se requiere para este problema)].

3.7. ¿Por qué cambian tanto los requerimientos? Después de todo, ¿la gente no sabe lo que quiere?

3.8. La mayoría de modelos de proceso ágil recomiendan la comunicación cara a cara. No obstante, los miembros del equipo de software y sus clientes tal vez estén alejados geográficamente. ¿Piensa usted que esto implica que debe evitarse la separación geográfica? ¿Se le ocurren formas de resolver este problema?

3.9. Escriba una historia de usuario XP que describa la característica de "lugares favoritos" o "marcadores" disponible en la mayoría de navegadores web.

3.10. ¿Qué es una solución en punta en XP?

3.11. Describa con sus propias palabras los conceptos de rediseño y programación en parejas de XP.

3.12. Haga otras lecturas y describa lo que es una caja de tiempo. ¿Cómo ayuda a un equipo DAS para que entregue incrementos de software en un corto periodo?

3.13. ¿Se logra el mismo resultado con la regla de 80% del MDSD y con el enfoque de la caja de tiempo del DAS?

3.14. Con el formato de patrón de proceso presentado en el capítulo 2, desarrolle uno para cualquiera de los patrones Scrum presentados en la sección 3.5.2.

3.15. ¿Por qué se le llama a Cristal familia de métodos ágiles?

3.16. Con el formato de característica DIC descrito en la sección 3.5.5, defina un conjunto de características para un navegador web. Luego desarrolle un conjunto de características para el primer conjunto.

3.17. Visite el sitio oficial de modelación ágil y elabore la lista completa de todos los principios fundamentales y secundarios del MA.

3.18. El conjunto de herramientas propuestas en la sección 3.6 da apoyo a muchos de los aspectos "suaves" de los métodos ágiles. Debido a que la comunicación es tan importante, recomiende un conjunto de herramientas reales que podría utilizarse para que los participantes de un equipo ágil se comuniquen mejor.

Lecturas adicionales y fuentes de información

La filosofía general y principios que subyacen al desarrollo de software ágil se estudian a profundidad en muchos de los libros mencionados a lo largo de este capítulo. Además, los textos de Shaw y Warden (*The Art of Agile Development*, O'Reilly Media, Inc., 2008), Hunt (*Agile Software Construction*, Springer, 2005) y Carmichael y Haywood (*Better Software Faster*, Prentice-Hall, 2002) presentan análisis útiles del tema. Aguanno (*Managing Agile Projects*, Multi-Media Publications, 2005), Highsmith (*Agile Project Management: Creating Innovative Products*, Addison-Wesley, 2004) y Larman (*Agile and Iterative Development: A Manager's Guide*, Addison-Wesley, 2003) presentan el punto de vista de la administración y consideran ciertos aspectos de la administración de proyectos. Highsmith (*Agile Software Development Ecosystems*, Addison-Wesley, 2002) expone una encuesta acerca de los principios, procesos y prácticas ágiles. Booch y sus colegas (*Balancing Agility and Discipline*, Addison-Wesley, 2004) hacen un análisis fructífero del delicado equilibrio entre agilidad y disciplina.

Martin (*Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice-Hall, 2009) explica los principios, patrones y prácticas que se requieren para desarrollar "código limpio" en un ambiente de ingeniería de software ágil. Leffingwell (*Scaling Software Agility: Best Practices for Large Enterprises*, Addison-Wesley, 2007) estudia estrategias para ampliar las prácticas ágiles en proyectos grandes. Lippert y Rook (*Refactoring in Large Software Projects: Performing Complex Restructurings Succesfully*, Wiley, 2006) analizan el uso del rediseño cuando se aplica a sistemas grandes y complejos. Stamelos y Sfetsos (*Agile Software Development Quality Assurance*, IGI Global, 2007) analizan las técnicas SQA que forman la filosofía ágil.

En la última década se han escrito decenas de libros sobre programación extrema. Beck (*Extreme Programming Explained: Embrace Change*, 2a. ed., Addison-Wesley, 2004) sigue siendo la referencia definitiva al respecto. Además, Jeffries y sus colegas (*Extreme Programming Installed*, Addison-Wesley, 2000), Succi y Marchesi (*Extreme Programming Examined*, Addison-Wesley, 2001), Newkirk y Martin (*Extreme Programming in Practice*, Addison-Wesley, 2001) y Auer y sus colegas (*Extreme Programming Applied: Play to Win*, Addison-Wesley, 2001) hacen un análisis detallado de XP y dan una guía para aplicarla de la mejor forma. McBreen (*Questioning Extreme Programming*, Addison-Wesley, 2003) adopta un enfoque crítico sobre XP, y define cuándo y dónde es apropiada. Un estudio profundo de la programación por parejas se presenta en McBreen (*Pair Programming Illuminated*, Addison-Wesley, 2003).

Highsmith [Hig00] analiza con detalle el DAS. Schwaber (*The Enterprise and Scrum*, Microsoft Press, 2007) estudia el empleo de Scrum para proyectos que tienen un efecto grande en los negocios. Los detalles de Scrum los estudian Schwaber y Beedle (*Agile Software Development with SCRUM*, Prentice-Hall, 2001). Algunos tratamientos útiles del MDSD han sido escritos por DSDM Consortium (*DSDM: Business Focused Development*, 2a. ed., Pearson Education, 2003) y Stapleton (*DSDM: The Method in Practice*, Addison-Wesley, 1997). Cockburn (*Crystal Clear*, Addison-Wesley, 2005) presenta un panorama excelente de la familia de procesos Cristal. Palmer y Felsing [Pal02] dan un tratamiento detallado del DIC. Carmichael y Haywood (*Better Software Faster*, Prentice-Hall, 2002) proporcionan otro análisis útil del DIC, que incluye un recorrido, paso a paso, por la mecánica del proceso. Poppendieck y Poppendieck (*Lean Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003) dan lineamientos para la administración y el control de proyectos ágiles. Ambler y Jeffries (*Agile Modeling*, Wiley, 2002) estudian el MA con cierta profundidad.

En internet existe una amplia variedad de fuentes de información sobre el desarrollo de software ágil. En el sitio web del libro hay una lista actualizada de referencias en la Red Mundial que son relevantes para el proceso ágil, en la dirección: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

PARTE

Dos

Modelado

En esta parte de la obra, aprenderá sobre los principios, conceptos y métodos que se usan para crear requerimientos de alta calidad y para diseñar modelos. En los capítulos que siguen se abordan preguntas como las siguientes:

- ¿Qué conceptos y principios guían la práctica de la ingeniería de software?
- ¿Qué son los requerimientos de ingeniería y cuáles son los conceptos subyacentes que llevan a un buen análisis de requerimientos?
- ¿Cómo se crean los requerimientos del modelo y cuáles son sus elementos?
- ¿Cuáles son los elementos de un buen diseño?
- ¿Cómo establece el diseño de la arquitectura una estructura para todas las demás acciones de diseño y qué modelos se utilizan?
- ¿Cómo se diseñan componentes de software de alta calidad?
- ¿Qué conceptos, modelos y métodos se aplican al diseñar una interfaz de usuario?
- ¿Qué es el diseño basado en patrones?
- ¿Qué estrategias y métodos especializados se emplean para diseñar *webapps*?

Una vez que se respondan estas preguntas, el lector estará mejor preparado para aplicar en la práctica la ingeniería de software.

CAPÍTULO

4 Principios que guían la práctica

Conceptos clave

Principios fundamentales 83
Principios que gobiernan lo siguiente:
codificación 94
comunicación 86
despliegue 96
diseño 92
modelado 90
planeación 88
pruebas 95
requerimientos 91

En un libro que explora las vidas y pensamientos de los ingenieros de software, Ellen Ullman [Ull97] ilustra una parte de su vida con el relato de lo que piensa un profesional del software cuando está bajo presión:

> No tengo idea de la hora que es. En esta oficina no hay ventanas ni reloj, sólo la pantalla de un horno de microondas que parpadea su LED de color rojo: 12:00, 12:00, 12:00. Joel y yo hemos estado programando durante varios días. Tenemos una falla, endemoniada y testaruda. Así que nos sentimos bien con el pulso rojo sin tiempo, como si fuera un pasmo de nuestros cerebros, de algún modo sincronizados al mismo ritmo del parpadeo...
>
> ¿En qué estamos trabajando? Los detalles se me escapan. Tal vez ayudamos a personas pobres y enfermas o mejoramos un conjunto de rutinas de bajo nivel de un protocolo de base de datos distribuida, no me importa. Debería importarme; en otra parte de mi ser —más tarde, quizá cuando salga de este cuarto lleno de computadoras— me preocuparé mucho de por qué y para quién y con qué propósito estoy escribiendo software. Pero ahora, no. He cruzado una membrana tras la que el mundo real y sus asuntos ya no importan. Soy ingeniera de software.

La anterior es una imagen tenebrosa de la práctica de la ingeniería de software, pero si se detienen un poco a pensarlo, muchos de los lectores de este libro se verán reflejados en ella.

Las personas que elaboran software de cómputo practican el arte, artesanía o disciplina[1] conocida como ingeniería de software. Pero, ¿qué es la "práctica" de la ingeniería de software? En un sentido general, es un conjunto de conceptos, principios, métodos y herramientas a los que un ingeniero de software recurre en forma cotidiana. La práctica permite que los gerentes

Una mirada rápida

¿Qué es? La práctica de la ingeniería de software es un conjunto amplio de principios, conceptos, métodos y herramientas que deben considerarse al planear y desarrollar software.

¿Quién lo hace? Los profesionales (ingenieros de software) y sus gerentes realizan varias tareas de ingeniería de software.

¿Por qué es importante? El proceso de software proporciona a todos los involucrados en la creación de un sistema o producto basado en computadora un mapa para llegar con éxito al destino. La práctica proporciona los detalles que se necesitarán para circular por la carretera. Indica dónde se localizan los puentes, los caminos cerrados y las bifurcaciones. Ayuda a entender los conceptos y principios que deben entenderse y seguirse a fin de llegar con seguridad y rapidez. Enseña a manejar, dónde disminuir la velocidad y en qué lugares acelerar. En el contexto de la ingeniería de software, la práctica es lo que se hace día tras día conforme el software evoluciona de idea a realidad.

¿Cuáles son los pasos? Son tres los elementos de la práctica que se aplican sin importar el modelo de proceso que se elija. Se trata de: principios, conceptos y métodos. Un cuarto elemento de la práctica —las herramientas— da apoyo a la aplicación de los métodos.

¿Cuál es el producto final? La práctica incluye las actividades técnicas que generan todos los productos del trabajo definidos por el modelo del proceso de software que se haya escogido.

¿Cómo me aseguro de que lo hice bien? En primer lugar, hay que tener una comprensión sólida de los principios que se aplican al trabajo (por ejemplo, el diseño) en cuestión. Después, asegúrese de que se escogió el método apropiado para el trabajo, use herramientas automatizadas cuando sean adecuadas para la tarea y sea firme respecto de la necesidad de técnicas de aseguramiento de la calidad de los productos finales que se generen.

1 Algunos escritores afirman que cualquiera de estos términos excluye a los otros. En realidad, la ingeniería de software es las tres cosas.

administren proyectos de software y que los ingenieros de software elaboren programas de cómputo. La práctica da al modelo del proceso de software el saber técnico y administrativo para realizar el trabajo. La práctica transforma un enfoque caprichoso y disperso en algo más organizado, más eficaz y con mayor probabilidad de alcanzar el éxito.

A lo largo de lo que resta del libro se estudiarán distintos aspectos de la práctica de la ingeniería de software. En este capítulo, la atención se pone en los principios y conceptos que la guían en lo general.

4.1 Conocimiento de la ingeniería de software

En un editorial publicado hace diez años en *IEEE Software*, Steve McConnell [McC99] hizo el siguiente comentario:

> Muchos trabajadores del software piensan que el conocimiento de la ingeniería de software casi exclusivamente consiste en tecnologías específicas: Java, Perl, html, C++, Linux, Windows NT, etc. Para programar computadoras es necesario conocer los detalles tecnológicos específicos. Si alguien pide al lector que escriba un programa en C++, tiene que saber algo sobre este lenguaje a fin de que el programa funcione.
>
> Es frecuente escuchar que el conocimiento del desarrollo de software tiene una vida media de tres años, lo que significa que la mitad de lo que es necesario saber hoy será obsoleto dentro de tres años. En el dominio del conocimiento relacionado con la tecnología es probable que eso se cumpla. Pero hay otra clase de conocimiento de desarrollo de software —algo que el autor considera como los "principios de la ingeniería de software"— que no tiene una vida media de tres años. Es factible que dichos principios sirvan al programador profesional durante toda su carrera.

McConnell continúa y plantea que el cuerpo de conocimientos de la ingeniería de software (alrededor del año 2000) ha evolucionado para convertirse en un "núcleo estable" que representa cerca de "75% del conocimiento necesario para desarrollar un sistema complejo". Pero, ¿qué es lo que hay dentro de ese núcleo estable?

Como dice McConnell, los principios fundamentales —ideas elementales que guían a los ingenieros de software en el trabajo que realizan— dan ahora un fundamento a partir del cual pueden aplicarse y evaluarse los modelos, métodos y herramientas de ingeniería.

4.2 Principios fundamentales

Cita:

"En teoría no hay diferencia entre la teoría y la práctica. Pero en la práctica sí la hay."

Jan van de Snepscheut

La práctica de la ingeniería de software está guiada por un conjunto de principios fundamentales que ayudan en la aplicación del proceso de software significativo y en la ejecución de métodos eficaces de ingeniería de software. En el nivel del proceso, los principios fundamentales establecen un fundamento filosófico que guía al equipo de software cuando realiza actividades estructurales y actividades sombrilla, cuando navega por el flujo del proceso y elabora un conjunto de productos del trabajo de la ingeniería de software. En el nivel de la práctica, los principios fundamentales definen un conjunto de valores y reglas que sirven como guía cuando se analiza un problema, se diseña una solución, se implementa y prueba ésta y cuando, al final, se entrega el software a la comunidad de usuarios.

En el capítulo 1 se identificó un conjunto de principios generales que amplían el proceso y práctica de la ingeniería de software: 1) agregar valor para los usuarios finales, 2) mantenerlo sencillo, 3) fijar la visión (del producto y el proyecto), 4) reconocer que otros consumen (y deben entender) lo que usted produce, 5) abrirse al futuro, 6) planear la reutilización y 7) ¡pensar! Aunque estos principios generales son importantes, se caracterizan en un nivel tan alto de abstracción que en ocasiones son difíciles de traducir en la práctica cotidiana de la ingeniería de

software. En las subsecciones que siguen se analizan con más detalle los principios fundamentales que guían el proceso y la práctica.

4.2.1 Principios que guían el proceso

En la parte 1 de este libro se estudia la importancia del proceso de software y se describen los abundantes modelos de proceso que se han propuesto para hacer el trabajo de ingeniería de software. Sin que importe que un modelo sea lineal o iterativo, prescriptivo o ágil, puede caracterizarse con el empleo de la estructura general del proceso aplicable a todos los modelos de proceso. Los siguientes principios fundamentales se aplican a la estructura y, por extensión, a todo proceso de software:

Todo proyecto y equipo son únicos. Esto significa que debe adaptar el proceso para que se ajuste mejor a sus necesidades.

Principio 1. ***Ser ágil.*** Ya sea que el modelo de proceso que se elija sea prescriptivo o ágil, son los principios básicos del desarrollo ágil los que deben gobernar el enfoque. Todo aspecto del trabajo que se haga debe poner el énfasis en la economía de acción: en mantener el enfoque técnico tan sencillo como sea posible, hacer los productos del trabajo que se generan tan concisos como se pueda y tomar las decisiones localmente, siempre que sea posible.

Principio 2. ***En cada etapa, centrarse en la calidad.*** La condición de salida para toda actividad, acción y tarea del proceso debe centrarse en la calidad del producto del trabajo que se ha generado.

Principio 3. ***Estar listo para adaptar.*** El proceso no es una experiencia religiosa, en él no hay lugar para el dogma. Cuando sea necesario, adapte su enfoque a las restricciones impuestas por el problema, la gente y el proyecto en sí.

Principio 4. ***Formar un equipo eficaz.*** El proceso y práctica de la ingeniería de software son importantes, pero el objetivo son las personas. Forme un equipo con organización propia en el que haya confianza y respeto mutuos.

Principio 5. ***Establecer mecanismos para la comunicación y coordinación.*** Los proyectos fallan porque la información importante cae en las grietas o porque los participantes no coordinan sus esfuerzos para crear un producto final exitoso. Éstos son aspectos de la administración que deben enfrentarse.

Cita:

"La verdad es que siempre se sabe lo que es correcto hacer. La parte difícil es hacerlo."

General H. Norman Schwarzkopf

Principio 6. ***Administrar el cambio.*** El enfoque puede ser formal o informal, pero deben establecerse mecanismos para administrar la forma en la que los cambios se solicitan, evalúan, aprueban e implementan.

Principio 7. ***Evaluar el riesgo.*** Son muchas las cosas que pueden salir mal cuando se desarrolla software. Es esencial establecer planes de contingencia.

Principio 8. ***Crear productos del trabajo que agreguen valor para otros.*** Sólo genere aquellos productos del trabajo que agreguen valor para otras actividades, acciones o tareas del proceso. Todo producto del trabajo que se genere como parte de la práctica de ingeniería de software pasará a alguien más. La lista de las funciones y características requeridas se dará a la persona (o personas) que desarrollará(n) un diseño, el diseño pasará a quienes generan código y así sucesivamente. Asegúrese de que el producto del trabajo imparte la información necesaria sin ambigüedades u omisiones.

La parte 4 de este libro se centra en aspectos de la administración del proyecto y del proceso, y analiza en detalle varios aspectos de cada uno de dichos principios.

4.2.2 Principios que guían la práctica

La práctica de la ingeniería de software tiene un solo objetivo general: entregar a tiempo software operativo de alta calidad que contenga funciones y características que satisfagan las ne-

cesidades de todos los participantes. Para lograrlo, debe adoptarse un conjunto de principios fundamentales que guíen el trabajo técnico. Estos principios tienen mérito sin que importen los métodos de análisis y diseño que se apliquen, ni las técnicas de construcción (por ejemplo, el lenguaje de programación o las herramientas automatizadas) que se usen o el enfoque de verificación y validación que se elija. Los siguientes principios fundamentales son vitales para la práctica de la ingeniería de software:

Los problemas son más fáciles de resolver cuando se subdividen en entidades separadas, distintas entre sí, solucionables individualmente y verificables.

Principio 1. ***Divide y vencerás.*** Dicho en forma más técnica, el análisis y el diseño siempre deben enfatizar la *separación de entidades* (SdE). Un problema grande es más fácil de resolver si se divide en un conjunto de elementos (o *entidades*). Lo ideal es que cada entidad entregue funcionalidad distinta que pueda desarrollarse, y en ciertos casos validarse, independientemente de otras entidades.

Principio 2. ***Entender el uso de la abstracción.*** En su parte medular, una abstracción es una simplificación de algún elemento complejo de un sistema usado para comunicar significado en una sola frase. Cuando se usa la abstracción *hoja de cálculo*, se supone que se comprende lo que es una hoja de cálculo, la estructura general de contenido que presenta y las funciones comunes que se aplican a ella. En la práctica de la ingeniería de software, se usan muchos niveles diferentes de abstracción, cada uno de los cuales imparte o implica significado que debe comunicarse. En el trabajo de análisis y diseño, un equipo de software normalmente comienza con modelos que representan niveles elevados de abstracción (por ejemplo, una hoja de cálculo) y poco a poco los refina en niveles más bajos de abstracción (como una *columna* o la función *SUM*).

Joel Spolsky [Spo02] sugiere que "todas las abstracciones no triviales hasta cierto punto son esquivas". El objetivo de una abstracción es eliminar la necesidad de comunicar detalles. Pero, en ocasiones, los efectos problemáticos precipitados por estos detalles se "filtran" por todas partes. Sin la comprensión de los detalles, no puede diagnosticarse con facilidad la causa de un problema.

Principio 3. ***Buscar la coherencia.*** Ya sea que se esté creando un modelo de los requerimientos, se desarrolle un diseño de software, se genere código fuente o se elaboren casos de prueba, el principio de coherencia sugiere que un contexto familiar hace que el software sea más fácil de usar. Como ejemplo, considere el diseño de una interfaz de usuario para una *webapp*. La colocación consistente de opciones de menú, el uso de un esquema coherencia de color y el uso coherencia de íconos reconocibles ayudan a hacer que la interfaz sea muy buena en el aspecto ergonómico.

Principio 4. ***Centrarse en la transferencia de información.*** El software tiene que ver con la transferencia de información: de una base de datos a un usuario final, de un sistema heredado a una *webapp*, de un usuario final a una interfaz gráfica de usuario (GUI, por sus siglas en inglés), de un sistema operativo a una aplicación, de un componente de software a otro... la lista es casi interminable. En todos los casos, la información fluye a través de una interfaz, y como consecuencia hay posibilidades de cometer errores, omisiones o ambigüedades. Este principio implica que debe ponerse atención especial al análisis, diseño, construcción y prueba de las interfaces.

Principio 5. ***Construir software que tenga modularidad eficaz.*** La separación de entidades (principio 1) establece una filosofía para el software. La *modularidad* proporciona un mecanismo para llevar a cabo dicha filosofía. Cualquier sistema complejo puede dividirse en módulos (componentes), pero la buena práctica de la ingeniería de software demanda más. La modularidad debe ser *eficaz*. Es decir, cada módulo debe centrarse exclusivamente en un aspecto bien delimitado del sistema: debe ser cohesivo en su función o restringido en el contenido que representa. Además, los módulos deben estar interconectados en forma

relativamente sencilla: cada módulo debe tener poco acoplamiento con otros módulos, fuentes de datos y otros aspectos ambientales.

Principio 6. *Buscar patrones.* Brad Appleton [App00] sugiere que:

> El objetivo de los patrones dentro de la comunidad de software es crear un cúmulo de bibliografía que ayude a los desarrolladores de software a resolver problemas recurrentes que surgen a lo largo del desarrollo. Los patrones ayudan a crear un lenguaje compartido para comunicar perspectiva y experiencia acerca de dichos patrones y sus soluciones. La codificación formal de estas soluciones y sus relaciones permite acumular con éxito el cuerpo de conocimientos que define nuestra comprensión de las buenas arquitecturas que satisfacen las necesidades de sus usuarios.

CONSEJO

Use patrones (véase el capítulo 12) a fin de acumular conocimiento y experiencia para las futuras generaciones de ingenieros de software.

Principio 7. *Cuando sea posible, representar el problema y su solución desde varias perspectivas diferentes.* Cuando un problema y su solución se estudian desde varias perspectivas distintas, es más probable que se tenga mayor visión y que se detecten los errores y omisiones. Por ejemplo, un modelo de requerimientos puede representarse con el empleo de un punto de vista orientado a los datos, a la función o al comportamiento (véanse los capítulos 6 y 7). Cada uno brinda un punto de vista diferente del problema y de sus requerimientos.

Principio 8. *Tener en mente que alguien dará mantenimiento al software.* El software será corregido en el largo plazo, cuando se descubran sus defectos, se adapte a los cambios de su ambiente y se mejore en el momento en el que los participantes pidan más capacidades. Estas actividades de mantenimiento resultan más fáciles si se aplica una práctica sólida de ingeniería de software a lo largo del proceso de software.

Estos principios no son todo lo que se necesita para elaborar software de alta calidad, pero establecen el fundamento para todos los métodos de ingeniería de software que se estudian en este libro.

4.3 Principios que guían toda actividad estructural

Cita:

"El ingeniero ideal es una mezcla… no es un científico, no es un matemático, no es un sociólogo ni un escritor; pero para resolver problemas de ingeniería utiliza conocimiento y técnicas de algunas o de todas esas disciplinas."

N. W. Dougherty

En las secciones que siguen se consideran los principios que tienen mucha relevancia para el éxito de cada actividad estructural genérica, definida como parte del proceso de software. En muchos casos, los principios que se estudian para cada una de las actividades estructurales son un refinamiento de los principios presentados en la sección 4.2. Tan sólo son principios fundamentales planteados en un nivel más bajo de abstracción.

4.3.1 Principios de comunicación

Antes de que los requerimientos del cliente se analicen, modelen o especifiquen, deben recabarse a través de la actividad de comunicación. Un cliente tiene un problema que parece abordable mediante una solución basada en computadora. Usted responde a la solicitud de ayuda del cliente. Ha comenzado la comunicación. Pero es frecuente que el camino que lleva de la comunicación a la comprensión esté lleno de agujeros.

La comunicación efectiva (entre colegas técnicos, con el cliente y otros participantes, y con los gerentes de proyecto) se encuentra entre las actividades más difíciles que deben enfrentarse. En este contexto, aquí se estudian principios de comunicación aplicados a la comunicación con el cliente. Sin embargo, muchos de ellos se aplican por igual en todas las formas de comunicación que ocurren dentro de un proyecto de software.

Principio 1. *Escuchar.* Trate de centrarse en las palabras del hablante en lugar de formular su respuesta a dichas palabras. Si algo no está claro, pregunte para aclararlo, pero evite las interrupciones constantes. Si una persona habla, *nunca* parezca usted beligerante en sus palabras o actos (por ejemplo, con giros de los ojos o movimientos de la cabeza).

Antes de comunicarse, asegúrese de que entiende el punto de vista de la otra parte, conozca un poco sus necesidades y después escuche.

Principio 2. ***Antes de comunicarse, prepararse.*** Dedique algún tiempo a entender el problema antes de reunirse con otras personas. Si es necesario, haga algunas investigaciones para entender el vocabulario propio del negocio. Si tiene la responsabilidad de conducir la reunión, prepare una agenda antes de que ésta tenga lugar.

Principio 3. ***Alguien debe facilitar la actividad.*** Toda reunión de comunicación debe tener un líder (facilitador) que: 1) mantenga la conversación en movimiento hacia una dirección positiva, 2) sea un mediador en cualquier conflicto que ocurra y 3) garantice que se sigan otros principios.

Cita:

"Las preguntas directas y las respuestas directas son el camino más corto hacia las mayores perplejidades."

Mark Twain

Principio 4. ***Es mejor la comunicación cara a cara.*** Pero por lo general funciona mejor cuando está presente alguna otra representación de la información relevante. Por ejemplo, un participante quizá genere un dibujo o documento en "borrador" que sirva como centro de la discusión.

Principio 5. ***Tomar notas y documentar las decisiones.*** Las cosas encuentran el modo de caer en las grietas. Alguien que participe en la comunicación debe servir como "secretario" y escribir todos los temas y decisiones importantes.

Principio 6. ***Perseguir la colaboración.*** La colaboración y el consenso ocurren cuando el conocimiento colectivo de los miembros del equipo se utiliza para describir funciones o características del producto o sistema. Cada pequeña colaboración sirve para generar confianza entre los miembros del equipo y crea un objetivo común para el grupo.

Principio 7. ***Permanecer centrado; hacer módulos con la discusión.*** Entre más personas participen en cualquier comunicación, más probable es que la conversación salte de un tema a otro. El facilitador debe formar módulos de conversación para abandonar un tema sólo después de que se haya resuelto (sin embargo, considere el principio 9).

Principio 8. ***Si algo no está claro, hacer un dibujo.*** La comunicación verbal tiene sus límites. Con frecuencia, un esquema o dibujo arroja claridad cuando las palabras no bastan para hacer el trabajo.

¿Qué pasa si no puede llegarse a un acuerdo con el cliente en algún aspecto relacionado con el proyecto?

Principio 9. ***a) Una vez que se acuerde algo, avanzar. b) Si no es posible ponerse de acuerdo en algo, avanzar. c) Si una característica o función no está clara o no puede aclararse en el momento, avanzar.*** La comunicación, como cualquier actividad de ingeniería de software, requiere tiempo. En vez de hacer iteraciones sin fin, las personas que participan deben reconocer que hay muchos temas que requieren análisis (véase el principio 2) y que "avanzar" es a veces la mejor forma de tener agilidad en la comunicación.

Principio 10. ***La negociación no es un concurso o un juego. Funciona mejor cuando las dos partes ganan.*** Hay muchas circunstancias en las que usted y otros participantes deben negociar funciones y características, prioridades y fechas de entrega. Si el equipo ha

INFORMACIÓN

La diferencia entre los clientes y los usuarios finales

Los ingenieros de software se comunican con muchos participantes diferentes, pero los clientes y los usuarios finales son quienes tienen el efecto más significativo en el trabajo técnico que se desarrollará. En ciertos casos, el cliente y el usuario final son la misma persona, pero para muchos proyectos son individuos distintos que trabajan para diferentes gerentes en distintas organizaciones de negocios.

Un *cliente* es la persona o grupo que 1) solicitó originalmente que se construyera el software, 2) define los objetivos generales del negocio para el software, 3) proporciona los requerimientos básicos del producto y 4) coordina la obtención de fondos para el proyecto. En un negocio de productos o sistema, es frecuente que el cliente sea el departamento de mercadotecnia. En un ambiente de tecnologías de la información (TI), el cliente tal vez sea un componente o departamento del negocio.

Un *usuario final* es la persona o grupo que 1) usará en realidad el software que se elabore para lograr algún propósito del negocio y 2) definirá los detalles de operación del software de modo que se alcance el propósito del negocio.

CasaSegura

Errores de comunicación

La escena: Lugar de trabajo del equipo de ingeniería de software.

Participantes: Jamie Lazar, Vinod Roman y Ed Robins, miembros del equipo de software.

La conversación:

Ed: ¿Qué has oído sobre el proyecto *CasaSegura*?

Vinod: La reunión de arranque está programada para la semana siguiente.

Jamie: Traté de investigar algo, pero no salió bien.

Ed: ¿Qué quieres decir?

Jamie: Bueno, llamé a Lisa Pérez. Ella es la encargada de mercadotecnia en esto.

Vinod: ¿Y…?

Jamie: Yo quería que me dijera las características y funciones de *CasaSegura*… esa clase de cosas. En lugar de ello, comenzó a hacerme preguntas sobre sistemas de seguridad, de vigilancia… No soy experto en eso.

Vinod: ¿Qué te dice eso?

(Jamie se encoge de hombros.)

Vinod: Será que mercadotecnia quiere que actuemos como consultores y mejor que hagamos alguna tarea sobre esta área de productos antes de nuestra junta de arranque. Doug dijo que quería que "colaboráramos" con nuestro cliente, así que será mejor que aprendamos cómo hacerlo.

Ed: Tal vez hubiera sido mejor ir a su oficina. Las llamadas por teléfono simplemente no sirven para esta clase de trabajos.

Jamie: Están en lo correcto. Tenemos que actuar juntos o nuestras primeras comunicaciones serán una batalla.

Vinod: Yo vi a Doug leyendo un libro acerca de "requerimientos de ingeniería". Apuesto a que enlista algunos principios de buena comunicación. Voy a pedírselo prestado.

Jamie: Buena idea… luego nos enseñas.

Vinod (sonríe): Sí, de acuerdo.

colaborado bien, todas las partes tendrán un objetivo común. Aun así, la negociación demandará el compromiso de todas las partes.

4.3.2 Principios de planeación

La actividad de comunicación ayuda a definir las metas y objetivos generales (por supuesto, sujetos al cambio conforme pasa el tiempo). Sin embargo, la comprensión de estas metas y objetivos no es lo mismo que definir un plan para lograrlo. La actividad de planeación incluye un conjunto de prácticas administrativas y técnicas que permiten que el equipo de software defina un mapa mientras avanza hacia su meta estratégica y sus objetivos tácticos.

Cita:

"Al prepararme para una batalla siempre descubro que los planes son inútiles, pero que la planeación es indispensable."

General Dwight D. Eisenhower

Créalo, es imposible predecir con exactitud cómo se desarrollará un proyecto de software. No existe una forma fácil de determinar qué problemas técnicos se encontrarán, qué información importante permanecerá oculta hasta que el proyecto esté muy avanzado, qué malos entendidos habrá o qué aspectos del negocio cambiarán. No obstante, un buen equipo de software debe planear con este enfoque.

Hay muchas filosofías de planeación.[2] Algunas personas son "minimalistas" y afirman que es frecuente que el cambio elimine la necesidad de hacer un plan detallado. Otras son "tradicionalistas" y dicen que el plan da un mapa eficaz y que entre más detalles tenga menos probable será que el equipo se pierda. Otros más son "agilistas" y plantean que tal vez sea necesario un "juego de planeación" rápido, pero que el mapa surgirá a medida que comience el "trabajo real" con el software.

WebRef

En la dirección **www.4pm.com/repository.htm**, hay excelentes materiales informativos sobre la planeación y administración de proyectos.

¿Qué hacer? En muchos proyectos, planear en exceso consume tiempo y es estéril (porque son demasiadas las cosas que cambian), pero planear poco es una receta para el caos. Igual que la mayoría de cosas de la vida, la planeación debe ser tomada con moderación, suficiente para que dé una guía útil al equipo, ni más ni menos. Sin importar el rigor con el que se haga la planeación, siempre se aplican los principios siguientes:

2 En la parte 4 de este libro hay un análisis detallado de la planeación y administración de proyectos de software.

Principio 1. ***Entender el alcance del proyecto.*** Es imposible usar el mapa si no se sabe a dónde se va. El alcance da un destino al equipo de software.

Principio 2. ***Involucrar en la actividad de planeación a los participantes del software.*** Los participantes definen las prioridades y establecen las restricciones del proyecto. Para incluir estas realidades, es frecuente que los ingenieros de software deban negociar la orden de entrega, los plazos y otros asuntos relacionados con el proyecto.

Principio 3. ***Reconocer que la planeación es iterativa.*** Un plan para el proyecto nunca está grabado en piedra. Para cuando el trabajo comience, es muy probable que las cosas hayan cambiado. En consecuencia, el plan deberá ajustarse para incluir dichos cambios. Además, los modelos de proceso iterativo incrementales dictan que debe repetirse la planeación después de la entrega de cada incremento de software, con base en la retroalimentación recibida de los usuarios.

Cita:

"El éxito es más una función del sentido común coherente que del genio."

An Wang

Principio 4. ***Estimar con base en lo que se sabe.*** El objetivo de la estimación es obtener un índice del esfuerzo, costo y duración de las tareas, con base en la comprensión que tenga el equipo sobre el trabajo que va a realizar. Si la información es vaga o poco confiable, entonces las estimaciones tampoco serán confiables.

Principio 5. ***Al definir el plan, tomar en cuenta los riesgos.*** Si ha identificado riesgos que tendrían un efecto grande y es muy probable que ocurran, entonces es necesario elaborar planes de contingencia. Además, el plan del proyecto (incluso la programación de actividades) deberá ajustarse para que incluya la posibilidad de que ocurran uno o más de dichos riesgos.

Principio 6. ***Ser realista.*** Las personas no trabajan 100% todos los días. En cualquier comunicación humana hay ruido. Las omisiones y ambigüedad son fenómenos de la vida. Los cambios ocurren. Aun los mejores ingenieros de software cometen errores. Éstas y otras realidades deben considerarse al establecer un proyecto.

PUNTO CLAVE

El término *granularidad* se refiere al detalle con el que se representan o efectúan algunos elementos de la planeación.

Principio 7. ***Ajustar la granularidad cuando se defina el plan.*** La *granularidad* se refiere al nivel de detalle que se adopta cuando se desarrolla un plan. Un plan con "mucha granularidad" proporciona detalles significativos en las tareas para el trabajo que se planea, en incrementos durante un periodo relativamente corto (por lo que el seguimiento y control suceden con frecuencia). Un plan con "poca granularidad" da tareas más amplias para el trabajo que se planea, para plazos más largos. En general, la granularidad va de poca a mucha conforme el tiempo avanza. En las siguientes semanas o meses, el proyecto se planea con detalles significativos. Las actividades que no ocurrirán en muchos meses no requieren mucha granularidad (hay demasiadas cosas que pueden cambiar).

Principio 8. ***Definir cómo se trata de asegurar la calidad.*** El plan debe identificar la forma en la que el equipo de software busca asegurar la calidad. Si se realizan revisiones técnicas,[3] deben programarse. Si durante la construcción va a utilizarse programación por parejas (véase el capítulo 3), debe definirse en forma explícita en el plan.

Principio 9. ***Describir cómo se busca manejar el cambio.*** Aun la mejor planeación puede ser anulada por el cambio sin control. Debe identificarse la forma en la que van a recibirse los cambios a medida que avanza el trabajo de la ingeniería de software. Por ejemplo, ¿el cliente tiene la posibilidad de solicitar un cambio en cualquier momento? Si se solicita uno, ¿está obligado el equipo a implementarlo de inmediato? ¿Cómo se evalúan el efecto y el costo del cambio?

3 Las revisiones técnicas se estudian en el capítulo 15.

Principio 10. ***Dar seguimiento al plan con frecuencia y hacer los ajustes que se requieran.*** Los proyectos de software se atrasan respecto de su programación. Por tanto, tiene sentido evaluar diariamente el avance, en busca de áreas y situaciones problemáticas en las que las actividades programadas no se apeguen al avance real. Cuando se detecten desviaciones, hay que ajustar el plan en consecuencia.

Para ser más eficaz, cada integrante del equipo de software debe participar en la actividad de planeación. Sólo entonces sus miembros "firmarán" el plan.

4.3.3 Principios de modelado

Se crean modelos para entender mejor la entidad real que se va a construir. Cuando ésta es física (por ejemplo, un edificio, un avión, una máquina, etc.), se construye un modelo de forma idéntica pero a escala. Sin embargo, cuando la entidad que se va a construir es software, el modelo debe adoptar una forma distinta. Debe ser capaz de representar la información que el software transforma, la arquitectura y las funciones que permiten que esto ocurra, las características que desean los usuarios y el comportamiento del sistema mientras la transformación tiene lugar. Los modelos deben cumplir estos objetivos en diferentes niveles de abstracción, en primer lugar con la ilustración del software desde el punto de vista del cliente y después con su representación en un nivel más técnico.

Los modelos de requerimientos representan los requerimientos del cliente. Los modelos del diseño dan una especificación concreta para la construcción del software.

En el trabajo de ingeniería de software se crean dos clases de modelos: de requerimientos y de diseño. Los *modelos de requerimientos* (también conocidos como *modelos de análisis*) representan los requerimientos del cliente mediante la ilustración del software en tres dominios diferentes: el de la información, el funcional y el de comportamiento. Los *modelos de diseño* representan características del software que ayudan a los profesionales a elaborarlo con eficacia: arquitectura, interfaz de usuario y detalle en el nivel de componente.

En su libro sobre modelado ágil, Scott Ambler y Ron Jeffries [Amb02b] definen un conjunto de principios de modelado[4] dirigidos a todos aquellos que usan el modelo de proceso ágil (véase el capítulo 3), pero que son apropiados para todos los ingenieros de software que efectúan acciones y tareas de modelado:

Principio 1. ***El equipo de software tiene como objetivo principal elaborar software, no crear modelos.*** Agilidad significa entregar software al cliente de la manera más rápida posible. Los modelos que contribuyan a esto son benéficos, pero deben evitarse aquellos que hagan lento el proceso o que den poca perspectiva.

Principio 2. ***Viajar ligero, no crear más modelos de los necesarios.*** Todo modelo que se cree debe actualizarse si ocurren cambios. Más importante aún es que todo modelo nuevo exige tiempo, que de otra manera se destinaría a la construcción (codificación y pruebas). Entonces, cree sólo aquellos modelos que hagan más fácil y rápido construir el software.

El objetivo de cualquier modelo es comunicar información. Para lograr esto, use un formato consistente. Suponga que usted no estará para explicar el modelo. Por eso, el modelo debe describirse por sí solo.

Principio 3. ***Tratar de producir el modelo más sencillo que describa al problema o al software.*** No construya software en demasía [Amb02b]. Al mantener sencillos los modelos, el software resultante también lo será. El resultado es que se tendrá un software fácil de integrar, de probar y de mantener (para que cambie). Además, los modelos sencillos son más fáciles de entender y criticar por parte de los miembros del equipo, lo que da como resultado un formato funcional de retroalimentación que optimiza el resultado final.

Principio 4. ***Construir modelos susceptibles al cambio.*** Suponga que sus modelos cambiarán, pero vigile que esta suposición no lo haga descuidado. Por ejemplo, como los

4 Para fines de este libro, se han abreviado y reescrito los principios mencionados en esta sección.

requerimientos se modificarán, hay una tendencia a ignorar los modelos. ¿Por qué? Porque se sabe que de todos modos cambiarán. El problema con esta actitud es que sin un modelo razonablemente completo de los requerimientos, se creará un diseño (modelo de diseño) que de manera invariable carecerá de funciones y características importantes.

Principio 5. ***Ser capaz de enunciar un propósito explícito para cada modelo que se cree.*** Cada vez que cree un modelo, pregúntese por qué lo hace. Si no encuentra una razón sólida para la existencia del modelo, no pierda tiempo en él.

Principio 6. ***Adaptar los modelos que se desarrollan al sistema en cuestión.*** Tal vez sea necesario adaptar a la aplicación la notación del modelo o las reglas; por ejemplo, una aplicación de juego de video quizá requiera una técnica de modelado distinta que el software incrustado que controla el motor de un automóvil en tiempo real.

Principio 7. ***Tratar de construir modelos útiles, pero olvidarse de elaborar modelos perfectos.*** Cuando un ingeniero de software construye modelos de requerimientos y diseño, alcanza un punto de rendimientos decrecientes. Es decir, el esfuerzo requerido para terminar por completo el modelo y hacerlo internamente consistente deja de beneficiarse por tener dichas propiedades. ¿Se sugiere que el modelado debe ser pobre o de baja calidad? La respuesta es "no". Pero el modelado debe hacerse con la mirada puesta en las siguientes etapas de la ingeniería de software. Las iteraciones sin fin para obtener un modelo "perfecto" no cumplen la necesidad de agilidad.

Principio 8. ***No ser dogmático respecto de la sintaxis del modelo. Si se tiene éxito para comunicar contenido, la representación es secundaria.*** Aunque cada miembro del equipo de software debe tratar de usar una notación consistente durante el modelado, la característica más importante del modelo es comunicar información que permita la realización de la siguiente tarea de ingeniería. Si un modelo tiene éxito en hacer esto, es perdonable la sintaxis incorrecta.

Principio 9. ***Si su instinto dice que un modelo no es el correcto a pesar de que se vea bien en el papel, hay razones para estar preocupado.*** Si usted es un ingeniero de software experimentado, confíe en su instinto. El trabajo de software enseña muchas lecciones, algunas en el nivel del inconsciente. Si algo le dice que un modelo de diseño está destinado a fracasar (aun cuando esto no pueda demostrarse en forma explícita), hay razones para dedicar más tiempo a su estudio o a desarrollar otro distinto.

Principio 10. ***Obtener retroalimentación tan pronto como sea posible.*** Todo modelo debe ser revisado por los miembros del equipo. El objetivo de estas revisiones es obtener retroalimentación para utilizarla a fin de corregir los errores de modelado, cambiar las interpretaciones equivocadas y agregar las características o funciones omitidas inadvertidamente.

Requerimientos de los principios de modelado. En las últimas tres décadas se han desarrollado numerosos métodos de modelado de requerimientos. Los investigadores han identificado los problemas del análisis de requerimientos y sus causas, y han desarrollado varias notaciones de modelado y los conjuntos heurísticos correspondientes para resolver aquéllos. Cada método de análisis tiene un punto de vista único. Sin embargo, todos están relacionados por ciertos principios operacionales:

Principio 1. ***Debe representarse y entenderse el dominio de información de un problema.*** El *dominio de información* incluye los datos que fluyen hacia el sistema (usuarios finales, otros sistemas o dispositivos externos), los datos que fluyen fuera del sistema (por la interfaz de usuario, interfaces de red, reportes, gráficas y otros medios) y los almacenamientos de datos que recaban y organizan objetos persistentes de datos (por ejemplo, aquellos que se conservan en forma permanente).

PUNTO CLAVE

El modelado del análisis se centra en tres atributos del software: la información que se va a procesar, la función que se va a entregar y el comportamiento que va a suceder.

Principio 2. ***Deben definirse las funciones que realizará el software.*** Las funciones del software dan un beneficio directo a los usuarios finales y también brindan apoyo interno para las características que son visibles para aquéllos. Algunas funciones transforman los datos que fluyen hacia el sistema. En otros casos, las funciones activan algún nivel de control sobre el procesamiento interno del software o sobre los elementos externos del sistema. Las funciones se describen en muchos y distintos niveles de abstracción, que van de un enunciado de propósito general a la descripción detallada de los elementos del procesamiento que deben invocarse.

Cita:

"En cualquier trabajo de diseño, el primer problema del ingeniero es descubrir cuál es realmente el problema."

Autor desconocido

Principio 3. ***Debe representarse el comportamiento del software (como consecuencia de eventos externos).*** El comportamiento del software de computadora está determinado por su interacción con el ambiente externo. Las entradas que dan los usuarios finales, el control de los datos efectuado por un sistema externo o la vigilancia de datos reunidos en una red son el motivo por el que el software se comporta en una forma específica.

Principio 4. ***Los modelos que representen información, función y comportamiento deben dividirse de manera que revelen los detalles en forma estratificada (o jerárquica).*** El modelado de los requerimientos es el primer paso para resolver un problema de ingeniería de software. Eso permite entender mejor el problema y proporciona una base para la solución (diseño). Los problemas complejos son difíciles de resolver por completo. Por esta razón, debe usarse la estrategia de *divide y vencerás*. Un problema grande y complejo se divide en subproblemas hasta que cada uno de éstos sea relativamente fácil de entender. Este concepto se llama *partición* o *separación de entidades*, y es una estrategia clave en el modelado de requerimientos.

Principio 5. ***El trabajo de análisis debe avanzar de la información esencial hacia la implementación en detalle.*** El modelado de requerimientos comienza con la descripción del problema desde la perspectiva del usuario final. Se describe la "esencia" del problema sin considerar la forma en la que se implementará la solución. Por ejemplo, un juego de video requiere que la jugadora "enseñe" a su protagonista en qué dirección avanzar cuando se mueve hacia un laberinto peligroso. Ésa es la esencia del problema. La implementación detallada (normalmente descrita como parte del modelo del diseño) indica cómo se desarrollará la esencia. Para el juego de video, quizá se use una entrada de voz, o se escriba un comando en un teclado, o tal vez un *joystick* (o *mouse*) apunte en una dirección específica, o quizá se mueva en el aire un dispositivo sensible al movimiento.

Con la aplicación de estos principios, un ingeniero de software aborda el problema en forma sistemática. Pero, ¿cómo se aplican estos principios en la práctica? Esta pregunta se responderá en los capítulos 5 a 7.

Principios del modelado del diseño. El modelo del diseño del software es análogo a los planos arquitectónicos de una casa. Se comienza por representar la totalidad de lo que se va a construir (por ejemplo, un croquis tridimensional de la casa) que se refina poco a poco para que guíe la construcción de cada detalle (por ejemplo, la distribución de la plomería). De manera similar, el modelo del diseño que se crea para el software da varios puntos de vista distintos del sistema.

Cita:

"Vea primero que el diseño es sabio y justo: eso comprobado, siga resueltamente; no para uno renunciar a rechazar el propósito de que ha resuelto llevar a cabo."

William Shakespeare

No escasean los métodos para obtener los distintos elementos de un diseño de software. Algunos son activados por datos, lo que hace que sea la estructura de éstos la que determine la arquitectura del programa y los componentes de procesamiento resultantes. Otros están motivados por el patrón, y usan información sobre el dominio del problema (el modelo de requerimientos) para desarrollar estilos de arquitectura y patrones de procesamiento. Otros más están orientados a objetos, y utilizan objetos del dominio del problema como impulsores de la creación de estructuras de datos y métodos que los manipulan. No obstante la variedad, todos ellos se apegan a principios de diseño que se aplican sin importar el método empleado.

Principio 1. ***El diseño debe poderse rastrear hasta el modelo de requerimientos.*** El modelo de requerimientos describe el dominio de información del problema, las funciones visibles para el usuario, el comportamiento del sistema y un conjunto de clases de requerimientos que agrupa los objetos del negocio con los métodos que les dan servicio. El modelo de diseño traduce esta información en una arquitectura, un conjunto de subsistemas que implementan las funciones principales y un conjunto de componentes que son la realización de las clases de requerimientos. Los elementos del modelo de diseño deben poder rastrearse en el modelo de requerimientos.

WebRef

En la dirección **cs.wwc.edu/~aabyan/Design/**, se encuentran comentarios profundos sobre el proceso de diseño, así como un análisis de la estética del diseño.

Principio 2. ***Siempre tomar en cuenta la arquitectura del sistema que se va a construir.*** La arquitectura del software (véase el capítulo 9) es el esqueleto del sistema que se va a construir. Afecta interfaces, estructuras de datos, flujo de control y comportamiento del programa, así como la manera en la que se realizarán las pruebas, la susceptibilidad del sistema resultante a recibir mantenimiento y mucho más. Por todas estas razones, el diseño debe comenzar con consideraciones de la arquitectura. Sólo después de establecida ésta deben considerarse los aspectos en el nivel de los componentes.

Principio 3. ***El diseño de los datos es tan importante como el de las funciones de procesamiento.*** El diseño de los datos es un elemento esencial del diseño de la arquitectura. La forma en la que los objetos de datos se elaboran dentro del diseño no puede dejarse al azar. Un diseño de datos bien estructurado ayuda a simplificar el flujo del programa, hace más fácil el diseño e implementación de componentes de software y más eficiente el procesamiento conjunto.

Cita:

"Las diferencias no son menores; por el contrario, son como las que había entre Salieri y Mozart. Un estudio tras otro muestran que los mejores diseñadores elaboran estructuras más rápidas, pequeñas, sencillas, claras y producidas con menos esfuerzo."

Frederick P. Brooks

Principio 4. ***Las interfaces (tanto internas como externas) deben diseñarse con cuidado.*** La manera en la que los datos fluyen entre los componentes de un sistema tiene mucho que ver con la eficiencia del procesamiento, la propagación del error y la simplicidad del diseño. Una interfaz bien diseñada hace que la integración sea más fácil y ayuda a quien la somete a prueba a validar las funciones componentes.

Principio 5. ***El diseño de la interfaz de usuario debe ajustarse a las necesidades del usuario final. Sin embargo, en todo caso debe resaltar la facilidad de uso.*** La interfaz de usuario es la manifestación visible del software. No importa cuán sofisticadas sean sus funciones internas, ni lo incluyentes que sean sus estructuras de datos, ni lo bien diseñada que esté su arquitectura, un mal diseño de la interfaz con frecuencia conduce a la percepción de que el software es "malo".

Principio 6. ***El diseño en el nivel de componentes debe tener independencia funcional.*** La independencia funcional es una medida de la "mentalidad única" de un componente de software. La funcionalidad que entrega un componente debe ser cohesiva, es decir, debe centrarse en una y sólo una función o subfunción.[5]

Principio 7. ***Los componentes deben estar acoplados con holgura entre sí y con el ambiente externo.*** El acoplamiento se logra de muchos modos: con una interfaz de componente, con mensajería, por medio de datos globales, etc. A medida que se incrementa el nivel de acoplamiento, también aumenta la probabilidad de propagación del error y disminuye la facilidad general de dar mantenimiento al software. Entonces, el acoplamiento de componentes debe mantenerse tan bajo como sea razonable.

Principio 8. ***Las representaciones del diseño (modelos) deben entenderse con facilidad.*** El propósito del diseño es comunicar información a los profesionales que generarán el código, a los que probarán el software y a otros que le darán mantenimiento en el futuro. Si el diseño es difícil de entender, no servirá como medio de comunicación eficaz.

5 En el capítulo 8 hay más análisis de la cohesión.

Principio 9. ***El diseño debe desarrollarse en forma iterativa. El diseñador debe buscar más sencillez en cada iteración.*** Igual que ocurre con casi todas las actividades creativas, el diseño ocurre de manera iterativa. Las primeras iteraciones sirven para mejorar el diseño y corregir errores, pero las posteriores deben buscar un diseño tan sencillo como sea posible.

Cuando se aplican en forma apropiada estos principios de diseño, se crea uno que exhibe factores de calidad tanto externos como internos [Mye78]. Los *factores de calidad externos* son aquellas propiedades del software fácilmente observables por los usuarios (por ejemplo, velocidad, confiabilidad, corrección y usabilidad). Los *factores de calidad internos* son de importancia para los ingenieros de software. Conducen a un diseño de alta calidad desde el punto de vista técnico. Para obtener factores de calidad internos, el diseñador debe entender los conceptos básicos del diseño (véase el capítulo 8).

4.3.4 Principios de construcción

Cita:

"Durante gran parte de mi vida he sido un mirón del software, y observo furtivamente el código sucio de otras personas. A veces encuentro una verdadera joya, un programa bien estructurado escrito en un estilo consistente, libre de errores, desarrollado de modo que cada componente es sencillo y organizado, y que está diseñado de modo que el producto es fácil de cambiar."

David Parnas

La actividad de construcción incluye un conjunto de tareas de codificación y pruebas que lleva a un software operativo listo para entregarse al cliente o usuario final. En el trabajo de ingeniería de software moderna, la codificación puede ser 1) la creación directa de lenguaje de programación en código fuente (por ejemplo, Java), 2) la generación automática de código fuente que usa una representación intermedia parecida al diseño del componente que se va a construir o 3) la generación automática de código ejecutable que utiliza un "lenguaje de programación de cuarta generación" (por ejemplo, Visual C++).

Las pruebas dirigen su atención inicial al componente, y con frecuencia se denomina *prueba unitaria*. Otros niveles de pruebas incluyen 1) *de integración* (realizadas mientras el sistema está en construcción), 2) *de validación*, que evalúan si los requerimientos se han satisfecho para todo el sistema (o incremento de software) y 3) *de aceptación*, que efectúa el cliente en un esfuerzo por utilizar todas las características y funciones requeridas. Los siguientes principios y conceptos son aplicables a la codificación y prueba:

Principios de codificación. Los principios que guían el trabajo de codificación se relacionan de cerca con el estilo, lenguajes y métodos de programación. Sin embargo, puede enunciarse cierto número de principios fundamentales:

Principios de preparación: ***Antes de escribir una sola línea de código, asegúrese de:***

- Entender el problema que se trata de resolver.
- Comprender los principios y conceptos básicos del diseño.
- Elegir un lenguaje de programación que satisfaga las necesidades del software que se va a elaborar y el ambiente en el que operará.
- Seleccionar un ambiente de programación que disponga de herramientas que hagan más fácil su trabajo.
- Crear un conjunto de pruebas unitarias que se aplicarán una vez que se haya terminado el componente a codificar.

Evite desarrollar un programa elegante que resuelva el problema equivocado. Ponga especial atención al primer principio de preparación.

Principios de programación: ***Cuando comience a escribir código, asegúrese de:***

- Restringir sus algoritmos por medio del uso de programación estructurada [Boh00].
- Tomar en consideración el uso de programación por parejas.
- Seleccionar estructuras de datos que satisfagan las necesidades del diseño.
- Entender la arquitectura del software y crear interfaces que son congruentes con ella.
- Mantener la lógica condicional tan sencilla como sea posible.

- Crear lazos anidados en forma tal que se puedan probar con facilidad.
- Seleccionar nombres significativos para las variables y seguir otros estándares locales de codificación.
- Escribir código que se documente a sí mismo.
- Crear una imagen visual (por ejemplo, líneas con sangría y en blanco) que ayude a entender.

Principios de validación: ***Una vez que haya terminado su primer intento de codificación, asegúrese de:***

- Realizar el recorrido del código cuando sea apropiado.
- Llevar a cabo pruebas unitarias y corregir los errores que se detecten.
- Rediseñar el código.

WebRef

En la dirección **www.literateprogramming.com/fpstyle.html**, hay una amplia variedad de vínculos a estándares de codificación.

Se han escrito más libros sobre programación (codificación) y sobre los principios y conceptos que la guían que sobre cualquier otro tema del proceso de software. Los libros sobre el tema incluyen obras tempranas sobre estilo de programación [Ker78], construcción de software práctico [McC04], perlas de programación [Ben99], el arte de programar [Knu98], temas pragmáticos de programación [Hun99] y muchísimos temas más. El análisis exhaustivo de estos principios y conceptos está más allá del alcance de este libro. Si tiene interés en profundizar, estudie una o varias de las referencias que se mencionan.

Principios de la prueba. En un libro clásico sobre las pruebas de software, Glen Myers [Mye79] enuncia algunas reglas que sirven bien como objetivos de prueba:

? ¿Cuáles son los objetivos de probar el software?

- La prueba es el proceso que ejecuta un programa con objeto de encontrar un error.
- Un buen caso de prueba es el que tiene alta probabilidad de encontrar un error que no se ha detectado hasta el momento.
- Una prueba exitosa es la que descubre un error no detectado hasta el momento.

CONSEJO

En un contexto amplio del diseño de software, recuerde que se comienza "por lo grande" y se centra en la arquitectura del software, y que se termina "en lo pequeño" y se atiende a los componentes. Para la prueba sólo se invierte el proceso.

Estos objetivos implican un cambio muy grande en el punto de vista de ciertos desarrolladores de software. Ellos avanzan contra la opinión común de que una prueba exitosa es aquella que no encuentra errores en el software. El objetivo es diseñar pruebas que detecten de manera sistemática diferentes clases de errores, y hacerlo con el mínimo tiempo y esfuerzo.

Si las pruebas se efectúan con éxito (de acuerdo con los objetivos ya mencionados), descubrirán errores en el software. Como beneficio secundario, la prueba demuestra que las funciones de software parecen funcionar de acuerdo con las especificaciones, y que los requerimientos de comportamiento y desempeño aparentemente se cumplen. Además, los datos obtenidos conforme se realiza la prueba dan una buena indicación de la confiabilidad del software y ciertas indicaciones de la calidad de éste como un todo. Pero las pruebas no pueden demostrar la inexistencia de errores y defectos; sólo demuestran que hay errores y defectos. Es importante recordar esto (que de otro modo parecería muy pesimista) cuando se efectúe una prueba.

Davis [Dav95b] sugiere algunos principios para las pruebas,[6] que se han adaptado para usarlos en este libro:

Principio 1. ***Todas las pruebas deben poder rastrearse hasta los requerimientos del cliente.***[7] El objetivo de las pruebas de software es descubrir errores. Entonces, los defec-

6 Aquí sólo se mencionan pocos de los principios de prueba de Davis. Para más información, consulte [Dav95b].

7 Este principio se refiere a las *pruebas funcionales*, por ejemplo, aquellas que se centran en los requerimientos. Las *pruebas estructurales* (las que se centran en los detalles de arquitectura o lógica) tal vez no aborden directamente los requerimientos específicos.

tos más severos (desde el punto de vista del cliente) son aquellos que hacen que el programa no cumpla sus requerimientos.

Principio 2. ***Las pruebas deben planearse mucho antes de que den comienzo.*** La planeación de las pruebas (véase el capítulo 17) comienza tan pronto como se termina el modelo de requerimientos. La definición detallada de casos de prueba principia apenas se ha concluido el modelo de diseño. Por tanto, todas las pruebas pueden planearse y diseñarse antes de generar cualquier código.

Principio 3. ***El principio de Pareto se aplica a las pruebas de software.*** En este contexto, el principio de Pareto implica que 80% de todos los errores no detectados durante las pruebas se relacionan con 20% de todos los componentes de programas. Por supuesto, el problema es aislar los componentes sospechosos y probarlos a fondo.

Principio 4. ***Las pruebas deben comenzar "en lo pequeño" y avanzar hacia "lo grande".*** Las primeras pruebas planeadas y ejecutadas por lo general se centran en componentes individuales. Conforme avanzan las pruebas, la atención cambia en un intento por encontrar errores en grupos integrados de componentes y, en última instancia, en todo el sistema.

Principio 5. ***No son posibles las pruebas exhaustivas.*** Hasta para un programa de tamaño moderado, el número de permutaciones de las rutas es demasiado grande. Por esta razón, durante una prueba es imposible ejecutar todas las combinaciones de rutas. Sin embargo, es posible cubrir en forma adecuada la lógica del programa y asegurar que se han probado todas las condiciones en el nivel de componentes.

4.3.5 Principios de despliegue

Como se dijo en la parte 1 del libro, la actividad del despliegue incluye tres acciones: entrega, apoyo y retroalimentación. Como la naturaleza de los modelos del proceso del software moderno es evolutiva o incremental, el despliegue ocurre no una vez sino varias, a medida que el software avanza hacia su conclusión. Cada ciclo de entrega pone a disposición de los clientes y usuarios finales un incremento de software operativo que brinda funciones y características utilizables. Cada ciclo de apoyo provee documentación y ayuda humana para todas las funciones y características introducidas durante los ciclos de despliegue realizados hasta ese momento. Cada ciclo de retroalimentación da al equipo de software una guía importante que da como resultado modificaciones de las funciones, de las características y del enfoque adoptado para el siguiente incremento.

Asegúrese de que su cliente sabe lo que puede esperar antes de que se entregue un incremento de software. De otra manera, puede apostar a que el cliente espera más de lo que usted le dará.

La entrega de un incremento de software representa un punto de referencia importante para cualquier proyecto de software. Cuando el equipo se prepara para entregar un incremento, deben seguirse ciertos principios clave:

Principio 1. ***Deben manejarse las expectativas de los clientes.*** Con demasiada frecuencia, el cliente espera más de lo que el equipo ha prometido entregar, y la desilusión llega de inmediato. Esto da como resultado que la retroalimentación no sea productiva y arruine la moral del equipo. En su libro sobre la administración de las expectativas, Naomi Karten [Kar94] afirma que "el punto de inicio de la administración de las expectativas es ser más consciente de lo que se comunica y de la forma en la que esto se hace". Ella sugiere que el ingeniero de software debe tener cuidado con el envío de mensajes conflictivos al cliente (por ejemplo, prometer más de lo que puede entregarse de manera razonable en el plazo previsto, o entregar más de lo que se prometió en un incremento de software y para el siguiente entregar menos).

Principio 2. ***Debe ensamblarse y probarse el paquete completo que se entregará.*** Debe ensamblarse en un CD-ROM u otro medio (incluso descargas desde web) todo el software ejecutable, archivos de datos de apoyo, documentos de ayuda y otra información rele-

vante, para después hacer una prueba beta exhaustiva con usuarios reales. Todos los *scripts* de instalación y otras características de operación deben ejecutarse por completo en tantas configuraciones diferentes de cómputo como sea posible (por ejemplo, hardware, sistemas operativos, equipos periféricos, configuraciones de red, etcétera).

Principio 3. ***Antes de entregar el software, debe establecerse un régimen de apoyo.*** Un usuario final espera respuesta e información exacta cuando surja una pregunta o problema. Si el apoyo es *ad hoc*, o, peor aún, no existe, el cliente quedará insatisfecho de inmediato. El apoyo debe planearse, los materiales respectivos deben prepararse y los mecanismos apropiados de registro deben establecerse a fin de que el equipo de software realice una evaluación categórica de las clases de apoyo solicitado.

Principio 4. Se ***deben proporcionar a los usuarios finales materiales de aprendizaje apropiados.*** El equipo de software entrega algo más que el software en sí. Deben desarrollarse materiales de capacitación apropiados (si se requirieran); es necesario proveer lineamientos para solución de problemas y, cuando sea necesario, debe publicarse "lo que es diferente en este incremento de software".[8]

Principio 5. ***El software defectuoso debe corregirse primero y después entregarse.*** Cuando el tiempo apremia, algunas organizaciones de software entregan incrementos de baja calidad con la advertencia de que los errores "se corregirán en la siguiente entrega". Esto es un error. Hay un adagio en el negocio del software que dice así: "Los clientes olvidarán pronto que entregaste un producto de alta calidad, pero nunca olvidarán los problemas que les causó un producto de mala calidad. El software se los recuerda cada día."

El software entregado brinda beneficios al usuario final, pero también da retroalimentación útil para el equipo que lo desarrolló. Cuando el incremento se libere, debe invitarse a los usuarios finales a que comenten acerca de características y funciones, facilidad de uso, confiabilidad y cualesquiera otras características.

4.4 Resumen

La práctica de la ingeniería de software incluye principios, conceptos, métodos y herramientas que los ingenieros de software aplican en todo el proceso de desarrollo. Todo proyecto de ingeniería de software es diferente. No obstante, existe un conjunto de principios generales que se aplican al proceso como un todo y a cada actividad estructural, sin importar cuál sea el proyecto o el producto.

Existe un conjunto de principios fundamentales que ayudan en la aplicación de un proceso de software significativo y en la ejecución de métodos de ingeniería de software eficaz. En el nivel del proceso, los principios fundamentales establecen un fundamento filosófico que guía al equipo de software cuando avanza por el proceso del software. En el nivel de la práctica, los principios fundamentales establecen un conjunto de valores y reglas que sirven como guía al analizar el diseño de un problema y su solución, al implementar ésta y al someterla a prueba para, finalmente, desplegar el software en la comunidad del usuario.

Los principios de comunicación se centran en la necesidad de reducir el ruido y mejorar el ancho de banda durante la conversación entre el desarrollador y el cliente. Ambas partes deben colaborar a fin de lograr la mejor comunicación.

Los principios de planeación establecen lineamientos para elaborar el mejor mapa del proceso hacia un sistema o producto terminado. El plan puede diseñarse sólo para un incremento

8 Durante la actividad de comunicación, el equipo de software debe determinar los tipos de materiales de ayuda que quiere el usuario.

del software, o para todo el proyecto. Sin que esto importe, debe definir lo que se hará, quién lo hará y cuándo se terminará el trabajo.

El modelado incluye tanto el análisis como el diseño, y describe representaciones cada vez más detalladas del software. El objetivo de los modelos es afirmar el entendimiento del trabajo que se va a hacer y dar una guía técnica a quienes implementarán el software. Los principios de modelado dan fundamento a los métodos y notación que se utilizan para crear representaciones del software.

La construcción incorpora un ciclo de codificación y pruebas en el que se genera código fuente para cierto componente y es sometido a pruebas. Los principios de codificación definen las acciones generales que deben tener lugar antes de que se escriba el código, mientras se escribe y una vez terminado. Aunque hay muchos principios para las pruebas, sólo uno predomina: la prueba es el proceso que lleva a ejecutar un programa con objeto de encontrar un error.

El despliegue ocurre cuando se presenta al cliente un incremento de software, e incluye la entrega, apoyo y retroalimentación. Los principios clave para la entrega consideran la administración de las expectativas del cliente y darle información de apoyo adecuada sobre el software. El apoyo demanda preparación anticipada. La retroalimentación permite al cliente sugerir cambios que tengan valor para el negocio y que brinden al desarrollador información para el ciclo iterativo siguiente de ingeniería de software.

Problemas y puntos por evaluar

4.1. Toda vez que la búsqueda de la calidad reclama recursos y tiempo, ¿es posible ser ágil y centrarse en ella?

4.2. De los ocho principios fundamentales que guían el proceso (lo que se estudió en la sección 4.2.1), ¿cuál cree que sea el más importante?

4.3. Describa con sus propias palabras el concepto de *separación de entidades.*

4.4. Un principio de comunicación importante establece que hay que "prepararse antes de comunicarse". ¿Cómo debe manifestarse esta preparación en los primeros trabajos que se hacen? ¿Qué productos del trabajo son resultado de la preparación temprana?

4.5. Haga algunas investigaciones acerca de cómo "facilitar" la actividad de comunicación (use las referencias que se dan u otras distintas) y prepare algunos lineamientos que se centren en la facilitación.

4.6. ¿En qué difiere la comunicación ágil de la comunicación tradicional de la ingeniería de software? ¿En qué se parecen?

4.7. ¿Por qué es necesario "avanzar"?

4.8. Investigue sobre la "negociación" para la actividad de comunicación y prepare algunos lineamientos que se centren sólo en ella.

4.9. Describa lo que significa *granularidad* en el contexto de la programación de actividades de un proyecto.

4.10. ¿Por qué son importantes los modelos en el trabajo de ingeniería de software? ¿Siempre son necesarios? ¿Hay calificadores para la respuesta que se dio sobre esta necesidad?

4.11. ¿Cuáles son los tres "dominios" considerados durante el modelado de requerimientos?

4.12. Trate de agregar un principio adicional a los que se mencionan en la sección 4.3.4 para la codificación.

4.13. ¿Qué es una prueba exitosa?

4.14. Diga si está de acuerdo o en desacuerdo con el enunciado siguiente: "Como entregamos incrementos múltiples al cliente, no debiéramos preocuparnos por la calidad en los primeros incrementos; en las iteraciones posteriores podemos corregir los problemas. Explique su respuesta.

4.15. ¿Por qué es importante la retroalimentación para el equipo de software?

Lecturas y fuentes de información adicionales

La comunicación con el cliente es una actividad de importancia crítica en la ingeniería de software, pero pocos de sus practicantes dedican tiempo a leer sobre ella. Withall (*Software Requirements Patterns*, Microsoft Press, 2007) presenta varios patrones útiles que analizan problemas en la comunicación. Sutliff (*User-Centred Requirements Engineering*, Springer, 2002) se centra mucho en los retos relacionados con la comunicación. Los libros de Weigers (*Software Requierements*, 2a. ed., Microsoft Press, 2003), Pardee (*To Satisfy and Delight Your Customer*, Dorset House, 1996) y Karten [Kar94] analizan a profundidad los métodos para tener una interacción eficaz con el cliente. Aunque su libro no se centra en el software, Hooks y Farry (*Customer Centered Products*, American Management Association, 2000) presentan lineamientos generales útiles para la comunicación con los clientes. Young (*Effective Requirements Practices*, Addison-Wesley, 2001) pone el énfasis en un "equipo conjunto" de clientes y desarrolladores que recaben los requerimientos en colaboración. Somerville y Kotonya (*Requirements Engineering: Processes and Techniques*, Wiley, 1998) analizan el concepto de "provocación" y las técnicas y otros requerimientos de los principios de ingeniería.

Los conceptos y principios de la comunicación y planeación son estudiados en muchos libros de administración de proyectos. Entre los más útiles se encuentran los de Bechtold (*Essentials of Software Project Management*, 2a. ed., Management Concepts, 2007), Wysocki (*Effective Project Management: Traditional, Adaptive, Extreme*, 4a. ed., Wiley, 2006), Leach (*Lean Project Management: Eight Principles for Success*, BookSurge Publishing, 2006) Hughes (*Software Project Management*, McGraw-Hill, 2005) y Stellman y Greene (*Applied Software Project Management*, O'Reilly Media, Inc., 2005).

Davis [Dav95] hizo una compilación excelente de referencias sobre principios de la ingeniería de software. Además, virtualmente todo libro al respecto contiene un análisis útil de los conceptos y principios para análisis, diseño y prueba. Entre los más utilizados (además de éste, claro) se encuentran los siguientes:

Abran, A., y J. Moore, *SWEBOK: Guide to the Software Engineering Body of Knowledge*, IEEE, 2002.

Christensen, M., y R. Thayer, *A Project Manager's Guide to Software Engineering Best Practices*, IEEE-CS Press (Wiley), 2002.

Jalote, P., *An Integrated Approach to Software Engineering*, Springer, 2006.

Pfleeger, S., *Software Engineering: Theory and Practice*, 3a. ed., Prentice-Hall, 2005.

Schach, S., *Object- Oriented and Classical Software Engineering*, McGraw-Hill, 7a. ed., 2006.

Sommerville, I., *Software Engineering*, 8a. ed., Addison-Wesley, 2006

Estos libros también presentan análisis detallados sobre los principios de modelado y construcción.

Los principios de modelado se estudian en muchos libros dedicados al análisis de requerimientos o diseño de software. Los libros de Lieberman (*The Art of Software Modeling*, Auerbach, 2007), Rosenberg y Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Roques (*UML in Practice,* Wiley, 2004) y Penker y Eriksson (*Business Modeling with UML: Business Patterns at Work*, Wiley, 2001) analizan los principios y métodos de modelado.

Todo ingeniero de software que trate de hacer diseño está obligado a leer el texto de Norman (*The Design of Everyday Things*, Currency/Doubleday, 1990). Winograd y sus colegas (*Bringing Design to Software*, Addison-Wesley, 1996) editaron una excelente colección de ensayos sobre aspectos prácticos del diseño de software. Constantine y Lockwood (*Software for Use,* Addison-Wesley, 1999) presenta los conceptos asociados con el "diseño centrado en el usuario". Tognazzini (*Tog on Software Design*, Addison-Wesley, 1995) presenta una reflexión filosófica útil sobre la naturaleza del diseño y el efecto que tienen las decisiones sobre la calidad y la capacidad del equipo para producir software que agregue mucho valor para su cliente. Stahl y sus colegas (*Model-Driven Software Development: Technology, Engineering*, Wiley, 2006) estudian los principios del desarrollo determinado por el modelo.

Son cientos los libros que abordan uno o más elementos de la actividad de construcción. Kernighan y Plauger [Ker78] escribieron un texto clásico sobre el estilo de programación, McConell [McC93] presenta lineamientos prácticos para la construcción de software, Bentley [Ben99] sugiere una amplia variedad de perlas de la programación, Knuth [Knu99] escribió una serie clásica de tres volúmenes acerca del arte de programar y Hunt [Hun99] sugiere lineamientos pragmáticos para la programación.

Myers y sus colegas (*The Art of Software Testing*, 2a. ed., Wiley, 2004) desarrollaron una revisión importante de su texto clásico y muchos principios importantes para la realización de pruebas. Los libros de Perry (*Effective Methods for Software Testing*, 3a. ed., Wiley 2006), Whittaker (*How to Break Software*, Addison-Wesley, 2002), Kaner y sus colegas (*Lessons Learned in Software Testing*, Wiley, 2001) y Marick (*The Craft of Software Testing*, Prentice-Hall, 1997) presentan por separado conceptos y principios importantes para hacer pruebas, así como muchas guías prácticas.

En internet existe una amplia variedad de fuentes de información sobre la práctica de ingeniería de software. En el sitio web del libro se encuentra una lista actualizada de referencias en la Red Mundial que son relevantes para la ingeniería de software: **www.mhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

5

COMPRENSIÓN DE LOS REQUERIMIENTOS

CONCEPTOS CLAVE

administración de los requerimientos 105
casos de uso 113
colaboración 107
concepción 102
despliegue de la función de calidad 111
elaboración 117
especificación 104
indagación 103
indagación de los requerimientos 108
ingeniería de requerimientos 102
modelo del análisis 117
negociación 121
participantes 106
patrones de análisis 120
productos del trabajo 112
puntos de vista 107
validación 105
validación de los requerimientos 122

Entender los requerimientos de un problema es una de las tareas más difíciles que enfrenta el ingeniero de software. Cuando se piensa por primera vez, no parece tan difícil desarrollar un entendimiento claro de los requerimientos. Después de todo, ¿acaso no sabe el cliente lo que se necesita? ¿No deberían tener los usuarios finales una buena comprensión de las características y funciones que le darán un beneficio? Sorprendentemente, en muchas instancias la respuesta a estas preguntas es "no". E incluso si los clientes y los usuarios finales explican sus necesidades, éstas cambiarán mientras se desarrolla el proyecto.

En el prólogo a un libro escrito por Ralph Young [You01] sobre las prácticas eficaces respecto de los requerimientos, escribí lo siguiente:

> Es la peor de las pesadillas. Un cliente entra a la oficina, toma asiento, lo mira a uno fijamente a los ojos y dice: "Sé que cree que entiende lo que digo, pero lo que usted no entiende es que lo que digo no es lo que quiero decir." Invariablemente, esto pasa cuando ya está avanzado el proyecto, después de que se han hecho compromisos con los plazos de entrega, que hay reputaciones en juego y mucho dinero invertido.
>
> Todos los que hemos trabajado en el negocio de los sistemas y del software durante algunos años hemos vivido la pesadilla descrita, pero pocos hemos aprendido a escapar. Batallamos cuando tratamos de obtener los requerimientos de nuestros clientes. Tenemos problemas para entender la información que obtenemos. Es frecuente que registremos los requerimientos de manera desorganizada y que dediquemos muy poco tiempo a verificar lo que registramos. Dejamos que el cambio nos controle en lugar de establecer mecanismos para controlarlo a él. En pocas palabras, fallamos en establecer un fundamento sólido para el sistema o software. Cada uno de los problemas es difícil. Cuando se combinan, el panorama es atemorizador aun para los gerentes y profesionales más experimentados. Pero hay solución.

UNA MIRADA RÁPIDA

¿Qué es? Antes de comenzar cualquier trabajo técnico es una buena idea aplicar un conjunto de tareas de ingeniería a los requerimientos. Éstas llevarán a la comprensión de cuál será el efecto que tendrá el software en el negocio, qué es lo que quiere el cliente y cómo interactuarán los usuarios finales con el software.

¿Quién lo hace? Los ingenieros de software (que en el mundo de las tecnologías de información a veces son llamados *ingenieros de sistemas* o *analistas*) y todos los demás participantes del proyecto (gerentes, clientes y usuarios) intervienen en la ingeniería de requerimientos.

¿Por qué es importante? Diseñar y construir un elegante programa de cómputo que resuelva el problema equivocado no satisface las necesidades de nadie. Por eso es importante entender lo que el cliente desea antes de comenzar a diseñar y a construir un sistema basado en computadora.

¿Cuáles son los pasos? La ingeniería de requerimientos comienza con la concepción, tarea que define el alcance y la naturaleza del problema que se va a resolver. Va seguida de la indagación, labor que ayuda a los participantes a definir lo que se requiere. Después sigue la elaboración, donde se refinan y modifican los requerimientos básicos. Cuando los participantes definen el problema, tiene lugar una negociación: ¿cuáles son las prioridades, qué es lo esencial, cuándo se requiere? Por último, se especifica el problema de algún modo y luego se revisa o valida para garantizar que hay coincidencia entre la comprensión que usted tiene del problema y la que tienen los participantes.

¿Cuál es el producto final? El objetivo de los requerimientos de ingeniería es proporcionar a todas las partes un entendimiento escrito del problema. Esto se logra por medio de varios productos del trabajo: escenarios de uso, listas de funciones y de características, modelos de requerimientos o especificaciones.

¿Cómo me aseguro de que lo hice bien? Se revisan con los participantes los productos del trabajo de la ingeniería de requerimientos a fin de asegurar que lo que se aprendió es lo que ellos quieren decir en realidad. Aquí cabe una advertencia: las cosas cambiarán aun después de que todas las partes estén de acuerdo, y seguirán cambiando durante todo el proyecto.

Es razonable afirmar que las técnicas que se estudiarán en este capítulo no son una "solución" verdadera para los retos que se mencionaron, pero sí proveen de un enfoque sólido para enfrentarlos.

5.1 Ingeniería de requerimientos

Cita:

"La parte más difícil al construir un sistema de software es decidir qué construir. Ninguna parte del trabajo invalida tanto al sistema resultante si ésta se hace mal. Nada es más difícil de corregir después."

Fred Brooks

El diseño y construcción de software de computadora es difícil, creativo y sencillamente divertido. En realidad, elaborar software es tan atractivo que muchos desarrolladores de software quieren ir directo a él antes de haber tenido el entendimiento claro de lo que se necesita. Argumentan que las cosas se aclararán a medida que lo elaboren, que los participantes en el proyecto podrán comprender sus necesidades sólo después de estudiar las primeras iteraciones del software, que las cosas cambian tan rápido que cualquier intento de entender los requerimientos en detalle es una pérdida de tiempo, que las utilidades salen de la producción de un programa que funcione y que todo lo demás es secundario. Lo que hace que estos argumentos sean tan seductores es que tienen algunos elementos de verdad.[1] Pero todos son erróneos y pueden llevar un proyecto de software al fracaso.

El espectro amplio de tareas y técnicas que llevan a entender los requerimientos se denomina *ingeniería de requerimientos*. Desde la perspectiva del proceso del software, la ingeniería de requerimientos es una de las acciones importantes de la ingeniería de software que comienza durante la actividad de comunicación y continúa en la de modelado. Debe adaptarse a las necesidades del proceso, del proyecto, del producto y de las personas que hacen el trabajo.

PUNTO CLAVE

La ingeniería de requerimientos establece una base sólida para el diseño y la construcción. Sin ésta, el software resultante tiene alta probabilidad de no satisfacer las necesidades del cliente.

La ingeniería de requerimientos tiende un puente para el diseño y la construcción. Pero, ¿dónde se origina el puente? Podría argumentarse que principia en los pies de los participantes en el proyecto (por ejemplo, gerentes, clientes y usuarios), donde se definen las necesidades del negocio, se describen los escenarios de uso, se delinean las funciones y características y se identifican las restricciones del proyecto. Otros tal vez sugieran que empieza con una definición más amplia del sistema, donde el software no es más que un componente del dominio del sistema mayor. Pero sin importar el punto de arranque, el recorrido por el puente lo lleva a uno muy alto sobre el proyecto, lo que le permite examinar el contexto del trabajo de software que debe realizarse; las necesidades específicas que deben abordar el diseño y la construcción; las prioridades que guían el orden en el que se efectúa el trabajo, y la información, las funciones y los comportamientos que tendrán un profundo efecto en el diseño resultante.

CONSEJO

Espere hacer un poco de diseño al recabar los requerimientos, y un poco de requerimientos durante el trabajo de diseño.

La ingeniería de requerimientos proporciona el mecanismo apropiado para entender lo que desea el cliente, analizar las necesidades, evaluar la factibilidad, negociar una solución razonable, especificar la solución sin ambigüedades, validar la especificación y administrar los requerimientos a medida de que se transforman en un sistema funcional [Tha97]. Incluye siete tareas diferentes: concepción, indagación, elaboración, negociación, especificación, validación y administración. Es importante notar que algunas de estas tareas ocurren en paralelo y que todas se adaptan a las necesidades del proyecto.

Cita:

"Las semillas de los desastres enormes del software por lo general se vislumbran en los tres primeros meses del inicio del proyecto."

Coper Jones

Concepción. ¿Cómo inicia un proyecto de software? ¿Existe un solo evento que se convierte en el catalizador de un nuevo sistema o producto basado en computadora o la necesidad evoluciona en el tiempo? No hay respuestas definitivas a estas preguntas. En ciertos casos, una conversación casual es todo lo que se necesita para desencadenar un trabajo grande de ingeniería de software. Pero en general, la mayor parte de proyectos comienzan cuando se identifica una necesidad del negocio o se descubre un nuevo mercado o servicio potencial. Los partici-

1 Esto es cierto en particular para los proyectos pequeños (menos de un mes) y muy pequeños, que requieren relativamente poco esfuerzo de software sencillo. A medida que el software crece en tamaño y complejidad, estos argumentos comienzan a ser falsos.

pantes de la comunidad del negocio (por ejemplo, los directivos, personal de mercadotecnia, gerentes de producto, etc.) definen un caso de negocios para la idea, tratan de identificar el ritmo y profundidad del mercado, hacen un análisis de gran visión de la factibilidad e identifican una descripción funcional del alcance del proyecto. Toda esta información está sujeta a cambio, pero es suficiente para desencadenar análisis con la organización de ingeniería de software.[2]

En la concepción del proyecto,[3] se establece el entendimiento básico del problema, las personas que quieren una solución, la naturaleza de la solución que se desea, así como la eficacia de la comunicación y colaboración preliminares entre los otros participantes y el equipo de software.

Indagación. En verdad que parece muy simple: preguntar al cliente, a los usuarios y a otras personas cuáles son los objetivos para el sistema o producto, qué es lo que va a lograrse, cómo se ajusta el sistema o producto a las necesidades del negocio y, finalmente, cómo va a usarse el sistema o producto en las operaciones cotidianas. Pero no es simple: es muy difícil.

Christel y Kang [Cri92] identificaron cierto número de problemas que se encuentran cuando ocurre la indagación:

¿Por qué es difícil llegar al entendimiento claro de lo que quiere el cliente?

- **Problemas de alcance.** La frontera de los sistemas está mal definida o los clientes o usuarios finales especifican detalles técnicos innecesarios que confunden, más que clarifican, los objetivos generales del sistema.
- **Problemas de entendimiento.** Los clientes o usuarios no están completamente seguros de lo que se necesita, comprenden mal las capacidades y limitaciones de su ambiente de computación, no entienden todo el dominio del problema, tienen problemas para comunicar las necesidades al ingeniero de sistemas, omiten información que creen que es "obvia", especifican requerimientos que están en conflicto con las necesidades de otros clientes o usuarios, o solicitan requerimientos ambiguos o que no pueden someterse a prueba.
- **Problemas de volatilidad.** Los requerimientos cambian con el tiempo.

Para superar estos problemas, debe enfocarse la obtención de requerimientos en forma organizada.

CONSEJO

La elaboración es algo bueno, pero hay que saber cuándo detenerse. La clave es describir el problema en forma que establezca una base firme para el diseño. Si se trabaja más allá de este punto, se está haciendo diseño.

Elaboración. La información obtenida del cliente durante la concepción e indagación se expande y refina durante la elaboración. Esta tarea se centra en desarrollar un modelo refinado de los requerimientos (véanse los capítulos 6 y 7) que identifique distintos aspectos de la función del software, su comportamiento e información.

La elaboración está motivada por la creación y mejora de escenarios de usuario que describan cómo interactuará el usuario final (y otros actores) con el sistema. Cada escenario de usuario se enuncia con sintaxis apropiada para extraer clases de análisis, que son entidades del dominio del negocio visibles para el usuario final. Se definen los atributos de cada clase de análisis y se identifican los servicios[4] que requiere cada una de ellas. Se identifican las relaciones y colaboración entre clases, y se producen varios diagramas adicionales.

Negociación. No es raro que los clientes y usuarios pidan más de lo que puede lograrse dado lo limitado de los recursos del negocio. También es relativamente común que distintos clientes

---

2 Si va a desarrollarse un sistema basado en computadora, los análisis comienzan en el contexto de un proceso de ingeniería de sistemas. Para más detalles de la ingeniería de sistemas, visite el sitio web de esta obra.

3 Recuerde que el proceso unificado (véase el capítulo 2) define una "fase de concepción" más amplia que incluye las fases de concepción, indagación y elaboración, que son estudiadas en dicho capítulo.

4 Un *servicio* manipula los datos agrupados por clase. También se utilizan los términos *operación* y *método*. Si no está familiarizado con conceptos de la orientación a objetos, consulte el apéndice 2, en el que se presenta una introducción básica.

En una negociación eficaz no debe haber ganador ni perdedor. Ambos lados ganan porque un "trato" con el que ambas partes pueden vivir es algo sólido.

o usuarios propongan requerimientos conflictivos con el argumento de que su versión es "esencial para nuestras necesidades especiales".

Estos conflictos deben reconciliarse por medio de un proceso de negociación. Se pide a clientes, usuarios y otros participantes que ordenen sus requerimientos según su prioridad y que después analicen los conflictos. Con el empleo de un enfoque iterativo que da prioridad a los requerimientos, se evalúa su costo y riesgo, y se enfrentan los conflictos internos; algunos requerimientos se eliminan, se combinan o se modifican de modo que cada parte logre cierto grado de satisfacción.

Especificación. En el contexto de los sistemas basados en computadora (y software), el término *especificación* tiene diferentes significados para distintas personas. Una especificación puede ser un documento escrito, un conjunto de modelos gráficos, un modelo matemático formal, un conjunto de escenarios de uso, un prototipo o cualquier combinación de éstos.

La formalidad y el formato de una especificación varían con el tamaño y complejidad del software que se va a construir.

Algunos sugieren que para una especificación debe desarrollarse y utilizarse una "plantilla estándar" [Som97], con el argumento de que esto conduce a requerimientos presentados en forma consistente y por ello más comprensible. Sin embargo, en ocasiones es necesario ser flexible cuando se desarrolla una especificación. Para sistemas grandes, el mejor enfoque puede ser un documento escrito que combine descripciones en un lenguaje natural con modelos gráficos. No obstante, para productos o sistemas pequeños que residan en ambientes bien entendidos, quizá todo lo que se requiera sea escenarios de uso.

INFORMACIÓN

Formato de especificación de requerimientos de software

Una *especificación de requerimientos de software* (ERS) es un documento que se crea cuando debe especificarse una descripción detallada de todos los aspectos del software que se va a elaborar, antes de que el proyecto comience. Es importante notar que una ERS formal no siempre está en forma escrita. En realidad, hay muchas circunstancias en las que el esfuerzo dedicado a la ERS estaría mejor aprovechado en otras actividades de la ingeniería de software. Sin embargo, se justifica la ERS cuando el software va a ser desarrollado por una tercera parte, cuando la falta de una especificación crearía problemas severos al negocio, si un sistema es complejo en extremo o si se trata de un negocio de importancia crítica.

Karl Wiegers [Wie03], de la empresa Process Impact Inc., desarrolló un formato útil (disponible en **www.processimpact.com/process_assets/srs_template.doc**) que sirve como guía para aquellos que deben crear una ERS completa. Su contenido normal es el siguiente:

Tabla de contenido
Revisión de la historia

1. Introducción
1.1 Propósito
1.2 Convenciones del documento
1.3 Audiencia objetivo y sugerencias de lectura
1.4 Alcance del proyecto
1.5 Referencias

2. Descripción general
2.1 Perspectiva del producto
2.2 Características del producto
2.3 Clases y características del usuario
2.4 Ambiente de operación
2.5 Restricciones de diseño e implementación
2.6 Documentación para el usuario
2.7 Suposiciones y dependencias

3. Características del sistema
3.1 Característica 1 del sistema
3.2 Característica 2 del sistema (y así sucesivamente)

4. Requerimientos de la interfaz externa
4.1 Interfaces de usuario
4.2 Interfaces del hardware
4.3 Interfaces del software
4.4 Interfaces de las comunicaciones

5. Otros requerimientos no funcionales
5.1 Requerimientos de desempeño
5.2 Requerimientos de seguridad
5.3 Requerimientos de estabilidad
5.4 Atributos de calidad del software

6. Otros requerimientos

Apéndice A: Glosario
Apéndice B: Modelos de análisis
Apéndice C: Lista de conceptos

Puede obtenerse una descripción detallada de cada ERS si se descarga el formato desde la URL mencionada antes.

Validación. La calidad de los productos del trabajo que se generan como consecuencia de la ingeniería de los requerimientos se evalúa durante el paso de validación. La validación de los requerimientos analiza la especificación[5] a fin de garantizar que todos ellos han sido enunciados sin ambigüedades; que se detectaron y corrigieron las inconsistencias, las omisiones y los errores, y que los productos del trabajo se presentan conforme a los estándares establecidos para el proceso, el proyecto y el producto.

Un aspecto clave durante la validación de los requerimientos es la consistencia. Utilice el modelo de análisis para asegurar que los requerimientos se han enunciado de manera consistente.

El mecanismo principal de validación de los requerimientos es la revisión técnica (véase el capítulo 15). El equipo de revisión que los valida incluye ingenieros de software, clientes, usuarios y otros participantes, que analizan la especificación en busca de errores de contenido o de interpretación, de aspectos en los que tal vez se requiera hacer aclaraciones, falta de información, inconsistencias (problema notable cuando se hace la ingeniería de productos o sistemas grandes) y requerimientos en conflicto o irreales (no asequibles).

INFORMACIÓN

Lista de verificación para validar requerimientos

Con frecuencia es útil analizar cada requerimiento en comparación con preguntas de verificación. A continuación se presentan algunas:

- ¿Los requerimientos están enunciados con claridad? ¿Podrían interpretarse mal?
- ¿Está identificada la fuente del requerimiento (por ejemplo, una persona, reglamento o documento)? ¿Se ha estudiado el planteamiento final del requerimiento en comparación con la fuente original?
- ¿El requerimiento está acotado en términos cuantitativos?
- ¿Qué otros requerimientos se relacionan con éste? ¿Están comparados con claridad por medio de una matriz de referencia cruzada u otro mecanismo?
- ¿El requerimiento viola algunas restricciones del dominio?
- ¿Puede someterse a prueba el requerimiento? Si es así, ¿es posible especificar las pruebas (en ocasiones se denominan criterios de validación) para ensayar el requerimiento?
- ¿Puede rastrearse el requerimiento hasta cualquier modelo del sistema que se haya creado?
- ¿Es posible seguir el requerimiento hasta los objetivos del sistema o producto?
- ¿La especificación está estructurada en forma que lleva a entenderlo con facilidad, con referencias y traducción fáciles a productos del trabajo más técnicos?
- ¿Se ha creado un índice para la especificación?
- ¿Están enunciadas con claridad las asociaciones de los requerimientos con las características de rendimiento, comportamiento y operación? ¿Cuáles requerimientos parecen ser implícitos?

Administración de los requerimientos. Los requerimientos para sistemas basados en computadora cambian, y el deseo de modificarlos persiste durante toda la vida del sistema. La administración de los requerimientos es el conjunto de actividades que ayudan al equipo del proyecto a identificar, controlar y dar seguimiento a los requerimientos y a sus cambios en cualquier momento del desarrollo del proyecto.[6] Muchas de estas actividades son idénticas a las técnicas de administración de la configuración del software (TAS) que se estudian en el capítulo 22.

---

5 Recuerde que la naturaleza de la especificación variará con cada proyecto. En ciertos casos, la "especificación" no es más que un conjunto de escenarios de usuario. En otros, la especificación tal vez sea un documento que contiene escenarios, modelos y descripciones escritas.

6 La administración formal de los requerimientos sólo se practica para proyectos grandes que tienen cientos de requerimientos identificables. Para proyectos pequeños, esta actividad tiene considerablemente menos formalidad.

HERRAMIENTAS DE SOFTWARE

Ingeniería de requerimientos

Objetivo: Las herramientas de la ingeniería de los requerimientos ayudan a reunir éstos, a modelarlos, administrarlos y validarlos.

Mecánica: La mecánica de las herramientas varía. En general, éstas elaboran varios modelos gráficos (por ejemplo, UML) que ilustran los aspectos de información, función y comportamiento de un sistema. Estos modelos constituyen la base de todas las demás actividades del proceso de software.

Herramientas representativas:[7]

En el sitio de Volere Requirements, en **www.volere.co.uk/tools.htm**, se encuentra una lista razonablemente amplia (y actualizada) de herramientas para la ingeniería de requerimientos. En los capítulos 6 y 7 se estudian las herramientas que sirven para modelar aquéllos. Las que se mencionan a continuación se centran en su administración.

EasyRM, desarrollada por Cybernetic Intelligence GmbH (**www.easy-rm.com**), construye un diccionario/glosario especial para proyectos, que contiene descripciones y atributos detallados de los requerimientos.

Rational RequisitePro, elaborada por Rational Software (**www-306.ibm.com/software/awdtools/reqpro/**), permite a los usuarios construir una base de datos de requerimientos, representar relaciones entre ellos y organizarlos, indicar su prioridad y rastrearlos.

En el sitio de Volere ya mencionado, se encuentran muchas herramientas adicionales para administrar requerimientos, así como en la dirección **www.jiludwig.com/Requirements_Management_Tools.html**

5.2 ESTABLECER LAS BASES

En el caso ideal, los participantes e ingenieros de software trabajan juntos en el mismo equipo.[8] En esas condiciones, la ingeniería de requerimientos tan sólo consiste en sostener conversaciones significativas con colegas que sean miembros bien conocidos del equipo. Pero es frecuente que en la realidad esto sea muy diferente.

Los clientes o usuarios finales tal vez se encuentren en ciudades o países diferentes, quizá sólo tengan una idea vaga de lo que se requiere, puede ser que tengan opiniones en conflicto sobre el sistema que se va a elaborar, que posean un conocimiento técnico limitado o que dispongan de poco tiempo para interactuar con el ingeniero que recabará los requerimientos. Ninguna de estas posibilidades es deseable, pero todas son muy comunes y es frecuente verse forzado a trabajar con las restricciones impuestas por esta situación.

En las secciones que siguen se estudian las etapas requeridas para establecer las bases que permiten entender los requerimientos de software a fin de que el proyecto comience en forma tal que se mantenga avanzando hacia una solución exitosa.

PUNTO CLAVE

Un *participante* es cualquier persona que tenga interés directo o que se beneficie del sistema que se va a desarrollar.

5.2.1 Identificación de los participantes

Sommerville y Sawyer [Som97] definen *participante* como "cualquier persona que se beneficie en forma directa o indirecta del sistema en desarrollo". Ya se identificaron los candidatos habituales: gerentes de operaciones del negocio, gerentes de producto, personal de mercadotecnia, clientes internos y externos, usuarios finales, consultores, ingenieros de producto, ingenieros de software e ingenieros de apoyo y mantenimiento, entre otros. Cada participante tiene un punto de vista diferente respecto del sistema, obtiene distintos beneficios cuando éste se desarrolla con éxito y corre distintos riesgos si fracasa el esfuerzo de construcción.

---

7 Las herramientas mencionadas aquí no son obligatorias sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

8 Este enfoque es ampliamente recomendable para proyectos que adoptan la filosofía de desarrollo de software ágil.

Durante la concepción, debe hacerse la lista de personas que harán aportes cuando se recaben los requerimientos (véase la sección 5.3). La lista inicial crecerá cuando se haga contacto con los participantes porque a cada uno se le hará la pregunta: "¿A quién más piensa que debe consultarse?"

Cita:

"Ponga a tres participantes en un cuarto y pregúnteles qué clase de sistema quieren. Es probable que escuche cuatro o más opiniones diferentes."

Anónimo

5.2.2 Reconocer los múltiples puntos de vista

Debido a que existen muchos participantes distintos, los requerimientos del sistema se explorarán desde muchos puntos de vista diferentes. Por ejemplo, el grupo de mercadotecnia se interesa en funciones y características que estimularán el mercado potencial, lo que hará que el nuevo sistema sea fácil de vender. Los gerentes del negocio tienen interés en un conjunto de características para que se elabore dentro del presupuesto y que esté listo para ocupar nichos de mercado definidos. Los usuarios finales tal vez quieran características que les resulten familiares y que sean fáciles de aprender y usar. Los ingenieros de software quizá piensen en funciones invisibles para los participantes sin formación técnica, pero que permitan una infraestructura que dé apoyo a funciones y características más vendibles. Los ingenieros de apoyo tal vez se centren en la facilidad del software para recibir mantenimiento.

Cada uno de estos integrantes (y otros más) aportará información al proceso de ingeniería de los requerimientos. A medida que se recaba información procedente de múltiples puntos de vista, los requerimientos que surjan tal vez sean inconsistentes o estén en conflicto uno con otro. Debe clasificarse toda la información de los participantes (incluso los requerimientos inconsistentes y conflictivos) en forma que permita a quienes toman las decisiones escoger para el sistema un conjunto de requerimientos que tenga coherencia interna.

5.2.3 Trabajar hacia la colaboración

Si en un proyecto de software hay involucrados cinco participantes, tal vez se tengan cinco (o más) diferentes opiniones acerca del conjunto apropiado de requerimientos. En los primeros capítulos se mencionó que, para obtener un sistema exitoso, los clientes (y otros participantes) debían colaborar entre sí (sin pelear por insignificancias) y con los profesionales de la ingeniería de software. Pero, ¿cómo se llega a esta colaboración?

El trabajo del ingeniero de requerimientos es identificar las áreas de interés común (por ejemplo, requerimientos en los que todos los participantes estén de acuerdo) y las de conflicto o incongruencia (por ejemplo, requerimientos que desea un participante, pero que están en conflicto con las necesidades de otro). Es la última categoría la que, por supuesto, representa un reto.

INFORMACIÓN

Uso de "puntos de prioridad"

Una manera de resolver requerimientos conflictivos y, al mismo tiempo, mejorar la comprensión de la importancia relativa de todos, es usar un esquema de "votación" con base en *puntos de prioridad*. Se da a todos los participantes cierto número de puntos de prioridad que pueden "gastarse" en cualquier número de requerimientos. Se presenta una lista de éstos y cada participante indica la importancia relativa de cada uno (desde su punto de vista) con la asignación de uno o más puntos de prioridad. Los puntos gastados ya no pueden utilizarse otra vez. Cuando un participante agota sus puntos de prioridad, ya no tiene la posibilidad de hacer algo con los requerimientos. El total de puntos asignados a cada requerimiento por los participantes da una indicación de la importancia general de cada requerimiento.

La colaboración no significa necesariamente que todos los requerimientos los defina un comité. En muchos casos, los participantes colaboran con la aportación de su punto de vista respecto de los requerimientos, pero un influyente "campeón del proyecto" (por ejemplo, el director

del negocio o un tecnólogo experimentado) toma la decisión final sobre los requerimientos que lo integrarán.

5.2.4 Hacer las primeras preguntas

Las preguntas que se hacen en la concepción del proyecto deben estar "libres del contexto" [Gau89]. El primer conjunto de ellas se centran en el cliente y en otros participantes, en las metas y beneficios generales. Por ejemplo, tal vez se pregunte:

Cita:

"Es mejor conocer algunas preguntas que todas las respuestas."

James Thurber

- ¿Quién está detrás de la solicitud de este trabajo?
- ¿Quién usará la solución?
- ¿Cuál será el beneficio económico de una solución exitosa?
- ¿Hay otro origen para la solución que se necesita?

Estas preguntas ayudan a identificar a todos los participantes con interés en el software que se va a elaborar. Además, las preguntas identifican el beneficio mensurable de una implementación exitosa y las posibles alternativas para el desarrollo de software personalizado.

Las preguntas siguientes permiten entender mejor el problema y hacen que el cliente exprese sus percepciones respecto de la solución:

? ¿Cuáles preguntas ayudarían a tener un entendimiento preliminar del problema?

- ¿Cuál sería una "buena" salida generada por una solución exitosa?
- ¿Qué problemas resolvería esta solución?
- ¿Puede mostrar (o describir) el ambiente de negocios en el que se usaría la solución?
- ¿Hay aspectos especiales del desempeño o restricciones que afecten el modo en el que se enfoque la solución?

Las preguntas finales se centran en la eficacia de la actividad de comunicación en sí. Gause y Weinberg [Gau89] las llaman "metapreguntas" y proponen la siguiente lista (abreviada):

Cita:

"El que hace una pregunta es tonto durante cinco minutos; el que no la hace será tonto para siempre."

Proverbio chino

- ¿Es usted la persona indicada para responder estas preguntas? ¿Sus respuestas son "oficiales"?
- ¿Mis preguntas son relevantes para el problema que se tiene?
- ¿Estoy haciendo demasiadas preguntas?
- ¿Puede otra persona dar información adicional?
- ¿Debería yo preguntarle algo más?

Estas preguntas (y otras) ayudarán a "romper el hielo" y a iniciar la comunicación, que es esencial para una indagación exitosa. Pero una reunión de preguntas y respuestas no es un enfoque que haya tenido un éxito apabullante. En realidad, la sesión de preguntas y respuestas sólo debe usarse para el primer encuentro y luego ser reemplazada por un formato de indagación de requerimientos que combine elementos de solución de problemas, negociación y especificación. En la sección 5.3 se presenta un enfoque de este tipo.

5.3 Indagación de los requerimientos

La indagación de los requerimientos (actividad también llamada *recabación de los requerimientos*) combina elementos de la solución de problemas, elaboración, negociación y especificación. A fin de estimular un enfoque colaborativo y orientado al equipo, los participantes trabajan juntos para identificar el problema, proponer elementos de la solución, negociar distintas visiones y especificar un conjunto preliminar de requerimientos para la solución [Zah90].[9]

9 En ocasiones se denomina a este enfoque *técnica facilitada de especificación de la aplicación* (TFEA).

5.3.1 Recabación de los requerimientos en forma colaborativa

Se han propuesto muchos enfoques distintos para recabar los requerimientos en forma colaborativa. Cada uno utiliza un escenario un poco diferente, pero todos son variantes de los siguientes lineamientos básicos:

¿Cuáles son los lineamientos básicos para conducir una reunión a fin de recabar los requerimientos en forma colaborativa?

- Tanto ingenieros de software como otros participantes dirigen o intervienen en las reuniones.
- Se establecen reglas para la preparación y participación.
- Se sugiere una agenda con suficiente formalidad para cubrir todos los puntos importantes, pero con la suficiente informalidad para que estimule el libre flujo de ideas.
- Un "facilitador" (cliente, desarrollador o participante externo) controla la reunión.
- Se utiliza un "mecanismo de definición" (que pueden ser hojas de trabajo, tablas sueltas, etiquetas adhesivas, pizarrón electrónico, grupos de conversación o foro virtual).

Cita:

"Dedicamos mucho tiempo —la mayor parte de todo el esfuerzo del proyecto— no a implementar o hacer pruebas, sino a tratar de decidir qué construir."

Brian Lawrence

La meta es identificar el problema, proponer elementos de la solución, negociar distintos enfoques y especificar un conjunto preliminar de requerimientos de la solución en una atmósfera que favorezca el logro de la meta. Para entender mejor el flujo de eventos conforme ocurren, se presenta un escenario breve que bosqueja la secuencia de hechos que llevan a la reunión para obtener requerimientos, a lo que sucede durante ésta y a lo que sigue después de ella.

Durante la concepción (véase la sección 5.2), hay preguntas y respuestas básicas que establecen el alcance del problema y la percepción general de lo que constituye una solución. Fuera de estas reuniones iniciales, el desarrollador y los clientes escriben una o dos páginas de "solicitud de producto".

Se selecciona un lugar, fecha y hora para la reunión, se escoge un facilitador y se invita a asistir a integrantes del equipo de software y de otras organizaciones participantes. Antes de la fecha de la reunión, se distribuye la solicitud de producto a todos los asistentes.

WebRef

La *solicitud conjunta de desarrollo* (SCD) es una técnica popular para recabar requerimientos. En la dirección **www.carolla.com/wp-jad.htm** se encuentra una buena descripción de ella.

Por ejemplo,[10] considere un extracto de una solicitud de producto escrita por una persona de mercadotecnia involucrada en el proyecto *CasaSegura*. Esta persona escribe la siguiente narración sobre la función de seguridad en el hogar que va a ser parte de *CasaSegura*:

> Nuestras investigaciones indican que el mercado para los sistemas de administración del hogar crece a razón de 40% anual. La primera función de *CasaSegura* que llevemos al mercado deberá ser la de seguridad del hogar. La mayoría de la gente está familiarizada con "sistemas de alarma", por lo que ésta deberá ser fácil de vender.
>
> La función de seguridad del hogar protegería, o reconocería, varias "situaciones" indeseables, como acceso ilegal, incendio y niveles de monóxido de carbono, entre otros. Emplearía sensores inalámbricos para detectar cada situación. Sería programada por el propietario y telefonearía en forma automática a una agencia de vigilancia cuando detectara una situación como las descritas.

CONSEJO

Si un sistema o producto servirá a muchos usuarios, asegúrese de que los requerimientos se obtengan de una franja representativa de ellos. Si sólo uno define todos los requerimientos, el riesgo de no aceptación es elevado.

En realidad, durante la reunión para recabar los requerimientos, otros contribuirían a esta narración y se dispondría de mucha más información. Pero aun con ésta habría ambigüedad, sería probable que existieran omisiones y ocurrieran errores. Por ahora bastará la "descripción funcional" anterior.

Mientras se revisa la solicitud del producto antes de la reunión, se pide a cada asistente que elabore una lista de objetos que sean parte del ambiente que rodeará al sistema, los objetos

10 Este ejemplo (con extensiones y variantes) se usa para ilustrar métodos importantes de la ingeniería de software en muchos de los capítulos siguientes. Como ejercicio, sería provechoso que el lector realizara su propia reunión para recabar requerimientos y que desarrollara un conjunto de listas para ella.

que producirá éste y los que usará para realizar sus funciones. Además, se solicita a cada asistente que haga otra lista de servicios (procesos o funciones) que manipulen o interactúen con los objetos. Por último, también se desarrollan listas de restricciones (por ejemplo, costo, tamaño, reglas del negocio, etc.) y criterios de desempeño (como velocidad y exactitud). Se informa a los asistentes que no se espera que las listas sean exhaustivas, pero sí que reflejen la percepción que cada persona tiene del sistema.

Cita:

"Los hechos no dejan de existir porque se les ignore."

Aldous Huxley

Entre los objetos descritos por *CasaSegura* tal vez estén incluidos el panel de control, detectores de humo, sensores en ventanas y puertas, detectores de movimiento, alarma, un evento (activación de un sensor), una pantalla, una computadora, números telefónicos, una llamada telefónica, etc. La lista de servicios puede incluir *configurar* el sistema, *preparar* la alarma, *vigilar* los sensores, *marcar* el teléfono, *programar* el panel de control y *leer* la pantalla (observe que los servicios actúan sobre los objetos). En forma similar, cada asistente desarrollará una lista de restricciones (por ejemplo, el sistema debe reconocer cuando los sensores no estén operando, debe ser amistoso con el usuario, debe tener una interfaz directa con una línea telefónica estándar, etc.) y de criterios de desempeño (un evento en un sensor debe reconocerse antes de un segundo, debe implementarse un esquema de prioridad de eventos, etcétera).

CONSEJO

Evite el impulso de desechar alguna idea de un cliente con expresiones como "demasiado costosa" o "imprácticа". La intención aquí es negociar una lista aceptable para todos. Para lograrlo, debe tenerse la mente abierta.

Las listas de objetos pueden adherirse a las paredes del cuarto con el empleo de pliegos de papel grandes o con láminas adhesivas, o escribirse en un tablero. Alternativamente, las listas podrían plasmarse en un boletín electrónico, sitio web interno o en un ambiente de grupo de conversación para revisarlas antes de la reunión. Lo ideal es que cada entrada de las listas pueda manipularse por separado a fin de combinar las listas o modificar las entradas y agregar otras. En esta etapa, están estrictamente prohibidos las críticas y el debate.

Una vez que se presentan las listas individuales acerca de un área temática, el grupo crea una lista, eliminando las entradas redundantes o agregando ideas nuevas que surjan durante el análisis, pero no se elimina ninguna. Después de crear listas combinadas para todas las áreas temáticas, sigue el análisis, coordinado por el facilitador. La lista combinada se acorta, se alarga o se modifica su redacción para que refleje de manera apropiada al producto o sistema que se va a desarrollar. El objetivo es llegar a un consenso sobre la lista de objetos, servicios, restricciones y desempeño del sistema que se va a construir.

En muchos casos, un objeto o servicio descrito en la lista requerirá mayores explicaciones. Para lograr esto, los participantes desarrollan *miniespecificaciones* para las entradas en las listas.[11] Cada miniespecificación es una elaboración de un objeto o servicio. Por ejemplo, la correspondiente al objeto **Panel de control** de *CasaSegura* sería así:

> El panel de control es una unidad montada en un muro, sus dimensiones aproximadas son de 9 por 5 pulgadas. Tiene conectividad inalámbrica con los sensores y con una PC. La interacción con el usuario tiene lugar por medio de un tablero que contiene 12 teclas. Una pantalla de cristal líquido de 3 por 3 pulgadas brinda retroalimentación al usuario. El software hace anuncios interactivos, como eco y funciones similares.

Las miniespecificaciones se presentan a todos los participantes para que sean analizadas. Se hacen adiciones, eliminaciones y otras modificaciones. En ciertos casos, el desarrollo de las miniespecificaciones descubrirá nuevos objetos, servicios o restricciones, o requerimientos de desempeño que se agregarán a las listas originales. Durante todos los análisis, el equipo debe posponer los aspectos que no puedan resolverse en la reunión. Se conserva una *lista de aspectos* para volver después a dichas ideas.

---

11 En vez de crear una miniespecificación, muchos equipos de software eligen desarrollar escenarios del usuario llamados *casos de uso*. Éstos se estudian en detalle en la sección 5.4 y en el capítulo 6.

CASASEGURA

Conducción de una reunión para recabar los requerimientos

La escena: Sala de juntas. Está en marcha la primera reunión para recabar los requerimientos.

Participantes: Jamie Lazar, integrante del equipo de software; Vinod Raman, miembro del equipo de software; Ed Robbins, miembro del equipo de software; Doug Miller, gerente de ingeniería de software; tres trabajadores de mercadotecnia; un representante de ingeniería del producto, y un facilitador.

La conversación:

Facilitador (apunta en un pizarrón): De modo que ésa es la lista actual de objetos y servicios para la función de seguridad del hogar.

Persona de mercadotecnia: Eso la cubre, desde nuestro punto de vista.

Vinod: ¿No dijo alguien que quería que toda la funcionalidad de *CasaSegura* fuera accesible desde internet? Eso incluiría la función de seguridad, ¿o no?

Persona de mercadotecnia: Sí, así es… tendremos que añadir esa funcionalidad y los objetos apropiados.

Facilitador: ¿Agrega eso algunas restricciones?

Jamie: Sí, tanto técnicas como legales.

Representante del producto: ¿Qué significa eso?

Jamie: Nos tendríamos que asegurar de que un extraño no pueda ingresar al sistema, desactivarlo y robar en el lugar o hacer algo peor. Mucha responsabilidad sobre nosotros.

Doug: Muy cierto.

Mercadotecnia: Pero lo necesitamos así… sólo asegúrense de impedir que ingrese un extraño.

Ed: Eso es más fácil de decir que de hacer.

Facilitador (interrumpe): No quiero que debatamos esto ahora. Anotémoslo como un aspecto y continuemos.

(Doug, que es el secretario de la reunión, toma debida nota.)

Facilitador: Tengo la sensación de que hay más por considerar aquí.

(El grupo dedica los siguientes 20 minutos a mejorar y aumentar los detalles de la función de seguridad del hogar.)

5.3.2 Despliegue de la función de calidad

El DFC define los requerimientos de forma que maximicen la satisfacción del cliente.

El *despliegue de la función de calidad* (DFC) es una técnica de administración de la calidad que traduce las necesidades del cliente en requerimientos técnicos para el software. El DFC "se concentra en maximizar la satisfacción del cliente a partir del proceso de ingeniería del software" [Zul92]. Para lograr esto, el DFC pone el énfasis en entender lo que resulta valioso para el cliente y luego despliega dichos valores en todo el proceso de ingeniería. El DFC identifica tres tipos de requerimientos [Zul92]:

Todos desean implementar muchos requerimientos emocionantes, pero hay que tener cuidado. Así es como empiezan a "quedar lisiados los requerimientos". Pero en contrapartida, los requerimientos emocionantes llevan a un avance enorme del producto…

Requerimientos normales. Objetivos y metas que se establecen para un producto o sistema durante las reuniones con el cliente. Si estos requerimientos están presentes, el cliente queda satisfecho. Ejemplos de requerimientos normales son los tipos de gráficos pedidos para aparecer en la pantalla, funciones específicas del sistema y niveles de rendimiento definidos.

Requerimientos esperados. Están implícitos en el producto o sistema y quizá sean tan importantes que el cliente no los mencione de manera explícita. Su ausencia causará mucha insatisfacción. Algunos ejemplos de requerimientos esperados son: fácil interacción humano/máquina, operación general correcta y confiable, y facilidad para instalar el software.

Requerimientos emocionantes. Estas características van más allá de las expectativas del cliente y son muy satisfactorias si están presentes. Por ejemplo, el software para un nuevo teléfono móvil viene con características estándar, pero si incluye capacidades inesperadas (como pantalla sensible al tacto, correo de voz visual, etc.) agrada a todos los usuarios del producto.

WebRef

En la dirección **www.qfdi.org** se encuentra información útil sobre el DFC.

Aunque los conceptos del DFC son aplicables en todo el proceso del software [Par96a], hay técnicas específicas de aquél que pueden aplicarse a la actividad de indagación de los requerimientos. El DFC utiliza entrevistas con los clientes, observación, encuestas y estudio de datos históricos (por ejemplo, reportes de problemas) como materia prima para la actividad de recabación

de los requerimientos. Después, estos datos se llevan a una tabla de requerimientos —llamada *tabla de la voz del cliente*— que se revisa con el cliente y con otros participantes. Luego se emplean varios diagramas, matrices y métodos de evaluación para extraer los requerimientos esperados y tratar de percibir requerimientos emocionantes [Aka04].

5.3.3 Escenarios de uso

A medida que se reúnen los requerimientos, comienza a materializarse la visión general de funciones y características del sistema. Sin embargo, es difícil avanzar hacia actividades más técnicas de la ingeniería de software hasta no entender cómo emplearán los usuarios finales dichas funciones y características. Para lograr esto, los desarrolladores y usuarios crean un conjunto de escenarios que identifican la naturaleza de los usos para el sistema que se va a construir. Los escenarios, que a menudo se llaman *casos de uso* [Jac92], proporcionan la descripción de la manera en la que se utilizará el sistema. Los casos de uso se estudian con más detalle en la sección 5.4.

CasaSegura

Desarrollo de un escenario preliminar de uso

La escena: Una sala de juntas, donde continúa la primera reunión para recabar los requerimientos.

Participantes: Jamie Lazar, integrante del equipo de software; Vinod Raman, miembro del equipo de software; Ed Robbins, miembro del equipo de software; Doug Miller, gerente de ingeniería de software; tres personas de mercadotecnia; un representante de ingeniería del producto, y un facilitador.

La conversación:

Facilitador: Hemos estado hablando sobre la seguridad para el acceso a la funcionalidad de *CasaSegura* si ha de ser posible el ingreso por internet. Me gustaría probar algo. Desarrollemos un escenario de uso para entrar a la función de seguridad.

Jamie: ¿Cómo?

Facilitador: Podríamos hacerlo de dos maneras, pero de momento mantengamos las cosas informales. Díganos (señala a una persona de mercadotecnia), ¿cómo visualiza el acceso al sistema?

Persona de mercadotecnia: Um… bueno, es la clase de cosa que haría si estuviera fuera de casa y tuviera que dejar entrar a alguien a ella —por ejemplo, una trabajadora doméstica o un técnico de reparaciones— que no tuviera el código de seguridad.

Facilitador (sonríe): Ésa es la razón por la que lo hace… dígame, ¿cómo lo haría en realidad?

Persona de mercadotecnia: Bueno… lo primero que necesitaría sería una PC. Entraría a un sitio web que mantendríamos para todos los usuarios de *CasaSegura*. Daría mi identificación de usuario y…

Vinod (interrumpe): La página web tendría que ser segura, encriptada, para garantizar que estuviéramos seguros y…

Facilitador (interrumpe): Ésa es buena información, Vinod, pero es técnica. Centrémonos en cómo emplearía el usuario final esta capacidad, ¿está bien?

Vinod: No hay problema.

Persona de mercadotecnia: Decía que entraría a un sitio web y daría mi identificación de usuario y dos niveles de clave.

Jamie: ¿Qué pasa si olvido mi clave?

Facilitador (interrumpe): Buena observación, Jamie, pero no entraremos a ella por ahora. Lo anotaremos y la llamaremos una *excepción*. Estoy seguro de que habrá otras.

Persona de mercadotecnia: Después de que introdujera las claves, aparecería una pantalla que representaría todas las funciones de *CasaSegura*. Seleccionaría la función de seguridad del hogar. El sistema pediría que verificara quién soy, pidiendo mi dirección o número telefónico o algo así. Entonces aparecería un dibujo del panel de control del sistema de seguridad y la lista de funciones que puede realizar —activar el sistema, desactivar el sistema o desactivar uno o más sensores—. Supongo que también me permitiría reconfigurar las zonas de seguridad y otras cosas como ésa, pero no estoy seguro.

(Mientras la persona de mercadotecnia habla, Doug toma muchas notas; esto forma la base para el primer escenario informal de uso. Alternativamente, hubiera podido pedirse a la persona de mercadotecnia que escribiera el escenario, pero esto se hubiera hecho fuera de la reunión.)

5.3.4 Indagación de los productos del trabajo

Los productos del trabajo generados como consecuencia de la indagación de los requerimientos variarán en función del tamaño del sistema o producto que se va a construir. Para la mayoría de sistemas, los productos del trabajo incluyen los siguientes:

¿Qué información se produce como consecuencia de recabar los requerimientos?

- Un enunciado de la necesidad y su factibilidad.
- Un enunciado acotado del alcance del sistema o producto.
- Una lista de clientes, usuarios y otros participantes que intervienen en la indagación de los requerimientos.
- Una descripción del ambiente técnico del sistema.
- Una lista de requerimientos (de preferencia organizados por función) y las restricciones del dominio que se aplican a cada uno.
- Un conjunto de escenarios de uso que dan perspectiva al uso del sistema o producto en diferentes condiciones de operación.
- Cualesquiera prototipos desarrollados para definir requerimientos.

Cada uno de estos productos del trabajo es revisado por todas las personas que participan en la indagación de los requerimientos.

5.4 DESARROLLO DE CASOS DE USO

En un libro que analiza cómo escribir casos de uso eficaces, Alistair Cockburn [Coc01b] afirma que "un caso de uso capta un contrato [...] [que] describe el comportamiento del sistema en distintas condiciones en las que el sistema responde a una petición de alguno de sus participantes [...]". En esencia, un caso de uso narra una historia estilizada sobre cómo interactúa un usuario final (que tiene cierto número de roles posibles) con el sistema en circunstancias específicas. La historia puede ser un texto narrativo, un lineamiento de tareas o interacciones, una descripción basada en un formato o una representación diagramática. Sin importar su forma, un caso de uso ilustra el software o sistema desde el punto de vista del usuario final.

PUNTO CLAVE

Los casos de uso se definen desde el punto de vista de un actor. Un actor es un papel que desempeñan las personas (usuarios) o los dispositivos cuando interactúan con el software.

El primer paso para escribir un caso de uso es definir un conjunto de "actores" que estarán involucrados en la historia. Los *actores* son las distintas personas (o dispositivos) que usan el sistema o producto en el contexto de la función y comportamiento que va a describirse. Los actores representan los papeles que desempeñan las personas (o dispositivos) cuando opera el sistema. Con una definición más formal, un *actor* es cualquier cosa que se comunique con el sistema o producto y que sea externo a éste. Todo actor tiene uno o más objetivos cuando utiliza el sistema.

Es importante notar que un actor y un usuario final no necesariamente son lo mismo. Un usuario normal puede tener varios papeles diferentes cuando usa el sistema, mientras que un actor representa una clase de entidades externas (gente, con frecuencia pero no siempre) que sólo tiene un papel en el contexto del caso de uso. Por ejemplo, considere al operador de una máquina (un usuario) que interactúa con la computadora de control de una celda de manufactura que contiene varios robots y máquinas de control numérico. Después de una revisión cuidadosa de los requerimientos, el software para la computadora de control requiere cuatro diferentes modos (papeles) para la interacción: modo de programación, modo de prueba, modo de vigilancia y modo de solución de problemas. Por tanto, es posible definir cuatro actores: programador, probador, vigilante y solucionador de problemas. En ciertos casos, el operador de la máquina desempeñará todos los papeles. En otros, distintas personas tendrán el papel de cada actor.

WebRef

Un artículo excelente sobre casos de uso puede descargarse desde la dirección **www.ibm.com/developerworks/webservices/library/codesign7.html**

Debido a que la indagación de los requerimientos es una actividad evolutiva, no todos los actores son identificados en la primera iteración. En ésta es posible identificar a los actores principales [Jac92], y a los secundarios cuando se sabe más del sistema. Los *actores principales* interactúan para lograr la función requerida del sistema y obtienen el beneficio previsto de éste. Trabajan con el software en forma directa y con frecuencia. Los *actores secundarios* dan apoyo al sistema, de modo que los primarios puedan hacer su trabajo.

Una vez identificados los actores, es posible desarrollar casos de uso. Jacobson [Jac92] sugiere varias preguntas[12] que debe responder un caso de uso:

¿Qué se necesita saber a fin de desarrollar un caso de uso eficaz?

- ¿Quién es el actor principal y quién(es) el(los) secundario(s)?
- ¿Cuáles son los objetivos de los actores?
- ¿Qué precondiciones deben existir antes de comenzar la historia?
- ¿Qué tareas o funciones principales son realizadas por el actor?
- ¿Qué excepciones deben considerarse al describir la historia?
- ¿Cuáles variaciones son posibles en la interacción del actor?
- ¿Qué información del sistema adquiere, produce o cambia el actor?
- ¿Tendrá que informar el actor al sistema acerca de cambios en el ambiente externo?
- ¿Qué información desea obtener el actor del sistema?
- ¿Quiere el actor ser informado sobre cambios inesperados?

En relación con los requerimientos básicos de *CasaSegura*, se definen cuatro actores: **propietario de la casa** (usuario), **gerente de arranque** (tal vez la misma persona que el **propietario de la casa**, pero en un papel diferente), **sensores** (dispositivos adjuntos al sistema) y **subsistema de vigilancia y respuesta** (estación central que vigila la función de seguridad de la casa de *CasaSegura*). Para fines de este ejemplo, consideraremos sólo al actor llamado **propietario de la casa**. Éste interactúa con la función de seguridad de la casa en varias formas distintas con el empleo del panel de control de la alarma o con una PC:

- Introduce una clave que permita todas las demás interacciones.
- Pregunta sobre el estado de una zona de seguridad.
- Interroga acerca del estado de un sensor.
- En una emergencia, oprime el botón de pánico.
- Activa o desactiva el sistema de seguridad.

Considerando la situación en la que el propietario de la casa usa el panel de control, a continuación se plantea el caso de uso básico para la activación del sistema:[13]

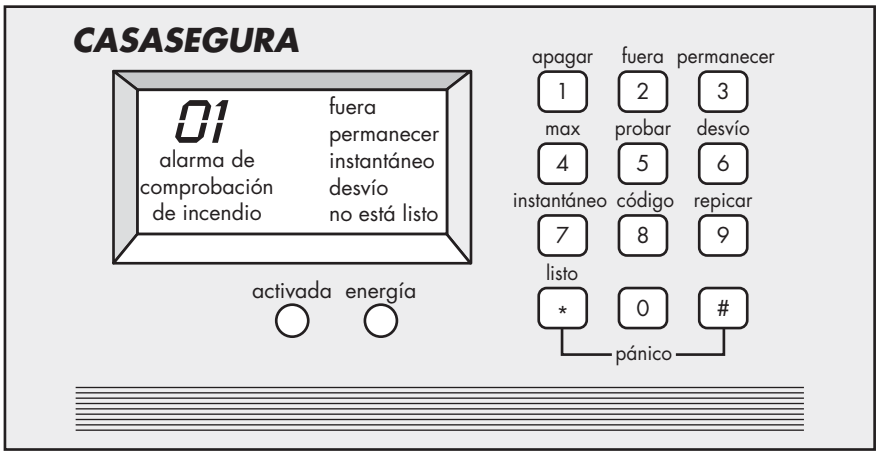
1. El propietario observa el panel de control de *CasaSegura* (véase la figura 5.1) para determinar si el sistema está listo para recibir una entrada. Si el sistema no está listo, se muestra el mensaje *no está listo* en la pantalla de cristal líquido y el propietario debe cerrar físicamente ventanas o puertas de modo que desaparezca dicho mensaje [el mensaje *no está listo* implica que un sensor está abierto; por ejemplo, que una puerta o ventana está abierta].
2. El propietario usa el teclado para introducir una clave de cuatro dígitos. La clave se compara con la que guarda el sistema como válida. Si la clave es incorrecta, el panel de control emitirá un sonido una vez y se reiniciará para recibir una entrada adicional. Si la clave es correcta, el panel de control espera otras acciones.
3. El propietario selecciona y teclea *permanecer* o *fuera* (véase la figura 5.1) para activar el sistema. La entrada *permanecer* activa sólo sensores perimetrales (se desactivan los sensores de detección de movimiento interior). La entrada *fuera* activa todos los sensores.
4. Cuando ocurre una activación, el propietario observa una luz roja de alarma.

---

12 Las preguntas de Jacobson se han ampliado para que den una visión más completa del contenido del caso de uso.

13 Observe que este caso de uso difiere de la situación en la que se accede al sistema a través de internet. En este caso, la interacción es por medio del panel de control y no con la interfaz de usuario gráfica (GUI) que se da cuando se emplea una PC.

FIGURA 5.1
Panel de control de *CasaSegura*



Es frecuente que los casos de uso se escriban de manera informal. Sin embargo, utilice el formato que se presenta aquí para asegurar que se incluyen todos los aspectos clave.

El caso de uso básico presenta una historia de alto nivel que describe la interacción entre el actor y el sistema.

En muchas circunstancias, los casos de uso son más elaborados a fin de que brinden muchos más detalles sobre la interacción. Por ejemplo, Cockburn [Coc01b] sugiere el formato siguiente para hacer descripciones detalladas de casos de uso:

Caso de uso:	*IniciarVigilancia*
Actor principal:	Propietario.
Objetivo en contexto:	Preparar el sistema para que vigile los sensores cuando el propietario salga de la casa o permanezca dentro.
Precondiciones:	El sistema se ha programado para recibir una clave y reconocer distintos sensores.
Disparador:	El propietario decide "preparar" el sistema, por ejemplo, para que encienda las funciones de alarma.

Escenario:

1. Propietario: observa el panel de control
2. Propietario: introduce una clave
3. Propietario: selecciona "permanecer" o "fuera"
4. Propietario: observa una luz roja de alarma que indica que *CasaSegura* ha sido activada.

Excepciones:

1. El panel de control *no está listo*: el propietario verifica todos los sensores para determinar cuáles están abiertos; los cierra.
2. La clave es incorrecta (el panel de control suena una vez): el propietario introduce la clave correcta.
3. La clave no es reconocida: debe contactarse el subsistema de vigilancia y respuesta para reprogramar la clave.
4. Se elige *permanecer*: el panel de control suena dos veces y se enciende un letrero luminoso que dice *permanecer*; se activan los sensores del perímetro.
5. Se selecciona *fuera*: el panel de control suena tres veces y se enciende un letrero luminoso que dice *fuera*; se activan todos los sensores.

Prioridad: Esencial, debe implementarse

Cuándo estará disponible: En el primer incremento

Frecuencia de uso: Muchas veces por día

Canal para el actor: A través de la interfaz del panel de control

Actores secundarios: Técnico de apoyo, sensores

Canales para los actores secundarios:

Técnico de apoyo: línea telefónica

Sensores: interfaces cableadas y frecuencia de radio

Aspectos pendientes:

1. ¿Debe haber una forma de activar el sistema sin usar clave o con una clave abreviada?
2. ¿El panel de control debe mostrar mensajes de texto adicionales?
3. ¿De cuánto tiempo dispone el propietario para introducir la clave a partir del momento en el que se oprime la primera tecla?
4. ¿Hay una forma de desactivar el sistema antes de que se active en realidad?

Los casos de uso para otras interacciones de **propietario** se desarrollarían en una forma similar. Es importante revisar con cuidado cada caso de uso. Si algún elemento de la interacción es ambiguo, es probable que la revisión del caso de uso lo detecte.

CasaSegura

Desarrollo de un diagrama de caso de uso de alto nivel

La escena: Sala de juntas, continúa la reunión para recabar los requerimientos.

Participantes: Jamie Lazar, miembro del equipo de software; Vinod Roman, integrante del equipo de software; Ed Robbins, integrante del equipo de software; Doug Miller, gerente de ingeniería de software; tres miembros de mercadotecnia; un representante de ingeniería del producto; un facilitador.

La conversación:

Facilitador: Hemos pasado un buen tiempo hablando de la función de seguridad del hogar de *CasaSegura*. Durante el receso hice un diagrama de caso de uso para resumir los escenarios importantes que forman parte de esta función. Veámoslo.

(Todos los asistentes observan la figura 5.2.)

Jamie: Estoy aprendiendo la notación UML.[14] Veo que la función de seguridad del hogar está representada por el rectángulo grande con óvalos en su interior, ¿verdad? ¿Y los óvalos representan los casos de uso que hemos escrito?

Facilitador: Sí. Y las figuras pegadas representan a los actores —personas o cosas que interactúan con el sistema según los describe el caso de uso…—; ¡ah! usé el cuadrado para representar un actor que no es persona… en este caso, sensores.

Doug: ¿Es válido eso en UML?

Facilitador: La legalidad no es lo importante. El objetivo es comunicar información. Veo que usar una figura humana para representar un equipo sería erróneo. Así que adapté las cosas un poco. No pienso que genere problemas.

Vinod: Está bien, entonces tenemos narraciones de casos de uso para cada óvalo. ¿Necesitamos desarrollarlas con base en los formatos sobre los que he leído?

Facilitador: Es probable, pero eso puede esperar hasta que hayamos considerado otras funciones de *CasaSegura*.

Persona de mercadotecnia: Esperen, he estado observando este diagrama y de pronto me doy cuenta de que hemos olvidado algo.

Facilitador: ¿De verdad? Dime, ¿qué hemos olvidado?

(La reunión continúa.)

14 En el apéndice 1 se presenta un breve método de aprendizaje de UML para aquellos lectores que no estén familiarizados con dicha notación.

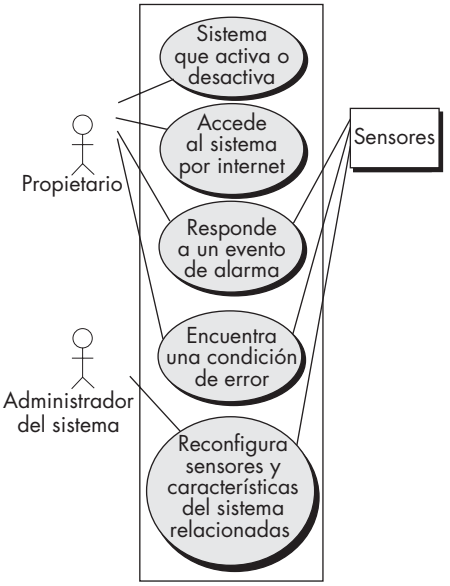


FIGURA 5.2
Diagrama de caso de uso de UML para la función de seguridad del hogar de *CasaSegura*

HERRAMIENTAS DE SOFTWARE

Desarrollo de un caso de uso

Objetivo: Ayudar a desarrollar casos de uso proporcionando formatos y mecanismos automatizados para evaluar la claridad y consistencia.

Mecánica: La mecánica de las herramientas varía. En general, las herramientas para casos de uso dan formatos con espacios en blanco para ser llenados y crear así casos eficaces. La mayor parte de la funcionalidad de los casos de uso está incrustada en un conjunto más amplio de funciones de ingeniería de los requerimientos.

Herramientas representativas[15]

La gran mayoría de herramientas de análisis del modelado basadas en UML dan apoyo tanto de texto como gráfico para el desarrollo y modelado de casos de uso.

Objects by Design
(www.objectsbydesign.com/tools/umltools_byCompany.html) proporciona vínculos exhaustivos con herramientas de este tipo.

5.5 ELABORACIÓN DEL MODELO DE LOS REQUERIMIENTOS[16]

El objetivo del modelo del análisis es describir los dominios de información, función y comportamiento que se requieren para un sistema basado en computadora. El modelo cambia en forma dinámica a medida que se aprende más sobre el sistema por construir, y otros participantes comprenden más lo que en realidad requieren. Por esa razón, el modelo del análisis es una fotografía de los requerimientos en cualquier momento dado. Es de esperar que cambie.

A medida que evoluciona el modelo de requerimientos, ciertos elementos se vuelven relativamente estables, lo que da un fundamento sólido para diseñar las tareas que sigan. Sin embargo, otros elementos del modelo son más volátiles, lo que indica que los participantes todavía no entienden bien los requerimientos para el sistema. En los capítulos 6 y 7 se presentan en

15 Las herramientas mencionadas aquí no son obligatorias, sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

16 En este libro se usan como sinónimos las expresiones *modelar el análisis* y *modelar los requerimientos*. Ambos se refieren a representaciones de los dominios de la información, funcional y de comportamiento que describen los requerimientos del problema.

detalle el modelo del análisis y los métodos que se usan para construirlo. En las secciones siguientes se da un panorama breve.

5.5.1 Elementos del modelo de requerimientos

Hay muchas formas diferentes de concebir los requerimientos para un sistema basado en computadora. Algunos profesionales del software afirman que es mejor seleccionar un modo de representación (por ejemplo, el caso de uso) y aplicarlo hasta excluir a todos los demás. Otros piensan que es más benéfico usar cierto número de modos de representación distintos para ilustrar el modelo de requerimientos. Los modos diferentes de representación fuerzan a considerar los requerimientos desde distintos puntos de vista, enfoque que tiene una probabilidad mayor de detectar omisiones, inconsistencia y ambigüedades.

Los elementos específicos del modelo de requerimientos están determinados por el método de análisis de modelado (véanse los capítulos 6 y 7) que se use. No obstante, la mayoría de modelos tiene en común un conjunto de elementos generales.

Siempre es buena idea involucrar a los participantes. Una de las mejores formas de lograrlo es hacer que cada uno escriba casos de uso que narren el modo en el que se utilizará el software.

Elementos basados en el escenario. El sistema se describe desde el punto de vista del usuario con el empleo de un enfoque basado en el escenario. Por ejemplo, los casos de uso básico (véase la sección 5.4) y sus diagramas correspondientes de casos de uso (véase la figura 5.2) evolucionan hacia otros más elaborados que se basan en formatos. Los elementos del modelo de requerimientos basados en el escenario con frecuencia son la primera parte del modelo en desarrollo. Como tales, sirven como entrada para la creación de otros elementos de modelado. La figura 5.3 ilustra un diagrama de actividades UML[17] para indagar los requerimientos y representarlos con el empleo de casos de uso. Se aprecian tres niveles de elaboración que culminan en una representación basada en el escenario.

Una forma de aislar las clases es buscar sustantivos descriptivos en un caso de usuario expresado con texto. Al menos algunos de ellos serán candidatos cercanos. Sobre esto se habla más en el capítulo 8.

Elementos basados en clases. Cada escenario de uso implica un conjunto de objetos que se manipulan cuando un actor interactúa con el sistema. Estos objetos se clasifican en clases: conjunto de objetos que tienen atributos similares y comportamientos comunes. Por ejemplo, para ilustrar la clase **Sensor** de la función de seguridad de *Casa Segura* (véase la figura 5.4), puede utilizarse un diagrama de clase UML. Observe que el diagrama enlista los atributos de los sensores (por ejemplo, nombre, tipo, etc.) y las operaciones (por ejemplo, *identificar* y *permitir*) que se aplican para modificarlos. Además de los diagramas de clase, otros elementos de modelado del análisis ilustran la manera en la que las clases colaboran una con otra y las relaciones e interacciones entre ellas. Esto se analiza con más detalle en el capítulo 7.

Elementos de comportamiento. El comportamiento de un sistema basado en computadora tiene un efecto profundo en el diseño que se elija y en el enfoque de implementación que se aplique. Por tanto, el modelo de requerimientos debe proveer elementos de modelado que ilustren el comportamiento.

Un estado es un modo de comportamiento observable desde el exterior. Los estímulos externos ocasionan transiciones entre los estados.

El *diagrama de estado* es un método de representación del comportamiento de un sistema que ilustra sus estados y los eventos que ocasionan que el sistema cambie de estado. Un *estado* es cualquier modo de comportamiento observable desde el exterior. Además, el diagrama de estado indica acciones (como la activación de un proceso, por ejemplo) tomadas como consecuencia de un evento en particular.

Para ilustrar el uso de un diagrama de estado, considere el software incrustado dentro del panel de control de *CasaSegura* que es responsable de leer las entradas que hace el usuario. En la figura 5.5 se presenta un diagrama de estado UML simplificado.

Además de las representaciones de comportamiento del sistema como un todo, también es posible modelar clases individuales. Sobre esto se presentan más análisis en el capítulo 7.

---

17 En el apéndice 1 se presenta un instructivo breve sobre UML, para aquellos lectores que no estén familiarizados con dicha notación.

FIGURA 5.3

Diagramas de actividad del UML para indagar los requerimientos

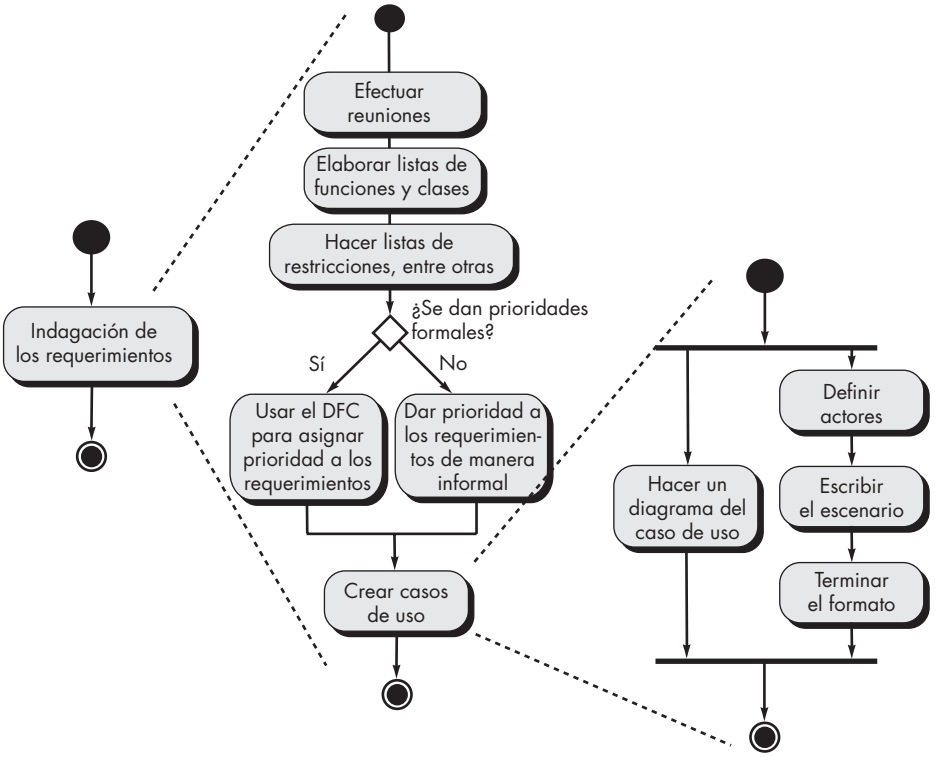


FIGURA 5.4

Diagrama de clase para un sensor

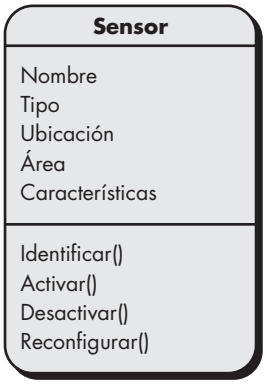
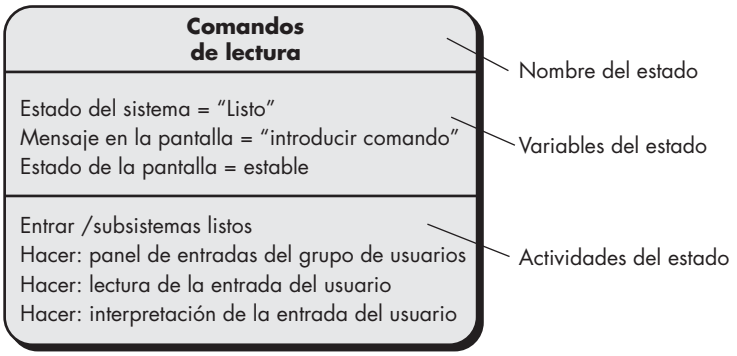


FIGURA 5.5

Notación UML del diagrama de estado



CasaSegura

Modelado preliminar del comportamiento

La escena: Sala de juntas, continúa la reunión de requerimientos.

Participantes: Jamie Lazar, integrante del equipo de software; Vinod Raman, miembro del equipo de software; Ed Robbins, integrante del equipo de software; Doug Miller, gerente de ingeniería de software; tres trabajadores de mercadotecnia; un representante de ingeniería del producto, y un facilitador.

La conversación:

Facilitador: Estamos por terminar de hablar sobre la funcionalidad de seguridad del hogar de *CasaSegura*. Pero antes, quisiera que analizáramos el comportamiento de la función.

Persona de mercadotecnia: No entiendo lo que quiere decir con *comportamiento*.

Ed (sonríe): Es cuando le das un "tiempo fuera" al producto si se porta mal.

Facilitador: No exactamente. Permítanme explicarlo.

(El facilitador explica al equipo encargado de recabar los requerimientos y los fundamentos de modelado del comportamiento.)

Persona de mercadotecnia: Esto parece un poco técnico. No estoy seguro de ser de ayuda aquí.

Facilitador: Seguro que lo serás. ¿Qué comportamiento se observa desde el punto de vista de un usuario?

Persona de mercadotecnia: Mmm... bueno, el sistema estará *vigilando* los sensores. *Leerá comandos* del propietario. *Mostrará* su estado.

Facilitador: ¿Ves?, lo puedes hacer.

Jamie: También estará *interrogando* a la PC para determinar si hay alguna entrada desde ella, por ejemplo, un acceso por internet o información sobre la configuración.

Vinod: Sí, en realidad, *configurar* el sistema es un estado por derecho propio.

Doug: Muchachos, lo hacen bien. Pensemos un poco más... ¿hay alguna forma de hacer un diagrama de todo esto?

Facilitador: Sí la hay, pero la dejaremos para la próxima reunión.

Elementos orientados al flujo. La información se transforma cuando fluye a través de un sistema basado en computadora. El sistema acepta entradas en varias formas, aplica funciones para transformarla y produce salidas en distintos modos. La entrada puede ser una señal de control transmitida por un transductor, una serie de números escritos con el teclado por un operador humano, un paquete de información enviado por un enlace de red o un archivo grande de datos recuperado de un almacenamiento secundario. La transformación quizá incluya una sola comparación lógica, un algoritmo numérico complicado o un enfoque de regla de inferencia para un sistema experto. La salida quizá encienda un diodo emisor de luz o genere un informe de 200 páginas. En efecto, es posible crear un modelo del flujo para cualquier sistema basado en computadora, sin importar su tamaño y complejidad. En el capítulo 7 se hace un análisis más detallado del modelado del flujo.

5.5.2 Patrones de análisis

Cualquiera que haya hecho la ingeniería de los requerimientos en varios proyectos de software ha observado que ciertos problemas son recurrentes en todos ellos dentro de un dominio de aplicación específico.[18] Estos *patrones de análisis* [Fow97] sugieren soluciones (por ejemplo, una clase, función o comportamiento) dentro del dominio de la aplicación que pueden volverse a utilizar cuando se modelen muchas aplicaciones.

Geyer-Schulz y Hahsler [Gey01] sugieren dos beneficios asociados con el uso de patrones de análisis:

> En primer lugar, los patrones de análisis aceleran el desarrollo de los modelos de análisis abstracto que capturan los principales requerimientos del problema concreto, debido a que proveen modelos de análisis reutilizables con ejemplos, así como una descripción de sus ventajas y limitaciones. En se-

18 En ciertos casos, los problemas vuelven a suceder sin importar el dominio de la aplicación. Por ejemplo, son comunes las características y funciones usadas para resolver problemas de la interfaz de usuario sin importar el dominio de la aplicación en consideración.

gundo lugar, los patrones de análisis facilitan la transformación del modelo de análisis en un modelo del diseño, sugiriendo patrones de diseño y soluciones confiables para problemas comunes.

Los patrones de análisis se integran en el modelo del análisis, haciendo referencia al nombre del patrón. También se guardan en un medio de almacenamiento de modo que los ingenieros de requerimientos usen herramientas de búsqueda para encontrarlos y aplicarlos. La información sobre el patrón de análisis (y otros tipos de patrones) se presenta en un formato estándar [Gey01][19] que se estudia con más detalle en el capítulo 12. En el capítulo 7 se dan ejemplos de patrones de análisis y más detalles de este tema.

5.6 Requerimientos de las negociaciones

Cita:

"Un compromiso es el arte de dividir un pastel en forma tal que todos crean que tienen el trozo mayor."

Ludwig Erhard

En un contexto ideal de la ingeniería de los requerimientos, las tareas de concepción, indagación y elaboración determinan los requerimientos del cliente con suficiente detalle como para avanzar hacia las siguientes actividades de la ingeniería de software. Desafortunadamente, esto rara vez ocurre. En realidad, se tiene que entrar en negociaciones con uno o varios participantes. En la mayoría de los casos, se pide a éstos que evalúen la funcionalidad, desempeño y otras características del producto o sistema, en contraste con el costo y el tiempo para entrar al mercado. El objetivo de esta negociación es desarrollar un plan del proyecto que satisfaga las necesidades del participante y que al mismo tiempo refleje las restricciones del mundo real (por ejemplo, tiempo, personas, presupuesto, etc.) que se hayan establecido al equipo del software.

Las mejores negociaciones buscan un resultado "ganar-ganar".[20] Es decir, los participantes ganan porque obtienen el sistema o producto que satisface la mayoría de sus necesidades y usted (como miembro del equipo de software) gana porque trabaja con presupuestos y plazos realistas y asequibles.

WebRef

Un artículo breve sobre la negociación para los requerimientos de software puede descargarse desde la dirección **www.alexander-egyed.com/publications/Software_Requirements_Negotiation-Some_Lessons_Learned.html**

Boehm [Boe98] define un conjunto de actividades de negociación al principio de cada iteración del proceso de software. En lugar de una sola actividad de comunicación con el cliente, se definen las actividades siguientes:

1. Identificación de los participantes clave del sistema o subsistema.
2. Determinación de las "condiciones para ganar" de los participantes.

Información

El arte de la negociación

Aprender a negociar con eficacia le servirá en su vida personal y técnica. Es útil considerar los lineamientos que siguen:

1. *Reconocer que no es una competencia.* Para tener éxito, ambas partes tienen que sentir que han ganado o logrado algo. Las dos tienen que llegar a un compromiso.
2. *Mapear una estrategia.* Decidir qué es lo que le gustaría lograr; qué quiere obtener la otra parte y cómo hacer para que ocurran las dos cosas.
3. *Escuchar activamente.* No trabaje en la formulación de su respuesta mientras la otra parte esté hablando. Escúchela. Es probable que obtenga conocimientos que lo ayuden a negociar mejor su posición.
4. *Centrarse en los intereses de la otra parte.* Si quiere evitar conflictos, no adopte posiciones inamovibles.
5. *No lo tome en forma personal.* Céntrese en el problema que necesita resolverse.
6. *Sea creativo.* Si están empantanados, no tenga miedo de pensar fuera de los moldes.
7. *Esté listo para comprometerse.* Una vez que se llegue a un acuerdo, no titubee; comprométase con él y cúmplalo.

19 En la bibliografía existen varias propuestas de formatos para patrones. Si el lector tiene interés, consulte [Fow97], [Gam95], [Yac03] y [Bus07], entre muchas otras fuentes.

20 Se han escrito decenas de libros acerca de las aptitudes para negociar (por ejemplo [Lew06], [Rai06] y [Fis06]). Es una de las aptitudes más importantes que pueda aprender. Lea alguno.

3. Negociación de las condiciones para ganar de los participantes a fin de reconciliarlas en un conjunto de condiciones ganar-ganar para todos los que intervienen (incluso el equipo de software).

La realización exitosa de estos pasos iniciales lleva a un resultado ganar-ganar, que se convierte en el criterio clave para avanzar hacia las siguientes actividades de la ingeniería de software.

CASASEGURA

El principio de una negociación

La escena: Oficina de Lisa Pérez, después de la primera reunión para recabar los requerimientos.

Participantes: Doug Miller, gerente de ingeniería de software, y Lisa Pérez, gerente de mercadotecnia.

La conversación:

Lisa: Pues escuché que la primera reunión salió realmente bien.

Doug: En realidad, sí. Enviaste buenos representantes... contribuyeron de verdad.

Lisa (sonríe): Sí; en realidad me dijeron que habían entrado y que no había sido una "actividad que les despejara la cabeza".

Doug (ríe): La próxima vez me aseguraré de quitarme la vena tecnológica... Mira, Lisa, creo que tenemos un problema para llegar a toda esa funcionalidad del sistema de seguridad para el hogar en las fechas que propone tu dirección. Sé que aún es temprano, pero hice un poco de planeación sobre las rodillas y...

Lisa (con el ceño fruncido): Lo debemos tener para esa fecha, Doug. ¿De qué funcionalidad hablas?

Doug: Supongo que podemos tener la funcionalidad completa en la fecha establecida, pero tendríamos que retrasar el acceso por internet hasta el segundo incremento.

Lisa: Doug, es el acceso por internet lo que da a *CasaSegura* su "súper" atractivo. Toda nuestra campaña de publicidad va a girar alrededor de eso. Lo tenemos que tener…

Doug: Entiendo la situación, de verdad. El problema es que para dar acceso por internet tendríamos que tener un sitio web por completo seguro y en operación. Esto requiere tiempo y personal. También tenemos que elaborar mucha funcionalidad adicional en la primera entrega… no creo que podamos hacerlo con los recursos que tenemos.

Lisa (todavía frunce el ceño): Ya veo, pero tienes que imaginar una manera de hacerlo. Tiene importancia crítica para las funciones de seguridad del hogar y también para otras… éstas podrían esperar hasta las siguientes entregas… estoy de acuerdo con eso.

Lisa y Doug parecen estar en suspenso, pero todavía deben negociar una solución a este problema. ¿Pueden "ganar" los dos en este caso? Si usted fuera el mediador, ¿qué sugeriría?

5.7 VALIDACIÓN DE LOS REQUERIMIENTOS

A medida que se crea cada elemento del modelo de requerimientos, se estudia para detectar inconsistencias, omisiones y ambigüedades. Los participantes asignan prioridades a los requerimientos representados por el modelo y se agrupan en paquetes de requerimientos que se implementarán como incrementos del software. La revisión del modelo de requerimientos aborda las preguntas siguientes:

? Cuando se revisan los requerimientos, ¿qué preguntas deben plantearse?

- ¿Es coherente cada requerimiento con los objetivos generales del sistema o producto?
- ¿Se han especificado todos los requerimientos en el nivel apropiado de abstracción? Es decir, ¿algunos de ellos tienen un nivel de detalle técnico que resulta inapropiado en esta etapa?
- El requerimiento, ¿es realmente necesario o representa una característica agregada que tal vez no sea esencial para el objetivo del sistema?
- ¿Cada requerimiento está acotado y no es ambiguo?
- ¿Tiene atribución cada requerimiento? Es decir, ¿hay una fuente (por lo general una individual y específica) clara para cada requerimiento?
- ¿Hay requerimientos en conflicto con otros?

- ¿Cada requerimiento es asequible en el ambiente técnico que albergará el sistema o producto?
- Una vez implementado cada requerimiento, ¿puede someterse a prueba?
- El modelo de requerimientos, ¿refleja de manera apropiada la información, la función y el comportamiento del sistema que se va a construir?
- ¿Se ha "particionado" el modelo de requerimientos en forma que exponga información cada vez más detallada sobre el sistema?
- ¿Se ha usado el patrón de requerimientos para simplificar el modelo de éstos? ¿Se han validado todos los patrones de manera apropiada? ¿Son consistentes todos los patrones con los requerimientos del cliente?

Éstas y otras preguntas deben plantearse y responderse para garantizar que el modelo de requerimientos es una reflexión correcta sobre las necesidades del participante y que provee un fundamento sólido para el diseño.

5.8 Resumen

Las tareas de la ingeniería de requerimientos se realizan para establecer un fundamento sólido para el diseño y la construcción. La ingeniería de requerimientos ocurre durante las actividades de comunicación y modelado que se hayan definido para el proceso general del software. Los miembros del equipo de software llevan a cabo siete funciones de ingeniería de requerimientos: concepción, indagación, elaboración, negociación, especificación, validación y administración.

En la concepción del proyecto, los participantes establecen los requerimientos básicos del problema, definen las restricciones generales del proyecto, así como las características y funciones principales que debe presentar el sistema para cumplir sus objetivos. Esta información se mejora y amplía durante la indagación, actividad en la que se recaban los requerimientos y que hace uso de reuniones que lo facilitan, DFC y el desarrollo de escenarios de uso.

La elaboración amplía aún más los requerimientos en un modelo: una colección de elementos basados en escenarios, clases y comportamiento, y orientados al flujo. El modelo hace referencia a patrones de análisis: soluciones para problemas de análisis que se ha observado que son recurrentes en diferentes aplicaciones.

Conforme se identifican los requerimientos y se crea su modelo, el equipo de software y otros participantes negocian la prioridad, la disponibilidad y el costo relativo de cada requerimiento. Además, se valida cada requerimiento y su modelo como un todo comparado con las necesidades del cliente a fin de garantizar que va a construirse el sistema correcto.

Problemas y puntos por evaluar

5.1. ¿Por qué muchos desarrolladores de software no ponen atención suficiente a la ingeniería de requerimientos? ¿Existen algunas circunstancias que puedan ignorarse?

5.2. El lector tiene la responsabilidad de indagar los requerimientos de un cliente que dice estar demasiado ocupado para tener una reunión. ¿Qué debe hacer?

5.3. Analice algunos de los problemas que ocurren cuando los requerimientos deben indagarse para tres o cuatro clientes distintos.

5.4. ¿Por qué se dice que el modelo de requerimientos representa una fotografía instantánea del sistema en el tiempo?

5.5. Suponga que ha convencido al cliente (es usted muy buen vendedor) para que esté de acuerdo con todas las demandas que usted hace como desarrollador. ¿Eso lo convierte en un gran negociador? ¿Por qué?

5.6. Desarrolle al menos tres "preguntas libres de contexto" adicionales que podría plantear a un participante durante la concepción.

5.7. Desarrolle un "kit" para recabar requerimientos. Debe incluir un conjunto de lineamientos a fin de llevar a cabo la reunión para recabar requerimientos y los materiales que pueden emplearse para facilitar la creación de listas y otros objetos que ayuden a definir los requerimientos.

5.8. Su profesor formará grupos de cuatro a seis estudiantes. La mitad de ellos desempeñará el papel del departamento de mercadotecnia y la otra mitad adoptará el del equipo para la ingeniería de software. Su trabajo es definir los requerimientos para la función de seguridad de *CasaSegura* descrita en este capítulo. Efectúe una reunión para recabar los requerimientos con el uso de los lineamientos presentados en este capítulo.

5.9. Desarrolle un caso de uso completo para una de las actividades siguientes:

a) Hacer un retiro de efectivo en un cajero automático.
b) Usar su tarjeta de crédito para pagar una comida en un restaurante.
c) Comprar acciones en la cuenta en línea de una casa de bolsa.
d) Buscar libros (sobre un tema específico) en una librería en línea.
e) La actividad que especifique su profesor.

5.10. ¿Qué representan las "excepciones" en un caso de uso?

5.11. Describa con sus propias palabras lo que es un *patrón de análisis*.

5.12. Con el formato presentado en la sección 5.5.2, sugiera uno o varios patrones de análisis para los siguientes dominios de aplicación:

a) Software de contabilidad.
b) Software de correo electrónico.
c) Navegadores de internet.
d) Software de procesamiento de texto.
e) Software para crear un sitio web.
f) El dominio de aplicación que diga su profesor.

5.13. ¿Qué significa *ganar-ganar* en el contexto de una negociación durante la actividad de ingeniería de los requerimientos?

5.14. ¿Qué piensa que pasa cuando la validación de los requerimientos detecta un error? ¿Quién está involucrado en su corrección?

Lecturas adicionales y fuentes de información

La ingeniería de requerimientos se estudia en muchos libros debido a su importancia crítica para la creación exitosa de cualquier sistema basado en computadoras. Hood *et al.* (*Requirements Management*, Springer, 2007) analizan varios aspectos de la ingeniería de los requerimientos que incluyen tanto la ingeniería de sistemas como la de software. Young (*The Requirements Engineering Handbook*, Artech House Publishers, 2007) presenta un análisis profundo de las tareas de la ingeniería de requerimientos. Wiegers (*More About Software Requirements*, Microsoft Press, 2006) menciona muchas técnicas prácticas para recabar y administrar los requerimientos. Hull *et al.* (*Requirements Engineering*, 2a. ed., Springer-Verlag, 2004), Bray (*An Introduction to Requirements Engineering*, Addison-Wesley, 2002), Arlow (*Requirements Engineering*, Addison-Wesley, 2001), Gilb (*Requirements Engineering*, Addison-Wesley, 2000), Graham (*Requirements Engineering and Rapid Development*, Addison-Wesley, 1999) y Sommerville y Kotonya (*Requirement Engineering: Processes and Techniques*, Wiley, 1998) son sólo algunos de los muchos libros dedicados al tema. Gottesdiener (*Requirements by Collaboration: Workshops for Defining Needs*, Addison-Wesley, 2002) proporciona una guía útil para quienes deben generar un ambiente de colaboración a fin de recabar los requerimientos con los participantes.

Lauesen (*Software Requirements: Styles and Techniques*, Addison-Wesley, 2002) presenta una recopilación exhaustiva de los métodos y notación para el análisis de requerimientos. Weigers (*Software Requirements*, Microsoft Press, 1999) y Leffingwell *et al.* (*Managing Software Requirements: A Use Case Approach*, 2a. ed., Addison-Wesley, 2003) presentan una colección útil de las mejores prácticas respecto de los requerimientos y sugieren lineamientos prácticos para la mayoría de los aspectos del proceso de su ingeniería.

En Withall (*Software Requirement Patterns*, Microsoft Press, 2007) se describe la ingeniería de requerimientos desde un punto de vista basado en los patrones. Ploesch (*Assertions, Scenarios and Prototypes*, Springer-Verlag, 2003) analiza técnicas avanzadas para desarrollar requerimientos de software. Windle y Abreo (*Software Requirements Using the Unified Process*, Prentice-Hall, 2002) estudian la ingeniería de los requerimientos en el contexto del proceso unificado y la notación UML. Alexander y Steven (*Writing Better Requirements*, Addison-Wesley, 2002) presentan un conjunto abreviado de lineamientos para escribir requerimientos claros, representarlos como escenarios y revisar el resultado final.

Es frecuente que el modelado de un caso de uso sea el detonante para crear todos los demás aspectos del modelo de análisis. El tema lo estudian mucho Rosenberg y Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality*, Addison-Wesley, 2005), Alexander y Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*, Wiley, 2004), Leffingwell *et al.* (*Managing Software Requirements: A Use Case Approach*, 2a. ed., Addison-Wesley, 2003) presentan una colección útil de las mejores prácticas sobre los requerimientos. Bittner y Spence (*Use Case Modeling*, Addison-Wesley, 2002), Cockburn [Coc01], Armour y Miller (*Advanced Use Cases Modeling: Software Systems*, Addison-Wesley, 2000) y Kulak *et al.* (*Use Cases: Requirements in Context*, Addison-Wesley, 2000) estudian la obtención de requerimientos con énfasis en el modelado del caso de uso.

En internet hay una variedad amplia de fuentes de información acerca de la ingeniería y análisis de los requerimientos. En el sitio web del libro, **www.mhhe.com/engcs/compsi/pressman/professional/olc/ser.htm**, se halla una lista actualizada de referencias en web que son relevantes para la ingeniería y análisis de los requerimientos.

CAPÍTULO

6 MODELADO DE LOS REQUERIMIENTOS: ESCENARIOS, INFORMACIÓN Y CLASES DE ANÁLISIS

CONCEPTOS CLAVE

análisis del dominio 129
análisis gramatical 143
asociaciones 152
casos de uso 132
clases de análisis. 143
diagrama de actividades . . . 137
diagrama de canal 138
modelado basado en clases 142
modelado basado en escenarios 131
modelado CRC. 148
modelado de datos 139
modelado de requerimientos 130
modelos UML 137
paquetes de análisis 154

En el nivel técnico, la ingeniería de software comienza con una serie de tareas de modelado que conducen a la especificación de los requerimientos y a la representación de un diseño del software que se va a elaborar. El modelo de requerimientos[1] —un conjunto de modelos, en realidad— es la primera representación técnica de un sistema.

En un libro fundamental sobre métodos para modelar los requerimientos, Tom DeMarco [DeM79] describe el proceso de la manera siguiente:

> Al mirar retrospectivamente los problemas y las fallas detectados en la fase de análisis, concluyo que es necesario agregar lo siguiente al conjunto de objetivos de dicha fase. Debe ser muy fácil dar mantenimiento a los productos del análisis. Esto se aplica en particular al Documento de Objetivos [especificación de los requerimientos del software]. Los problemas grandes deben ser enfrentados con el empleo de un método eficaz para dividirlos. La especificación victoriana original resulta caduca. Deben usarse gráficas, siempre que sea posible. Es necesario diferenciar las consideraciones lógicas [esenciales] y las físicas [implementación]... Finalmente, se necesita... algo que ayude a dividir los requerimientos y a documentar dicha partición antes de elaborar la especificación... algunos medios para dar seguimiento a las interfaces y evaluarlas... nuevas herramientas para describir la lógica y la política, algo mejor que un texto narrativo.

UNA MIRADA RÁPIDA

¿Qué es? La palabra escrita es un vehículo maravilloso para la comunicación, pero no necesariamente es la mejor forma de representar los requerimientos de software de computadora. El modelado de los requerimientos utiliza una combinación de texto y diagramas para ilustrarlos en forma que sea relativamente fácil de entender y, más importante, de revisar para corregir, completar y hacer congruente.

¿Quién lo hace? Un ingeniero de software (a veces llamado "analista") construye el modelo con el uso de los requerimientos recabados del cliente.

¿Por qué es importante? Para validar los requerimientos del software se necesita estudiarlos desde varios puntos de vista diferentes. En este capítulo se considerará el modelado de los requerimientos desde tres perspectivas distintas: modelos basados en el escenario, modelos de datos (información) y modelos basados en la clase. Cada una representa a los requerimientos en una "dimensión" diferente, con lo que aumenta la probabilidad de detectar errores, de que afloren las inconsistencias y de que se revelen las omisiones.

¿Cuáles son los pasos? El modelado basado en escenarios es una representación del sistema desde el punto de vista del usuario. El modelado basado en datos recrea el espacio de información e ilustra los objetos de datos que manipulará el software y las relaciones entre ellos. El modelado orientado a clases define objetos, atributos y relaciones. Una vez que se crean los modelos preliminares, se mejoran y analizan para evaluar si están claros y completos, y si son consistentes. En el capítulo 7 se amplían con representaciones adicionales las dimensiones del modelado descritas aquí, lo que da un punto de vista más sólido de los requerimientos.

¿Cuál es el producto final? Para construir el modelo de requerimientos, se escoge una amplia variedad de representaciones basadas en texto y en diagramas. Cada una de dichas representaciones da una perspectiva de uno o más de los elementos del modelo.

¿Cómo me aseguro de que lo hice bien? Los productos del trabajo para modelar los requerimientos deben revisarse para saber si son correctos, completos y consistentes. Deben reflejar las necesidades de todos los participantes y establecer el fundamento desde el que se realizará el diseño.

---

1 En ediciones anteriores de este libro, se usó el término *modelo de análisis*, en lugar de *modelo de requerimientos*. En esta edición, el autor decidió usar ambas expresiones para designar la actividad que define distintos aspectos del problema por resolver. *Análisis* es lo que ocurre cuando se obtienen los *requerimientos*.

Aunque DeMarco escribió hace más de un cuarto de siglo acerca de los atributos del modelado del análisis, sus comentarios aún son aplicables a los métodos y notación modernos del modelado de los requerimientos.

6.1 Análisis de los requerimientos

El análisis de los requerimientos da como resultado la especificación de las características operativas del software, indica la interfaz de éste y otros elementos del sistema, y establece las restricciones que limitan al software. El análisis de los requerimientos permite al profesional (sin importar si se llama *ingeniero de software*, *analista* o *modelista*) construir sobre los requerimientos básicos establecidos durante las tareas de concepción, indagación y negociación, que son parte de la ingeniería de los requerimientos (véase el capítulo 5).

Cita:

"Cualquier 'vista' de los requerimientos es insuficiente para entender o describir el comportamiento deseado de un sistema complejo."

Alan M. Davis

La acción de modelar los requerimientos da como resultado uno o más de los siguientes tipos de modelo:

- *Modelos basados en el escenario* de los requerimientos desde el punto de vista de distintos "actores" del sistema.
- *Modelos de datos*, que ilustran el dominio de información del problema.
- *Modelos orientados a clases*, que representan clases orientadas a objetos (atributos y operaciones) y la manera en la que las clases colaboran para cumplir con los requerimientos del sistema.
- *Modelos orientados al flujo*, que representan los elementos funcionales del sistema y la manera como transforman los datos a medida que se avanza a través del sistema.
- *Modelos de comportamiento*, que ilustran el modo en el que se comparte el software como consecuencia de "eventos" externos.

PUNTO CLAVE

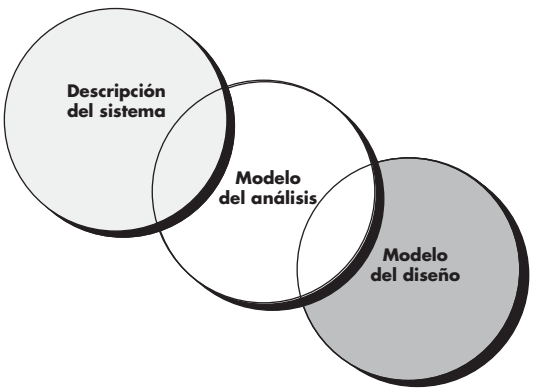
El modelo de análisis y la especificación de requerimientos proporcionan un medio para evaluar la calidad una vez construido el software.

Estos modelos dan al diseñador del software la información que se traduce en diseños de arquitectura, interfaz y componentes. Por último, el modelo de requerimientos (y la especificación de requerimientos de software) brinda al desarrollador y al cliente los medios para evaluar la calidad una vez construido el software.

Este capítulo se centra en el *modelado basado en escenarios*, técnica que cada vez es más popular entre la comunidad de la ingeniería de software; el *modelado basado en datos*, más especializado, apropiado en particular cuando debe crearse una aplicación o bien manipular un espacio complejo de información; y el *modelado orientado a clases*, representación de las clases orientada a objetos y a las colaboraciones resultantes que permiten que funcione el sistema. En

FIGURA 6.1

El modelo de requerimientos como puente entre la descripción del sistema y el modelo del diseño



el capítulo 7 se analizan los modelos orientados al flujo, al comportamiento, basados en el patrón y en *webapps*.

6.1.1 Objetivos y filosofía general

Cita:

"Los requerimientos no son arquitectura. No son diseño ni la interfaz de usuario. Los requerimientos son las necesidades."

Andrew Hunt y David Thomas

Durante el modelado de los requerimientos, la atención se centra en *qué*, no en *cómo*. ¿Qué interacción del usuario ocurre en una circunstancia particular?, ¿qué objetos manipula el sistema?, ¿qué funciones debe realizar el sistema?, ¿qué comportamientos tiene el sistema?, ¿qué interfaces se definen? y ¿qué restricciones son aplicables?[2]

En los capítulos anteriores se dijo que en esta etapa tal vez no fuera posible tener la especificación completa de los requerimientos. El cliente quizá no esté seguro de qué es lo que requiere con precisión para ciertos aspectos del sistema. Puede ser que el desarrollador esté inseguro de que algún enfoque específico cumpla de manera apropiada la función y el desempeño. Estas realidades hablan a favor de un enfoque iterativo para el análisis y el modelado de los requerimientos. El analista debe modelar lo que se sabe y usar el modelo como base para el diseño del incremento del software.[3]

PUNTO CLAVE

El modelo de análisis debe describir lo que quiere el cliente, establecer una base para el diseño y un objetivo para la validación.

El modelo de requerimientos debe lograr tres objetivos principales: 1) describir lo que requiere el cliente, 2) establecer una base para la creación de un diseño de software y 3) definir un conjunto de requerimientos que puedan validarse una vez construido el software. El modelo de análisis es un puente entre la descripción en el nivel del sistema que se centra en éste en lo general o en la funcionalidad del negocio que se logra con la aplicación de software, hardware, datos, personas y otros elementos del sistema y un diseño de software (véanse los capítulos 8 a 13) que describa la arquitectura de la aplicación del software, la interfaz del usuario y la estructura en el nivel del componente. Esta relación se ilustra en la figura 6.1.

Es importante observar que todos los elementos del modelo de requerimientos pueden rastrearse directamente hasta las partes del modelo del diseño. No siempre es posible la división clara entre las tareas del análisis y las del diseño en estas dos importantes actividades del modelado. Invariablemente, ocurre algo de diseño como parte del análisis y algo de análisis se lleva a cabo durante el diseño.

6.1.2 Reglas prácticas del análisis

Arlow y Neustadt [Arl02] sugieren cierto número de reglas prácticas útiles que deben seguirse cuando se crea el modelo del análisis:

? ¿Hay lineamientos básicos que nos ayuden a hacer el trabajo de análisis de los requerimientos?

- *El modelo debe centrarse en los requerimientos que sean visibles dentro del problema o dentro del dominio del negocio. El nivel de abstracción debe ser relativamente elevado.* "No se empantane en los detalles" [Arl02] que traten de explicar cómo funciona el sistema.
- *Cada elemento del modelo de requerimientos debe agregarse al entendimiento general de los requerimientos del software y dar una visión del dominio de la información, de la función y del comportamiento del sistema*.
- *Hay que retrasar las consideraciones de la infraestructura y otros modelos no funcionales hasta llegar a la etapa del diseño.* Es decir, quizá se requiera una base de datos, pero las clases necesarias para implementarla, las funciones requeridas para acceder a ella y el comportamiento que tendrá cuando se use sólo deben considerarse después de que se haya terminado el análisis del dominio del problema.

---

2 Debe notarse que, a medida que los clientes tienen más conocimientos tecnológicos, hay una tendencia hacia la especificación del *cómo* tanto como del *qué*. Sin embargo, la atención debe centrarse en el *qué*.

3 En un esfuerzo por entender mejor los requerimientos para el sistema, el equipo del software tiene la alternativa de escoger la creación de un prototipo (véase el capítulo 2).

Cita:

"Los problemas que es benéfico atacar demuestran su beneficio con un contragolpe."

Piet Hein

- *Debe minimizarse el acoplamiento a través del sistema.* Es importante representar las relaciones entre las clases y funciones. Sin embargo, si el nivel de "interconectividad" es extremadamente alto, deben hacerse esfuerzos para reducirlo.
- *Es seguro que el modelo de requerimientos agrega valor para todos los participantes.* Cada actor tiene su propio uso para el modelo. Por ejemplo, los participantes de negocios deben usar el modelo para validar los requerimientos; los diseñadores deben usarlo como pase para el diseño; el personal de aseguramiento de la calidad lo debe emplear como ayuda para planear las pruebas de aceptación.
- *Mantener el modelo tan sencillo como se pueda.* No genere diagramas adicionales si no agregan nueva información. No utilice notación compleja si basta una sencilla lista.

6.1.3 Análisis del dominio

WebRef

En la dirección **www.iturls.com/English/SoftwareEngineering/SE_mod5.asp**, existen muchos recursos útiles para el análisis del dominio.

Al estudiar la ingeniería de requerimientos (en el capítulo 5), se dijo que es frecuente que haya patrones de análisis que se repiten en muchas aplicaciones dentro de un dominio de negocio específico. Si éstos se definen y clasifican en forma tal que puedan reconocerse y aplicarse para resolver problemas comunes, la creación del modelo del análisis es más expedita. Más importante aún es que la probabilidad de aplicar patrones de diseño y componentes de software ejecutable se incrementa mucho. Esto mejora el tiempo para llegar al mercado y reduce los costos de desarrollo.

Pero, ¿cómo se reconocen por primera vez los patrones de análisis y clases? ¿Quién los define, clasifica y prepara para usarlos en los proyectos posteriores? La respuesta a estas preguntas está en el *análisis del dominio*. Firesmith [Fir93] lo describe del siguiente modo:

> El análisis del dominio del software es la identificación, análisis y especificación de los requerimientos comunes, a partir de un dominio de aplicación específica, normalmente para usarlo varias veces en múltiples proyectos dentro del dominio de la aplicación [...] [El análisis del dominio orientado a objetos es] la identificación, análisis y especificación de capacidades comunes y reutilizables dentro de un dominio de aplicación específica en términos de objetos, clases, subensambles y estructuras comunes.

PUNTO CLAVE

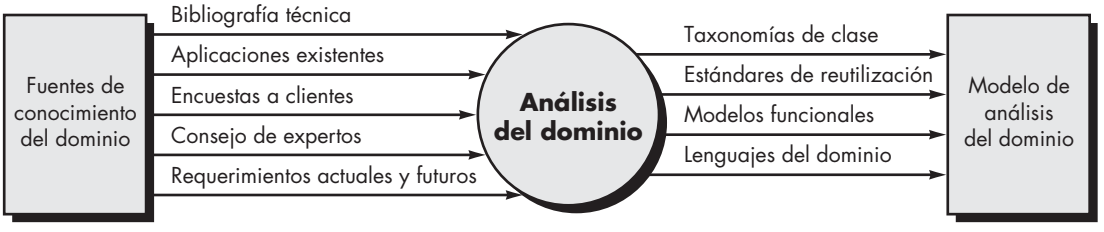
El análisis del dominio no busca en una aplicación específica, sino en el dominio en el que reside la aplicación. El objetivo es identificar elementos comunes para la solución de problemas, que sean útiles en todas las aplicaciones dentro del dominio.

El "dominio de aplicación específica" se extiende desde el control electrónico de aviones hasta la banca, de los juegos de video en multimedios al software incrustado en equipos médicos. La meta del análisis del dominio es clara: encontrar o crear aquellas clases o patrones de análisis que sean aplicables en lo general, de modo que puedan volverse a usar.[4]

Con el empleo de la terminología que se introdujo antes en este libro, el análisis del dominio puede considerarse como una actividad sombrilla para el proceso del software. Esto significa que el análisis del dominio es una actividad de la ingeniería de software que no está conectada

FIGURA 6.2

Entradas y salidas para el análisis del dominio



4 Un punto de vista complementario del análisis del dominio "involucra el modelado de éste, de manera que los ingenieros del software y otros participantes aprendan más al respecto [...] no todas las clases de dominio necesariamente dan como resultado el desarrollo de clases reutilizables [...]" [Let03a].

CasaSegura

Análisis del dominio

La escena: Oficina de Doug Miller, después de una reunión con personal de mercadotecnia.

Participantes: Doug Miller, gerente de ingeniería de software, y Vinod Raman, miembro del equipo de ingeniería de software.

La conversación:

Doug: Te necesito para un proyecto especial, Vinod. Voy a retirarte de las reuniones para recabar los requerimientos.

Vinod (con el ceño fruncido): Muy mal. Ese formato en verdad funciona… Estaba sacando algo de ahí. ¿Qué pasa?

Doug: Jamie y Ed te cubrirán. De cualquier manera, el departamento de mercadotecnia insiste en que en la primera entrega de *CasaSegura* dispongamos de la capacidad de acceso por internet junto con la función de seguridad para el hogar. Estamos bajo fuego en esto… sin tiempo ni personal suficiente, así que tenemos que resolver ambos problemas a la vez: la interfaz de PC y la interfaz de web.

Vinod (confundido): No sabía que el plan era entregar… ni siquiera hemos terminado de recabar los requerimientos.

Doug (con una sonrisa tenue): Lo sé, pero los plazos son tan breves que decidí comenzar ya la estrategia con mercadotecnia… de cualquier modo, revisaremos cualquier plan tentativo una vez que tengamos la información de todas las juntas que se efectuarán para recabar los requerimientos.

Vinod: Está bien, ¿entonces? ¿Qué quieres que haga?

Doug: ¿Sabes qué es el "análisis del dominio"?

Vinod: Algo sé. Buscas patrones similares en aplicaciones que hagan lo mismo que la que estés elaborando. Entonces, si es posible, calcas los patrones y los reutilizas en tu trabajo.

Doug: No estoy seguro de que la palabra sea *calcar*, pero básicamente tienes razón. Lo que me gustaría que hicieras es que comenzaras a buscar interfaces de usuario ya existentes para sistemas que controlen algo como *CasaSegura*. Quiero que propongas un conjunto de patrones y clases de análisis que sean comunes tanto a la interfaz basada en PC que estará en el hogar como a la basada en un navegador al que se accederá por internet.

Vinod: Ahorraríamos tiempo si las hiciéramos iguales… ¿por qué no las hacemos así?

Doug: Ah… es grato tener gente que piense como lo haces tú. Ése es el meollo del asunto: ahorraremos tiempo y esfuerzo si las dos interfaces son casi idénticas; las implementamos con el mismo código y acabamos con la insistencia de mercadotecnia.

Vinod: ¿Entonces, qué quieres?, ¿clases, patrones de análisis, patrones de diseño?

Doug: Todo eso. Nada formal en este momento. Sólo quiero que comencemos despacio con nuestros trabajos de análisis interno y de diseño.

Vinod: Iré a nuestra biblioteca de clases y veré qué tenemos. También usaré un formato de patrones que vi en un libro que leí hace unos meses.

Doug: Bien. Manos a la obra.

con ningún proyecto de software. En cierta forma, el papel del analista del dominio es similar al de un maestro herrero en un ambiente de manufactura pesada. El trabajo del herrero es diseñar y fabricar herramientas que utilicen muchas personas que hacen trabajos similares pero no necesariamente iguales. El papel del analista de dominio[5] es descubrir y definir patrones de análisis, clases de análisis e información relacionada que pueda ser utilizada por mucha gente que trabaje en aplicaciones similares, pero que no son necesariamente las mismas.

Cita:

"… el análisis es frustrante, está lleno de relaciones interpersonales complejas, indefinidas y difíciles. En una palabra, es fascinante. Una vez atrapado, los antiguos y fáciles placeres de la construcción de sistemas nunca más volverán a satisfacerte."

Tom DeMarco

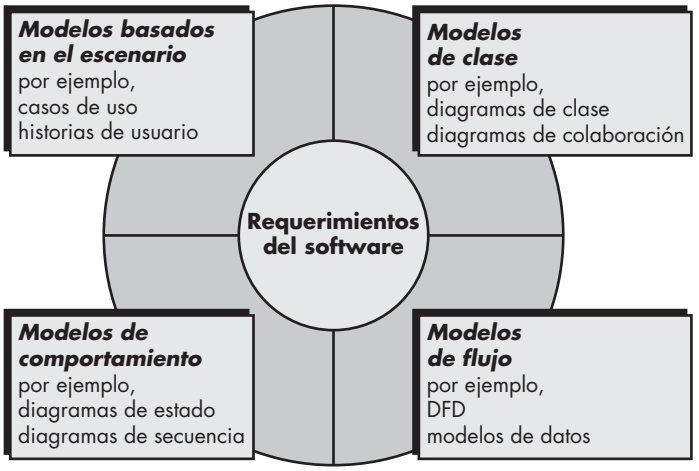
La figura 6.2 [Ara89] ilustra entradas y salidas clave para el proceso de análisis del dominio. Las fuentes de conocimiento del dominio se mapean con el fin de identificar los objetos que pueden reutilizarse a través del dominio.

6.1.4 Enfoques del modelado de requerimientos

Un enfoque del modelado de requerimientos, llamado *análisis estructurado*, considera que los datos y los procesos que los transforman son entidades separadas. Los objetos de datos se modelan de modo que se definan sus atributos y relaciones. Los procesos que manipulan a los objetos de datos se modelan en forma que se muestre cómo transforman a los datos a medida que los objetos que se corresponden con ellos fluyen por el sistema.

5 No suponga que el ingeniero de software no necesita entender el dominio de la aplicación tan sólo porque hay un analista del dominio trabajando. Todo miembro del equipo del software debe entender algo del dominio en el que se va a colocar el software.

FIGURA 6.3
Elementos del modelo de análisis



¿Cuáles son los diferentes puntos de vista que se usan para describir el modelo de requerimientos?

Un segundo enfoque del modelado del análisis, llamado *análisis orientado a objetos*, se centra en la definición de las clases y en la manera en la que colaboran uno con el otro para cumplir los requerimientos. El UML y el proceso unificado (véase el capítulo 2) están orientados a objetos, sobre todo.

Aunque el modelo de requerimientos propuesto en este libro combina características de ambos enfoques, los equipos de software escogen con frecuencia uno y excluyen todas las representaciones del otro. La pregunta no es cuál es mejor, sino qué combinación de representaciones proporcionará a los participantes el mejor modelo de requerimientos del software y el puente más eficaz para el diseño del mismo.

Cita:

"¿Por qué construimos modelos? ¿Por qué no construir sólo el sistema? La respuesta es que los construimos para que resalten o enfaticen ciertas características críticas de un sistema, al tiempo que ignoran otros aspectos del mismo."

Ed Yourdon

Cada elemento del modelo de requerimientos (véase la figura 6.3) presenta el problema desde diferentes puntos de vista. Los elementos basados en el escenario ilustran cómo interactúa el usuario con el sistema y la secuencia específica de actividades que ocurren cuando se utiliza el software. Los elementos basados en la clase modelan los objetos que el sistema manipulará, las operaciones que se aplicarán a ellos para realizar dicha manipulación, las relaciones (algunas jerárquicas) entre los objetos y las colaboraciones que ocurrirán entre las clases que se definan. Los elementos del comportamiento ilustran la forma en la que los eventos externos cambian el estado del sistema o las clases que residen dentro de éste. Por último, los elementos orientados al flujo representan al sistema como una transformación de la información e ilustran la forma en la que se transforman los objetos de datos cuando fluyen a través de las distintas funciones del sistema.

El modelado del análisis lleva a la obtención de cada uno de estos elementos de modelado. Sin embargo, el contenido específico de cada elemento (por ejemplo, los diagramas que se emplean para construir el elemento y el modelo) tal vez difiera de un proyecto a otro. Como se ha dicho varias veces en este libro, el equipo del software debe trabajar para mantenerlo sencillo. Sólo deben usarse elementos de modelado que agreguen valor al modelo.

6.2 MODELADO BASADO EN ESCENARIOS

Aunque el éxito de un sistema o producto basado en computadora se mide de muchas maneras, la satisfacción del usuario ocupa el primer lugar de la lista. Si se entiende cómo desean interactuar los usuarios finales (y otros actores) con un sistema, el equipo del software estará mejor preparado para caracterizar adecuadamente los requerimientos y hacer análisis significativos y

modelos del diseño. Entonces, el modelado de los requerimientos con UML[6] comienza con la creación de escenarios en forma de casos de uso, diagramas de actividades y diagramas tipo carril de natación.

6.2.1 Creación de un caso preliminar de uso

Cita:

"[Los casos de uso] simplemente son una ayuda para definir lo que existe fuera del sistema (actores) y lo que debe realizar el sistema (casos de uso)."

Ivar Jacobson

Alistair Cockburn caracteriza un caso de uso como un "contrato para el comportamiento" [Coc01b]. Como se dijo en el capítulo 5, el "contrato" define la forma en la que un actor[7] utiliza un sistema basado en computadora para alcanzar algún objetivo. En esencia, un caso de uso capta las interacciones que ocurren entre los productores y consumidores de la información y el sistema en sí. En esta sección se estudiará la forma en la que se desarrollan los casos de uso como parte de los requerimientos de la actividad de modelado.[8]

En el capítulo 5 se dijo que un caso de uso describe en lenguaje claro un escenario específico desde el punto de vista de un actor definido. Pero, ¿cómo se sabe sobre qué escribir, cuánto escribir sobre ello, cuán detallada hacer la descripción y cómo organizarla? Son preguntas que deben responderse si los casos de uso han de tener algún valor como herramienta para modelar los requerimientos.

CONSEJO

En ciertas situaciones, los casos de uso se convierten en el mecanismo dominante de la ingeniería de requerimientos. Sin embargo, esto no significa que deban descartarse otros métodos de modelado cuando resulten apropiados.

¿Sobre qué escribir? Las dos primeras tareas de la ingeniería de requerimientos —concepción e indagación— dan la información que se necesita para comenzar a escribir casos de uso. Las reuniones para recabar los requerimientos, el DEC, y otros mecanismos para obtenerlos se utilizan para identificar a los participantes, definir el alcance del problema, especificar los objetivos operativos generales, establecer prioridades, delinear todos los requerimientos funcionales conocidos y describir las cosas (objetos) que serán manipuladas por el sistema.

Para comenzar a desarrollar un conjunto de casos de uso, se enlistan las funciones o actividades realizadas por un actor específico. Éstas se obtienen de una lista de las funciones requeridas del sistema, por medio de conversaciones con los participantes o con la evaluación de los diagramas de actividades (véase la sección 6.3.1) desarrollados como parte del modelado de los requerimientos.

CASASEGURA

Desarrollo de otro escenario preliminar de uso

La escena: Sala de juntas, durante la segunda reunión para recabar los requerimientos.

Participantes: Jamie Lazar, miembro del equipo del software; Ed Robbins, integrante del equipo del software; Doug Miller, gerente de ingeniería de software; tres miembros de mercadotecnia; un representante de ingeniería del producto, y un facilitador.

La conversación:

Facilitador: Es hora de que hablemos sobre la función de vigilancia de *CasaSegura*. Vamos a desarrollar un escenario de usuario que accede a la función de vigilancia.

Jamie: ¿Quién juega el papel del actor aquí?

Facilitador: Creo que Meredith (persona de mercadotecnia) ha estado trabajando en dicha funcionalidad. ¿Por qué no adoptas tú ese papel?

Meredith: Quieres que lo hagamos de la misma forma que la vez pasada, ¿verdad?

Facilitador: Sí... en cierto modo.

Meredith: Bueno, es obvio que la razón de la vigilancia es permitir que el propietario de la casa la revise cuando se encuentre fuera, así como poder grabar y reproducir el video que se grabe... esa clase de cosas.

Ed: ¿Usaremos compresión para guardar el video?

---

6 En todo el libro se usará UML como notación para elaborar modelos. En el apéndice 1 se ofrece un método breve de enseñanza para aquellos lectores que no estén familiarizados con lo más básico de dicha notación.

7 Un actor no es una persona específica sino el rol que desempeña ésta (o un dispositivo) en un contexto específico. Un actor "llama al sistema para que entregue uno de sus servicios" [Coc01b].

8 Los casos de uso son una parte del modelado del análisis de importancia especial para las interfaces. El análisis de la interfaz se estudia en detalle en el capítulo 11.

Facilitador: Buena pregunta, Ed, pero por ahora pospondremos los aspectos de la implementación. ¿Meredith?

Meredith: Bien, básicamente hay dos partes en la función de vigilancia... la primera configura el sistema, incluso un plano de la planta —tiene que haber herramientas que ayuden al propietario a hacer esto—, y la segunda parte es la función real de vigilancia. Como el plano es parte de la actividad de configuración, me centraré en la función de vigilancia.

Facilitador (sonríe): Me quitaste las palabras de la boca.

Meredith: Mmm... quiero tener acceso a la función de vigilancia, ya sea por PC o por internet. Tengo la sensación de que el acceso por internet se usaría con más frecuencia. De cualquier manera, quisiera poder mostrar vistas de la cámara en una PC y controlar el ángulo y acercamiento de una cámara en particular. Especificaría la cámara seleccionándola en el plano de la casa. También quiero poder bloquear el acceso a una o más cámaras con una clave determinada. Además, desearía tener la opción de ver pequeñas ventanas con vistas de todas las cámaras y luego escoger una que desee agrandar.

Jamie: Ésas se llaman vistas reducidas.

Meredith: Bien, entonces quiero vistas reducidas de todas las cámaras. También quisiera que la interfaz de la función de vigilancia tuviera el mismo aspecto y sensación que todas las demás del sistema *CasaSegura*. Quiero que sea intuitiva, lo que significa que no tenga que leer un manual para usarla.

Facilitador: Buen trabajo. Ahora, veamos esta función con un poco más de detalle…

La función (subsistema) de vigilancia de *CasaSegura* estudiada en el recuadro identifica las funciones siguientes (lista abreviada) que va a realizar el actor **propietario**:

- Seleccionar cámara para ver.
- Pedir vistas reducidas de todas las cámaras.
- Mostrar vistas de las cámaras en una ventana de PC.
- Controlar el ángulo y acercamiento de una cámara específica.
- Grabar la salida de cada cámara en forma selectiva.
- Reproducir la salida de una cámara.
- Acceder por internet a la vigilancia con cámaras.

A medida que avanzan las conversaciones con el participante (quien juega el papel de propietario), el equipo que recaba los requerimientos desarrolla casos de uso para cada una de las funciones estudiadas. En general, los casos de uso se escriben primero en forma de narración informal. Si se requiere más formalidad, se reescribe el mismo caso con el empleo de un formato estructurado, similar al propuesto en el capítulo y que se reproduce en un recuadro más adelante, en esta sección.

Para ilustrar esto, considere la función *acceder a la vigilancia con cámaras por internet-mostrar vistas de cámaras* **(AVC-MVC)**. El participante que tenga el papel del actor llamado **propietario** escribiría una narración como la siguiente:

Caso de uso: acceder a la vigilancia con cámaras por internet, mostrar vistas de cámaras (AVC-MVC)

Actor: propietario

Si estoy en una localidad alejada, puedo usar cualquier PC con un software de navegación apropiado para entrar al sitio web de *Productos CasaSegura*. Introduzco mi identificación de usuario y dos niveles de claves; una vez validadas, tengo acceso a toda la funcionalidad de mi sistema instalado. Para acceder a la vista de una cámara específica, selecciono "vigilancia" de los botones mostrados para las funciones principales. Luego selecciono "escoger una cámara" y aparece el plano de la casa. Después elijo la cámara que me interesa. Alternativamente, puedo ver la vista de todas las cámaras simultáneamente si selecciono "todas las cámaras". Una vez que escojo una, selecciono "vista" y en la ventana que cubre la cámara aparece una vista con velocidad de un cuadro por segundo. Si quiero cambiar entre las cámaras, selecciono "escoger una cámara" y desaparece la vista original y de nuevo se muestra el plano de la casa. Después, selecciono la cámara que me interesa. Aparece una nueva ventana de vistas.

Una variación de la narrativa del caso de uso presenta la interacción como una secuencia ordenada de acciones del usuario. Cada acción está representada como enunciado declarativo. Al visitar la función **ACS-DCV**, se escribiría lo siguiente:

Caso de uso: acceder a la vigilancia con cámaras por internet, mostrar vistas de cámaras (AVC-MVC)

Actor: propietario

1. El propietario accede al sitio web *Productos CasaSegura*.
2. El propietario introduce su identificación de usuario.
3. El propietario escribe dos claves (cada una de al menos ocho caracteres de longitud).
4. El sistema muestra los botones de todas las funciones principales.
5. El propietario selecciona "vigilancia" de los botones de las funciones principales.
6. El propietario elige "seleccionar una cámara".
7. El sistema presenta el plano de la casa.
8. El propietario escoge el ícono de una cámara en el plano de la casa.
9. El propietario selecciona el botón "vista".
10. El sistema presenta la ventana de vista identificada con la elección de la cámara.
11. El sistema muestra un video dentro de la ventana a velocidad de un cuadro por segundo.

Cita:

"Los casos de uso se emplean en muchos procesos [de software]. Nuestro favorito es el que es iterativo y guiado por el riesgo."

Gerl Schneider y Jason Winters

Es importante observar que esta presentación en secuencia no considera interacciones alternativas (la narración fluye con más libertad y representa varias alternativas). Los casos de este tipo en ocasiones se denominan *escenarios primarios* [Sch98a].

6.2.2 Mejora de un caso de uso preliminar

Para entender por completo la función que describe un caso de uso, es esencial describir interacciones alternativas. Después se evalúa cada paso en el escenario primario, planteando las preguntas siguientes [Sch98a]:

? Cuando desarrollo un caso de uso, ¿cómo examino los cursos alternativos de acción?

- *¿El actor puede emprender otra acción en este punto?*
- *¿Es posible que el actor encuentre alguna condición de error en este punto?* Si así fuera, ¿cuál podría ser?
- *En este punto, ¿es posible que el actor encuentre otro comportamiento (por ejemplo, alguno que sea invocado por cierto evento fuera del control del actor)?* En ese caso, ¿cuál sería?

Las respuestas a estas preguntas dan como resultado la creación de un conjunto de *escenarios secundarios* que forman parte del caso de uso original, pero que representan comportamientos alternativos. Por ejemplo, considere los pasos 6 y 7 del escenario primario ya descrito:

6. El propietario elige "seleccionar una cámara".
7. El sistema presenta el plano de la casa.

¿El actor puede emprender otra acción en este punto? La respuesta es "sí". Al analizar la narración de flujo libre, el actor puede escoger mirar vistas de todas las cámaras simultáneamente. Entonces, un escenario secundario sería "observar vistas instantáneas de todas las cámaras".

¿Es posible que el actor encuentre alguna condición de error en este punto? Cualquier número de condiciones de error puede ocurrir cuando opera un sistema basado en computadora. En este contexto, sólo se consideran las condiciones que sean probables como resultado directo de la acción descrita en los pasos 6 o 7. De nuevo, la respuesta es "sí". Tal vez nunca se haya configurado un plano con íconos de cámara. Entonces, elegir "seleccionar una cámara" da como

resultado una condición de error: "No hay plano configurado para esta casa."[9] Esta condición de error se convierte en un escenario secundario.

En este punto, ¿es posible que el actor encuentre otro comportamiento (por ejemplo, alguno que sea invocado por cierto evento fuera del control del actor)? Otra vez, la respuesta es "sí". A medida que ocurran los pasos 6 y 7, el sistema puede hallar una condición de alarma. Esto dará como resultado que el sistema desplegará una notificación especial de alarma (tipo, ubicación, acción del sistema) y proporcionará al actor varias opciones relevantes según la naturaleza de la alarma. Como este escenario secundario puede ocurrir en cualquier momento para prácticamente todas las interacciones, no se vuelve parte del caso de uso **AVC-MVC**. En vez de ello, se desarrollará un caso de uso diferente —**Condición de alarma encontrada**— al que se hará referencia desde otros casos según se requiera.

Cada una de las situaciones descritas en los párrafos precedentes se caracteriza como una excepción al caso de uso. Una *excepción* describe una situación (ya sea condición de falla o alternativa elegida por el actor) que ocasiona que el sistema presente un comportamiento algo distinto.

Cockburn [Coc01b] recomienda el uso de una sesión de "lluvia de ideas" para obtener un conjunto razonablemente complejo de excepciones para cada caso de uso. Además de las tres preguntas generales ya sugeridas en esta sección, también deben explorarse los siguientes aspectos:

- *¿Existen casos en los que ocurra alguna "función de validación" durante este caso de uso?* Esto implica que la función de validación es invocada y podría ocurrir una potencial condición de error.
- *¿Hay casos en los que una función (o actor) de soporte falle en responder de manera apropiada?* Por ejemplo, una acción de usuario espera una respuesta pero la función que ha de responder se cae.
- *¿El mal desempeño del sistema da como resultado acciones inesperadas o impropias?* Por ejemplo, una interfaz con base en web responde con demasiada lentitud, lo que da como resultado que un usuario haga selecciones múltiples en un botón de procesamiento. Estas selecciones se forman de modo equivocado y, en última instancia, generan un error.

La lista de extensiones desarrollada como consecuencia de preguntar y responder estas preguntas debe "racionalizarse" [Coc01b] con el uso de los siguientes criterios: una excepción debe describirse dentro del caso de uso si el software la puede detectar y debe manejarla una vez detectada. En ciertos casos, una excepción precipitará el desarrollo de otro caso de uso (el de manejar la condición descrita).

6.2.3 Escritura de un caso de uso formal

En ocasiones, para modelar los requerimientos es suficiente con los casos de uso informales presentados en la sección 6.2.1. Sin embargo, cuando un caso de uso involucra una actividad crítica o cuando describe un conjunto complejo de etapas con un número significativo de excepciones, es deseable un enfoque más formal.

El caso de uso **AVC-MVC** mostrado en el recuadro de la página 136 sigue el guión común para los casos de uso formales. El *objetivo en contexto* identifica el alcance general del caso de

9 En este caso, otro actor, **administrador del sistema**, tendría que configurar el plano de la casa, instalar e inicializar todas las cámaras (por ejemplo, asignar una identificación a los equipos) y probar cada una para garantizar que se encuentren accesibles por el sistema y a través del plano de la casa.

uso. La *precondición* describe lo que se sabe que es verdadero antes de que inicie el caso de uso. El *disparador* (o *trigger*) identifica el evento o condición que "hace que comience el caso de uso" [Coc01b]. El *escenario* enlista las acciones específicas que requiere el actor, y las respuestas apropiadas del sistema. Las *excepciones* identifican las situaciones detectadas cuando se mejora el caso de uso preliminar (véase la sección 6.2.2). Pueden incluirse o no encabezados adicionales y se explican por sí mismos en forma razonable.

CASASEGURA

Formato de caso de uso para vigilancia

Caso de uso: Acceder a la vigilancia con cámaras por internet, mostrar vistas de cámaras (AVC-MVC).

Iteración: 2, última modificación: 14 de enero por V. Raman.

Actor principal: Propietario.

Objetivo en contexto: Ver la salida de las cámaras colocadas en la casa desde cualquier ubicación remota por medio de internet.

Precondiciones: El sistema debe estar configurado por completo; deben obtenerse las identificaciones y claves de usuario apropiadas.

Disparador: El propietario decide ver dentro de la casa mientras está fuera.

Escenario:

1. El propietario se registra en el sitio web *Productos CasaSegura*.
2. El propietario introduce su identificación de usuario.
3. El propietario proporciona dos claves (cada una con longitud de al menos ocho caracteres).
4. El sistema despliega todos los botones de las funciones principales.
5. El propietario selecciona "vigilancia" entre los botones de funciones principales.
6. El propietario escoge "seleccionar una cámara".
7. El sistema muestra el plano de la casa.
8. El propietario selecciona un ícono de cámara en el plano de la casa.
9. El propietario pulsa el botón "vista".
10. El sistema muestra la ventana de la vista de la cámara identificada.
11. El sistema presenta una salida de video dentro de la ventana de vistas, con una velocidad de un cuadro por segundo.

Excepciones:

1. La identificación o las claves son incorrectas o no se reconocen (véase el caso de uso **Validar identificación y claves**).
2. La función de vigilancia no está configurada para este sistema (el sistema muestra el mensaje de error apropiado; véase el caso de uso **Configurar la función de vigilancia**).
3. El propietario selecciona "Mirar vistas reducidas de todas las cámaras" (véase el caso de uso **Mirar vistas reducidas de todas las cámaras**).
4. No se dispone o no se ha configurado el plano de la casa (se muestra el mensaje de error apropiado y véase el caso de uso **Configurar plano de la casa**).
5. Se encuentra una condición de alarma (véase el caso de uso **Condición de alarma encontrada**).

Prioridad: Moderada, por implementarse después de las funciones básicas.

Cuándo estará disponible: En el tercer incremento.

Frecuencia de uso: Frecuencia moderada.

Canal al actor: A través de un navegador con base en PC y conexión a internet.

Actores secundarios: Administrador del sistema, cámaras.

Canales a los actores secundarios:

1. Administrador del sistema: sistema basado en PC.
2. Cámaras: conectividad inalámbrica.

Asuntos pendientes:

1. ¿Qué mecanismos protegen el uso no autorizado de esta capacidad por parte de los empleados de *Productos CasaSegura*?
2. Es suficiente la seguridad? El acceso ilegal a esta característica representaría una invasión grave de la privacidad.
3. ¿Será aceptable la respuesta del sistema por internet dado el ancho de banda que requieren las vistas de las cámaras?
4. ¿Desarrollaremos una capacidad que provea el video a una velocidad más alta en cuadros por segundo cuando se disponga de conexiones con un ancho de banda mayor?

WebRef

¿Cuándo se ha terminado de escribir casos de uso? Para un análisis benéfico de esto, consulte la dirección **ootips.org/use-cases-done.html**

En muchos casos, no hay necesidad de crear una representación gráfica de un escenario de uso. Sin embargo, la representación con diagramas facilita la comprensión, en particular cuando el escenario es complejo. Como ya se dijo en este libro, UML cuenta con la capacidad de hacer diagramas de casos de uso. La figura 6.4 ilustra un diagrama de caso de uso preliminar para el producto *CasaSegura*. Cada caso de uso está representado por un óvalo. En esta sección sólo se estudia el caso de uso **AVC-MVC**.

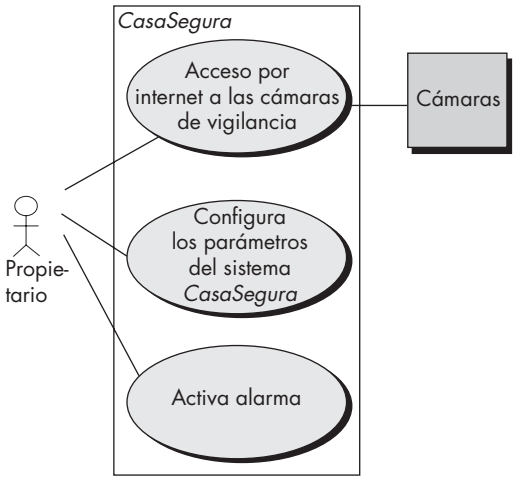


FIGURA 6.4
Diagrama de caso de uso preliminar para el sistema *CasaSegura*

Toda notación de modelado tiene sus limitaciones, y la del caso de uso no es la excepción. Como cualquier otra forma de descripción escrita, un caso de uso es tan bueno como lo sea(n) su(s) autor(es). Si la descripción es poco clara, el caso de uso será confuso o ambiguo. Un caso de uso se centra en los requerimientos funcionales y de comportamiento, y por lo general es inapropiado para requerimientos disfuncionales. Para situaciones en las que el modelo de requerimientos deba tener detalle y precisión significativos (por ejemplo, sistemas críticos de seguridad), tal vez no sea suficiente un caso de uso.

Sin embargo, el modelado basado en escenarios es apropiado para la gran mayoría de todas las situaciones que encontrará un ingeniero de software. Si se desarrolla bien, el caso de uso proporciona un beneficio sustancial como herramienta de modelado.

6.3 MODELOS UML QUE PROPORCIONAN EL CASO DE USO

Hay muchas situaciones de modelado de requerimientos en las que un modelo basado en texto —incluso uno tan sencillo como un caso de uso— tal vez no brinde información en forma clara y concisa. En tales casos, es posible elegir de entre una amplia variedad de modelos UML gráficos.

6.3.1 Desarrollo de un diagrama de actividades

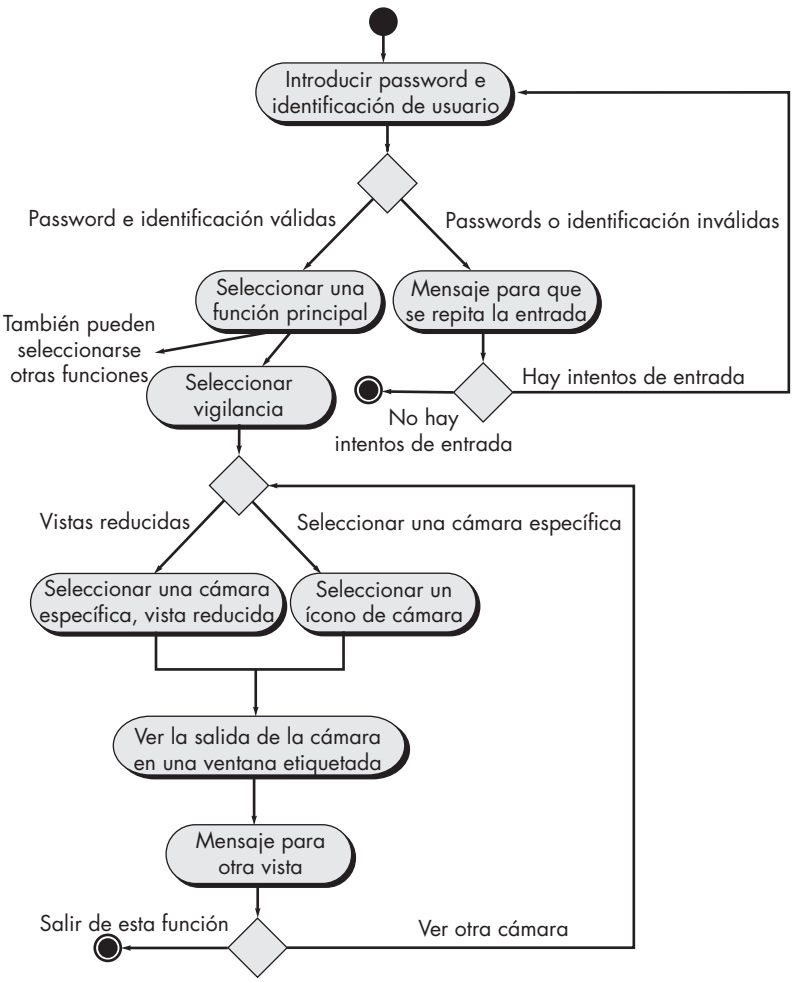
PUNTO CLAVE
Un diagrama de actividades UML representa las acciones y decisiones que ocurren cuando se realiza cierta función.

El diagrama de actividad UML enriquece el caso de uso al proporcionar una representación gráfica del flujo de interacción dentro de un escenario específico. Un diagrama de actividades es similar a uno de flujo, y utiliza rectángulos redondeados para denotar una función específica del sistema, flechas para representar flujo a través de éste, rombos de decisión para ilustrar una ramificación de las decisiones (cada flecha que salga del rombo se etiqueta) y líneas continuas para indicar que están ocurriendo actividades en paralelo. En la figura 6.5 se presenta un diagrama de actividades para el caso de uso **AVC-MVC**. Debe observarse que el diagrama de actividades agrega detalles adicionales que no se mencionan directamente (pero que están implícitos) en el caso de uso.

Por ejemplo, un usuario quizá sólo haga algunos intentos de introducir su **identificación** y **password**. Esto se representa por el rombo de decisión debajo de "Mensaje para que se repita la entrada".

FIGURA 6.5

Diagrama de actividades para la función Acceder a la vigilancia con cámaras por internet, mostrar vistas de cámaras.



6.3.2 Diagramas de canal (*swimlane*)

PUNTO CLAVE

Un diagrama de canal (*swimlane*) representa el flujo de acciones y decisiones e indica qué actores efectúan cada una de ellas.

El *diagrama de canal* de UML es una variación útil del diagrama de actividades y permite representar el flujo de actividades descritas por el caso de uso; al mismo tiempo, indica qué actor (si hubiera muchos involucrados en un caso específico de uso) o clase de análisis (se estudia más adelante, en este capítulo) es responsable de la acción descrita por un rectángulo de actividad. Las responsabilidades se representan con segmentos paralelos que dividen el diagrama en forma vertical, como los canales o carriles de una alberca.

Son tres las clases de análisis: **Propietario**, **Cámara** e **Interfaz**, que tienen responsabilidad directa o indirecta en el contexto del diagrama de actividades representado en la figura 6.5. En relación con la figura 6.6, el diagrama de actividades se reacomodó para que las actividades asociadas con una clase de análisis particular queden dentro del canal de dicha clase. Por ejemplo, la clase **Interfaz** representa la interfaz de usuario como la ve el propietario. El diagrama de actividades tiene dos mensajes que son responsabilidad de la interfaz: "mensaje para que se repita la entrada" y "mensaje para otra vista". Estos mensajes y las decisiones asociadas con ellos caen dentro del canal **Interfaz**. Sin embargo, las flechas van de ese canal de regreso al de **Propietario**, donde ocurren las acciones de éste.

Cita:

"Un buen modelo guía el pensamiento; uno malo lo desvía."

Brian Marick

Los casos de uso, junto con los diagramas de actividades y de canal, están orientados al procedimiento. Representan la manera en la que los distintos actores invocan funciones específicas (u otros pasos del procedimiento) para satisfacer los requerimientos del sistema. Pero

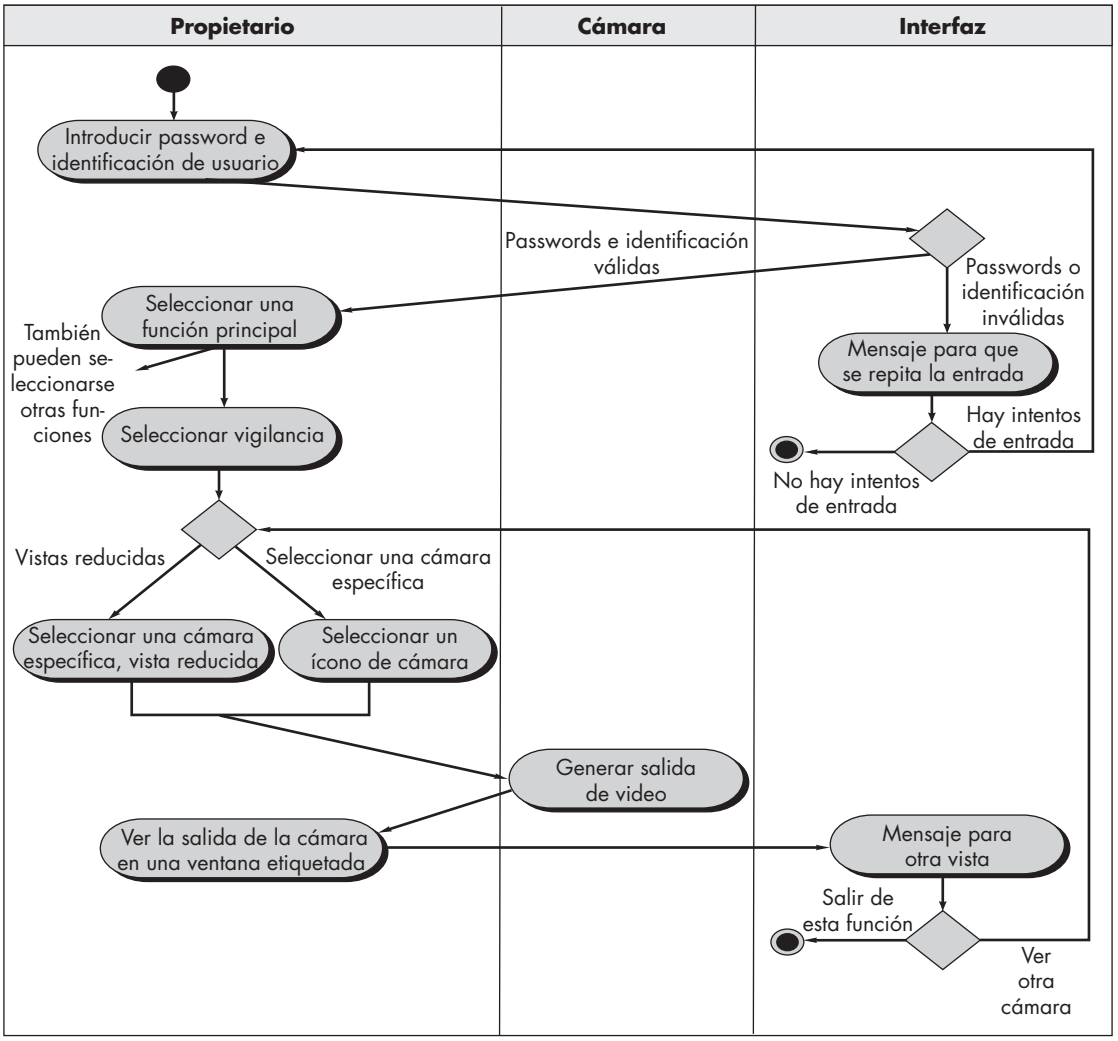


FIGURA 6.6
Diagrama de canal para la función Acceder a la vigilancia con cámaras por internet, mostrar vistas de cámaras

una vista del procedimiento de los requerimientos representa una sola dimensión del sistema. En la sección 6.4 se estudia el espacio de información y la forma en la que se representan los datos de requerimientos.

6.4 CONCEPTOS DE MODELADO DE DATOS

WebRef
En la dirección **www.datamodel.org**, hay información útil sobre el modelado de datos.

Si los requerimientos del software incluyen la necesidad de crear, ampliar o hacer interfaz con una base de datos, o si deben construirse y manipularse estructuras de datos complejas, el equipo del software tal vez elija crear un *modelo de datos* como parte del modelado general de los requerimientos. Un ingeniero o analista de software define todos los objetos de datos que se procesan dentro del sistema, la relación entre ellos y otro tipo de información que sea pertinente para las relaciones. El *diagrama entidad-relación* (DER) aborda dichos aspectos y representa todos los datos que se introducen, almacenan, transforman y generan dentro de una aplicación.

? ¿Cómo se manifiesta un objeto de datos en el contexto de una aplicación?

6.4.1 Objetos de datos

Un *objeto de datos* es una representación de información compuesta que debe ser entendida por el software. *Información compuesta* quiere decir algo que tiene varias propiedades o atributos

diferentes. Por tanto, el ancho (un solo valor) no sería un objeto de datos válido, pero las **dimensiones** (que incorporan altura, ancho y profundidad) sí podrían definirse como un objeto.

Un objeto de datos puede ser una entidad externa (por ejemplo, cualquier cosa que produzca o consuma información), una cosa (por ejemplo, un informe o pantalla), una ocurrencia (como una llamada telefónica) o evento (por ejemplo, una alarma), un rol (un vendedor), una unidad organizacional (por ejemplo, el departamento de contabilidad), un lugar (como una bodega) o estructura (como un archivo). Por ejemplo, una **persona** o un **auto** pueden considerarse como objetos de datos en tanto cada uno se define en términos de un conjunto de atributos. La descripción del objeto de datos incorpora a ésta todos sus atributos.

PUNTO CLAVE

Un objeto de datos es una representación de cualquier información compuesta que se procese en el software.

Un objeto de datos contiene sólo datos —dentro de él no hay referencia a las operaciones que se apliquen sobre los datos—.[10] Entonces, el objeto de datos puede representarse en forma de tabla, como la que se muestra en la figura 6.7. Los encabezados de la tabla reflejan atributos del objeto. En este caso, un auto se define en términos de **fabricante, modelo, número de serie, tipo de carrocería, color** y **propietario**. El cuerpo de la tabla representa instancias específicas del objeto de datos. Por ejemplo, un Chevy Corvette es una instancia del objeto de datos **auto**.

6.4.2 Atributos de los datos

Los *atributos de los datos* definen las propiedades de un objeto de datos y tienen una de tres diferentes características. Se usan para 1) nombrar una instancia del objeto de datos, 2) describir la instancia o 3) hacer referencia a otra instancia en otra tabla. Además, debe definirse como identificador uno o más de los atributos —es decir, el atributo identificador se convierte en una "llave" cuando se desea encontrar una instancia del objeto de datos—. En ciertos casos, los valores para el (los) identificador(es) son únicos, aunque esto no es una exigencia. En relación con el objeto de datos **auto**, un identificador razonable sería el **número de serie**.

PUNTO CLAVE

Los atributos nombran a un objeto de datos, describen sus características y, en ciertos casos, hacen referencia a otro objeto.

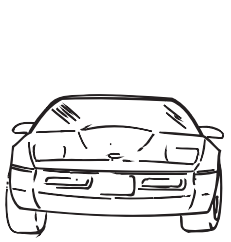
El conjunto de atributos que es apropiado para un objeto de datos determinado se define entendiendo el contexto del problema. Los atributos para **auto** podrían servir bien para una aplicación que usara un departamento de vehículos motorizados, pero serían inútiles para una compañía automotriz que necesitara hacer software de control de manufactura. En este último caso, los atributos para **auto** quizá también incluyan **número de serie, tipo de carrocería** y **color**, pero tendrían que agregarse muchos otros (por ejemplo, **código interior, tipo de tracción, indicador de paquete de recorte, tipo de transmisión**, etc.) para hacer de **auto** un objeto significativo en el contexto de control de manufactura.

WebRef

Para aquellos que intentan hacer modelado de datos, es importante un concepto llamado "normalización". En la dirección **www.datamodel.org** se encuentra una introducción útil.

FIGURA 6.7

Representación tabular de objetos de datos



Nombrar los atributos

Liga un objeto de datos con otro, en este caso el propietario

Identificador | Atributos descriptivos | Atributos referenciales

Fabricante	Modelo	Número de serie	Carrocería	Color	Propietario
Lexus	LS400	AB123. . .	Sedán	Blanco	RSP
Chevy	Corvette	X456. . .	Deportivo	Rojo	CCD
BMW	750iL	XZ765. . .	Cupé	Blanco	LJL
Ford	Taurus	Q12A45. . .	Sedán	Azul	BLF

Instancia

10 Esta distinción separa al objeto de datos de la clase u objeto definidos como parte del enfoque orientado a objetos (véase el apéndice 2).

INFORMACIÓN

Objetos de datos y clases orientadas a objetos: ¿son lo mismo?

Al analizar objetos de datos es común que surja una pregunta: ¿un objeto de datos es lo mismo que una clase orientada[11] a objetos? La respuesta es "no".

Un objeto de datos define un aspecto de datos compuestos; es decir, incorpora un conjunto de características de datos individuales (atributos) y da al conjunto un nombre (el del objeto de datos).

Una clase orientada a objetos encierra atributos de datos, pero también incorpora las operaciones (métodos) que los manipulan y que están determinadas por dichos atributos. Además, la definición de clases implica una infraestructura amplia que es parte del enfoque de la ingeniería de software orientada a objetos. Las clases se comunican entre sí por medio de mensajes, se organizan en jerarquías y tienen características hereditarias para los objetos que son una instancia de una clase.

6.4.3 Relaciones

Las relaciones indican la manera en la que los objetos de datos se conectan entre sí.

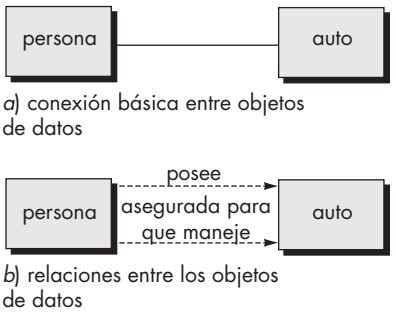
Los objetos de datos están conectados entre sí de diferentes maneras. Considere dos objetos de datos, **persona** y **auto**. Estos objetos se representan con la notación simple que se ilustra en la figura 6.8*a*). Se establece una conexión entre **persona** y **auto** porque ambos objetos están relacionados. Pero, ¿cuál es esa relación? Para determinarlo, debe entenderse el papel de las personas (propiedad, en este caso) y los autos dentro del contexto del software que se va a elaborar. Se establece un conjunto de parejas objeto/relación que definan las relaciones relevantes. Por ejemplo,

- Una persona *posee* un auto.
- Una persona *es asegurada para que maneje* un auto.

Las relaciones *posee* y *es asegurada para que maneje* definen las conexiones relevantes entre **persona** y **auto**. La figura 6.8*b*) ilustra estas parejas objeto-relación. Las flechas en esa figura dan información importante sobre la dirección de la relación y es frecuente que reduzcan las ambigüedades o interpretaciones erróneas.

FIGURA 6.8

Relaciones entre objetos de datos



11 Los lectores que no estén familiarizados con los conceptos y terminología de la orientación a objetos deben consultar el breve instructivo que se presenta en el apéndice 2.

INFORMACIÓN

Diagramas entidad-relación

La pareja objeto-relación es la piedra angular del modelo de datos. Estas parejas se representan gráficamente con el uso del diagrama entidad-relación (DER).[12] El DER fue propuesto por primera vez por Peter Chen [Che77] para diseñar sistemas de bases de datos relacionales y ha sido ampliado por otras personas. Se identifica un conjunto de componentes primarios para el DER: objetos de datos, atributos, relaciones y distintos indicadores de tipo. El propósito principal del DER es representar objetos de datos y sus relaciones.

Ya se presentó la notación DER básica. Los objetos de datos se representan con un rectángulo etiquetado. Las relaciones se indican con una línea etiquetada que conecta objetos. En ciertas variantes del DER, la línea de conexión contiene un rombo con la leyenda de la relación. Las conexiones entre los objetos de datos y las relaciones se establecen con el empleo de varios símbolos especiales que indican cardinalidad y modalidad.[13] Si el lector está interesado en obtener más información sobre el modelado de datos y el diagrama entidad-relación, consulte [Hob06] o [Sim05].

HERRAMIENTAS DE SOFTWARE

Modelado de datos

Objetivo: Las herramientas de modelado de datos dan a un ingeniero de software la capacidad de representar objetos de datos, sus características y relaciones. Se usan sobre todo para aplicaciones de grandes bases de datos y otros proyectos de sistemas de información, y proveen medios automatizados para crear diagramas completos de entidad-relación, diccionarios de objetos de datos y modelos relacionados.

Mecánica: Las herramientas de esta categoría permiten que el usuario describa objetos de datos y sus relaciones. En ciertos casos, utilizan notación DER. En otros, modelan relaciones con el empleo de un mecanismo diferente. Es frecuente que las herramientas en esta categoría se usen como parte del diseño de una base de datos y que permitan la creación de su modelo con la generación de un esquema para sistemas comunes de administración de bases de datos comunes (DBMS).

Herramientas representativas:[14]

AllFusion ERWin, desarrollada por Computer Associates (**www3.ca.com**), ayuda en el diseño de objetos de datos, estructura apropiada y elementos clave para las bases de datos.

ER/Studio, desarrollada por Embarcadero Software (**www.embarcadero.com**), da apoyo al modelado entidad-relación.

Oracle Designer, desarrollada por Oracle Systems (**www.oracle.com**), "modela procesos de negocios, entidades y relaciones de datos [que] se transforman en diseños para los que se generan aplicaciones y bases de datos completas".

Visible Analyst, desarrollada por Visible Systems (**www.visible.com**), da apoyo a varias funciones de modelado del análisis, incluso modelado de datos.

6.5 MODELADO BASADO EN CLASES

El modelado basado en clases representa los objetos que manipulará el sistema, las operaciones (también llamadas *métodos* o *servicios*) que se aplicarán a los objetos para efectuar la manipulación, las relaciones (algunas de ellas jerárquicas) entre los objetos y las colaboraciones que tienen lugar entre las clases definidas. Los elementos de un modelo basado en clases incluyen las clases y los objetos, atributos, operaciones, modelos clase-responsabilidad-colaborador (CRC), diagramas de colaboración y paquetes. En las secciones siguientes se presenta una serie de lineamientos informales que ayudarán a su identificación y representación.

---

12 Aunque algunas aplicaciones de diseño de bases de datos aún emplean el DER, ahora se utiliza la notación UML (véase el apéndice 1) para el diseño de datos.

13 La *cardinalidad* de una pareja objeto-relación especifica "el número de ocurrencias de uno [objeto] que se relaciona con el número de ocurrencias de otro [objeto]" [Til93]. La *modalidad* de una relación es 0 si no hay necesidad explícita para que ocurra la relación o si ésta es opcional. La modalidad es 1 si una ocurrencia de la relación es obligatoria.

14 Las herramientas mencionadas aquí no son obligatorias sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

6.5.1 Identificación de las clases de análisis

Cita:

"El problema realmente difícil es descubrir en primer lugar cuáles son los objetos correctos [clases]."

Carl Argila

Al mirar una habitación, se observa un conjunto de objetos físicos que se identifican, clasifican y definen fácilmente (en términos de atributos y operaciones). Pero cuando se "ve" el espacio del problema de una aplicación de software, las clases (y objetos) son más difíciles de concebir.

Se comienza por identificar las clases mediante el análisis de los escenarios de uso desarrollados como parte del modelo de requerimientos y la ejecución de un "análisis gramatical" [Abb83] sobre los casos de uso desarrollados para el sistema que se va a construir. Las clases se determinan subrayando cada sustantivo o frase que las incluya para introducirlo en una tabla simple. Deben anotarse los sinónimos. Si la clase (sustantivo) se requiere para implementar una solución, entonces forma parte del espacio de solución; de otro modo, si sólo es necesaria para describir la solución, es parte del espacio del problema.

Pero, ¿qué debe buscarse una vez identificados todos los sustantivos? Las *clases de análisis* se manifiestan en uno de los modos siguientes:

? ¿Cómo se manifiestan las clases en tantos elementos del espacio de solución?

- *Entidades externas* (por ejemplo, otros sistemas, dispositivos y personas) que producen o consumen la información que usará un sistema basado en computadora.
- *Cosas* (reportes, pantallas, cartas, señales, etc.) que forman parte del dominio de información para el problema.
- *Ocurrencias o eventos* (como una transferencia de propiedad o la ejecución de una serie de movimientos de un robot) que suceden dentro del contexto de la operación del sistema.
- *Roles* (gerente, ingeniero, vendedor, etc.) que desempeñan las personas que interactúan con el sistema.
- *Unidades organizacionales* (división, grupo, equipo, etc.) que son relevantes para una aplicación.
- *Lugares* (piso de manufactura o plataforma de carga) que establecen el contexto del problema y la función general del sistema.
- *Estructuras* (sensores, vehículos de cuatro ruedas, computadoras, etc.) que definen una clase de objetos o clases relacionadas de éstos.

Esta clasificación sólo es una de muchas propuestas en la bibliografía.[15] Por ejemplo, Budd [Bud96] sugiere una taxonomía de clases que incluye *productores* (fuentes) y *consumidores* (sumideros) de datos, *administradores de datos*, *vista*, *clases de observador* y *clases de auxiliares*.

También es importante darse cuenta de lo que no son las clases u objetos. En general, una clase nunca debe tener un "nombre de procedimiento imperativo" [Cas89]. Por ejemplo, si los desarrolladores del software de un sistema de imágenes médicas definieron un objeto con el nombre **InvertirImagen** o incluso **InversióndeImagen**, cometerían un error sutil. La **Imagen** obtenida del software podría ser, por supuesto, una clase (algo que es parte del dominio de la información). La inversión de la imagen es una operación que se aplica al objeto. Es probable que la inversión esté definida como una operación para el objeto **Imagen**, pero no lo estaría como clase separada con la connotación "inversión de imagen". Como afirma Cashman [Cas89]: "el intento de la orientación a objetos es contener, pero mantener separados, los datos y las operaciones sobre ellos".

Para ilustrar cómo podrían definirse las clases del análisis durante las primeras etapas del modelado, considere un análisis gramatical (los sustantivos están subrayados, los verbos apa-

15 En la sección 6.5.4 se estudia otra clasificación importante que define las clases *entidad*, *frontera* y *controladora*.

recen en cursivas) de una narración de procesamiento[16] para la función de seguridad de *CasaSegura*.

La función de seguridad CasaSegura *permite* que el propietario *configure* el sistema de seguridad cuando se *instala*, *vigila* todos los sensores *conectados* al sistema de seguridad e *interactúa* con el propietario a través de internet, una PC o panel de control.

Durante la instalación, la PC de CasaSegura se utiliza para *programar* y *configurar* el sistema. Se asigna a cada sensor un número y tipo, se programa un password maestro para *activar* y *desactivar* el sistema y se *introducen* números telefónicos para *marcar* cuando ocurre un evento de sensor.

Cuando se *reconoce* un evento de sensor, el software *invoca* una alarma audible instalada en el sistema. Después de un tiempo de retraso que *especifica* el propietario durante las actividades de configuración del sistema, el software marca un número telefónico de un servicio de monitoreo, *proporciona* información acerca de la ubicación y *reporta* la naturaleza del evento detectado. El número telefónico se *vuelve a marcar* cada 20 segundos hasta que se *obtiene* la conexión telefónica.

El propietario *recibe* información de seguridad a través de un panel de control, la PC o un navegador, lo que en conjunto se llama interfaz. La interfaz *despliega* mensajes de aviso e información del estado del sistema en el panel de control, la PC o la ventana del navegador. La interacción del propietario tiene la siguiente forma…

CONSEJO

El análisis gramatical no es a prueba de todo, pero da un impulso excelente para arrancar si se tienen dificultades para definir objetos de datos y las transformaciones que operan sobre ellos.

Con los sustantivos se proponen varias clases potenciales:

Clase potencial	Clasificación general
propietario	rol de entidad externa
sensor	entidad externa
panel de control	entidad externa
instalación	ocurrencia
sistema (alias sistema de seguridad)	cosa
número, tipo	no objetos, atributos de sensor
password maestro	cosa
número telefónico	cosa
evento de sensor	ocurrencia
alarma audible	entidad externa
servicio de monitoreo	unidad organizacional o entidad externa

La lista continuará hasta que se hayan considerado todos los sustantivos en la narrativa de procesamiento. Observe que cada entrada en la lista se llama objeto potencial. El lector debe considerar cada una antes de tomar la decisión final.

Coad y Yourdon [Coa91] sugieren seis características de selección que deben usarse cuando se considere cada clase potencial para incluirla en el modelo de análisis:

? ¿Cómo determino si una clase potencial debe, en realidad, ser una clase de análisis?

1. *Información retenida.* La clase potencial será útil durante el análisis sólo si debe recordarse la información sobre ella para que el sistema pueda funcionar.
2. *Servicios necesarios*. La clase potencial debe tener un conjunto de operaciones identificables que cambien en cierta manera el valor de sus atributos.

16 Una narración de procesamiento es similar al caso de uso en su estilo, pero algo distinto en su propósito. La narración de procesamiento hace una descripción general de la función que se va a desarrollar. No es un escenario escrito desde el punto de vista de un actor. No obstante, es importante observar que el análisis gramatical también puede emplearse para todo caso de uso desarrollado como parte de la obtención de requerimientos (indagación).

3. *Atributos múltiples.* Durante el análisis de los requerimientos, la atención debe estar en la información "principal"; en realidad, una clase con un solo atributo puede ser útil durante el diseño, pero es probable que durante la actividad de análisis se represente mejor como un atributo de otra clase.
4. *Atributos comunes*. Para la clase potencial se define un conjunto de atributos y se aplican éstos a todas las instancias de la clase.
5. *Operaciones comunes*. Se define un conjunto de operaciones para la clase potencial y éstas se aplican a todas las instancias de la clase.
6. *Requerimientos esenciales.* Las entidades externas que aparezcan en el espacio del problema y que produzcan o consuman información esencial para la operación de cualquier solución para el sistema casi siempre se definirán como clases en el modelo de requerimientos.

Cita:

"Las clases luchan, algunas triunfan, otras son eliminadas."

Mao Tse Tung

Para que se considere una clase legítima para su inclusión en el modelo de requerimientos, un objeto potencial debe satisfacer todas (o casi todas) las características anteriores. La decisión de incluir clases potenciales en el modelo de análisis es algo subjetiva, y una evaluación posterior tal vez haga que un objeto se descarte o se incluya de nuevo. Sin embargo, el primer paso del modelado basado en clases es la definición de éstas, y deben tomarse las medidas respectivas (aun las subjetivas). Con esto en mente, se aplicarán las características de selección a la lista de clases potenciales de *CasaSegura*:

Clase potencial	Número de característica que se aplica
propietario	rechazada: 1 y 2 fallan, aunque la 6 aplica
sensor	aceptada: se aplican todas
panel de control	aceptada: se aplican todas
instalación	rechazada
sistema (alias sistema de seguridad)	aceptada: se aplican todas
número, tipo	rechazada: 3 fallan, atributos de sensores
password maestro	rechazada: 3 falla
número telefónico	rechazada: 3 falla
evento de sensor	aceptada: se aplican todas
alarma audible	aceptada: se aplican 2, 3, 4, 5 y 6
servicio de monitoreo	rechazada: 1 y 2 fallan aunque la 6 aplica

Debe notarse que: 1) la lista anterior no es exhaustiva; para completar el modelo tendrían que agregarse clases adicionales; 2) algunas de las clases potenciales rechazadas se convertirán en atributos para otras que sí fueron aceptadas (por ejemplo, **número** y **tipo** son atributos de **Sensor**, y **password maestro** y **número telefónico** pueden convertirse en atributos de **Sistema**); 3) diferentes enunciados del problema harían que se tomaran decisiones distintas para "aceptar o rechazar" (por ejemplo, si cada propietario tuviera una clave individual o se identificara con reconocimiento de voz, la clase **Propietario** satisfaría las características 1 y 2, y se aceptaría).

6.5.2 Especificación de atributos

PUNTO CLAVE

Los atributos son el conjunto de objetos de datos que definen por completo la clase en el contexto del problema.

Los *atributos* describen una clase que ha sido seleccionada para incluirse en el modelo de requerimientos. En esencia, son los atributos los que definen la clase (esto aclara lo que significa la clase en el contexto del espacio del problema). Por ejemplo, si se fuera a construir un sistema que analiza estadísticas de jugadores de béisbol profesionales, los atributos de la clase **Jugador** serían muy distintos de los que tendría la misma clase cuando se usara en el contexto del sis-

tema de pensiones de dicho deporte. En la primera, atributos tales como **nombre, porcentaje de bateo, porcentaje de fildeo, años jugados** y **juegos jugados** serían relevantes. Para la segunda, algunos de los anteriores sí serían significativos, pero otros se sustituirían (o se crearían) por atributos tales como **salario promedio, crédito hacia el retiro completo, opciones del plan de pensiones elegido, dirección de correo**, etcétera.

Para desarrollar un conjunto de atributos significativos para una clase de análisis, debe estudiarse cada caso de uso y seleccionar aquellas "cosas" que "pertenezcan" razonablemente a la clase. Además, debe responderse la pregunta siguiente para cada clase: "¿qué aspectos de los datos (compuestos o elementales) definen por completo esta clase en el contexto del problema en cuestión?"

Para ilustrarlo, se considera la clase **Sistema** definida para *CasaSegura*. El propietario configura la función de seguridad para que refleje la información de los sensores, la respuesta de la alarma, la activación o desactivación, la identificación, etc. Estos datos compuestos se representan del modo siguiente:

información de identificación = identificación del sistema + número telefónico de verificación + estado del sistema
información de respuesta de la alarma = tiempo de retraso + número telefónico
información de activación o desactivación = password maestro + número de intentos permisibles + password temporal

Cada uno de los datos a la derecha del signo igual podría definirse más, hasta un nivel elemental, pero para nuestros propósitos constituye una lista razonable de atributos para la clase **Sistema** (parte sombreada de la figura 6.9).

Los sensores forman parte del sistema general *CasaSegura*, pero no están enlistados como datos o atributos en la figura 6.9. **Sensor** ya se definió como clase, y se asociarán múltiples objetos **Sensor** con la clase **Sistema**. En general, se evita definir algo como atributo si más de uno va a asociarse con la clase.

6.5.3 Definición de las operaciones

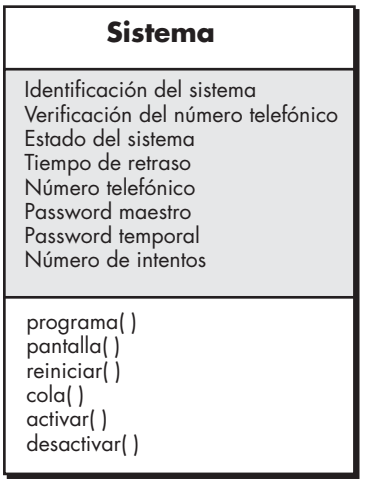
Cuando se definen operaciones para una clase de análisis, hay que centrarse en el comportamiento orientado al problema y no en los comportamientos requeridos para su implementación.

Las *operaciones* definen el comportamiento de un objeto. Aunque existen muchos tipos distintos de operaciones, por lo general se dividen en cuatro categorías principales: 1) operaciones que manipulan datos en cierta manera (por ejemplo, los agregan, eliminan, editan, seleccionan, etc.), 2) operaciones que realizan un cálculo, 3) operaciones que preguntan sobre el estado de un objeto y 4) operaciones que vigilan un objeto en cuanto a la ocurrencia de un evento de control. Estas funciones se llevan a cabo con operaciones sobre los atributos o sobre asociaciones de éstos (véase la sección 6.5.5). Por tanto, una operación debe tener "conocimiento" de la naturaleza de los atributos y de las asociaciones de la clase.

Como primera iteración al obtener un conjunto de operaciones para una clase de análisis, se estudia otra vez una narración del procesamiento (o caso de uso) y se eligen aquellas que pertenezcan razonablemente a la clase. Para lograr esto, de nuevo se efectúa el análisis gramatical y se aíslan los verbos. Algunos de éstos serán operaciones legítimas y se conectarán con facilidad a una clase específica. Por ejemplo, de la narración del procesamiento de *CasaSegura* ya presentada en este capítulo, se observa que "se *asigna* a sensor un número y tipo" o "se *programa* un password maestro para *activar y desactivar* el sistema" indican cierto número de cosas:

- Que una operación *asignar( )* es relevante para la clase **Sensor**.
- Que se aplicará una operación *programar( )* a la clase **Sistema**.
- Que *activar( )* y *desactivar( )* son operaciones que se aplican a la clase **Sistema**.

FIGURA 6.9
Diagrama de clase para la clase sistema



Hasta no hacer más investigaciones, es probable que la operación *programar( )* se divida en cierto número de suboperaciones específicas adicionales que se requieren para configurar el sistema. Por ejemplo, *programar( )* implica la especificación de números telefónicos, la configuración de las características del sistema (por ejemplo, crear la tabla de sensores, introducir las características de la alarma, etc.) y la introducción de la(s) clave(s). Pero, de momento, *programar( )* se especifica como una sola operación.

Además del análisis gramatical, se obtiene más perspectiva sobre otras operaciones si se considera la comunicación que ocurre entre los objetos. Éstos se comunican con la transmisión de mensajes entre sí. Antes de continuar con la especificación de operaciones, se estudiará esto con más detalle.

CASASEGURA

Modelos de clase

La escena: Cubículo de Ed, cuando comienza el modelado de los requerimientos.

Participantes: Jamie, Vinod y Ed, todos ellos miembros del equipo de ingeniería de software para *CasaSegura*.

La conversación:

[Ed ha estado trabajando para obtener las clases a partir del formato del caso de uso para AVC-MVC (presentado en un recuadro anterior de este capítulo) y expone a sus colegas las que ha obtenido].

Ed: Entonces, cuando el propietario quiere escoger una cámara, la tiene que elegir del plano. Definí una clase llamada **Plano**. Éste es el diagrama.
(Observan la figura 6.10.)

Jamie: Entonces, **Plano** es un objeto que agrupa paredes, puertas, ventanas y cámaras. Eso significa esas líneas con leyendas, ¿verdad?

Ed: Sí, se llaman "asociaciones". Una clase se asocia con otra de acuerdo con las asociaciones que se ven (las asociaciones se estudian en la sección 6.5.5).

Vinod: Es decir, el plano real está constituido por paredes que contienen en su interior cámaras y sensores. ¿Cómo sabe el plano dónde colocar estos objetos?

Ed: No lo sabe, pero las otras clases sí. Mira los atributos de, digamos, **SegmentodePared**, que se usa para construir una pared. El segmento de muro tiene coordenadas de inicio y final, y la operación *draw( )* hace el resto.

Jamie: Y lo mismo vale para las ventanas y puertas. Parece como si cámara tuviera algunos atributos adicionales.

Ed: Sí. Los necesito para dar información del alcance y el acercamiento.

Vinod: Tengo una pregunta. ¿Por qué tiene la cámara una identificación pero las demás no? Veo que tienes un atributo llamado **ParedSiguiente**. ¿Cómo sabe **SegmentodePared** cuál será la pared siguiente?

Ed: Buena pregunta, pero, como dijimos, ésa es una decisión de diseño, por lo que la voy a retrasar hasta...

Jamie: Momento... Apuesto a que ya lo has imaginado.

Ed (sonríe con timidez): Es cierto, voy a usar una estructura de lista que modelaré cuando vayamos a diseñar. Si somos puristas en cuanto a separar el análisis y el diseño, el nivel de detalle podría parecer sospechoso.

Jamie: Me parece muy bien, pero tengo más preguntas.

(Jamie hace preguntas que dan como resultado modificaciones menores.)

Vinod: ¿Tienes tarjetas CRC para cada uno de los objetos? Si así fuera, debemos actuar con ellas, sólo para estar seguros de que no hemos omitido nada.

Ed: No estoy seguro de cómo hacerlas.

Vinod: No es difícil y en verdad convienen. Te mostraré.

6.5.4 Modelado clase-responsabilidad-colaborador (CRC)

Cita:

"Un propósito de las tarjetas CRC es que fallen pronto, con frecuencia y en forma barata. Es mucho más barato desechar tarjetas que reorganizar una gran cantidad de código fuente."

C. Horstman

El modelado clase-responsabilidad-colaborador (CRC) [Wir90] proporciona una manera sencilla de identificación y organización de las clases que son relevantes para los requerimientos de un sistema o producto. Ambler [Amb95] describe el modelado CRC en la siguiente forma:

> Un modelo CRC en realidad es un conjunto de tarjetas índice estándar que representan clases. Las tarjetas se dividen en tres secciones. En la parte superior de la tarjeta se escribe el nombre de la clase, en la parte izquierda del cuerpo se enlistan las responsabilidades de la clase y en la derecha, los colaboradores.

En realidad, el modelo CRC hace uso de tarjetas índice reales o virtuales. El objetivo es desarrollar una representación organizada de las clases. Las *responsabilidades* son los atributos y ope-

FIGURA 6.10

Diagrama de clase para Plano (véase el análisis en el recuadro)

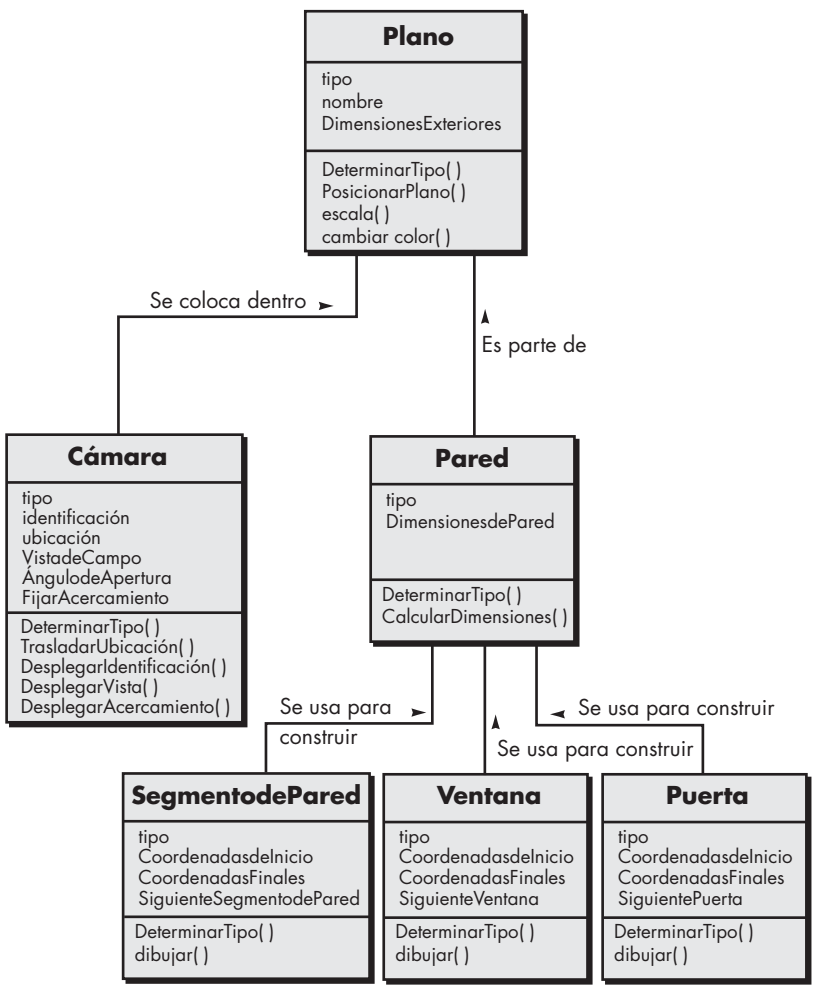
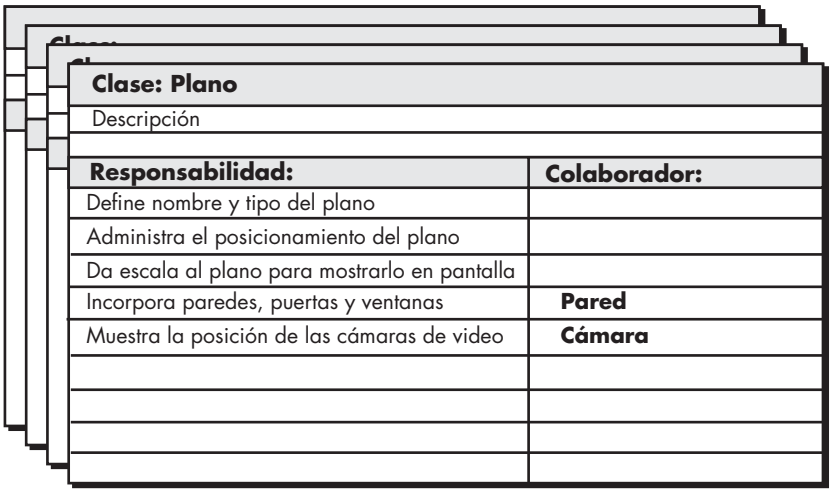


FIGURA 6.11
Modelo de tarjeta CRC índice



raciones relevantes para la clase. En pocas palabras, una responsabilidad es "cualquier cosa que la clase sepa o haga" [Amb95]. Los *colaboradores* son aquellas clases que se requieren para dar a una clase la información necesaria a fin de completar una responsabilidad. En general, una *colaboración* implica una solicitud de información o de cierta acción.

En la figura 6.11 se ilustra una tarjeta CRC índice sencilla para la clase **Plano**: la lista de responsabilidades en la tarjeta CRC es preliminar y está sujeta a agregados o modificaciones. Las clases **Pared** y **Cámara** se anotan frente a la responsabilidad que requerirá su colaboración.

WebRef
En la dirección **www.theumlcafe.com/a0079.htm** hay un análisis excelente de estos tipos de clase.

Clases. Al inicio de este capítulo se presentaron lineamientos básicos para identificar clases y objetos. La taxonomía de tipos de clase presentada en la sección 6.5.1 puede ampliarse con las siguientes categorías:

- *Clases de entidad*, también llamadas clases *modelo* o *de negocio*, se extraen directamente del enunciado del problema (por ejemplo, **Plano** y **Sensor**). Es común que estas clases representen cosas almacenadas en una base de datos y que persistan mientras dure la aplicación (a menos que se eliminen en forma específica).
- *Clases de frontera* se utilizan para crear la interfaz (por ejemplo, pantallas interactivas o reportes impresos) que el usuario mira y con la que interactúa cuando utiliza el software. Los objetos de entidad contienen información que es importante para los usuarios, pero no se muestran por sí mismos. Las clases de frontera se diseñan con la responsabilidad de administrar la forma en la que se presentan a los usuarios los objetos de entidad. Por ejemplo, una clase de frontera llamada **VentanadeCámara** tendría la responsabilidad de desplegar la salida de una cámara de vigilancia para el sistema *CasaSegura*.
- *Clases de controlador* administran una "unidad de trabajo" [UML03] de principio a fin. Es decir, las clases de controlador están diseñadas para administrar 1) la creación o actualización de objetos de entidad, 2) las instancias de los objetos de frontera en tanto obtienen información de los objetos de entidad, 3) la comunicación compleja entre conjuntos de objetos y 4) la validación de datos comunicados entre objetos o entre el usuario y la aplicación. En general, las clases de controlador no se consideran hasta haber comenzado la actividad de diseño.

Cita:
"Pueden clasificarse científicamente los objetos en tres grandes categorías: los que no funcionan, los que se descomponen y los que se pierden."
Rusell Baker

¿Qué lineamientos se aplican para asignar responsabilidades a las clases?

Responsabilidades. En las secciones 6.5.2 y 6.5.3 se presentaron los lineamientos básicos para identificar responsabilidades (atributos y operaciones). Wirfs-Brock *et al.* [Wir90] sugieren cinco lineamientos para asignar responsabilidades a las clases:

1. **La inteligencia del sistema debe estar distribuida entre las clases para enfrentar mejor las necesidades del problema.** Toda aplicación contiene cierto grado de inteligencia, es decir, lo que el sistema sabe y lo que puede hacer. Esta inteligencia se distribuye entre las clases de diferentes maneras. Las clases "tontas" (aquellas que tienen pocas responsabilidades) pueden modelarse para que actúen como subordinadas de ciertas clases "inteligentes" (las que tienen muchas responsabilidades). Aunque este enfoque hace directo el flujo del control en un sistema, tiene algunas desventajas: concentra toda la inteligencia en pocas clases, lo que hace que sea más difícil hacer cambios, y tiende a que se requieran más clases y por ello más trabajo de desarrollo.

   Si la inteligencia del sistema tiene una distribución más pareja entre las clases de una aplicación, cada objeto sabe algo, sólo hace unas cuantas cosas (que por lo general están bien identificadas) y la cohesión del sistema mejora.[17] Esto facilita el mantenimiento del software y reduce el efecto de los resultados colaterales del cambio.

   Para determinar si la inteligencia del sistema está distribuida en forma apropiada, deben evaluarse las responsabilidades anotadas en cada modelo de tarjeta CRC índice a fin de definir si alguna clase tiene una lista demasiado larga de responsabilidades. Esto indica una concentración de inteligencia.[18] Además, las responsabilidades de cada clase deben tener el mismo nivel de abstracción. Por ejemplo, entre las operaciones enlistadas para una clase agregada llamada **RevisarCuenta**, un revisor anota dos responsabilidades: *hacer el balance de la cuenta* y *eliminar comprobaciones concluidas*. La primera operación (responsabilidad) implica un procedimiento matemático complejo y lógico. La segunda es una simple actividad de oficina. Como estas dos operaciones no están en el mismo nivel de abstracción, *eliminar comprobaciones concluidas* debe colocarse dentro de las responsabilidades de **RevisarEntrada**, clase que está incluida en la clase agregada **RevisarCuenta**.
2. **Cada responsabilidad debe enunciarse del modo más general posible.** Este lineamiento implica que las responsabilidades generales (tanto atributos como operaciones) deben residir en un nivel elevado de la jerarquía de clases (porque son generales y se aplicarán a todas las subclases).
3. **La información y el comportamiento relacionado con ella deben residir dentro de la misma clase.** Esto logra el principio orientado a objetos llamado *encapsulamiento*. Los datos y los procesos que los manipulan deben empacarse como una unidad cohesiva.
4. **La información sobre una cosa debe localizarse con una sola clase, y no distribuirse a través de muchas.** Una sola clase debe tener la responsabilidad de almacenar y manipular un tipo específico de información. En general, esta responsabilidad no debe ser compartida por varias clases. Si la información está distribuida, es más difícil dar mantenimiento al software y más complicado someterlo a prueba.
5. **Cuando sea apropiado, las responsabilidades deben compartirse entre clases relacionadas.** Hay muchos casos en los que varios objetos relacionados deben tener el mismo comportamiento al mismo tiempo. Por ejemplo, considere un juego de video que deba tener en la pantalla las clases siguientes: **Jugador**, **CuerpodelJugador**, **BrazosdelJugador**, **PiernasdelJugador** y **CabezadelJugador**. Cada una de estas clases tiene sus propios atributos (como **posición**, **orientación**, **color** y **velocidad**) y todas deben actualizarse y desplegarse a medida que el usuario manipula una palanca de juego. Las res-

17 La cohesión es un concepto de diseño que se estudia en el capítulo 8.

18 En tales casos, puede ser necesario dividir la clase en una multiplicidad de ellas o completar subsistemas con el objeto de distribuir la inteligencia de un modo más eficaz.

ponsabilidades *actualizar( )* y *desplegar( )* deben, por tanto, ser compartidas por cada uno de los objetos mencionados. **Jugador** sabe cuando algo ha cambiado y requiere *actualizarse( )*. Colabora con los demás objetos para obtener una nueva posición u orientación, pero cada objeto controla su propio despliegue en la pantalla.

Colaboraciones. Una clase cumple sus responsabilidades en una de dos formas: 1) usa sus propias operaciones para manipular sus propios atributos, con lo que satisface una responsabilidad particular o 2) colabora con otras clases. Wirfs-Brock *et al.* [Wir90] definen las colaboraciones del modo siguiente:

> Las colaboraciones representan solicitudes que hace un cliente a un servidor para cumplir con sus responsabilidades. Una colaboración es la materialización del contrato entre el cliente y el servidor [...] Decimos que un objeto colabora con otro si, para cumplir una responsabilidad, necesita enviar al otro objeto cualesquiera mensajes. Una sola colaboración fluye en una dirección: representa una solicitud del cliente al servidor. Desde el punto de vista del cliente, cada una de sus colaboraciones está asociada con una responsabilidad particular implementada por el servidor.

Las colaboraciones se identifican determinando si una clase puede cumplir cada responsabilidad. Si no es así, entonces necesita interactuar con otra clase. Ésa es una colaboración.

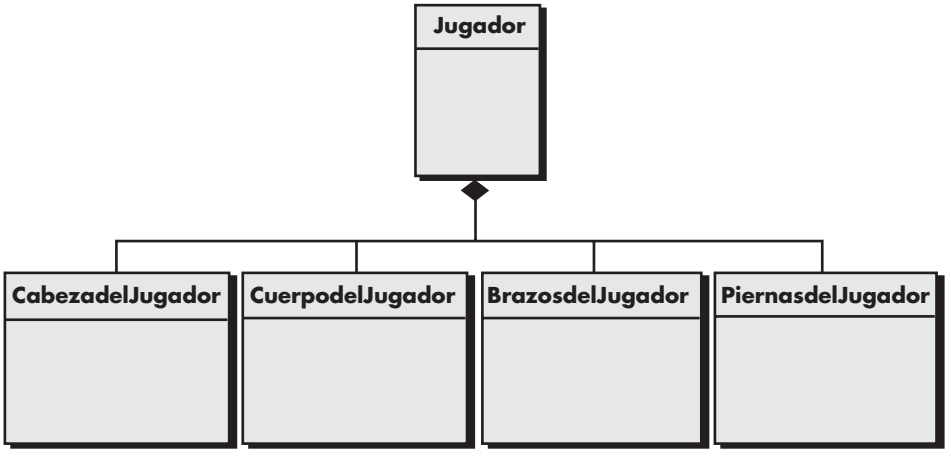
Como ejemplo, considere la función de seguridad de *CasaSegura*. Como parte del procedimiento de activación, el objeto **PaneldeControl** debe determinar si están abiertos algunos sensores. Se define una responsabilidad llamada *determinar-estado-delsensor( )*. Si los sensores están abiertos, **PaneldeControl** debe fijar el atributo **estado** como "no está listo". La información del sensor se adquiere de cada objeto **Sensor**. Por tanto, la responsabilidad *determinar-estado-delsensor( )* se cumple sólo si **PaneldeControl** trabaja en colaboración con **Sensor**.

Para ayudar a identificar a los colaboradores, se estudian tres relaciones generales diferentes entre las clases [Wir90]: 1) la relación *es-parte-de*, 2) la relación *tiene-conocimiento-de* y 3) la relación *depende-de*. En los párrafos siguientes se analizan brevemente cada una de estas tres responsabilidades generales.

Todas las clases que forman parte de una clase agregada se conectan a ésta por medio de una relación *es-parte-de*. Considere las clases definidas por el juego mencionado antes, la clase **CuerpodelJugador** *es-parte-de* **Jugador**, igual que **BrazosdelJugador, PiernasdelJugador** y **CabezadelJugador**. En UML, estas relaciones se representan como el agregado que se ilustra en la figura 6.12.

Cuando una clase debe adquirir información de otra, se establece la relación *tiene-conocimiento-de*. La responsabilidad *determinar-estado-delsensor( )* ya mencionada es un ejemplo de ello.

FIGURA 6.12
Una clase agregada compuesta



La relación *depende-de* significa que dos clases tienen una dependencia que no se determina por *tiene-conocimiento-de* ni por *es-parte-de*. Por ejemplo, **CabezadelJugador** siempre debe estar conectada a **CuerpodelJugador** (a menos que el juego de video sea particularmente violento), pero cada objeto puede existir sin el conocimiento directo del otro. Un atributo del objeto **CabezadelJugador**, llamado **posición-central**, se determina a partir de la posición central de **CuerpodelJugador**. Esta información se obtiene por medio de un tercer objeto, **Jugador**, que la obtiene de **CuerpodelJugador**. Entonces, **CabezadelJugador** *depende-de* **CuerpodelJugador**.

En todos los casos, el nombre de la clase colaboradora se registra en el modelo de tarjeta CRC índice, junto a la responsabilidad que produce la colaboración. Por tanto, la tarjeta índice contiene una lista de responsabilidades y las colaboraciones correspondientes que hacen que se cumplan (véase la figura 6.11).

Cuando se ha desarrollado un modelo CRC completo, los participantes lo revisan con el empleo del enfoque siguiente [Amb95]:

1. Se da a todos los participantes que intervienen en la revisión (del modelo CRC) un subconjunto del modelo de tarjetas índice CRC. Deben separarse aquellas que colaboran (de modo que ningún revisor deba tener dos tarjetas que colaboren).
2. Todos los escenarios de casos de uso (y los diagramas correspondientes) deben organizarse en dos categorías.
3. El líder de la revisión lee el caso de uso en forma deliberada. Cuando llega a un objeto con nombre, entrega una ficha a la persona que tenga la tarjeta índice de la clase correspondiente. Por ejemplo, un caso de uso de *CasaSegura* contiene la narración siguiente:

   El propietario observa el panel de control de *CasaSegura* para determinar si el sistema está listo para recibir una entrada. Si el sistema no está listo, el propietario debe cerrar físicamente las puertas y ventanas de modo que el indicador *listo* aparezca [un indicador *no está listo* implica que un sensor se encuentra abierto, es decir, que una puerta o ventana está abierta].

   Cuando en la narración del caso de uso el líder de la revisión llega a "panel de control", entrega la ficha a la persona que tiene la tarjeta índice **PaneldeControl**. La frase "implica que un sensor está abierto" requiere que la tarjeta índice contenga una responsabilidad que validará esta implicación (esto lo logra la responsabilidad *determinar-estado-delsensor( )*. Junto a la responsabilidad, en la tarjeta índice se encuentra el **Sensor** colaborador. Entonces, la ficha pasa al objeto **Sensor**.
4. Cuando se pasa la ficha, se pide al poseedor de la tarjeta **Sensor** que describa las responsabilidades anotadas en la tarjeta. El grupo determina si una (o más) de las responsabilidades satisfacen el requerimiento del caso de uso.
5. Si las responsabilidades y colaboraciones anotadas en las tarjetas índice no se acomodan al caso de uso, éstas se modifican. Lo anterior tal vez incluya la definición de nuevas clases (y las tarjetas CRC índice correspondientes) o la especificación en las tarjetas existentes de responsabilidades o colaboraciones nuevas o revisadas.

Este modo de operar continúa hasta terminar el caso de uso. Cuando se han revisado todos los casos de uso, continúa el modelado de los requerimientos.

6.5.5 Asociaciones y dependencias

PUNTO CLAVE

Una asociación define una relación entre clases. La multiplicidad define cuántas de una clase se relacionan con cuántas de otra clase.

En muchos casos, dos clases de análisis se relacionan de cierto modo con otra, en forma muy parecida a como dos objetos de datos se relacionan entre sí (véase la sección 6.4.3). En UML, estas relaciones se llaman *asociaciones*. Al consultar la figura 6.10, la clase **Plano** se define con la identificación de un conjunto de asociaciones entre **Plano** y otras dos clases, **Cámara** y **Pa-**

CasaSegura

Modelos CRC

La escena: Cubículo de Ed cuando comienza el modelado de los requerimientos.

Participantes: Vinod y Ed, miembros del equipo de ingeniería de software de *CasaSegura*.

La conversación:

[Vinod ha decidido enseñar a Ed con un ejemplo cómo desarrollar las tarjetas CRC.]

Vinod: Mientras tú trabajabas en la vigilancia y Jamie lo hacía con la seguridad, yo estaba en la función de administración del hogar.

Ed: ¿Cuál es el estado de eso? Mercadotecnia cambia lo que quiere a cada rato.

Vinod: Aquí está la primera versión de caso de uso para toda la función... la mejoramos un poco, pero debe darte el panorama general...

Caso de uso: Función de administración de *CasaSegura*.

Narración: Queremos usar la interfaz de administración del hogar en una PC o en una conexión de internet para controlar los dispositivos electrónicos que tengan controladores de interfaz inalámbrica. El sistema debe permitir encender y apagar focos específicos, controlar aparatos conectados a una interfaz inalámbrica y fijar el sistema de calefacción y aire acondicionado a la temperatura que desee. Para hacer esto, quiero seleccionar los aparatos en el plano de la casa. Cada equipo debe estar identificado en el plano. Como característica opcional, quiero controlar todos los equipos audiovisuales: sonido, televisión, DVD, grabadoras digitales, etcétera.

Con una sola selección, quiero preparar toda la casa para distintas situaciones. Una es *casa*, otra es *salir*, la tercera es *viaje nocturno* y la cuarta es *viaje largo*. Todas estas situaciones tienen especificaciones que se aplicarán a todos los equipos. En los estados de *viaje nocturno* y *viaje largo*, el sistema debe encender y apagar focos en momentos elegidos al azar (para que parezca que hay alguien en casa) y controlar el sistema de calefacción y aire acondicionado. Debo poder hacer esta preparación por internet, con la protección de claves adecuadas...

Ed: ¿El personal de hardware ya tiene listas todas las interfaces inalámbricas?

Vinod (sonríe): Están trabajando en eso; dicen que no hay problema. De cualquier forma, obtuve muchas clases para la administración del hogar y podemos usar una como ejemplo. Tomemos la clase **InterfazdeAdministracióndelHogar**.

Ed: Bien... entonces, las responsabilidades son... los atributos y operaciones para la clase, y las colaboraciones son las clases que indican las responsabilidades.

Vinod: Pensé que no habías entendido el concepto CRC.

Ed: Un poco, quizá, pero continúa.

Vinod: Aquí está mi definición de la clase **InterfazdeAdministracióndelHogar**.

Atributos:

PaneldeOpciones: contiene información sobre los botones que permiten al usuario seleccionar funcionalidad.

PaneldeSituación: contiene información acerca de los botones que permiten que el usuario seleccione la situación.

Plano: igual que el objeto de vigilancia, pero éste muestra los equipos.

ÍconosdeAparatos: informa sobre los íconos que representan luces, aparatos, calefacción y aire acondicionado, etcétera.

PaneldeAparatos: simula el panel de control de un aparato o equipo; permite controlarlo.

Operaciones:

DesplegarControl( ), SeleccionarControl( ), DesplegarSituación( ), SeleccionarSituación( ), AccederaPlano( ), SeleccionarÍconodeEquipo( ), DesplegarPaneldeEquipo( ), AccederaPaneldeEquipo( ),...

Clase: InterfazdeAdministracióndelHogar

Responsabilidad	**Colaborador**
DesplegarControl( )	**PaneldeOpciones** (clase)
SeleccionarControl( )	**PaneldeOpciones** (clase)
DesplegarSituación( )	**PaneldeSituación** (clase)
SeleccionarSituación( )	**PaneldeSituación** (clase)
AccederaPlano( )	**Plano** (clase) . . .

. . .

Ed: De modo que cuando se invoque a operación *AccederaPlano( )*, colabora con el objeto **Plano** de igual manera que el que desarrollamos para vigilancia. Espera, aquí tengo su descripción (ven la figura 6.10).

Vinod: Exactamente. Y si quisiéramos revisar todo el modelo de la clase, podríamos comenzar con esta tarjeta índice, luego iríamos a la del colaborador y de ahí a una de los colaboradores del colaborador, y así sucesivamente.

Ed: Buena forma de encontrar omisiones o errores.

Vinod: Sí.

red. La clase **Pared** está asociada con tres clases que permiten que se construya ésta, y que son **SegmentodePared, Ventana** y **Puerta**.

En ciertos casos, una asociación puede definirse con más detalle si se indica *multiplicidad*. En relación con la figura 6.10, un objeto **Pared** se construye a partir de uno o más objetos **SegmentodePared**. Además, el objeto **Pared** puede contener 0 o más objetos **Ventana** y 0 o más objetos **Puerta**. Estas restricciones de multiplicidad se ilustran en la figura 6.13, donde "uno o

FIGURA 6.13
Multiplicidad

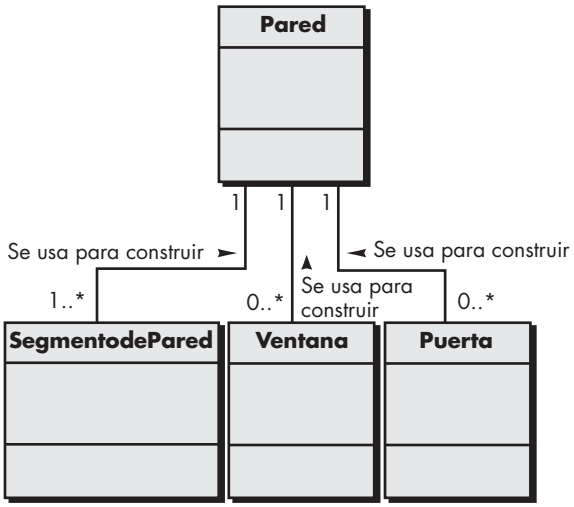
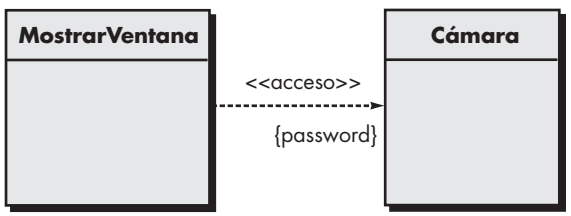


FIGURA 6.14
Dependencias



más" se representa con 1...*, y para "0 o más" se usa 0...*. En LMU, el asterisco indica una frontera ilimitada en ese rango.[19]

¿Qué es un estereotipo?

Sucede con frecuencia que entre dos clases de análisis existe una relación cliente-servidor. En tales casos, una clase cliente depende de algún modo de la clase servidor, y se establece una *relación de dependencia*. Las dependencias están definidas por un estereotipo. Un *estereotipo* es un "mecanismo extensible" [Arl02] dentro del UML que permite definir un elemento especial de modelado con semántica y especialización determinadas. En UML, los estereotipos se representan entre paréntesis dobles angulares (por ejemplo, <<estereotipo>>).

PUNTO CLAVE
Un paquete se utiliza para ensamblar un conjunto de clases relacionadas

Como ilustración de una dependencia simple dentro del sistema de vigilancia *CasaSegura*, un objeto **Cámara** (la clase servidora, en este caso) proporciona una imagen a un objeto **MostrarVentana** (la clase cliente). La relación entre estos dos objetos no es una asociación simple sino de dependencia. En el caso de uso escrito para la vigilancia (que no se presenta aquí), debe darse una clave especial a fin de observar ubicaciones específicas de las cámaras. Una forma de lograr esto es hacer que **Cámara** pida un password y luego asegure el permiso a **MostrarVentana** para que presente el video. Esto se representa en la figura 6.14, donde <<acceso>> implica que el uso de la salida de cámara se controla con una clave especial.

6.5.6 Paquetes de análisis

Una parte importante del modelado del análisis es la categorización. Es decir, se clasifican distintos elementos del modelo de análisis (por ejemplo, casos de uso y clases de análisis) de ma-

19 Como parte de una asociación, pueden indicarse otras relaciones de multiplicidad: una a una, una a muchas, muchas a muchas, una a un rango específico con límites inferior y superior, y otras.

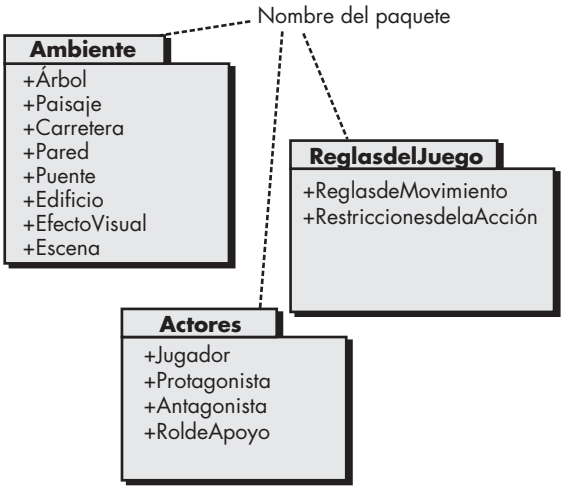


FIGURA 6.15
Paquetes

nera que se agrupen en un paquete —llamado *paquete de análisis*— al que se da un nombre representativo.

Para ilustrar el uso de los paquetes de análisis, considere el juego de video que se mencionó antes. A medida que se desarrolla el modelo de análisis para el juego de video, se obtiene un gran número de clases. Algunas se centran en el ambiente del juego —las escenas visuales que el usuario ve cuando lo usa—. En esta categoría quedan clases tales como **Árbol, Paisaje, Carretera, Pared, Puente, Edificio** y **EfectoVisual**. Otras se centran en los caracteres dentro del juego y describen sus características físicas, acciones y restricciones. Pueden definirse clases como **Jugador** (ya descrita), **Protagonista**, **Antagonista** y **RolesdeApoyo**. Otras más describen las reglas del juego —cómo se desplaza un jugador por el ambiente—. Candidatas para esto son clases como **ReglasdeMovimiento** y **RestriccionesdelaAcción**. Pueden existir muchas otras categorías. Estas clases se agrupan en los paquetes de análisis que se observan en la figura 6.15.

El signo *más* (suma) que precede al nombre de la clase de análisis en cada paquete, indica que las clases tienen visibilidad pública, por lo que son accesibles desde otros paquetes. Aunque no se aprecia en la figura, hay otros símbolos que preceden a un elemento dentro de un paquete. El signo *menos* (resta) indica que un elemento queda oculto desde todos los demás paquetes. Y el símbolo # señala que un elemento es accesible sólo para los paquetes contenidos dentro de un paquete dado.

6.6 RESUMEN

El objetivo del modelado de los requerimientos es crear varias representaciones que describan lo que necesita el cliente, establecer una base para generar un diseño de software y definir un conjunto de requerimientos que puedan ser validados una vez construido el software. El modelo de requerimientos cruza la brecha entre la representación del sistema que describe el sistema en su conjunto y la funcionalidad del negocio, y un diseño de software que describe la arquitectura de la aplicación del software, la interfaz de usuario y la estructura de componentes.

Los modelos basados en el escenario ilustran los requerimientos del software desde el punto de vista del usuario. El caso de uso —descripción, hecha con una narración o un formato, de una interacción entre un actor y el software— es el principal elemento del modelado. El caso de uso se obtiene durante la indagación de los requerimientos y define las etapas clave de una función o interacción específica. El grado de formalidad del caso de uso y su nivel de detalle varía, pero

el resultado final da las entradas necesarias a todas las demás actividades del modelado. Los escenarios también pueden ser descritos con el uso de un diagrama de actividades —representación gráfica parecida a un diagrama de flujo que ilustra el flujo del procesamiento dentro de un escenario específico—. Los diagramas de canal (swimlane) ilustran la forma en la que se asigna el flujo del procesamiento a distintos actores o clases.

El modelado de datos se utiliza para describir el espacio de información que será construido o manipulado por el software. El modelado de datos comienza con la representación de los objetos de datos —información compuesta que debe ser entendida por el software—. Se identifican los atributos de cada objeto de datos y se describen las relaciones entre estos objetos.

El modelado basado en clases utiliza información obtenida de los elementos del modelado basado en el escenario y en datos, para identificar las clases de análisis. Se emplea un análisis gramatical para obtener candidatas a clase, atributos y operaciones, a partir de narraciones basadas en texto. Se definen criterios para definir una clase. Para definir las relaciones entre clases, se emplean tarjetas índice clase-responsabilidad-colaborador. Además, se aplican varios elementos de la notación UML para definir jerarquías, relaciones, asociaciones, agregaciones y dependencias entre clases. Se emplean paquetes de análisis para clasificar y agrupar clases, de manera que sean más manejables en sistemas grandes.

Problemas y puntos por evaluar

6.1. ¿Es posible comenzar a codificar de inmediato después de haber creado un modelo de análisis? Explique su respuesta y luego defienda el punto de vista contrario.

6.2. Una regla práctica del análisis es que el modelo "debe centrarse en los requerimientos visibles dentro del dominio del problema o negocio". ¿Qué tipos de requerimientos *no* son visibles en dichos dominios? Dé algunos ejemplos.

6.3. ¿Cuál es el propósito del análisis del dominio? ¿Cómo se relaciona con el concepto de patrones de requerimientos?

6.4. ¿Es posible desarrollar un modelo de análisis eficaz sin desarrollar los cuatro elementos que aparecen en la figura 6.3? Explique su respuesta.

6.5. Se pide al lector que construya uno de los siguientes sistemas:

a) Sistema de inscripción a la universidad basado en red.
b) Sistema de procesamiento de órdenes basado en web para una tienda de computadoras.
c) Sistema de facturación simple para un negocio pequeño.
d) Libro de cocina basado en internet, construido en un horno eléctrico o de microondas.

Seleccione el sistema que le interese y desarrolle un diagrama entidad-relación que describa los objetos de datos, relaciones y atributos.

6.6. El departamento de obras públicas de una gran ciudad ha decidido desarrollar un sistema de seguimiento y reparación de baches, basado en web (SSRB).

Cuando se reportan los baches, se registran en "sistema de reparación del departamento de obras públicas" y se les asigna un número de identificación, almacenado según la calle, tamaño (en una escala de 1 a 10), ubicación (en medio, cuneta, etc.), distrito (se determina con la dirección en la calle) y prioridad de reparación (determinada por el tamaño del bache). Los datos de la orden de trabajo se asocian con cada bache e incluyen su ubicación y tamaño, número de identificación del equipo de reparación, número de personas en dicho equipo, equipo asignado, horas dedicadas a la reparación, estado del bache (trabajo en proceso, reparado, reparación temporal, no reparado), cantidad de material de relleno utilizado y costo de la reparación (calculado a partir de las horas dedicadas, número de personas, materiales y equipo empleado). Por último, se crea un archivo de daños para mantener la información sobre daños reportados debido al bache, y se incluye el nombre y dirección del ciudadano, número telefónico, tipo de daño y cantidad de dinero por el daño. El SSRB es un sistema en línea, todas las solicitudes se harán en forma interactiva.

a) Dibuje un diagrama UML para el caso de uso del sistema SSRB. Tendrá que hacer algunas suposiciones sobre la manera en la que un usuario interactúa con el sistema.
b) Desarrolle un modelo de clase para el sistema SSRB.

6.7. Escriba un caso de uso basado en formato para el sistema de administración del hogar *CasaSegura* descrito de manera informal en el recuadro de la sección 6.5.4.

6.8. Desarrolle un conjunto completo de tarjetas índice de modelo CRC, sobre el producto o sistema que elija como parte del problema 6.5.

6.9. Revise con sus compañeros las tarjetas índice CRC. ¿Cuántas clases, responsabilidades y colaboradores adicionales fueron agregados como consecuencia de la revisión?

6.10. ¿Qué es y cómo se usa un paquete de análisis?

Lecturas adicionales y fuentes de información

Los casos de uso son el fundamento de todos los enfoques del modelado de los requerimientos. El tema se analiza con amplitud en Rosenberg y Stephens (*Use Case Driven Object Modeling with UML: Theory and Practice*, Apress, 2007), Denny (*Succeeding with Use Cases: Working Smart to Deliver Quality*, Addison-Wesley, 2005), Alexander y Maiden (eds.) (*Scenarios, Stories, Use Cases: Through the Systems Development Life-Cycle*, Wiley, 2004), Bittner y Spence (*Use Case Modeling,* Addison-Wesley, 2002), Cockburn [Coc01b] y en otras referencias mencionadas en los capítulos 5 y 6.

El modelado de datos constituye un método útil para examinar el espacio de información. Los libros de Hoberman [Hob06] y Simsion y Witt [Sim05] hacen tratamientos razonablemente amplios. Además, Allen y Terry (*Beginning Relational Data Modeling*, 2a. ed., Apress, 2005), Allen (*Data Modeling for Everyone*, Word Press, 2002), Teorey *et al.* (*Database Modeling and Design: Logical Design*, 4a. ed., Morgan Kaufmann, 2005) y Carlis y Maguire (*Mastering Data Modeling*, Addison-Wesley, 2000) presentan métodos de aprendizaje detallados para crear modelos de datos de calidad industrial. Un libro interesante escrito por Hay (*Data Modeling Patterns*, Dorset House, 1995) presenta patrones comunes de modelos de datos que se encuentran en muchos negocios diferentes.

Análisis de técnicas de modelado UML que pueden aplicarse tanto para el análisis como para el diseño se encuentran en O'Docherty (*Object-Oriented Analysis and Design: Understanding System Development with UML 2.0*, Wiley, 2005), Arlow y Neustadt (*UML, 2 and the Unified Process*, 2a. ed., Addison-Wesley, 2005), Roques (*UML in Practice*, Wiley, 2004), Dennis *et al.* (*Systems Analysis and Design with UML Version 2.0*, Wiley, 2004), Larman (*Applying UML and Patterns*, 2a. ed., Prentice-Hall, 2001) y Rosenberg y Scott (*Use Case Driven Object Modeling with UML*, Addison-Wesley, 1999).

En internet existe una amplia variedad de fuentes de información sobre el modelado de requerimientos. En el sitio web del libro, en **www.mhhe.com/engcs/comsci/pressman/professioal/olc/ser.htm**, se halla una lista actualizada de referencias en web que son relevantes para el modelado del análisis.

CAPÍTULO

7 MODELADO DE LOS REQUERIMIENTOS: FLUJO, COMPORTAMIENTO, PATRONES Y WEBAPPS

CONCEPTOS CLAVE

diagramas de secuencia 168
especificación del proceso. . . 163
modelado de la navegación 180
modelo de comportamiento 165
modelo de configuración. . . . 179
modelo del contenido. 176
modelo de flujo de control . . 162
modelo de flujo de datos . . . 159
modelo de interacción 177
modelo funcional. 178
patrones de análisis 169
webapps 174

Después de estudiar en el capítulo 6 los casos de uso, modelado de datos y modelos basados en clase, es razonable preguntar: "¿no son suficientes representaciones del modelado de los requerimientos?"

La única respuesta razonable es: "depende".

Para ciertos tipos de software, el caso de uso puede ser la única representación para modelar los requerimientos que se necesite. Para otros, se escoge un enfoque orientado a objetos y se desarrollan modelos basados en clase. Pero en otras situaciones, los requerimientos de las aplicaciones complejas demandan el estudio de la manera como se transforman los objetos de datos cuando se mueven a través del sistema; cómo se comporta una aplicación a consecuencia de eventos externos; si el conocimiento del dominio existente puede adaptarse al problema en cuestión; o, en el caso de sistemas y aplicaciones basados en web, cómo unificar el contenido y la funcionalidad para dar al usuario final la capacidad de navegar con éxito por una *webapp* a fin de lograr sus objetivos.

7.1 REQUERIMIENTOS QUE MODELAN LAS ESTRATEGIAS

Un punto de vista del modelado de los requerimientos, llamada *análisis estructurado*, considera como entidades separadas los datos y los procesos que los transforman. Los objetos de datos se modelan en una forma que define sus atributos y relaciones. Los procesos que manipulan objetos de datos se modelan de una forma que muestra cómo transforman los datos cuando los

UNA MIRADA RÁPIDA

¿Qué es? El modelo de requerimientos tiene muchas dimensiones diferentes. En este capítulo, el lector aprenderá acerca de modelos orientados al flujo, de modelos de comportamiento y de las consideraciones especiales del análisis de requerimientos que entran en juego cuando se desarrollan *webapps*. Cada una de estas representaciones de modelado complementa los casos de uso, modelos de datos y modelos basados en clases que se estudiaron en el capítulo 6.

¿Quién lo hace? Un ingeniero de software (a veces llamado analista) construye el modelo con el uso de los requerimientos recabados entre varios participantes.

¿Por qué es importante? La perspectiva de los requerimientos del software crece en proporción directa al número de dimensiones distintas del modelado de los requerimientos. Aunque quizá no se tenga el tiempo, los recursos o la inclinación para desarrollar cada representación sugerida en este capítulo y en el anterior, debe reconocerse que cada enfoque diferente de modelado proporciona una forma distinta de ver el problema. En consecuencia, el lector (y otros participantes) estará mejor preparado para evaluar si ha especificado en forma apropiada aquello que debe lograrse.

¿Cuáles son los pasos? El modelado orientado al flujo da una indicación de la forma en la que las funciones de procesamiento transforman los objetos de datos. El modelado del comportamiento ilustra los estados del sistema y sus clases, así como el efecto que tienen los eventos sobre dichos estados. El modelado basado en patrones utiliza el conocimiento del dominio existente para facilitar el análisis de los requerimientos. Los modelos de requerimientos con *webapps* están adaptados especialmente para representar requerimientos relacionados con contenido, interacción, función y configuración.

¿Cuál es el producto final? Para el modelado de los requerimientos, es posible escoger una gran variedad de formas basadas en texto y diagramas. Cada una de estas representaciones da una perspectiva de uno o más de los elementos del modelo.

¿Cómo me aseguro de que lo hice bien? Debe revisarse si los productos del trabajo del modelado de los requerimientos son correctos, completos y congruentes. Deben reflejar las necesidades de todos los participantes y establecer los fundamentos desde los que se llevará a cabo el diseño.

objetos de datos fluyen por el sistema. Un segundo enfoque del modelado de análisis, llamado *análisis orientado a objetos*, se centra en la definición de clases y en el modo en el que colaboran una con otra para cumplir con los requerimientos del cliente.

Aunque el modelo de análisis que se propone en este libro combina características de ambos enfoques, es frecuente que los equipos del software elijan uno de ellos y excluyan las representaciones del otro. La pregunta no es cuál es mejor, sino qué combinación de representaciones dará a los participantes el mejor modelo de los requerimientos del software y cuál será el mejor puente para cruzar la brecha hacia el diseño del software.

7.2 MODELADO ORIENTADO AL FLUJO

Aunque algunos ingenieros de software perciben el modelado orientado al flujo como una técnica obsoleta, sigue siendo una de las notaciones más usadas actualmente para hacer el análisis de los requerimientos.[1] Si bien el *diagrama de flujo de datos* (DFD) y la información relacionada no son una parte formal del UML, se utilizan para complementar los diagramas de éste y amplían la perspectiva de los requerimientos y del flujo del sistema.

Algunas personas afirman que los DFD son obsoletos y que no hay lugar para ellos en la práctica moderna. Ese punto de vista excluye un modo potencialmente útil de representación en el nivel del análisis. Si ayuda, use DFD.

El DFD adopta un punto de vista del tipo entrada-proceso-salida para el sistema. Es decir, los objetos de datos entran al sistema, son transformados por elementos de procesamiento y los objetos de datos que resultan de ello salen del software. Los objetos de datos se representan con flechas con leyendas y las transformaciones, con círculos (también llamados burbujas). El DFD se presenta en forma jerárquica. Es decir, el primer modelo de flujo de datos (en ocasiones llamado DFD de nivel 0 o *diagrama de contexto*) representa al sistema como un todo. Los diagramas posteriores de flujo de datos mejoran el diagrama de contexto y dan cada vez más detalles en los niveles siguientes.

7.2.1 Creación de un modelo de flujo de datos

Cita:

"El propósito de los diagramas de flujo de datos es proveer un puente semántico entre los usuarios y los desarrolladores de sistemas."

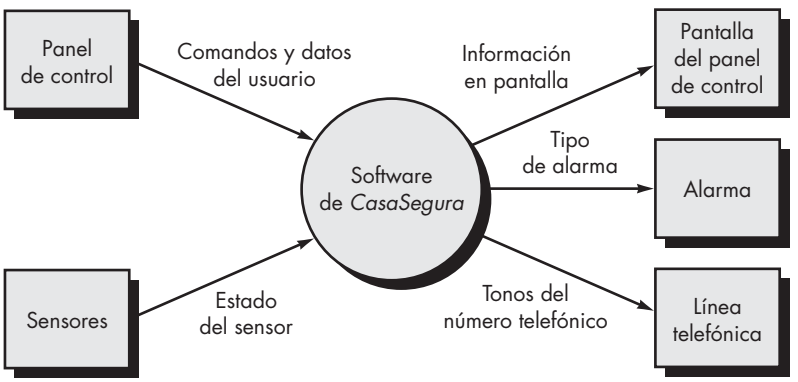
Kenneth Kozar

El diagrama de flujo de datos permite desarrollar modelos del dominio de la información y del dominio funcional. A medida que el DFD se mejora con mayores niveles de detalle, se efectúa la descomposición funcional implícita del sistema. Al mismo tiempo, la mejora del DFD da como resultado el refinamiento de los datos conforme avanzan por los procesos que constituyen la aplicación.

Unos cuantos lineamientos sencillos ayudan muchísimo durante la elaboración del diagrama de flujo de los datos: 1) el nivel 0 del diagrama debe ilustrar el software o sistema como una sola

FIGURA 7.1

DFD en el nivel de contexto para la función de seguridad de *CasaSegura*



1 El modelado del flujo de datos es una actividad fundamental del *análisis estructurado*.

burbuja; 2) debe anotarse con cuidado las entradas y salidas principales; 3) la mejora debe comenzar por aislar procesos candidatos, objetos de datos y almacenamiento de éstos, para representarlos en el siguiente nivel; 4) todas las flechas y burbujas deben etiquetarse con nombres significativos; 5) de un nivel a otro, debe mantenerse la *continuidad del flujo de información*,[2] y 6) debe mejorarse una burbuja a la vez. Existe la tendencia natural a complicar innecesariamente el diagrama de flujo de los datos. Esto sucede cuando se trata de ilustrar demasiados detalles en una etapa muy temprana o representar aspectos de procedimiento del software en lugar del flujo de la información.

PUNTO CLAVE

Conforme se mejora cada nivel del DFD, debe mantenerse la continuidad del flujo de la información. Esto significa que las entradas y salidas en cierto nivel deben ser las mismas en un nivel mejorado.

Para ilustrar el uso del DFD y la notación relacionada, consideremos de nuevo la función de seguridad de *CasaSegura*. En la figura 7.1 se muestra un DFD de nivel 0 para dicha función. Las *entidades externas* principales (cuadrados) producen información para uso del sistema y consumen información generada por éste. Las flechas con leyendas representan objetos de datos o jerarquías de éstos. Por ejemplo, los **comandos** y **datos del usuario** agrupan todos los comandos de configuración, todos los comandos de activación/desactivación, todas las diferentes interacciones y todos los datos que se introducen para calificar o expandir un comando.

Ahora debe expandirse el DFD de nivel 0 a un modelo de flujo de datos de nivel 1. Pero, ¿cómo hacerlo? Según el enfoque sugerido en el capítulo 6, debe aplicarse un "análisis gramatical" [Abb83] a la narración del caso de uso que describe la burbuja en el nivel del contexto. Es decir, se aíslan todos los sustantivos (y frases sustantivadas) y verbos (y frases verbales) en la narración del procesamiento de *CasaSegura* obtenida durante la primera reunión realizada para recabar los requerimientos. Recordemos el análisis gramatical del texto que narra el procesamiento presentado en la sección 6.5.1:

CONSEJO

El análisis gramatical no es a prueba de todo, pero da un impulso excelente para arrancar si se tienen dificultades para definir objetos de datos y las transformaciones que operan sobre ellos.

> La función de seguridad CasaSegura *permite* que el propietario *configure* el sistema de seguridad cuando se *instala*, *vigila* todos los sensores *conectados* al sistema de seguridad e *interactúa* con el propietario a través de internet, una PC o un panel de control.
>
> Durante la instalación, la PC de CasaSegura se utiliza para *programar* y *configurar* el sistema. Se asigna a cada sensor un número y un tipo, se programa una clave maestra para *activar* y *desactivar* el sistema, y se *introducen* números telefónicos para *marcar* cuando ocurre un evento de sensor.
>
> Cuando se *reconoce* un evento de sensor, el software *invoca* una alarma audible instalada en el sistema. Después de un tiempo de retraso que especifica el propietario durante las actividades de configuración del sistema, el software marca un número telefónico de un servicio de monitoreo, *proporciona* información acerca de la ubicación y *reporta* la naturaleza del evento detectado. El número telefónico *vuelve a marcarse* cada 20 segundos hasta que se *obtiene* la conexión telefónica.
>
> El propietario *recibe* información de seguridad a través de un panel de control, la PC o un navegador, lo que en conjunto se llama interfaz. La interfaz *despliega* en el panel de control, en la PC o en la ventana del navegador mensajes de aviso e información del estado del sistema. La interacción del propietario tiene la siguiente forma...

CONSEJO

Asegúrese de que la narración del procesamiento que se va a analizar gramaticalmente está escrita con el mismo nivel de abstracción.

En relación con el análisis gramatical, los verbos son los procesos de *CasaSegura* y se representarán como burbujas en un DFD posterior. Los sustantivos son entidades externas (cuadros), datos u objetos de control (flechas) o almacenamiento de datos (líneas dobles). De lo estudiado en el capítulo 6 se recuerda que los sustantivos y verbos se asocian entre sí (por ejemplo, a cada sensor se asigna un número y tipo; entonces, **número** y **tipo** son atributos del objeto de datos **sensor**). De modo que al realizar un análisis gramatical de la narración de procesamiento en cualquier nivel del DFD, se genera mucha información útil sobre la manera de proceder para la mejora del nivel siguiente. En la figura 7.2 se presenta un DFD de nivel 1 con el empleo de esta información. El proceso en el nivel de contexto que se ilustra en la figura 7.1 ha sido expandido

2 Es decir, los objetos de datos que entran al sistema o a cualquier transformación en cierto nivel deben ser los mismos objetos de datos (o sus partes constitutivas) que entran a la transformación en un nivel mejorado.

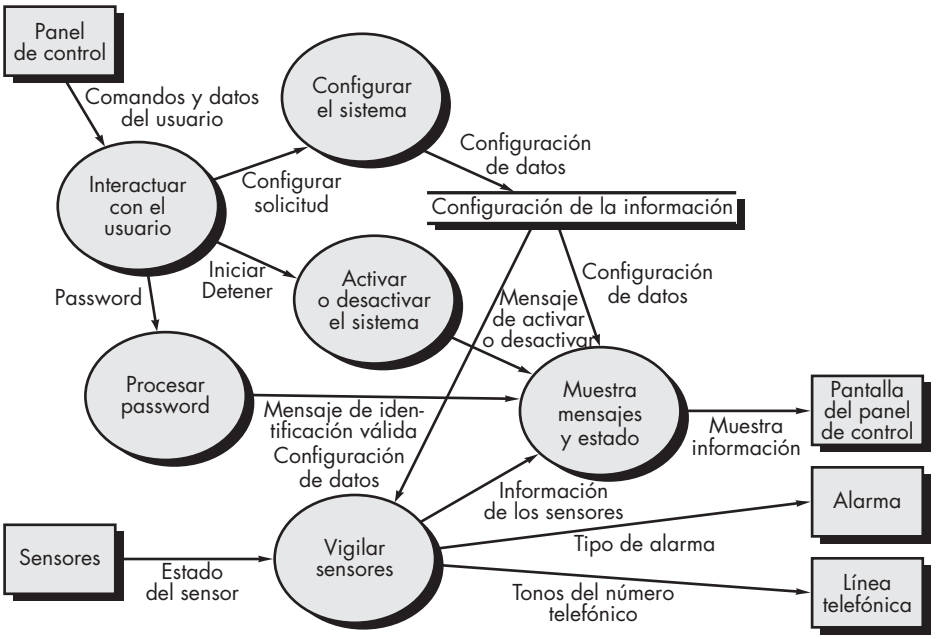


FIGURA 7.2
DFD de nivel 1 para la función de seguridad de *CasaSegura*

a seis procesos derivados del estudio del análisis gramatical. De manera similar, el flujo de información entre procesos del nivel 1 ha surgido de dicho análisis. Además, entre los niveles 0 y 1 se mantiene la continuidad del flujo de información.

Los procesos representados en el nivel 1 del DFD pueden mejorarse más hacia niveles inferiores. Por ejemplo, el proceso *vigilar sensores* se mejora en el DFD de nivel 2, como se aprecia en la figura 7.3. De nuevo, observe que entre los niveles se ha mantenido la continuidad del flujo de información.

La mejoría de los DFD continúa hasta que cada burbuja realiza una función simple. Es decir, hasta que el proceso representado por la burbuja ejecuta una función que se implementaría fácilmente como componente de un programa. En el capítulo 8 se estudia un concepto llamado

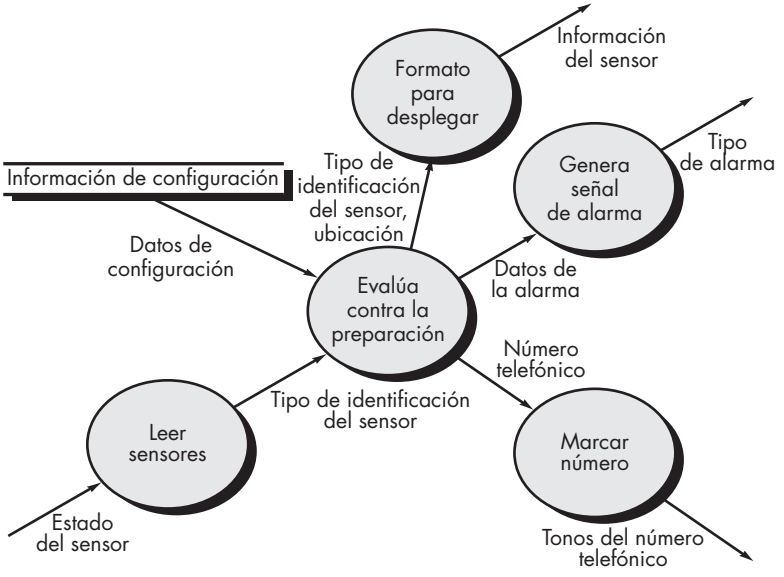


FIGURA 7.3
DFD de nivel 2 que mejora el proceso *vigilar sensores*

cohesión, que se utiliza para evaluar el objeto del procesamiento de una función dada. Por ahora, se trata de mejorar los DFD hasta que cada burbuja tenga "un solo pensamiento".

7.2.2 Creación de un modelo de flujo de control

Para ciertos tipos de aplicaciones, el modelo de datos y el diagrama de flujo de datos es todo lo que se necesita para obtener una visión significativa de los requerimientos del software. Sin embargo, como ya se dijo, un gran número de aplicaciones son "motivadas" por eventos y no por datos, producen información de control en lugar de reportes o pantallas, y procesan información con mucha atención en el tiempo y el desempeño. Tales aplicaciones requieren el uso del *modelado del flujo de control*, además de modelar el flujo de datos.

Se dijo que un evento o aspecto del control se implementa como valor booleano (por ejemplo, verdadero o falso, encendido o apagado, 1 o 0) o como una lista discreta de condiciones (vacío, bloqueado, lleno, etc.). Se sugieren los lineamientos siguientes para seleccionar eventos candidatos potenciales:

¿Cómo seleccionar los eventos potenciales para un diagrama de flujo de control, de estado o CSPEC?

- Enlistar todos los sensores que son "leídos" por el software.
- Enlistar todas las condiciones de interrupción.
- Enlistar todos los "interruptores" que son activados por un operador.
- Enlistar todas las condiciones de los datos.
- Revisar todos los "aspectos de control" como posibles entradas o salidas de especificación del control, según el análisis gramatical de sustantivos y verbos que se aplicó a la narración del procesamiento.
- Describir el comportamiento de un sistema con la identificación de sus estados, identificar cómo se llega a cada estado y definir las transiciones entre estados.
- Centrarse en las posibles omisiones, error muy común al especificar el control; por ejemplo, se debe preguntar: "¿hay otro modo de llegar a este estado o de salir de él?"

Entre los muchos eventos y aspectos del control que forman parte del software de *CasaSegura*, se encuentran **evento de sensor** (por ejemplo, un sensor se descompone), **bandera de cambio** (señal para que la pantalla cambie) e **interruptor iniciar/detener** (señal para encender o apagar el sistema).

7.2.3 La especificación de control

Una *especificación de control* (CSPEC) representa de dos maneras distintas el comportamiento del sistema (en el nivel desde el que se hizo referencia a él).[3] La CSPEC contiene un diagrama de estado que es una especificación secuencial del comportamiento. También puede contener una tabla de activación del programa, especificación combinatoria del comportamiento.

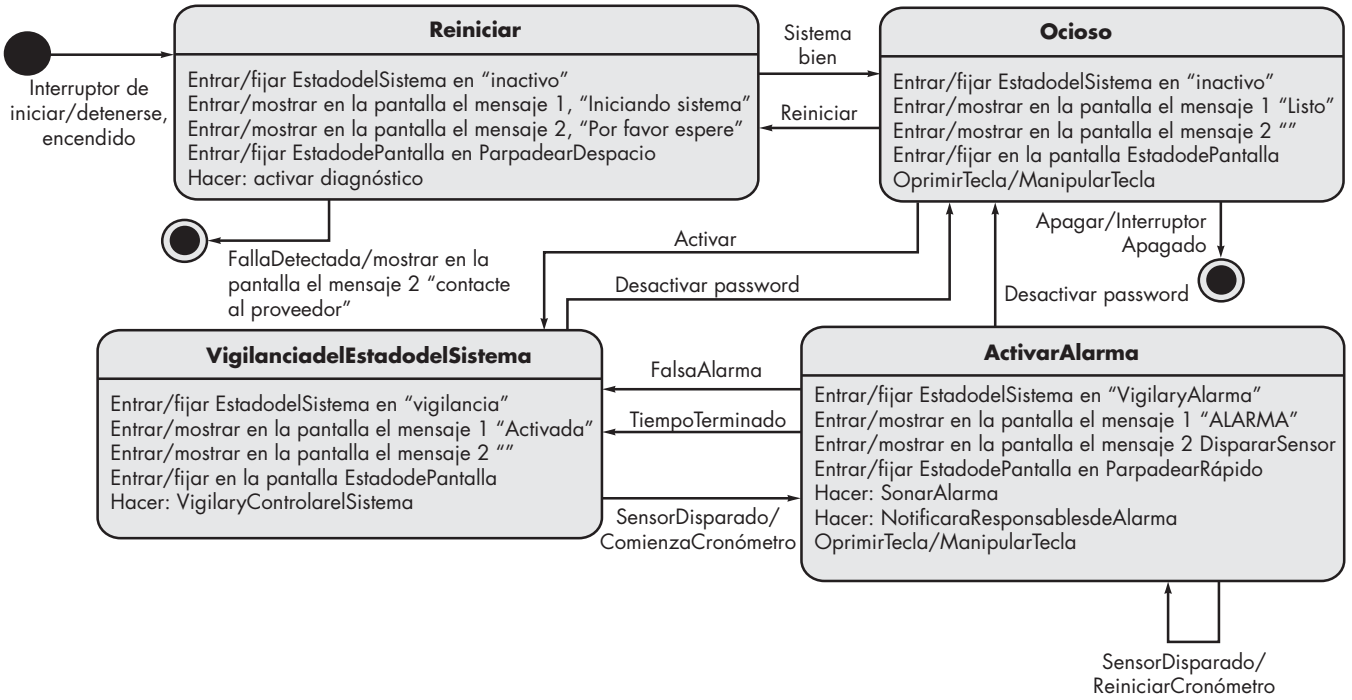
La figura 7.4 ilustra un diagrama de estado preliminar[4] para el nivel 1 del modelo de flujo de control para *CasaSegura*. El diagrama indica cómo responde el sistema a eventos conforme pasa por los cuatro estados definidos en este nivel. Con la revisión del diagrama de estado se determina el comportamiento del sistema, y, lo que es más importante, se investiga si existen "agujeros" en el comportamiento especificado.

Por ejemplo, el diagrama de estado (véase la figura 7.4) indica que las transiciones del estado **Ocioso** ocurren si el sistema se reinicia, se activa o se apaga. Si el sistema se activa (por ejem-

3 En la sección 7.3 se presenta notación adicional de modelado por comportamiento.

4 La notación del diagrama de estado que se emplea aquí sigue la del UML. En el análisis estructurado se dispone de un "diagrama de transición de estado", pero el formato UML es mejor en contenido y representación de la información.

FIGURA 7.4 **Diagrama de estado para la función de seguridad de *CasaSegura***



plo, se enciende el sistema de alarma), ocurre una transición al estado **VigilanciadelEstadodelSistema**, los mensajes en la pantalla cambian como se muestra y se invoca el proceso *SistemadeVigilanciayControl*. Fuera del estado **SistemadeVigilanciayControl** ocurren dos transiciones: 1) cuando se desactiva el sistema hay una transición de regreso al estado **Ocioso**; 2) cuando se dispara un sensor en el estado **ActivarAlarma**. Durante la revisión se consideran todas las transiciones y el contenido de todos los estados.

La tabla de activación del proceso (TAP) es un modo algo distinto de representar el comportamiento. La TAP representa la información contenida en el diagrama de estado en el contexto de los procesos, no de los estados. Es decir, la tabla indica cuáles procesos (burbujas) serán invocados en el modelo del flujo cuando ocurra un evento. La TAP se usa como guía para un diseñador que debe construir una ejecución que controle los procesos representados en este nivel. En la figura 7.5 se aprecia una TAP para el nivel 1 del modelo de flujo del software de *CasaSegura*.

La CSPEC describe el comportamiento del sistema, pero no da información acerca del funcionamiento interno de los procesos que se activan como resultado de dicho comportamiento. En la sección 7.2.4 se estudia la notación de modelación que da esta información.

7.2.4 La especificación del proceso

La *especificación del proceso* (PSPEC) se utiliza para describir todos los procesos del modelo del flujo que aparecen en el nivel final de la mejora. El contenido de la especificación del proceso incluye el texto narrativo, una descripción del lenguaje de diseño del programa[5] del algoritmo del proceso, ecuaciones matemáticas, tablas o diagramas de actividad UML. Si se da una PSPEC

5 El lenguaje de diseño del programa (LDP) mezcla la sintaxis del lenguaje de programación con el texto narrativo a fin de dar detalles del diseño del procedimiento. En el capítulo 10 se analiza el LDP.

FIGURA 7.5

Tabla de activación del proceso para la función de seguridad de *CasaSegura*

eventos de entrada						
evento de sensor	0	0	0	0	1	0
bandera de parpadeo	0	0	1	1	0	0
interruptor de iniciar o detener	0	1	0	0	0	0
estado de la acción en pantalla terminado	0	0	0	1	0	0
en marcha	0	0	1	0	0	0
tiempo terminado	0	0	0	0	0	1
salida						
señal de alarma	0	0	0	0	1	0
activación del proceso						
vigilar y controlar el sistema	0	1	0	0	1	1
activar o desactivar el sistema	0	1	0	0	0	0
mostrar mensajes y estado	1	0	1	1	1	1
interactuar con el usuario	1	0	0	1	0	1

PUNTO CLAVE

La PSPEC es una "miniespecificación" de cada transformación en el nivel más bajo de mejora de un DFD.

que acompañe a cada burbuja del modelo del flujo, se crea una "miniespecificación" que sirve como guía para diseñar la componente del software que implementará la burbuja.

Para ilustrar el uso de la PSPEC, considere la transformación *procesar password* representada en el modelo de flujo de la figura 7.2. La PSPEC de esta función adopta la forma siguiente:

> **PSPEC: procesar password (en el panel de control).** La transformación *procesar password* realiza la validación en el panel de control para la función de seguridad de *CasaSegura*. *Procesar password* recibe un password de cuatro dígitos de la función *interactuar con usuario*. Primero, el password se compara con el password maestro almacenado dentro del sistema. Si el password maestro coincide, se pasa <mensaje de identificación válida = verdadero> a la función *mostrar mensaje y estado*. Si el

CASASEGURA

Modelado del flujo de datos

La escena: Cubículo de Jamie, después de que terminó la última junta para recabar los requerimientos.

Participantes: Jamie, Vinod y Ed, miembros del equipo de ingeniería de software de *CasaSegura*.

La conversación:

(Jamie presenta a Ed y a Vinod los dibujos que hizo de los modelos que se muestran en las figuras 7.1 a 7.5.)

Jamie: En la universidad tomé un curso de ingeniería de software y aprendí esto. El profesor dijo que es un poco anticuado, pero, saben, me ayuda a aclarar las cosas.

Ed: Está muy bien. Pero no veo ninguna clase de objetos ahí.

Jamie: No... Esto sólo es un modelo del flujo con un poco de comportamiento ilustrado.

Vinod: Así que estos DFD representan la E-P-S del software, ¿o no?

Ed: ¿E-P-S?

Vinod: Entrada-proceso-salida. En realidad, los DFD son muy intuitivos... si se observan un rato, indican cómo fluyen los objetos de datos por el sistema y cómo se transforman mientras lo hacen.

Ed: Parece que pudiéramos convertir cada burbuja en un componente ejecutable... al menos en el nivel más bajo del DFD.

Jamie: Ésa es la mejor parte, sí se puede. En realidad, hay una forma de traducir los DFD a una arquitectura de diseño.

Ed: ¿En verdad?

Jamie: Sí, pero primero tenemos que desarrollar un modelo completo de los requerimientos, y esto no lo es.

Vinod: Bueno, es un primer paso, pero vamos a tener que enfrentar los elementos basados en clases y también los aspectos de comportamiento, aunque el diagrama de estado y la TAP hacen algo de eso.

Ed: Tenemos mucho trabajo por hacer y poco tiempo.

(Entra al cubículo Doug, el gerente de ingeniería de software.)

Doug: Así que dedicaremos los siguientes días a desarrollar el modelo de los requerimientos, ¿eh?

Jamie (con orgullo): Ya comenzamos.

Doug: Qué bueno, tenemos mucho trabajo por hacer y poco tiempo.

(Los tres ingenieros de software se miran entre sí y sonríen.)

password maestro no coincide, los cuatro dígitos se comparan con una tabla de passwords secundarios (que deben asignarse para recibir invitados o trabajadores que necesiten entrar a la casa cuando el propietario no esté presente). Si el password coincide con una entrada de la tabla, se pasa <mensaje de identificación válida = verdadero> a la función *mostrar mensaje y estado*. Si no coinciden, se pasa <mensaje de identificación válida = falso> a la función de mostrar mensaje y estado.

Si en esta etapa se desean detalles algorítmicos adicionales, también puede incluirse una representación del lenguaje de diseño del programa (LDP) como parte de la PSPEC. Sin embargo, muchos profesionales piensan que la versión LDP debe posponerse hasta comenzar el diseño de los componentes.

HERRAMIENTAS DE SOFTWARE

Análisis estructurado

Objetivo: Las herramientas de análisis estructurado permiten que un ingeniero de software cree modelos de datos, de flujo y de comportamiento en una forma que permite la consistencia y continuidad con facilidad para hacer la revisión, edición y ampliación. Los modelos creados con estas herramientas dan al ingeniero de software la perspectiva de la representación del análisis y lo ayudan a eliminar errores antes de que éstos se propaguen al diseño o, lo que sería peor, a la implementación.

Mecánica: Las herramientas de esta categoría son un "diccionario de datos", como la base de datos central para describir todos los objetos de datos. Una vez definidas las entradas del diccionario, se crean diagramas entidad-relación y se desarrollan las jerarquías de los objetos. Las características de los diagramas de flujo de datos permiten que sea fácil crear este modelo gráfico y también proveen de características para generar PSPEC y CSPEC. Asimismo, las herramientas de análisis permiten que el ingeniero de software produzca modelos de comportamiento con el empleo del diagrama de estado como notación operativa.

Herramientas representativas.[6]

MacA&D, WinA&D, desarrolladas por software Excel (**www.excelsoftware.com**), brinda un conjunto de herramientas de análisis y diseño sencillas y baratas para computadoras Mac y Windows.

MetaCASE Workbench, desarrollada por MetaCase Consulting (**www.metacase.com**), es una metaherramienta utilizada para definir un método de análisis o diseño (incluso análisis estructurado) y sus conceptos, reglas, notaciones y generadores.

System Architect, desarrollado por Popkin Software (**www.popkin.com**), da una amplia variedad de herramientas de análisis y diseño, incluso para modelar datos y hacer análisis estructurado.

7.3 CREACIÓN DE UN MODELO DE COMPORTAMIENTO

? ¿Cómo se modela la reacción del software ante algún evento externo?

La notación de modelado que hemos estudiado hasta el momento representa elementos estáticos del modelo de requerimientos. Es hora de hacer la transición al comportamiento dinámico del sistema o producto. Para hacerlo, dicho comportamiento se representa como función de eventos y tiempo específicos.

El *modelo de comportamiento* indica la forma en la que responderá el software a eventos o estímulos externos. Para generar el modelo deben seguirse los pasos siguientes:

1. Evaluar todos los casos de uso para entender por completo la secuencia de interacción dentro del sistema.
2. Identificar los eventos que conducen la secuencia de interacción y que entienden el modo en el que éstos se relacionan con objetos específicos.
3. Crear una secuencia para cada caso de uso.
4. Construir un diagrama de estado para el sistema.
5. Revisar el modelo de comportamiento para verificar la exactitud y consistencia.

En las secciones siguientes se estudia cada uno de estos pasos.

6 Las herramientas mencionadas aquí no son obligatorias sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

7.3.1 Identificar los eventos con el caso de uso

En el capítulo 6 se aprendió que el caso de uso representa una secuencia de actividades que involucra a los actores y al sistema. En general, un evento ocurre siempre que el sistema y un actor intercambian información. En la sección 7.2.3 se dijo que un evento *no* es la información que se intercambia, sino el hecho de que se intercambió la información.

Un caso de uso se estudia para efectos del intercambio de información. Para ilustrarlo, volvamos al caso de uso de una parte de la función de seguridad de *CasaSegura*.

> El propietario utiliza el teclado para escribir un password de cuatro dígitos. El password se compara con el password válido guardado en el sistema. Si el password es incorrecto, el panel de control emitirá un sonido una vez y se reiniciará para recibir entradas adicionales. Si el password es correcto, el panel de control queda en espera de otras acciones.

Las partes subrayadas del escenario del caso de uso indican eventos. Debe identificarse un actor para cada evento, anotarse la información que se intercambia y enlistarse cualesquiera condiciones o restricciones.

Como ejemplo de evento común considere la frase subrayada en el caso de uso "el propietario utiliza el teclado para escribir un password de cuatro dígitos". En el contexto del modelo de los requerimientos, el objeto **PropietariodeCasa**[7] transmite un evento al objeto **PaneldeControl**. El evento tal vez se llame *password introducido*. La información que se transfiere son los cuatro dígitos que constituyen el password, pero ésta no es una parte esencial del modelo de comportamiento. Es importante observar que ciertos eventos tienen un efecto explícito en el flujo del control del caso de uso, mientras que otros no lo tienen. Por ejemplo, el evento *password introducido* no cambia explícitamente el flujo del control del caso de uso, pero los resultados del evento *password comparado* (derivado de la interacción el "password se compara con el password válido guardado en el sistema") tendrán un efecto explícito en el flujo de información y control del software *CasaSegura*.

Una vez identificados todos los eventos, se asignan a los objetos involucrados. Los objetos son responsables de la generación de eventos (por ejemplo, **Propietario** genera el evento *password introducido*) o de reconocer los eventos que hayan ocurrido en cualquier lugar (**PaneldeControl** reconoce el resultado binario del evento *password comparado*).

7.3.2 Representaciones de estado

En el contexto del modelado del comportamiento deben considerarse dos caracterizaciones diferentes de los estados: 1) el estado de cada clase cuando el sistema ejecuta su función y 2) el estado del sistema según se observa desde el exterior cuando realiza su función.[8]

PUNTO CLAVE

El sistema tiene estados que representan un comportamiento específico observable desde el exterior; una clase tiene estados que representan su comportamiento cuando el sistema realiza sus funciones.

El estado de una clase tiene características tanto pasivas como activas [Cha93]. Un *estado pasivo* es sencillamente el estado actual de todos los atributos de un objeto. Por ejemplo, el estado pasivo de la clase **Jugador** (en la aplicación de juego de video que se vio en el capítulo 6) incluiría los atributos actuales **posición** y **orientación** de **Jugador**, así como otras características de éste que sean relevantes para el juego (por ejemplo, un atributo que incluya **deseos mágicos restantes**). El *estado activo* de un objeto indica el estado actual del objeto conforme pasa por una transformación o procesamiento continuos. La clase **Jugador** tal vez tenga los siguientes estados activos: *moverse, en descanso, herido, en curación, atrapado, perdido*, etc. Debe ocurrir un evento (en ocasiones llamado *disparador o trigger*) para forzar a un objeto a hacer una transición de un estado activo a otro.

---

7 En este ejemplo se supone que cada usuario (propietario) que interactúe con *CasaSegura* tiene un password de identificación, por lo que es un objeto legítimo.

8 Los diagramas de estado presentados en el capítulo 6 y en la sección 7.3.2 ilustran el estado del sistema. En esta sección, nuestro análisis se centrará en el estado de cada clase dentro del modelo del análisis.

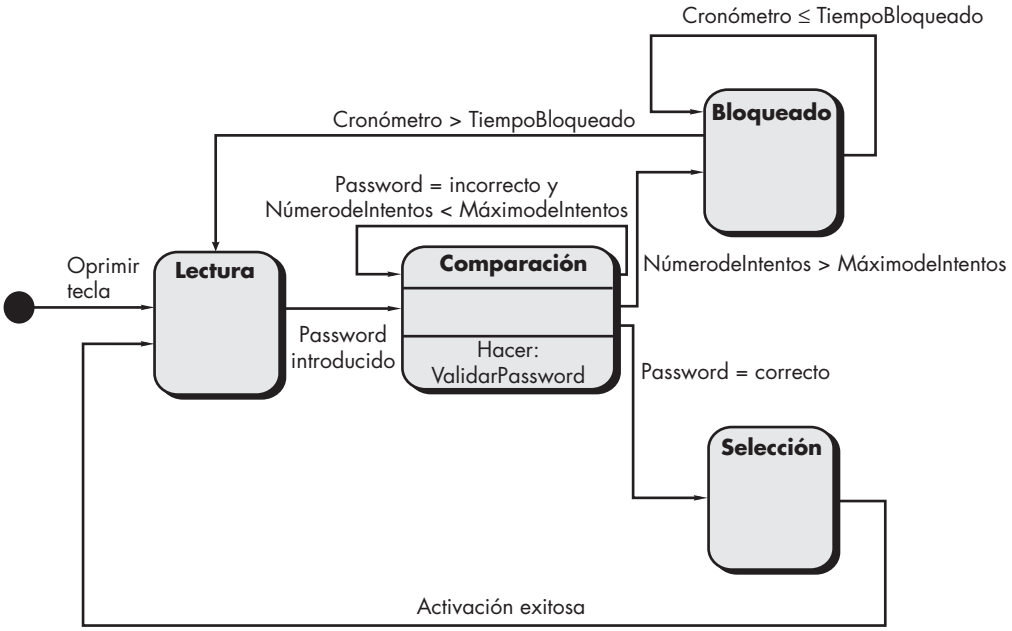


FIGURA 7.6
Diagrama de estado para la clase PaneldeControl

En los párrafos que siguen se analizan dos representaciones distintas del comportamiento. La primera indica la manera en la que una clase individual cambia su estado con base en eventos externos, y la segunda muestra el comportamiento del software como una función del tiempo.

Diagramas de estado para clases de análisis. Un componente de modelo de comportamiento es un diagrama de estado UML[9] que representa estados activos para cada clase y los eventos (disparadores) que causan cambios en dichos estados activos. La figura 7.6 ilustra un diagrama de estado para el objeto **PaneldeControl** en la función de seguridad de *CasaSegura*.

Cada flecha que aparece en la figura 7.6 representa una transición de un estado activo de un objeto a otro. Las leyendas en cada flecha representan el evento que dispara la transición. Aunque el modelo de estado activo da una perspectiva útil de la "historia de la vida" de un objeto, es posible especificar información adicional para llegar a más profundidad en la comprensión del comportamiento de un objeto. Además de especificar el evento que hace que la transición ocurra, puede especificarse una guardia y una acción [Cha93]. Una *guardia* es una condición booleana que debe satisfacerse para que tenga lugar una transición. Por ejemplo, la guardia para la transición del estado de "lectura" al de "comparación" en la figura 7.6 se determina con el análisis del caso de uso:

Si (password de entrada = 4 dígitos) entonces comparar con el password guardado

En general, la guardia para una transición depende del valor de uno o más atributos de un objeto. En otras palabras, depende del estado pasivo del objeto.

Una *acción* sucede en forma concurrente con la transición de estado o como consecuencia de ella, y por lo general involucra una o más operaciones (responsabilidades) del objeto. Por

9 Si el lector no está familiarizado con UML, puede consultar en el apéndice 1 una breve introducción a esta importante notación de modelación.

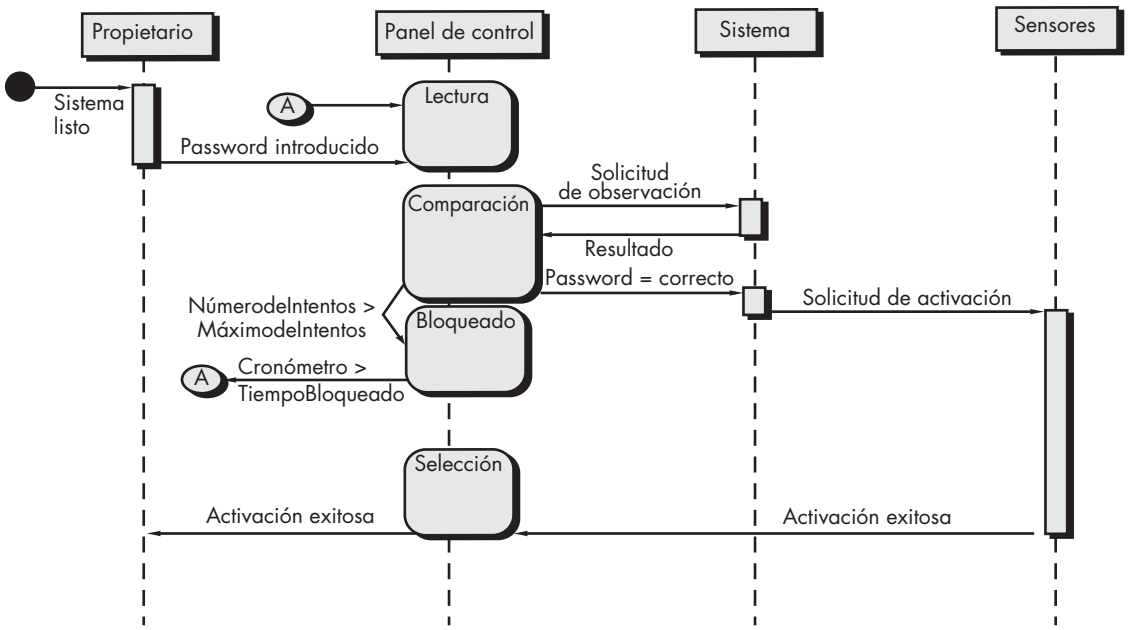


FIGURA 7.7
Diagrama de secuencia (parcial) para la función *CasaSegura*

ejemplo, la acción conectada con el evento *password introducido* (véase la figura 7.6) es una operación llamada *ValidarPassword ( )* que accede a un objeto **password** y realiza una comparación dígito por dígito para validar el password introducido.

Diagramas de secuencia. El segundo tipo de representación del comportamiento, llamado *diagrama de secuencia* en UML, indica la forma en la que los eventos provocan transiciones de un objeto a otro. Una vez identificados los objetos por medio del análisis del caso de uso, el modelador crea un diagrama de secuencia: representación del modo en el que los eventos causan el flujo de uno a otro como función del tiempo. En esencia, el diagrama de secuencia es una versión taquigráfica del caso de uso. Representa las clases password y los eventos que hacen que el comportamiento avance de una clase a otra.

A diferencia de un diagrama de estado que representa el comportamiento sin fijarse en las clases involucradas, un diagrama de secuencia representa el comportamiento, describiendo la forma en la que las clases pasan de un estado a otro.

La figura 7.7 ilustra un diagrama parcial de secuencia para la función de seguridad de *CasaSegura*. Cada flecha representa un evento (derivado de un caso de uso) e indica la forma en la que éste canaliza el comportamiento entre los objetos de *CasaSegura*. El tiempo se mide en la dirección vertical (hacia abajo) y los rectángulos verticales angostos representan el que toma el procesamiento de una actividad. Los estados se presentan a lo largo de una línea de tiempo vertical.

El primer evento, *sistema listo*, se deriva del ambiente externo y canaliza el comportamiento al objeto **Propietario**. El propietario introduce un password. El evento *solicitud de observación* pasa al **Sistema**, que observa el password en una base de datos sencilla y devuelve un *resultado* (*encontrada o no encontrada*) a **PaneldeControl** (ahora en el estado de *comparación*). Un password válido da como resultado el evento *password = correcto* hacia **Sistema**, lo que activa a **Sensores** con un evento de *solicitud de activación*. Por último, el control pasa de nuevo al propietario con el evento *activación exitosa*.

Una vez que se ha desarrollado un diagrama de secuencia completo, todos los eventos que causan transiciones entre objetos del sistema se recopilan en un conjunto de eventos de entrada y de salida (desde un objeto). Esta información es útil en la generación de un diseño eficaz para el sistema que se va a construir.

HERRAMIENTAS DE SOFTWARE

Modelación de análisis generalizado en UML

Objetivo: Las herramientas de modelado del análisis dan la capacidad de desarrollar modelos basados en el escenario, en la clase y en el comportamiento con el uso de notación UML.

Mecánica: Las herramientas en esta categoría dan apoyo a toda la variedad de diagramas UML requeridos para construir un modelo del análisis (estas herramientas también apoyan el modelado del diseño). Además de los diagramas, las herramientas en esta categoría 1) hacen revisiones respecto de la consistencia y corrección para todos los diagramas UML, 2) proveen vínculos para producir el diseño y generar el código, y 3) construyen una base de datos que permite administrar y evaluar modelos UML grandes requeridos en sistemas complejos.

Herramientas representativas:[10]

Las herramientas siguientes apoyan toda la variedad de diagramas UML que se requieren para modelar el análisis:

ArgoUML es una herramienta de fuente abierta disponible en **argouml.tigris.org**.
Enterprise Architect, desarrollada por Sparx Systems (**www.sparxsystems.com.au**).
PowerDesigner, desarrollada por Sybase (**www.sybase.com**).
Rational Rose, desarrollada por IBM (Rational) (**www01.ibm.com/software/rational/**).
System Architect, desarrollada por Popkin Software (**www.popkin.com**).
UML Studio, desarrollada por Pragsoft Corporation (**www.pragsoft.com**).
Visio, desarrollada por Microsoft (**www.microsoft.com**).
Visual UML, desarrollada por Visual Object Modelers (**www.visualuml.com**).

7.4 PATRONES PARA EL MODELADO DE REQUERIMIENTOS

Los patrones de software son un mecanismo para capturar conocimiento del dominio, en forma que permita que vuelva a aplicarse cuando se encuentre un problema nuevo. En ciertos casos, el conocimiento del dominio se aplica a un nuevo problema dentro del mismo dominio de la aplicación. En otros, el conocimiento del dominio capturado por un patrón puede aplicarse por analogía a otro dominio de una aplicación diferente por completo.

El autor original de un patrón de análisis no "crea" el patrón, sino que lo *descubre* a medida que se realiza el trabajo de ingeniería de requerimientos. Una vez descubierto el patrón, se documenta describiendo "explícitamente el problema general al que es aplicable el patrón, la solución prescrita, las suposiciones y restricciones del uso del patrón en la práctica y, con frecuencia, alguna otra información sobre éste, como la motivación y las fuerzas que impulsan el empleo del patrón, el análisis de las ventajas y desventajas del mismo y referencias a algunos ejemplos conocidos de su empleo en aplicaciones prácticas" [Dev01].

En el capítulo 5 se presentó el concepto de patrones de análisis y se indicó que éstos representan una solución que con frecuencia incorpora una clase, función o comportamiento dentro del dominio de aplicación. El patrón vuelve a utilizarse cuando se hace el modelado de los requerimientos para una aplicación dentro del dominio.[11] Los patrones de análisis se guardan en un depósito para que los miembros del equipo de software usen herramientas de búsqueda para encontrarlos y volverlos a emplear. Una vez seleccionado un patrón apropiado, se integra en el modelo de requerimientos, haciendo referencia a su nombre.

7.4.1 Descubrimiento de patrones de análisis

El modelo de requerimientos está formado por una amplia variedad de elementos: basados en el escenario (casos de uso), orientados a datos (el modelo de datos), basados en clases, orientados al flujo y del comportamiento. Cada uno de estos elementos estudia el problema desde

10 Las herramientas mencionadas aquí no son obligatorias sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.
11 En el capítulo 12 se presenta un análisis a profundidad del uso de patrones durante el diseño del software.

una perspectiva diferente y da la oportunidad de descubrir patrones que tal vez suceden en un dominio de aplicación o por analogía en distintos dominios de aplicación.

El elemento más fundamental en la descripción de un modelo de requerimientos es el caso de uso. En el contexto de este análisis, un conjunto coherente de casos de uso sirve como base para descubrir uno o más patrones de análisis. Un *patrón de análisis semántico* (PAS) "es un patrón que describe un conjunto pequeño de casos de uso coherentes que describen a su vez una aplicación general" [Fer00].

Considere el siguiente caso de uso preliminar para el software que se requiere a fin de controlar y vigilar una cámara de visión real y un sensor de proximidad para un automóvil:

> **Caso de uso:** ***Vigilar el movimiento en reversa***
>
> **Descripción:** Cuando se coloca el vehículo en *reversa*, el software de control permite que se transmita un video a una pantalla que está en el tablero, desde una cámara colocada en la parte posterior. El software superpone varias líneas de orientación y distancia en la pantalla a fin de que el operador del auto mantenga la orientación cuando éste se mueve en reversa. El software de control también vigila un sensor de proximidad con el fin de determinar si un objeto se encuentra dentro de una distancia de 10 pies desde la parte trasera del carro. Esto frenará al vehículo de manera automática si el sensor de proximidad indica que hay un objeto a *x* pies de la defensa trasera, donde *x* se determina con base en la velocidad del automóvil.

Este caso de uso implica varias funciones que se mejorarían y elaborarían (en un conjunto coherente de casos de uso) durante la reunión para recabar y modelar los requerimientos. Sin importar cuánta elaboración se logre, los casos de uso sugieren un PAS sencillo pero con amplias aplicaciones (la vigilancia y control de sensores y actuadores de un sistema físico con base en software). En este caso, los "sensores" dan información en video sobre la proximidad. El "actuador" es el sistema de frenado del vehículo (que se invoca si hay un objeto muy cerca de éste). Pero en un caso más general, se descubre un patrón de aplicación muy amplio.

En muchos dominios distintos de aplicación, se requiere software para vigilar sensores y controlar actuadores físicos. Se concluye que podría usarse mucho un patrón de análisis que describa los requerimientos generales para esta capacidad. El patrón, llamado **Actuador-Sensor**, se aplicaría como parte del modelo de requerimientos para *CasaSegura* y se analiza en la sección 7.4.2, a continuación.

7.4.2 Ejemplo de patrón de requerimientos: Actuador-Sensor[12]

Uno de los requerimientos de la función de seguridad de *CasaSegura* es la capacidad de vigilar sensores de seguridad (por ejemplo, sensores de frenado, de incendio, de humo o contenido de CO, de agua, etc.). Las extensiones basadas en internet para *CasaSegura* requerirán la capacidad de controlar el movimiento (por ejemplo, apertura, acercamiento, etc.) de una cámara de seguridad dentro de una residencia. La implicación es que el software de *CasaSegura* debe manejar varios sensores y "actuadores" (como los mecanismos de control de las cámaras).

Konrad y Cheng [Kon02] sugieren un patrón llamado **Actuador-Sensor** que da una guía útil para modelar este requerimiento dentro del software de *CasaSegura*. A continuación se presenta una versión abreviada del patrón **Actuador-Sensor**, desarrollada originalmente para aplicaciones automotrices.

Nombre del patrón. **Actuador-Sensor**

Objetivo. Especifica distintas clases de sensores y actuadores en un sistema incrustado.

Motivación. Por lo general, los sistemas incrustados tienen varias clases de sensores y actuadores, conectados en forma directa o indirecta con una unidad de control. Aunque muchos de

12 Esta sección se adaptó de [Kon02] con permiso de los autores.

los sensores y actuadores se ven muy distintos, su comportamiento es lo bastante similar como para estructurarlos en un patrón. Éste ilustra la forma de especificar los sensores y actuadores para un sistema, incluso los atributos y operaciones. El patrón **Actuador-Sensor** usa un mecanismo para *jalar* (solicitud explícita de información) **SensoresPasivos** y otro mecanismo para *empujar* (emisión de información) los **SensoresActivos**.

Restricciones

- Cada sensor pasivo debe tener algún método para leer la entrada de un sensor y los atributos que representan al valor del sensor.
- Cada sensor activo debe tener capacidades para emitir mensajes actualizados cuando su valor cambie.
- Cada sensor activo debe enviar un *latido de vida*, mensaje de estado que se emite cada cierto tiempo para detectar fallas.
- Cada actuador debe tener un método para invocar la respuesta apropiada determinada por el **CálculodeComponente**.
- Cada sensor y actuador deben tener una función implementada para revisar su propio estado de operación.
- Cada sensor y actuador debe ser capaz de someter a prueba la validez de los valores recibidos o enviados y fijar su estado de operación si los valores se encuentran fuera de las especificaciones.

Aplicabilidad. Es útil en cualquier sistema en el que haya varios sensores y actuadores.

Estructura. En la figura 7.8 se presenta un diagrama de clase UML para el patrón **Actuador-Sensor**. **Actuador**, **SensorPasivo** y **SensorActivo** son clases abstractas y están escritas con letra cursiva. En este patrón hay cuatro tipos diferentes de sensores y actuadores. Las clases **Booleano**, **Entero** y **Real** representan los tipos más comunes de sensores y actuadores. Las clases complejas de éstos son aquellas que usan valores que no se representan con facilidad en términos de tipos de datos primitivos, tales como los de un radar. No obstante, estos equipos

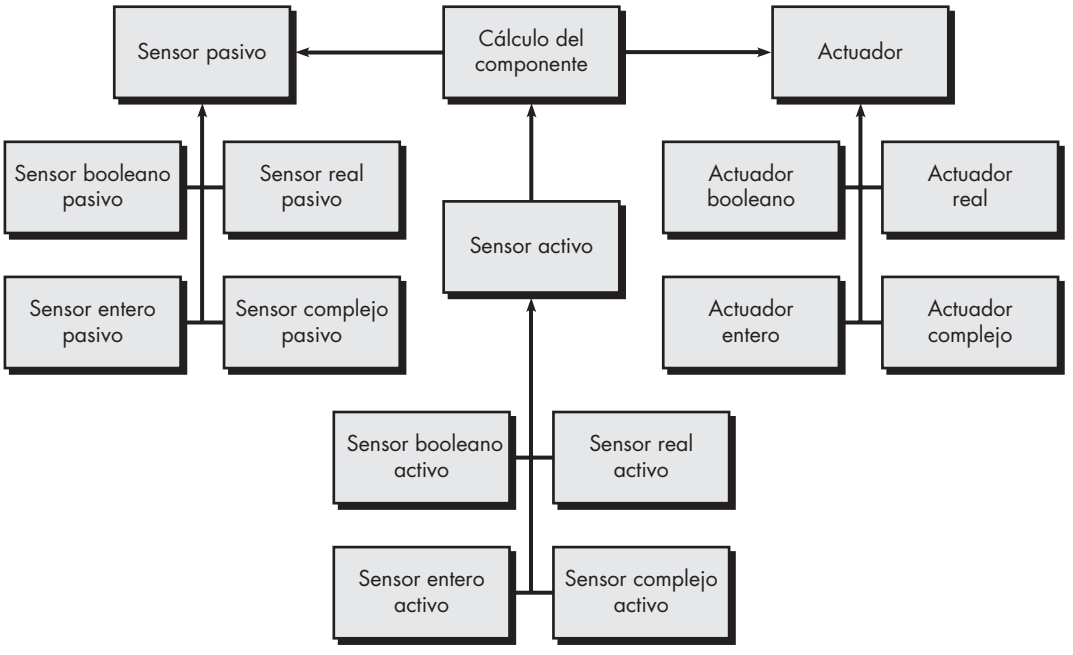
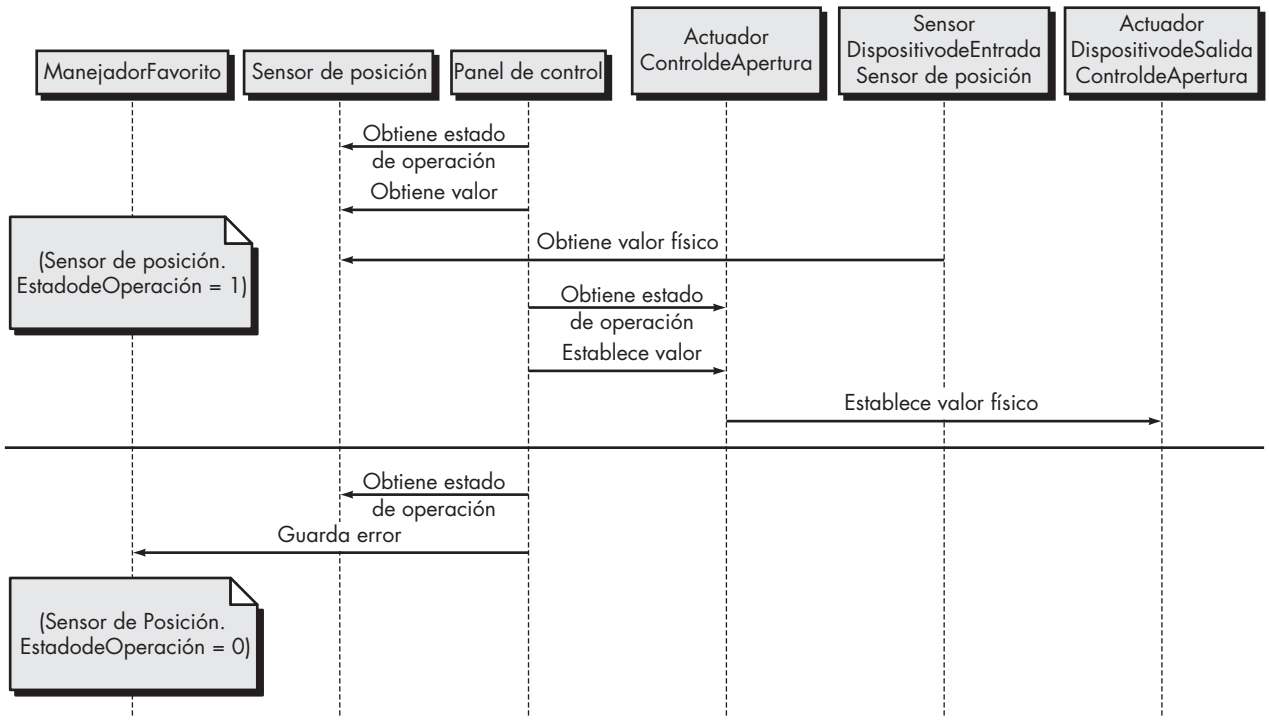


FIGURA 7.8

Diagrama de secuencia UML para el patrón Actuador-Sensor.
***Fuente:* Adaptado de [Kon02], con permiso.**

FIGURA 7.9 **Diagrama de clase UML para el patrón Actuador-Sensor.** ***Fuente:*** **Reimpreso de [Kon02], con permiso.**



deben heredar la interfaz de las clases abstractas, ya que deben tener funciones básicas, tales como consultar los estados de operación.

Comportamiento: La figura 7.9 presenta un diagrama de secuencia UML para un ejemplo de patrón **Actuador-Sensor** según podría aplicarse a la función de *CasaSegura* que controla el posicionamiento (como la apertura y el acercamiento) de una cámara de seguridad. Aquí, el **PaneldeControl**[13] consulta un sensor (uno de posición pasiva) y un actuador (control de apertura) para comprobar el estado de operación con fines de diagnóstico antes de leer o establecer un valor. Los mensajes *Establecer un Valor Físico* y *Obtener un Valor Físico* no son mensajes entre objetos. En vez de ello, describen la interacción entre los dispositivos físicos del sistema y sus contrapartes de software. En la parte inferior del diagrama, bajo la línea horizontal, el **SensordePosición** reporta que el estado de operación es igual a cero. Entonces, el **CálculodeComponente** (representado como **PaneldeControl**) envía el código de error para una falla de posición de un sensor al **ManejadordeFallas**, que decidirá cómo afecta este error al sistema y qué acciones se requieren. Obtiene los datos de los sensores y calcula la respuesta requerida por parte de los actuadores.

Participantes. Esta sección de la descripción de patrones "clasifica las clases u objetos incluidos en el patrón de requerimientos" [Kon02] y describe las responsabilidades de cada clase u objeto (véase la figura 7.8). A continuación se presenta una lista abreviada:

- **Resumen de SensorPasivo:** Define una interfaz para los sensores pasivos.
- **SensorBooleanoPasivo:** Define los sensores booleanos pasivos.
- **SensorEnteroPasivo:** Define los sensores enteros pasivos.

13 El patrón original usa la frase general **CálculodeComponente**.

- **SensorRealPasivo:** Define los sensores reales pasivos.
- **Resumen de SensorActivo:** Define una interfaz para los sensores activos.
- **SensorBooleanoActivo:** Define los sensores booleanos activos.
- **SensorEnteroActivo:** Define los sensores enteros activos.
- **SensorRealActivo:** Define los sensores reales activos.
- **Resumen de actuador:** Define una interfaz para los actuadores.
- **ActuadorBooleano:** Define los actuadores booleanos.
- **ActuadorEntero:** Define los actuadores enteros.
- **ActuadorReal:** Define los actuadores reales.
- **CálculodeComponente:** Parte central del controlador; obtiene los datos de los sensores y calcula la respuesta requerida para los actuadores.
- **SensorComplejoActivo:** Los sensores complejos activos tienen la funcionalidad básica de la clase **SensorActivo**, pero es necesario especificar métodos y atributos adicionales más elaborados.
- **SensorComplejoPasivo:** Los sensores complejos pasivos tienen la funcionalidad básica de la clase abstracta **SensorPasivo**, pero se necesita especificar métodos y atributos adicionales más elaborados.
- **ActuadorComplejo:** Los actuadores complejos también tienen la funcionalidad básica de la clase abstracta **Actuador**, pero se requiere especificar métodos y atributos adicionales más elaborados.

Colaboraciones. Esta sección describe cómo interactúan los objetos y clases entre sí, y cómo efectúa cada uno sus responsabilidades.

- Cuando **CálculodeComponente** necesita actualizar el valor de un **SensorPasivo**, consulta a los sensores y solicita el valor enviando el mensaje apropiado.
- Los **SensoresActivos** no son consultados. Inician la transmisión de los valores del sensor a la unidad de cálculo, con el uso del método apropiado para establecer el valor en **CálculodeComponente**. Durante un tiempo especificado, envían un latido de vida al menos una vez con el fin de actualizar sus parámetros de tiempo con el reloj del sistema.
- Cuando **CálculodeComponente** necesita establecer el valor de un actuador, envía el valor a éste.
- **CálculodeComponente** consulta y establece el estado de operación de los sensores y actuadores por medio de los métodos apropiados. Si un estado de operación es cero, entonces se envía el error al **ManejadordeFallas**, clase que contiene métodos para manejar mensajes de error, tales como reiniciar un mecanismo más elaborado de recuperación o un dispositivo de respaldo. Si no es posible la recuperación, entonces el sistema sólo usa el último valor conocido para el sensor o uno preestablecido.
- Los **SensoresActivos** ofrecen métodos para agregar o retirar los evaluadores o evalúan rangos de los componentes que quieren que reciban los mensajes en caso de un cambio de valor.

Consecuencias

1. Las clases sensor y actuador tienen una interfaz común.
2. Sólo puede accederse a los atributos de clase a través de mensajes y la clase decide si se aceptan o no. Por ejemplo, si se establece el valor de un actuador por arriba del

máximo, entonces la clase actuador tal vez no acepte el mensaje, o quizá emplee un valor máximo preestablecido.

3. La complejidad del sistema es potencialmente reducida debido a la uniformidad de las interfaces para los actuadores y sensores.

La descripción del patrón de requerimientos también da referencias acerca de otros requerimientos y patrones de diseño relacionados.

7.5 Modelado de requerimientos para webapps[14]

Es frecuente que los desarrolladores de web manifiesten escepticismo cuando se plantea la idea del análisis de los requerimientos para webapps. Acostumbran decir: "después de todo, el proceso de desarrollo en web debe ser ágil y el análisis toma tiempo. Nos hará ser lentos justo cuando necesitemos diseñar y construir la *webapp*".

El análisis de los requerimientos lleva tiempo, pero resolver el problema equivocado toma aún más tiempo. La pregunta que debe responder todo desarrollador en web es sencilla: ¿estás seguro de que entiendes los requerimientos del problema? Si la respuesta es un "sí" inequívoco, entonces tal vez sea posible omitir el modelado de los requerimientos, pero si la respuesta es "no", entonces ésta debe llevarse a cabo.

7.5.1 ¿Cuánto análisis es suficiente?

El grado en el que se profundice en el modelado de los requerimientos para las *webapps* depende de los factores siguientes:

- Tamaño y complejidad del incremento de la *webapp*.
- Número de participantes (el análisis ayuda a identificar los requerimientos conflictivos que provienen de distintas fuentes).
- Tamaño del equipo de la *webapp*.
- Grado en el que los miembros del equipo han trabajado juntos antes (el análisis ayuda a desarrollar una comprensión común del proyecto).
- Medida en la que el éxito de la organización depende directamente del éxito de la *webapp*.

El inverso de los puntos anteriores es que a medida que el proyecto se hace más chico, que el número de participantes disminuye, que el equipo de desarrollo es más cohesivo y que la aplicación es menos crítica, es razonable aplicar un enfoque más ligero para el análisis.

Aunque es una buena idea analizar el problema *antes* de que comience el diseño, no es verdad que *todo* el análisis deba preceder a *todo* el diseño. En realidad, el diseño de una parte específica de la *webapp* sólo demanda un análisis de los requerimientos que afectan a sólo esa parte de la *webapp*. Como un ejemplo proveniente de *CasaSegura*, podría diseñarse con validez la estética general del sitio web (formatos, colores, etc.) sin tener que analizar los requerimientos funcionales de las capacidades de comercio electrónico. Sólo se necesita analizar aquella parte del problema que sea relevante para el trabajo de diseño del incremento que se va a entregar.

7.5.2 Entrada del modelado de los requerimientos

En el capítulo 2 se analizó una versión ágil del proceso de software general que puede aplicarse cuando se hace la ingeniería de las *webapps*. El proceso incorpora una actividad de comunica-

14 Esta sección se adaptó de Pressman y Lowe [Pre08], con permiso.

ción que identifica a los participantes y las categorías de usuario, el contexto del negocio, las metas definidas de información y aplicación, requerimientos generales de *webapps* y los escenarios de uso, información que se convierte en la entrada del modelado de los requerimientos. Esta información se representa en forma de descripciones hechas en lenguaje natural, a grandes rasgos, en bosquejos y otras representaciones no formales.

El análisis toma esta información, la estructura con el empleo de un esquema de representación definido formalmente (donde sea apropiado) y luego produce como salida modelos más rigurosos. El modelo de requerimientos brinda una indicación detallada de la verdadera estructura del problema y da una perspectiva de la forma de la solución.

En el capítulo 6 se introdujo la función **AVC-MVC** (vigilancia con cámaras). En ese momento, esta función parecía relativamente clara y se describió con cierto detalle como parte del caso de uso (véase la sección 6.2.1). Sin embargo, la revisión del caso de uso quizá revele información oculta, ambigua o poco clara.

Algunos aspectos de esta información faltante emergerían de manera natural durante el diseño. Los ejemplos quizá incluyan el formato específico de los botones de función, su aspecto y percepción estética, el tamaño de las vistas instantáneas, la colocación del ángulo de las cámaras y el plano de la casa, o incluso minucias tales como las longitudes máxima y mínima de las claves. Algunos de estos aspectos son decisiones de diseño (como el aspecto de los botones) y otros son requerimientos (como la longitud de las claves) que no influyen de manera fundamental en los primeros trabajos de diseño.

Pero cierta información faltante sí podría influir en el diseño general y se relaciona más con la comprensión real de los requerimientos. Por ejemplo:

P_1: ¿Cuál es la resolución del video de salida que dan las cámaras de *CasaSegura*?

P_2: ¿Qué ocurre si se encuentra una condición de alarma mientras la cámara está siendo vigilada?

P_3: ¿Cómo maneja el sistema las cámaras con vistas panorámicas y de acercamiento?

P_4: ¿Qué información debe darse junto con la vista de la cámara (por ejemplo, ubicación, fecha y hora, último acceso, etcétera)?

Ninguna de estas preguntas fue identificada o considerada en el desarrollo inicial del caso de uso; no obstante, las respuestas podrían tener un efecto significativo en los diferentes aspectos del diseño.

Por tanto, es razonable concluir que aunque la actividad de comunicación provea un buen fundamento para entender, el análisis de los requerimientos mejora este entendimiento al dar una interpretación adicional. Como la estructura del problema se delinea como parte del modelo de requerimientos, invariablemente surgen preguntas. Son éstas las que llenan los huecos y, en ciertos casos, en realidad ayudan a encontrarlos.

En resumen, la información obtenida durante la actividad de comunicación será la entrada del modelo de los requerimientos, cualquiera que sea, desde un correo electrónico informal hasta un proyecto detallado con escenarios de uso exhaustivos y especificaciones del producto.

7.5.3 Salida del modelado de los requerimientos

El análisis de los requerimientos provee un mecanismo disciplinado para representar y evaluar el contenido y funcionamiento de las *webapp*, los modos de interacción que hallarán los usuarios y el ambiente e infraestructura en las que reside la *webapp*.

Cada una de estas características se representa como un conjunto de modelos que permiten que los requerimientos de la *webapp* sean analizados en forma estructurada. Si bien los modelos específicos dependen en gran medida de la naturaleza de la *webapp*, hay cinco clases principales de ellos:

- **Modelo de contenido:** identifica el espectro completo de contenido que dará la *webapp*. El contenido incluye datos de texto, gráficos e imágenes, video y sonido.
- **Modelo de interacción:** describe la manera en que los usuarios interactúan con la *webapp*.
- **Modelo funcional:** define las operaciones que se aplicarán al contenido de la *webapp* y describe otras funciones de procesamiento que son independientes del contenido pero necesarias para el usuario final.
- **Modelo de navegación:** define la estrategia general de navegación para la *webapp*.
- **Modelo de configuración:** describe el ambiente e infraestructura en la que reside la *webapp*.

Es posible desarrollar cada uno de estos modelos con el empleo de un esquema de representación (llamado con frecuencia "lenguaje") que permite que su objetivo y estructura se comuniquen y evalúen con facilidad entre los miembros del equipo de ingeniería de web y otros participantes. En consecuencia, se identifica una lista de aspectos clave (como errores, omisiones, inconsistencias, sugerencias de mejora o modificaciones, puntos de aclaración, etc.) para trabajar sobre ellos.

7.5.4 Modelo del contenido de las *webapps*

El modelo de contenido incluye elementos estructurales que dan un punto de vista importante de los requerimientos del contenido de una *webapp*. Estos elementos estructurales agrupan los objetos del contenido y todas las clases de análisis, entidades visibles para el usuario que se crean o manipulan cuando éste interactúa con la *webapp*.[15]

El contenido puede desarrollarse antes de la implementación de la *webapp*, mientras ésta se construye o cuando ya opera. En cualquier caso, se incorpora por referencia de navegación en la estructura general de la *webapp*. Un *objeto de contenido* es una descripción de un producto en forma de texto, un artículo que describe un evento deportivo, una fotografía tomada en éste, la respuesta de un usuario en un foro de análisis, una representación animada de un logotipo corporativo, una película corta de un discurso o una grabación en audio para una presentación con diapositivas. Los objetos de contenido pueden almacenarse como archivos separados, incrustarse directamente en páginas web u obtenerse en forma dinámica de una base de datos. En otras palabras, un objeto de contenido es cualquier aspecto de información cohesiva que se presente al usuario final.

Los objetos de contenido se determinan directamente a partir de casos de uso, estudiando la descripción del escenario respecto de referencias directas e indirectas al contenido. Por ejemplo, se establece en **CasaSeguraAsegurada.com** una *webapp* que da apoyo a *CasaSegura*. Un caso de uso, *Comprar componentes seleccionados de CasaSegura*, describe el escenario que se requiere para comprar un componente de *CasaSegura* y que contiene la siguiente oración:

> Podré obtener información descriptiva y de precios de cada componente del producto.

El modelo de contenido debe ser capaz de describir el objeto de contenido **Componente**. En muchas circunstancias, para definir los requerimientos para el contenido que debe diseñarse e implementarse, es suficiente una lista sencilla de los objetos de contenido, junto con la descripción breve de cada uno. Sin embargo, en ciertos casos, el modelo de contenido se beneficia de un análisis más rico que ilustre en forma gráfica las relaciones entre los objetos de contenido y la jerarquía que mantiene una *webapp*.

Por ejemplo, tome en cuenta el *árbol de datos* [Sri01] creado por el componente **CasaSeguraAsegurada.com** que aparece en la figura 7.10. El árbol representa una jerarquía de informa-

15 Las clases de análisis se estudiaron en el capítulo 6.

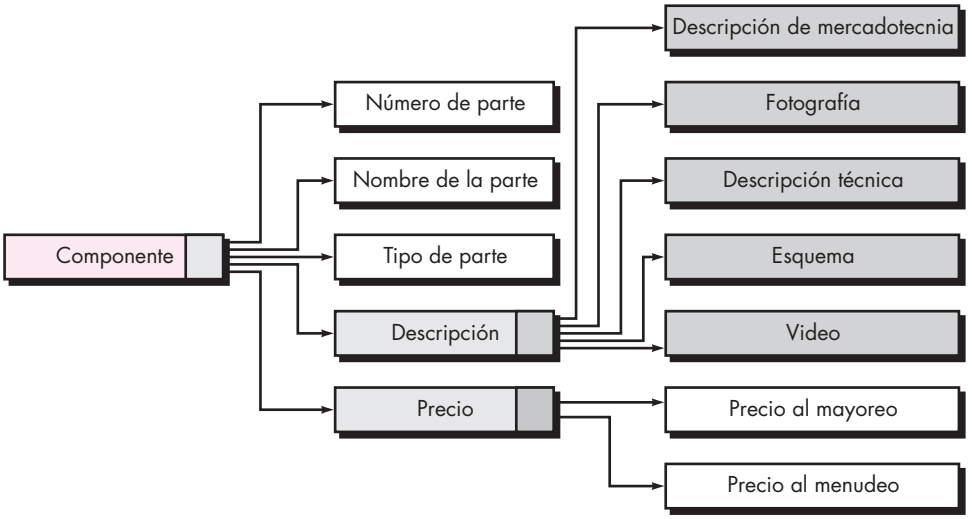


FIGURA 7.10
Árbol de datos para el componente CasaSeguraAsegurada.com

ción que se utiliza para describir un componente. Los aspectos de datos simples o compuestos (uno o más valores de los datos) se representan con rectángulos sin sombra. Los objetos de contenido se representan con rectángulos con sombra. En la figura, **descripción** está definida por cinco objetos (los rectángulos sombreados). En ciertos casos, uno o más de estos objetos se mejorará más conforme se expanda el árbol de datos.

Es posible crear un árbol de datos para cualquier contenido que se componga de múltiples objetos de contenido y aspectos de datos. El árbol de datos se desarrolla como un esfuerzo para definir relaciones jerárquicas entre los objetos de contenido y para dar un medio de revisión del contenido a fin de que se descubran las omisiones e inconsistencias antes de que comience el diseño. Además, el árbol de datos sirve como base para diseñar el contenido.

7.5.5 Modelo de la interacción para *webapps*

La gran mayoría de *webapps* permiten una "conversación" entre un usuario final y funcionalidad, contenido y comportamiento de la aplicación. Esta conversación se describe con el uso de un modelo de *interacción* que se compone de uno o más de los elementos siguientes: 1) casos de uso, 2) diagramas de secuencia, 3) diagramas de estado[16] y 4) prototipos de la interfaz de usuario.

En muchas instancias, basta un conjunto de casos de uso para describir la interacción en el nivel del análisis (durante el diseño se introducirán más mejoras y detalles). Sin embargo, cuando la secuencia de interacción es compleja e involucra múltiples clases de análisis o muchas tareas, es conveniente ilustrarla de forma más rigurosa mediante un diagrama.

El formato de la interfaz de usuario, el contenido que presenta, los mecanismos de interacción que implementa y la estética general de las conexiones entre el usuario y la *webapp* tienen mucho que ver con la satisfacción de éste y con el éxito conjunto del software. Aunque se afirme que la creación de un prototipo de interfaz de usuario es una actividad de diseño, es una buena idea llevarla a cabo durante la creación del modelo de análisis. Entre más pronto se revise la representación física de la interfaz de usuario, más probable es que los consumidores finales obtengan lo que desean. En el capítulo 11 se estudia con detalle el diseño de interfaces de usuario.

16 Los diagramas de secuencia y los de estado se modelan con el empleo de notación UML. Los diagramas de estado se describen en la sección 7.3. Para mayores detalles, consulte el apéndice 1.

Como hay muchas herramientas para construir *webapps* baratas y poderosas en sus funciones, es mejor crear el prototipo de la interfaz con el empleo de ellas. El prototipo debe implementar los vínculos de navegación principales y representar la pantalla general en forma muy parecida a la que se construirá. Por ejemplo, si van a ponerse a disposición del usuario final cinco funciones principales del sistema, el prototipo debe representarlas tal como las verá cuando entre por primera vez a la *webapp*. ¿Se darán vínculos gráficos? ¿Dónde se desplegará el menú de navegación? ¿Qué otra información verá el usuario? Preguntas como éstas son las que debe responder el prototipo.

7.5.6 Modelo funcional para las *webapps*

Muchas *webapps* proporcionan una amplia variedad de funciones de computación y manipulación que se asocian directamente con el contenido (porque lo utilizan o porque lo producen) y es frecuente que sean un objetivo importante de la interacción entre el usuario y la *webapp*. Por esta razón, deben analizarse los requerimientos funcionales y modelarlos cuando sea necesario.

El *modelo funcional* enfrenta dos elementos de procesamiento de la *webapp*, cada uno de los cuales representa un nivel distinto de abstracción del procedimiento: 1) funciones observables por los usuarios que entrega la *webapp* a éstos y 2) las operaciones contenidas en las clases de análisis que implementan comportamientos asociados con la clase.

La funcionalidad observable por el usuario agrupa cualesquiera funciones de procesamiento que inicie directamente el usuario. Por ejemplo, una *webapp* financiera tal vez implemente varias funciones de finanzas (como una calculadora de ahorros para una colegiatura universitaria o un fondo para el retiro). Estas funciones en realidad se implementan con el uso de operaciones dentro de clases de análisis, pero desde el punto de vista del usuario final; el resultado visible es la función (más correctamente, los datos que provee la función).

En un nivel más bajo de abstracción del procedimiento, el modelo de requerimientos describe el procesamiento que se realizará por medio de operaciones de clase de análisis. Estas operaciones manipulan los atributos de clase y se involucran como clases que colaboran entre sí para lograr algún comportamiento que se desea.

Sin que importe el nivel de abstracción del procedimiento, el diagrama de actividades UML se utiliza para representar detalles de éste. En el nivel de análisis, los diagramas de actividades deben usarse sólo donde la funcionalidad sea relativamente compleja. Gran parte de la complejidad de muchas *webapps* ocurre no en las funciones que proveen, sino en la naturaleza de la información a que se accede y en las formas en las que se manipula.

Un ejemplo de complejidad relativa de la funcionalidad para **CasaSeguraAsegurada.com** se aborda en un caso de uso llamado *Obtener recomendaciones para la distribución de sensores en mi espacio*. El usuario ya ha desarrollado la distribución del espacio que se vigilará y en este caso de uso selecciona dicha distribución y solicita ubicaciones recomendables para los sensores dentro de ella. **CasaSeguraAsegurada.com** responde con la representación gráfica de la distribución por medio de información adicional acerca de la ubicación recomendable para los sensores. La interacción es muy sencilla, el contenido es algo más complejo, pero la funcionalidad subyacente es muy sofisticada. El sistema debe realizar un análisis relativamente complejo de la planta del piso para determinar el conjunto óptimo de sensores. Debe examinar las dimensiones de la habitación, la ubicación de puertas y ventanas, y coordinar éstas con la capacidad y especificaciones de los sensores. ¡No es una tarea fácil! Para describir el procesamiento de este caso de uso se utiliza un conjunto de diagramas de actividades.

El segundo ejemplo es el caso de uso *Controlar cámaras*. En éste, la interacción es relativamente sencilla, pero existe el potencial de una funcionalidad compleja, dado que dicha operación "sencilla" requiere una comunicación compleja con dispositivos ubicados en posiciones remotas y a los que se accede por internet. Una complicación adicional se relaciona con la ne-

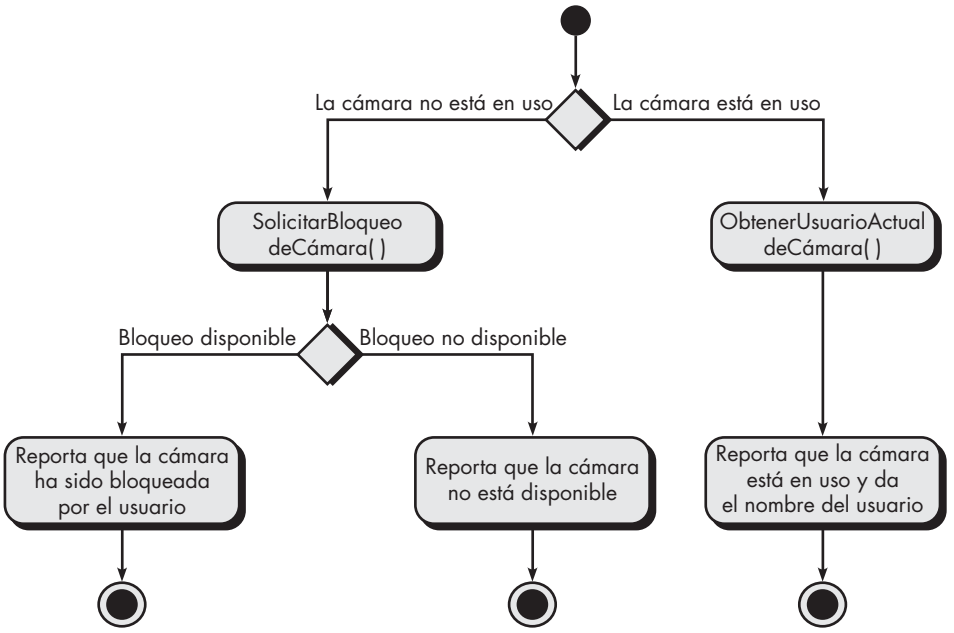


FIGURA 7.11
Diagrama de actividades para la operación *TomarControl deCámara( )*

gociación del control cuando varias personas autorizadas tratan de vigilar o controlar un mismo sensor al mismo tiempo.

La figura 7.11 ilustra el diagrama de actividades para la operación *TomarControldeCámara* que forma parte de la clase de análisis **Cámara** usada dentro del caso de uso *Controlar cámaras*. Debe observarse que con el flujo de procedimiento se invocan dos operaciones adicionales: *SolicitarBloqueodeCámara ( )*, que trata de bloquear la cámara para este usuario, y *ObtenerUsuarioActualdeCámara ( )*, que recupera el nombre del usuario que controla en ese momento la cámara. Los detalles de construcción indican cómo se invocan estas operaciones, y los de la interfaz para cada operación no se señalan hasta que comienza el diseño de la *webapp*.

7.5.7 Modelos de configuración para las *webapps*

En ciertos casos, el modelo de configuración no es sino una lista de atributos del lado del servidor y del lado del cliente. Sin embargo, para *webapps* más complejas, son varias las dificultades de configuración (por ejemplo, distribuir la carga entre servidores múltiples, arquitecturas caché, bases de datos remotas, distintos servidores que atienden a varios objetos en la misma página web, etc.) que afectan el análisis y diseño. El *diagrama de despliegue UML* se utiliza en situaciones en las que deben considerarse arquitecturas de configuración compleja.

Para **CasaSeguraAsegurada.com**, deben especificarse el contenido y funcionalidad públicos a fin de que sean accesibles a través de todos los clientes principales de web (como aquéllos con 1 por ciento o más de participación en el mercado).[17] A la inversa, es aceptable restringir las funciones más complejas de control y vigilancia (que sólo es accesible para los usuarios tipo **Propietario**) a un conjunto más pequeño de clientes. El modelo de configuración para **CasaSeguraAsegurada.com** también especificará la operación cruzada con las bases de datos de productos y aplicaciones de vigilancia.

17 La determinación de la participación en el mercado para los navegadores es notoriamente problemática y varía en función de cuál fuente se utilice. No obstante, en el momento de escribir este libro, Internet Explorer y Firefox eran los únicos que sobrepasaban 30 por ciento, y Mozilla, Opera y Safari los únicos que superaban de manera consistente 1 por ciento.

7.5.8 Modelado de la navegación

Para modelar la navegación se considera cómo navegará cada categoría de usuario de un elemento de la *webapp* (como un objeto de contenido) a otro. La mecánica de navegación se define como parte del diseño. En esa etapa debe centrarse la atención en los requerimientos generales de navegación. Deben considerarse las preguntas siguientes:

- ¿Ciertos elementos deben ser más fáciles de alcanzar (requieren menos pasos de navegación) que otros? ¿Cuál es la prioridad de presentación?
- ¿Debe ponerse el énfasis en ciertos elementos para forzar a los usuarios a navegar en esa dirección?
- ¿Cómo deben manejarse los errores en la navegación?
- ¿Debe darse prioridad a la navegación hacia grupos de elementos relacionados y no hacia un elemento específico?
- ¿La navegación debe hacerse por medio de vínculos, acceso basado en búsquedas o por otros medios?
- ¿Debe presentarse a los usuarios ciertos elementos con base en el contexto de acciones de navegación previas?
- ¿Debe mantenerse un registro de usuarios de la navegación?
- ¿Debe estar disponible un mapa completo de la navegación (en oposición a un solo vínculo para "regresar" o un apuntador dirigido) en cada punto de la interacción del usuario?
- ¿El diseño de la navegación debe estar motivado por los comportamientos del usuario más comunes y esperados o por la importancia percibida de los elementos definidos de la *webapp*?
- ¿Un usuario puede "guardar" su navegación previa a través de la *webapp* para hacer expedito el uso futuro?
- ¿Para qué categoría de usuario debe diseñarse la navegación óptima?
- ¿Cómo deben manejarse los vínculos externos hacia la *webapp*? ¿Con la superposición de la ventana del navegador existente? ¿Como nueva ventana del navegador? ¿En un marco separado?

Estas preguntas y muchas otras deben plantearse y responderse como parte del análisis de la navegación.

Usted y otros participantes también deben determinar los requerimientos generales para la navegación. Por ejemplo, ¿se dará a los usuarios un "mapa del sitio" y un panorama de toda la estructura de la *webapp*? ¿Un usuario puede hacer una "visita guiada" que resalte los elementos más importantes (objetos y funciones de contenido) con que se disponga? ¿Podrá acceder un usuario a los objetos o funciones de contenido con base en atributos definidos de dichos elementos (por ejemplo, un usuario tal vez desee acceder a todas las fotografías de un edificio específico o a todas las funciones que permiten calcular el peso)?

7.6 Resumen

Los modelos orientados al flujo se centran en el flujo de objetos de datos a medida que son transformados por las funciones de procesamiento. Derivados del análisis estructurado, los modelos orientados al flujo usan el diagrama de flujo de datos, notación de modelación que ilustra la manera en la que se transforma la entrada en salida cuando los objetos de datos se mueven a través del sistema. Cada función del software que transforme datos es descrita por la

especificación o narrativa de un proceso. Además del flujo de datos, este elemento de modelación también muestra el flujo del control, representación que ilustra cómo afectan los eventos al comportamiento de un sistema.

El modelado del comportamiento ilustra el comportamiento dinámico. El modelo de comportamiento utiliza una entrada basada en el escenario, orientada al flujo y elementos basados en clases para representar los estados de las clases de análisis y al sistema como un todo. Para lograr esto, se identifican los estados y se definen los eventos que hacen que una clase (o el sistema) haga una transición de un estado a otro, así como las acciones que ocurren cuando se efectúa dicha transición. Los diagramas de estado y de secuencia son la notación que se emplea para modelar el comportamiento.

Los patrones de análisis permiten a un ingeniero de software utilizar el conocimiento del dominio existente para facilitar la creación de un modelo de requerimientos. Un patrón de análisis describe una característica o función específica del software que puede describirse con un conjunto coherente de casos de uso. Especifica el objetivo del patrón, la motivación para su uso, las restricciones que limitan éste, su aplicabilidad en distintos dominios de problemas, la estructura general del patrón, su comportamiento y colaboraciones, así como información suplementaria.

El modelado de los requerimientos para las *webapps* utiliza la mayoría, si no es que todos, los elementos de modelado que se estudian en el libro. Sin embargo, dichos elementos se aplican dentro de un conjunto de modelos especializados que se abocan al contenido, interacción, función, navegación y configuración cliente-servidor en la que reside la *webapp*.

Problemas y puntos por evaluar

7.1. ¿Cuál es la diferencia fundamental entre el análisis estructurado y las estrategias orientadas a objetos para hacer el análisis de los requerimientos?

7.2. En un diagrama de flujo de datos, ¿una flecha representa un flujo del control u otra cosa?

7.3. ¿Qué es la "continuidad del flujo de información" y cómo se aplica cuando se mejora el diagrama de flujo de datos?

7.4. ¿Cómo se utiliza el análisis gramatical en la creación de un DFD?

7.5. ¿Qué es una especificación del control?

7.6. ¿Son lo mismo una PSPEC y un caso de uso? Si no es así, explique las diferencias.

7.7. Hay dos tipos diferentes de "estados" que los modelos del comportamiento pueden representar. ¿Cuáles son?

7.8. ¿En qué difiere un diagrama de secuencia de un diagrama de estado? ¿En qué se parecen?

7.9. Sugiera tres patrones de requerimientos para un teléfono inalámbrico moderno y escriba una descripción breve de cada uno. ¿Estos patrones podrían usarse para otros equipos? Dé un ejemplo.

7.10. Seleccione uno de los patrones desarrollados en el problema 7.9 y desarrolle una descripción del patrón razonablemente completa, similar en contenido y estilo a la que se presentó en la sección 7.4.2.

7.11. ¿Cuánto modelado del análisis piensa que se requeriría para **CasaSeguraAsegurada.com**? ¿Se necesitaría cada uno de los tipos de modelo descritos en la sección 7.5.3?

7.12. ¿Cuál es el propósito del modelo de interacción para una *webapp*?

7.13. Un modelo funcional de *webapp* debe retrasarse hasta el diseño. Diga los pros y contras de este argumento.

7.14. ¿Cuál es el propósito de un modelo de configuración?

7.15. ¿En qué difiere el modelo de navegación del modelo de interacción?

Lecturas adicionales y fuentes de información

Se han publicado decenas de libros sobre el análisis estructurado. Todos cubren el tema de manera adecuada, pero algunos son excelentes. DeMarco y Plauger escribieron un clásico (*Structured Analysis and System Specification*, Pearson, 1985) que sigue siendo una buena introducción a la notación básica. Los libros escritos por Kendall y Kendall (*Systems Analysis and Design*, 5a. ed., Prentice-Hall, 2002), Hoffer *et al*. (*Modern Systems Analysis and Design*, Addison-Wesley, 3a. ed., 2001), Davis y Yen (*The Information System Consultant's Handbook: Systems Analysis and Design*, CRC Press, 1998) y Modell (*A Professional's Guide to Systems Analysis*, 2a. ed., McGraw-Hill, 1996) son buenas referencias. El escrito por Yourdon (*Modern Structured Analysis*, Yourdon-Press, 1989) sobre el tema está entre las fuentes más exhaustivas publicadas hasta la fecha.

El modelado del comportamiento presenta un punto de vista dinámico e importante del comportamiento de un sistema. Los libros de Wagner *et al.* (*Modeling Software with Finite State Machines: A Practical Approach*, Auerbach, 2006) y Boerger y Staerk (*Abstract State Machines*, Springer, 2003) presentan un análisis completo de los diagramas de estado y de otras representaciones del comportamiento.

La mayoría de textos escritos sobre patrones de software se centran en el diseño de éste. Sin embargo, los libros de Evans (*Domain-Driven Design*, Addison-Wesley, 2003) y Fowler ([Fow03] y ([Fow97]) abordan específicamente los patrones de análisis.

Pressman y Lowe presentan un tratamiento profundo del modelado del análisis para *webapps* [Pre08]. Los artículos contenidos dentro de una antología editada por Murugesan y Desphande (*Web Engineering: Managing Diversity and Complexity of Web Application Development*, Springer, 2001) analizan distintos aspectos de los requerimientos para las *webapps*. Además, la edición anual de *Proceedings of the International Conference on Web Engineering* analiza en forma regular aspectos del modelado de los requerimientos.

En internet hay una amplia variedad de fuentes de información sobre modelado de los requerimientos. En el sitio web del libro, **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, se encuentra una lista actualizada de referencias que hay en la red mundial, relevantes para el modelado del análisis.

CAPÍTULO

8

CONCEPTOS DE DISEÑO

CONCEPTOS CLAVE

abstracción 189
arquitectura 190
aspectos 194
atributos de la calidad 187
buen diseño 187
cohesión 193
diseño de datos 199
diseño del software 188
diseño orientado a objeto . . 195
división de problemas 191
independencia funcional 193
lineamientos de la calidad . . 186
modularidad 191
ocultamiento de información 192
patrones 191
proceso de diseño 186
rediseño 195
refinamiento 194

El diseño de software agrupa el conjunto de principios, conceptos y prácticas que llevan al desarrollo de un sistema o producto de alta calidad. Los principios de diseño establecen una filosofía general que guía el trabajo de diseño que debe ejecutarse. Deben entenderse los conceptos de diseño antes de aplicar la mecánica de éste, y la práctica del diseño en sí lleva a la creación de distintas representaciones del software que sirve como guía para la actividad de construcción que siga.

El diseño es crucial para el éxito de la ingeniería de software. A principios de la década de 1990, Mitch Kapor, creador de Lotus 1-2-3, publicó en *Dr. Dobbs Journal* un "manifiesto del diseño de software". Decía lo siguiente:

> ¿Qué es el diseño? Es donde se está con un pie en dos mundos —el de la tecnología y el de las personas y los propósitos humanos— que tratan de unificarse...
>
> Vitruvio, romano crítico de arquitectura, afirmaba que los edificios bien diseñados eran aquellos que tenían resistencia, funcionalidad y belleza. Lo mismo se aplica al buen software. *Resistencia:* un programa no debe tener ningún error que impida su funcionamiento. *Funcionalidad:* un programa debe se apropiado para los fines que persigue. *Belleza:* la experiencia de usar el programa debe ser placentera. Éstos son los comienzos de una teoría del diseño de software.

El objetivo del diseño es producir un modelo o representación que tenga resistencia, funcionalidad y belleza. Para lograrlo, debe practicarse la diversificación y luego la convergencia. Belady [Bel81] afirma que "la diversificación es la adquisición de un repertorio de alternativas, materia prima del diseño: componentes, soluciones con los componentes y conocimiento, todo lo cual

UNA MIRADA RÁPIDA

¿Qué es? El diseño es lo que casi todo ingeniero quiere hacer. Es el lugar en el que las reglas de la creatividad —los requerimientos de los participantes, las necesidades del negocio y las consideraciones técnicas— se unen para formular un producto o sistema. El diseño crea una representación o modelo del software, pero, a diferencia del modelo de los requerimientos (que se centra en describir los datos que se necesitan, la función y el comportamiento), el modelo de diseño proporciona detalles sobre arquitectura del software, estructuras de datos, interfaces y componentes que se necesitan para implementar el sistema.

¿Quién lo hace? Ingenieros de software llevan a cabo cada una de las tareas del diseño.

¿Por qué es importante? El diseño permite modelar el sistema o producto que se va a construir. Este modelo se evalúa respecto de la calidad y su mejora antes de generar código; después, se efectúan pruebas y se involucra a muchos usuarios finales. El diseño es el lugar en el que se establece la calidad del software.

¿Cuáles son los pasos? El diseño representa al software de varias maneras. En primer lugar, debe representarse la arquitectura del sistema o producto. Después se modelan las interfaces que conectan al software con los usuarios finales, con otros sistemas y dispositivos, y con sus propios componentes constitutivos. Por último, se diseñan los componentes del software que se utilizan para construir el sistema. Cada una de estas perspectivas representa una acción de diseño distinta, pero todas deben apegarse a un conjunto básico de conceptos de diseño que guíe el trabajo de producción de software.

¿Cuál es el producto final? El trabajo principal que se produce durante el diseño del software es un modelo de diseño que agrupa las representaciones arquitectónicas, interfaces en el nivel de componente y despliegue.

¿Cómo me aseguro de que lo hice bien? El modelo de diseño es evaluado por el equipo de software en un esfuerzo por determinar si contiene errores, inconsistencias u omisiones, si existen mejores alternativas y si es posible implementar el modelo dentro de las restricciones, plazo y costo que se hayan establecido.

está contenido en catálogos, libros de texto y en la mente". Una vez que se reúne este conjunto diversificado de información, deben escogerse aquellos elementos del repertorio que cumplan los requerimientos definidos por la ingeniería y por el modelo de análisis (capítulos 5 a 7). A medida que esto ocurre, se evalúan las alternativas, algunas se rechazan, se converge en "una configuración particular de componentes y, con ello, en la creación del producto final" [Bel81].

La diversificación y la convergencia combinan la intuición y el criterio con base en la experiencia en la construcción de entidades similares, un conjunto de principios heurísticos que guían la forma en la que evoluciona el modelo, un conjunto de criterios que permiten evaluar la calidad y un proceso iterativo que finalmente conduce a una representación del diseño definitivo.

El diseño del software cambia continuamente conforme evolucionan los nuevos métodos, surgen mejores análisis y se obtiene una comprensión más amplia.[1] Incluso hoy, la mayor parte de las metodologías de diseño de software carece de profundidad, flexibilidad y naturaleza cuantitativa, que normalmente se asocian con las disciplinas de diseño de ingeniería más clásicas. No obstante, sí existen métodos para diseñar software, se dispone de criterios para el diseño con calidad y se aplica la notación del diseño. En este capítulo, se estudian los conceptos y principios fundamentales aplicables a todo el diseño de software, los elementos del modelo del diseño y el efecto que tienen los patrones en el proceso de diseño. En los capítulos 9 a 13 se presentarán varias metodologías de diseño de software, según se aplican en la obtención de arquitecturas e interfaces en el nivel de componente, así como a enfoques de diseño basados en patrones y orientados a web.

8.1 Diseño en el contexto de la ingeniería de software

Cita:

"El milagro más común de la ingeniería de software es la transición del análisis al diseño y de éste al código."

Richard Due'

El diseño de software se ubica en el área técnica de la ingeniería de software y se aplica sin importar el modelo del proceso que se utilice. El diseño del software comienza una vez que se han analizado y modelado los requerimientos, es la última acción de la ingeniería de software dentro de la actividad de modelado y prepara la etapa de **construcción** (generación y prueba de código).

Cada uno de los elementos del modelo de requerimientos (capítulos 6 y 7) proporciona información necesaria para crear los cuatro modelos de diseño necesarios para la especificación completa del diseño. En la figura 8.1 se ilustra el flujo de la información durante el diseño del software. El trabajo de diseño es alimentado por el modelo de requerimientos, manifestado por elementos basados en el escenario, en la clase, orientados al flujo, y del comportamiento. El empleo de la notación y de los métodos de diseño estudiados en los últimos capítulos produce diseños de los datos o clases, de la arquitectura, de la interfaz y de los componentes.

CONSEJO

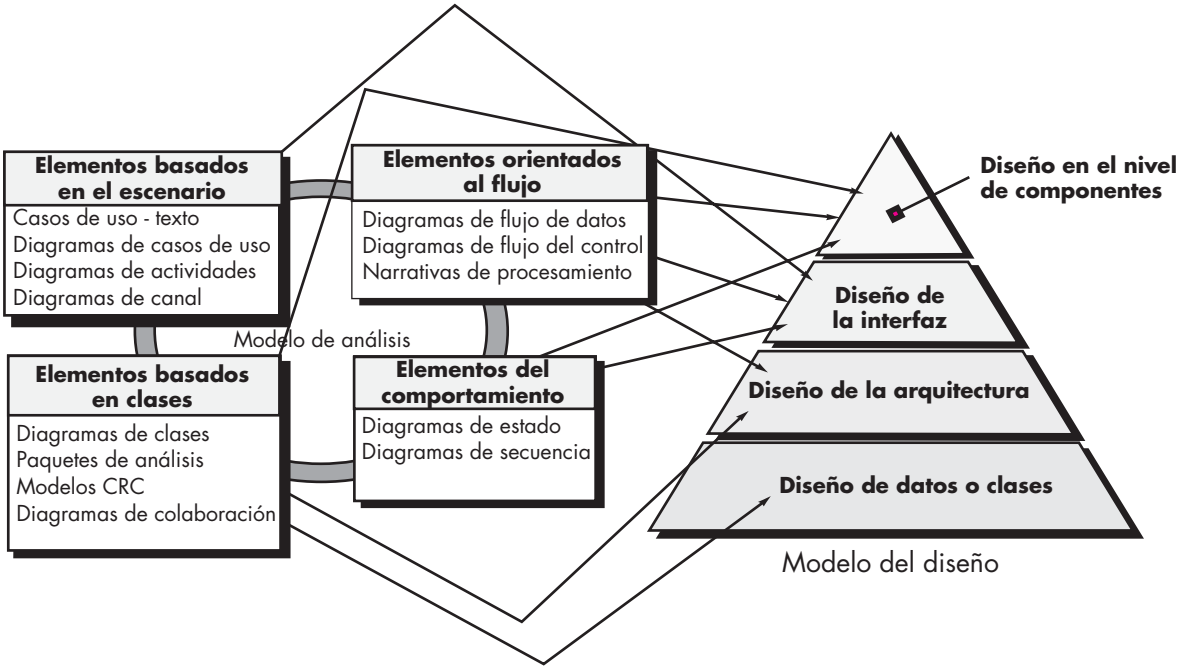
El diseño del software siempre debe comenzar con el análisis de los datos, pues son el fundamento de todos los demás elementos del diseño. Una vez obtenido el fundamento, se obtiene la arquitectura. Sólo entonces deben realizarse otros trabajos del diseño.

El diseño de datos o clases transforma los modelos de clases (capítulo 6) en realizaciones de clases de diseño y en las estructuras de datos que se requieren para implementar el software. Los objetos y relaciones definidos en el diagrama CRC y el contenido detallado de los datos ilustrado por los atributos de clase y otros tipos de notación dan la base para el diseño de los datos. Parte del diseño de clase puede llevarse a cabo junto con el diseño de la arquitectura del software. Un diseño más detallado de las clases tiene lugar cuando se diseña cada componente del software.

El diseño de la arquitectura define la relación entre los elementos principales de la estructura del software, los estilos y patrones de diseño de la arquitectura que pueden usarse para alcanzar

1 Aquellos lectores interesados en la filosofía del diseño de software pueden consultar el inquietante análisis de Philippe Kruchen sobre el diseño "posmoderno" [Kru05a].

FIGURA 8.1 Traducción del modelo de requerimientos al modelo de diseño



Cita:

"Hay dos formas de construir un diseño del software. Una es hacerlo tan simple que sea obvio que no hay deficiencias y la otra es hacerlo tan complicado que no haya deficiencias obvias. El primer método es mucho más difícil."

C. A. R. Hoare

los requerimientos definidos por el sistema y las restricciones que afectan la forma en la que se implementa la arquitectura [Sha96]. La representación del diseño de la arquitectura —el marco de un sistema basado en computadora— se obtiene del modelo de los requerimientos.

El diseño de la interfaz describe la forma en la que el software se comunica con los sistemas que interactúan con él y con los humanos que lo utilizan. Una interfaz implica un flujo de información (por ejemplo, datos o control) y un tipo específico de comportamiento. Entonces, los modelos de escenarios de uso y de comportamiento dan mucha de la información requerida para diseñar la interfaz.

El diseño en el nivel de componente transforma los elementos estructurales de la arquitectura del software en una descripción de sus componentes en cuanto a procedimiento. La información obtenida a partir de los modelos basados en clase, flujo y comportamiento sirve como la base para diseñar los componentes.

Durante el diseño se toman decisiones que en última instancia afectarán al éxito de la construcción del software y, de igual importancia, a la facilidad con la que puede darse mantenimiento al software. Pero, ¿por qué es tan importante el diseño?

La importancia del diseño del software se resume en una palabra: *calidad*. El diseño es el sitio en el que se introduce calidad en la ingeniería de software. Da representaciones del software que pueden evaluarse en su calidad. Es la única manera de traducir con exactitud a un producto o sistema terminado los requerimientos de los participantes. Es el fundamento de toda la ingeniería de software y de las actividades que dan el apoyo que sigue. Sin diseño se corre el riesgo de obtener un sistema inestable, que falle cuando se hagan cambios pequeños, o uno que sea difícil de someter a prueba, o en el que no sea posible evaluar la calidad hasta que sea demasiado tarde en el proceso de software, cuando no queda mucho tiempo y ya se ha gastado mucho dinero.

CasaSegura

Diseño versus *codificación*

La escena: El cubículo de Jamie, cuando el equipo se prepara para traducir a diseño los requerimientos.

Participantes: Jamie, Vinod y Ed, miembros del equipo de ingeniería de software para *CasaSegura*.

La conversación:

Jamie: Ustedes saben, Doug [el gerente del equipo] está obsesionado con el diseño. Tengo que ser honesto, lo que realmente amo es codificar. Denme C++ o Java y soy feliz.

Ed: No… te gusta diseñar.

Jamie: No me estás escuchando; codificar es lo mío.

Vinod: Creo que Ed quiere decir que en realidad no es codificar lo que te gusta; te gusta diseñar y expresarlo en código. El código es el lenguaje que usas para representar el diseño.

Jamie: ¿Y qué tiene de malo?

Vinod: El nivel de abstracción.

Jamie: ¿Qué?

Ed: Un lenguaje de programación es bueno para representar detalles tales como estructuras de datos y algoritmos, pero no es tan bueno para representar la arquitectura o la colaboración entre componentes… algo así.

Vinod: Y una arquitectura complicada arruina al mejor código.

Jamie (piensa unos momentos): Entonces, dicen que no puede representarse la arquitectura con código... eso no es cierto.

Vinod: Claro que es posible implicar la arquitectura con el código, pero en la mayor parte de lenguajes de programación, es muy difícil lograr un panorama rápido y amplio de la arquitectura con el análisis del código.

Ed: Y eso es lo que queremos hacer antes de empezar a codificar.

Jamie: Está bien, tal vez diseñar y codificar sean cosas distintas, pero aún así me gusta más codificar.

8.2 El proceso de diseño

El diseño de software es un proceso iterativo por medio del cual se traducen los requerimientos en un "plano" para construir el software. Al principio, el plano ilustra una visión holística del software. Es decir, el diseño se representa en un nivel alto de abstracción, en el que se rastrea directamente el objetivo específico del sistema y los requerimientos más detallados de datos, funcionamiento y comportamiento. A medida que tienen lugar las iteraciones del diseño, las mejoras posteriores conducen a niveles menores de abstracción. Éstos también pueden rastrearse hasta los requerimientos, pero la conexión es más sutil.

8.2.1 Lineamientos y atributos de la calidad del software

Cita:

"…escribir un fragmento inteligente de código que funcione es una cosa; diseñar algo que dé apoyo a largo plazo a una empresa es otra muy diferente".

C. Ferguson

A través del proceso de diseño se evalúa la calidad de éste de acuerdo con la serie de revisiones técnicas que se estudia en el capítulo 15. McGlaughlin [McG91] sugiere tres características que funcionan como guía para evaluar un buen diseño:

- Debe implementar todos los requerimientos explícitos contenidos en el modelo de requerimientos y dar cabida a todos los requerimientos implícitos que desean los participantes.
- Debe ser una guía legible y comprensible para quienes generan el código y para los que lo prueban y dan el apoyo posterior.
- Debe proporcionar el panorama completo del software, y abordar los dominios de los datos, las funciones y el comportamiento desde el punto de vista de la implementación.

En realidad, cada una de estas características es una meta del proceso de diseño. Pero, ¿cómo se logran?

Lineamientos de la calidad. A fin de evaluar la calidad de una representación del diseño, usted y otros miembros del equipo de software deben establecer los criterios técnicos de un buen diseño. En la sección 8.3 se estudian conceptos de diseño que también sirven como crite-

rios de calidad del software. En este momento, considere los siguientes lineamientos para el diseño:

¿Cuáles son las características de un buen diseño?

1. Debe tener una arquitectura que 1) se haya creado con el empleo de estilos o patrones arquitectónicos reconocibles, 2) esté compuesta de componentes con buenas características de diseño (éstas se analizan más adelante, en este capítulo), y 3) se implementen en forma evolutiva,[2] de modo que faciliten la implementación y las pruebas.
2. Debe ser modular, es decir, el software debe estar dividido de manera lógica en elementos o subsistemas.
3. Debe contener distintas representaciones de datos, arquitectura, interfaces y componentes.
4. Debe conducir a estructuras de datos apropiadas para las clases que se van a implementar y que surjan de patrones reconocibles de datos.
5. Debe llevar a componentes que tengan características funcionales independientes.
6. Debe conducir a interfaces que reduzcan la complejidad de las conexiones entre los componentes y el ambiente externo.
7. Debe obtenerse con el empleo de un método repetible motivado por la información obtenida durante el análisis de los requerimientos del software.
8. Debe representarse con una notación que comunique con eficacia su significado.

Estos lineamientos de diseño no se logran por azar. Se consiguen con la aplicación de los principios de diseño fundamentales, una metodología sistemática y con revisión.

INFORMACIÓN

Evaluación de la calidad del diseño. La revisión técnica

El diseño es importante porque permite que un equipo de software evalúe la calidad[3] de éste antes de que se implemente, momento en el que es fácil y barato corregir errores, omisiones o inconsistencias. Pero, ¿cómo se evalúa la calidad durante el diseño? El software no puede someterse a prueba porque no hay nada ejecutable. ¿Qué hacer?

Durante el diseño, la calidad se evalúa por medio de la realización de una serie de revisiones técnicas (RT). Las RT se estudian con detalle en el capítulo 15,[4] pero es útil hacer un resumen de dicha técnica en este momento. Una revisión técnica es una reunión celebrada por miembros del equipo de software. Por lo general, participan dos, tres o cuatro personas, en función del alcance de la información del diseño que se revisará. Cada persona tiene un papel: el *líder de la revisión* planea la reunión, establece la agenda y coordina la junta; el *secretario* toma notas para que no se pierda nada; el *productor* es la persona cuyo trabajo (por ejemplo, el diseño de un componente del software) se revisa. Antes de la reunión, se entrega a cada persona del equipo una copia del producto del trabajo de diseño y se le pide que la lea y que busque errores, omisiones o ambigüedades. El objetivo al comenzar la reunión es detectar todos los problemas del producto, de modo que puedan corregirse antes de que comience la implementación. Es común que la RT dure entre 90 minutos y 2 horas. Al final de ella, el equipo de revisión determina si se requiere de otras acciones por parte del productor a fin de que se apruebe el producto como porción del modelo del diseño final.

Cita:

"La calidad no es algo que se deje arriba de los sujetos y objetos como si fuera el remate de un árbol de navidad."

Robert Pirsig

Atributos de la calidad. Hewlett-Packard [Gra87] desarrolló un conjunto de atributos de la calidad del software a los que se dio el acrónimo FURPS: funcionalidad, usabilidad, confiabilidad, rendimiento y mantenibilidad. Los atributos de calidad FURPS representan el objetivo de todo diseño de software:

2 Para sistemas pequeños, en ocasiones el diseño puede desarrollarse en forma lineal.

3 Los factores de calidad que se estudian en el capítulo 23 ayudan al equipo de revisión cuando evalúa aquélla.

4 Tal vez el lector considere oportuno revisar el capítulo 15 en este momento. Las revisiones técnicas son una parte crítica del proceso de diseño y un mecanismo importante para lograr su calidad.

CONSEJO

Los diseñadores del software tienden a centrarse en el problema que se va a resolver. No olvide que los atributos FURPS siempre forman parte del problema. Deben tomarse en cuenta.

- La *funcionalidad* se califica de acuerdo con el conjunto de características y capacidades del programa, la generalidad de las funciones que se entregan y la seguridad general del sistema.
- La *usabilidad* se evalúa tomando en cuenta factores humanos (véase el capítulo 11), la estética general, la consistencia y la documentación.
- La *confiabilidad* se evalúa con la medición de la frecuencia y gravedad de las fallas, la exactitud de los resultados que salen, el tiempo medio para que ocurra una falla (TMPF), la capacidad de recuperación ante ésta y lo predecible del programa.
- El *rendimiento* se mide con base en la velocidad de procesamiento, el tiempo de respuesta, el uso de recursos, el conjunto y la eficiencia.
- La *mantenibilidad* combina la capacidad del programa para ser ampliable (extensibilidad), adaptable y servicial (estos tres atributos se denotan con un término más común: *mantenibilidad*), y además que pueda probarse, ser compatible y configurable (capacidad de organizar y controlar los elementos de la configuración del software, véase el capítulo 22) y que cuente con la facilidad para instalarse en el sistema y para que se detecten los problemas.

No todo atributo de la calidad del software se pondera por igual al diseñarlo. Una aplicación tal vez se aboque a lo funcional con énfasis en la seguridad. Otra quizá busque rendimiento con la mira puesta en la velocidad de procesamiento. En una tercera se persigue la confiabilidad. Sin importar la ponderación, es importante observar que estos atributos de la calidad deben tomarse en cuenta cuando comienza el diseño, *no* cuando haya terminado éste y la construcción se encuentre en marcha.

8.2.2 La evolución del diseño del software

Cita:

"Un diseñador sabe que alcanzó la perfección no cuando no hay nada por agregar, sino cuando no hay nada que quitar."

Antoine de Saint-Exupery

La evolución del diseño del software es un proceso continuo que ya ha cubierto casi seis décadas. Los primeros trabajos de diseño se concentraban en criterios para el desarrollo de programas modulares [Den73] y en métodos para mejorar estructuras de software con un enfoque de arriba abajo [Wir71]. Los aspectos de procedimiento del diseño evolucionaron hacia una filosofía llamada *programación estructurada* [Dah72], [Mil72]. Los trabajos posteriores propusieron métodos para traducir el flujo de datos [Ste74] o la estructura de éstos (por ejemplo, [Jac75], [War74]) a una definición de diseño. Los enfoques más nuevos (por ejemplo, [Jac92], [Gam95]) propusieron un enfoque orientado a objeto para diseñar derivaciones. En los últimos tiempos, el énfasis al desarrollar software se pone en la arquitectura de éste [Kru06] y en los patrones de diseño susceptibles de emplearse para implementar arquitecturas y niveles más bajos de abstracciones del diseño (por ejemplo, [Hol06], [Sha05]). Se da cada vez más importancia a los métodos orientados al aspecto (por ejemplo, [Cla05], [Jac04]), al desarrollo orientado al modelo [Sch06] y a las pruebas [Ast04], que se concentran en llegar a una modularidad eficaz y a la estructura arquitectónica de los diseños que se generan.

¿Qué características son comunes en todos los métodos de diseño?

En la industria del software se aplican varios métodos de diseño, aparte de los ya mencionados. Igual que los métodos de análisis presentados en los capítulos 6 y 7, cada método de diseño de software introduce heurística y notación únicas, así como un punto de vista sobre lo que caracteriza a la calidad en el diseño. No obstante, todos estos métodos tienen algunas características en común: 1) un mecanismo para traducir el modelo de requerimientos en una representación del diseño, 2) una notación para representar las componentes funcionales y sus interfaces, 3) una heurística para mejorar y hacer particiones y 4) lineamientos para evaluar la calidad.

Sin importar el método de diseño que se utilice, debe aplicarse un conjunto de conceptos básicos al diseño en el nivel de datos, arquitectura, interfaz y componente. En las secciones que siguen se estudian estos conceptos.

CONJUNTO DE TAREAS

Conjunto de tareas generales para el diseño

1. Estudiar el modelo del dominio de la información y diseñar las estructuras de datos apropiadas para los objetos de datos y sus atributos.
2. Seleccionar un estilo de arquitectura que sea adecuado para el software con el uso del modelo de análisis.
3. Hacer la partición del modelo de análisis en subsistemas de diseño y asignar éstos dentro de la arquitectura:
   - Asegúrese de que cada subsistema sea cohesivo en sus funciones.
   - Diseñe interfaces del subsistema.
   - Asigne clases de análisis o funciones a cada subsistema.
4. Crear un conjunto de clases de diseño o componentes:
   - Traduzca la descripción de clases de análisis a una clase de diseño.
   - Compare cada clase de diseño con los criterios de diseño; considere los aspectos hereditarios.
   - Defina métodos y mensajes asociados con cada clase de diseño.
   - Evalúe y seleccione patrones de diseño para una clase de diseño o subsistema.
   - Revise las clases de diseño y, si se requiere, modifíquelas.
5. Diseñar cualesquiera interfaces requeridas con sistemas o dispositivos externos.
6. Diseñar la interfaz de usuario.
   - Revise los resultados del análisis de tareas.
   - Especifique la secuencia de acciones con base en los escenarios de usuario.
   - Cree un modelo de comportamiento de la interfaz.
   - Defina los objetos de la interfaz y los mecanismos de control.
   - Revise el diseño de la interfaz y, si se requiere, modifíquelo.
7. Efectuar el diseño en el nivel de componente.
   - Especifique todos los algoritmos en un nivel de abstracción relativamente bajo.
   - Mejore la interfaz de cada componente.
   - Defina estructuras de datos en el nivel de componente.
   - Revise cada componente y corrija todos los errores que se detecten.
8. Desarrollar un modelo de despliegue.

8.3 CONCEPTOS DE DISEÑO

Durante la historia de la ingeniería de software, ha evolucionado un conjunto de conceptos fundamentales sobre su diseño. Aunque con el paso de los años ha variado el grado de interés en cada concepto, todos han soportado la prueba del tiempo. Cada uno da al diseñador del software el fundamento desde el que pueden aplicarse métodos de diseño sofisticados. Todos ayudan a responder las preguntas siguientes:

- ¿Qué criterios se usan para dividir el software en sus componentes individuales?
- ¿Cómo se extraen los detalles de la función o la estructura de datos de la representación conceptual del software?
- ¿Cuáles son los criterios uniformes que definen la calidad técnica de un diseño de software?

M. A. Jackson [Jac75] dijo: "El principio de la sabiduría [para un ingeniero de software] es reconocer la diferencia que hay entre hacer que un programa funcione y lograr que lo haga bien". Los conceptos fundamentales del diseño del software proveen la estructura necesaria para "hacerlo bien".

En las secciones que siguen, se da un panorama breve de los conceptos importantes del diseño de software, tanto del desarrollo tradicional como del orientado a objeto.

Cita:

"La abstracción es uno de los modos fundamentales con los que los humanos luchamos con la complejidad."

Grady Booch

8.3.1 Abstracción

Cuando se considera una solución modular para cualquier problema, es posible plantear muchos niveles de abstracción. En el más elevado se enuncia una solución en términos gruesos con el uso del lenguaje del ambiente del problema. En niveles más bajos de abstracción se da la descripción más detallada de la solución. La terminología orientada al problema se acopla con la que se orienta a la implementación, en un esfuerzo por enunciar la solución. Por último,

en el nivel de abstracción más bajo se plantea la solución, de modo que pueda implementarse directamente.

Cuando se desarrollan niveles de abstracción distintos, se trabaja para crear abstracciones tanto de procedimiento como de datos. Una *abstracción de procedimiento* es una secuencia de instrucciones que tienen una función específica y limitada. El nombre de la abstracción de procedimiento implica estas funciones, pero se omiten detalles específicos. Un ejemplo de esto sería la palabra *abrir*, en el caso de una puerta. *Abrir* implica una secuencia larga de pasos del procedimiento (caminar hacia la puerta, llegar y tomar el picaporte, girar éste y jalar la puerta, retroceder para que la puerta se abra, etcétera).[5]

CONSEJO

Como diseñador, trabaje mucho para obtener abstracciones tanto de procedimiento como de datos que sirvan para el problema en cuestión. Será aún mejor si sirvieran para un dominio completo de problemas.

Una *abstracción de datos* es un conjunto de éstos con nombre que describe a un objeto de datos. En el contexto de la abstracción de procedimiento *abrir*, puede definirse una abstracción de datos llamada **puerta**. Como cualquier objeto de datos, la abstracción de datos para **puerta** agruparía un conjunto de atributos que describirían la puerta (tipo, dirección del abatimiento, mecanismo de apertura, peso, dimensiones, etc.). Se concluye que la abstracción de procedimiento *abrir* usaría información contenida en los atributos de la abstracción de datos **puerta**.

8.3.2 Arquitectura

WebRef

En la dirección **www.sei.cmu.edu/ata/ata_init.html** hay un análisis profundo de la arquitectura del software.

La *arquitectura del software* alude a "la estructura general de éste y a las formas en las que ésta da integridad conceptual a un sistema" [Sha95a]. En su forma más sencilla, la arquitectura es la estructura de organización de los componentes de un programa (módulos), la forma en la que éstos interactúan y la estructura de datos que utilizan. Sin embargo, en un sentido más amplio, los componentes se generalizan para que representen los elementos de un sistema grande y sus interacciones.

Una meta del diseño del software es obtener una aproximación arquitectónica de un sistema. Ésta sirve como estructura a partir de la cual se realizan las actividades de diseño más detalladas. Un conjunto de patrones arquitectónicos permite que el ingeniero de software resuelva problemas de diseño comunes.

Cita:

"Una arquitectura del software es el producto del trabajo de desarrollo que tiene la rentabilidad más alta para una inversión en cuanto a calidad, secuencia de actividades y costo."

Len Bass *et al.*

Shaw y Garlan [Sha95a] describen un conjunto de propiedades que deben especificarse como parte del diseño de la arquitectura:

Propiedades estructurales. Este aspecto de la representación del diseño arquitectónico define los componentes de un sistema (módulos, objetos, filtros, etc.) y la manera en la que están agrupados e interactúan unos con otros. Por ejemplo, los objetos se agrupan para que encapsulen tanto datos como el procedimiento que los manipula e interactúen invocando métodos.

Propiedades extrafuncionales. La descripción del diseño arquitectónico debe abordar la forma en la que la arquitectura del diseño satisface los requerimientos de desempeño, capacidad, confiabilidad, seguridad y adaptabilidad, así como otras características del sistema.

Familias de sistemas relacionados. El diseño arquitectónico debe basarse en patrones repetibles que es común encontrar en el diseño de familias de sistemas similares. En esencia, el diseño debe tener la capacidad de reutilizar bloques de construcción arquitectónica.

CONSEJO

No deje al azar la arquitectura. Si lo hace, pasará el resto del proyecto forzándola para que se ajuste al diseño. Diseñe la arquitectura explícitamente.

Dada la especificación de estas propiedades, el diseño arquitectónico se representa con el uso de uno o más de varios modelos diferentes [Gar95]. Los *modelos estructurales* representan la arquitectura como un conjunto organizado de componentes del programa. Los *modelos de marco* aumentan el nivel de abstracción del diseño, al tratar de identificar patrones de diseño arquitectónico repetibles que se encuentran en tipos similares de aplicaciones. Los *modelos dinámicos* abordan los aspectos estructurales de la arquitectura del programa e indican cómo cambia la

5 Sin embargo, debe notarse que un conjunto de operaciones puede reemplazarse con otro, en tanto la función que implica la abstracción de procedimiento sea la misma. Por tanto, los pasos requeridos para implementar *abrir* cambiarían mucho si la puerta fuera automática y tuviera un sensor.

estructura o la configuración del sistema en función de eventos externos. Los *modelos del proceso* se centran en el diseño del negocio o proceso técnico al que debe dar acomodo el sistema. Por último, los *modelos funcionales* se usan para representar la jerarquía funcional de un sistema.

Para representar estos modelos, se ha desarrollado cierto número de *lenguajes de descripción arquitectónica* (LDA) [Sha95b]. Aunque han sido propuestos muchos LDA diferentes, la mayoría tiene mecanismos para describir los componentes del sistema y la manera en la que se conectan entre sí.

Debe observarse que hay un debate acerca del papel que tiene la arquitectura en el diseño. Algunos investigadores afirman que la obtención de la arquitectura del software debe separarse del diseño y que ocurre entre las acciones de la ingeniería de requerimientos y las del diseño más convencional. Otros piensan que la definición de la arquitectura es parte integral del proceso de diseño. En el capítulo 9 se estudia la forma en la que se caracteriza la arquitectura del software y su papel en el diseño.

Cita:

"Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente, por lo que describe el núcleo de la solución de ese problema, en forma tal que puede usarse ésta un millón de veces sin repetir lo mismo ni una sola vez."

Christopher Alexander

8.3.3 Patrones

Brad Appleton define un *patrón de diseño* de la manera siguiente: "Es una mezcla con nombre propio de puntos de vista que contienen la esencia de una solución demostrada para un problema recurrente dentro de cierto contexto de necesidades en competencia" [App00]. Dicho de otra manera, un patrón de diseño describe una estructura de diseño que resuelve un problema particular del diseño dentro de un contexto específico y entre "fuerzas" que afectan la manera en la que se aplica y en la que se utiliza dicho patrón.

El objetivo de cada patrón de diseño es proporcionar una descripción que permita a un diseñador determinar 1) si el patrón es aplicable al trabajo en cuestión, 2) si puede volverse a usar (con lo que se ahorra tiempo de diseño) y 3) si sirve como guía para desarrollar un patrón distinto en funciones o estructura. En el capítulo 12 se estudian los patrones de diseño.

8.3.4 División de problemas

La *división de problemas* es un concepto de diseño que sugiere que cualquier problema complejo puede manejarse con más facilidad si se subdivide en elementos susceptibles de resolverse u optimizarse de manera independiente. Un *problema* es una característica o comportamiento que se especifica en el modelo de los requerimientos para el software. Al separar un problema en sus piezas más pequeñas y por ello más manejables, se requiere menos esfuerzo y tiempo para resolverlo.

El argumento para separar los problemas puede llevarse demasiado lejos. Si se divide un problema en un número muy grande de problemas muy pequeños, será fácil resolver cada uno de éstos, pero unificarlos en la solución (integración) será muy difícil.

Si para dos problemas, p_1 y p_2, la complejidad que se percibe para p_1 es mayor que la percibida para p_2, entonces se concluye que el esfuerzo requerido para resolver p_1 es mayor que el necesario para resolver p_2. Como caso general, este resultado es intuitivamente obvio. Lleva más tiempo resolver un problema difícil.

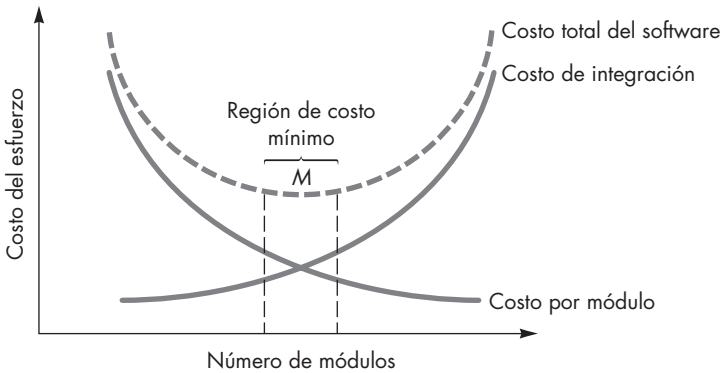
También se concluye que cuando se combinan dos problemas, con frecuencia la complejidad percibida es mayor que la suma de la complejidad tomada por separado. Esto lleva a la estrategia de divide y vencerás, pues es más fácil resolver un problema complejo si se divide en elementos manejables. Esto tiene implicaciones importantes en relación con la modularidad del software.

La división de problemas se manifiesta en otros conceptos de diseño relacionados: modularidad, aspectos, independencia de funcionamiento y mejora. Cada uno de éstos se estudiará en las secciones siguientes.

8.3.5 Modularidad

La modularidad es la manifestación más común de la división de problemas. El software se divide en componentes con nombres distintos y abordables por separado, en ocasiones llamados *módulos*, que se integran para satisfacer los requerimientos del problema.

FIGURA 8.2
Modularidad y costo del software



Se ha dicho que "la modularidad es el único atributo del software que permite que un programa sea manejable en lo intelectual" [Mye78]. El software monolítico (un programa grande compuesto de un solo módulo) no es fácil de entender para un ingeniero de software. El número de trayectorias de control, alcance de referencia, número de variables y complejidad general haría que comprenderlo fuera casi imposible. En función de las circunstancias, el diseño debe descomponerse en muchos módulos con la esperanza de que sea más fácil entenderlos y, en consecuencia, reducir el costo requerido para elaborar el software.

Según el punto de vista de la división de problemas, sería posible concluir que si el software se dividiera en forma indefinida, el esfuerzo requerido para desarrollarlo ¡sería despreciable por pequeño! Desafortunadamente, hay otras fuerzas que entran en juego y que hacen que esta conclusión sea (tristemente) inválida. De acuerdo con la figura 8.2, el esfuerzo (costo) de desarrollar un módulo individual de software disminuye conforme aumenta el número total de módulos. Dado el mismo conjunto de requerimientos, tener más módulos significa tamaños individuales más pequeños. Sin embargo, a medida que se incrementa el número de módulos, el esfuerzo (costo) asociado con su integración también aumenta. Estas características llevan a una curva de costo total como la que se muestra en la figura. Existe un número, *M*, de módulos que arrojarían el mínimo costo de desarrollo, pero no se dispone de las herramientas necesarias para predecir *M* con exactitud.

? ¿Cuál es el número correcto de módulos para un sistema dado?

Las curvas que aparecen en la figura 8.2 constituyen una guía útil al considerar la modularidad. Deben hacerse módulos, pero con cuidado para permanecer en la cercanía de *M*. Debe evitarse hacer pocos o muchos módulos. Pero, ¿cómo saber cuál es la cercanía de *M*? ¿Cuán modular debe hacerse el software? Las respuestas a estas preguntas requieren la comprensión de otros conceptos de diseño que se analizan más adelante en este capítulo.

Debe hacerse un diseño (y el programa resultante) con módulos, de manera que el desarrollo pueda planearse con más facilidad, que sea posible definir y desarrollar los incrementos del software, que los cambios se realicen con más facilidad, que las pruebas y la depuración se efectúen con mayor eficiencia y que el mantenimiento a largo plazo se lleve a cabo sin efectos colaterales de importancia.

8.3.6 Ocultamiento de información

PUNTO CLAVE
El objetivo de ocultar la información es esconder los detalles de las estructuras de datos y el procesamiento tras una interfaz de módulo. No es necesario que los usuarios de éste los conozcan.

El concepto de modularidad lleva a una pregunta fundamental: "¿Cómo descomponer una solución de software para obtener el mejor conjunto de módulos?" El principio del ocultamiento de información sugiere que los módulos se "caractericen por decisiones de diseño que se oculten (cada una) de las demás". En otras palabras, deben especificarse y diseñarse módulos, de forma que la información (algoritmos y datos) contenida en un módulo sea inaccesible para los que no necesiten de ella.

El ocultamiento implica que la modularidad efectiva se logra definiendo un conjunto de módulos independientes que intercambien sólo aquella información necesaria para lograr la función del software. La abstracción ayuda a definir las entidades de procedimiento (o informativas) que constituyen el software. El ocultamiento define y hace cumplir las restricciones de acceso tanto a los detalles de procedimiento como a cualquier estructura de datos local que utilice el módulo [Ros75].

El uso del ocultamiento de información como criterio de diseño para los sistemas modulares proporciona los máximos beneficios cuando se requiere hacer modificaciones durante las pruebas, y más adelante, al dar mantenimiento al software. Debido a que la mayoría de los datos y detalles del procedimiento quedan ocultos para otras partes del software, es menos probable que los errores inadvertidos introducidos durante la modificación se propaguen a distintos sitios dentro del software.

8.3.7 Independencia funcional

El concepto de independencia funcional es resultado directo de la separación de problemas y de los conceptos de abstracción y ocultamiento de información. En escritos cruciales sobre el diseño de software, Wirth [Wir71] y Parnas [Par72] mencionan técnicas de mejora que promueven la independencia modular. Los trabajos posteriores de Stevens, Myers y Constantine [Ste74] dan solidez al concepto.

La independencia funcional se logra desarrollando módulos con funciones "miopes" que tengan "aversión" a la interacción excesiva con otros módulos. Dicho de otro modo, debe diseñarse software de manera que cada módulo resuelva un subconjunto específico de requerimientos y tenga una interfaz sencilla cuando se vea desde otras partes de la estructura del programa. Es lógico preguntar por qué es importante la independencia.

? ¿Por qué debe tratarse de crear módulos independientes?

El software con modularidad eficaz, es decir, con módulos independientes, es más fácil de desarrollar porque su función se subdivide y las interfaces se simplifican (cuando el desarrollo es efectuado por un equipo hay que considerar las ramificaciones). Los módulos independientes son más fáciles de mantener (y probar) debido a que los efectos secundarios causados por el diseño o por la modificación del código son limitados, se reduce la propagación del error y es posible obtener módulos reutilizables. En resumen, la independencia funcional es una clave para el buen diseño y éste es la clave de la calidad del software.

PUNTO CLAVE

La cohesión es un indicador cualitativo del grado en el que un módulo se centra en hacer una sola cosa.

La independencia se evalúa con el uso de dos criterios cualitativos: la cohesión y el acoplamiento. La *cohesión* es un indicador de la fortaleza relativa funcional de un módulo. El *acoplamiento* lo es de la independencia relativa entre módulos.

La cohesión es una extensión natural del concepto de ocultamiento de información descrito en la sección 8.3.6. Un módulo cohesivo ejecuta una sola tarea, por lo que requiere interactuar poco con otros componentes en otras partes del programa. En pocas palabras, un módulo cohesivo debe (idealmente) hacer sólo una cosa. Aunque siempre debe tratarse de lograr mucha cohesión (por ejemplo, una sola tarea), con frecuencia es necesario y aconsejable hacer que un componente de software realice funciones múltiples. Sin embargo, para lograr un buen diseño hay que evitar los componentes "esquizofrénicos" (módulos que llevan a cabo funciones no relacionadas).

PUNTO CLAVE

El acoplamiento es un indicador cualitativo del grado en el que un módulo está conectado con otros y con el mundo exterior.

El acoplamiento es una indicación de la interconexión entre módulos en una estructura de software, y depende de la complejidad de la interfaz entre módulos, del grado en el que se entra o se hace referencia a un módulo y de qué datos pasan a través de la interfaz. En el diseño de software, debe buscarse el mínimo acoplamiento posible. La conectividad simple entre módulos da como resultado un software que es más fácil de entender y menos propenso al "efecto de oleaje" [Ste74], ocasionado cuando ocurren errores en un sitio y se propagan por todo el sistema.

8.3.8 Refinamiento

Existe la tendencia a pasar de inmediato a los detalles e ignorar los pasos del refinamiento. Esto genera errores y hace que el diseño sea mucho más difícil de revisar. Realice refinamiento stepwise.

El refinamiento stepwise es una estrategia de diseño propuesta originalmente por Niklaus Wirth [Wir71]. Un programa se elabora por medio del refinamiento sucesivo de los detalles del procedimiento. Se desarrolla una jerarquía con la descomposición de un enunciado macroscópico de la función (abstracción del procedimiento) en forma escalonada hasta llegar a los comandos del lenguaje de programación.

En realidad, el refinamiento es un proceso de *elaboración*. Se comienza con un enunciado de la función (o descripción de la información), definida en un nivel de abstracción alto. Es decir, el enunciado describe la función o información de manera conceptual, pero no dice nada sobre los trabajos internos de la función o de la estructura interna de la información. Después se trabaja sobre el enunciado original, dando más y más detalles conforme tiene lugar el refinamiento (elaboración) sucesivo.

La abstracción y el refinamiento son conceptos complementarios. La primera permite especificar internamente el procedimiento y los datos, pero elimina la necesidad de que los "extraños" conozcan los detalles de bajo nivel. El refinamiento ayuda a revelar estos detalles a medida que avanza el diseño. Ambos conceptos permiten crear un modelo completo del diseño conforme éste evoluciona.

8.3.9 Aspectos

Cita:

"Es difícil leer un libro sobre los principios de la magia sin echar una mirada de vez en cuando a la portada para asegurarse de que no es un texto sobre diseño de software."

Bruce Tognazzini

Conforme tiene lugar el análisis de los requerimientos, surge un conjunto de "preocupaciones" que "incluyen requerimientos, casos de uso, características, estructuras de datos, calidad del servicio, variantes, fronteras de las propiedades intelectuales, colaboraciones, patrones y contratos" [AOS07]. Idealmente, un modelo de requerimientos se organiza de manera que permita aislar cada preocupación (requerimiento) a fin de considerarla en forma independiente. Sin embargo, en la práctica, algunas de estas preocupaciones abarcan todo el sistema y no es fácil dividirlas en compartimientos.

Cuando comienza el diseño, los requerimientos son refinados en una representación de diseño modular. Considere dos requerimientos, *A* y *B*. El *A interfiere* con el *B* "si se ha elegido una descomposición [refinamiento] en la que *B* no puede satisfacerse sin tomar en cuenta a *A*" [Ros04].

Por ejemplo, considere dos requerimientos para la *webapp* **CasaSeguraAsegurada.com**. El requerimiento *A* se describe con el caso de uso AVC-DVC analizado en el capítulo 6. Un refinamiento del diseño se centraría en aquellos módulos que permitieran que usuarios registrados accedieran al video de cámaras situadas en un espacio. El requerimiento *B* es de seguridad y establece que *un usuario registrado debe ser validado antes de que use* **CasaSeguraAsegurada.com**. Este requerimiento es aplicable a todas las funciones disponibles para los usuarios registrados de *CasaSegura*. Cuando ocurre el refinamiento del diseño, *A** es una representación del diseño para el requerimiento *A*, y *B** es otra para el requerimiento *B*. Por tanto, *A** y *B** son representaciones de las preocupaciones, y *B* interfiere* con *A**.

Una preocupación de interferencia es alguna característica del sistema que se aplica a través de muchos requerimientos distintos.

Un *aspecto* es una representación de una preocupación de interferencia. Entonces, la representación del diseño, *B**, del requerimiento *un usuario registrado debe ser validado antes de que use* **CasaSeguraAsegurada.com** es un aspecto de la *webapp CasaSegura*. Es importante identificar aspectos, de modo que el diseño les pueda dar acomodo conforme sucede el refinamiento y la división en módulos. En un contexto ideal, un aspecto se implementa como módulo (componente) separado y no como fragmentos de software "dispersos" o "regados" en muchos componentes [Ban06]. Para lograr esto, la arquitectura del diseño debe apoyar un mecanismo para definir aspecto: un módulo que permita implementar la preocupación en todas aquellas con las que interfiera.

8.3.10 Rediseño

WebRef
En la dirección **www.refactoring.com**, se encuentran recursos excelentes para el rediseño.

WebRef
En **http://c2.com/cgi/wiki?RefactoringPatterns**, se encuentran varios patrones de rediseño.

Una actividad de diseño importante que se sugiere para muchos métodos ágiles (véase el capítulo 3) es el *rediseño*, técnica de reorganización que simplifica el diseño (o código) de un componente sin cambiar su función o comportamiento. Fowler [Fow00] define el rediseño del modo siguiente: "Es el proceso de cambiar un sistema de software en forma tal que no se altera el comportamiento externo del código [diseño], pero sí se mejora su estructura interna."

Cuando se rediseña el software, se examina el diseño existente en busca de redundancias, elementos de diseño no utilizados, algoritmos ineficientes o innecesarios, estructuras de datos mal construidas o inapropiadas y cualquier otra falla del diseño que pueda corregirse para obtener un diseño mejor. Por ejemplo, una primera iteración de diseño tal vez genere un componente con poca cohesión (realiza tres funciones que tienen poca relación entre sí). Después de un análisis cuidadoso, se decide rediseñar el componente en tres componentes separados, cada uno con mucha cohesión. El resultado será un software más fácil de integrar, de probar y de mantener.

CASASEGURA

Conceptos de diseño

La escena: Cubículo de Vinod, cuando comienza el modelado del diseño.

Participantes: Vinod, Jamie y Ed, miembros del equipo de ingeniería del software de *CasaSegura*. También Shakira, nueva integrante del equipo.

La conversación:

[Los cuatro miembros del equipo acaban de regresar de un seminario matutino llamado "Aplicación de los conceptos básicos del diseño", ofrecido por una profesora local de ciencias de la computación.]

Vinod: ¿Les dejó algo el seminario?

Ed: Sabíamos la mayor parte de lo que trató, pero creo que no fue mala idea escucharlo de nuevo.

Jamie: Cuando estudiaba la carrera de ciencias de la computación, nunca entendí, en realidad, por qué era tan importante, como decían, ocultar información.

Vinod: Por… la línea de base… es una técnica para reducir la propagación del error en un programa. En realidad, la independencia funcional hace lo mismo.

Shakira: Yo no estudié una carrera de computación, así que mucho de lo que dijo el instructor fue nuevo para mí. Soy capaz de generar buen código y rápido. No veo por qué es tan importante todo eso.

Jamie: He visto tu trabajo, Shak, y aplicas de manera natural mucho de lo que se habló… ésa es la razón por la que funcionan bien tus diseños y códigos.

Shakira (sonríe): Bueno, siempre trato de realizar la partición del código, hacer que se aboque a una cosa, construir interfaces sencillas y restringidas, reutilizar código siempre que se pueda… esa clase de cosas.

Ed: Modularidad, independencia funcional, ocultamiento, patrones… ya veo.

Jamie: Todavía recuerdo el primer curso de programación que tomé… nos enseñaron a refinar el código en forma iterativa.

Vinod: Lo mismo puede aplicarse al diseño, ya sabes.

Vinod: Los únicos conceptos que no había escuchado antes fueron los de "aspectos" y "rediseño".

Shakira: Eso se utiliza en programación extrema.

Ed: Sí. No es muy diferente del refinamiento, sólo que lo haces una vez terminado el diseño o código. Si me preguntan, diré que es algo así como una etapa de optimización del software.

Jamie: Volvamos al diseño de *CasaSegura*. Pienso que mientras desarrollemos el modelo de su diseño, debemos poner estos conceptos en nuestra lista de revisión.

Vinod: Estoy de acuerdo. Pero es importante que todos nos comprometamos a pensar en ellos al hacer el diseño.

8.3.11 Conceptos de diseño orientados a objeto

El paradigma de la orientación a objeto (OO) se utiliza mucho en la ingeniería de software moderna. El apéndice 2 está pensado para aquellos lectores que no estén familiarizados con los conceptos de diseño OO, tales como clases y objetos, herencia, mensajes y polimorfismo, entre otros.

8.3.12 Clases de diseño

El modelo de requerimientos define un conjunto de clases de análisis (capítulo 6). Cada una describe algún elemento del dominio del problema y se centra en aspectos de éste que son visibles para el usuario. El nivel de abstracción de una clase de análisis es relativamente alto.

Conforme el diseño evoluciona, se definirá un conjunto de *clases de diseño* que refinan las clases de análisis, dando detalles del diseño que permitirán que las clases se implementen y generen una infraestructura para el software que apoye la solución de negocios. Pueden desarrollarse cinco tipos diferentes de clases de diseño, cada una de las cuales representa una capa distinta de la arquitectura del diseño [Amb01]:

¿Qué tipos de clases crea el diseñador?

- *Clases de usuario de la interfaz*. Definen todas las abstracciones necesarias para la interacción humano-computadora (IHC). En muchos casos, la IHC ocurre dentro del contexto de una *metáfora* (por ejemplo, cuaderno de notas, formato de orden, máquina de fax, etc.) y las clases del diseño para la interfaz son representaciones visuales de los elementos de la metáfora.
- *Clases del dominio de negocios*. Es frecuente que sean refinamientos de las clases de análisis definidas antes. Las clases identifican los atributos y servicios (métodos) que se requieren para implementar algunos elementos del dominio de negocios.
- *Clases de proceso*. Implantan abstracciones de negocios de bajo nivel que se requieren para administrar por completo las clases de dominio de negocios.
- *Clases persistentes*. Representan almacenamientos de datos (por ejemplo, una base de datos) que persistirán más allá de la ejecución del software.
- *Clases de sistemas*. Implantan las funciones de administración y control del software que permiten que el sistema opere y se comunique dentro de su ambiente de computación y con el mundo exterior.

A medida que se forma la arquitectura, el nivel de abstracción se reduce cuando cada clase de análisis se transforma en una representación del diseño. Es decir, las clases de análisis representan objetos de datos (y servicios asociados que se aplican a éstos) que usan la terminología del dominio del negocio. Las clases de diseño presentan muchos más detalles técnicos como guía para su implementación.

Arlow y Neustadt [Arl02] sugieren que se revise cada clase de diseño para asegurar que esté "bien formada". Definen cuatro características de las clases de diseño bien formadas:

¿Qué es una clase de diseño "bien formada"?

Completa y suficiente. Una clase de diseño debe ser el encapsulado total de todos los atributos y métodos que sea razonable esperar (con base en una interpretación comprensible del nombre de la clase) y que existan para la clase. Por ejemplo, la clase **Escena** definida para el software de la edición de video será completa sólo si contiene todos los atributos y métodos que se asocian de manera razonable con la creación de una escena de video. La suficiencia asegura que la clase de diseño contiene sólo los métodos que bastan para lograr el objetivo de la clase, ni más ni menos.

Primitivismo. Los métodos asociados con una clase de diseño deben centrarse en el cumplimiento de un servicio para la clase. Una vez implementado el servicio con un método, la clase no debe proveer otro modo de hacer lo mismo. Por ejemplo, la clase **VideoClip** para el software de la edición de video tal vez tenga los atributos **punto-inicial** y **punto-final** que indiquen los puntos de inicio y fin del corto (observe que el video original cargado en el sistema puede ser más extenso que el corto utilizado). Los métodos *EstablecerPuntoInicial ( )* y *EstablecerPuntoFinal ( )* proporcionan los únicos medios para establecer los puntos de comienzo y terminación del corto.

Mucha cohesión. Una clase de diseño cohesiva tiene un conjunto pequeño y centrado de responsabilidades; para implementarlas emplea atributos y métodos de objetivo único. Por

ejemplo, la clase **VideoClip** quizá contenga un conjunto de métodos para editar el corto de video. La cohesión se mantiene en tanto cada método se centre sólo en los atributos asociados con el corto.

Poco acoplamiento. Dentro del modelo de diseño, es necesario que las clases de diseño colaboren una con otra. Sin embargo, la colaboración debe mantenerse en un mínimo aceptable. Si un modelo de diseño está muy acoplado (todas las clases de diseño colaboran con todas las demás), el sistema es difícil de implementar, probar y mantener con el paso del tiempo. En general, las clases de diseño dentro de un subsistema deben tener sólo un conocimiento limitado de otras clases. Esta restricción se llama *Ley de Demeter* [Lie03] y sugiere que un método sólo debe enviar mensajes a métodos que están en clases vecinas.[6]

CASASEGURA

Refinamiento de una clase de análisis en una clase de diseño

La escena: El cubículo de Ed, cuando comienza el modelado del diseño.

Participantes: Vinod y Ed, miembros del equipo de ingeniería de software de *CasaSegura*.

La conversación:

[Ed está trabajando en la clase **PlanodelaPlanta** (véanse el recuadro en la sección 6.5.3 y la figura 6.10) y la ha refinado para el modelo del diseño.]

Ed: Entonces recuerdas la clase **PlanodelaPlanta**, ¿verdad? Se usa como parte de las funciones de vigilancia y administración de la casa.

Vinod (afirma con la cabeza): Sí, recuerdo que la usamos como parte de nuestros análisis CRC para la administración de la casa.

Ed: Así es. De cualquier modo, la estoy mejorando para el diseño. Quiero mostrarte cómo implantaremos en realidad la clase **PlanodelaPlanta**. Mi idea es implementarla como un conjunto de listas ligadas [una estructura de datos específica] de modo que… tuve que refinar la clase de análisis **PlanodelaPlanta** (véase la figura 6.10) y, en verdad, simplificarla.

Vinod: La clase de análisis sólo mostraba cosas en el dominio del problema, bueno, en la pantalla de la computadora, visibles para el usuario final, ¿de acuerdo?

Ed: Sí, pero para la clase de diseño **PlanodelaPlanta**, he tenido que agregar algunas cosas específicas para la implantación. Necesité mostrar que **PlanodelaPlanta** es un agregado de segmentos —de ahí la clase **Segmento**— y que la clase **Segmento** está compuesta de listas para segmentos de pared, ventanas, puertas, etc. La clase **Cámara** colabora con **PlanodelaPlanta** y, obviamente, hay muchas cámaras en el piso.

Vinod: Ah… veamos la ilustración de esta nueva clase de diseño, **PlanodelaPlanta**.

[Ed muestra a Vinod el diagrama que aparece en la figura 8.3.]

Vinod: Bien, ya veo lo que tratas de hacer. Esto te permite modificar el plano de la planta con facilidad porque los nuevos temas se agregan, o eliminan de la lista (el agregado), sin problemas.

Ed (asiente): Sí, creo que funcionará.

Vinod: También yo.

8.4 EL MODELO DEL DISEÑO

El modelo del diseño puede verse en dos dimensiones distintas, como se ilustra en la figura 8.4. La *dimensión del proceso* indica la evolución del modelo del diseño conforme se ejecutan las tareas de éste como parte del proceso del software. La *dimensión de la abstracción* representa el nivel de detalle a medida que cada elemento del modelo de análisis se transforma en un equivalente de diseño y luego se mejora en forma iterativa. En relación con la figura 8.4, la línea punteada indica la frontera entre los modelos de análisis y de diseño. En ciertos casos, es posible hacer una distinción clara entre ambos modelos. En otros, el modelo de análisis se mezcla poco a poco con el de diseño y la distinción es menos obvia.

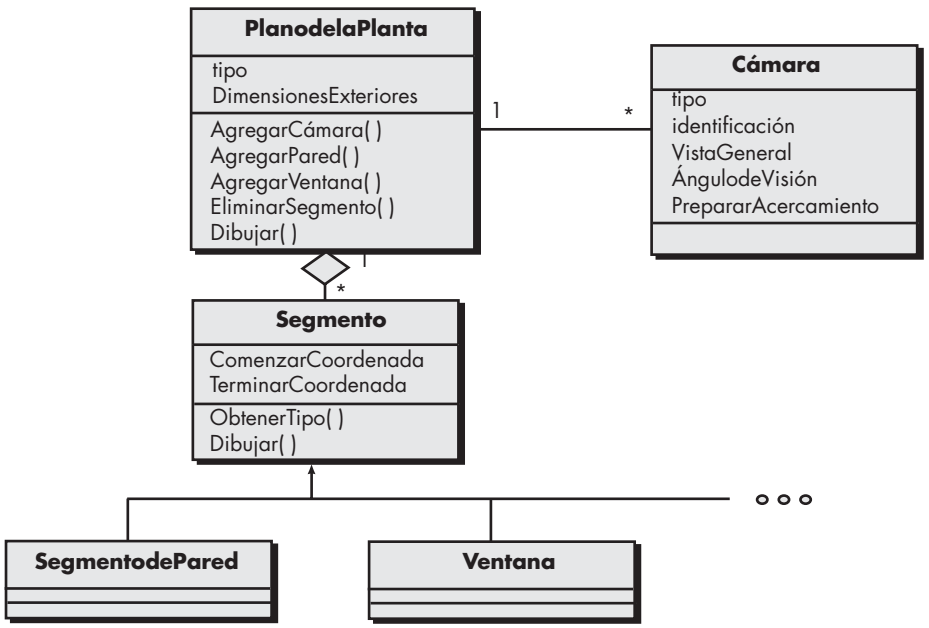
Los elementos del modelo de diseño usan muchos de los diagramas UML[7] que se utilizaron en el modelo del análisis. La diferencia es que estos diagramas se refinan y elaboran como parte

6 Una manera menos formal de la Ley de Demeter es: "cada unidad debe hablar sólo con sus amigas: no hablar con extraños".

7 En el apéndice 1 se encuentra un método de enseñanza sobre los conceptos y notación básica del UML.

FIGURA 8.3

Clase de diseño para PlanodelaPlanta y composición del agregado para ella (véase el análisis en el recuadro)



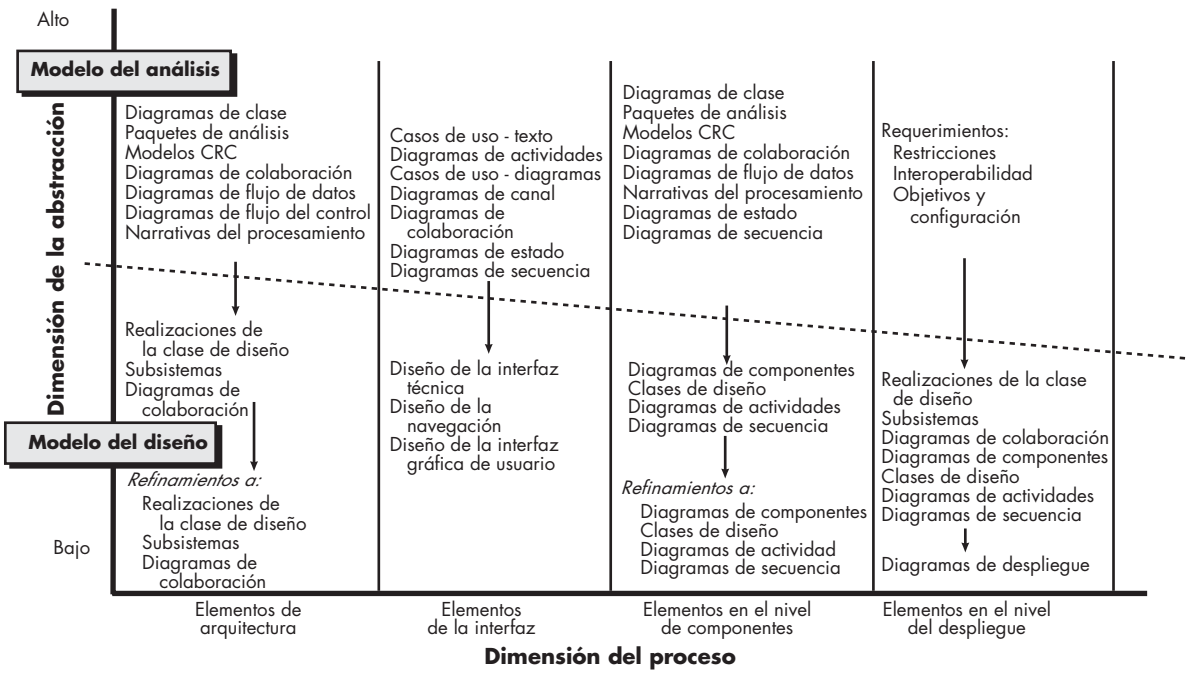
PUNTO CLAVE

El modelo de diseño tiene cuatro elementos principales: datos, arquitectura, componentes e interfaz.

del diseño; se dan más detalles específicos de la implantación y se hace énfasis en la estructura y en el estilo arquitectónico, en los componentes que residen dentro de la arquitectura y en las interfaces entre los componentes y el mundo exterior.

No obstante, debe observarse que los elementos del modelo indicados a lo largo del eje horizontal no siempre se desarrollan en forma secuencial. En la mayoría de los casos, el diseño preliminar de la arquitectura establece la etapa y va seguido del diseño de la interfaz y del dise-

FIGURA 8.4 Dimensiones del modelo de diseño



Cita:

"Las preguntas acerca de si el diseño es necesario o digno de pagarse están más allá de la discusión: el diseño es inevitable. La alternativa al buen diseño es el mal diseño, no la falta de diseño."

Douglas Martin

ño del nivel de los componentes, los cuales con frecuencia ocurren en paralelo. El modelo de despliegue por lo general se retrasa hasta que el diseño haya sido desarrollado por completo.

Es posible aplicar patrones de diseño en cualquier punto de este proceso (véase el capítulo 12). Estos patrones permiten aplicar el conocimiento del diseño a problemas específicos del dominio que han sido encontrados y resueltos por otras personas.

8.4.1 Elementos del diseño de datos

Igual que otras actividades de la ingeniería de software, el diseño de datos (en ocasiones denominado *arquitectura de datos*) crea un modelo de datos o información que se representa en un nivel de abstracción elevado (el punto de vista del usuario de los datos). Este modelo de los datos se refina después en forma progresiva hacia representaciones más específicas de la implementación que puedan ser procesadas por el sistema basado en computadora. En muchas aplicaciones de software, la arquitectura de los datos tendrá una influencia profunda en la arquitectura del software que debe procesarlo.

PUNTO CLAVE

En el nivel de la arquitectura (aplicación), el diseño de los datos se centra en archivos o bases de datos; en el de los componentes, el diseño de datos considera las estructuras de datos que se requieren para implementar objetos de datos locales.

La estructura de los datos siempre ha sido parte importante del diseño de software. En el nivel de componentes del programa, del diseño de las estructuras de datos y de los algoritmos requeridos para manipularlos, es esencial la creación de aplicaciones de alta calidad. En el nivel de la aplicación, la traducción de un modelo de datos (obtenido como parte de la ingeniería de los requerimientos) a una base de datos es crucial para lograr los objetivos de negocios de un sistema. En el nivel de negocios, el conjunto de información almacenada en bases de datos incompatibles y reorganizados en un "data warehouse" permite la minería de datos o descubrimiento de conocimiento que tiene un efecto en el éxito del negocio en sí. En cada caso, el diseño de los datos juega un papel importante. El diseño de datos se estudia con más detalle en el capítulo 9.

8.4.2 Elementos del diseño arquitectónico

Cita:

"Puede usarse una goma en la mesa de dibujo o un marro en el sitio construido."

Frank Lloyd Wright

El *diseño de la arquitectura* del software es el equivalente del plano de una casa. Éste ilustra la distribución general de las habitaciones, su tamaño, forma y relaciones entre ellas, así como las puertas y ventanas que permiten el movimiento entre los cuartos. El plano da una visión general de la casa. Los elementos del diseño de la arquitectura dan la visión general del software.

El modelo arquitectónico [Sha96] proviene de tres fuentes: 1) información sobre el dominio de la aplicación del software que se va a elaborar, 2) los elementos específicos del modelo de requerimientos, tales como diagramas de flujo de datos o clases de análisis, sus relaciones y colaboraciones para el problema en cuestión y 3) la disponibilidad de estilos arquitectónicos (capítulo 9) y sus patrones (capítulo 12).

Por lo general, el elemento de diseño arquitectónico se ilustra como un conjunto de sistemas interconectados, con frecuencia obtenidos de paquetes de análisis dentro del modelo de requerimientos. Cada subsistema puede tener su propia arquitectura (por ejemplo, la interfaz gráfica de usuario puede estar estructurada de acuerdo con un estilo de arquitectura preexistente para interfaces de usuario). En el capítulo 9 se presentan técnicas para obtener elementos específicos del modelo arquitectónico.

Cita:

"El público está más familiarizado con el mal diseño que con el buen diseño. En realidad, está condicionado para que prefiera el mal diseño porque es con el que vive. Lo nuevo le parece amenazador; lo viejo le da seguridad."

Paul Rand

8.4.3 Elementos de diseño de la interfaz

El diseño de la interfaz para el software es análogo al conjunto de trazos (y especificaciones) detalladas para las puertas, ventanas e instalaciones de una casa. Tales dibujos ilustran el tamaño y forma de puertas y ventanas, la manera en la que operan, la forma en la que llegan las instalaciones de servicios (agua, electricidad, gas, teléfono, etc.) a la vivienda y se distribuyen entre las habitaciones indicadas en el plano. Indican dónde está el timbre de la puerta, si se usará un intercomunicador para anunciar la presencia de un visitante y cómo se va a instalar el

sistema de seguridad. En esencia, los planos (y especificaciones) detallados para las puertas, ventanas e instalaciones externas nos dicen cómo fluyen las cosas y la información hacia dentro y fuera de la casa y dentro de los cuartos que forman parte del plano. Los elementos de diseño de la interfaz del software permiten que la información fluya hacia dentro y afuera del sistema, y cómo están comunicados los componentes que son parte de la arquitectura.

PUNTO CLAVE

Hay tres partes para el elemento de diseño de la interfaz: la interfaz de usuario, las interfaces dirigidas hacia el sistema externo a la aplicación y las interfaces orientadas hacia los componentes dentro de ésta.

Hay tres elementos importantes del diseño de la interfaz: 1) la interfaz de usuario (IU), 2) las interfaces externas que tienen que ver con otros sistemas, dispositivos, redes y otros productores o consumidores de información y 3) interfaces internas que involucran a los distintos componentes del diseño. Estos elementos del diseño de la interfaz permiten que el software se comunique externamente y permita la comunicación y colaboración internas entre los componentes que constituyen la arquitectura del software.

El diseño de la IU (denominada cada vez con más frecuencia *diseño de la usabilidad*) es una acción principal de la ingeniería de software y se estudia con detalle en el capítulo 11. El diseño de la usabilidad incorpora elementos estéticos (como distribución, color, gráficos, mecanismos de interacción, etc.), elementos ergonómicos (por ejemplo, distribución y colocación de la información, metáforas, navegación por la IU, etc.) y elementos técnicos (como patrones de la IU y patrones reutilizables). En general, la IU es un subsistema único dentro de la arquitectura general de la aplicación.

Cita:

"De vez en cuando apártate, relájate un poco, para que cuando regreses al trabajo tu criterio sea más seguro. Toma algo de distancia porque entonces el trabajo parece más pequeño y es posible apreciar una porción mayor con una sola mirada, de modo que se detecta con facilidad la falta de armonía y proporción."

Leonardo da Vinci

El diseño de interfaces externas requiere información definitiva sobre la entidad a la que se envía información o desde la que se recibe. En todo caso, esta información debe recabarse durante la ingeniería de requerimientos (capítulo 5) y verificarse una vez que comienza el diseño de la interfaz.[8] El diseño de interfaces externas debe incorporar la revisión en busca de errores y (cuando sea necesario) las medidas de seguridad apropiadas.

El diseño de las interfaces internas se relaciona de cerca con el diseño de componentes (véase el capítulo 10). Las realizaciones del diseño de las clases de análisis representan todas las operaciones y esquemas de mensajería que se requieren para permitir la comunicación y colaboración entre las operaciones en distintas clases. Cada mensaje debe diseñarse para que contenga la información que se requiere transmitir y los requerimientos específicos de la función de la operación que se ha solicitado. Si para el diseño se elige el enfoque clásico de un proceso de entrada-salida, la interfaz de cada componente del software se diseña con base en las representaciones del flujo de datos y en la funcionalidad descrita en una narrativa de procesamiento.

WebRef

En la dirección **www.useit.com**, se encuentra información sumamente valiosa sobre el diseño de la IU.

En ciertos casos, una interfaz se modela en forma muy parecida a la de una clase. En el UML se define *interfaz* del modo siguiente [OMG03a]: "Es un especificador para las operaciones visibles desde el exterior [públicas] de una clase, un componente u otro clasificador (incluso subsistemas), sin especificar su estructura interna." En pocas palabras, una interfaz es un conjunto de operaciones que describen alguna parte del comportamiento de una clase y dan acceso a aquéllas.

Por ejemplo, la función de seguridad de *CasaSegura* hace uso del panel de control que permite que el propietario controle ciertos aspectos de la función de seguridad. En una versión avanzada del sistema, las funciones del panel de control podrían implementarse a través de un PDA inalámbrico o un teléfono móvil.

Cita:

"Un error común que comete la gente cuando trata de diseñar algo a prueba de tontos es subestimar la ingenuidad de los completamente tontos."

Douglas Adams

La clase **PaneldeControl** (véase la figura 8.5) proporciona el comportamiento asociado con un teclado, por lo que debe implementar las operaciones *LeerTeclazo ( )* y *DecodificarTecla ( )*. Si estas operaciones se van a dar a otras clases (en este caso, **PDAInalámbrico** y **TeléfonoMóvil**), es útil definir una interfaz como la de la figura. La interfaz, llamada **Teclado**, se ilustra como un estereotipo <<interfaz>> o como un círculo pequeño con leyenda y conectado a la clase

8 Las características de la interfaz pueden cambiar con el tiempo. Por tanto, un diseñador debe cerciorarse de que la especificación para ella sea exacta y completa.

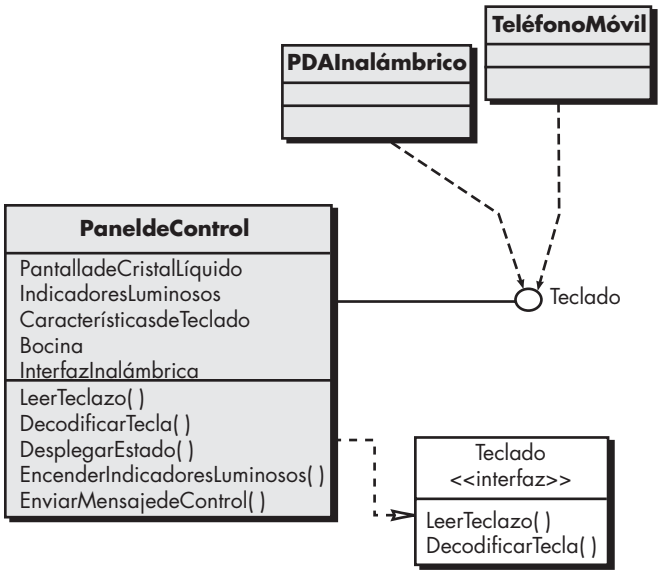


FIGURA 8.5
Representación de la interfaz para PaneldeControl

con una línea. La interfaz se define sin atributos y con el conjunto de operaciones que sean necesarias para lograr el comportamiento de un teclado.

La línea punteada con un triángulo abierto en su extremo (en la figura 8.5) indica que la clase **PaneldeControl** proporciona las operaciones de **Teclado** como parte de su comportamiento. En UML, esto se caracteriza como una *realización.* Es decir, parte del comportamiento de **PaneldeControl** se implementará con la realización de las operaciones de **Teclado**. Éstas se darán a otras clases que accedan a la interfaz.

8.4.4 Elementos del diseño en el nivel de los componentes

El diseño en el nivel de los componentes del software es el equivalente de los planos (y especificaciones) detallados de cada habitación de la casa. Estos dibujos ilustran el cableado y la plomería de cada cuarto, la ubicación de cajas eléctricas e interruptores, grifos, coladeras, regaderas, tinas, drenajes, gabinetes y closets. También describen el tipo de piso que se va a usar, las molduras que se van a aplicar y todos los detalles asociados con una habitación. El diseño de componentes para el software describe por completo los detalles internos de cada componente. Para lograrlo, este diseño define estructuras de datos para todos los objetos de datos locales y detalles algorítmicos para todo el procesamiento que tiene lugar dentro de un componente, así como la interfaz que permite el acceso a todas las operaciones de los componentes (comportamientos).

Cita:

"Los detalles no son los detalles. Constituyen el diseño."

Charles Eames

En el contexto de la ingeniería de software orientada a objeto, un componente se representa en forma de diagrama UML, como se ilustra en la figura 8.6. En ésta, aparece un componente llamado **AdministraciónSensor** (parte de la función de seguridad de *CasaSegura*). Una flecha punteada conecta al componente con una clase llamada **Sensor**, a él asignada. El compo-

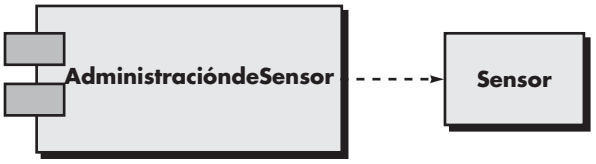


FIGURA 8.6
Diagrama de componente UML

nente **AdministracióndeSensor** lleva a cabo funciones asociadas con los sensores de *CasaSegura*, incluso su vigilancia y configuración. En el capítulo 10 se analizan más los diagramas de componentes.

Los detalles del diseño de un componente se modelan en muchos niveles de abstracción diferentes. Se utiliza un diagrama de actividades UML para representar la lógica del procesamiento. El flujo detallado del procedimiento para un componente se representa con seudocódigo (representación que se parece a un lenguaje de programación y que se describe en el capítulo 10) o con alguna otra forma diagramática (como un diagrama de flujo o de cajas). La estructura algorítmica sigue las reglas establecidas para la programación estructurada (por ejemplo, un conjunto de construcciones restringidas de procedimiento). Las estructuras de datos, seleccionadas con base en la naturaleza de los objetos de datos que se van a procesar, por lo general se modelan con el empleo de seudocódigo del lenguaje de programación que se usará para la implementación.

8.4.5 Elementos del diseño del despliegue

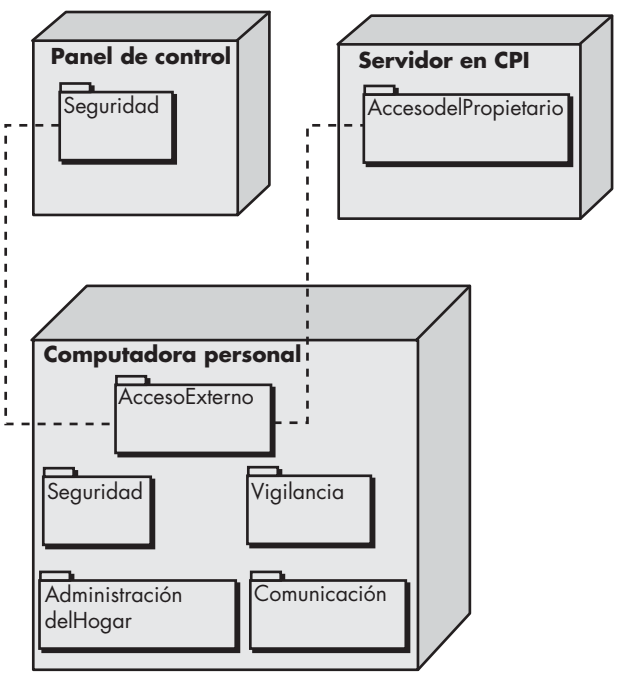
Los elementos del diseño del despliegue indican la forma en la que se acomodarán la funcionalidad del software y los subsistemas dentro del ambiente físico de la computación que lo apoyará. Por ejemplo, los elementos del producto *CasaSegura* se configuran para que operen dentro de tres ambientes de computación principales: una PC en la casa, el panel de control de *CasaSegura* y un servidor alojado en CPI Corp. (que provee el acceso al sistema a través de internet).

PUNTO CLAVE

Los diagramas de despliegue comienzan en forma de descriptor, donde el ambiente de despliegue se describe en términos generales. Después se utilizan formas de instancia y se describen explícitamente los elementos de la configuración.

Durante el diseño se desarrolla un diagrama de despliegue que después se refina, como se ilustra en la figura 8.7. En ella aparecen tres ambientes de computación (en realidad habría más, con sensores y cámaras, entre otros). Se indican los subsistemas (funcionalidad) que están alojados dentro de cada elemento de computación. Por ejemplo, la computadora personal aloja subsistemas que implementan características de seguridad, vigilancia, administración del hogar y comunicaciones. Además, se ha diseñado un subsistema de acceso externo para manejar todos los intentos de acceder al sistema *CasaSegura* desde un lugar externo. Cada subsistema se elaboraría para que indicara los componentes que implementa.

FIGURA 8.7
Diagrama de despliegue UML



El diagrama que aparece en la figura 8.7 es un *formato descriptor*. Esto significa que el diagrama de despliegue muestra el ambiente de computación, pero no indica de manera explícita los detalles de la configuración. Por ejemplo, no se da mayor identificación de la "computadora personal". Puede ser una Mac o basarse en Windows, una estación de trabajo Sun o un sistema Linux. Estos detalles se dan cuando se regrese al diagrama de despliegue en el *formato de instancia* en las etapas finales del diseño, o cuando comience la construcción. Se identifica cada instancia del despliegue (una configuración específica, llamada *configuración del hardware*).

8.5 Resumen

El diseño del software comienza cuando termina la primera iteración de la ingeniería de requerimientos. El objetivo del diseño del software es aplicar un conjunto de principios, conceptos y prácticas que llevan al desarrollo de un sistema o producto de alta calidad. La meta del diseño es crear un modelo de software que implantará correctamente todos los requerimientos del usuario y causará placer a quienes lo utilicen. Los diseñadores del software deben elegir entre muchas alternativas de diseño y llegar a la solución que mejor se adapte a las necesidades de los participantes en el proyecto.

El proceso de diseño va de una visión "panorámica" del software a otra más cercana que define el detalle requerido para implementar un sistema. El proceso comienza por centrarse en la arquitectura. Se definen los subsistemas, se establecen los mecanismos de comunicación entre éstos, se identifican los componentes y se desarrolla la descripción detallada de cada uno. Además, se diseñan las interfaces externa, interna y de usuario.

Los conceptos de diseño han evolucionado en los primeros 60 años de trabajo de la ingeniería de software. Describen atributos de software de computadora que debe presentarse sin importar el proceso que se elija para hacer la ingeniería, los métodos de diseño que se apliquen o los lenguajes de programación que se utilicen. En esencia, los conceptos de diseño ponen el énfasis en la necesidad de la abstracción como mecanismo para crear componentes reutilizables de software, en la importancia de la arquitectura como forma de entender mejor la estructura general de un sistema, en los beneficios de la ingeniería basada en patrones como técnica de diseño de software con capacidad comprobada, en el valor de la separación de problemas y de la modularidad eficaz como forma de elaborar software más entendible, más fácil de probar y de recibir mantenimiento, en las consecuencias de ocultar información como mecanismo para reducir la propagación de los efectos colaterales cuando hay errores, en el efecto de la independencia funcional como criterio para construir módulos eficaces, en el uso del refinamiento como mecanismo de diseño, en una consideración de los aspectos que interfieren con los requerimientos del sistema, en la aplicación del rediseño para optimizar el diseño obtenido y en la importancia de las clases orientadas a objetos y de las características relacionadas con ellos.

El modelo del diseño incluye cuatro elementos distintos. Conforme se desarrolla cada uno, surge una visión más completa del diseño. El elemento arquitectónico emplea información obtenida del dominio de la aplicación, del modelo de requerimientos y de los catálogos disponibles de patrones y estilos para obtener una representación estructural completa del software, de sus subsistemas y componentes. Los elementos del diseño de la interfaz modelan las interfaces internas y externas y la de usuario. Los elementos en el nivel de componentes definen cada uno de los módulos (componentes) que constituyen la arquitectura. Por último, los elementos del diseño albergan la arquitectura, sus componentes y las interfaces dirigidas hacia la configuración física en la que se alojará el software.

Problemas y puntos por evaluar

8.1. Cuando se "escribe" un programa, ¿se diseña software? ¿En qué difieren el diseño de software y la codificación?

8.2. Si el diseño del software no es un programa (y no lo es), entonces, ¿qué es?

8.3. ¿Cómo se evalúa la calidad del diseño del software?

8.4. Estudie el conjunto de tareas presentado para el diseño. ¿Dónde se evalúa la calidad en dicho conjunto? ¿Cómo se logra? ¿Cómo se consiguen los atributos de calidad estudiados en la sección 8.2.1?

8.5. Dé ejemplos de tres abstracciones de datos y de las abstracciones de procedimiento que se usan para manipularlas.

8.6. Describa con sus propias palabras la arquitectura de software.

8.7. Sugiera un patrón de diseño que encuentre en una categoría de objetos cotidianos (por ejemplo, electrónica de consumo, automóviles, aparatos, etc.). Describa el patrón en forma breve.

8.8. Describa con sus propias palabras la separación de problemas. ¿Hay algún caso en el que no sea apropiada la estrategia de divide y vencerás? ¿Cómo afecta esto al argumento a favor de la modularidad?

8.9. ¿Cuándo debe implementarse un diseño modular como software monolítico? ¿Cómo se logra esto? ¿El rendimiento es la única justificación para la implementación de software monolítico?

8.10. Analice la relación entre el concepto de ocultamiento de información como atributo de la modularidad efectiva y el de independencia de los módulos.

8.11. ¿Cómo se relacionan los conceptos de acoplamiento y portabilidad del software? Dé ejemplos que apoyen su punto de vista.

8.12. Aplique un "enfoque de refinamiento stepwise" para desarrollar tres niveles distintos de abstracciones del procedimiento para uno o más de los programas siguientes: *a*) un revisor de la escritura que, dada una cantidad numérica de dinero, imprima ésta en las palabras que se requieren normalmente en un cheque. *b*) una resolución en forma iterativa de las raíces de una ecuación trascendente. *c*) un algoritmo de programación de tareas simples para un sistema operativo.

8.13. Considere el software requerido para implementar la capacidad de navegación (con un GPS) en un dispositivo móvil de comunicación portátil. Describa dos o tres preocupaciones de interferencia que se presentarían. Analice la manera en la que se representaría como aspecto una de estas preocupaciones.

8.14. ¿"Rediseñar" significa que se modifica todo el diseño en forma iterativa? Si no es así, ¿qué significa?

8.15. Describa en breves palabras cada uno de los cuatro elementos del modelo del diseño.

Lecturas adicionales y fuentes de información

Donald Norman ha escrito dos libros (*The Design of Everyday Things*, Doubleday, 1990, y *The Psychology of Everyday Things*, Harpercollins, 1988) que se han convertido en clásicos de la bibliografía sobre el diseño y una "obligación" de lectura para quien diseñe cualquier cosa que utilicen los humanos. Adams (*Conceptual Blockbusting*, 3a. ed., Addison-Wesley, 1986) escribió un libro cuya lectura es esencial para los diseñadores que deseen ampliar su forma de pensar. Por último, el texto clásico de Polya (*How to Solve It*, 2a. ed., Princeton University Press, 1988) proporciona un proceso general de solución de problemas que ayuda a los diseñadores de software cuando se enfrentan a problemas complejos.

En la misma tradición, Winograd *et al.* (*Bringing Design to Software*, Addison-Wesley, 1996) analizan los diseños de software que funcionan, los que no y su por qué. Un libro fascinante editado por Wixon y Ramsey (*Field Methods Casebook for Software Design*, Wiley, 1996) sugiere métodos de investigación de campo (muy parecidos a los de los antropólogos) para entender cómo hacen el trabajo los usuarios finales y luego diseñar el software que satisfaga sus necesidades. Beyer y Holtzblatt (*Contextual Design: A Customer-Centered Approach to Systems Designs*, Academic Press, 1997) ofrecen otro punto de vista del diseño de software que integra al consumidor o usuario en cada aspecto del proceso de diseño. Bain (*Emergent Design*, Addison-Wesley, 2008) acopla los patrones, el rediseño y el desarrollo orientado a pruebas en un enfoque de diseño eficaz.

Un tratamiento exhaustivo del diseño en el contexto de la ingeniería de software es presentado por Fox (*Introduction to Software Engineering Design*, Addison-Wesley, 2006) y Zhu (*Software Design Methodology*, Butterworth-Heinemann, 2005). McConnell (*Code Complete*, 2a. ed., Microsoft Press, 2004) plantea un estudio excelente de los aspectos prácticos del diseño de software de alta calidad. Robertson (*Simple Program Design*, 3a. ed., Boyd y Fraser Publishing, 1999) presenta un análisis introductorio del diseño de software, útil para quienes se inician en el estudio del tema. Budgen (*Software Design*, 2a. ed., Addison-Wesley, 2004) presenta

varios métodos populares de diseño y los compara entre sí. Fowler *y sus colegas* (*Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999) estudian técnicas para la optimización incremental de diseños de software. Rosenberg y Stevens (*Use Case Driven Object Modeling with UML*, Apress, 2007) estudian el desarrollo de diseños orientados a objeto con el empleo de casos de uso como fundamento.

En una antología editada por Freeman y Wasserman (*Software Design Techniques*, 4a. ed., IEEE, 1983), hay una excelente revisión histórica del diseño de software. Esta edición contiene muchas reimpresiones de los artículos clásicos que forman la base de las tendencias actuales del diseño del software. Card y Glass (*Measuring Software Design Quality*, Prentice-Hall, 1990) presentan mediciones de la calidad procedentes de los campos de la técnica y la administración.

En internet hay una variedad amplia de fuentes de información acerca del diseño de software. En el sitio web del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, se encuentra una lista actualizada de referencias existentes en la red mundial que son relevantes para el diseño de software y para la ingeniería de diseño.

CAPÍTULO

9 DISEÑO DE LA ARQUITECTURA

CONCEPTOS CLAVE

arquetipos 218
arquitectura 207
  alternativas 221
  centrada en datos 213
  complejidad 224
  componentes 219
  diseño 217
  en capas 214
  estilos 211
  flujo de datos 213
  formato 210
  géneros 209
  orientada a objeto 214
  patrones 215
  refinamiento 219
ATAM 222
instancias 220
lenguaje de descripción arquitectónica 224
mapeo 225
rediseño 228

Se ha descrito al diseño como un proceso de etapas múltiples en el que, a partir de los requerimientos de información, se sintetizan las representaciones de los datos y la estructura del programa, las características de la interfaz y los detalles del procedimiento. Esta descripción la amplía Freeman [Fre80], como sigue:

> El diseño es una actividad que tiene que ver con la toma de decisiones importantes, con frecuencia de naturaleza estructural. Comparte con la programación el objetivo de abstraer una representación de la información y de las secuencias de procesamiento, pero en los extremos el grado de detalle es muy distinto. El diseño elabora representaciones coherentes y bien planeadas de programas, que se concentran en las relaciones de las partes en el nivel más alto y en las operaciones lógicas involucradas en los niveles bajos.

Como se señaló en el capítulo 8, el diseño está motivado por la información. Los métodos de diseño del software provienen de los tres dominios del modelo de análisis. Los dominios de datos, funciones y comportamiento sirven como guía para la creación del diseño del software.

En este capítulo se presentan los métodos requeridos para crear "representaciones coherentes y bien planeadas" de las capas de datos y de la arquitectura del modelo de diseño. El objetivo es brindar un enfoque sistemático para la obtención del diseño arquitectónico, el plano preliminar a partir del cual se elabora el software.

UNA MIRADA RÁPIDA

¿Qué es? El diseño arquitectónico representa la estructura de los datos y de los componentes del programa que se requieren para construir un sistema basado en computadora. Considera el estilo de arquitectura que adoptará el sistema, la estructura y las propiedades de los componentes que lo constituyen y las interrelaciones que ocurren entre sus componentes arquitectónicos.

¿Quién lo hace? Aunque es un ingeniero de software quien puede diseñar tanto los datos como la arquitectura, es frecuente que si deben construirse sistemas grandes y complejos, el trabajo lo realicen especialistas. El diseñador de una base de datos o data warehouse crea la arquitectura de los datos para un sistema. El "arquitecto del sistema" selecciona un estilo arquitectónico apropiado a partir de los requerimientos obtenidos durante el análisis de los datos.

¿Por qué es importante? El lector no intentaría construir una casa sin un plano, ¿o sí? Tampoco comenzaría los planos con el dibujo de la plomería del lugar. Antes de preocuparse por los detalles, necesitaría tener el panorama general: la casa en sí. Eso es lo que hace el diseño arquitectónico, da el panorama y asegura que sea el correcto.

¿Cuáles son los pasos? El diseño de la arquitectura comienza con el diseño de los datos y continúa con la obtención de una o más representaciones de la estructura arquitectónica del sistema. Se analizan alternativas de estilos o patrones arquitectónicos para llegar a la estructura más adecuada para los requerimientos del usuario y para los atributos de calidad. Una vez seleccionada la alternativa, se elabora la arquitectura con el empleo de un método de diseño.

¿Cuál es el producto final? Durante el diseño arquitectónico se crea un modelo de arquitectura que incluye la arquitectura de los datos y la estructura del programa. Además, se describen las propiedades y relaciones (interacciones) que hay entre los componentes.

¿Cómo me aseguro de que lo hice bien? En cada etapa se revisan los productos del trabajo del diseño del software para que sean claros, correctos, completos y consistentes con los requerimientos y entre sí.

9.1 ARQUITECTURA DEL SOFTWARE

En un libro clave sobre el tema, Shaw y Garlan [Sha96] plantean lo siguiente sobre la arquitectura del software:

> Desde el primer programa que se dividió en módulos, los sistemas de software han tenido arquitecturas y los programadores han sido los responsables de las interacciones entre los módulos y las propiedades globales del ensamble. Históricamente, las arquitecturas han estado implícitas: accidentes de implementación o sistemas heredados del pasado. Los desarrolladores de buen software han adoptado con frecuencia uno o varios patrones de arquitectura como estrategias para la organización del sistema, pero los utilizan de manera informal y no tienen manera de hacerlos explícitos en el sistema resultante.

En el presente, la arquitectura de software eficaz y su representación y diseño explícitos se han vuelto los temas dominantes en la ingeniería de software.

9.1.1 ¿Qué es la arquitectura?

Cita:

"La arquitectura de un sistema es un marco general que describe su forma y estructura: sus componentes y la manera en la que ajustan entre sí".

Jerrold Grochow

Cuando se piensa en la arquitectura de una construcción, llegan a la mente muchos atributos distintos. En el nivel más sencillo, se considera la forma general de la estructura física. Pero, en realidad, la arquitectura es mucho más que eso. Es la manera en la que los distintos componentes del edificio se integran para formar un todo cohesivo. Es la forma en la que la construcción se adapta a su ambiente y se integra a los demás edificios en la vecindad. Es el grado en el que el edificio cumple con su propósito y en el que satisface las necesidades del propietario. Es la sensación estética de la estructura —el efecto visual de la edificación— y el modo en el que se combinan texturas, colores y materiales para crear la fachada en el exterior y el "ambiente de vida" en el interior. Es los pequeños detalles: diseño de las lámparas, tipo de piso, color de las cortinas... la lista es casi interminable. Y, finalmente, es arte.

PUNTO CLAVE

La arquitectura del software debe modelar la estructura de un sistema y la manera en la que los datos y componentes del procedimiento colaboran entre sí.

Pero la arquitectura también es algo más. Son los "miles de decisiones, tanto grandes como pequeñas" [Tyt05]. Algunas de éstas se toman en una etapa temprana del diseño y tienen un efecto profundo en todas las demás acciones. Otras se dejan para más adelante, con lo que se eliminan las restricciones prematuras que llevarían a una mala implementación del estilo arquitectónico.

Pero, ¿qué es la arquitectura del software? Bass, Clements y Kazman [Bas03] definen este término tan elusivo de la manera siguiente:

> La arquitectura del software de un programa o sistema de cómputo es la estructura o estructuras del sistema, lo que comprende a los componentes del software, sus propiedades externas visibles y las relaciones entre ellos.

La arquitectura no es el software operativo. Es una representación que permite 1) analizar la efectividad del diseño para cumplir los requerimientos establecidos, 2) considerar alternativas arquitectónicas en una etapa en la que hacer cambios al diseño todavía es relativamente fácil y 3) reducir los riesgos asociados con la construcción del software.

Cita:

"Cásate con tu arquitectura de prisa, arrepiéntete en tu tiempo libre."

Barry Boehm

Esta definición pone el énfasis en el papel de los "componentes del software" en cualquier representación arquitectónica. En el contexto del diseño de la arquitectura, un componente del software puede ser algo tan simple como un módulo de programa o una clase orientada a objeto, pero también puede ampliarse para que incluya bases de datos y "middleware" que permitan la configuración de una red de clientes y servidores. Las propiedades de los componentes son aquellas características necesarias para entender cómo interactúan unos componentes con otros. En el nivel arquitectónico, no se especifican las propiedades internas (por ejemplo, detalles de un algoritmo). Las relaciones entre los componentes pueden ser tan simples como una invocación de procedimiento de un módulo a otro o tan complejos como un protocolo de acceso a una base de datos.

Ciertos miembros de la comunidad de la ingeniería de software [Kaz03] hacen una diferencia entre las acciones asociadas con la obtención de una arquitectura de software (lo que el autor llama "diseño de la arquitectura") y las que se aplican para obtener el diseño del software. Como dijo un revisor de esta edición:

> Hay una diferencia entre los términos *arquitectura* y *diseño*. Un *diseño* es una instancia de una *arquitectura*, similar a un objeto que es una instancia de una clase. Por ejemplo, considere la arquitectura de cliente-servidor. Con esta arquitectura es posible diseñar de muchos modos un sistema de software basado en red, con el uso de una plataforma Java (Java EE) o Microsoft (estructura .NET). Entonces, hay una arquitectura, pero con base en ella pueden crearse muchos diseños. Así, no es válido mezclar "arquitectura" y "diseño".

WebRef

En la dirección **www2.umassd.edu/SECenter/SAResources.html**, se encuentran apuntadores útiles para muchos sitios de arquitectura de software.

Aunque el autor está de acuerdo con que un diseño de software es una instancia de una arquitectura específica de software, los elementos y estructuras que se definen como parte de ésta son el origen de todo diseño que evolucione a partir de ellos. El diseño comienza con la consideración de la arquitectura.

En este libro, el diseño de la arquitectura de software considera dos niveles de la pirámide del diseño (véase la figura 8.1): el diseño de los datos y el de la arquitectura. En el contexto del análisis precedente, el diseño de los datos permite representar el componente de datos de la arquitectura con definiciones de sistemas y clase convencionales (que incluyen atributos y operaciones) en sistemas orientados a objeto. El diseño arquitectónico se centra en la representación de la estructura de los componentes del software, sus propiedades e interacciones.

9.1.2 ¿Por qué es importante la arquitectura?

Cita:

"La arquitectura es demasiado importante para dejarla en manos de una sola persona, no importa cuán brillante sea."

Scott Ambler

En un libro dedicado a la arquitectura del software, Bass *et al*. [Bas03] identifican tres razones clave por las que es importante la arquitectura del software:

- Las representaciones de la arquitectura del software permiten la comunicación entre todas las partes (participantes) interesadas en el desarrollo de un sistema basado en computadora.
- La arquitectura resalta las primeras decisiones que tendrán un efecto profundo en todo el trabajo de ingeniería de software siguiente y, también importante, en el éxito último del sistema como entidad operacional.
- La arquitectura "constituye un modelo relativamente pequeño y asequible por la vía intelectual sobre cómo está estructurado el sistema y la forma en la que sus componentes trabajan juntos" [Bas03].

El modelo del diseño de la arquitectura y los patrones arquitectónicos contenidos dentro de éste son transferibles. Es decir, los géneros, estilos y patrones arquitectónicos pueden aplicarse al diseño de otros sistemas y representan un conjunto de abstracciones que permite a los ingenieros de software describir la arquitectura en formas predecibles.

PUNTO CLAVE

El modelo arquitectónico da un punto de vista de la Gestalt del sistema, lo que permite que el ingeniero de software lo examine como un todo.

9.1.3 Descripciones arquitectónicas

Cada uno de nosotros tiene una imagen mental de lo que significa la palabra *arquitectura*. Sin embargo, la realidad es que tiene significados diferentes para distintas personas. La conclusión es que los diversos participantes verán una arquitectura desde puntos de vista diferentes motivados por varios conjuntos de preocupaciones. Esto implica que una descripción arquitectónica en realidad es un conjunto de productos del trabajo que reflejan puntos de vista distintos del sistema.

Por ejemplo, el arquitecto de un gran edificio de oficinas debe trabajar con distintos participantes. La preocupación principal del propietario de la edificación (un participante) es garantizar el placer estético y que brinde suficiente espacio de oficinas e infraestructura para garantizar su rentabilidad. Por tanto, el arquitecto debe desarrollar una descripción con el empleo de pers-

pectivas del edificio que se apeguen a las preocupaciones del dueño. Los puntos de vista empleados son dibujos del edificio en tres dimensiones (para ilustrar el aspecto estético) y un conjunto de planos en dos dimensiones que expliquen la preocupación por el espacio de oficinas y la infraestructura.

El esfuerzo debe centrarse en representaciones de la arquitectura que guiarán todos los demás aspectos del diseño. Hay que dedicar tiempo a revisar con cuidado la arquitectura. Un error aquí tendrá un efecto negativo a largo plazo.

Pero el espacio de oficinas tiene muchos otros participantes, incluido el fabricante de acero estructural que proveerá dicho material para la estructura del edificio. Necesita información arquitectónica detallada sobre el acero que soportará al edificio, incluso de las vigas tipo I, sus dimensiones, conectividad, materiales y muchos otros detalles. A estas preocupaciones se abocan diferentes productos del trabajo que representan distintos puntos de vista de la arquitectura. Los planos especializados (otro punto de vista) de la estructura de acero de la edificación se centran sólo en una de las muchas preocupaciones del fabricante.

La descripción de la arquitectura de un sistema basado en software debe tener características análogas a las mencionadas para el edificio de oficinas. Tyree y Ackerman [Tyr05] recalcan esto así: "Los desarrolladores desean lineamientos claros y decisivos sobre la forma de proceder con el diseño. Los consumidores desean la comprensión clara de los cambios ambientales que deben ocurrir y las garantías de que la arquitectura satisfará las necesidades de negocios. Otros arquitectos desean una comprensión clara y notable de los aspectos clave de la arquitectura." Cada uno de estos "deseos" se refleja en un punto de vista diferente representado con el uso de una perspectiva distinta.

IEEE Computer Society hizo la propuesta IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*, [IEE00], con los siguientes objetivos: 1) establecer un marco conceptual con un vocabulario que se use durante el diseño de la arquitectura del software, 2) proporcionar lineamientos detallados para representar una descripción arquitectónica y 3) estimular las mejores prácticas del diseño arquitectónico.

El estándar IEEE define una *descripción arquitectónica* (DA) como "un conjunto de productos para documentar una arquitectura". La descripción en sí se representa con el uso de perspectivas múltiples, donde cada *perspectiva* es "una representación del sistema completo desde el punto de vista de un conjunto de preocupaciones relacionadas [de un participante]". Una *perspectiva* se crea de acuerdo con reglas y convenciones definidas en un *punto de vista*: "especificación de las convenciones para construir y usar una perspectiva" [IEE00]. En este capítulo se estudian varios productos del trabajo que se utilizan para desarrollar distintas perspectivas de la arquitectura del software.

9.1.4 Decisiones arquitectónicas

Cada perspectiva desarrollada como parte de una descripción arquitectónica se aboca a una preocupación de un participante específico. Para desarrollar cada perspectiva (y la descripción arquitectónica en su conjunto), el arquitecto del sistema considera varias alternativas y decide en última instancia las características arquitectónicas específicas que satisfagan del mejor modo la preocupación. Entonces, las decisiones arquitectónicas en sí mismas se consideran una perspectiva de la arquitectura. Las razones por las que se tomaron las decisiones dan una visión de un sistema y su concordancia con las preocupaciones del participante.

Como arquitecto simbólico, el lector puede usar el formato sugerido en el recuadro para documentar cada decisión importante. Al hacerlo, presenta la racionalidad de su trabajo y deja un registro histórico que será útil cuando deban hacerse modificaciones del diseño.

9.2 Géneros arquitectónicos

Aunque los principios subyacentes del diseño arquitectónico se aplican a todos los tipos de la arquitectura, con frecuencia será el *género* arquitectónico el que dicte el enfoque específico para la estructura que deba construirse. En el contexto del diseño arquitectónico, el *género* implica

INFORMACIÓN

Formato para la descripción de una decisión arquitectónica

Toda decisión arquitectónica de importancia puede documentarse para que posteriormente la revisen los participantes que deseen entender la descripción de la arquitectura propuesta. El formato que se presenta en este recuadro es una versión adaptada y abreviada de otro, propuesto por Tyree y Ackerman [Tyr05].

Aspecto del diseño: Se describen los aspectos del diseño arquitectónico que se van a abordar.

Resolución: Se establece el enfoque escogido para abordar el aspecto de diseño.

Categoría: Se especifica la categoría de diseño a que se aboca el aspecto y la resolución (por ejemplo, diseño de datos, estructura del contenido, estructura del componente, integración, presentación, etcétera).

Suposiciones: Se indican cualesquiera suposiciones que ayuden a dar forma a la decisión.

Restricciones: Se especifican todas las restricciones ambientales que ayuden a conformar la decisión (como los estándares tecnológicos, patrones disponibles y aspectos relacionados con el diseño).

Alternativas: Se describen con brevedad las alternativas de diseño arquitectónico que se consideraron y la razón por la que se rechazaron.

Argumento: Se establece por qué se eligió la resolución sobre las demás alternativas.

Implicaciones: Se indican las consecuencias que tendrá la toma de la decisión en el diseño. ¿Cómo afectará la resolución a otros aspectos del diseño de la arquitectura? ¿La resolución restringe de algún modo al diseño?

Decisiones relacionadas: ¿Qué otros documentos se relacionan con esta decisión?

Preocupaciones relacionadas: ¿Qué otros requerimientos se relacionan con esta decisión?

Productos finales: Se indica dónde se reflejará esta decisión en la descripción arquitectónica.

Notas: Se hace referencia a las anotaciones del equipo u otra clase de documentación que se haya empleado para tomar la decisión.

Diversos estilos arquitectónicos pueden aplicarse a un género específico (también conocido como dominio de aplicación).

una categoría específica dentro del dominio general del software. Dentro de cada categoría hay varias subcategorías. Por ejemplo, dentro del género *edificios*, se encuentran los siguientes estilos generales: casas, condominios, edificios de departamentos, edificios de oficinas, edificios industriales, bodegas, etc. Dentro de cada estilo general habrá estilos más específicos (véase la sección 9.3). Cada estilo tendrá una estructura que puede describirse con el uso de un conjunto de patrones predecibles.

En su texto evolutivo *Handbook of Software Architecture* [Boo08], Grady Booch sugiere los siguientes géneros arquitectónicos para sistemas basados en software:

- **Inteligencia artificial:** Sistemas que simulan o incrementan la cognición humana, su locomoción u otros procesos orgánicos.
- **Comerciales y no lucrativos:** Sistemas que son fundamentales para la operación de una empresa de negocios.
- **Comunicaciones:** Sistemas que proveen la infraestructura para transferir y manejar datos, para conectar usuarios de éstos o para presentar datos en la frontera de una infraestructura.
- **Contenido de autor:** Sistemas que se emplean para crear o manipular artefactos de texto o multimedios.
- **Dispositivos:** Sistemas que interactúan con el mundo físico a fin de brindar algún servicio puntual a un individuo.
- **Entretenimiento y deportes:** Sistemas que administran eventos públicos o que proveen una experiencia grupal de entretenimiento.
- **Financieros:** Sistemas que proporcionan la infraestructura para transferir y manejar dinero y otros títulos.
- **Juegos:** Sistemas que dan una experiencia de entretenimiento a individuos o grupos.

Cita:

"Programar sin una arquitectura o diseño general en mente es como explorar una caverna sólo con una linterna: no sabes dónde has estado, a dónde vas ni dónde estás."

Danny Thorpe

- **Gobierno:** Sistemas que dan apoyo a la conducción y operaciones de una institución política local, estatal, federal, global o de otro tipo.
- **Industrial:** Sistemas que simulan o controlan procesos físicos.
- **Legal:** Sistemas que dan apoyo a la industria jurídica.
- **Médicos:** Sistemas que diagnostican, curan o contribuyen a la investigación médica.
- **Militares:** Sistemas de consulta, comunicaciones, comando, control e inteligencia (o C4I), así como de armas ofensivas y defensivas.
- **Sistemas operativos:** Sistemas que están inmediatamente instalados en el hardware para dar servicios de software básico.
- **Plataformas:** Sistemas que se encuentran en los sistemas operativos para brindar servicios avanzados.
- **Científicos:** Sistemas que se emplean para hacer investigación científica y aplicada.
- **Herramientas:** Sistemas que se utilizan para desarrollar otros sistemas.
- **Transporte:** Sistemas que controlan vehículos acuáticos, terrestres, aéreos o espaciales.
- **Utilidades:** Sistemas que interactúan con otro software para brindar algún servicio específico.

Desde el punto de vista del diseño arquitectónico, cada género representa un desafío único. Por ejemplo, considere la arquitectura del software de un sistema de juego. Esta clase de sistemas, en ocasiones llamados *aplicaciones interactivas de inmersión*, requieren el cómputo de algoritmos intensivos, gráficas avanzadas en computadora, fuentes de datos continuas en multimedios, interactividad en tiempo real a través de dispositivos de entrada convencionales y no convencionales, y otras preocupaciones especializadas.

Alexander Francois [Fra03] sugiere una arquitectura del software para *inmerpresencia*[1] que se aplica a un ambiente de juegos. Él describe la arquitectura de la manera siguiente:

> La ASI (Arquitectura de Software de Inmerpresencia) es un modelo nuevo de arquitectura de software para diseñar, analizar e implementar aplicaciones que realizan un procesamiento distribuido, asíncrono y paralelo de flujos de datos generales. El objetivo de la ASI es proveer un marco universal para la implementación distribuida de algoritmos y su fácil integración en sistemas complejos [...] El modelo de procesamiento de datos extensibles subyacentes y el modelo de procesamiento híbrido (depósito y transmisión de mensajes compartidos), asíncrono y en paralelo permiten la manipulación natural y eficiente de flujos de datos generales con el uso indistinto de librerías ya existentes o código original. La modularidad del estilo facilita el desarrollo de código distribuido, pruebas y reutilización, así como el diseño e integración rápida del sistema y su mantenimiento y evolución.

El análisis detallado del ASI queda fuera del alcance de este libro. No obstante, es importante reconocer que el género de sistemas de juegos puede abordarse con un estilo arquitectónico (véase la sección 9.3) diseñado específicamente para resolver preocupaciones de sistemas de juego. Si el lector tiene especial interés, consulte [Fra03].

9.3 Estilos arquitectónicos

Cuando un constructor usa la frase "vestíbulo central colonial" para describir una casa, la mayor parte de personas familiarizadas con viviendas en Estados Unidos se hará una imagen general de ella y de cuál es su probable distribución. El constructor usó un *estilo arquitectónico* como mecanismo descriptivo para diferenciar la casa de otros estilos (por ejemplo, de dos aguas, finca

1 Francois utiliza el término *inmerpresencia* para denotar aplicaciones interactivas de inmersión.

Cita:

"En el fondo de la mente de todo artista hay un patrón o tipo de arquitectura."

G. K. Chesterton

campestre, Cabo Cod, etc.). Pero, lo que es más importante, el estilo arquitectónico también es una plantilla para la construcción. Deben definirse más detalles, especificar sus dimensiones finales, agregar características personalizadas, determinar los materiales de construcción, pero el estilo (un "vestíbulo central colonial") orienta al constructor en su trabajo.

El software construido para sistemas basados en computadora también tiene uno de muchos estilos arquitectónicos. Cada estilo describe una categoría de sistemas que incluye 1) un conjunto de componentes (como una base de datos o módulos de cómputo) que realizan una función requerida por el sistema, 2) un conjunto de conectores que permiten la "comunicación, coordinación y cooperación" entre los componentes, 3) restricciones que definen cómo se integran los componentes para formar el sistema y 4) modelos semánticos que permiten que un diseñador entienda las propiedades generales del sistema al analizar las propiedades conocidas de sus partes constituyentes [Bas03].

¿Qué es un estilo arquitectónico?

Un estilo arquitectónico es una transformación que se impone al diseño de todo el sistema. El objetivo es establecer una estructura para todos los componentes del sistema. En el caso en el que ha de hacerse la reingeniería de una arquitectura ya existente (véase el capítulo 29), la imposición de un estilo arquitectónico dará como resultado cambios fundamentales en la estructura del software, incluida la reasignación de las funciones de los componentes [Bos00].

WebRef

Los estilos arquitectónicos basados en atributos (ABAS) pueden usarse como bloques de construcción para las arquitecturas de software. En la dirección **www.sei.cmu.edu/architecture/abas.html**, hay información al respecto.

Un patrón arquitectónico, como un estilo de arquitectura, impone la transformación del diseño de una arquitectura. Sin embargo, un patrón difiere de un estilo en varias formas fundamentales: 1) el alcance del patrón es menos amplio y se centra en un aspecto de la arquitectura más que en el total de ésta, 2) un patrón impone una regla a la arquitectura, describe la manera en la que el software manejará ciertos aspectos de su funcionalidad en el nivel de la infraestructura (por ejemplo, la concurrencia) [Bos00], 3) los patrones arquitectónicos (véase la sección 9.4) tienden a abocarse a aspectos específicos del comportamiento en el contexto de la arquitectura (por ejemplo, cómo manejarán la sincronización o las interrupciones las aplicaciones en tiempo real). Los patrones se utilizan junto con un estilo arquitectónico para dar forma a la estructura

INFORMACIÓN

Estructuras arquitectónicas canónicas

En esencia, la arquitectura del software representa una estructura en la que cierta colección de entidades (con frecuencia llamados *componentes*) está conectada por un conjunto de relaciones definidas (usualmente llamadas *conectores*). Tanto las componentes como los conectores están asociados con un conjunto de *propiedades* que permiten que el diseñador diferencie los tipos de componentes y conectores que pueden usarse. Pero, ¿qué clases de estructuras (componentes, conectores y propiedades) se utilizan para describir una arquitectura? Bass y Kazman [Bas03] sugieren cinco estructuras canónicas o fundamentales:

Estructura funcional. Los componentes representan entidades de función o procesamiento. Los conectores representan interfaces que proveen la capacidad de "usar" o "pasar datos a" un componente. Las propiedades describen la naturaleza de los componentes y la organización de las interfaces.

Estructura de implementación. "Los componentes son paquetes, clases, objetos, procedimientos, funciones, métodos, etc., que son vehículos para empacar funciones en varios niveles de abstracción" [Bas03]. Los conectores incluyen la capacidad de pasar datos y control, compartir datos, "usar" y "ser una instancia de". Las propiedades se centran en las características de la calidad (por ejemplo, facilidad de recibir mantenimiento, ser reutilizables, etc.) que surgen cuando se implementa la estructura.

Estructura de concurrencia. Los componentes representan "unidades de concurrencia" que están organizadas como tareas o trayectorias paralelas. "Las relaciones [conectores] incluyen sincronizarse-con, tiene-mayor-prioridad-que, envía-datos-a, no-corre-sin y no-corre-con. Las propiedades relevantes para esta estructura incluyen prioridad, anticipación y tiempo de ejecución" [Bas03].

Estructura física. Esta estructura es similar al modelo de despliegue desarrollado como parte del diseño. Los componentes son el hardware físico en el que reside el software. Los conectores son las interfaces entre los componentes del hardware y las propiedades incluyen la capacidad, ancho de banda y rendimiento, entre otros atributos.

Estructura de desarrollo. Esta estructura define los componentes, productos del trabajo y otras fuentes de información que se requieren a medida que avanza la ingeniería de software. Los conectores representan las relaciones entre los productos del trabajo; las propiedades identifican las características de cada aspecto.

Cada una de estas estructuras presenta un punto de vista de la arquitectura del software y expone información que es útil para el equipo de software a medida que realiza la modelación y construcción.

general de un sistema. En la sección 9.3.1 se consideran los estilos y patrones arquitectónicos comunes para el software.

9.3.1 Breve taxonomía de estilos de arquitectura

Cita:

"El uso de patrones y estilos de diseño está presente en todas las disciplinas de la ingeniería."

Mary Shaw y David Garlan

Aunque en los últimos 60 años se han creado millones de sistemas basados en computadoras, la gran mayoría se clasifica en un número relativamente pequeño de estilos de arquitectura:

Arquitecturas centradas en los datos. En el centro de esta arquitectura se halla un almacenamiento de datos (como un archivo o base de datos) al que acceden con frecuencia otros componentes que actualizan, agregan, eliminan o modifican los datos de cierto modo dentro del almacenamiento. La figura 9.1 ilustra un estilo común centrado en datos. El software cliente accede a un repositorio (depósito) central. En ciertos casos éste es pasivo. Es decir, el software cliente accede a los datos en forma independiente de cualesquiera cambios que éstos sufran o de las acciones del software de otro cliente. Una variante de este enfoque transforma el depósito en un "pizarrón" que envía notificaciones al software cliente cuando hay un cambio en datos de interés del cliente.

Las arquitecturas centradas en datos promueven la *integrabilidad* [Bas03]. Es decir, los componentes del software pueden ser cambiados y agregarse otros nuevos, del cliente, a la arquitectura sin problemas con otros clientes (porque los componentes del cliente operan en forma independiente). Además, pueden pasarse datos entre clientes con el uso de un mecanismo de pizarrón (el componente pizarrón sirve para coordinar la transferencia de información entre clientes). Los componentes del cliente ejecutan los procesos con independencia.

Arquitecturas de flujo de datos. Esta arquitectura se aplica cuando datos de entrada van a transformarse en datos de salida a través de una serie de componentes computacionales o manipuladores. Un patrón de tubo y filtro (véase la figura 9.2) tiene un conjunto de componentes, llamados *filtros*, conectados por *tubos* que transmiten datos de un componente al siguiente. Cada filtro trabaja en forma independiente de aquellos componentes que se localizan arriba o abajo del flujo; se diseña para esperar una entrada de datos de cierta forma y produce datos de salida (al filtro siguiente) en una forma especificada. Sin embargo, el filtro no requiere ningún conocimiento de los trabajos que realizan los filtros vecinos.

Si el flujo de datos degenera en una sola línea de transformaciones, se denomina lote secuencial. Esta estructura acepta un lote de datos y luego aplica una serie de componentes secuenciales (filtros) para transformarlos.

FIGURA 9.1
Arquitectura centrada en datos

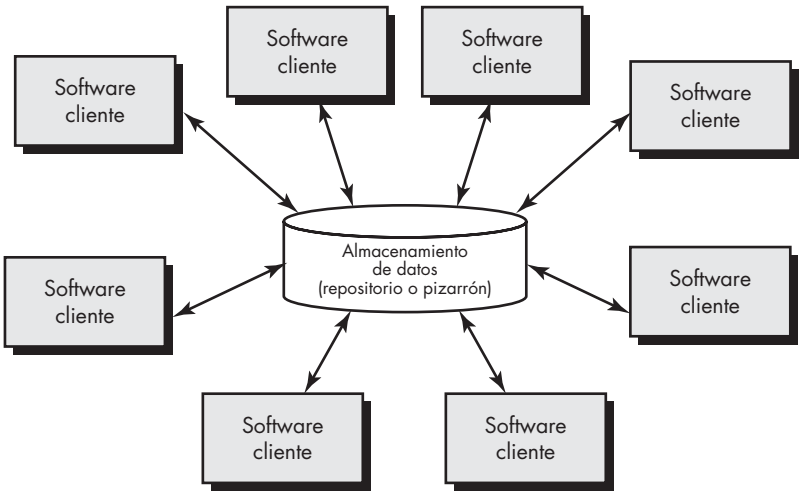
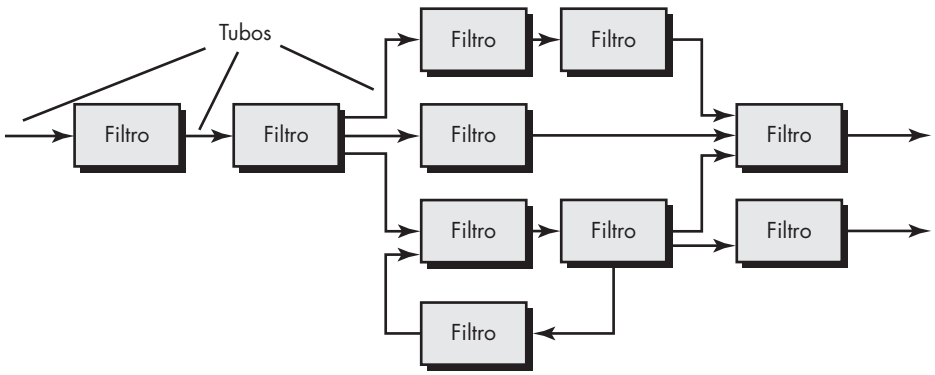


FIGURA 9.2
Arquitectura de flujo de datos



Tubos y filtros

Arquitecturas de llamar y regresar. Este estilo arquitectónico permite obtener una estructura de programa que es relativamente fácil de modificar y escalar. Dentro de esta categoría existen varios subestilos [Bas03]:

- *Arquitecturas de programa principal/subprograma*. Esta estructura clásica de programa descompone una función en una jerarquía de control en la que un programa "principal" invoca cierto número de componentes de programa que a su vez invocan a otros. La figura 9.3 ilustra una arquitectura de este tipo.
- *Arquitecturas de llamada de procedimiento remoto.* Los componentes de una arquitectura de programa principal/subprograma están distribuidos a través de computadoras múltiples en una red.

Arquitecturas orientadas a objetos. Los componentes de un sistema incluyen datos y las operaciones que deben aplicarse para manipularlos. La comunicación y coordinación entre los componentes se consigue mediante la transmisión de mensajes.

Arquitecturas en capas. En la figura 9.4 se ilustra la estructura básica de una arquitectura en capas. Se define un número de capas diferentes; cada una ejecuta operaciones que se aproximan progresivamente al conjunto de instrucciones de máquina. En la capa externa, los compo-

FIGURA 9.3
Arquitectura de programa principal/subprograma

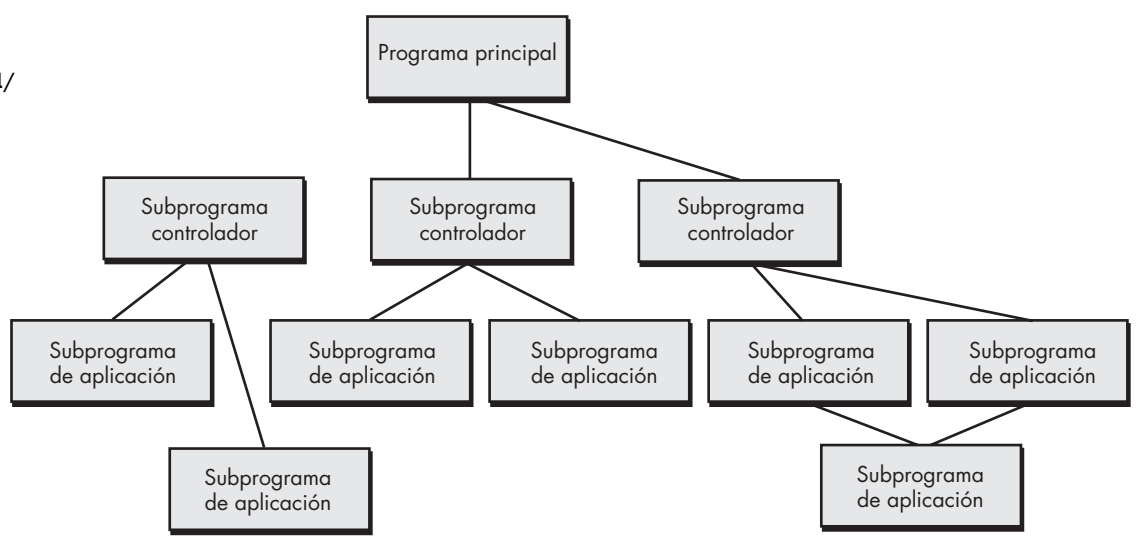
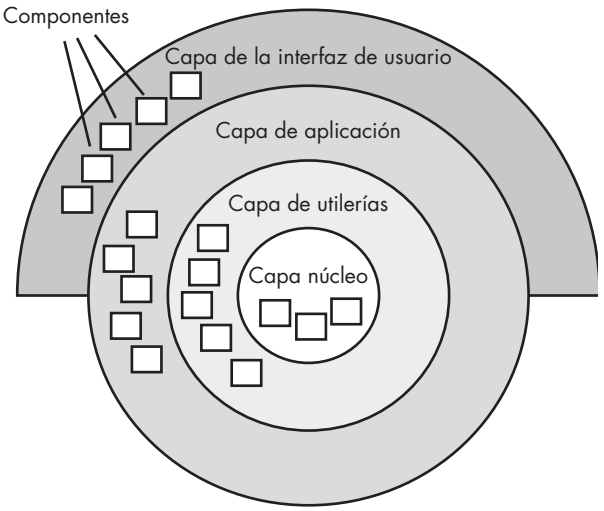


FIGURA 9.4
Arquitectura en capas



nentes atienden las operaciones de la interfaz de usuario. En la interna, los componentes realizan la interfaz con el sistema operativo. Las capas intermedias proveen servicios de utilerías y funciones de software de aplicación.

Estos estilos arquitectónicos tan sólo son un pequeño subconjunto de los que están disponibles.[2] Una vez que la ingeniería de requerimientos revela las características y restricciones del sistema que se va a elaborar, se elige el estilo arquitectónico o la combinación de patrones que se ajusten mejor a esas características y restricciones. En muchos casos, más de un patrón es apropiado y es posible diseñar y evaluar estilos arquitectónicos alternativos. Por ejemplo, en muchas aplicaciones de bases de datos se combina un estilo en capas (apropiado para la mayoría de sistemas) con una arquitectura centrada en datos.

9.3.2 Patrones arquitectónicos

Cita:

"Tal vez está en la planta baja. Déjame subir a revisar."

M. C. Escher

A medida que se desarrolle el modelo de requerimientos, se observará que el software debe enfrentar cierto número de problemas amplios que abarcan toda la aplicación. Por ejemplo, el modelo de requerimientos para virtualmente cualquier aplicación de comercio electrónico se enfrenta con el siguiente problema: *¿Cómo ofrecer una amplia variedad de bienes a un grupo extenso de consumidores y permitir que los adquieran en línea?*

El modelo de requerimientos también define un contexto en el que debe responderse esta pregunta. Por ejemplo, un negocio de comercio electrónico que venda equipo de golf de consumo operará en un contexto diferente que otro que venda equipo industrial de precio alto a corporaciones medianas y pequeñas. Además, hay varias limitantes y restricciones que afectan la manera en la que se aborda este problema para resolverlo.

Los patrones arquitectónicos se abocan a un problema de aplicación específica dentro de un contexto dado y sujeto a limitaciones y restricciones. El patrón propone una solución arquitectónica que sirve como base para el diseño de la arquitectura.

En una sección anterior, se dijo que la mayor parte de aplicaciones caen dentro de un dominio o género específico, y que para éstas son apropiados uno o más estilos de arquitectura. Por ejemplo, el estilo de arquitectura general para una aplicación podría ser el de llamar y regresar o el que está orientado a objeto. Pero dentro de ese estilo se encontrará un conjunto de problemas comu-

2 Para un estudio detallado de estilos y patrones arquitectónicos, véase [Bus07], [Gor06], [Roz05], [Bas03], [Bos00] y [Hof00].

CasaSegura

Elección de un estilo de arquitectura

La escena: Cubículo de Jamie, cuando comienza la modelación del diseño.

Participantes: Jamie y Ed, miembros del equipo de ingeniería de software de *CasaSegura*.

La conversación:

Ed (frunce el ceño): Hemos estado modelando la función de seguridad con el empleo de UML, ya sabes, clases, relaciones y demás. Así que supongo que la arquitectura orientada a objeto[3] es la elección apropiada.

Jamie: ¿Pero…?

Ed: Pero… tengo problemas para visualizar lo que es una arquitectura orientada a objeto. Entiendo la arquitectura de llamar y regresar, que es algo así como una jerarquía de proceso convencional, pero la orientada a objeto… no sé, parece algo amorfo.

Jamie (sonríe): Amorfo, ¿eh?

Ed: Sí… lo que quiero decir es que no visualizo una estructura real, sólo clases de diseño que flotan en el espacio.

Jamie: Bueno, eso no es cierto. Hay jerarquías de clase… piensa en la jerarquía (agregado) que hicimos para el objeto **Plano** [figura 8.3]. Una arquitectura orientada a objetos es una combinación de esta estructura y de las interconexiones —ya sabes, colaboraciones— entre las clases. Puede mostrarse con la descripción completa de los atributos y operaciones, los mensajes que hay y la estructura de las clases.

Ed: Voy a dedicar una hora a mapear una arquitectura de llamar y regresar; luego volveré a considerar la orientada a objeto.

Jamie: Doug no tendrá problemas con eso. Dijo que deben considerarse alternativas arquitectónicas. Por cierto, no hay ninguna razón para no usar estas arquitecturas combinadas entre sí.

Ed: Bueno. Estoy en eso.

nes que se abordan mejor con patrones arquitectónicos específicos. En el capítulo 12 se presentan algunos de estos problemas y se hace un estudio más completo de los patrones de arquitectura.

9.3.3 Organización y refinamiento

Debido a que es frecuente que el proceso de diseño permita varias alternativas de arquitectura, es importante establecer un conjunto de criterios de diseño que se usan para evaluar el diseño arquitectónico obtenido. Las preguntas que siguen [Bas03] dan una visión del estilo de arquitectura:

? ¿Cómo se evalúa el estilo arquitectónico que se haya obtenido?

Control. ¿Cómo se administra el control dentro de la arquitectura? ¿Existe una jerarquía de control distinta? Si es así, ¿cuál es el papel de los componentes dentro de esta jerarquía de control? ¿Cómo transfieren el control los componentes dentro del sistema? ¿Cómo lo comparten? ¿Cuál es la topología del control (por ejemplo, la forma geométrica que adopta el control)? ¿El control está sincronizado o los componentes operan en forma asincrónica?

Datos. ¿Cómo se comunican los datos entre los componentes? ¿El flujo de datos es continuo o los objetos de datos pasan al sistema en forma esporádica? ¿Cuál es el modo de transferencia de datos (pasan de un componente a otro o se dispone de ellos globalmente para compartirse entre los componentes del sistema)? ¿Existen componentes de datos (como un pizarrón o depósito) y, si así fuera, cuál es su papel? ¿Cómo interactúan los componentes funcionales con los componentes de datos? ¿Los componentes de datos son pasivos o activos (el componente de datos actúa activamente con otros componentes del sistema)? ¿Cómo interactúan los datos y el control dentro del sistema?

---

3 Podría afirmarse que la arquitectura de *CasaSegura* debe considerarse en un nivel más alto que la que se comenta. *CasaSegura* tiene varios subsistemas: vigilancia de las funciones del hogar, vigilancia del sitio de la empresa y el subsistema que corre en la PC del propietario. Dentro de los subsistemas son comunes los procesos concurrentes (por ejemplo, los que vigilan sensores) y los que manejan eventos. En este nivel, algunas decisiones de arquitectura se toman durante la ingeniería del producto, pero el diseño arquitectónico dentro de la ingeniería de software muy bien debe considerar estos aspectos.

Estas preguntas dan al diseñador una evaluación temprana de la calidad del diseño y proporcionan el fundamento para hacer análisis más detallados de la arquitectura.

9.4 Diseño arquitectónico

Cita:

"Un doctor puede sepultar sus errores, pero un arquitecto sólo puede aconsejar a su cliente que siembre enredaderas."

Frank Lloyd Wright

Cuando comienza el diseño arquitectónico, el software que se va a desarrollar debe situarse en contexto, es decir, el diseño debe definir las entidades externas (otros sistemas, dispositivos, personas, etc.) con las que interactúa el software y la naturaleza de dicha interacción. Esta información por lo general se adquiere a partir del modelo de los requerimientos y de toda la que se reunió durante la ingeniería de éstos. Una vez que se ha modelado el contexto y descrito todas las interfaces externas del software, se identifica un conjunto de arquetipos de arquitectura. Un *arquetipo* es una abstracción (similar a una clase) que representa un elemento de comportamiento del sistema. El conjunto de arquetipos provee una colección de abstracciones que deben modelarse en cuanto a la arquitectura si el sistema ha de construirse, pero los arquetipos por sí mismos no dan suficientes detalles para la implementación. Por tanto, el diseñador especifica la estructura del sistema, definiendo y refinando los componentes del software que implementan cada arquetipo. Este proceso sigue en forma iterativa hasta que se obtiene una estructura arquitectónica completa. En las secciones que siguen se estudia cada una de estas tareas de diseño arquitectónico con un poco más de detalle.

9.4.1 Representación del sistema en contexto

PUNTO CLAVE

El contexto arquitectónico representa la manera en la que el software interactúa con las entidades externas a sus fronteras.

En el nivel de diseño arquitectónico, el arquitecto del software usa un *diagrama de contexto arquitectónico* (DCA) para modelar la manera en la que el software interactúa con entidades más allá de sus fronteras. La estructura general del diagrama de contexto arquitectónico se ilustra en la figura 9.5.

? ¿Cómo interactúan los sistemas uno con el otro?

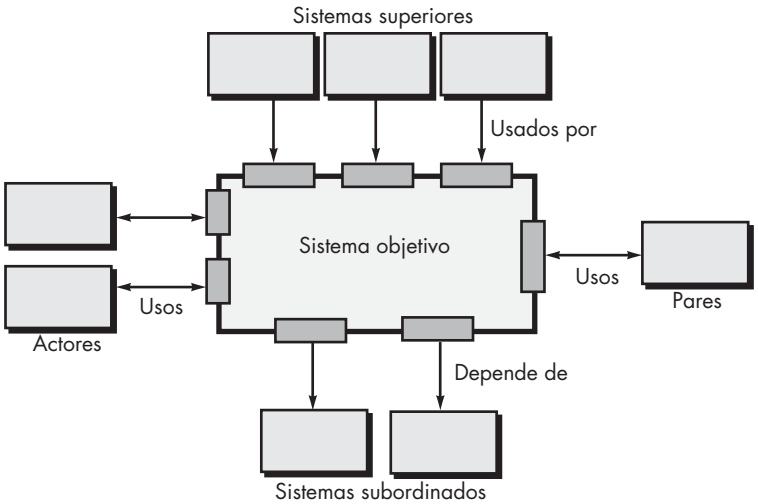
En relación con dicha figura, los sistemas que interactúan con el *sistema objetivo* (aquel para el que va a desarrollarse un diseño arquitectónico) están representados como sigue:

- *Sistemas superiores*: aquellos que utilizan al sistema objetivo como parte de algún esquema de procesamiento de alto nivel.
- *Sistemas subordinados*: los que son usados por el sistema objetivo y proveen datos o procesamiento que son necesarios para completar las funciones del sistema objetivo.
- *Sistemas entre iguales*: son los que interactúan sobre una base de igualdad (por ejemplo, la información se produce o consume por los iguales y por el sistema objetivo).
- *Actores*: entidades (personas, dispositivos, etc.) que interactúan con el sistema objetivo mediante la producción o consumo de información que es necesaria para el procesamiento de los requerimientos.

Cada una de estas entidades externas se comunica con el sistema objetivo a través de una interfaz (rectángulos pequeños sombreados).

Para ilustrar el empleo del DCA, considere la función de seguridad del hogar del producto *CasaSegura*. El controlador general del producto *CasaSegura* y el sistema basado en internet son superiores respecto de la función de seguridad y se muestran por arriba de la función en la figura 9.6. La función de vigilancia es un *sistema entre iguales* y en las versiones posteriores del producto usa (es usada por) la función de seguridad del hogar. El propietario y los paneles de control son actores que producen y consumen información usada o producida por el software de seguridad. Por último, los sensores se utilizan por el software de seguridad y se muestran como subordinados a éste.

FIGURA 9.5
Diagrama de contexto arquitectónico
Fuente: Adaptado de [Bos00].



Como parte del diseño arquitectónico, tendrían que especificarse los detalles de cada interfaz mostrada en la figura 9.6. En esta etapa deben identificarse todos los datos que fluyen hacia dentro y fuera del sistema objetivo.

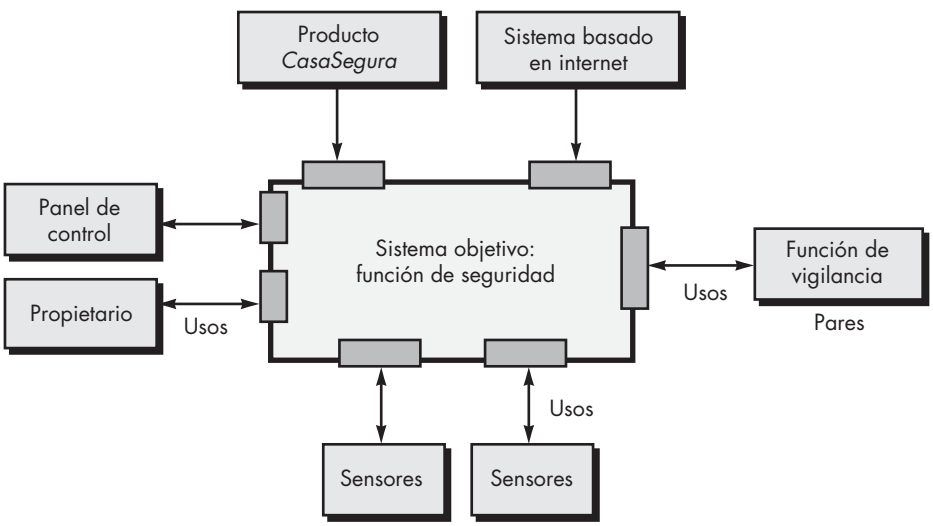
9.4.2 Definición de arquetipos

PUNTO CLAVE
Los arquetipos son los bloques constructivos de un diseño arquitectónico.

Un *arquetipo* es una clase o un patrón que representa una abstracción fundamental de importancia crítica para el diseño de una arquitectura para el sistema objetivo. En general, se requiere de un conjunto relativamente pequeño de arquetipos a fin de diseñar sistemas incluso algo complejos. La arquitectura del sistema objetivo está compuesta de estos arquetipos, que representan elementos estables de la arquitectura, pero que son implementadas en muchos modos diferentes con base en el comportamiento del sistema.

En muchos casos, los arquetipos se obtienen con el estudio de las clases de análisis definidas como parte del modelo de los requerimientos. Al continuar el análisis de la función de seguridad de *CasaSegura*, se definirán los arquetipos siguientes:

FIGURA 9.6
Diagrama de contexto arquitectónico para la función de seguridad de *CasaSegura*



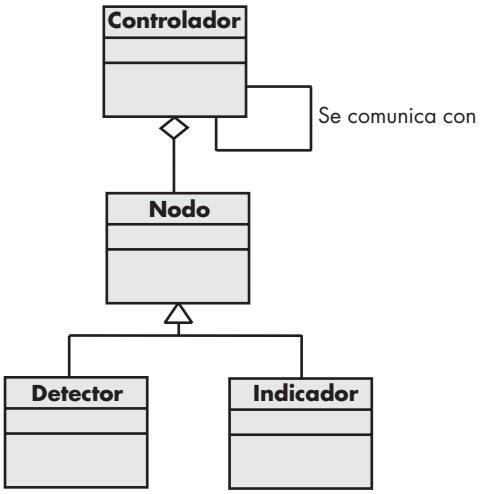


FIGURA 9.7
Relaciones de UML para los arquetipos de la función de seguridad de *CasaSegura*
Fuente: Adaptado de [Bos00].

- **Nodo.** Representa una colección cohesiva de elementos de entrada y salida de la función de seguridad del hogar. Por ejemplo, un nodo podría comprender 1) varios sensores y 2) varios indicadores de alarma (salida).
- **Detector.** Abstracción que incluye todos los equipos de detección que alimentan con información al sistema objetivo.
- **Indicador.** Abstracción que representa todos los mecanismos (como la sirena de la alarma, luces, campana, etc.) que indican que está ocurriendo una condición de alarma.
- **Controlador.** Abstracción que ilustra el mecanismo que permite armar o desarmar un nodo. Si los controladores residen en una red, tienen la capacidad de comunicarse entre sí.

En la figura 9.7 se muestra cada uno de estos arquetipos con el empleo de notación UML. Recuerde que los arquetipos constituyen la base de la arquitectura, pero son abstracciones que deben refinarse a medida que avanza el diseño arquitectónico. Por ejemplo, **Detector** se refinaría en una jerarquía de clase de sensores.

9.4.3 Refinamiento de la arquitectura hacia los componentes

Cita:

"La estructura de un sistema de software provee la ecología en la que el código nace, crece y muere. Un hábitat bien diseñado permite la evolución exitosa de todos los componentes necesarios en un sistema de software."

R. Pattis

Conforme la arquitectura se refina hacia los componentes, comienza a emerger la estructura del sistema. Pero, ¿cómo se eligen estos componentes? Para responder esta pregunta se comienza con las clases descritas como parte del modelo de requerimientos.[4] Estas clases de análisis representan entidades dentro del dominio de aplicación (negocio) que deben enfrentarse dentro de la arquitectura del software. Entonces, el dominio de aplicación es una fuente para la obtención y refinamiento de los componentes. Otra fuente es el dominio de la infraestructura. La arquitectura debe albergar muchas componentes de la infraestructura que hagan posible los componentes de la aplicación, pero que no tengan conexión con el dominio de ésta. Por ejemplo, los componentes de administración de memoria, de comunicación, de base de datos y de administración de tareas con frecuencia están integrados en la arquitectura del software.

Las interfaces ilustradas en el diagrama de contexto de la arquitectura (sección 9.4.1) implican uno o más componentes especializados que procesan los datos que fluyen a través de la

4 Si se elige un enfoque convencional (no orientado a objeto), los componentes se obtienen a partir de los datos del modelo del flujo. Este enfoque se estudia brevemente en la sección 9.6.

interfaz. En ciertos casos (por ejemplo, una interfaz de usuario gráfica) debe diseñarse una arquitectura completa con muchos componentes para el subsistema.

Al seguir con el ejemplo de la función de seguridad de *CasaSegura*, debe definirse el conjunto de componentes de alto nivel que se aboque a las funciones siguientes:

- *Administración de la comunicación externa*: coordina la comunicación de la función de seguridad con entidades externas, tales como otros sistemas basados en internet y la notificación externa de una alarma.
- *Procesamiento del panel de control*: administra toda la funcionalidad del panel de control.
- *Administración de detectores*: coordina el acceso a todos los detectores del sistema.
- *Procesamiento de alarmas*: verifica y actúa en todas las condiciones de alarma.

Cada uno de estos componentes de alto nivel tendría que elaborarse en forma iterativa para después posicionarlo dentro de la arquitectura general de *CasaSegura*. Para cada uno se definirían las clases de diseño (con los atributos y operaciones apropiadas). Sin embargo, es importante observar que los detalles de diseño de todos los atributos y operaciones no se especificarían hasta abordar el diseño en el nivel de componentes (véase el capítulo 10).

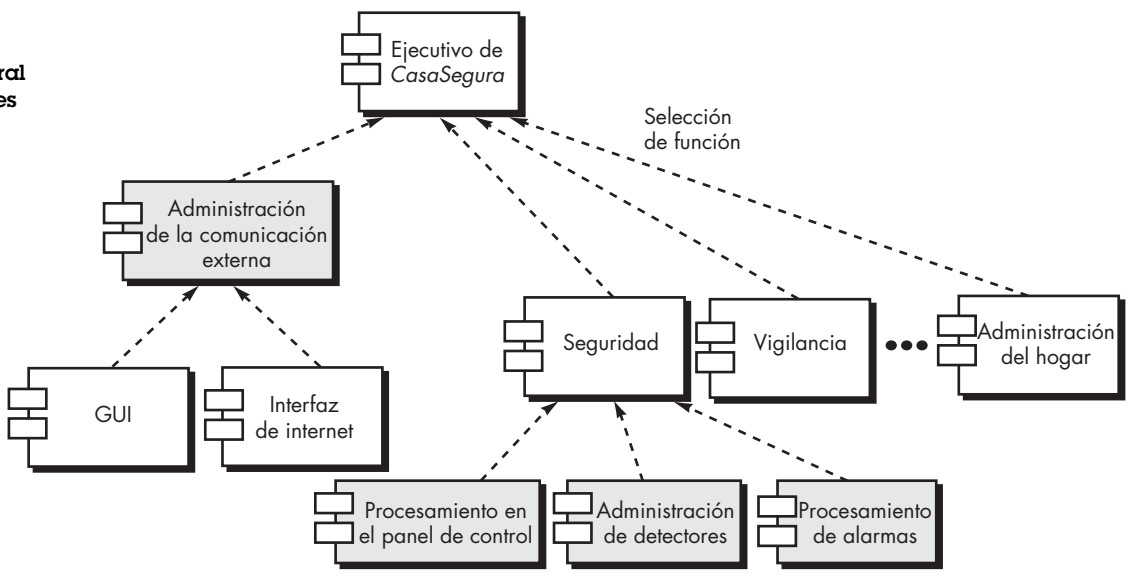
La estructura arquitectónica general (representada como diagrama de componentes UML) se ilustra en la figura 9.8. Las transacciones son adquiridas por *administración de la comunicación externa* a medida que se mueven desde los componentes que procesan la GUI de *CasaSegura* y la interfaz de internet. Esta información es administrada por un componente ejecutivo de *CasaSegura* que selecciona la función apropiada del producto (la seguridad, en este caso). El componente *procesamiento en el panel de control* interactúa con el propietario para activar o desactivar la función de seguridad. El componente *administración de detectores* prueba los sensores para detectar una condición de alarma y el componente *procesamiento de alarmas* produce la salida cuando se detecta una alarma.

9.4.4 Descripción de las instancias del sistema

El diseño arquitectónico modelado hasta este momento todavía es de nivel relativamente alto. Se ha representado el contexto del sistema, se definieron los arquetipos que indican las abstracciones importantes dentro del dominio del problema, es visible la estructura general del sistema

FIGURA 9.8

Estructura arquitectónica general para los componentes de alto nivel de *CasaSegura*



y están identificadas las componentes principales del software. Sin embargo, es necesario más refinamiento (recuerde que todo el diseño es iterativo).

Para lograr esto, se desarrollan instancias de la arquitectura. Esto significa que la arquitectura se aplica a un problema específico con objeto de demostrar que tanto ella como sus componentes son apropiados.

La figura 9.9 ilustra instancias de la arquitectura de *CasaSegura* para el sistema de seguridad. Los componentes ilustrados en la figura 9.8 se elaboraron para mostrar más detalles. Por ejemplo, el componente *administración de detectores* interactúa con un componente de infraestructura *programador* que implementa la prueba de cada objeto *sensor* usado por el sistema de seguridad. Una elaboración similar se lleva a cabo para cada uno de los componentes representados en la figura 9.8.

HERRAMIENTAS DE SOFTWARE

Diseño arquitectónico

Objetivo: Las herramientas de diseño arquitectónico modelan la estructura general del software mediante la representación de la interfaz de los componentes, de sus dependencias y relaciones, así como de sus interacciones.

Mecánica: La mecánica de las herramientas varía. En la mayoría de casos, la capacidad de diseño de la arquitectura es parte de las funciones provistas por herramientas automatizadas para el modelado del análisis y el diseño.

Herramientas representativas:[5]

Adalon, desarrollada por Synthis Corp. (**www.synthis.com**), es una herramienta de diseño especializada para diseñar y construir arquitecturas de componentes específicos basados en web.

ObjectiF, desarrollada por microTOOL GmbH (**www.microtool.de/objectiF/en/**), es una herramienta de diseño basada en UML que conduce a arquitecturas (como Coldfusion, J2EE, Fusebox, etc.) adecuadas para la ingeniería de software basada en componentes (véase el capítulo 29).

Rational Rose, desarrollada por Rational (**www-306.ibm.com/software/rational/**), es una herramienta de diseño basada en UML que proporciona apoyo a todos los aspectos del diseño de la arquitectura.

9.5 EVALUACIÓN DE LOS DISEÑOS ALTERNATIVOS PARA LA ARQUITECTURA

En su libro sobre evaluación de las arquitecturas de software, Clements *et al.* [Cle03] afirman lo siguiente:

> Para decirlo sin rodeos, una arquitectura es una apuesta, una adivinanza sobre el éxito de un sistema. ¿No sería agradable saber por adelantado si se apostó a un ganador en vez de esperar hasta que el sistema esté casi terminado para saber si cumplirá con los requerimientos o no? Si usted compra un sistema o paga por su desarrollo, ¿no querría tener alguna seguridad de que va por el camino correcto? Si usted mismo es el arquitecto, ¿no le gustaría tener una buena manera de validar sus intuiciones y experiencia para que pudiera dormir por la noche con la certeza de que la confianza puesta en su diseño está bien fundamentada?

En verdad, las respuestas a estas preguntas tendrían un valor. El diseño da como resultado varias alternativas de arquitectura, cada una de las cuales se evalúa para determinar cuál es la más apropiada para el problema por resolver. En las secciones que siguen se presentan dos enfoques diferentes para la evaluación de diseños arquitectónicos alternativos. El primero utiliza un método iterativo para evaluar negociaciones en el diseño. El segundo aplica una técnica seudocuantitativa para evaluar la calidad del diseño.

---

5 Las herramientas mencionadas aquí no son obligatorias; sólo son una muestra en esta categoría. En la mayoría de casos, sus nombres son marcas registradas por sus desarrolladores respectivos.

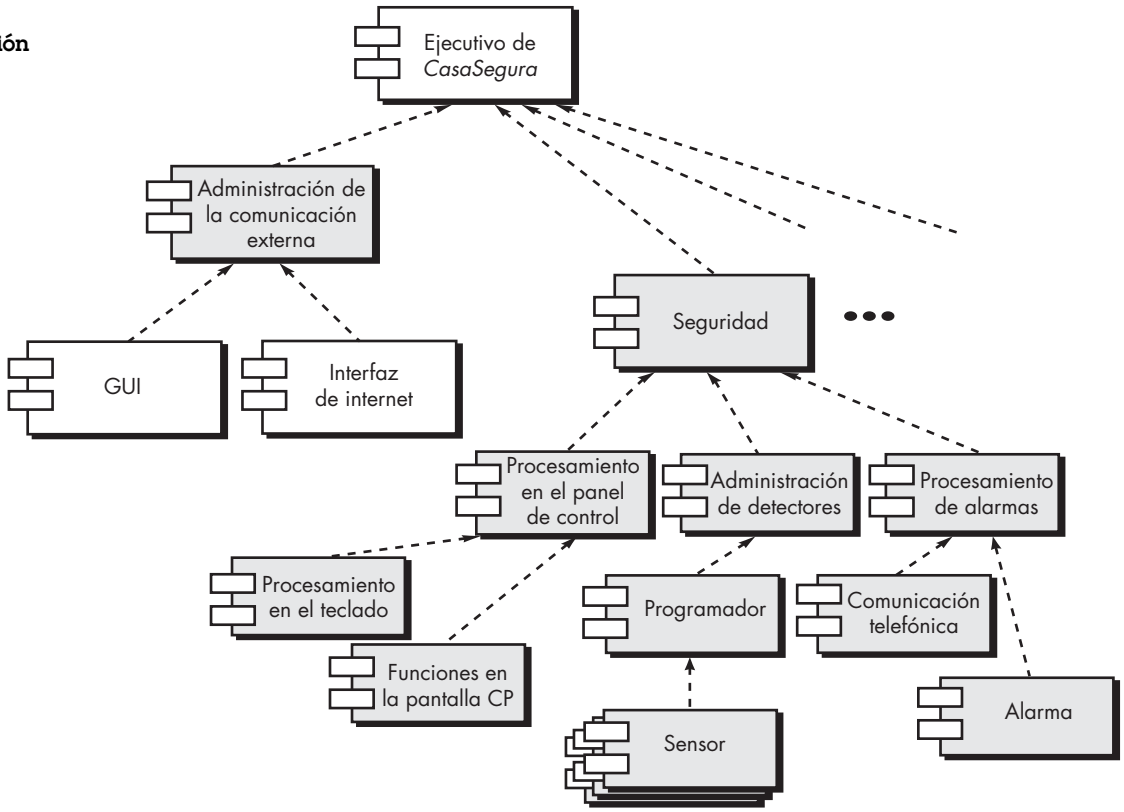


FIGURA 9.9
Instancias de la función de seguridad con la elaboración de componentes

9.5.1 Método de la negociación para analizar la arquitectura

WebRef
En la dirección **www.sei.cmu.edu/activities/architecture/ata_method.html**, puede obtenerse mucha información sobre el ATAM.

El Instituto de Ingeniería de Software (SEI, por sus siglas en inglés) desarrolló el *método de la negociación para analizar la arquitectura* (Architecture trade-off analysis method -ATAM) [Kaz98], que establece un proceso de evaluación iterativo para arquitecturas de software. Las actividades de análisis del diseño que se mencionan a continuación se llevan a cabo en forma iterativa:

1. *Escenarios de investigación.* Se desarrolla un conjunto de casos de uso (véanse los capítulos 5 y 6) para representar al sistema desde el punto de vista del usuario.
2. *Obtención de los requerimientos y restricciones, y descripción del ambiente*. Esta información se determina como parte de la ingeniería de requerimientos y se utiliza para estar seguros de que se han detectado todas las preocupaciones de los participantes.
3. *Descripción de los estilos o patrones de arquitectura elegidos para abordar los escenarios y requerimientos.* Debe describirse el estilo arquitectónico con el empleo de las siguientes perspectivas arquitectónicas:
   - *Perspectiva modular* para el análisis de las asignaciones de trabajo con componentes y grado en el que se logra el ocultamiento de información.
   - *Perspectiva del proceso* para el análisis del desempeño del sistema.
   - *Perspectiva del flujo de datos* para analizar el grado en el que la arquitectura satisface los requerimientos funcionales.
4. *Evaluación de los atributos de calidad, considerando cada atributo por separado.* El número de atributos de la calidad elegidos para el análisis es una función del tiempo disponible para la revisión y el grado en el que los atributos de calidad son relevantes para el sistema en cuestión. Los atributos de calidad para evaluar el diseño arquitectónico

incluyen confiabilidad, desempeño, seguridad, facilidad de mantenimiento, flexibilidad, facilidad de hacer pruebas, portabilidad, reutilización e interactuación.

5. *Identificación de la sensibilidad de los atributos de calidad de varios atributos arquitectónicos para un estilo de arquitectura específico.* Eso se lleva a cabo haciendo cambios pequeños en la arquitectura a fin de determinar la sensibilidad que tiene un atributo de calidad, por ejemplo, el desempeño ante el cambio. Cualesquiera atributos que se vean afectados en forma significativa por la variación de la arquitectura se denominan *puntos sensibles*.
6. *Crítica de las arquitecturas candidatas (desarrollado en el paso 3) con el uso del análisis de sensibilidad realizado en el paso 5.* El SEI describe este enfoque de la manera siguiente [Kaz98]:

> Una vez determinados los puntos sensibles de la arquitectura, la detección de puntos de negociación es simplemente la identificación de los elementos de la arquitectura a los que atributos múltiples son sensibles. Por ejemplo, el desempeño de una arquitectura cliente-servidor podría ser muy sensible al número de servidores (aumenta el desempeño, en cierto rango, con el incremento del número de servidores) [...] Entonces, el número de servidores es un punto de negociación con respecto de esta arquitectura.

Estos seis pasos representan la primera iteración ATAM. Con base en los resultados de los pasos 5 y 6, se eliminan algunas arquitecturas alternativas o se modifican una o varias de las restantes a fin de representarlas con más detalle para después volver a aplicar el ATAM.[6]

CasaSegura

Evaluación de la arquitectura

La escena: Oficina de Doug Miller, cuando avanza la modelación del diseño arquitectónico.

Participantes: Vinod, Jamie y Ed, miembros del equipo de ingeniería de software de *CasaSegura*, y Doug Miller, gerente del grupo de ingeniería de software.

La conversación:

Doug: Sé que están desarrollando un par de diferentes arquitecturas para el producto *CasaSegura*, y eso es bueno. Mi pregunta es, ¿cómo vamos a elegir la mejor?

Ed: Estoy trabajando en un estilo de llamada y regreso, luego alguno de los dos, Jamie o yo, derivará a una arquitectura OO.

Doug: Muy bien. ¿Y cómo podemos elegir?

Jamie: En mi último año de estudios de ciencias de la computación, tomé un curso de diseño y recuerdo que había varios modos de hacerlo.

Vinod: Los hay, pero son algo académicos. Miren, pienso que podemos evaluarlos y escoger el correcto con el uso de casos y escenarios.

Doug: ¿No es lo mismo?

Vinod: No si hablas de evaluar la arquitectura. Ya tenemos un conjunto completo de casos de uso. Así que los aplicaremos a las dos arquitecturas y veremos cómo reacciona cada sistema y cómo funcionan los componentes y conectores en el contexto del caso de uso.

Ed: Buena idea. Nos aseguramos de no dejar nada fuera.

Vinod: Es cierto, pero también nos dice si el diseño arquitectónico tiene rizos o si el sistema tiene que volver sobre sí mismo en un lazo para hacer el trabajo.

Jamie: Los escenarios no sólo son otro nombre de los casos de uso.

Vinod: No, en este caso, un escenario implica algo diferente.

Doug: Hablas de un escenario de calidad o de un escenario de cambio, ¿verdad?

Vinod: Sí. Lo que hacemos es regresar con los participantes y preguntarles cómo les gustaría que *CasaSegura* cambiara, digamos, en los próximos tres años. Ya sabes, nuevas versiones, características, esa clase de cosas. Construimos un conjunto de escenarios de cambio. También desarrollamos un conjunto de escenarios de calidad que defina los atributos que nos gustaría ver en la arquitectura del software.

Jamie: Y los aplicamos a las alternativas.

Vinod: Exacto. Elegiremos el estilo que mejor maneje los casos de uso y los escenarios.

6 El *Método de Análisis de la Arquitectura del Software* (MAAS) es una alternativa al ATAM y es de mucha utilidad para los lectores interesados en el análisis arquitectónico. Puede descargarse un artículo sobre el MAAS de la dirección www.sei.cmu.edu/publications/articles/saam-metho-propert-sas.html

9.5.2 Complejidad arquitectónica

Una técnica útil para evaluar la complejidad general de una arquitectura propuesta es considerar las dependencias entre los componentes dentro de la arquitectura. Estas dependencias están motivadas por el flujo de la información o por el control dentro del sistema. Zhao [Zha98] sugiere tres tipos de dependencias:

> Las *dependencias compartidas* representan relaciones entre consumidores que usan los mismos recursos o productores que producen para los mismos consumidores. Por ejemplo, para dos componentes **u** y **v**, si **u** y **v** se refieren a los mismos datos globales, entonces existe una relación de dependencia compartida entre **u** y **v**.
>
> Las *dependencias de flujo* representan relaciones de dependencia entre productores y consumidores de recursos. Por ejemplo, para dos componentes **u** y **v**, si **u** debe completarse para que el control pase a **v** (prerrequisito), o si **u** se comunica con **v** por medio de parámetros, entonces existe una relación de dependencia de flujo entre **u** y **v**.
>
> Las *dependencias de restricción* representan restricciones en el flujo relativo del control entre un conjunto de actividades. Por ejemplo, si dos componentes **u** y **v** no pueden ejecutarse al mismo tiempo (son mutuamente excluyentes), entonces existe una relación de dependencia de restricción entre **u** y **v**.

Las dependencias compartidas y el flujo propuestos por Zhao son similares al concepto de acoplamiento estudiado en el capítulo 8. El acoplamiento es un importante concepto de diseño aplicable en el nivel arquitectónico y de componente. En el capítulo 23 se estudian criterios de medida sencillos para evaluar el acoplamiento.

9.5.3 Lenguajes de descripción arquitectónica

El arquitecto de una casa tiene un conjunto de herramientas y notación estandarizadas que permiten que el diseño se represente sin ambigüedades y que sea comprensible. Aunque el arquitecto de software dispone de la notación UML, para un enfoque más formal de la especificación del diseño arquitectónico se necesitan otras formas de diagramas y algunas herramientas relacionadas.

El *lenguaje de descripción arquitectónica* (LDA) provee la semántica y sintaxis para describir una arquitectura de software. Hofmann *et al*. [Hof01] sugieren que un LDA debe brindar al diseñador la capacidad de desintegrar los componentes arquitectónicos, integrar componentes in-

HERRAMIENTAS DE SOFTWARE

Lenguajes de descripción arquitectónica

El resumen siguiente de varios LDA importantes fue preparado por Richard Land (Lan02) y se publica con el permiso de su autor. Debe observarse que los primeros cinco LDA fueron desarrollados con fines de investigación y no son productos comerciales.

Rapide (http://poset.stanford.edu/rapide/) construye a partir del concepto de conjuntos parcialmente ordenados, con lo que genera estructuras de programación muy nuevas (pero aparentemente poderosas).

UniCon (www.cs.cmu.edu/~UniCon) es "un lenguaje de descripción arquitectónica que busca ayudar a los diseñadores a definir arquitecturas de software en términos de abstracciones que les parezcan útiles".

Aesop (www.cs.cmu.edu/~able/aesop/) aborda el problema de la reutilización del estilo. Con este lenguaje es posible definir estilos y usarlos cuando se construye un sistema real.

Wright (www.cs.cmu.edu/~able/wright/) es un lenguaje formal que incluye los elementos siguientes: *componentes* con *puertos*, *conectores* con *roles* y *pegamento* para unir roles con puertos. Los estilos arquitectónicos se formalizan en el lenguaje con predicados, lo que permite revisiones estáticas para determinar la consistencia y completitud de una arquitectura.

Acme (www.cs.cmu.edu/~acme/) puede considerarse como un LDA de segunda generación, ya que su objetivo es identificar una clase de mínimo común denominador de los LDA.

UML (www.uml.org/) incluye muchos de los artefactos necesarios para hacer descripciones arquitectónicas: procesos, nodos, perspectivas, etc. UML es apropiado para hacer descripciones informales tan sólo porque se trata de un estándar ampliamente comprendido. Sin embargo, carece de toda la fortaleza que se necesita para hacer una descripción arquitectónica adecuada.

dividuales en bloques arquitectónicos más grandes y representar las interfaces (mecanismos de conexión) que hay entre los componentes. Una vez establecidas las técnicas descriptivas basadas en lenguaje para el diseño de la arquitectura, es más probable que, a medida que el diseño evoluciona, se obtengan métodos de evaluación eficaces para las arquitecturas.

9.6 Mapeo de la arquitectura con el uso del flujo de datos

Los estilos arquitectónicos analizados en la sección 9.3.1 representan arquitecturas radicalmente distintas. Por ello, no sorprende que no exista un mapeo exhaustivo que efectúe la transición del modelo de requerimientos a varios estilos de arquitectura. En realidad, no hay un mapeo práctico para ciertos estilos y el diseñador debe enfocar la traducción de los requerimientos a su diseño con el empleo de las técnicas descritas en la sección 9.4.

Para ilustrar un enfoque al mapeo arquitectónico, considere la arquitectura denominada de llamada y regreso, estructura muy común para muchos tipos de sistemas. La arquitectura de llamada y regreso reside dentro de otras más sofisticadas que ya se analizaron en este capítulo. Por ejemplo, la arquitectura de uno o más componentes de una arquitectura cliente-servidor podría denominarse de llamada y regreso.

Una técnica de mapeo llamada *diseño estructurado* [You79] se caracteriza con frecuencia como método de diseño orientado al flujo porque provee una transición conveniente de un diagrama de flujo de datos (véase el capítulo 7) a la arquitectura del software.[7] La transición del flujo de la información (representada con el diagrama de flujo de datos o DFD) a estructura de programa se consigue como parte de un proceso de seis pasos: 1) se establece el tipo de flujo de información, 2) se indican las fronteras del flujo, 3) se mapea el DFD en la estructura del programa, 4) se define la jerarquía del control, 5) se refina la estructura resultante con el empleo de criterios de medición para el diseño y heurísticos, y 6) se mejora y elabora la descripción arquitectónica.

Como ejemplo breve del mapeo de flujo de datos, se presenta uno de "transformación" paso a paso para una pequeña parte de la función de seguridad *CasaSegura*.[8] Con objeto de realizar el mapeo, debe determinarse el tipo de flujo de la información. Un tipo de flujo de información se llama *flujo de transformación* si presenta una cualidad lineal. Los datos fluyen al sistema con una *trayectoria de flujo de entrada* en la que se transforman de una representación del mundo exterior a una forma internalizada. Una vez internalizados, se procesan en un *centro de transformación*. Por último, salen del sistema por una *trayectoria de flujo de salida* que transforma los datos a una forma del mundo exterior.[9]

9.6.1 Mapeo de transformación

El mapeo de transformación es un conjunto de pasos de diseño que permite mapear un DFD con características de flujo de transformación en un estilo arquitectónico específico. Para ilustrar este enfoque, de nuevo consideraremos la función de seguridad de *CasaSegura*.[10] Un elemento del modelo de análisis es un conjunto de diagramas de flujo de datos que describen el flujo de información dentro de la función de seguridad. Para mapear estos diagramas de flujo de datos en una arquitectura de software deben darse los siguientes pasos de diseño:

---

7 Debe observarse que durante el método de mapeo también se utilizan otros elementos del modelo de requerimientos.

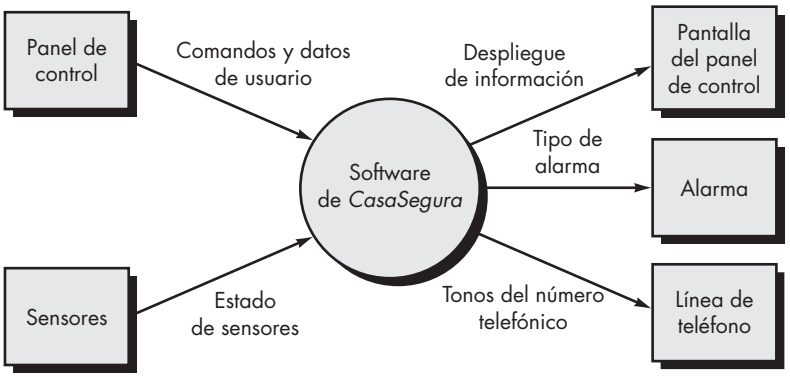
8 En el sitio web del libro se presenta un análisis más detallado del diseño estructurado.

9 En este ejemplo no se considera otro tipo importante de flujo de información, pero se aborda en el ejemplo de diseño estructurado que se presenta en el sitio web del libro.

10 Sólo se considera la parte de la función de seguridad de *CasaSegura* que utiliza el panel de control. Aquí no se incluyen otras características analizadas en el libro.

FIGURA 9.10

Diagrama de flujo de datos para la función de seguridad de *CasaSegura*



Paso 1. Revisión del modelo del sistema fundamental. El modelo del sistema fundamental o diagrama de contexto ilustra la función de seguridad como una transformación única, y representa a los productores y consumidores externos de los datos que fluyen hacia dentro y fuera de la función. La figura 9.10 ilustra un modelo de contexto de nivel 0, y la figura 9.11 muestra el flujo de datos refinado para la función de seguridad.

CONSEJO

Si en este momento se mejora más el diagrama de flujo de datos, trate de obtener burbujas que presenten mucha cohesión.

Paso 2. Revisar y mejorar los diagramas de flujos de datos para el software. La información obtenida del modelo de requerimientos se refina para producir más detalles. Por ejemplo, se estudia el DFD de nivel 2 para *vigilar sensores* (véase la figura 9.12) y se obtiene el diagrama de flujo de datos de nivel 3 que se presenta en la figura 9.13. En el nivel 3, cada transformación en el diagrama de flujo de datos presenta una cohesión relativamente grande (consulte el capítulo 8). Es decir, el proceso implícito por una transformación realiza una sola y distintiva función que se implementa como componente en el software de *CasaSegura*. Entonces, el DFD de la figura 9.13 contiene detalles suficientes para hacer un "primer corte" en el diseño de la arquitectura del subsistema *vigilar sensores*, y se continúa con más refinamiento.

FIGURA 9.11

Diagrama de flujo de datos de nivel 1 para la función de seguridad de *CasaSegura*

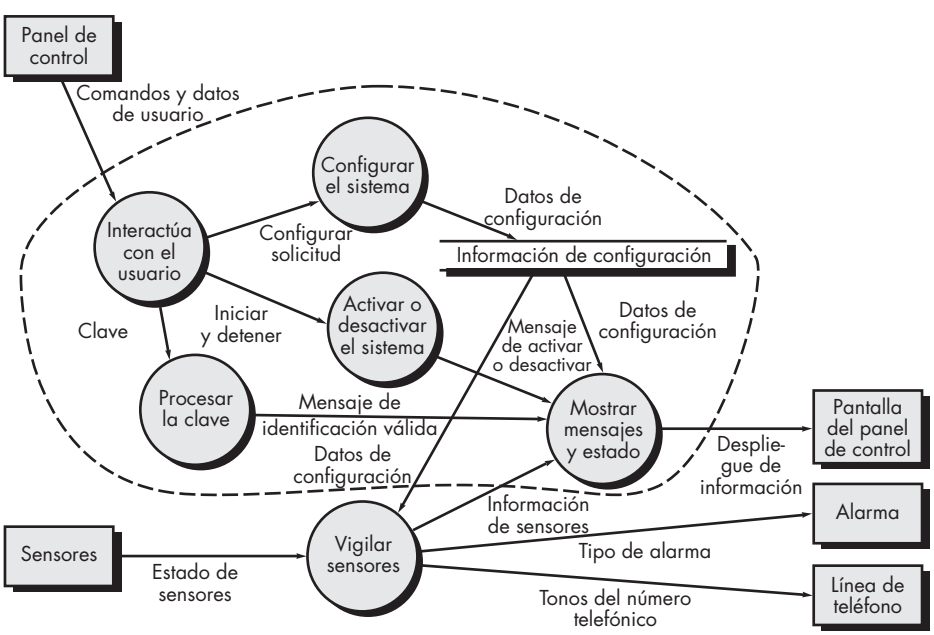
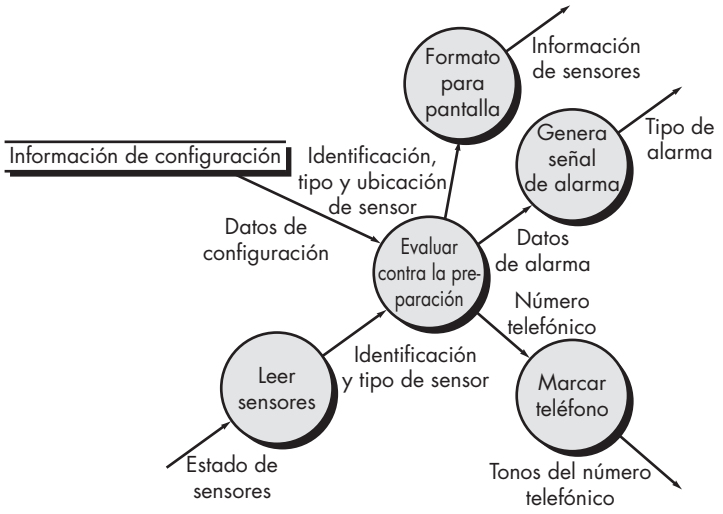


FIGURA 9.12

Diagrama de flujo de datos de nivel 2 que mejora la transformación *vigilar sensores*



PUNTO CLAVE

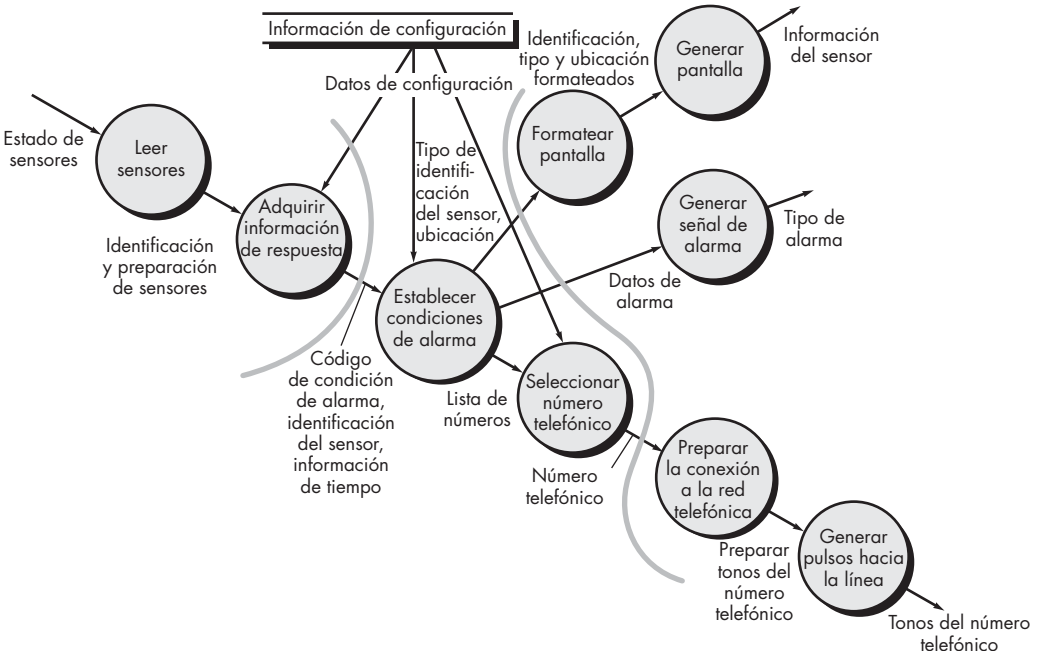
Será frecuente encontrar varios tipos de flujo de datos dentro del mismo modelo orientado al flujo. Los flujos se dividen y la estructura del programa se obtiene con el uso del mapeo apropiado.

Paso 3. Determinar si el DFD tiene características de flujo de transformación o de transacción.[11] Al evaluar el DFD (véase la figura 9.13) se observa que los datos entran al software por una trayectoria de ingreso y lo abandonan por tres trayectorias de salida. Por tanto, se adoptará una característica general de transformación para el flujo de la información.

Paso 4. Aísle el centro de transformación, especificando las fronteras de entrada y salida del flujo. Los datos de entrada fluyen por una trayectoria en la que la información pasa de su forma externa a su forma interna; el flujo de salida convierte los datos internalizados a su forma externa. Las fronteras de los flujos de entrada y salida quedan abiertas a la interpretación.

FIGURA 9.13

Diagrama de flujo de datos de nivel 3 para *vigilar sensores* con fronteras del flujo



11 En el flujo de transacción, un solo concepto de datos, llamado *transacción*, ocasiona que el flujo de datos se ramifique a través de cierto número de trayectorias definidas por la naturaleza de la transacción.

En un esfuerzo por explorar estructuras alternativas para el programa, varíe las fronteras del flujo. Esto toma poco tiempo y da una perspectiva importante.

Es decir, diferentes diseñadores tal vez seleccionen como ubicación de la frontera diferentes puntos en el flujo. De hecho, es posible obtener soluciones alternativas al diseño si se varía la colocación de las fronteras del flujo. Aunque debe tenerse cuidado al seleccionar las fronteras, la variación de una burbuja a lo largo de la trayectoria del flujo tendrá por lo general poco efecto en la estructura final del programa.

Para el ejemplo, en la figura 9.13 se ilustran las fronteras del flujo como curvas sombreadas que corren de arriba abajo a través del flujo. Las transformaciones (burbujas) que constituyen el centro de transformación quedan dentro de esas dos fronteras sombreadas. Es posible dar argumentos para reajustar una frontera (por ejemplo, podría proponerse una frontera para el flujo de entrada que separara *leer sensores* de *adquirir información de respuesta*). En este diseño, el énfasis en esta etapa de diseño debe estar en la selección de fronteras razonables y no en la iteración extensa en la colocación de las divisiones.

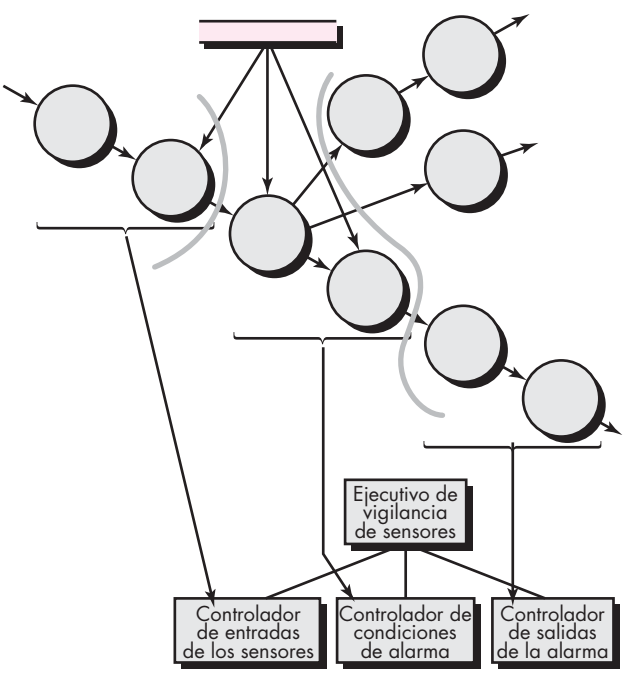
Paso 5. Realizar el "rediseño de primer nivel". La arquitectura del programa obtenida con este mapeo da como resultado una distribución del control de arriba abajo. El *rediseño* lleva a una estructura de programa en la que los componentes de alto nivel ejecutan la toma de decisiones y los de bajo nivel realizan la mayor parte del trabajo de entrada, computación y salida. Los componentes de nivel medio llevan a cabo cierto control y moderan las cantidades de trabajo.

Cuando se encuentra una transformación, se mapea un diagrama de flujo de datos para una estructura específica (una de llamar y regresar) que provea control para el procesamiento de información de entrada, transformación y salida. En la figura 9.14 se ilustra este rediseño de primer nivel para el subsistema *vigilar sensores*. Un controlador principal (llamado *ejecutivo de vigilancia de sensores*) reside en la parte superior de la estructura del programa y coordina las siguientes funciones de control subordinadas:

- Un controlador del procesamiento de la información de entrada, llamado *controlador de entradas de los sensores*, coordina la recepción de todos los datos que llegan.
- Un controlador del flujo de transformación, llamado *controlador de condiciones de la alarma*, supervisa todas las operaciones sobre los datos en forma internalizada (por ejemplo, un módulo que invoque varios procedimientos de transformación de datos.

FIGURA 9.14

Rediseño de primer nivel para *vigilar sensores*



CONSEJO

En esta etapa no se debe ser dogmático. Tal vez sea necesario establecer dos o más controladores para el procesamiento de las entradas o la computación, con base en la complejidad del sistema que se va a elaborar. Si el sentido común sugiere este enfoque, ¡adelante!

- Un controlador de procesamiento de información de salida, llamado *controlador de salidas de la alarma*, coordina la producción de información de salida.

Aunque la figura 9.14 sugiere una estructura de tres dientes, los flujos complejos que hay en sistemas grandes tal vez requieran dos o más módulos de control para cada una de las funciones de control generales descritas con anterioridad. El número de módulos en el primer nivel debe limitarse al mínimo necesario para ejecutar las funciones de control y mantener buenas características de independencia de funciones.

Paso 6. Realizar "rediseño de segundo nivel". El rediseño de segundo nivel se logra con el mapeo de transformaciones individuales (burbujas) de un diagrama de flujo de datos en módulos apropiados dentro de la arquitectura. Se comienza en la frontera del centro de transformación y se avanza hacia afuera a lo largo de las trayectorias de entrada y salida; las transformaciones se mapean en niveles subordinados de la estructura del software. En la figura 9.15 se presenta el enfoque general del rediseño de segundo nivel.

CONSEJO

Hay que eliminar los módulos de control redundantes. Es decir, si un módulo de control no hace nada más que controlar a otro módulo, su función controladora debe llevarse a un módulo de nivel más alto.

Aunque la figura 9.15 ilustra un mapeo uno a uno entre las transformaciones del diagrama de flujo y los módulos de software, es frecuente que haya distintos mapeos. Es posible combinar y representar dos o incluso tres burbujas como un solo componente, o una sola burbuja puede expandirse a dos o más componentes. Son consideraciones prácticas y mediciones de calidad del diseño las que dictan el resultado del rediseño de segundo nivel. La revisión y refinamiento tal vez produzcan cambios en esta estructura, pero sirven como diseño de "primera iteración".

El rediseño de segundo nivel para el flujo de entrada se presenta de la misma manera. El rediseño se logra de nuevo avanzando hacia afuera a partir de la frontera del centro de trans-

FIGURA 9.15

Rediseño de segundo nivel para *vigilar sensores*

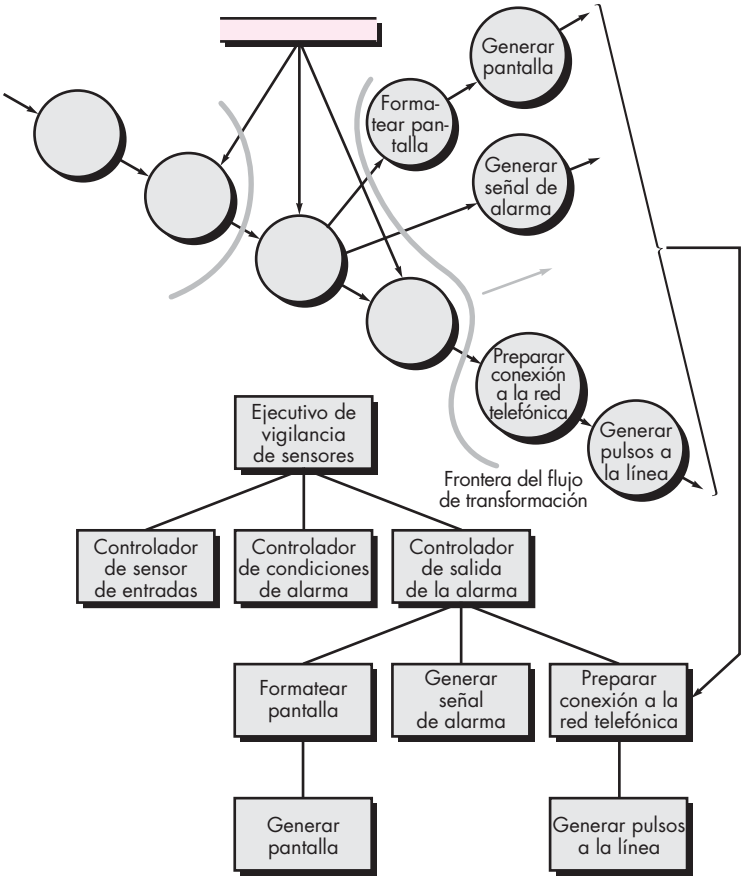
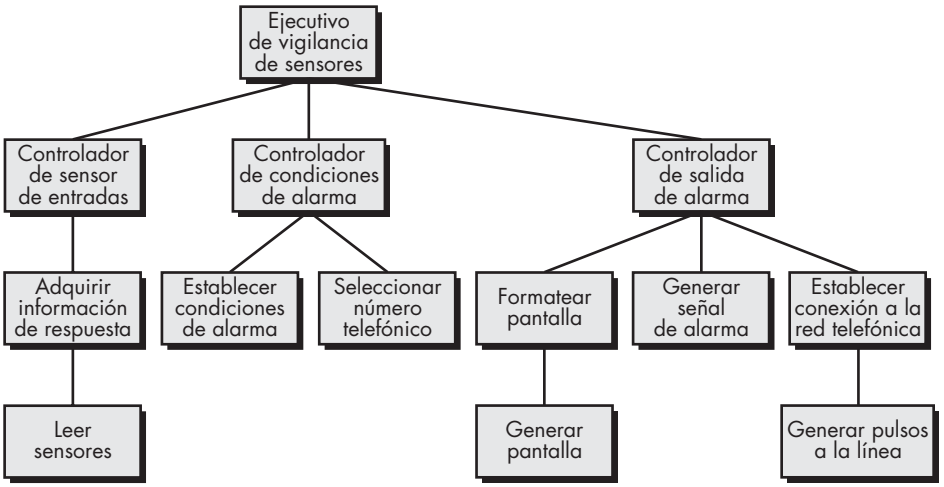


FIGURA 9.16
Estructura de primera iteración para *vigilar sensores*



CONSEJO

Conserve módulos "trabajadores" en un nivel bajo de la estructura del programa. Esto llevará a una arquitectura fácil de mantener.

formación en el lado del flujo de entrada. El centro de transformación del software del subsistema *vigilar sensores* se mapea de modo un poco distinto. Cada conversión de datos o transformación de cálculo de la parte de transformación del diagrama de flujo de datos se mapea en un módulo subordinado al controlador de la transformación. En la figura 9.16 se presenta la arquitectura completa de primera iteración.

Los componentes así mapeados que se aprecian en la figura 9.16 representan un diseño inicial de la arquitectura del software. Aunque los componentes reciben su nombre de manera que se implique la función, para cada uno debe escribirse una narración breve (adaptada de la especificación del proceso desarrollada para la transformación de datos y que se generó durante el modelado de los requerimientos). La narración describe la interfaz del componente, las estructuras internas de los datos, un relato de las funciones y el análisis breve de las restricciones y otros rasgos especiales (como los archivos de entrada y salida, características que dependen del hardware, requerimientos especiales de tiempo, etcétera).

Cita:

"Hacerlo tan sencillo como sea posible. Pero no más."

Albert Einstein

Paso 7. Refinar la arquitectura de primera iteración con el empleo de heurísticos de diseño para mejorar la calidad del software. Siempre es posible refinar la arquitectura de primera iteración, aplicando conceptos de independencia de funciones (véase el capítulo 8). Los componentes hacen explosión o implosión para producir un rediseño sensible, la separación de problemas, buena cohesión, acoplamiento mínimo y, lo más importante, una estructura que puede implementarse sin dificultad, probar sin confusión y mantener sin grandes problemas.

Los refinamientos son impuestos por el análisis y los métodos de evaluación descritos en la sección 9.5, así como por consideraciones prácticas y el sentido común. Por ejemplo, hay ocasiones en las que el controlador para el flujo de datos de entrada es totalmente innecesario, cuando se requiere algún procesamiento de las entradas en un componente subordinado al controlador de la transformación, cuando no es posible evitar mucho acoplamiento debido a datos globales o cuando no se logran características estructurales óptimas. El arbitraje final lo constituyen los requerimientos del software acoplados con el criterio humano.

El objetivo de los siete pasos anteriores es desarrollar una representación arquitectónica del software. Es decir, una vez definida la estructura, se evalúa y mejora considerándola como un todo. Las modificaciones que se hacen en este momento exigen poco trabajo adicional, pero tienen un efecto profundo en la calidad del software.

Debe hacerse una pausa y considerar la diferencia entre el enfoque de diseño descrito y el proceso de "escribir programas". Si el código es la única representación del software, usted y

CasaSegura

Refinación de la arquitectura de primer corte

La escena: El cubículo de Jamie, cuando comienza la modelación del diseño.

Participantes: Jamie y Ed, miembros del equipo de ingeniería de software de *CasaSegura*.

La conversación:

[Ed acaba de terminar el diseño de primer corte del subsistema de vigilancia de sensores. Se detiene para solicitar la opinión de Jamie.]

Ed: Pues bien, aquí está la arquitectura que obtuve.

[Ed muestra a Jamie la figura 9.16, y ella la estudia unos momentos.]

Jamie: Está muy bien, pero creo que podemos hacer algo para que sea más sencilla… y mejor.

Ed: ¿Como qué?

Jamie: Bueno, ¿por qué usaste el componente *controlador de sensores de entrada*?

Ed: Porque se necesita un controlador para el mapeo.

Jamie: No en realidad. El controlador no hace gran cosa, ya que estamos manejando una sola trayectoria de flujo para los datos de entrada. Puede eliminarse el controlador sin que pase nada.

Ed: Puedo vivir con eso. Lo cambiaré y…

Jamie (sonríe): Espera… También podemos hacer la implosión de los componentes *establecer condiciones de alarma* y *seleccionar número telefónico*. El controlador de la transformación que presentas en realidad no es necesario y la poca disminución en la cohesión es tolerable.

Ed: Simplificación, ¿eh?

Jamie: Sí. Y al hacer refinamientos sería buena idea hacer la implosión de los componentes *formatear pantalla* y *generar pantalla*. El formateo de la pantalla para el panel de control es algo sencillo. Puede definirse un nuevo módulo llamado *producir pantalla*.

Ed (dibuja): Entonces, ¿esto es lo que piensas que debemos hacer?

[Muestra a Jamie la figura 9.17.]

Jamie: Es un buen comienzo.

sus colegas tendrán grandes dificultades para evaluarlo o mejorarlo en un nivel global u holístico y, en verdad, tendrán dificultades porque "los árboles no los dejarán ver el bosque".

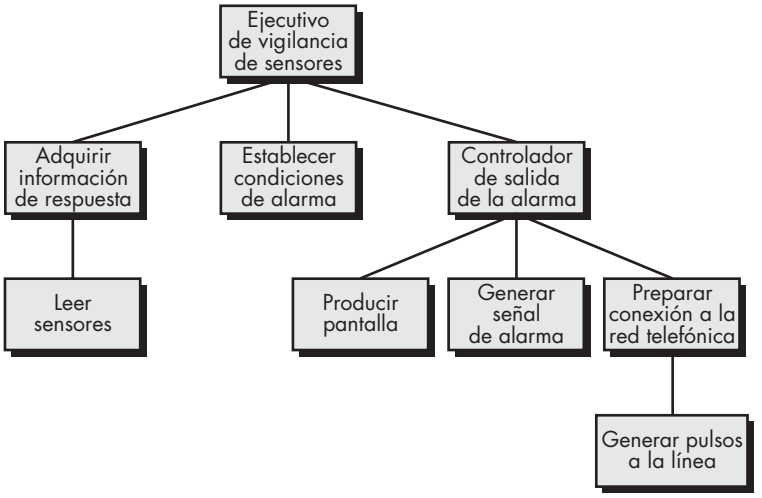
9.6.2 Refinamiento del diseño arquitectónico

? ¿Qué pasa después de que se generó la arquitectura?

Cualquier análisis del refinamiento del diseño debería ir precedido de este comentario: "Recuerde que un 'diseño óptimo' que no funcione tiene un mérito cuestionable." Debe ocuparse de desarrollar una representación del software que satisfaga todos los requerimientos funcionales y de desempeño, y darle mérito según sus mediciones y heurísticos de diseño.

Debe estimularse el refinamiento de la arquitectura del software durante las primeras etapas del diseño. Como ya se dijo en este capítulo, hay estilos alternativos para la arquitectura que es posible obtener, refinar y evaluar en busca del "mejor" enfoque. Éste, dirigido a la optimización,

Figura 9.17
Estructura refinada del programa para *vigilar sensores*



es uno de los verdaderos beneficios que se logran con el desarrollo de una representación de la arquitectura del software.

Es importante observar que la sencillez estructural con frecuencia refleja tanto elegancia como eficiencia. El refinamiento del diseño debe perseguir el menor número de componentes que sea consistente con la modularidad efectiva y la estructura de datos menos compleja que cumpla de modo adecuado con los requerimientos de información.

9.7 Resumen

La arquitectura del software proporciona una visión holística del sistema que se va a construir. Ilustra la estructura y organización de los componentes del software, sus propiedades y conexiones. Los componentes del software incluyen módulos de programa y las distintas representaciones de datos que manipula éste. Por tanto, el diseño de los datos es parte integral de la generación de la arquitectura del software. Ésta subraya las primeras decisiones respecto del diseño y provee un mecanismo para considerar los beneficios de las estructuras alternativas para el sistema.

Dentro de un género arquitectónico dado, hay varios estilos y patrones diferentes disponibles para el ingeniero de software. Cada estilo describe una categoría de sistemas que agrupa un conjunto de componentes que realizan una función requerida por el sistema; un grupo de conectores que permiten comunicación, coordinación y cooperación entre los componentes; restricciones que definen cómo pueden integrarse éstos para formar el sistema y modelos semánticos que permiten que un diseñador entienda las propiedades generales del sistema.

En un sentido general, el diseño arquitectónico se obtiene con el empleo de cuatro pasos. En primer lugar, el sistema debe representarse en contexto. Es decir, el diseñador debe definir las entidades externas con las que interactúa el software y la naturaleza de la interacción. Una vez especificado el contexto, el diseñador debe identificar un conjunto de abstracciones de alto nivel, llamadas *arquetipos*, que representan elementos fundamentales del comportamiento o función del sistema. Ya que se definieron las abstracciones, el diseño comienza a avanzar cerca del dominio de la implementación. Se identifican los componentes y se representan dentro del contexto de una arquitectura que les da apoyo. Por último, se desarrollan instancias específicas de la arquitectura para "probar" el diseño en el contexto del mundo real.

Como ejemplo sencillo del diseño arquitectónico, el método del mapeo presentado en este capítulo usa las características del flujo de datos para obtener un estilo arquitectónico de uso muy común. El diagrama de flujo de datos se mapea en la estructura del programa con el uso del enfoque del mapeo de la transformación. Éste se aplica a un flujo de información que presente fronteras distintas entre los datos de entrada y los de salida. El diagrama de flujo de datos se mapea en una estructura que asigna el control a la entrada, al procesamiento y a la salida junto con tres jerarquías de módulos diseñados por separado. Una vez que se tiene la arquitectura, se elabora y analiza mediante criterios de calidad.

Problemas y puntos por evaluar

9.1. Con el uso de la arquitectura de una casa o edificio como metáfora, establezca comparaciones con la arquitectura del software. ¿En qué se parecen las disciplinas de la arquitectura clásica y la del software? ¿En qué difieren?

9.2. Diga dos o tres ejemplos de aplicaciones para cada uno de los estilos arquitectónicos mencionados en la sección 9.3.1.

9.3. Algunos de los estilos arquitectónicos citados en la sección 9.3.1 tienen naturaleza jerárquica, mientras que otros no. Elabore una lista de cada tipo. ¿Cómo se implementarían los estilos arquitectónicos que no son jerárquicos?

9.4. Los términos *estilo arquitectónico*, *patrón arquitectónico* y *marco* (que no se estudia en este libro) surgen con frecuencia en los análisis de la arquitectura del software. Investigue y describa en qué difiere cada uno de ellos de los demás.

9.5. Seleccione una aplicación con la que esté familiarizado. Responda cada una de las preguntas planteadas para el control y los datos de la sección 9.3.3.

9.6. Investige el ATAM (en [Kaz98]) y presente un análisis detallado de los seis pasos presentados en la sección 9.5.1.

9.7. Si no lo ha hecho, termine el problema 6.6. Use los métodos de diseño descritos en este capítulo para desarrollar una arquitectura del software para el SSRB.

9.8. Utilice un diagrama de flujo y una narración del procesamiento para describir un sistema basado en computadora que tenga distintas características de transformación del flujo. Defina las fronteras del sistema y mapee el diagrama de flujo de los datos en una arquitectura del software con el empleo de la técnica descrita en la sección 9.6.1.

Lecturas adicionales y fuentes de información

La bibliografía sobre arquitectura del software ha crecido mucho en la última década. Los libros escritos por Gorton (*Essential Software Architecture*, Springer, 2006), Reekie y McAdam (*A Software Architecture Primer*, Angophora Press, 2006), Albin (*The Art of Software Architecture*, Wiley, 2003) y Bass *et al.* (*Software Architecture in Practice*, 2a. ed., Addison-Wesley, 2002), presentan introducciones útiles a un área del conocimiento con muchos retos intelectuales.

Buschman *et al.* (*Pattern-Oriented Software Architecture*, Wiley, 2007) y Kuchana (*Software Architecture Design Patterns in Java*, Auerbach, 2004) estudian aspectos orientados al patrón del diseño arquitectónico. Rozanski y Woods (*Software Systems Architecture*, Addison-Wesley, 2005), Fowler (*Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003), Clements *et al.* (*Documenting Software Architecture: View and Beyond*, Addison-Wesley, 2002), Bosch [Bos00] y Hofmeister *et al.* [Hof00] hacen análisis profundos de la arquitectura del software.

Hennesey y Patterson (*Computer Architecture*, 4a. ed., Morgan-Kaufmann, 2007) adoptan un punto de vista notable, por ser cuantitativo, para los aspectos del diseño de la arquitectura del software. Clements *et al.* (*Evaluating Software Architectures*, Addison-Wesley, 2002) analizan los aspectos asociados con la evaluación de alternativas arquitectónicas y la selección de la mejor arquitectura para un dominio dado de problemas.

Los libros dedicados a la implementación sobre la arquitectura abordan el diseño arquitectónico dentro de un ambiente o tecnología específicos de desarrollo. Marks y Bell (*Sevice-Oriented Architecture*, Wiley, 2006) estudian el enfoque de diseño que relaciona los negocios y los recursos computacionales con los requerimientos definidos por los clientes. Stahl *et al*. (*Model-Driven Software Development*, Wiley, 2006) analizan la arquitectura dentro del contexto de los enfoques de modelado dirigidos al dominio. Radaideh y Al-Ameed (*Architecture of Reliable Web Applications Software*, GI Global, 2007) consideran arquitecturas apropiadas para *webapps*. Clements y Norhrop (*Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001) estudian el diseño de arquitecturas que dan apoyo a líneas de productos de software. Shanley (*Protected Mode Software Architecture*, Addison-Wesley, 1996) proporciona una guía del diseño arquitectónico para cualquier persona que diseñe sistemas basados en PC que operen en tiempo real, o para sistemas operativos de tareas múltiples o manejadores de dispositivos.

La investigación actual sobre arquitectura del software se documenta cada año en *Proceedings of the International Workshop on Software Architecture*, publicación patrocinada por ACM y otras organizaciones de cómputo, y en *Proceedings of the International Conference on Software Engineering*.

En internet se encuentra una amplia variedad de fuentes de información sobre el diseño arquitectónico. En el sitio web del libro, **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, se halla una lista actualizada de referencias que hay en la red mundial, relevantes para el diseño de la arquitectura.

CAPÍTULO

10 DISEÑO EN EL NIVEL DE COMPONENTES

CONCEPTOS CLAVE

acoplamiento 244
cohesión 243
componentes
  adaptación 258
  calificación 257
  clasificación 260
  combinación 259
  orientación
  a objetos 235
  tradicional 252
  webapp 251
desarrollo basado
en componentes 256
diseño del contenido 251
ingeniería del dominio 257
lineamientos de diseño 242
notación del diseño
tabular 254

El diseño en el nivel de componentes tiene lugar una vez terminado el diseño de la arquitectura. En esta etapa se ha establecido la estructura general de los datos y del programa del software. El objetivo es traducir el modelo del diseño a software operativo. Pero el nivel de abstracción del modelo de diseño existente es relativamente alto y el del programa operativo es bajo. La traducción es difícil y está abierta a la introducción de errores sutiles que son difíciles de detectar y de corregir en las etapas posteriores del proceso del software. En una conferencia famosa, Edsgar Dijkstra, investigador importante que ha contribuido mucho a nuestra comprensión del diseño de software, dijo [Dij72]:

> El software parece ser diferente de muchos otros productos en los que la regla es que a mejor calidad se da un mayor precio. Aquellos que desean un software en verdad confiable descubrirán que deben encontrar un medio para evitar de inicio la mayoría de los posibles errores; como resultado, el proceso de programación se hace más barato [...] los programadores eficaces no tienen que perder su tiempo depurando errores: no deben introducirlos al arrancar.

Aunque estas palabras fueron expresadas hace muchos años, siguen siendo ciertas. Al traducir el modelo del diseño a código fuente, deben seguirse principios de diseño que no sólo hagan la traducción sino que también eviten la "introducción de errores desde el principio".

Es posible representar el diseño en el nivel del componente con el uso de un lenguaje de programación. En esencia, el programa se crea con el empleo del modelo de diseño arquitectónico como guía. Un enfoque alternativo consiste en representar el diseño en el nivel de los componentes con alguna representación intermedia (gráfica, tabular o basada en texto) que se traduzca con facilidad a código fuente. Sin que importe el mecanismo utilizado para representar

UNA MIRADA RÁPIDA

¿Qué es? Durante el diseño arquitectónico, se define un conjunto completo de componentes de software. Pero las estructuras internas de datos y detalles de procesamiento de cada componente no están representadas en un nivel de abstracción cercano al código. El diseño en el nivel de componentes define las estructuras de datos, algoritmos, características de la interfaz y mecanismos de comunicación asignados a cada componente del software.

¿Quién lo hace? Un ingeniero de software es quien realiza el diseño en el nivel de componentes.

¿Por qué es importante? Antes de elaborar el software, se tiene que ser capaz de determinar si funcionará. El diseño en el nivel de componentes lo representa en forma tal que permite revisar los detalles del diseño para garantizar su corrección y su consistencia con otras representaciones del diseño (por ejemplo, los datos y el diseño de la arquitectura y la interfaz). Esto proporciona un medio para evaluar si funcionarán las estructuras de datos, interfaces y algoritmos.

¿Cuáles son los pasos? Las representaciones de diseño de datos, arquitectura e interfaces constituyen el fundamento para el diseño en el nivel de componentes. La definición de clase o narrativa de procesamiento de cada componente se traduce a un diseño detallado que utiliza formas de diagrama o basadas en texto que especifican las estructuras de datos internas, los detalles de la interfaz local y la lógica del procesamiento. La notación del diseño incluye diagramas UML y formatos complementarios. Se especifica el diseño del procedimiento con el empleo de construcciones de programación estructurada. Con frecuencia es posible obtener componentes de software reutilizable ya existentes, en lugar de construir nuevos.

¿Cuál es el producto final? El producto principal que se genera en esta etapa es el diseño de cada componente, representado con notación gráfica, tabular o basada en texto.

¿Cómo me aseguro de que lo hice bien? Se efectúa la revisión del diseño. Esto se hace para determinar durante las primeras etapas de diseño si las estructuras de datos, interfaces, secuencias de procesamiento y condiciones lógicas son correctas y si producirán los datos apropiados o la transformación del control asignado al componente.

a éste, las estructuras de datos, interfaces y algoritmos definidos deben apegarse a varios lineamientos de diseño bien establecidos que ayudan a evitar los errores conforme evoluciona el diseño del procedimiento. En este capítulo se estudian estos lineamientos y los métodos disponibles para cumplirlos.

10.1 ¿QUÉ ES UN COMPONENTE?

Cita:

"Los detalles no son detalles. Son el diseño."

Charles Eames

Un *componente* es un bloque de construcción de software de cómputo. Con más formalidad, la *Especificación OMG del Lenguaje de Modelado Unificado* [OMG03a] define un componente como "una parte modular, desplegable y sustituible de un sistema, que incluye la implantación y expone un conjunto de interfaces".

Como se dijo en el capítulo 9, los componentes forman la arquitectura del software y, en consecuencia, juegan un papel en el logro de los objetivos y de los requerimientos del sistema que se va a construir. Como los componentes se encuentran en la arquitectura del software, deben comunicarse y colaborar con otros componentes y con entidades (otros sistemas, dispositivos, personas, etc.) que existen fuera de las fronteras del software.

El verdadero significado del término *componente* difiere en función del punto de vista del ingeniero de software que lo use. En las secciones que siguen, se estudian tres visiones importantes de lo que es un componente y cómo se emplea en el desarrollo de la modelación del diseño.

10.1.1 Una visión orientada a objetos

PUNTO CLAVE

Desde un punto de vista orientado a objetos, un componente es un conjunto de clases que colaboran.

En el contexto de la ingeniería de software orientada a objetos, un componente contiene un conjunto de clases que colaboran.[1] Cada clase dentro de un componente se elabora por completo para que incluya todos los atributos y operaciones relevantes para su implantación. Como parte de la elaboración del diseño, también deben definirse todas las interfaces que permiten que las clases se comuniquen y colaboren con otras clases de diseño. Para lograr esto, se comienza con el modelo de requerimientos y se elaboran clases de análisis (para los componentes que se relacionan con el dominio del problema) y clases de infraestructura (para los componentes que dan servicios de apoyo para el dominio del problema).

Para ilustrar el proceso de la elaboración del diseño, considere el software que se va a elaborar para un taller de impresión avanzada. El objetivo general del software es obtener los requerimientos que plantea el cliente en el mostrador, presupuestar un trabajo de impresión y después pasar éste a una instalación automatizada de producción. Durante la ingeniería de requerimientos se obtuvo una clase de análisis llamada **ImprimirTrabajo**. En la parte superior de la figura 10.1 aparecen los atributos y operaciones definidos durante el análisis. En el diseño de la arquitectura se definió **ImprimirTrabajo** como un componente dentro de la arquitectura del software y está representado con la notación abreviada UML[2] que se muestra en la parte media derecha de la figura. Observe que **ImprimirTrabajo** tiene dos interfaces, *CalcularTrabajo*, que provee la capacidad de obtener el costo del trabajo, e *IniciarTrabajo*, que pasa el trabajo a través de las instalaciones de producción. Éstas se encuentran representadas con los símbolos de "paleta" que aparecen en el lado izquierdo de la caja del componente.

CONSEJO

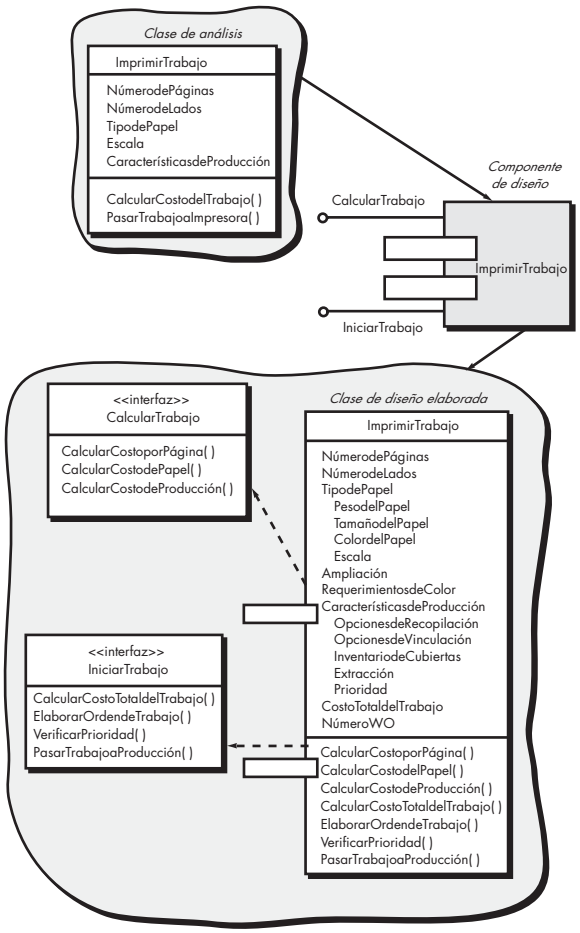
Recuerde que el modelado del análisis y del diseño son acciones iterativas. Es probable que la elaboración de la clase de análisis original requiera de etapas adicionales, que con frecuencia van seguidas de etapas de modelado del diseño que representan la clase de diseño elaborada (los detalles del componente).

El diseño en el nivel del componente comienza en este punto. Deben elaborarse los detalles del componente **ImprimirTrabajo** para que den información suficiente que guíe la implantación. La clase de análisis original se lleva a cabo para dar cuerpo a todos los atributos y operaciones requeridos para implantar la clase así como el componente **ImprimirTrabajo**. En la parte inferior derecha de la figura 10.1, la clase de diseño elaborada **ImprimirTrabajo** contiene

1 En ciertos casos, un componente contiene una sola clase.

2 Los lectores que no estén familiarizados con la notación UML deben consultar el apéndice 1.

FIGURA 10.1
Elaboración de un componente de diseño



información más detallada de los atributos, así como una descripción amplia de las operaciones que se necesitan para implantar el componente. Las interfaces *CalcularTrabajo* e *IniciarTrabajo* implican comunicación y colaboración con otros componentes (que no se muestran). Por ejemplo, la operación *CalcularCostoporPágina ( )* (que forma parte de la interfaz *CalcularTrabajo*) podría colaborar con un componente llamado **TabladeValuación** que contuviera información sobre los precios del trabajo. La operación *VerificarPrioridad ( )* (parte de la interfaz *IniciarTrabajo*) quizá colabore con un componente de nombre **FiladeTrabajos** para determinar los tipos y prioridades de trabajos que están en espera de su producción.

Esta actividad de elaboración se aplica a cada componente definido como parte del diseño de la arquitectura. Una vez concluida, se aplica más elaboración a cada atributo, operación e interfaz. Deben especificarse las estructuras de datos apropiadas para cada atributo. Además, se diseñan los detalles algorítmicos requeridos para implantar la lógica del procesamiento asociada con cada operación. Este diseño del procedimiento se analiza más adelante, en este capítulo. Por último, se diseñan los mecanismos requeridos para implantar la interfaz. Para el software orientado a objetos, esto incluye la descripción de todos los mensajes que se requieren para efectuar la comunicación dentro del sistema.

10.1.2 La visión tradicional

En el contexto de la ingeniería de software tradicional, un componente es un elemento funcional de un programa que incorpora la lógica del procesamiento, las estructuras de datos internas que

Cita:

"Invariablemente se observa que un sistema complejo que funciona ha evolucionado a partir de un sistema sencillo que funcionaba."

John Gall

se requieren para implantar la lógica del procesamiento y una interfaz que permite la invocación del componente y el paso de los datos. Dentro de la arquitectura del software se encuentra un componente tradicional, también llamado *módulo*, que tiene tres funciones importantes: 1) como *componente de control* que coordina la invocación de todos los demás componentes del dominio del problema, 2) como *componente del dominio del problema* que implanta una función completa o parcial que requiere el cliente y 3) como *componente de infraestructura* que es responsable de las funciones que dan apoyo al procesamiento requerido en el dominio del problema.

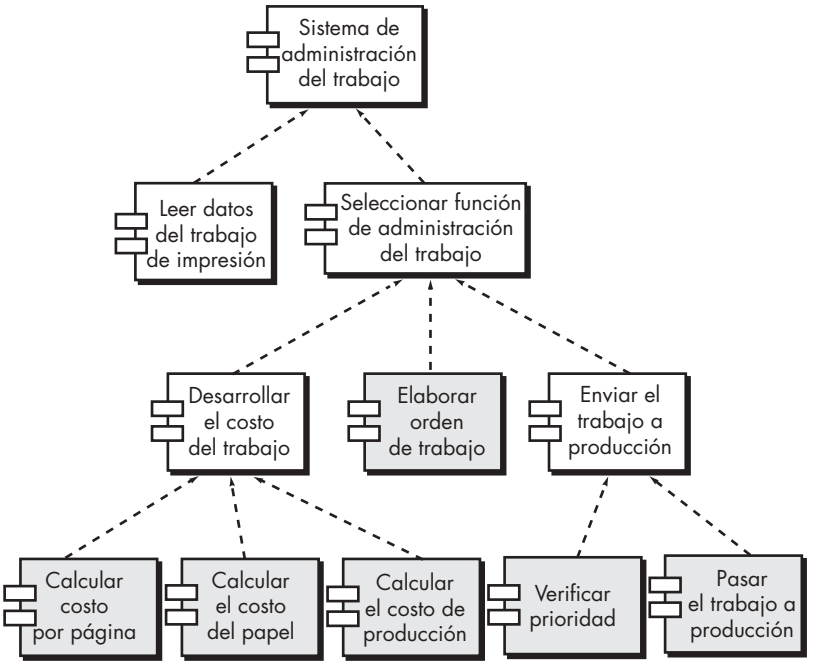
Igual que los componentes orientados a objetos, los componentes tradicionales del software provienen del modelo de análisis. Sin embargo, en este caso, el elemento de datos orientado al flujo del modelo de análisis sirve de base para su obtención. Cada transformación (burbuja) representada en los niveles más bajos del diagrama de flujo de datos se mapea (véase la sección 9.6) en una jerarquía de módulos. Cerca de la parte superior de la jerarquía (arquitectura del programa) se hallan los componentes del control (módulos) y hacia la parte inferior de ella tienden a encontrarse los del dominio del problema. Para lograr una modularidad efectiva, cuando se elaboran los componentes se aplican conceptos de diseño, como la independencia de funciones (véase el capítulo 8).

Para ilustrar este proceso de elaboración del diseño de componentes tradicionales, considere otra vez el software que debe elaborarse para un taller de impresión avanzada. Durante el modelado de los requerimientos se obtendrá un conjunto de diagramas de flujo de datos. Suponga que éstos se mapean en la arquitectura que se aprecia en la figura 10.2. Cada rectángulo representa un componente del software. Observe que los que están sombreados son equivalentes en su función y operaciones a los definidos para la clase **ImprimirTrabajo** que se analizó en la sección 10.1.1. Sin embargo, en este caso, cada operación se representa como módulo aislado que se invoca como se indica en la figura. Para controlar el procesamiento se utilizan otros módulos, por lo que son componentes de control.

Cada módulo de la figura 10.2 se elabora durante el diseño en el nivel de componentes. La interfaz del módulo se define explícitamente. Es decir, se representa todo objeto de datos o

FIGURA 10.2

Gráfica de la estructura de un sistema tradicional



Conforme se elabora el diseño para cada componente del software, la atención pasa al diseño de estructuras de datos específicas y al diseño del procedimiento para manipularlas. Sin embargo, no hay que olvidar la arquitectura que debe albergar los componentes o las estructuras de datos globales que den servicio a muchos componentes.

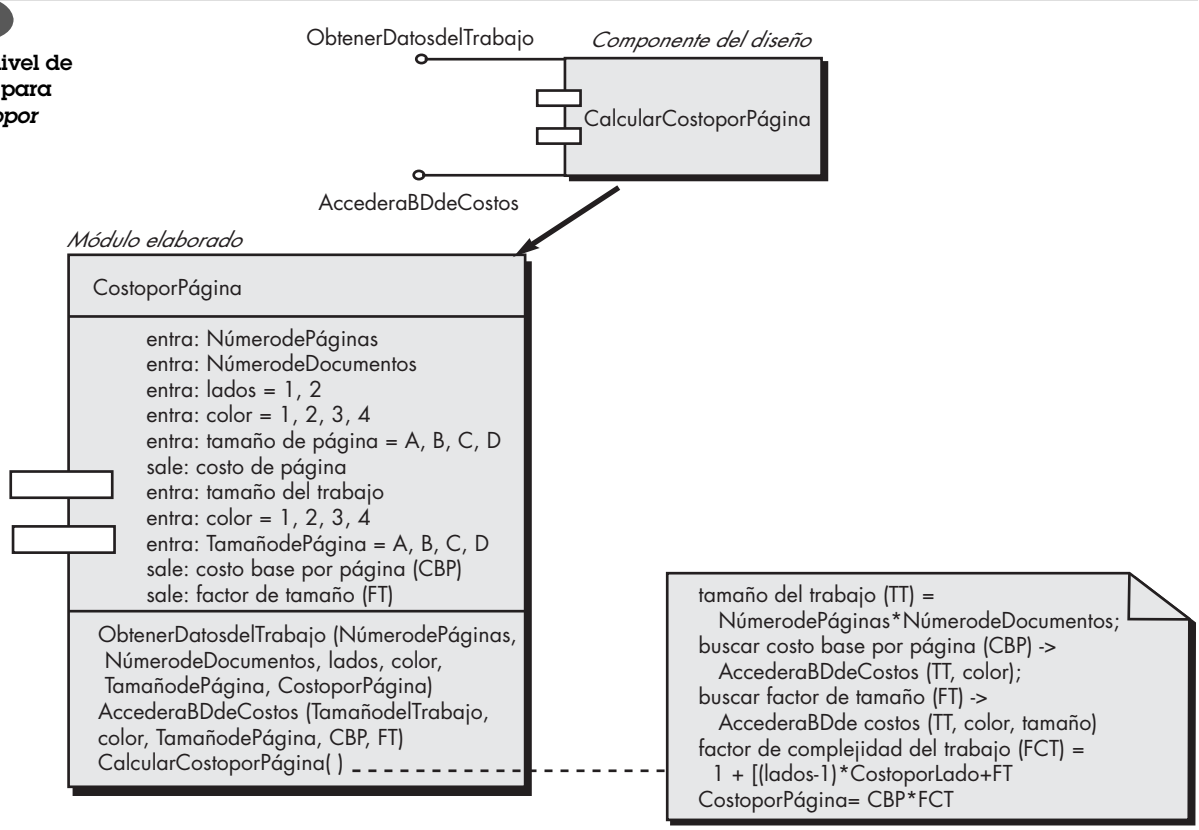
control que fluya a través de la interfaz. Se definen las estructuras de datos que se utilicen en el interior del módulo. El algoritmo que permite que el módulo cumpla su función prevista se diseña con el empleo del enfoque de refinamiento por etapas que se estudió en el capítulo 8. El comportamiento del módulo se representa en ocasiones con un diagrama de estado.

Para ilustrar este proceso, considere el módulo *CalcularCostoporPágina*. El objetivo de este módulo es calcular el costo de impresión por página con base en las especificaciones dadas por el cliente. Los datos requeridos para realizar esta función son: **número de páginas en el documento, número total de documentos que se va a producir, impresión por uno o dos lados, requerimientos de color y requerimientos de tamaño**. Estos datos se pasan a *CalcularCostoporPágina* a través de la interfaz del módulo. *CalcularCostoporPágina* usa estos datos para determinar el costo por página con base en el tamaño y complejidad del trabajo, que es función de todos los datos proporcionados al módulo a través de la interfaz. El costo por página es inversamente proporcional al tamaño del trabajo y directamente proporcional a su complejidad.

La figura 10.3 representa el diseño en el nivel de componentes con el uso de notación UML modificada. El módulo *CalcularCostoporPágina* accede a los datos invocando el módulo *ObtenerDatosdelTrabajo*, que permite que todos los datos relevantes pasen al componente, y una interfaz de base de datos, *AccederBDdeCostos*, que permite que el módulo acceda a una base de datos que contiene todos los costos de impresión. A medida que avanza el diseño, se elabora el módulo *CalcularCostoporPágina* para que provea los detalles del algoritmo y de la interfaz (véase la figura 10.3). Los detalles del algoritmo se representan con el uso del texto de seudocódigo que aparece en la figura, o con un diagrama de actividades UML. Las interfaces se representan como una colección de objetos de datos o conceptos de entrada y salida. La elaboración del diseño continúa hasta que haya detalles suficientes que guíen la construcción del componente.

FIGURA 10.3

Diseño en el nivel de componentes para *CalcularCostopor Página*



10.1.3 Visión relacionada con el proceso

La visión orientada a objetos y la tradicional del diseño en el nivel de componentes, presentadas en las secciones 10.1.1 y 10.1.2, suponen que el componente se diseña desde la nada. Es decir, que se crea un nuevo componente con base en las especificaciones obtenidas del modelo de requerimientos. Por supuesto, existe otro enfoque.

En las últimas dos décadas, la comunidad de la ingeniería de software ha puesto el énfasis en la necesidad de elaborar sistemas que utilicen componentes de software o patrones de diseño ya existentes. En esencia, a medida que avanza el trabajo de diseño se dispone de un catálogo de diseño probado o de componentes en el nivel de código. Conforme se desarrolla la arquitectura del software, se escogen del catálogo componentes o patrones de diseño y se usan para construir la arquitectura. Como estos componentes fueron construidos teniendo en mente lo reutilizable, se dispone totalmente de la descripción de su interfaz, de las funciones que realizan y de la comunicación y colaboración que requieren. En la sección 10.6 se estudian algunos aspectos importantes de la ingeniería de software basada en componentes.

INFORMACIÓN

Estándares y marcos basados en componentes

Uno de los elementos clave que conducen al éxito o fracaso de la ingeniería de estándares basados en componentes es su disponibilidad, que en ocasiones recibe el nombre de *middleware*. El *middleware* es una colección de componentes de infraestructura que permiten que los componentes del dominio del problema se comuniquen entre sí a través de una red o dentro de un sistema complejo. Los ingenieros de software que deseen usar el desarrollo basado en componentes como proceso de software pueden elegir entre los estándares siguientes:

OMG CORBA-www.corba.org/
Microsoft COM-www.microsoft.com/com/tech/complus.asp
Microsoft.NET-http://msdn2.microsoft.com/en-us/netframework/default.aspx
Sun JavaBeans-http://java.sun.com/products/ejb/

Estos sitios web tienen una amplia variedad de métodos de enseñanza, documentos en limpio, herramientas y recursos generados con dichos estándares importantes de middleware.

10.2 DISEÑO DE COMPONENTES BASADOS EN CLASE

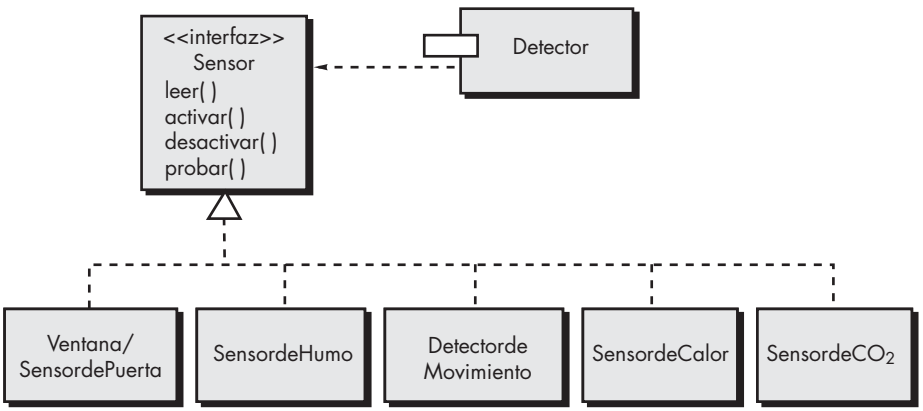
Como ya se dijo, el diseño en el nivel de componentes se basa en la información desarrollada como parte del modelo de requerimientos (capítulos 6 y 7) y se representa como parte del modelo arquitectónico (véase el capítulo 9). Cuando se escoge un enfoque de ingeniería orientado al software, el diseño en el nivel de componentes se centra en la elaboración de clases específicas del dominio del problema y en el refinamiento de las clases de infraestructura contenidas en el modelo de requerimientos. La descripción detallada de los atributos, operaciones e interfaces que emplean dichas clases es el detalle de diseño que se requiere como precursor de la actividad de construcción.

10.2.1 Principios básicos del diseño

Hay cuatro principios básicos que son aplicables al diseño en el nivel de componentes y que han sido ampliamente aceptados para la aplicación de la ingeniería de software orientada a objetos. La motivación subyacente para aplicar estos principios es crear diseños que sean más factibles de cambiar, así como reducir la propagación de efectos colaterales cuando se hagan cambios. Estos principios pueden usarse como guía cuando se desarrolle cada componente del software.

Principio Abierto-Cerrado (PAC). *"Un módulo [componente] debe ser abierto para la extensión pero cerrado para la modificación"* [Mar00]. Este enunciado parece ser una contradicción, pero representa una de las características más importantes de un buen diseño en el nivel de

FIGURA 10.4
Seguimiento del PAC



componentes. Dicho en pocas palabras, debe especificarse el componente en forma tal que permita extenderlo (dentro del dominio funcional a que está dirigido) sin necesidad de hacerle modificaciones internas (en el nivel del código o de la lógica). Para lograr esto, se crean abstracciones que sirven como búfer entre la funcionalidad que sea probable extender y la clase de diseño en sí.

Por ejemplo, suponga que la función de seguridad *CasaSegura* utiliza la clase **Detector** que debe revisar el estado de cada tipo de sensor de seguridad. Es probable que, conforme pase el tiempo, crezca el número y tipos de sensores de seguridad. Si la lógica de procesamiento interno se implementa como una secuencia de comandos si-entonces-en otro caso, cada uno dirigido a un tipo diferente de sensor, cuando se agregue uno nuevo se requerirá una lógica de procesamiento interno adicional (otro si-entonces-en otro caso). Esto sería una violación del PAC.

CASASEGURA

El PAC en acción

La escena: El cubículo de Vinod.

Participantes: Vinod y Shakira, miembros del equipo de ingeniería de software de *CasaSegura*.

La conversación:

Vinod: Me acaba de llamar Doug [gerente del equipo]. Dice que mercadotecnia quiere agregar un nuevo sensor.

Shakira (con sonrisa cómplice): No otra vez, por favor…

Vinod: Sí… y no vas a creer con lo que salieron…

Shakira: Sorpréndeme.

Vinod (ríe): Lo llaman *sensor de angustia del perro*.

Shakira: ¿Qué dijiste?

Vinod: Es para la gente que deja su mascota en departamentos o condominios o casas que están muy cerca una de otra. El perro comienza a ladrar, el vecino se enoja y se queja. Con este sensor, si el perro ladra durante, digamos, más de un minuto, el sensor hace sonar una alarma especial que llama al propietario a su teléfono móvil.

Shakira: Bromeas, ¿verdad?

Vinod: No, no. Doug quiere saber cuánto tiempo nos llevaría agregar eso a la función de seguridad.

Shakira (piensa un momento): No mucho… mira [muestra a Vinod la figura 10.4]. Hemos aislado las clases reales sensor atrás de la interfaz **sensor**. Cuando tengamos las especificaciones para el sensor perrito, será pan comido agregarlo. Lo único que tendré que hacer es crear un componente apropiado para él… mmm, una clase. El componente **Detector** no cambiará en absoluto.

Vinod: Entonces diré a Doug que no hay problema.

Shakira: Conociendo a Doug, nos estará vigilando; yo no le daría el asunto del perrito hasta la siguiente entrega.

Vinod: No está mal, pero podrías implantarlo ahora si él quisiera, ¿o no?

Shakira: Sí, la forma en la que diseñamos la interfaz me permite hacerlo sin problemas.

Vinod (piensa un momento): ¿Has oído hablar del Principio Abierto-Cerrado?

Shakira (encoge los hombros): Nunca.

Vinod (sonríe): No hay problema.

En la figura 10.4 se ilustra una forma de seguir el PAC para la clase **Detector**. La interfaz *sensor* presenta una consistente visión de los componentes sensores para los detectores. Si se agregara un nuevo tipo de sensor, no se requeriría hacer ningún cambio para la clase **Detector** (componente). Se preservaría el PAC.

Principio de sustitución de Liskov (PSL). *"Las subclases deben ser sustituibles por sus clases de base"* [Mar00]. Este principio de diseño, originalmente propuesto por Barbara Liskov [Lis88], sugiere que un componente que use una clase de base debe funcionar bien si una clase derivada de la clase base pasa al componente. El PSL demanda que cualquier clase derivada de una clase de base debe respetar cualquier contrato implícito entre la clase de base y los componentes que la usan. En el contexto de este análisis, un "contrato" es una *precondición* que debe ser verdadera antes de que el componente use una clase de base y una *poscondición* que debe ser verdadera después de ello. Cuando se crean clases derivadas hay que asegurarse de que respeten la precondición y la poscondición.

Si omite el diseño y pasa al código, sólo recuerde que el diseño es la última "concreción". Estaría violando el PID.

Principio de Inversión de la Dependencia (PID). *"Dependa de las abstracciones. No dependa de las concreciones"* [Mar00]. Como se vio en el estudio del PAC, las abstracciones son el lugar en el que es posible ampliar un diseño sin muchas dificultades. Entre más dependa un componente de otros componentes concretos (y no de abstracciones tales como una interfaz), más difícil será ampliarlo.

Principio de segregación de la interfaz (PSI). *"Es mejor tener muchas interfaces específicas del cliente que una sola de propósito general"* [Mar00]. Hay muchas instancias en las que múltiples componentes del cliente usan las operaciones que provee una clase servidor. El PSI sugiere que debe crearse una interfaz especializada que atienda a cada categoría principal de clientes. En la interfaz de ese cliente, sólo deben especificarse aquellas operaciones que sean relevantes para una categoría particular de clientes.

Por ejemplo, considere la clase **PlanodelaCasa** que se usó en las funciones de seguridad y vigilancia de *CasaSegura* (véase el capítulo 6). Para las funciones de seguridad, **PlanodelaCasa** se utiliza sólo durante las actividades de configuración y emplea las operaciones *SituarDispositivo( )*, *MostrarDispositivo( )*, *AgruparDispositivo( )* y *QuitarDispositivo( )* para situar, mostrar, agrupar y quitar sensores del plano de la casa. La función de vigilancia de *CasaSegura* usa las cuatro operaciones mencionadas para la seguridad, pero también requiere operaciones especiales para administrar cámaras: *MostrarFOV( )* y *MostrarIdentificacióndeDispositivo( )*. Entonces, el PSI sugiere que los componentes cliente de las dos funciones de *CasaSegura* tienen interfaces especializadas definidas para ellas. La interfaz para la seguridad incluiría sólo las operaciones *SituarDispositivo( )*, *MostrarDispositivo( )*, *AgruparDispositivo( )* y *QuitarDispositivo( )*. La interfaz para vigilancia incorporaría esas mismas operaciones pero también *MostrarFOV( )* y *MostrarIdentificacióndeDispositivo( )*.

Aunque los principios de diseño en el nivel de componentes son una guía útil, los componentes no existen en el vacío. En muchos casos, los componentes o clases individuales están organizados en subsistemas o paquetes. Es razonable preguntar cómo debe ocurrir esta actividad de agrupamiento. ¿Exactamente cómo deben organizarse los componentes conforme avanza el diseño? Martin [Mar00] propone principios adicionales de agrupamiento que son aplicables al diseño en el nivel de componentes:

Para que los componentes sean reutilizables, su diseño requiere algo más que un buen diseño técnico. También exige mecanismos efectivos de configuración (véase el capítulo 22).

Principio de equivalencia de la liberación de la reutilización (PER). *"El gránulo de reutilización es el gránulo de liberación"* [Mar00]. Cuando las clases o componentes se diseñan para ser reutilizables, existe un contrato implícito que se establece entre el desarrollador de la entidad reutilizable y las personas que la emplearán. El desarrollador se compromete a establecer un sistema que controle la liberación para que dé apoyo y mantenimiento a las versiones anteriores de la entidad mientras los usuarios se actualizan poco a poco con la versión más nueva.

En lugar de abordar cada clase individual, es frecuente que sea mejor agrupar las que sean reutilizables en paquetes que puedan manejarse y controlar a medida que evolucionen las nuevas versiones.

Principio de cierre común (PCC). *"Las clases que cambian juntas pertenecen a lo mismo"* [Mar00]. Las clases deben empacarse en forma cohesiva. Es decir, cuando las clases se agrupan como parte de un diseño, deben estar dirigidas a la misma área de funciones o comportamiento. Cuando deba cambiar alguna característica de dicha área, es probable que sólo aquellas clases que haya dentro del paquete requieran modificación. Esto lleva a un control de cambios y a un manejo de la liberación más eficaces.

Principio de la reutilización común (PRC). *"Las clases que no se reutilizan juntas no deben agruparse juntas"* [Mar00]. Cuando cambia una o más clases dentro de un paquete, cambia el número de liberación del paquete. Entonces, todas las demás clases o paquetes que permanecen en el paquete que cambió deben actualizarse con la liberación más reciente y someterse a pruebas a fin de garantizar que la nueva versión opera sin problemas. Si las clases no se agrupan de manera cohesiva, es posible que se cambie una clase sin relación junto con las demás que hay dentro del paquete. Esto generará integración y pruebas innecesarias. Por esta razón, sólo las clases que se reutilicen juntas deben incluirse dentro de un paquete.

10.2.2 Lineamientos de diseño en el nivel de componentes

Además de los principios estudiados en la sección 10.2.1, conforme avanza el diseño en el nivel de componentes se aplican lineamientos prácticos a los componentes, a sus interfaces y a las características de dependencia y herencia que tengan algún efecto en el diseño resultante. Ambler [Amb02b] sugiere los lineamientos siguientes:

? ¿Qué es lo que hay que tomar en cuenta al dar nombre a los componentes?

Componentes. Deben establecerse convenciones para dar nombre a los componentes que se especifique que forman parte del modelo arquitectónico, para luego mejorarlos y elaborarlos como parte del modelo en el nivel de componentes. Los nombres de los componentes arquitectónicos deben provenir del dominio del problema y significar algo para todos los participantes que vean el modelo arquitectónico. Por ejemplo, el nombre de la clase **PlanodelaCasa** tiene un significado para todos los que lo lean, aunque no tengan formación técnica. Por otro lado, los componentes de infraestructura o clases elaboradas en el nivel de componentes deben recibir un nombre que tenga un significado específico de la implantación. Si como parte de la implantación de **PlanodelaCasa** va a administrarse una lista vinculada, es apropiada la operación *AdministrarLista( )*, aun si una persona sin capacitación técnica pudiera interpretarlo mal.[3]

Pueden usarse estereotipos para ayudar a identificar la naturaleza de los componentes en el nivel de diseño detallado. Por ejemplo, **<<infraestructura>>** debiera usarse para identificar un componente de infraestructura, **<<basededatos>>** podría emplearse para identificar una base de datos que dé servicio a una o más clases de diseño o a todo el sistema; se usaría **<<tabla>>** para identificar una tabla dentro de una base de datos.

Interfaces. Las interfaces dan información importante sobre la comunicación y la colaboración (también nos ayudan a cumplir el PAC). Sin embargo, la representación sin restricciones de las interfaces tiende a complicar los diagramas de componentes. Ambler [Amb02c] recomienda que 1) si los diagramas aumentan en complejidad, en lugar del enfoque formal del UML con recuadro y flecha, debe representarse la interfaz con una paleta; 2) en aras de la consistencia, las interfaces deben fluir a partir del lado izquierdo del recuadro del componente; 3) sólo deben aparecer aquellas interfaces que sean relevantes para el componente que se está considerando,

3 No es probable que alguien de mercadotecnia o de la organización cliente (de tipo no técnico) analice la información de diseño detallado.

aun si estuvieran disponibles otras. Estas recomendaciones buscan simplificar la naturaleza visual de los diagramas UML de componentes.

Dependencias y herencia. Para tener una mejor legibilidad, es buena idea modelar las dependencias de izquierda a derecha y la herencia de abajo (clases obtenidas) hacia arriba (clases base). Además, las interdependencias de componentes deben representarse por medio de interfaces y no con la dependencia componente a componente. Según la filosofía del PAC, esto ayudará a hacer que sea más fácil dar mantenimiento al sistema.

10.2.3 Cohesión

En el capítulo 8 se describió la cohesión como la "unidad de objetivo" de un componente. En el contexto del diseño en el nivel de componentes para los sistemas orientados a objetos, la *cohesión* implica que un componente o clase sólo contiene atributos y operaciones que se relacionan de cerca uno con el otro y con la clase o componente en sí. Lethbridge y Laganiére [Let01] definen varios tipos diferentes de cohesión (se listan en función del nivel de cohesión):[4]

Funcional. Lo tienen sobre todo las operaciones; este nivel de cohesión ocurre cuando un componente realiza un cálculo y luego devuelve el resultado.

De capa. Lo tienen los paquetes, componentes y clases; este tipo de cohesión ocurre cuando una capa más alta accede a los servicios de otra más baja, pero ésta no tiene acceso a las superiores. Por ejemplo, considere el requerimiento de la función de seguridad de *CasaSegura* para hacer una llamada telefónica si se detecta una alarma. Podría definirse un conjunto de paquetes en capas, como se aprecia en la figura 10.5. Los paquetes sombreados contienen componentes de infraestructura. Es posible realizar el acceso del paquete del panel de control hacia abajo.

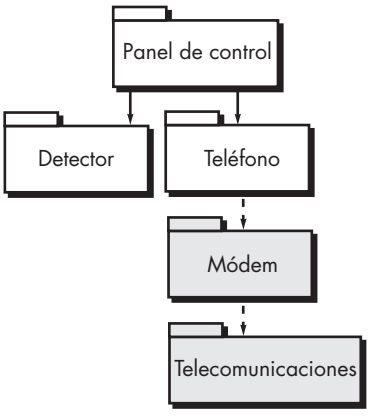
De comunicación. Todas las operaciones que acceden a los mismos datos se definen dentro de una clase. En general, tales clases se centran únicamente en los datos en cuestión, acceden a ellos y los guardan.

CONSEJO

Aunque es instructivo entender los distintos niveles de cohesión, es más importante tener presente el concepto general cuando se diseñen componentes. Mantenga la cohesión tan grande como sea posible.

Las clases y componentes que tienen cohesión funcional, de capa y comunicación son relativamente fáciles de implantar, probar y mantener. Siempre que sea posible, deben alcanzarse estos niveles de cohesión. Sin embargo, es importante notar que en ocasiones hay aspectos pragmáticos del diseño y de la implantación que obligan a optar por niveles de cohesión más bajos.

FIGURA 10.5
Cohesión de capa



4 En general, entre más alto sea el nivel de cohesión, el componente es más fácil de implantar, probar y mantener.

CasaSegura

La cohesión en acción

La escena: Cubículo de Jamie.

Participantes: Jamie y Ed, miembros del equipo de ingeniería de software que trabajan en la función de vigilancia.

La conversación:

Ed: Tengo un diseño de primer corte del componente **cámara**.

Jamie: ¿Quieres revisarlo rápido?

Ed: Sí… pero en realidad quisiera que me dijeras algo.

(Con señas, Jamie lo invita a que continúe.)

Ed: Originalmente definimos cinco operaciones para **cámara**. Mira…

DeterminarTipo( ) dice el tipo de cámara.

CambiarUbicación( ) permite mover la cámara por el plano de la casa.

MostrarIdentificación( ) obtiene la identificación de la cámara y la muestra cerca de su ícono.

MostrarVista( ) presenta el campo de visión de la cámara en forma gráfica.

MostrarAcercamiento( ) muestra gráficamente la amplificación de la cámara.

Ed: Las diseñé por separado y son operaciones muy simples. Por eso pensé que sería una buena idea combinar todas las operaciones de la pantalla en una sola que denominé *MostrarCámara( )* y que mostrará la identificación, vista y acercamiento. ¿Cómo la ves?

Jamie (hace una mueca): No estoy seguro de que sea una buena idea.

Ed (frunce el seño): ¿Por qué? Todas esas pequeñas operaciones pueden dar dolores de cabeza.

Jamie: El problema de que las combinemos es que se pierde cohesión, ya sabes, la operación *MostrarCámara( )* no tendrá un único objetivo.

Ed (un poco exasperado): ¿Y qué? Todo este asunto requerirá menos de 100 líneas de código fuente, si acaso. Será más fácil de implantar… creo.

Jamie: ¿Y qué pasa si decidimos cambiar la forma en la que representamos el campo de visión?

Ed: Sólo se pasa a la operación *MostrarCámara( )* y se hace la modificación.

Jamie: ¿Qué hay con los efectos colaterales?

Ed: ¿Qué quieres decir?

Jamie: Bueno, digamos que se hace el cambio, pero, sin darnos cuenta, se genera un problema al mostrar en la pantalla la identificación.

Ed: No sería tan torpe.

Jamie: Tal vez no, pero, ¿qué tal si alguien de apoyo tiene que hacer la modificación dentro de dos años? Tal vez no entenderá la operación tan bien como tú, y, ¿quién sabe?, podría ser torpe.

Ed: Entonces, ¿estás en contra?

Jamie: Tú eres el diseñador… es tu decisión… sólo asegúrate de que entiendes las consecuencias de la poca cohesión.

Ed (piensa un momento): Tal vez haga operaciones de pantalla separadas.

Jamie: Buena decisión.

10.2.4 Acoplamiento

En el estudio anterior del análisis y el diseño, se dijo que la comunicación y la colaboración eran elementos esenciales de cualquier sistema orientado a objetos. Sin embargo, esta característica tan importante (y necesaria) tiene un lado oscuro. A medida que aumentan la comunicación y colaboración (es decir, conforme se eleva la "conectividad" entre las clases), la complejidad del sistema también se incrementa. Y si la complejidad aumenta, también crece la dificultad de implantar, probar y dar mantenimiento al software.

A medida que se elabora el diseño de cada componente del software, la atención pasa al diseño de las estructuras específicas de los datos y al diseño de procedimientos para manipularlas. Sin embargo, no hay que olvidar la arquitectura que debe albergar los componentes de las estructuras globales de los datos, que tal vez atiendan a muchos componentes.

El *acoplamiento* es la medición cualitativa del grado en el que las clases se conectan una con otra. Conforme las clases (y componentes) se hacen más interdependientes, el acoplamiento crece. Un objetivo importante del diseño en el nivel de componente es mantener el acoplamiento tan bajo como sea posible.

El acoplamiento de las clases se manifiesta de varias maneras. Lethbridge y Laganiére [Let01] definen las siguientes categorías de acoplamiento:

Acoplamiento de contenido. Tiene lugar cuando un componente "modifica subrepticiamente datos internos en otro componente" [Let01]. Esto viola el ocultamiento de la información, concepto básico del diseño.

Acoplamiento común. Sucede cuando cierto número de componentes hacen uso de una variable global. Aunque a veces esto es necesario (por ejemplo, para establecer valores de-

finidos que se utilizan en toda la aplicación), el acoplamiento común lleva a la propagación incontrolada del error y a efectos colaterales imprevistos cuando se hacen los cambios.

Acoplamiento del control. Tiene lugar si la operación *A( )* invoca a la operación *B( )* y pasa una bandera de control a *B*. La bandera "dirige" entonces el flujo de la lógica dentro de *B*. El problema con esta forma de acoplamiento es que un cambio no relacionado en *B* puede dar como resultado la necesidad de cambiar el significado de la bandera de control que pasa *A*. Si esto se pasa por alto ocurrirá un error.

Acoplamiento de molde. Se presenta cuando se declara a **ClaseB** como un tipo para un argumento de una operación de **ClaseA**. Como **ClaseB** ahora forma parte de la definición de **ClaseA**, la modificación del sistema se vuelve más compleja.

Acoplamiento de datos. Ocurre si las operaciones pasan cadenas largas de argumentos de datos. El "ancho de banda" de la comunicación entre clases y componentes crece y la complejidad de la interfaz se incrementa. Se hace más difícil hacer pruebas y dar mantenimiento.

Acoplamiento de rutina de llamada. Tiene lugar cuando una operación invoca a otra. Este nivel de acoplamiento es común y con frecuencia muy necesario. Sin embargo, aumenta la conectividad del sistema.

Acoplamiento de tipo de uso. Ocurre si el componente **A** usa un tipo de datos definidos en el componente **B** (esto ocurre siempre que "una clase declara una variable de instancia o una variable local como si tuviera otra clase para su tipo" [Let01]). Si cambia la definición de tipo, también debe cambiar todo componente que la utilice.

Acoplamiento de inclusión o importación. Pasa cuando el componente **A** importa o incluye un paquete o el contenido del componente **B**.

Acoplamiento externo. Sucede si un componente se comunica o colabora con componentes de infraestructura (por ejemplo, funciones del sistema operativo, capacidad de la base de datos, funciones de telecomunicación, etc.). Aunque este tipo de acoplamiento es necesario, debe limitarse a un número pequeño de componentes o clases dentro de un sistema.

El software debe tener comunicación interna y externa. Por tanto, el acoplamiento es un hecho de la vida. Sin embargo, el diseñador debe trabajar para reducirlo siempre que sea posible, y entender las ramificaciones que tiene el acoplamiento abundante cuando no puede evitarse.

CasaSegura

El acoplamiento en acción

La escena: Cubículo de Shakira.

Participantes: Vinod y Shakira, miembros del equipo de software de CasaSegura, que trabajan en la función de seguridad.

La conversación:

Shakira: Tuve lo que considero una gran idea… entonces lo pensé un poco y me pareció que no era tan buena. Al final la deseché, pero pensé en hacerla para ustedes.

Vinod: Seguro. ¿Cuál es la idea?

Shakira: Bueno, cada uno de los sensores reconoce una condición de alarma de algún tipo, ¿verdad?

Vinod (sonríe): Por eso se llaman sensores, Shakira.

Shakira (exasperada): Sarcasmo, Vinod, tienes que mejorar tus habilidades interpersonales.

Vinod: ¿Decías?

Shakira: Bien, de cualquier modo, me pregunté… por qué no crear una operación dentro de cada objeto de sensor llamada *HacerLlamada( )* que colaboraría directamente con el componente **SaleLlamada**, bueno, con una interfaz hacia el componente **SaleLlamada**.

Vinod (pensativo): Quieres decir, ¿eso en vez de hacer que esa colaboración ocurra fuera de un componente como **PaneldeControl** o algún otro?

Shakira: Sí… Pero entonces me dije que eso significaba que cada objeto de sensor estaría conectado al componente **SaleLlamada**, y que eso querría decir que estaría acoplado de manera indirecta con el mundo exterior y… bueno, pensé que sólo complicaría las cosas.

Vinod: Estoy de acuerdo. En este caso, es mejor idea dejar que la interfaz del sensor pase información a **PaneldeControl** y que inicie la llamada de salida. Además, diferentes sensores tal vez darían como resultado números telefónicos distintos. Tú no querrías que el sensor guardara esa información, porque si cambiara…

Shakira: No lo sentí bien.

Vinod: La heurística del diseño para el acoplamiento nos dice que no está bien.

Shakira: Pues sí.

10.3 Realización del diseño en el nivel de componentes

Cita:

"Si hubiera tenido más tiempo habría escrito una carta más breve."

Blas Pascal

Antes, en este capítulo, se dijo que el diseño en el nivel de componentes es de naturaleza elaborativa. Debe transformarse la información de los modelos de requerimientos y arquitectónico a una representación de diseño que dé suficientes detalles para guiar la actividad de construcción (codificación y pruebas). Los pasos siguientes representan un conjunto de tareas comunes para el diseño en el nivel de componentes cuando se aplica a un sistema orientado a objetos.

Paso 1. Identificar todas las clases de diseño que correspondan al dominio del problema. Con el uso del modelo de requerimientos y arquitectónico, se elabora cada clase de análisis y componente de la arquitectura según se describió en la sección 10.1.1.

CONSEJO

Si se trabaja en un ambiente sin espías, los primeros tres pasos se dirigen a refinar los objetos de datos y las funciones de procesamiento (transformaciones) identificadas como parte del modelo de requerimientos.

Paso 2. Identificar todas las clases de diseño que correspondan al dominio de la infraestructura. Estas clases no están descritas en el modelo de los requerimientos y con frecuencia se pierden a partir del modelo arquitectónico; sin embargo, deben describirse en este punto. Como se dijo, las clases y componentes en esta categoría incluyen componentes de la interfaz gráfica de usuario (con frecuencia disponibles como componentes reutilizables), componentes del sistema operativo y componentes de administración de objetos y datos.

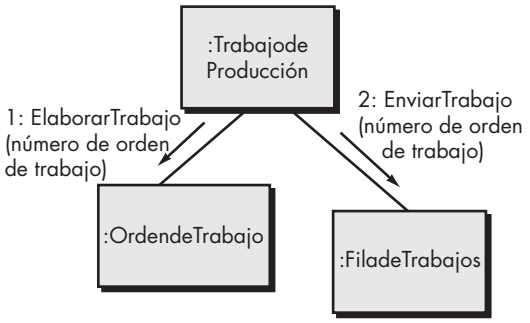
Paso 3. Elaborar todas las clases de diseño que no sean componentes reutilizables. La elaboración requiere que se describan en detalle todas las interfaces, atributos y operaciones necesarios para implantar la clase. Mientras se realiza esta tarea, deben considerarse los heurísticos del diseño (como la cohesión y el acoplamiento del componente).

Paso 3*a*. Especificar detalles del mensaje cuando colaboren clases o componentes. El modelo de requerimientos utiliza un diagrama de colaboración para mostrar la forma en la que las clases de análisis colaboran una con la otra. A medida que avanza el diseño en el nivel de componentes, en ocasiones es útil mostrar los detalles de estas colaboraciones especificando las estructuras de los mensajes que pasan entre los objetos de un sistema. Aunque esta actividad de diseño es opcional, se usa como precursor de la especificación de interfaces que muestren el modo en el que se comunican y colaboran los componentes del sistema.

La figura 10.6 ilustra un diagrama sencillo de colaboración para el sistema de impresión que ya se mencionó. Los objetos **TrabajodeProducción**, **OrdendeTrabajo** y **FiladeTrabajos** colaboran para preparar un trabajo de impresión a fin de ejecutar una secuencia de producción. Los mensajes entre los objetos se transmiten como lo ilustran las flechas en la figura. Durante la modelación de los requerimientos, los mensajes se especifican como se aprecia. Sin embargo, conforme el diseño avanza, cada mensaje se elabora expandiendo su sintaxis de la manera siguiente [Ben02]:

```
[condición de guardia] expresión de secuencia (devuelve valor) :=
nombre del mensaje (lista de argumentos)
```

FIGURA 10.6
Diagrama de colaboración con mensajería



donde **[condición de guardia]** se escribe en Lenguaje de Restricción de Objetos (LRO)[5] y especifica cualesquiera condiciones que deban cumplirse antes de que pueda enviarse el mensaje; **expresión de secuencia** es un valor entero (u otro indicador de ordenación, por ejemplo, 3.1.2) que indica el orden secuencial en el que se envía el mensaje; **(devuelve valor)** es el nombre de la información que devuelve la operación invocada por el mensaje; **nombre del mensaje** identifica la operación que va a invocarse y **(lista de argumentos)** es la lista de atributos que se pasan a la operación.

Paso 3*b*. Identificar interfaces apropiadas para cada componente. En el contexto del diseño en el nivel de componentes, una interfaz UML es un "grupo de operaciones visibles externamente (para el público). La interfaz no contiene estructura interna, ni atributos ni asociaciones..." [Ben02]. Dicho con más formalidad, una interfaz es el equivalente de una clase abstracta que provee una conexión controlada entre clases de diseño. En la figura 10.1 se ilustra la elaboración de interfaces. En esencia, las operaciones definidas para la clase de diseño se clasifican en una o más clases abstractas. Cada operación dentro de la clase abstracta (la interfaz) debe ser cohesiva, es decir, debe tener un procesamiento que se centre en una función o subfunción limitada.

En relación con la figura 10.1, puede afirmarse que la interfaz *IniciarTrabajo* no tiene suficiente cohesión. En realidad, ejecuta tres subfunciones diferentes: elaborar una orden de trabajo, verificar la prioridad del trabajo y pasar el trabajo a producción. La interfaz debe rediseñarse. Un enfoque podría consistir en volver a estudiar las clases de diseño y definir una nueva, **OrdendeTrabajo**, que se haría cargo de todas las actividades asociadas con la formación de una orden de trabajo. La operación *ElaborarOrdendeTrabajo( )* se vuelve parte de esa clase. De manera similar, se definiría una clase **FiladeTrabajos** que incorporaría la operación *VerificarPrioridad( )*. Una clase **TrabajodeProducción** podría incorporar toda la información asociada con un trabajo que pasara a las instalaciones de producción. La interfaz *IniciarTrabajo* podría adoptar la forma que se muestra en la figura 10.7. Ahora la interfaz *IniciarTrabajo* es cohesiva y se centra en una función. Las interfaces asociadas con **TrabajodeProducción, OrdendeTrabajo** y **FiladeTrabajos** también tienen un solo objetivo.

Paso 3*c*. Elaborar atributos y definir tipos y estructuras de datos requeridos para implantarlos. En general, las estructuras y tipos de datos usados para definir atributos se definen en el contexto del lenguaje de programación que se va a usar para la implantación. El UML define un tipo de datos del atributo que usa la siguiente sintaxis:

nombre : tipo-de-expresión = valor-inicial {cadena de propiedades}

5 En el apéndice 1 se analiza brevemente el LRO.

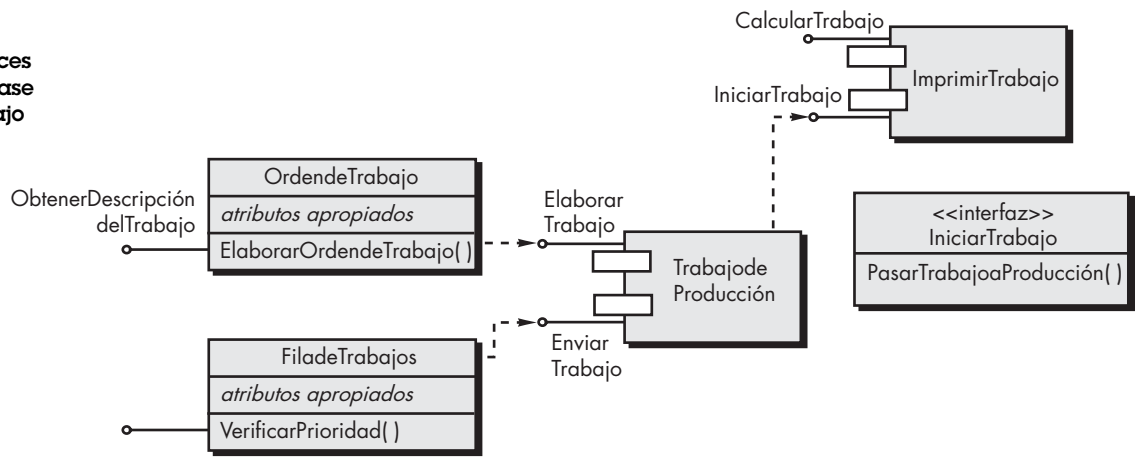


FIGURA 10.7
Rediseño de interfaces y definiciones de clase para ImprimirTrabajo

donde **nombre** es el nombre del atributo, **tipo-de-expresión** es el tipo de datos, **valor-inicial** es el valor que toma el atributo cuando se crea un objeto y **cadena de propiedades** define una propiedad o característica del atributo.

Durante la primera iteración del diseño en el nivel de componentes, los atributos normalmente se describen por su nombre. De nuevo, en la figura 10.1, la lista de atributos para **ImprimirTrabajo** sólo enlista los nombres de los atributos. No obstante, a medida que avanza la elaboración del diseño, cada atributo se define con el formato de atributos UML mencionado. Por ejemplo, **Tipodepapel-peso** se define del modo siguiente:

Tipodepapel-peso: cadena = "A" {contiene 1 de 4 valores – A, B, C o D}

que define a **Tipodepapel-peso** como una variable de cadena que se inicializa en el valor A y que puede adoptar uno de cuatro valores del conjunto {A, B, C, D}.

Si un atributo aparece en forma repetida en varias clases de diseño y tiene una estructura relativamente compleja, es mejor crear una clase separada para que lo albergue.

Paso 3*d*. Describir en detalle el flujo del procesamiento dentro de cada operación. Esto se logra con el uso de seudocódigo basado en lenguaje de programación o con un diagrama UML de actividades. Cada componente del software se elabora a través de cierto número de iteraciones que apliquen paso a paso el concepto de refinamiento (capítulo 8).

La primera iteración define cada operación como parte de la clase de diseño. En cada caso, la operación debe caracterizarse en forma tal que asegure que haya mucha cohesión; es decir, la operación debe estar dirigida a la ejecución de una sola función o subfunción. La siguiente iteración no hace más que expandir el nombre de la operación. Por ejemplo, la operación *CalcularCostodelPapel( )* que aparece en la figura 10.1 se expande de la manera siguiente:

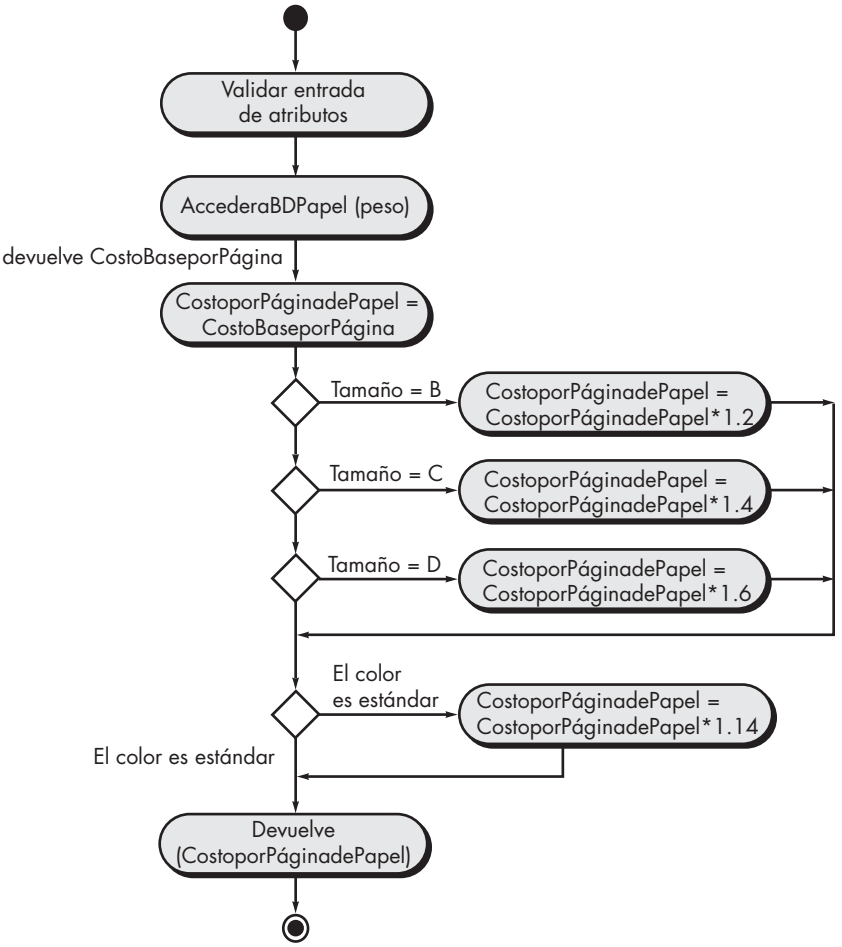
CalcularCostodelPapel (peso, tamaño, color): numérico

Esto indica que *CalcularCostodelPapel( )* requiere como entrada los atributos **peso**, **tamaño** y **color**, y da como salida un valor numérico (valor en dólares).

Para refinar el diseño del componente, utilice una elaboración stepwise. Siempre pregunte, "¿hay una forma de simplificar esto y que aun así se logre el mismo resultado?"

Si el algoritmo requerido para implantar *CalcularCostodelPapel( )* es sencillo y entendido por todos, tal vez no sea necesaria una mayor elaboración del diseño. El ingeniero de software que haga la codificación proveerá los detalles necesarios para implantar la operación. Sin embargo, si el algoritmo fuera más complejo o difícil de entender, se requeriría elaborar más el diseño. La figura 10.8 ilustra un diagrama UML de actividades para *CalcularCostodelPapel( )*. Cuando se usan diagramas de actividades para especificar el diseño en el nivel de componentes, por lo

FIGURA 10.8
Diagrama de actividades UML para calcular *CostodelPapel( )*



general se representan en un nivel de abstracción que es algo mayor que el código fuente. En la sección 10.5.3 se estudia un enfoque alternativo: el uso de seudocódigo para hacer las especificaciones del diseño.

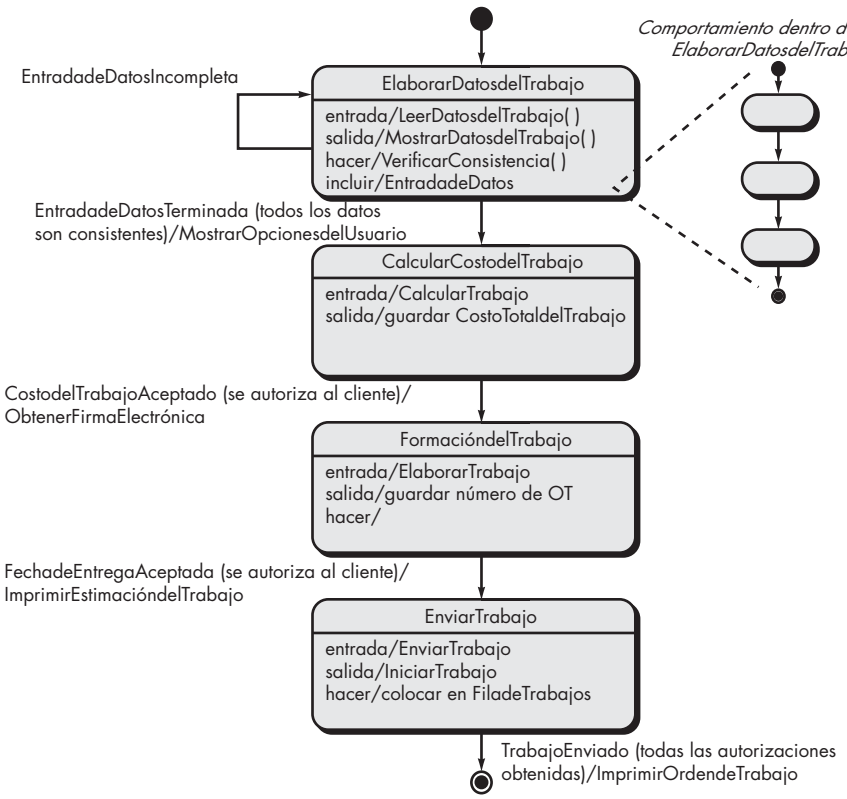
Paso 4. Describir las fuentes persistentes de datos (bases de datos y archivos) e identificar las clases requeridas para administrarlos. Es normal que las bases de datos y archivos trasciendan la descripción del diseño de un componente individual. En la mayoría de casos, estos almacenamientos persistentes de datos se especifican al inicio como parte del diseño de la arquitectura. No obstante, a medida que avanza la elaboración del diseño, es frecuente que sea útil dar detalles adicionales sobre la estructura y organización de dichas fuentes persistentes de datos.

Paso 5. Desarrollar y elaborar representaciones del comportamiento para una clase o componente. Los diagramas de estado UML fueron utilizados como parte del modelo de los requerimientos para representar el comportamiento observable desde el exterior del sistema y el más localizado de las clases de análisis individuales. Durante el diseño en el nivel de componentes, en ocasiones es necesario modelar el comportamiento de una clase de diseño.

El comportamiento dinámico de un objeto (instancias de una clase de diseño cuando el programa se ejecuta) se ve afectado por eventos externos a él y por el estado en curso (modo de comportamiento) del objeto. Para entender el comportamiento dinámico de un objeto, deben estudiarse todos los casos de uso que sean relevantes para la clase de diseño a lo largo de su

FIGURA 10.9

Fragmento de un diagrama de estado para la clase ImprimirTrabajo



vida. Estos casos de uso dan información que ayuda a delinear los eventos que afectan al objeto y los estados en los que éste reside conforme pasa el tiempo y suceden los eventos. Las transiciones entre estados (dictadas por los eventos) se representan con un diagrama de estado [Ben02] como el que se ilustra en la figura 10.9.

La transición de un estado al otro (representada por un rectángulo con esquinas redondeadas) ocurre como consecuencia de un evento que tiene la forma siguiente:

Nombre-del-evento (lista-de-parámetros) [guardar-condición] / expresión de acción

donde **nombre-del-evento** identifica al evento, **lista-de-parámetros** incorpora datos asociados con el evento, **guardar-condición** se escribe en Lenguaje de Restricción de Objetos (LRO) y especifica una condición que debe cumplirse para que pueda ocurrir el evento, y **expresión de acción** define una acción que ocurre a medida que toma lugar la transición.

En la figura 10.9, cada estado define acciones de *entrada/* y *salida/* que ocurren cuando sucede la transición al estado y fuera de éste, respectivamente. En la mayor parte de los casos, estas acciones corresponden a operaciones que son relevantes para la clase que se modela. El indicador *hacer/* proporciona un mecanismo para señalar actividades que tienen lugar mientras se está en el estado y el indicador *incluir/* proporciona un medio para elaborar el comportamiento incrustando más detalles del diagrama de estado en la definición de un estado.

Es importante observar que el modelo del comportamiento con frecuencia contiene información que no es obvia a primera vista en otros modelos del diseño. Por ejemplo, el análisis cuidadoso del diagrama de estado en la figura 10.9 indica que el comportamiento dinámico de la clase **ImprimirTrabajo** es contingente mientras no se obtengan dos aprobaciones de datos de costo y programación para el trabajo de impresión. Sin las aprobaciones (la condición de guar-

dar garantiza que el cliente está autorizado para aprobar), el trabajo de impresión no puede enviarse porque no hay forma de alcanzar el estado *EnviarTrabajo*.

Paso 6. Elaborar diagramas de despliegue para dar más detalles de la implantación. Los diagramas de despliegue (véase el capítulo 8) se utilizan como parte del diseño de la arquitectura y se representan en forma de descriptor. De este modo, las funciones principales de un sistema (que con frecuencia se representan como subsistemas) se representan en el contexto del ambiente de computación que las contendrá.

Durante el diseño en el nivel de componentes, pueden elaborarse diagramas de despliegue que representen la ubicación de paquetes de componentes clave. Sin embargo, en un diagrama de componentes, éstos por lo general no se representan de manera individual a fin de evitar la complejidad del diagrama. En ciertos casos, se elaboran diagramas de despliegue en forma de instancia en ese momento. Esto significa que se especifican los ambientes del hardware y el sistema operativo que se emplearán, y se indica la ubicación de los paquetes de componentes dentro de este ambiente.

Paso 7. Rediseñar cada representación del diseño en el nivel de componentes y siempre considerar alternativas. En este libro se hace hincapié en que el diseño es un proceso iterativo. El primer modelo en el nivel de componentes que se crea no será tan completo, consistente o exacto como la enésima iteración que se realice. Es esencial rediseñar a medida que se ejecuta el trabajo de diseño.

Además, no debe adoptarse una visión de túnel. Siempre hay soluciones alternativas para el diseño y los mejores diseñadores toman en cuenta todas ellas (o a la mayoría) antes de decidirse por el modelo de diseño final. Desarrolle las alternativas y estudie con cuidado cada una, con los principios y conceptos del diseño presentados en el capítulo 8 y en éste.

10.4 DISEÑO EN EL NIVEL DE COMPONENTES PARA WEBAPPS

Es frecuente que cuando se trata de sistemas y aplicaciones basados en web, la frontera entre el contenido y la función sea borrosa. Por tanto, es razonable preguntar: ¿qué es un componente de *webapps*?

En el contexto de este capítulo, un componente de *webapp* es 1) una función cohesiva bien definida que manipula contenido o da procesamiento de cómputo o de datos para un usuario final o 2) un paquete cohesivo de contenido y funciones que brindan al usuario final alguna capacidad solicitada. Entonces, el diseño en el nivel de componentes de *webapps* con frecuencia incorpora elementos de diseño del contenido y de las funciones.

10.4.1 Diseño del contenido en el nivel de componente

El diseño del contenido en el nivel de componentes se centra en objetos de contenido y en la forma en la que se empacan para su presentación a un usuario final de *webapps*. Por ejemplo, considere una capacidad de vigilancia con video dentro de **CasaSeguraAsegurada.com**. Entre muchas otras capacidades, el usuario selecciona y controla cualesquiera cámaras representadas en el plano de la casa, solicita imágenes instantáneas de todas las cámaras y muestra un video desde cualquiera de ellas. Además, el usuario tiene la posibilidad de abrir el ángulo o de hacer acercamientos con una cámara por medio de los íconos apropiados de control.

Para la capacidad de vigilancia con video, pueden definirse varios componentes potenciales de contenido: 1) objetos de contenido que representen la distribución del espacio (el plano de la casa) con íconos adicionales que representen la ubicación de sensores y cámaras de video, 2) el conjunto de imágenes instantáneas de video (cada una es un objeto de datos separado) y 3) la ventana del video de una cámara específica. Cada uno de estos componentes recibe un nombre por separado y se manipula como paquete.

Considere el plano de la casa con cuatro cámaras colocadas estratégicamente en una casa. A solicitud del usuario, se captura una toma de video desde cada cámara y se identifica como un objeto de contenido generado en forma dinámica, **TomadeVideo*N***, donde *N* identifica las cámaras 1 a 4. Un componente de contenido, llamado **ImágenesInstantáneas**, combina los cuatro objetos de contenido **TomadeVideo*N*** y los muestra en la página de vigilancia con video.

La formalidad del diseño del contenido en el nivel de componentes debe adaptarse a las características de la *webapp* que se va a elaborar. En muchos casos, los objetos de contenido no necesitan estar organizados como componentes y pueden manipularse en forma individual. Sin embargo, a medida que aumentan el tamaño y la complejidad (de la *webapp*, los objetos de contenido y sus interrelaciones), es necesario organizar el contenido en forma que sea fácil hacer referencia y manipular el diseño.[6] Además, si el contenido es muy dinámico (por ejemplo, el de un sitio de subastas en línea), es importante establecer un modelo estructural claro que incorpore los componentes del contenido.

10.4.2 Diseño de las funciones en el nivel de componentes

Las aplicaciones web modernas proporcionan funciones de procesamiento cada vez más sofisticadas que: 1) producen un procesamiento localizado que genera contenido y capacidad de navegación en forma dinámica; 2) dan capacidad de computación o procesamiento de datos que resultan adecuados para el dominio del negocio de la *webapp*; 3) brindan consultas y acceso avanzado a una base de datos, y 4) establecen interfaces de datos con sistemas corporativos externos. Para lograr las capacidades anteriores (y muchas otras), se diseñan componentes de la *webapp* que tengan forma similar a la de los componentes del software convencional.

Las funciones de la *webapp* se entregan como una serie de componentes desarrollados en paralelo con la arquitectura de la información que garantice que sean consistentes. En esencia, se comienza con la consideración del modelo de requerimientos y de la arquitectura inicial de la información, para luego estudiar el modo en el que las funciones afectan la interacción del usuario con la aplicación, la información que se presenta y las tareas que el usuario realiza.

Durante el diseño de la arquitectura, el contenido y funciones de la *webapp* se combinan para crear una *arquitectura funcional*, que es una representación del dominio de funciones de la *webapp* y describe los componentes funcionales clave de la *webapp* y la forma en la que interactúan una con otra.

Por ejemplo, las funciones de abrir el ángulo y hacer acercamientos de la capacidad de vigilancia con video para **CasaSeguraAsegurada.com** se implementan como las operaciones *recorrer( )* y *zoom( )*, que forman parte de la clase **Cámara**. En cualquier caso, la funcionalidad implícita para *recorrer( )* y *zoom( )* deben implantarse como módulos dentro de **CasaSeguraAsegurada.com**.

10.5 Diseño de componentes tradicionales

PUNTO CLAVE

La programación estructurada es una técnica de diseño que limita el flujo de la lógica a tres construcciones: secuencia, condición y repetición.

Los fundamentos del diseño en el nivel de componentes para el software tradicional[7] se establecieron a principios de la década de 1960 y se formalizaron con el trabajo de Edsger Dijkstra *et al.* ([Boh66], [Dij65] y [Dij76b]). A finales de esa época, ellos propusieron el empleo de un conjunto de construcciones lógicas restringidas con las que pudiera elaborarse cualquier programa.

6 Los componentes del contenido también pueden volverse a utilizar en otras *webapps*.

7 Un componente tradicional de software implementa un elemento de procesamiento abocado a una función o subfunción en el dominio del problema, o cierta capacidad en el dominio de la infraestructura. Los componentes tradicionales, llamados con frecuencia módulos, procedimientos o subrutinas, no incluyen datos en la misma forma en la que lo hacen los componentes orientados a objetos.

Las construcciones ponían el énfasis en el "mantenimiento del dominio funcional". Es decir, cada construcción tenía una estructura lógica predecible que se introducía al principio y salía por el final, lo que permitía que el lector siguiera el flujo del procedimiento con más facilidad.

Las construcciones son secuencia, condición y repetición. La *secuencia* implementa pasos de procesamiento que son esenciales en la especificación de cualquier algoritmo. La *condición* proporciona el medio para seleccionar un procesamiento con base en algún suceso lógico y la *repetición* permite la ejecución de lazos. Estas tres construcciones son fundamentales para la *programación estructurada*, técnica importante del diseño en el nivel de componentes.

Las construcciones estructuradas fueron propuestas para limitar el diseño del software orientado al procedimiento a un número pequeño de estructuras lógicas predecibles. La medición de la complejidad (véase el capítulo 23) indica que el uso de las construcciones estructuradas reduce la complejidad del programa y con ello mejora la legibilidad y la facilidad de realizar pruebas y de dar mantenimiento. El uso de un número limitado de construcciones lógicas también contribuye a un proceso de comprensión humana que los sicólogos denominan *lotificación*. Para entender este proceso, considere el lector la forma en la que está leyendo esta página. No necesita leer las letras en lo individual, sino reconocer patrones o grupos de letras que forman palabras o frases. Las construcciones estructuradas son grupos lógicos que permiten al lector reconocer elementos de procedimiento de un módulo, en vez de leer el diseño o el código línea por línea. La comprensión mejora cuando se encuentran patrones lógicos que es fácil reconocer.

Con tan sólo las tres construcciones estructuradas es posible diseñar e implantar cualquier programa, sin importar el área de aplicación o complejidad técnica. Sin embargo, debe notarse que su empleo dogmático con frecuencia ocasiona dificultades prácticas. En la sección 10.5.1 se estudia esto con más detalles.

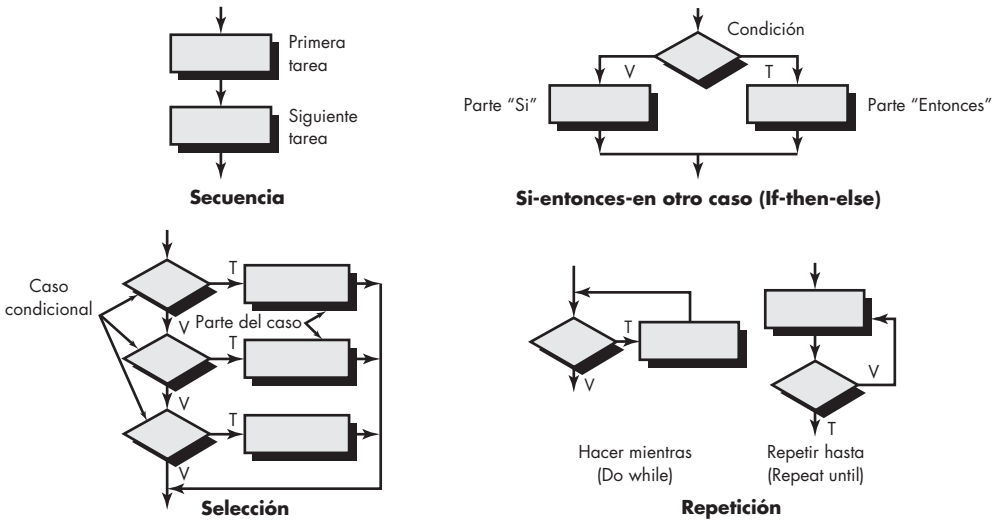
10.5.1 Notación gráfica de diseño

"Una imagen vale más que mil palabras", pero es importante saber de qué imagen se trata y cuáles serían las mil palabras. No hay duda de que herramientas gráficas, como el diagrama UML de actividades o el diagrama de flujo, constituyen patrones gráficos útiles que ilustran fácilmente detalles de procedimiento. No obstante, si se hace mal uso de las herramientas gráficas, surge una imagen equivocada que conduce al software equivocado.

El diagrama de actividades permite representar la secuencia, condición y repetición —todos los elementos de que consta la programación estructurada— y es descendiente de un diseño gráfico anterior (que todavía se utiliza mucho) llamado *diagrama de flujo*. Como cualquier diagrama de actividades, el de flujo es muy simple. Se emplea un rectángulo para indicar un paso de procesamiento. Un rombo representa una condición lógica y las flechas indican el flujo del control. La figura 10.10 ilustra tres construcciones estructuradas. La *secuencia* se representa como dos cajas de procesamiento conectadas por una línea (flecha) de control. La *condición*, también llamada *si-entonces-en otro caso*, se ilustra como un rombo de decisión que, si se da el estado de "verdadero", hace que ocurra la *parte del entonces*, y si el estado es "falso", se invoca el procesamiento de la *parte en otro caso*. La *repetición* se representa con el uso de dos formas ligeramente distintas. Las pruebas *hacer mientras*, prueban una condición y ejecutan repetidamente un lazo de tareas mientras la condición sea verdadera. La parte *repetir hasta* primero ejecuta el lazo de la tarea y después prueba una condición y repite la tarea hasta que la condición se vuelve falsa. La construcción *selección* (o *caso seleccionar*) que se aprecia en la figura en realidad es una extensión de la cláusula *si-entonces-en otro caso*. Un parámetro se somete a prueba por medio de decisiones sucesivas hasta que ocurre una condición de "verdadero" y se ejecuta el procesamiento por la trayectoria de la *parte caso*.

En general, el uso dogmático de construcciones estructuradas introduce ineficiencia cuando se requiere una salida de un grupo de lazos o condiciones anidadas. Más importante aún, la complicación adicional de todas las pruebas lógicas y de la ruta de salida llega a oscurecer el

FIGURA 10.10
Construcciones de los diagramas de flujo



flujo de control del software, aumenta la posibilidad de errores y tiene un efecto negativo en la legibilidad y facilidad de dar mantenimiento al software. ¿Qué hacer?

Hay dos opciones: 1) rediseñar la representación del procedimiento de modo que no se requiera que la "rama de salida" esté en una ubicación anidada en el flujo del control o 2) violar en forma controlada las construcciones estructuradas; es decir, diseñar una rama restringida fuera del flujo anidado. Es obvio que la opción 1 es el enfoque ideal, pero la 2 se consigue sin violar el espíritu de la programación estructurada.

10.5.2 Notación del diseño tabular

CONSEJO

Úsese una tabla de decisión cuando dentro de un componente se halle un conjunto complejo de condiciones y acciones.

En muchas aplicaciones de software, se requiere un módulo para evaluar una combinación compleja de combinaciones de condiciones y seleccionar acciones apropiadas con base en éstas. Las *tablas de decisión* [Hur83] proporcionan una notación que traduce las acciones y condiciones (descritas en la narración del procesamiento o caso de uso) a una forma tabular. Es difícil malinterpretar la tabla e incluso se puede usar como entrada legible por una máquina que la use en un algoritmo dirigido por aquélla.

En la figura 10.11 se muestra la organización de la tabla de decisión. La tabla se divide en cuatro secciones. El cuadrante superior izquierdo contiene la lista de todas las condiciones. El inferior izquierdo contiene la lista de condiciones que son posibles con base en combinaciones de las condiciones. Los cuadrantes del lado derecho forman una matriz que indica combinaciones de las condiciones y las correspondientes acciones que ocurrirán para una combinación específica de éstas. Por tanto, cada columna de la matriz se interpreta como *regla de procesamiento.* Para desarrollar una tabla de decisión se emplean los pasos siguientes:

? ¿Cómo elaboro una tabla de decisión?

1. Enlistar todas las acciones asociadas con un procedimiento (o componente) específico.
2. Enlistar todas las condiciones (o decisiones tomadas) durante la ejecución del procedimiento.
3. Asociar conjuntos específicos de condiciones con acciones específicas, con la eliminación de las combinaciones o con condiciones imposibles; de manera alternativa, desarrollar toda posible permutación de las condiciones.
4. Definir reglas indicando qué acciones suceden para un conjunto dado de condiciones.

Para ilustrar el uso de una tabla de decisión, considere el siguiente extracto de un caso de uso informal propuesto para el sistema del taller de impresión:

FIGURA 10.11
Nomenclatura de una tabla de decisión

	Reglas					
Condiciones	**1**	**2**	**3**	**4**	**5**	**6**
Cliente regular	T	T				
Cliente plateado			T	T		
Cliente dorado					T	T
Descuento especial	F	T	F	T	F	T
Acciones						
Sin descuento	✓					
Aplicar 8% de descuento			✓	✓		
Aplicar 15% de descuento					✓	✓
Aplicar un porcentaje adicional de descuento		✓		✓		✓

Se definen tres tipos de clientes: regular, plateado y dorado (que se asignan de acuerdo con la cantidad de negocios que realice el cliente con el taller de impresión en un periodo de 12 meses). Un cliente regular recibe tarifas normales por la impresión y entrega. Uno plateado obtiene un descuento de 8 por ciento sobre todas sus compras y es colocado por delante de todos los clientes regulares en la fila de trabajos. Un cliente dorado recibe 15 por ciento de descuento sobre los precios de lista y es puesto adelante de los clientes regulares y de los plateados en la fila de trabajos. Además de los descuentos anteriores, se aplica un descuento especial de *x* porcentaje a cualquier cliente según el criterio de la administración.

La figura 10.11 ilustra una representación de tabla de decisión del caso de uso anterior. Cada una de las seis reglas indica una de las seis condiciones posibles. Por regla general, la tabla de decisión se usa de manera eficaz para dar otra notación de diseño orientado al procedimiento.

10.5.3 Lenguaje de diseño del programa

El *lenguaje de diseño del programa* (LDP), también llamado *castellano estructurado* o *seudocódigo*, incorpora la estructura lógica de un lenguaje de programación y la expresividad de forma libre de un lenguaje natural (como el castellano). Se incrusta el texto de la narración (en castellano) en una sintaxis de programación parecida al idioma. Para mejorar la aplicación del LDP se utilizan herramientas automatizadas (ver [Cai03]).

Una sintaxis básica de LDP debe incluir construcciones para: definir los componentes, describir la interfaz, hacer la declaración de los datos, estructurar bloques, hacer construcciones condicionales, de repetición y de entrada y salida (E/S). Debe observarse que el LDP se amplía a fin de que incluya palabras clave para hacer un procesamiento de tareas múltiples o concurrentes, manejar interrupciones, sincronía entre procesos y muchas otras características. El diseño de la aplicación para el que se utilizará el LDP es lo que debe dictar la forma final para el lenguaje del diseño. En el ejemplo siguiente se presenta el formato y semántica de algunas de estas construcciones de LDP.

Para ilustrar el uso del LDP, piense en un diseño orientado al procedimiento para la función de seguridad de *CasaSegura* que se estudió en secciones anteriores. El sistema vigila alarmas de incendio, humo, robo, inundación y temperatura (por ejemplo, si el sistema de calefacción falla cuando el propietario está fuera durante el invierno), genera un sonido de alarma y hace una llamada al servicio de vigilancia con la generación de un mensaje de voz sintetizada.

Recuerde que el LDP *no* es un lenguaje de programación. Se adapta según se requiera sin preocuparse por errores de sintaxis. Sin embargo, el diseño para el software de vigilancia ten-

dría que revisarse (¿ve algún problema el lector?) y mejorarse más para poder escribir código. El siguiente LDP[8] muestra la elaboración del diseño del procedimiento para una versión temprana de un componente de administración de una alarma.

```
componente AdministrarAlarma;
El objetivo de este componente es administrar los interruptores del panel de control y las entradas desde los
sensores por tipo, y actuar en cualquier condición de alarma que se encuentre.
    Establecer valores de inicio para EstadodelSistema (valor devuelto), todos los grupos de datos inician
    todos los puertos del sistema y reinician todo el hardware
    verificar los InterruptoresdelPaneldeControl (ipc)
        si ipc = "probar" entonces invocar que alarma se coloque en "on"
        si ipc = "AlarmaApagada" entonces invocar que alarma se coloque en "off"
        si ipc = "NuevoValorAsignado" entonces invocar EntradadelTeclado
        si ipc = "AlarmaContraRoboApagada" invocar DesactivarAlarma;
        •
        •
        •
        prestablecido para ipc = ninguno
    reiniciar todos los ValordeSeñal e interruptores
    hacer para todos los sensores
        invocar el procedimiento VerificarSensor con la devolución de ValordeSeñal
        si ValordeSeñal > asignar [TipodeAlarma]
            entonces MensajeTelefónico = mensaje [TipodeAlarma]
                iniciar SonidodeAlarma en "on" para SegundosdeTiempo de alarma
                iniciar estado del sistema = "CondicióndeAlarma"
                ComienzaPar
                    invocar procedimiento de alarma con "on", SegundosdeTiempo de alarma;
                    invocar procedimiento de teléfono con TipodeAlarma, NúmeroTelefónico
                TerminaPar
            en otro caso salta
        TerminaSi
    TerminaHacerPara
termina AdministrarAlarma
```

Observe que el diseñador del componente **AdministrarAlarma** usó la construcción **ComienzaPar... TerminaPar**, que especifica un bloque paralelo. Todas las tareas especificadas dentro de **ComienzaPar** se ejecutan en paralelo. En este caso, no se consideran los detalles de la implantación.

10.6 Desarrollo basado en componentes

En el contexto de la ingeniería de software, la reutilización es una idea tanto antigua como nueva. Los programadores han reutilizado ideas, abstracciones y procesos desde los primeros días de la computación, pero el enfoque inicial que aplicaban era *ad hoc*. Hoy en día, los sistemas basados en computadoras, complejos y de alta calidad, deben elaborarse en plazos muy cortos y demandan un enfoque más organizado de la reutilización.

8 El nivel de detalle representado por el LDP se define localmente. Algunas personas prefieren la descripción más natural orientada al lenguaje, mientras que otras escogen algo más cercano al código.

La *ingeniería de software basada en componentes* (ISBC) es un proceso que pone el énfasis en el diseño y construcción de sistemas basados en computadora que emplean "componentes" reutilizables de software. Clements [Cle95] describe la ISBC del modo siguiente:

> La ISBC adopta la filosofía "compra, no hagas" propuesta por Fred Brooks y otros. Del mismo modo que las subrutinas de los primeros tiempos liberaron al programador de pensar en los detalles, la ISBC traslada el énfasis de la programación a la combinación de sistemas de software. La implantación ha dado paso a la integración como el aspecto importante.

Pero surgen varias preguntas. ¿Es posible construir sistemas complejos con el ensamble de componentes reutilizables de software procedentes de un catálogo? ¿Se logra esto en forma eficaz en cuanto a costo y tiempo? ¿Pueden establecerse incentivos apropiados que estimulen a los ingenieros de software a reutilizar, más que a reinventar? ¿La dirección está dispuesta a incurrir en los gastos adicionales asociados a la creación de componentes de software reutilizables? ¿Es posible crear la biblioteca de componentes necesarios para conseguir la reutilización en forma tal que sea accesible a quienes los necesitan? ¿Quienes necesitan los componentes que ya existen pueden encontrarlos?

Cada vez más, la respuesta a estas preguntas es "sí". En el resto de esta sección se estudian algunos aspectos que deben tomarse en cuenta para hacer que la ISBC tenga éxito en una organización de ingeniería de software.

Cita:

"La ingeniería del dominio trata de encontrar aspectos en común en los sistemas a fin de identificar los componentes aplicables a muchos de ellos…"

Paul Clements

10.6.1 Ingeniería del dominio

El objetivo de la *ingeniería del dominio* es identificar, construir, catalogar y diseminar un conjunto de componentes de software que sean aplicables al software existente y al del futuro en un dominio particular de aplicaciones.[9] La meta general es establecer mecanismos que permitan que los ingenieros de software compartan dichos componentes —los reutilicen— cuando trabajen en sistemas nuevos o ya existentes. La ingeniería del dominio incluye tres actividades principales: análisis, construcción y diseminación.

CONSEJO

El proceso de análisis que se estudia en esta sección se centra en los componentes reutilizables. Sin embargo, el análisis de sistemas con componentes comerciales separados (CCS) completos (por ejemplo, aplicaciones de comercio electrónico y automatización de la fuerza de ventas) también forma parte del análisis del dominio.

El enfoque general del *análisis del dominio* se caracteriza con frecuencia dentro del contexto de la ingeniería de software orientada a objetos. Los pasos de este proceso se definen como sigue:

1. Definir el dominio que se va a investigar.
2. Clasificar los aspectos extraídos del dominio.
3. Reunir una muestra representativa de aplicaciones en el dominio.
4. Analizar cada aplicación en la muestra y definir clases de análisis.
5. Desarrollar un modelo de los requerimientos para las clases.

Es importante observar que el análisis del dominio es aplicable a cualquier paradigma de la ingeniería de software y se aplica tanto al desarrollo convencional como al orientado a objetos.

10.6.2 Calificación, adaptación y combinación de los componentes

La ingeniería del dominio genera la biblioteca de componentes reutilizables que se requieren para la práctica de la ingeniería de software basada en componentes. Algunos de estos componentes reutilizables se desarrollan en la propia empresa, mientras que otros provienen de aplicaciones existentes y otros más se adquieren de terceras personas.

Desafortunadamente, la existencia de componentes reutilizables no garantiza que éstos se integren con facilidad o eficacia en la arquitectura escogida para una nueva aplicación. Es por

9 En el capítulo 9 se mencionan géneros arquitectónicos que identifican dominios específicos de aplicación.

esta razón que cuando se propone el empleo de un componente se aplica una secuencia de acciones de desarrollo basadas en componentes.

Calificación de componentes. La calificación de componentes garantiza que un componente candidato ejecute la función requerida, "encaje" en forma adecuada en el estilo arquitectónico (véase el capítulo 9) especificado para el sistema y tenga las características de calidad (rendimiento, confiabilidad, usabilidad) que se requieren para la aplicación.

La descripción de la interfaz proporciona información útil sobre la operación y uso de un componente de software, pero no toda la que se requiere para determinar si el componente propuesto es en verdad capaz de reutilizarse con eficacia en una nueva aplicación. Entre los muchos factores que se consideran durante la calificación de un componente se encuentran los que se mencionan a continuación [Bro96]:

¿Cuáles son los factores que se consideran durante la calificación de componentes?

- Aplicación de programación de la interfaz (API).
- Herramientas de desarrollo e integración requeridas por el componente.
- Requerimientos durante la puesta en marcha, incluidos el uso de recursos (como la memoria de almacenamiento), sincronía o velocidad y protocolo de redes.
- Requerimientos de servicio, incluidos interfaces del sistema operativo y apoyo de otros componentes.
- Características de seguridad, incluidos controles de acceso y protocolo de autentificación.
- Suposiciones incrustadas en el diseño, incluido el empleo de algoritmos, numéricos o no, específicos.
- Manejo de excepciones.

Cada uno de estos factores es relativamente fácil de evaluar cuando se proponen componentes reutilizables desarrollados en la propia empresa. Si durante el desarrollo de un componente se aplican buenas prácticas de ingeniería de software, es posible responder las preguntas que están implícitas en la lista. Sin embargo, es mucho más difícil determinar los trabajos internos de los componentes comerciales separados (CCS) o de los adquiridos a terceras personas porque la única información disponible es la especificación de la interfaz en sí misma.

Adaptación de componentes. En la situación ideal, la ingeniería del dominio crea una biblioteca de componentes que se integra con facilidad en la arquitectura de una aplicación. La implicación de una "integración fácil" es que 1) se han implementado métodos consistentes de administración de recursos para todos los componentes que hay en la biblioteca; 2) para todos los componentes existen actividades comunes, tales como administración de datos, y 3) se han implementado de manera consistente interfaces dentro de la arquitectura y con el ambiente externo.

CONSEJO

Además de evaluar si se justifica el costo de adaptación para la reutilización, también debe analizarse si es posible lograr la funcionalidad que se requiere de manera rentable.

En realidad, incluso si un componente ha sido calificado para el uso dentro de una aplicación de arquitectura, surgen conflictos en una o más de las áreas mencionadas. Para evitar estos conflictos, en ocasiones se emplea una técnica de adaptación llamada *envoltura de componentes* [Bro96]. Cuando un equipo de software tiene acceso total al diseño y código interno de un componente (que con frecuencia no es el caso a menos que se utilicen componentes CCS de fuente abierta), se aplica la *envoltura de caja blanca*. Como su contraparte en las pruebas del software (véase el capítulo 18), la envoltura de caja blanca estudia los detalles del procesamiento interno del componente y hace modificaciones en el nivel de código para eliminar cualquier conflicto. Se aplica la *envoltura de caja gris* cuando la biblioteca de componentes proporciona un lenguaje de extensión del componente o API que permite que los conflictos se eliminen o se anulen. La *envoltura de caja negra* requiere la introducción de procesamiento previo y posterior en la interfaz del componente para eliminar o anular los conflictos. Debe determinarse si se justifica el

esfuerzo para envolver de manera adecuada un componente o si en vez de ello debe hacerse la ingeniería de un componente especializado (diseñado para eliminar los conflictos que surjan).

Combinación de componentes. La tarea de combinar componentes ensambla componentes calificados, adaptados y con la ingeniería necesaria para incluirse en la arquitectura establecida para una aplicación. Para lograr esto, debe establecerse una infraestructura que ligue los componentes en un sistema operativo. La infraestructura (por lo general una biblioteca de componentes especializados) provee un modelo para la coordinación de componentes y servicios específicos que permite que éstos se coordinen entre sí y ejecuten tareas comunes.

Como el efecto potencial de la reutilización y la ISBC en la industria del software es enorme, varias compañías y consorcios han propuesto estándares para el software de los componentes.[10]

WebRef

En la dirección **ww.omg.org** se encuentra la información más reciente sobre la arquitectura de transacciones de objetos comunes solicitados (GAO/ATOCS).

GAO/ATOCS. El Grupo de Administración de Objetos publicó un documento titulado *arquitectura de intercambio de objetos comunes solicitados* (GAO/ATOCS). Un intercambio de objetos solicitados (IOS) proporciona varios servicios que permiten que los componentes reutilizables (objetos) se comuniquen con otros componentes, sin importar su ubicación dentro de un sistema.

WebRef

La información más reciente sobre MCO y .NET se obtiene en la dirección **www.microsoft.com/COM** y **msdn2.microsoft.com/en-us/netframeworkdefault.aspx.**

MCO de Microsoft y .NET. Microsoft desarrolló un *modelo de componentes de objetos* (MCO) que proporciona las especificaciones para el uso de componentes producidos por varios vendedores dentro de un campo de aplicación y que son compatibles con el sistema operativo Windows. Desde el punto de vista de la aplicación, "la atención no se pone en la forma de implementar [los objetos MCO], sino en el hecho de que el objeto tiene una interfaz que es compatible con el sistema y que usa el sistema de componentes para comunicarse con otros objetos MCO" [Har98a]. La estructura .NET de Microsoft incluye MCO y da una biblioteca de clases reutilizables que cubre una amplia variedad de dominios de aplicación.

WebRef

La información más reciente sobre JavaBeans se obtiene en **java.sun.com/products/javabeans/docs/**.

Componentes JavaBeans de Sun. El sistema de componentes JavaBeans es una infraestructura de la ISBC, portátil e independiente de la plataforma, desarrollada con el uso del lenguaje de programación Java. El sistema de componentes JavaBeans incluye un conjunto de herramientas llamado *Kit de Desarrollo Bean* (KDB) que permite que los desarrolladores 1) analicen la forma en la que funcionan los Beans (componentes) existentes; 2) personalicen su comportamiento y apariencia; 3) establezcan mecanismos para la coordinación y comunicación; 4) desarrollen Beans especiales para el uso en una aplicación específica, y 5) prueben y evalúen el comportamiento Bean.

Ninguno de estos estándares domina la industria. Aunque muchos desarrolladores se han estandarizado teniendo como modelo uno, es probable que las organizaciones grandes de software elijan el empleo de un estándar con base en las categorías y plataformas de aplicación que se elijan.

10.6.3 Análisis y diseño para la reutilización

Aunque la ISBC promueve el uso de componentes de software ya existentes, hay veces en las que deben desarrollarse otros nuevos para integrarlos con CCS disponibles y con otros propios. Como estos nuevos componentes se vuelven miembros de la biblioteca de la empresa de componentes reutilizables, debe hacerse la ingeniería para su reutilización.

Los conceptos de diseño, tales como abstracción, ocultamiento, independencia de funciones, refinamiento y programación estructurada, así como los métodos orientados a objeto, pruebas,

10 Greg Olsen [Ols06] hace un análisis excelente de los esfuerzos pasados y presentes de la industria para hacer de la ISBC una realidad.

aseguramiento de la calidad del software (ACS) y métodos de comprobación de la corrección (véase el capítulo 21), contribuyen a la creación de componentes de software reutilizables. En esta subsección se estudian aspectos específicos de la reutilización que son complementarios de las prácticas sólidas de la ingeniería de software.

El modelo de requerimientos se analiza para determinar qué elementos apuntan hacia componentes reutilizables ya existentes. Los elementos del modelo de los requerimientos se comparan con los componentes reutilizables en un proceso que en ocasiones se conoce como "ajuste de especificaciones" [Bel95]. Si el ajuste de especificaciones señala hacia un componente que ya existe y que se ajusta a las necesidades de la aplicación en cuestión, se extrae el componente de la biblioteca de reutilización (repositorio) y se emplea en el diseño de un nuevo sistema. Si no es posible encontrar el componente (por ejemplo, no hay coincidencia), se crea uno nuevo. Es en este punto —cuando se comienza a crear el componente nuevo— donde debe considerarse el *diseño para la reutilización* (DPR).

Como ya se dijo, el DPR requiere que se apliquen conceptos y principios sólidos del diseño de software (véase el capítulo 8). Pero las características del dominio de la aplicación también deben tomarse en cuenta. Binder [Bin93] sugiere varios aspectos clave[11] que constituyen la base del diseño para la reutilización:

El DPR es muy difícil si hay que hacer las interfaces o integrar los componentes con sistemas heredados o con sistemas múltiples cuya arquitectura y protocolos de interfaz son incongruentes.

Datos estándar. El dominio de la aplicación debe investigarse y tienen que identificarse las estructuras de datos globales estándar (como las de archivos o una base de datos completa). Todos los componentes del diseño se caracterizan para hacer uso de estas estructuras de datos estándar.

Protocolos de interfaz estándar. Deben establecerse tres niveles de protocolos de interfaz: la naturaleza de las interfaces intramodulares, el diseño de las interfaces externas técnicas (no humanas) y la interfaz humano-computadora.

Plantillas de programa. Se elige un estilo arquitectónico (véase el capítulo 9) que sirve como plantilla para el diseño de la arquitectura del nuevo software.

Una vez establecidos los datos estándar, interfaces y plantillas del programa, se tiene una estructura en la cual crear el diseño. Los componentes nuevos que conforman ésta tienen una probabilidad mayor de tener un uso posterior.

10.6.4 Clasificación y recuperación de componentes

Considere una biblioteca universitaria grande. Se encuentran disponibles cientos de miles de libros, revistas y otras fuentes de información. Pero para acceder a dichas fuentes debe desarrollarse un esquema de clasificación. Para navegar en este enorme volumen de información, los bibliotecarios han definido un esquema de clasificación que incluye un código de la Biblioteca del Congreso, palabras clave, nombres de los autores y otras entradas para el índice. Todos ellos permiten que el usuario encuentre fácil y rápidamente la fuente que necesita.

Ahora, considere un repositorio grande de componentes en el que se encuentran decenas de miles de componentes de software reutilizables. Pero, ¿cómo se encuentra el que se necesita? Al tratar de responder esta pregunta surge otra: ¿cómo describimos los componentes de software en términos no ambiguos y clasificables? Éstas son preguntas difíciles para las que todavía no hay una respuesta definitiva. En esta sección se estudian las direcciones actuales que permitirán que los ingenieros de software naveguen a través de las bibliotecas de la reutilización.

Un componente de software reutilizable puede describirse de muchas formas, pero la descripción ideal incluye lo que Tracz [Tra95] llama *modelo 3C*: concepto, contenido y contexto. El *concepto* de un componente de software es "la descripción de lo que hace el componente"

11 En general, las preparaciones del DPR deben tomarse como parte de la ingeniería del dominio.

[Whi95]. Se describe por completo la interfaz al componente y se identifica la semántica, representada en el contexto de las condiciones previas y posteriores. El concepto debe comunicar el objetivo del componente. El *contenido* de un componente describe cómo se lleva a cabo el concepto. En esencia, el contenido es información que queda oculta a los usuarios casuales y que sólo necesitan conocer aquellos que pretenden modificar o probar el componente. El *contexto* coloca un componente de software reutilizable en su dominio de aplicabilidad. Es decir, al especificar las características conceptuales, operativas y de implantación, el contexto permite que un ingeniero de software encuentre el componente apropiado para que se cumplan los requerimientos de la aplicación.

Concepto, contenido y contexto deben traducirse en un esquema concreto de especificación para que tengan uso práctico. Se han escrito decenas de textos y artículos sobre esquemas de clasificación para componentes de software reutilizables (por ejemplo, consulte en [Cec06] el panorama de las tendencias actuales).

La clasificación permite encontrar y recuperar componentes que son candidatos a la reutilización, pero debe existir un ambiente propicio para integrarlos con eficacia. Éste tiene las características siguientes:

? ¿Cuáles son las características clave de un ambiente de reutilización de componentes?

- Una base de datos capaz de almacenar componentes de software y la información de clasificación necesaria para recuperarlos.
- Un sistema de administración de la biblioteca que dé acceso a la base de datos.
- Un sistema de recuperación de componentes de software (por ejemplo, un agente de solicitud de objetos) que permita que la aplicación de un cliente recupere componentes y servicios del servidor de la biblioteca.
- Herramientas de ISBC que apoyen la integración de componentes reutilizados en un diseño o implantación nuevos.

Cada una de estas funciones interactúa con los confines de una biblioteca de reutilización o se halla incrustada en ella.

La *biblioteca de reutilización* es un elemento de un repositorio mayor de software (véase el capítulo 22) y brinda herramientas para el almacenamiento de componentes de software, así

HERRAMIENTAS DE SOFTWARE

ISBC

Objetivo: Ayudar a modelar, diseñar, revisar e integrar los componentes de software como parte de un sistema mayor.

Mecánica: Las mecánicas de las herramientas varían. En general, las herramientas de ISBC ayudan en una o más de las siguientes capacidades: especificar y modelar la arquitectura del software, investigar y seleccionar los componentes de software disponibles; integrar los componentes.

Herramientas representativas[12]

ComponentSource (**www.componentsource.com**) proporciona una variedad amplia de componentes (y herramientas) de software CCS apoyada dentro de muchos estándares de componentes distintos.

Component Manager, desarrollado por Flashline (**www.flashline.com**), "es una aplicación que permite, promueve y mide la reutilización de componentes de software".

Select Component Factory, desarrollado por Select Business Solutions (**www.selectbs.com**), "es un conjunto integrado de productos para diseñar software, revisar el diseño, administrar servicios o componentes, manejar requerimientos y generar código".

Software Through Pictures-ACD, distribuido por Aonix (**www.aonix.com**), permite el modelado exhaustivo con el uso de UML para el modelo OMG orientado a la arquitectura, un enfoque de la ISBC abierto y neutral para el vendedor.

12 Las herramientas mencionadas aquí no son obligatorias, sólo son una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

como una amplia variedad de productos finales reutilizables (especificaciones, diseños, patrones, estructuras, fragmentos de código, casos de prueba, guías del usuario, etc.). La biblioteca incluye una base de datos y las herramientas necesarias para hacer consultas y recuperar los componentes. El esquema de clasificación de componentes es la base de las consultas a la biblioteca.

WebRef

En la dirección **www.cbd-hq.com/** se encuentra un conjunto exhaustivo de recursos de la ISBC.

Es frecuente que las consultas se clasifiquen con el empleo del elemento de contexto del modelo 3C mencionado. Si una consulta inicial da como resultado una lista grande de componentes candidatos, se afina la consulta para depurar la lista. Después se extrae la información de concepto y contenido (una vez encontrados los componentes candidatos) que ayuden a seleccionar el componente apropiado.

10.7 Resumen

El proceso de diseño en el nivel de componentes incluye una secuencia de actividades que reduce poco a poco el nivel de abstracción con el que se representa el software. El diseño en el nivel de componentes ilustra en definitiva al software en un nivel de abstracción cercano al código.

Es posible adoptar tres puntos de vista diferentes en el nivel de diseño, en función de la naturaleza del software que se va a desarrollar. El enfoque orientado a objetos se centra en la elaboración de clases de diseño que provienen tanto del dominio del problema como de la infraestructura. El punto de vista tradicional mejora tres tipos diferentes de componentes o módulos: los de control, los del dominio del problema y los de la infraestructura. En ambos casos se aplican los principios y conceptos básicos del diseño que llevan a un software de alta calidad. Cuando se considera al diseño en el nivel de componentes desde un punto de vista del proceso, se llega a componentes de software reutilizables y a patrones de diseño que son elementos cruciales de la ingeniería de software basada en componentes.

Conforme se elaboran las clases, varios principios y conceptos importantes guían al diseñador. Las ideas agrupadas en el Principio Abierto-Cerrado y en el Principio de Inversión de la Dependencia, así como conceptos tales como el acoplamiento y la cohesión, guían al ingeniero de software en la construcción de componentes de software susceptibles de someterse a prueba, implantarse y recibir mantenimiento. Para hacer el diseño en el nivel de componentes en este contexto, se elaboran las clases especificando detalles de la mensajería, identificando las interfaces apropiadas, elaborando atributos y definiendo estructuras de datos que las implementen, describiendo el flujo del procesamiento dentro de cada operación y representando el comportamiento en el nivel de clase o componente. Una actividad esencial en cada caso es el diseño iterativo (rediseñar).

El diseño tradicional en el nivel de componentes requiere la representación de estructuras de datos, interfaces y algoritmos para un módulo de programa con detalle suficiente para guiar la generación del código fuente del lenguaje de programación. A fin de lograr esto, el diseñador usa una de varias notaciones de diseño que representan los detalles en el nivel de componente en un formato gráfico, tabular o basado en texto.

El diseño en el nivel de componentes para *webapps* considera tanto el contenido como la funcionalidad tal como es entregada por un sistema basado en web. El diseño del contenido en el nivel de componentes se centra en objetos de contenido y en la manera en la que se empacan para su presentación en una *webapp* al usuario final. El diseño funcional para las *webapps* se centra en funciones de procesamiento que manipulan contenido, realizan cálculos, consultan y acceden a una base de datos y establecen interfaces con otros sistemas. Se aplican todos los principios y lineamientos del diseño en el nivel de componentes.

La programación estructurada es una filosofía de diseño orientado al procedimiento que limita el número y tipo de construcciones lógicas usadas para representar los detalles algorítmi-

cos. El objetivo de la programación estructurada es auxiliar al diseñador en la definición de algoritmos que sean menos complejos y por ello más fáciles de leer, probar y mantener.

La ingeniería de software basada en componentes identifica, construye, cataloga y disemina un conjunto de componentes de software en un dominio particular de aplicación. Después, estos componentes se califican, adaptan e integran para usarlos en un sistema nuevo. Los componentes reutilizables deben diseñarse dentro de un ambiente que establezca para cada dominio de aplicación estructuras de datos estándar, protocolos de interfaz y arquitecturas de programa.

Problemas y puntos por evaluar

10.1. En ocasiones resulta difícil definir el término *componente*. Primero dé una definición general y luego otras más explícitas para el software orientado a objetos y para el tradicional. Por último, elija tres lenguajes de programación con los que esté familiarizado e ilustre la manera en la que cada uno define un componente.

10.2. ¿Por qué son necesarios los componentes de control en el software tradicional y por qué en general no se requieren en el orientado a objetos?

10.3. Describa con sus propias palabras el PAC. ¿Por qué es importante crear abstracciones que sirvan como interfaz entre los componentes?

10.4. Describa el PID con sus propias palabras. ¿Qué pasaría si un diseñador dependiera demasiado de las concreciones?

10.5. Seleccione tres componentes que haya desarrollado recientemente y evalúe los tipos de cohesión que presente cada uno. Si tuviera que definir el beneficio principal de la cohesión, ¿cuál sería?

10.6. Elija tres componentes que haya elaborado hace poco y evalúe los tipos de acoplamiento que tenga cada uno. Si definiera el principal beneficio del poco acoplamiento, ¿qué diría?

10.7. ¿Es razonable decir que los componentes del dominio del problema nunca deben tener acoplamiento externo? Si está de acuerdo, ¿qué tipos de componente tendrían acoplamiento externo?

10.8. Desarrolle 1) una clase de diseño elaborada, 2) descripciones de interfaz, 3) un diagrama de actividades para una de las operaciones dentro de la clase de diseño y 4) un diagrama de estado detallado para una de las clases de *CasaSegura* que se estudiaron en los capítulos anteriores.

10.9. ¿Son lo mismo el refinamiento *stepwise* y el rediseño? Si no es así, ¿en qué difieren?

10.10. ¿Qué es un componente de *webapp*?

10.11. Seleccione una parte pequeña de un programa existente (de 50 a 75 líneas de código). Separe las construcciones de programación estructurada con cuadros que dibuje alrededor de ellas en el código fuente. ¿El extracto de programa tiene construcciones que violan la filosofía de la programación estructurada? Si es así, rediseñe el código para que se apegue a las construcciones de programación estructurada. Si no es así, ¿qué observa en los cuadros que dibujó?

10.12. Todos los lenguajes modernos de programación implementan las construcciones de programación estructurada. Dé ejemplos de tres lenguajes de programación.

10.13. Seleccione un componente codificado pequeño y represéntelo con 1) un diagrama de actividades, 2) un diagrama de flujo, 3) una tabla de decisión y 4) LDP.

10.14. ¿Por qué es importante la "lotificación" en el proceso de revisión del diseño en el nivel de componentes?

Lecturas adicionales y fuentes de información

En los últimos años se han publicado muchos libros sobre el desarrollo basado en componentes y acerca de su reutilización. Apperly *et al.* (*Service- and Component-Based Development*, Addison-Wesley, 2003), Heineman y Councill (*Component Based Software Engineering*, Addison-Wesley, 2001), Brown (*Large Scale Compo-*

nent-Based Development, Prentice-Hall, 2000), Allen (*Realizing e-Business with Components*, Addison-Wesley, 2000), Herzum y Sims (*Business Component Factory*, Wiley, 1999), Allen, Frost y Yourdon (*Component-Based Development for Enterprise Systems: Applying the Select Perspective*, Cambridge University Press, 1998) cubren todos los aspectos importantes del proceso de la ISBC. Cheesman y Daniels (*UML Components*, Addison-Wesley, 2000) estudian la ISBC con énfasis en el UML.

Gao *et al.* (*Testing and Quality Assurance for Component-Based Software*, Artech House, 2006) y Gross (*Component-Based Software Testing with UML*, Springer, 2005) estudian las pruebas y el aseguramiento de la calidad en los sistemas basados en componentes.

En los años recientes han sido publicadas decenas de libros que describen estándares de la industria basados en componentes. Estas obras se dirigen a los trabajos internos de los estándares, pero también consideran muchos temas importantes de la ISBC.

El trabajo de Linger, Mills y Witt (*Structured Programming-Theory and Practice*, Addison-Wesley, 1979) permanece como el análisis definitivo del tema. El texto contiene un buen LDP y también estudios detallados de las ramificaciones de la programación estructurada. Otros libros que se centran en el diseño orientado al procedimiento para sistemas tradicionales incluyen los de Robertson (*Simple Program Design*, 3a. ed., Course Technology, 2000), Farrell (*A Guide to Programming Logic and Design*, Course Technology, 1999), Bentley (*Programming Pearls*, 2a. ed., Addison-Wesley, 1999) y Dahl (*Structured Programming*, Academic Press, 1997).

Son relativamente pocos los libros dedicados en exclusivo al diseño en el nivel de componentes. En general, los libros de lenguajes de programación están dirigidos al diseño del procedimiento con cierto detalle, pero siempre en el contexto del lenguaje que trata el libro. Son cientos los títulos disponibles.

En internet hay una amplia variedad de fuentes de información sobre diseño en el nivel de componentes. En el sitio web del libro, **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, existe una lista actualizada de referencias en la red mundial que son relevantes para el diseño en el nivel de componentes.

CAPÍTULO

11

DISEÑO DE LA INTERFAZ DE USUARIO

CONCEPTOS CLAVE

accesibilidad 283
análisis de la interfaz 272
  consistente 268
  diseño 278
  modelos 264
análisis de la tarea 273
análisis del usuario 272
control 266
diseño de una interfaz para webapps 284
elaboración de la tarea 275
evaluación del diseño. 290
herramientas de ayuda 282
internacionalización 283
leyendas de los comandos . . 283
manejo de errores 282
memorización 267
principios y lineamientos . . . 285
proceso 271
reglas doradas 266
tiempo de respuesta 281
usabilidad. 269

Vivimos en un mundo de productos de alta tecnología, y virtualmente todos ellos —electrónica para el consumidor, equipo industrial, sistemas corporativos, sistemas militares, software de computadoras personales y *webapps*— requieren interacción humana. Si un producto ha de alcanzar el éxito, debe tener buena *usabilidad*: medición cualitativa de la facilidad y eficiencia con la que un humano emplea las funciones y características que ofrece el producto de alta tecnología.

La usabilidad importa, ya sea que una interfaz haya sido diseñada para un reproductor de música digital o para el sistema de control de armas de un avión de combate. Si los mecanismos de la interfaz están bien diseñados, el usuario se desliza por la interacción a un ritmo suave que hace que el trabajo se realice sin esfuerzo. Pero si la interfaz fue mal concebida, el usuario avanza y retrocede, y el resultado final es frustración y poca eficiencia en el trabajo.

Durante las tres primeras décadas de la era de la computación, la usabilidad no era la preocupación dominante de quienes elaboraban software. En su libro clásico sobre diseño, Donald Norman [Nor88] afirmaba que ya era tiempo de cambiar de actitud:

> Para producir tecnología que se adapte a los seres humanos, es necesario estudiar a éstos. Pero en la actualidad tendemos a estudiar sólo a la primera. El resultado es que se exige a las personas que se adapten a la tecnología. Es tiempo de que esta tendencia se revierta, es el momento de que la tecnología se adapte a las personas.

A medida que los tecnólogos estudiaban la interacción humana, surgieron dos aspectos dominantes. El primero fue que se identificaron *reglas doradas* (que se estudian en la sección 11.1). Éstas se aplican a toda interacción humana con productos de la tecnología. El segundo fue que se definió un conjunto de *mecanismos de interacción* que permitieron a los diseñadores de software construir sistemas que implantaban en forma correcta las reglas doradas. Esos meca-

UNA MIRADA RÁPIDA

¿Qué es? El diseño de la interfaz de usuario crea un medio eficaz de comunicación entre los seres humanos y la computadora. Siguiendo un conjunto de principios de diseño de la interfaz, el diseño identifica los objetos y acciones de ésta y luego crea una plantilla de pantalla que constituye la base del prototipo de la interfaz de usuario.

¿Quién lo hace? Un ingeniero de software diseña la interfaz de usuario con la aplicación de un proceso iterativo que sigue principios de diseño predefinidos.

¿Por qué es importante? Si el software es difícil de usar, fuerza al usuario a cometer errores, o si frustra sus esfuerzos para alcanzar las metas, entonces no le gustará, sin que importe el poder computacional que tenga, el contenido que entregue o las funciones que ofrezca. La interfaz tiene que estar bien hecha porque moldea la percepción que el usuario tiene del software.

¿Cuáles son los pasos? El diseño de la interfaz de usuario comienza con la identificación de los requerimientos del usuario, la tarea y el ambiente. Una vez identificadas las tareas del usuario, se crean y analizan los escenarios para éste y se define un conjunto de objetos y acciones de la interfaz. Esto forma la base para crear una plantilla de la pantalla que ilustra el diseño gráfico y la colocación de los iconos, la definición de textos descriptivos, la especificación y títulos de las ventanas, y la especificación de aspectos mayores y menores del menú. Con el empleo de herramientas, se hace el prototipo, se implementa en definitiva el modelo del diseño y se evalúa la calidad del resultado.

¿Cuál es el producto final? Se crean los escenarios del usuario y se generan los formatos de la pantalla. Se desarrolla un prototipo de la interfaz y se modifica de manera iterativa.

¿Cómo me aseguro de que lo hice bien? Los usuarios "prueban" un prototipo de la interfaz y la retroalimentación de esta prueba se utiliza para la siguiente modificación iterativa del prototipo.

nismos de interacción, llamados de manera colectiva *interfaz gráfica del usuario* (GUI), eliminaron algunos de los problemas más notables asociados con las interfaces humanas. Pero aun en un "mundo Windows", todos hemos encontrado interfaces de usuario que son difíciles de aprender y usar, que son confusas y van contra la intuición, que no perdonan y, en muchos casos, resultan frustrantes por completo. No obstante, alguien dedicó tiempo y energía a la elaboración de estas interfaces, y no es probable que su constructor haya creado dichos problemas con toda intención.

11.1 Las reglas doradas

En su libro sobre el diseño de la interfaz, Theo Mandel [Man97] acuñó tres *reglas doradas*:

1. Dejar el control al usuario.
2. Reducir la carga de memoria del usuario.
3. Hacer que la interfaz sea consistente.

En realidad, estas reglas doradas constituyen la base de un conjunto de principios de diseño de la interfaz de usuario que guían este aspecto tan importante del diseño del software.

11.1.1 Dejar el control al usuario

Cita:

"Es mejor diseñar la experiencia del usuario que corregirla."

Jon Meads

Durante una sesión para recabar los requerimientos de un nuevo y gran sistema de información, se preguntó a una usuaria clave acerca de los atributos de la interfaz gráfica basada en ventanas.

"Lo que realmente me gustaría", respondió con solemnidad, "es un sistema que lea mi mente. Que sepa lo que quiero hacer antes de que necesite hacerlo y que sea fácil para mí obtener eso que quiero. Eso es todo, sólo eso".

Mi primera reacción fue afirmar con la cabeza y sonreír, pero me detuve un momento. No había absolutamente nada descabellado en la solicitud de la usuaria. Quería un sistema que reaccionara a sus necesidades y la ayudara para que las cosas se hicieran. Deseaba controlar la computadora, no que ésta la controlara a ella.

La mayor parte de limitaciones y restricciones que impone un diseñador pretenden simplificar el modo de interacción. Pero, ¿para quién?

Como diseñador, tal vez se sienta tentado a introducir restricciones y limitantes que simplifiquen la implantación de la interfaz. El resultado puede ser una interfaz fácil de construir, pero que sea frustrante utilizar. Mandel [Man97] define cierto número de principios de diseño que permiten que el usuario tenga el control:

Definir modos de interacción de manera que no se obligue al usuario a realizar acciones innecesarias o no deseadas. Un modo de interacción es el estado actual de la interfaz. Por ejemplo, si en el menú de un procesador de textos se selecciona *revisar ortografía*, el software pasa al modo de revisión de la ortografía. No hay razón para obligar al usuario a permanecer en este modo si acaso desea hacer una pequeña edición del texto. El usuario debe poder entrar y salir del modo con poco o ningún esfuerzo.

Dar una interacción flexible. Debido a que diferentes usuarios tienen distintas preferencias para la interacción, debe darse la posibilidad de elegir. Por ejemplo, el software debe permitir que el usuario interactúe por medio de comandos introducidos con el teclado, el ratón, una pluma digitalizadora, una pantalla sensible al tacto o un mecanismo de reconocimiento de voz. Pero no todas las acciones son accesibles a través de cualquier mecanismo de interacción. Por ejemplo, piénsese en la dificultad de usar comandos del teclado (o entradas con la voz) para hacer un dibujo complicado.

Permitir que la interacción del usuario sea interrumpible y también reversible. El usuario debe poder suspender la secuencia de su trabajo (aun cuando consista en una secuencia de acciones) para hacer otra cosa (sin perder el trabajo realizado hasta ese momento). También debe poder "deshacer" cualquier acción.

Facilitar la interacción a medida que aumenta la habilidad y permitir que aquélla se personalice. Es frecuente que los usuarios realicen la misma secuencia de interacciones en forma repetida. Es benéfico diseñar un mecanismo de "macros" que permita que un usuario avanzado personalice la interfaz para facilitar la interacción.

Cita:

"Siempre deseé que mi computadora fuera tan fácil de usar como mi teléfono. Mi deseo se ha vuelto realidad. Ya no sé cómo se utiliza mi teléfono."

Bjarne Stronstrup (iniciador de C++)

Ocultar los tecnicismos internos al usuario ocasional. La interfaz de usuario debe introducirlo al mundo virtual de la aplicación. El usuario no debe tener que preocuparse del sistema operativo, de las funciones de administración de archivos ni de ninguna otra tecnología de computación secreta. En esencia, la interfaz no debe requerir que el usuario interactúe en un nivel "interno" de la máquina (nunca debería tener que escribir comandos del sistema operativo desde una aplicación de software).

Diseñar la interacción directa con objetos que aparezcan en la pantalla. El usuario tiene la sensación de control cuando puede manipular los objetos que se necesitan a fin de ejecutar un trabajo en la misma forma en la que lo haría si el objeto fuera algo físico. Por ejemplo, una interfaz de aplicación que le permita "estirar" un objeto (modificar su tamaño) es una implementación de manipulación directa.

11.1.2 Reducir la necesidad de que el usuario memorice

Entre más cosas tenga que recordar el usuario, más fácil será que cometa errores al interactuar con el sistema. Es por esto que una interfaz de usuario bien diseñada no sobrecarga la memoria del usuario. Siempre que sea posible, el sistema debe "recordar" la información pertinente y ayudar al usuario con un escenario de interacción que lo ayude a recordar. Mandel [Man97] define los siguientes principios de diseño que permiten que una interfaz reduzca la necesidad de que el usuario memorice:

Reducir la demanda de memoria de corto plazo. Cuando los usuarios se involucran en tareas complejas, la demanda de memoria de corto plazo es significativa. La interfaz debe diseñarse para disminuir la necesidad de recordar acciones, entradas y resultados del pasado. Esto se logra dando claves visuales que permitan al usuario reconocer acciones anteriores, en lugar de que tenga que recordarlas.

Hacer que lo preestablecido sea significativo. Lo que al principio se dé por preestablecido debe tener sentido para el usuario promedio, pero éste debería poder especificar sus preferencias individuales. Sin embargo, debe disponerse de la opción de "reiniciar" para restablecer los valores originales.

Definir atajos que sean intuitivos. Cuando se utilice nemotecnia para ejecutar una función del sistema (como la secuencia Ctrl-B para invocar la función de buscar), debe estar ligada con la acción, de modo que sea fácil de recordar (por ejemplo, con la primera letra de la tarea que se va a realizar).

La distribución visual de la interfaz debe basarse en una metáfora del mundo real. Por ejemplo, un sistema de pagos debe usar una metáfora de chequera y talonario que guíe al usuario a través del proceso de pago. Esto permite que el usuario se base en claves visuales que comprende bien, en vez de tener que memorizar una secuencia críptica de interacciones.

Revelar información de manera progresiva. La interfaz debe estar organizada de manera jerárquica. Es decir, la información acerca de una tarea, objeto o comportamiento debe presen-

tarse primero en un nivel de generalización elevado. Después de que con el ratón el usuario manifieste interés, deben darse más detalles. Un ejemplo, común para muchas aplicaciones de procesamiento de textos, es la función de subrayar. La función en sí es una de varias en el menú *estilo del texto*. No obstante, no se enlista cada una de las herramientas para subrayar. El usuario debe hacer *clic* en la opción de subrayar; después se presentan todas las opciones para esta función (una raya, doble raya, línea punteada, etcétera).

CASASEGURA

Violación de la regla dorada de la interfaz de usuario

La escena: Cubículo de Vinod, cuando comienza el diseño de la interfaz de usuario.

Participantes: Vinod y Jamie, miembros del equipo de ingeniería de software de CasaSegura.

La conversación:

Jamie: He estado pensando en la interfaz de la función de vigilancia.

Vinod (sonríe): Es bueno pensar.

Jamie: Creo que tal vez podamos simplificar algunas cosas.

Vinod: ¿Qué quieres decir?

Jamie: Bueno, qué tal si eliminamos el plano de la casa. Es bueno, pero va a requerir mucho esfuerzo de desarrollo. En vez de eso, puede pedirse al usuario que especifique la cámara que quiere ver y que luego despliegue el video en una ventana.

Vinod: ¿Cómo recordaría el propietario cuántas cámaras hay y dónde están?

Jamie (algo irritado): Él es el propietario; debe saberlo.

Vinod: ¿Y si no es así?

Jamie: Debería.

Vinod: Eso no es lo que estamos discutiendo… ¿qué pasa si lo olvida?

Jamie: Bueno, podríamos darle una lista de cámaras y sus ubicaciones.

Vinod: Es posible, pero, ¿por qué debería pedir una lista?

Jamie: Bueno, damos la lista la pida o no.

Vinod: Eso está mejor. Al menos no tendrá que recordar cosas que le podemos dar.

Jamie (piensa unos instantes): Pero, ¿te gusta o no el plano de la casa?

Vinod: Mmm.

Jamie: ¿Cuál piensas que le agrade más a mercadotecnia?

Vinod: Bromeas, ¿verdad?

Jamie: No.

Vinod: Mmm… el plano… adoran las características bonitas de los productos… no les importa cuál es más fácil de elaborar.

Jamie (suspira): Bien, quizá hagamos ambos prototipos.

Vinod: Buena idea… así dejamos que el cliente decida.

11.1.3 Hacer consistente la interfaz

Cita:

"Las cosas que parezcan diferentes deben actuar en forma diferente. Las cosas que parezcan iguales deben actuar igual."

Larry Marine

La interfaz debe presentar y obtener información en forma consistente. Esto implica: 1) que toda la información se organice de acuerdo con reglas de diseño que se respeten en todas las pantallas desplegadas, 2) que los mecanismos de entrada se limiten a un conjunto pequeño usado en forma consistente en toda la aplicación, y 3) que los mecanismos para pasar de una tarea a otra se definan e implementen de modo consistente. Mandel [Man97] define varios principios de diseño que ayudan a que la interfaz tenga consistencia:

Permita que el usuario coloque la tarea en curso en un contexto significativo. Muchas interfaces implementan capas complejas de interacciones con decenas de imágenes en la pantalla. Es importante dar indicadores (títulos en las ventanas, iconos gráficos, código de colores consistente, etc.) que permitan al usuario conocer el contexto del trabajo en curso. Además, debe poder determinar de dónde viene y qué alternativas hay para hacer la transición a una nueva tarea.

Mantener la consistencia en toda la familia de aplicaciones. Todas las aplicaciones (o productos) que hay en un grupo deben implementar las mismas reglas de diseño a fin de que se mantenga la consistencia en toda la interacción.

Si los modelos interactivos anteriores han creado expectativas en el usuario, no haga cambios a menos de que haya una razón ineludible para ello. Una vez que una secuencia interactiva en particular se ha convertido en un estándar (como el uso de alt-G para guardar un archivo), el usuario la espera en toda aplicación que emplea. Un cambio (como utilizar alt-G para invocar la función de modificar la escala) generará confusión.

Los principios de diseño de la interfaz analizados en esta sección y en las anteriores dan una guía básica. En las que siguen, el lector aprenderá acerca del proceso de diseño de la interfaz en sí.

INFORMACIÓN

Usabilidad

En un artículo útil sobre usabilidad, Larry Constantine [Con95] plantea una pregunta significativa sobre el tema: "En definitiva, ¿qué quieren los usuarios?" Y responde así:

> Lo que los usuarios desean son buenas herramientas. Todos los sistemas de software, desde sistemas operativos y lenguajes hasta aplicaciones de entrada de datos y apoyo a la toma de decisiones, sólo son herramientas. Los usuarios finales esperan lo mismo que nosotros de las herramientas que usamos. Quieren sistemas fáciles de aprender y que les ayuden a hacer su trabajo. Quieren software que no los haga más lentos, que no tenga trucos o los confunda; eso no significa que sea más fácil cometer errores o más difícil terminar el trabajo.

Constantine afirma que la usabilidad no proviene de la estética, de mecanismos de interacción avanzados o de interfaces inteligentes. En vez de eso, se obtiene cuando la arquitectura de la interfaz se ajusta a las necesidades de las personas que la emplearán.

Es ilusorio llegar a una definición formal de usabilidad. Donahue *et al*. [Don99] la definen de la manera siguiente: "La usabilidad es una medida de cuán bien un sistema de cómputo […] facilita el aprendizaje, ayuda a quienes lo emplean a recordar lo aprendido, reduce la probabilidad de cometer errores, les permite ser eficientes y los deja satisfechos con el sistema."

La única forma de determinar si existe "usabilidad" en un sistema que se construye es evaluarla o probarla. Los usuarios interactúan con el sistema y responden las preguntas siguientes [Con95]:

- ¿El sistema es utilizable sin ayuda o enseñanza continua?
- ¿Las reglas de interacción ayudan a un usuario preparado a trabajar con eficiencia?
- ¿Los mecanismos de interacción se hacen más flexibles a medida que los usuarios conocen más?
- ¿Se ha adaptado el sistema al ambiente físico y social en el que se usará?
- ¿El usuario está al tanto del estado del sistema? ¿Sabe en todo momento dónde está?
- ¿La interfaz está estructurada de manera lógica y consistente?
- ¿Los mecanismos, iconos y procedimientos de interacción son consistentes en toda la interfaz?
- ¿La interacción prevé errores y ayuda al usuario a corregirlos?
- ¿La interfaz es tolerante a los errores que se cometen?
- ¿Es sencilla la interacción?

Si cada una de estas preguntas obtiene un "sí" como respuesta, es probable que se haya logrado la usabilidad.

Entre los muchos beneficios mensurables que se obtienen de un sistema usable se encuentran los siguientes [Don99]: mayores ventas y más satisfacción del consumidor, ventaja competitiva, mejores evaluaciones en los medios, recomendaciones de boca en boca, menores costos de apoyo, más productividad del usuario final, menos costos de capacitación y documentación, y disminución de la probabilidad de litigios por parte de clientes insatisfechos.

11.2 ANÁLISIS Y DISEÑO DE LA INTERFAZ DE USUARIO

WebRef

En la dirección **www.useit.com** se encuentra una excelente fuente de información acerca del diseño de la interfaz de usuario.

El proceso general de análisis y diseño de la interfaz de usuario comienza con la creación de diferentes modelos del funcionamiento del sistema (según se percibe desde fuera). Se empieza delineando las tareas orientadas al usuario —y a la computadora— que se requieren a fin de obtener el funcionamiento del sistema, para luego considerar los aspectos que se aplican a todos los diseños de interfaz. Se emplean herramientas para hacer prototipos e implementar el modelo del diseño, y los usuarios finales evalúan la calidad.

11.2.1 Análisis y modelos del diseño de la interfaz

Cuando se analiza y diseña la interfaz de usuario, entran en juego cuatro diferentes modelos. Un ingeniero (o el encargado del software) establece un *modelo de usuario*, el ingeniero de soft-

Cita:

"Si hay un 'truco' en ella, la interfaz de usuario está acabada."

Douglas Anderson

ware crea un *modelo del diseño*, el usuario final desarrolla una imagen mental que frecuentemente se nombra *modelo mental* o *percepción del sistema*, y los implementadores del sistema crean un *modelo de implementación*. Desafortunadamente, cada uno de estos modelos difiere en forma significativa. El papel del diseñador de la interfaz es conciliar estas diferencias y obtener una representación consistente de la interfaz.

El modelo del usuario establece el perfil de los usuarios finales del sistema. En su columna introductoria sobre el "diseño centrado en el usuario", Jeff Patton [Pat07] afirma lo siguiente:

> La verdad es que los diseñadores y desarrolladores —incluido yo— piensan con frecuencia en los usuarios finales. Sin embargo, en ausencia de un modelo mental fuerte de usuarios específicos, los sustituimos con nosotros. Esto no es centrarse en el usuario, sino en uno mismo.

Para construir una interfaz de usuario eficaz, "todo diseño debe comenzar con la comprensión de los usuarios que se busca, lo que incluye los perfiles de edad, género, condiciones físicas, educación, antecedentes culturales o étnicos, motivación, metas y personalidad" [Shn04]. Además, los usuarios se clasifican como:

CONSEJO

Incluso un usuario principiante quiere atajos; aun los usuarios frecuentes conocedores en ocasiones necesitan ser guiados. Hay que darles lo que necesitan.

Principiantes. Sin conocimiento sintáctico[1] del sistema y poco conocimiento semántico[2] de la aplicación o uso de la computadora en general.

Usuarios intermitentes que saben. Con conocimiento semántico razonable de la aplicación, pero relativamente poco recuerdo de la información sintáctica necesaria para usar la interfaz.

Usuarios frecuentes conocedores. Con buen conocimiento semántico y sintáctico, que con frecuencia les despierta el "síndrome del usuario poderoso"; es decir, individuos que buscan atajos y modos de interacción abreviados.

PUNTO CLAVE

El modelo mental del usuario conforma el modo en el que éste percibe la interfaz y señala si satisface sus necesidades.

El *modelo mental* del usuario (percepción del sistema) es la imagen del sistema que los usuarios finales llevan en la cabeza. Por ejemplo, si se pidiera a un usuario de un procesador de texto en particular que describiera su operación, lo que guiaría su respuesta sería la percepción que tuviera del sistema. La exactitud de la descripción dependerá del perfil del usuario (por ejemplo, en el mejor de los casos, los principiantes darán una respuesta esquemática) y de la familiaridad general con el software en el dominio de la aplicación. Un usuario que entienda bien los procesadores de texto, pero que haya trabajado con el procesador específico una sola vez, tal vez esté más preparado para hacer una descripción más completa de su funcionamiento que el principiante que haya pasado semanas tratando de entender el sistema.

El *modelo de implementación* combina la manifestación externa del sistema basado en computadora (la vista y sensación de la interfaz) con toda la información de apoyo (libros, manuales, videos, archivos de ayuda, etc.) que describe la sintaxis y semántica de la interfaz. Cuando el modelo de la implementación y el modelo mental del usuario coinciden, quienes utilizan el software por lo general se sienten cómodos con éste y lo usan de manera eficaz. Para lograr esta "fusión" de los modelos, el modelo del diseño debe haberse desarrollado de manera que incluya la información contenida en el modelo del usuario, y el modelo de la implementación debe reflejar de manera exacta la información sintáctica y semántica de la interfaz.

Cita:

"… hay que poner atención en lo que los usuarios hacen, no en lo que dicen."

Jakob Nielsen

Los modelos descritos en esta sección son "abstracciones de lo que el usuario hace o piensa que hace o de lo que alguien piensa que debe hacerse cuando se usa un sistema interactivo" [Mon84]. En esencia, estos modelos permiten que el diseñador de la interfaz satisfaga un ele-

1 En este contexto, *conocimiento sintáctico* se refiere a la mecánica de interacción que se requiere para usar con eficacia la interfaz.

2 El término *conocimiento semántico* se refiere al sentido subyacente de la aplicación: el entendimiento de las funciones que se realizan, el significado de la entrada y salida, y las metas y objetivos del sistema.

mento clave del principio más importante del diseño de la interfaz de usuario: "Conocer al usuario, conocer las tareas."

11.2.2 El proceso

El proceso de análisis y diseño de interfaces de usuario es iterativo y se representa con un modelo espiral similar al que se estudió en el capítulo 2. En relación con la figura 11.1, el proceso de análisis y diseño de la interfaz de usuario comienza en el interior de la espiral e incluye cuatro actividades estructurales distintas [Man97]: 1) análisis y modelado de la interfaz, 2) diseño de ésta, 3) construcción y 4) validación. La espiral que se presenta en la figura 11.1 implica que cada una de dichas tareas tendrá lugar más de una vez y que cada recorrido del contorno de la espiral representa una elaboración mayor de los requerimientos y del diseño resultante. En la mayoría de los casos, la actividad de modelado involucra la hechura de prototipos, única forma práctica de validar lo que se haya diseñado.

Cita:

"Es mejor diseñar la experiencia del usuario que rectificarla."

Jon Meads

El *análisis de la interfaz* se centra en el perfil de los usuarios que interactuarán con el sistema. Se registra el nivel de habilidad, la comprensión del negocio y la receptividad general hacia el nuevo sistema; también se definen diferentes categorías de usuarios. Se recaban los requerimientos de cada una de éstas. En esencia, se trabaja para entender la percepción del sistema (véase la sección 11.2.1) para cada clase de usuarios.

Una vez definidos los requerimientos generales, se lleva a cabo un detallado *análisis de la tarea*. Asimismo, se identifican, describen y elaboran aquellas tareas que el usuario realice para alcanzar las metas del sistema (por medio de varios recorridos de la espiral). En la sección 11.3, se estudia con más detalle el análisis de la tarea. Por último, el análisis del ambiente del usuario se centra en las características físicas del lugar de trabajo. Entre las preguntas por responder se encuentran las siguientes:

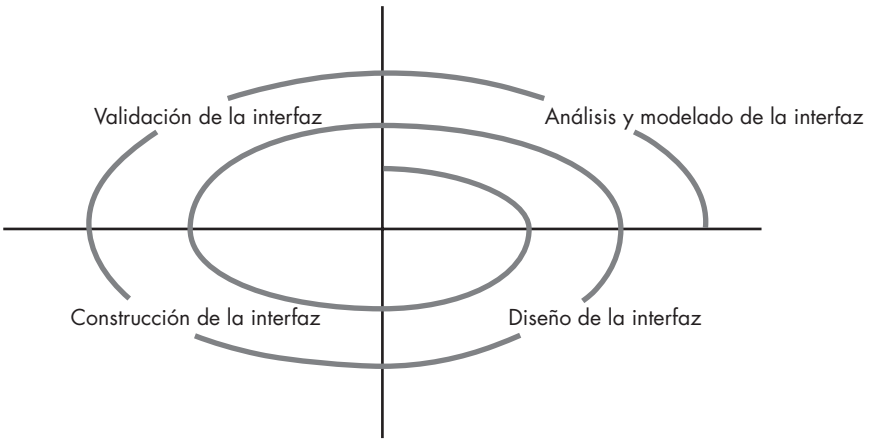
? ¿Qué se necesita saber sobre el ambiente cuando comienza el diseño de la interfaz de usuario?

- ¿Dónde se encontrará físicamente la interfaz?
- ¿El usuario estará sentado, de pie o haciendo otras tareas no relacionadas con la interfaz?
- ¿El hardware de la interfaz cumple las restricciones de espacio, iluminación o ruido?
- ¿Hay consideraciones especiales de factores humanos generadas por los factores ambientales?

La información recabada como parte del análisis se utiliza para crear un modelo de análisis de la interfaz. Con este modelo como base comienza la acción de diseñar.

FIGURA 11.1

Proceso de diseño de la interfaz de usuario



La meta del *diseño de la interfaz* es definir un conjunto de objetos y acciones de ésta (y sus representaciones en la pantalla) que permitan al usuario efectuar todas las tareas definidas en forma tal que cumpla cada meta de la usabilidad definida para el sistema. El diseño de la interfaz se estudia con más detalle en la sección 11.4.

La *construcción de la interfaz* comienza por lo general con la creación de un prototipo que permite evaluar los escenarios de uso. A medida que avanza el proceso de diseño, se emplea un grupo de herramientas de la interfaz de usuario (véase la sección 11.5) para terminar de construirla.

La *validación de la interfaz* se centra en: 1) la capacidad de la interfaz para implementar correctamente todas las tareas del usuario, incluir todas las variaciones de éstas y alcanzar todos los requerimientos generales del usuario; 2) el grado en el que la interfaz es fácil de usar y de aprender y 3) la aceptación que tiene por parte del usuario como herramienta útil en su trabajo.

Como ya se dijo, las actividades descritas en esta sección ocurren de manera iterativa. Por esto, no es necesario intentar especificar cada uno de los detalles (del modelo de análisis o de diseño) en la primera etapa. En los pasos posteriores del proceso, se elaboran los detalles de la tarea, la información de diseño y las características de operación de la interfaz.

11.3 Análisis de la interfaz[3]

Un aspecto clave de todos los modelos del proceso de la ingeniería de software es éste: *entender el problema antes de tratar de diseñar una solución*. En el caso del diseño de la interfaz de usuario, entender el problema significa comprender: 1) a las personas (usuarios finales) que interactuarán con el sistema a través de la interfaz, 2) las tareas que los usuarios finales deban realizar como parte de su trabajo, 3) el contenido que se presenta como parte de la interfaz y 4) el ambiente en el que se efectuarán estas tareas. En las secciones siguientes se analizan estos elementos del análisis de la interfaz, en un intento por establecer un fundamento sólido para las tareas de diseño siguientes.

11.3.1 Análisis del usuario

Es probable que la frase "interfaz de usuario" sea toda la justificación que se necesita para dedicar algo de tiempo a entender al usuario antes de preocuparse por los aspectos técnicos. Ya se dijo que cada usuario tiene una imagen mental del software y que ésta puede ser diferente de la que se forman otros usuarios. Además, la imagen mental del usuario tal vez sea muy distinta de la del modelo del diseño que hizo el ingeniero de software. La única manera en la que se logra hacer converger la imagen mental y el modelo de diseño es trabajando para entender a los usuarios y la forma en la que usarán el sistema. Para ello, se utiliza información procedente de una variedad amplia de fuentes:

¿Cómo se aprende lo que el usuario quiere de la interfaz?

Entrevistas. Éste es el enfoque más directo, los miembros del equipo de software se reúnen con los usuarios para entender mejor sus necesidades, motivaciones, cultura laboral y una multitud de aspectos adicionales. Esto se logra en reuniones individuales o a través de grupos de enfoque.

Información de ventas. El personal de ventas habla con los usuarios de manera regular y recaba información que ayuda al equipo de software a clasificarlos y a entender mejor sus requerimientos.

3 Es razonable afirmar que esta sección debiera formar parte del capítulo 5, 6 o 7, pues en éstos se estudian aspectos del análisis de los requerimientos. Se situó aquí porque el análisis y el diseño de la interfaz se relacionan de cerca entre sí, y es frecuente que la frontera sea nebulosa.

CONSEJO

Sobre todo, dedique tiempo a hablar con los usuarios reales, pero con cuidado. Una opinión enfática no necesariamente significa que la mayoría de usuarios esté de acuerdo con ella.

Información de mercadotecnia. El análisis del mercado es invaluable para la definición de segmentos del mercado y para la comprensión sobre cómo usará el software en formas sutilmente distintas cada uno de estos segmentos.

Información de apoyo. El equipo de apoyo habla a diario con los usuarios. Constituye la fuente de información más probable acerca de lo que funciona y lo que no, lo que les gusta a los usuarios y lo que les desagrada, qué características generan preguntas y cuáles son fáciles de usar.

Las preguntas siguientes (adaptadas de [Hac98]) ayudarán al lector a entender mejor a los usuarios de un sistema:

PUNTO CLAVE

¿Cómo se aprende sobre la demografía y las características de los usuarios finales?

- ¿Los usuarios son profesionales capacitados, técnicos, oficinistas o trabajadores de manufactura?
- ¿Qué nivel de educación formal tiene el usuario promedio?
- ¿Los usuarios son capaces de aprender mediante materiales escritos o han manifestado el deseo de recibir enseñanzas en un aula?
- ¿Los usuarios son mecanógrafos expertos o tienen fobia a los teclados?
- ¿Cuál es el rango de edades de la comunidad de usuarios?
- ¿Los usuarios estarán representados sobre todo por un género?
- ¿Cómo se compensa a los usuarios por el trabajo que realizan?
- ¿Los usuarios trabajan en un horario normal de oficina o hasta terminar el trabajo que hacen?
- ¿El software va a ser parte integral del trabajo de los usuarios o sólo lo emplearán de manera ocasional?
- ¿Cuál es el idioma principal de los usuarios?
- ¿Cuáles son las consecuencias si el usuario comete un error al emplear el sistema?
- ¿Los usuarios son expertos en el tema en el que está centrado el sistema?
- ¿Los usuarios quieren saber sobre la tecnología que hay tras la interfaz?

Una vez respondidas estas preguntas, se sabrá quiénes son los usuarios finales, qué es probable que los motive y agrade, cómo se clasificarían en distintas clases o perfiles de usuarios, cuáles son sus modelos mentales del sistema y cómo debe caracterizarse la interfaz de usuario para que satisfaga sus necesidades.

11.3.2 Análisis y modelado de la tarea

La meta del análisis de la tarea es responder las siguientes preguntas:

PUNTO CLAVE

La meta del usuario es cumplir una o más de las tareas a través de la interfaz. Para ello, ésta debe brindar mecanismos que le permitan realizarlas.

- ¿Qué trabajo realizará el usuario en circunstancias específicas?
- ¿Qué tareas y subtareas se efectuarán cuando el usuario haga su trabajo?
- ¿Qué dominio de problema específico manipulará el usuario al realizar su labor?
- ¿Cuál es la secuencia de las tareas (el flujo del trabajo)?
- ¿Cuál es la jerarquía de las tareas?

Para responder estas preguntas, debe recurrirse a las técnicas que ya se estudiaron en el libro, pero aplicadas en este caso a la interfaz de usuario.

Casos de uso. En capítulos anteriores se aprendió que el caso de uso describe la manera en la que un actor (en el contexto del diseño de la interfaz de usuario, un actor siempre es una persona) interactúa con el sistema. Cuando se utiliza como parte del análisis de la tarea, el caso

de uso se desarrolla con objeto de mostrar la forma en la que un usuario final lleva a cabo alguna tarea específica relacionada con el trabajo. En la mayoría de las veces, el caso de uso se escribe en un estilo informal (un simple párrafo) en primera persona. Por ejemplo, suponga que una empresa pequeña de software quiere elaborar un sistema de diseño asistido por computadora dirigido explícitamente a diseñadores de interiores. Para entender mejor la forma en la que realizan su trabajo, se pide a diseñadores reales que describan una función específica del diseño. Cuando se preguntó a un diseñador de interiores: "¿Cómo decide si se coloca mobiliario en una habitación?", respondió con el siguiente caso de uso informal:

> Comienzo por hacer un plano de la habitación, de las dimensiones y de la ubicación de puertas y ventanas. Me preocupa mucho cómo entra la luz al cuarto, la vista que hay por las ventanas (si es bella, quiero dirigir la atención hacia ese lugar), la longitud de una pared no obstruida y el flujo del movimiento por la habitación. Después me fijo en la lista de muebles que el cliente haya elegido: mesas, sillas, sofá, gabinetes; la lista de accesorios: lámparas, tapetes, pinturas, esculturas, plantas, adornos, y tomo notas acerca de cualquier deseo que mi cliente tenga. Después dibujo cada objeto de mis listas en un formato a escala de la casa. Pongo letreros en lo que haya dibujado y utilizo lápiz porque siempre las muevo. Luego hago una perspectiva (dibujo tridimensional) del cuarto para dar a mi cliente la sensación de cómo se vería.

Este caso de uso presenta la descripción básica de una tarea importante del trabajo para el sistema de diseño asistido por computadora. De él pueden extraerse tareas, objetos y el flujo general de la interacción. Además, pueden concebirse otras características del sistema que agradarían al diseñador de interiores. Por ejemplo, podría tomarse una fotografía digital a través de cada ventana de la habitación. Cuando se elabore la perspectiva, podría representarse la vista exterior real que habrá en todas ellas.

CasaSegura

Casos de uso para el diseño de la interfaz de usuario

La escena: Cubículo de Vinod, cuando sigue el diseño de la interfaz de usuario.

Participantes: Vinod y Jamie, miembros del equipo de ingeniería de software de *CasaSegura*.

La conversación:

Jamie: Vi a nuestro contacto de mercadotecnia y le pedí que escribiera un caso de uso para la interfaz de vigilancia.

Vinod: ¿Desde el punto de vista de quién?

Jamie: Del propietario, ¿de quién más?

Vinod: También está el papel del administrador del sistema; aun si fuera el propietario, sería un punto de vista distinto. El "administrador" prepara el sistema, lo configura, elige el plano, sitúa las cámaras…

Jamie: Todo lo que le pedí fue que desempeñara el papel de una propietaria que quiere ver el video.

Vinod: Está bien. Es uno de los comportamientos principales de la interfaz de la función de vigilancia. Pero también vamos a tener que examinar el comportamiento de la administración del sistema.

Jamie (irritado): Tienes razón.

[Jamie sale para encontrarse con la persona de mercadotecnia. Regresa pocas horas después.]

Jamie: Tuve suerte, la encontré y trabajamos juntos en el caso de uso del administrador. Básicamente vamos a definir "administración" como una función aplicable a todas las demás de *CasaSegura*. Aquí está lo que obtuvimos.

[Jamie presenta el caso de uso informal a Vinod.]

Caso de uso informal: Quiero poder preparar o editar la plantilla del sistema en cualquier momento. Cuando preparo el sistema, selecciono una función de administración. Ésta pregunta si deseo hacer una nueva sesión o editar una ya existente. Si selecciono una nueva, el sistema presenta una pantalla de dibujo que permite dibujar el plano de la casa en una cuadrícula. Habrá iconos para paredes, ventanas y puertas, de manera que sea fácil dibujarlas. Sólo estiro los iconos a sus longitudes apropiadas. El sistema mostrará las longitudes en pies o metros (puedo elegir el sistema de medidas). Selecciono en una biblioteca sensores y cámaras, y las coloco en el plano de la casa. Etiqueto cada una o el sistema lo hace de manera automática. Puedo establecer los valores de los sensores y cámaras desde menús apropiados. Si elijo editarlos, puedo mover sensores o cámaras, agregar otros nuevos o eliminar los ya existentes, editar el plano de la casa y editar los parámetros de sensores y cámaras. En todo caso, espero que el sistema haga una comprobación consistente y que me ayude a evitar los errores.

Vinod (después de leer el escenario): Bien, es probable que haya algunos patrones de diseño útiles [véase el capítulo 12] o componentes reutilizables para las interfaces gráficas de usuario para los programas de dibujo. Apuesto 50 dólares a que implementamos una parte o el total de la interfaz del administrador con el uso de ellos.

Jamie: De acuerdo. Lo verificaré.

Elaboración de la tarea. En el capítulo 8 se vio la elaboración paso a paso (también llamada descomposición de funciones o refinamiento *stepwise* o por etapas) como un mecanismo para mejorar las tareas del procesamiento requeridas para que el software realice alguna función deseada. El análisis de la tarea para el diseño de la interfaz utiliza un enfoque de elaboración para ayudar a entender las actividades humanas que la interfaz de usuario debe incluir.

La elaboración de la tarea es muy útil, pero también puede ser peligrosa. No porque haya hecho una tarea debe suponer que no existe otra forma de realizarla, y que no se intentará cuando se implemente la interfaz de usuario.

El análisis de la tarea se aplica en dos formas. Como ya se dijo, es frecuente que un sistema interactivo basado en computadora se utilice para remplazar una actividad manual o semimanual. Para entender las tareas que deben realizarse a fin de lograr el objetivo de la actividad, deben entenderse las tareas que llevan a cabo las personas (con el enfoque manual) y luego mapearlas en un conjunto de tareas similar (pero no necesariamente idéntico) que se implementen en el contexto de la interfaz de usuario. En forma alternativa, puede estudiarse una especificación existente para obtener una solución basada en computadora y derivar un conjunto de tareas de usuario que incluirán al modelo de usuario, al del diseño y la percepción del sistema.

Sin importar el enfoque general del análisis de la tarea, primero deben definirse y clasificarse las tareas. Ya se dijo que un enfoque es la elaboración paso a paso. Por ejemplo, volvamos a considerar el sistema de diseño asistido por computadora para diseñadores de interiores. Al mirar trabajar a un diseñador de interiores, verá que su labor comprende varias actividades principales: distribución de los muebles (recuerde el caso de uso mencionado), selección de telas y materiales, elección de cubiertas de paredes y ventanas, presentación (al cliente), cotización y compras. Cada una de estas tareas principales se divide en subtareas. Por ejemplo, con el uso de la información contenida en el caso de uso, la distribución del mobiliario se desglosa en las tareas siguientes: 1) dibujar un plano con base en las dimensiones de la habitación, 2) colocar puertas y ventanas en las ubicaciones apropiadas, 3a) usar plantillas de muebles para dibujar en el plano bosquejos del mobiliario a escala, 3b) usar plantillas de moldes para dibujar en el plano formas a escala, 4) mover los bosquejos de muebles y las formas para obtener la mejor colocación, 5) poner leyendas en todos los bosquejos de muebles y formas, 6) dibujar dimensiones para mostrar la ubicación y 7) dibujar una perspectiva para el cliente. Para cada una de las demás tareas principales podría emplearse un enfoque similar.

Es posible desglosar aún más las subtareas 1 a 7. Las subtareas 1 a 6 mejorarán, manipulando la información y realizando acciones dentro de la interfaz de usuario. Por otro lado, la subtarea 7 se ejecuta en forma automática en el software y dará como resultado poca interacción directa con el usuario.[4] El modelo del diseño de la interfaz debe incluir cada una de estas tareas en forma consistente con el modelo del usuario (perfil de un diseñador de interiores "común") y con la percepción del sistema (lo que el diseñador de interiores espera de un sistema automatizado).

Elaboración del objeto. En vez de centrarse en las tareas que debe realizar un usuario, puede examinarse el caso de uso y la demás información obtenida del usuario para extraer los objetos físicos que usa el diseñador de interiores. Estos objetos se dividen en clases: se definen los atributos de las clases, y la evaluación de las acciones aplicadas a cada objeto proporciona

4 Sin embargo, tal vez no sea éste el caso. El diseñador de interiores podría desear especificar la perspectiva que se va a dibujar, la escala, el uso del color y otra información. El caso de uso relacionado con el dibujo de la perspectiva daría la información necesaria para efectuar esta tarea.

CONSEJO

Aunque la elaboración del objeto es útil, no debe usarse como enfoque único. Durante el análisis de la tarea debe tomarse en cuenta la opinión del usuario.

una lista de operaciones. Por ejemplo, la plantilla de muebles se traduce en una clase llamada **Mobiliario** con atributos que incluirían **tamaño, forma y ubicación**, entre otros. El diseñador de interiores *seleccionaría* el objeto de la clase **Mobiliario**, lo *movería* a cierta posición en el plano (otro objeto, en este contexto), *dibujaría* el bosquejo del mueble, etc. Las tareas *seleccionar*, *mover* y *dibujar* son operaciones. El modelo del análisis de la interfaz de usuario no daría una implementación literal para cada una de estas operaciones. No obstante, a medida que se elabore el diseño, se definen los detalles de cada operación.

Análisis del flujo del trabajo. Cuando varios usuarios distintos, cada uno con diferentes roles, utilizan una interfaz de usuario, a veces es necesario ir más allá del análisis de la tarea y de la elaboración del objeto, y aplicar el *análisis del flujo del trabajo*. Esta técnica permite entender cómo se efectúa un proceso de trabajo cuando están involucradas varias personas (y roles). Considere una compañía que trate de automatizar por completo el proceso de surtir y entregar recetas de medicinas. Todo el proceso[5] giraría alrededor de una aplicación basada en web accesible para médicos (o sus asistentes), farmacéuticos y pacientes. El flujo del trabajo puede representarse con eficacia con un diagrama UML de canal (variación del diagrama de actividades).

Sólo consideraremos una pequeña parte del proceso de trabajo: la situación que ocurre cuando un paciente solicita una nueva entrega. La figura 11.2 presenta el diagrama de canal que indica las tareas y decisiones para cada uno de los tres roles ya mencionados. Esta información se obtiene de entrevistas o de casos de uso escritos por cada actor. No obstante, el flujo de eventos (mostrados en la figura) permite reconocer cierto número de características clave de la interfaz:

Cita:

"Es mucho mejor adaptar la tecnología al usuario que obligar a éste a adaptarse a la tecnología."

Larry Marine

1. Cada usuario implementa tareas a través de la interfaz: entonces, la vista y sensación de la interfaz diseñada para el paciente será distinta de la que se definió para farmacéuticos o médicos.
2. El diseño de la interfaz para farmacéuticos y médicos debe incluir acceso, y desplegar la información desde fuentes secundarias (por ejemplo, acceder al inventario para el farmacéutico y acceder a información acerca de medicaciones alternativas para el médico).
3. Muchas de las actividades anotadas en el diagrama de canal se elaboran más con el uso del análisis de la tarea o la elaboración del objeto (por ejemplo, *Surtir receta* implicaría la entrega de una orden por correo, una visita a la farmacia o visitar un centro de distribución de medicinas especiales).

Representación jerárquica. Al comenzar a analizar la interfaz, ocurre un proceso de elaboración. Una vez establecido el flujo de los trabajos, se define una jerarquía de tareas para cada tipo de usuario. Ésta proviene de la elaboración paso a paso de cada tarea identificada para el usuario. Por ejemplo, considere las tareas de usuario siguientes y la jerarquía de las subtareas.

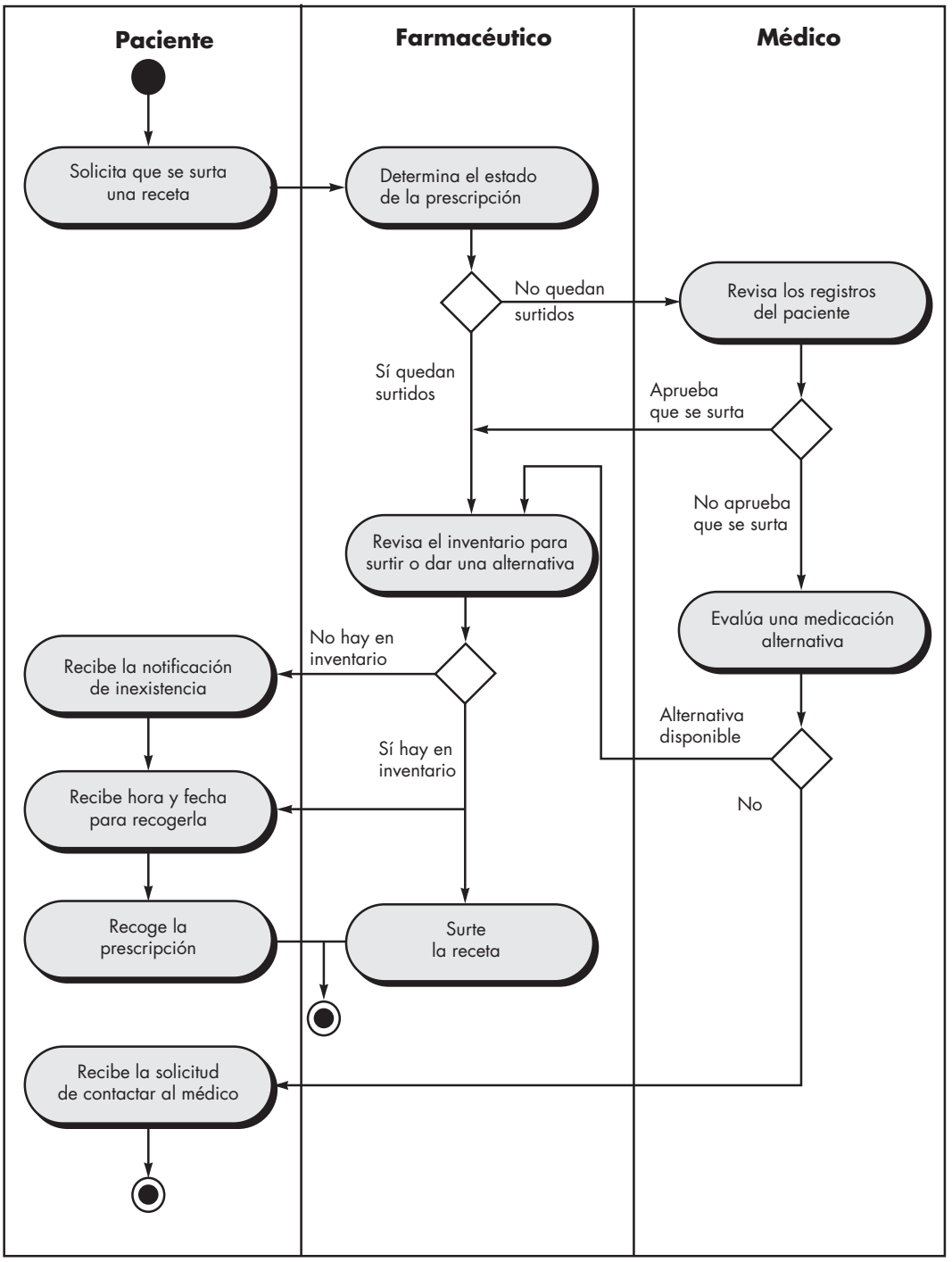
Tarea del usuario: *Solicitar que se surta una receta*

- *Dar información de identificación.*
  - *Especificar nombre.*
  - *Especificar identificación de usuario.*
  - *Especificar PIN y clave.*
- *Especificar número de prescripción.*
- *Especificar la fecha en la que se requiere surtir.*

Para terminar la tarea, se definen tres subtareas. Una de ellas, *dar información de identificación*, se elabora más en tres subsubtareas adicionales.

5 Este ejemplo se adaptó de [Hac98].

FIGURA 11.2
Diagrama de canal para la función de surtir receta



11.3.3 Análisis del contenido de la pantalla

Las tareas identificadas en la sección 11.3.2 conducen a la presentación de varios tipos diferentes de contenido. Para las aplicaciones modernas, el contenido de la pantalla varía de reportes basados en caracteres (como una hoja de cálculo), gráficas (histograma, modelo tridimensional, fotografía de alguien) o información especializada (por ejemplo, archivos de audio o video). Las

técnicas de modelado del análisis estudiadas en los capítulos 6 y 7 identifican los objetos de datos de salida producidos por una aplicación. Estos objetos de datos pueden ser: 1) generados por componentes (no relacionados con la interfaz) en otras partes de la aplicación, 2) adquiridos a partir de datos almacenados en una base accesible desde la aplicación o 3) transmitidos desde sistemas externos a la aplicación en cuestión.

Durante esta etapa del análisis de la interfaz, se toma en cuenta el formato y la estética del contenido (según la despliega la interfaz). Entre las preguntas planteadas y respondidas, se encuentran las siguientes:

¿Cómo se determina el formato y la estética del contenido desplegado como parte de la interfaz de usuario?

- ¿Se asignan diferentes tipos de datos a sitios consistentes en la geografía de la pantalla (por ejemplo, las fotografías aparecen siempre en la esquina superior derecha)?
- ¿El usuario puede personalizar la ubicación del contenido en la pantalla?
- ¿Se asigna una identificación apropiada a todo el contenido que hay en la pantalla?
- Si se presenta un reporte grande, ¿cómo debe dividirse para facilitar su comprensión?
- ¿Se dispondrá de mecanismos para pasar directamente a información resumida de grandes conjuntos de datos?
- ¿Las salidas gráficas estarán a escala para que se ajusten a los bordes del dispositivo de pantalla que se utilice?
- ¿Cómo se empleará el color para mejorar la comprensión?
- ¿De qué manera se presentará al usuario los mensajes de error y las advertencias?

Las respuestas a estas (y otras) preguntas ayudarán a establecer los requerimientos de la presentación del contenido.

11.3.4 Análisis del ambiente de trabajo

Hackos y Redish [Hac98] estudian la importancia del análisis del ambiente de trabajo cuando afirman lo siguiente:

> Las personas no realizan aisladas su trabajo. Están influidas por la actividad que las rodea, las características físicas del sitio de trabajo, el tipo de equipo que usan y las relaciones laborales que tienen con las demás personas. Si los productos que usted diseña no se ajustan al ambiente, su uso será difícil o frustrante.

En ciertas aplicaciones se coloca la interfaz de usuario de un sistema basado en computadora en una "ubicación amigable" (con iluminación adecuada, buena altura de la pantalla, acceso fácil al teclado, etc.), pero en otras (como en una fábrica o en la cabina de un avión) la iluminación es menos que buena, el ruido es notable, un teclado o ratón no son opción y la posición de la pantalla no es la ideal. El diseñador de la interfaz puede estar restringido por factores que se confabulan contra la facilidad del uso.

Además de los factores del ambiente físico, también entra en juego la cultura del sitio de trabajo. ¿Se medirá de algún modo la interacción con el sistema (por ejemplo, el tiempo de cada transacción o la exactitud de ésta)? ¿Tendrán que compartir información dos o más personas para que se dé una entrada? ¿Cómo se apoyará a los usuarios del sistema? Estas preguntas y muchas otras relacionadas deben responderse antes de que comience el diseño de la interfaz.

11.4 Etapas del diseño de la interfaz

Una vez concluido el análisis de la interfaz, todas las tareas (u objetos y acciones) requeridas por el usuario final habrán sido identificadas en detalle y comenzará la actividad de diseño de la interfaz. El diseño de la interfaz, como todo el de la ingeniería de software, es un proceso

Cita:

"El diseño interactivo [es] una mezcla tersa de las artes gráficas, la tecnología y la psicología."

Brad Wieners

iterativo. Cada etapa del diseño de la interfaz de usuario ocurre varias veces, en las que se elabora y refina la información desarrollada en la etapa anterior.

Aunque se han propuesto muchos modelos diferentes para el modelo de la interfaz de usuario ([Nor86], [Nie00]), todos sugieren alguna combinación de las etapas siguientes:

1. Definir objetos y acciones de la interfaz (operaciones) con el uso de la información desarrollada en el análisis de la interfaz (sección 11.3).
2. Definir eventos (acciones del usuario) que harán que cambie el estado de la interfaz de usuario. Hay que modelar este comportamiento.
3. Ilustrar cada estado de la interfaz como lo vería en la realidad el usuario final.
4. Indicar cómo interpreta el usuario el estado del sistema a partir de la información provista a través de la interfaz.

En ciertos casos, se comienza con bosquejos de cada estado de la interfaz (lo que vería el usuario en diferentes circunstancias) y después se trabaja hacia atrás para definir objetos, acciones y otra información importante del diseño. Sin importar la secuencia de las tareas de diseño, debe 1) siempre apegarse a las reglas doradas estudiadas en la sección 11.1, 2) modelar la forma en la que se va a implementar la interfaz y 3) considerar el ambiente (por ejemplo, tecnología de la pantalla, sistema operativo, herramientas de desarrollo, etc.) que se empleará.

11.4.1 Aplicación de las etapas de diseño de la interfaz

Una etapa importante del diseño de la interfaz es la definición de objetos de la interfaz y de las acciones que se aplican a ellos. Para lograr esto, se elaboran escenarios de uso en forma muy parecida a la descrita en el capítulo 6. Es decir, se escribe un caso de uso. Se aíslan los sustantivos (objetos) y verbos (acciones) a fin de crear una lista de objetos y acciones.

Una vez definidos éstos y elaborados en forma iterativa, se clasifican por tipo. Se identifican los objetos blanco, fuente y aplicación. Un *objeto fuente* (icono de reporte, por ejemplo) se arrastra y se deja sobre un *objeto blanco* (como un icono de impresora). La consecuencia de esta acción es que se crea un informe en papel. Un *objeto de aplicación* representa datos específicos de la aplicación que no se manipulan directamente como parte de la interacción en la pantalla. Por ejemplo, se usa una lista de correo para guardar nombres. La lista en sí tal vez se ordene, fusione o purgue (acciones basadas en menús), pero no se arrastra y suelta por medio de alguna interacción del usuario.

Cuando se está satisfecho con la definición de todos los objetos y acciones importantes (para una iteración del diseño), se realiza la distribución de la pantalla. Como otras actividades del diseño de la interfaz, la distribución de la pantalla es un proceso interactivo en el que se llevan a cabo el diseño gráfico y la colocación de iconos, la definición de descripciones de texto en la pantalla, la especificación y títulos de las ventanas, y la definición de aspectos grandes y pequeños del menú. Si hubiera una metáfora apropiada del mundo real para la aplicación, éste es el momento en el que se especifica, y la distribución se organiza en forma tal que complemente a la metáfora.

Para dar una ilustración breve de las etapas de diseño mencionadas, considere un escenario de uso para el sistema *CasaSegura* (estudiado en capítulos anteriores). A continuación se presenta un caso de uso preliminar (escrito por el propietario) para la interfaz:

> **Caso de uso preliminar:** Quiero tener acceso a mi sistema de *CasaSegura* a través de internet, desde cualquier lugar remoto. Con el uso de un software de navegación que opere en mi computadora portátil (cuando estoy en el trabajo o de viaje), determino el estado del sistema de alarma, activo o desactivo el sistema, vuelvo a configurar zonas de seguridad y veo diferentes habitaciones de la casa por medio de las cámaras de video instaladas.

Para acceder a *CasaSegura* desde una ubicación remota, doy una identificación y una clave. Éstas definen niveles de acceso (por ejemplo, no a todos los usuarios se les permite reconfigurar el sistema) y dan seguridad. Una vez validadas, reviso el estado del sistema y lo cambio activando o desactivando *CasaSegura*. Reconfiguro el sistema con el plano de la casa, con la vista de cada uno de los sensores de seguridad, con el despliegue de cada zona ya configurada y con la modificación de las zonas según se requiera. Puedo ver el interior de la casa por medio de cámaras de video colocadas estratégicamente. Abro el ángulo de visión y acerco la toma que da cada cámara de las diferentes vistas del interior.

Con base en este caso de uso, se identifican las siguientes tareas del propietario, objetos y datos:

- *acceder* al sistema *CasaSegura*
- *introducir* una **identificación** y **clave** que permitan el acceso remoto
- *comprobar* el **estado del sistema**
- *activar* o *desactivar* el **sistema** *CasaSegura*
- *mostrar* el **plano** y las **ubicaciones de los sensores**
- *mostrar* las **zonas** del plano
- *cambiar* las **zonas** en el plano
- *mostrar* las **ubicaciones de las cámaras de video** en el plano
- *seleccionar* una **cámara de video** para ver
- *ver* **imágenes de video** (cuatro cuadros por segundo)
- *abrir* el **ángulo** o *acercar* la **cámara de video**

De esta lista de tareas del propietario se extraen los objetos (en negritas) y las acciones (en cursiva). La mayoría de objetos anotados son de aplicación. Sin embargo, la **ubicación de cámara de video** (objeto fuente) se arrastra y suelta en **cámara de video** (objeto blanco) para crear una **imagen de video** (ventana con pantalla de video).

Aunque las herramientas automatizadas pueden ser útiles en el desarrollo de prototipos de la distribución, en ocasiones todo lo que se necesita es lápiz y papel.

Se crea un bosquejo preliminar de la distribución de la pantalla para la vigilancia con video (véase la figura 11.3).[6] Para invocar la imagen de video, se selecciona un icono de ubicación de cámara de video, *C*, ubicado en el plano que aparece en la ventana de vigilancia. En este caso, a continuación se arrastra una ubicación de cámara en la sala (S) y se suelta en el icono de cámara de video que aparece en la parte superior izquierda de la pantalla. Aparece la ventana de video y muestra una toma de la cámara localizada en la sala. Los controles deslizantes de acercamiento y apertura se utilizan para controlar el aumento y dirección de la imagen de video. Para seleccionar una vista de otra cámara, el usuario simplemente arrastra y suelta un icono diferente de ubicación de cámara en el correspondiente a la cámara que hay en la esquina superior izquierda de la pantalla.

El bosquejo de distribución tendría que darse con una expansión de cada objeto de menú de la barra, que indicarían cuáles acciones están disponibles en el modo de vigilancia con video (estado). Durante el diseño de la interfaz se crearía un conjunto completo de bosquejos para cada una de las tareas del propietario anotadas en el escenario de uso.

11.4.2 Patrones de diseño de la interfaz de usuario

Las interfaces de usuario gráficas se han vuelto tan comunes que ha surgido una amplia variedad de patrones de diseño de ellas. Como ya se dijo en este libro, un patrón de diseño es una

6 Observe que esto difiere un poco de la implementación de estas características en los capítulos anteriores. Esto tal vez se considere un primer bosquejo del diseño y represente una alternativa digna de tomarse en cuenta.

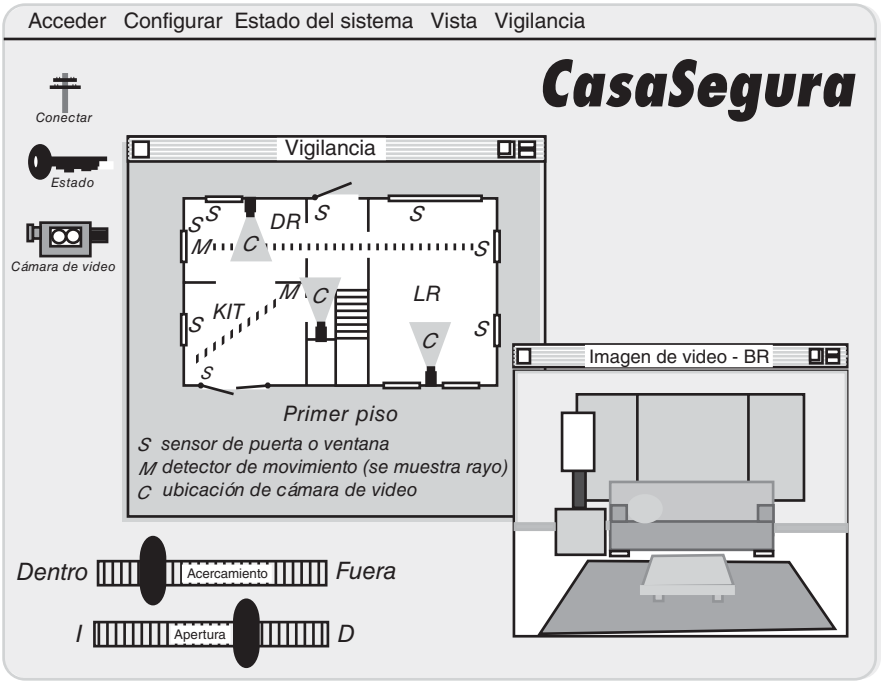


FIGURA 11.3
Distribución preliminar de la pantalla

abstracción que prescribe una solución de diseño para un problema de diseño bien delimitado.

WebRef

Se ha propuesto una amplia variedad de patrones de diseño de interfaces gráficas. En la dirección **www.hcipatterns.org** se encuentran varios vínculos hacia sitios de patrones.

Como ejemplo de un problema de diseño de la interfaz que es común encontrar, considere una situación en la que un usuario debe introducir una o más fechas, a veces varios meses antes. Hay muchas soluciones para este sencillo problema y se han propuesto varios patrones distintos. Laakso [Laa00] sugiere un patrón llamado **CalendarStrip** que produce un calendario continuo, giratorio, en el que se resalta la fecha actual y se eligen las futuras, tomándolas del calendario. La metáfora del calendario es bien conocida por todos los usuarios y da un mecanismo eficaz para situar en contexto una fecha futura.

En la última década se han propuesto muchos patrones de diseño de la interfaz. En el capítulo 12 se presenta un análisis más detallado de los patrones de diseño de la interfaz de usuario. Además, Erickson [Eri08] proporciona vínculos a muchos grupos basados en web.

11.4.3 Aspectos del diseño

A medida que evoluciona una interfaz de usuario, casi siempre surgen cuatro aspectos comunes del diseño: tiempo de respuesta del sistema, herramientas de ayuda para el usuario, manejo de información errónea y leyendas de los comandos. Desafortunadamente, son muchos los diseñadores que no enfrentan estos aspectos hasta que es relativamente tarde en el proceso de diseño (a veces sucede que la primera sospecha de que existe un problema no ocurre hasta que ya existe un prototipo operativo). Es frecuente que el resultado sean iteraciones innecesarias, retrasos en el proyecto y frustración del usuario final. Es mucho mejor establecer cada uno de ellos como un aspecto del diseño que debe tomarse en cuenta al comenzar el diseño del software, cuando es fácil y barato hacer cambios.

Tiempo de respuesta. El tiempo de respuesta del sistema es la queja principal en muchas aplicaciones interactivas. En general, se mide desde el momento en el que el usuario ejecuta alguna acción de control (por ejemplo, oprime la tecla de "enter" o hace clic en el ratón) y hasta que el software responde con la salida o acción deseada.

El tiempo de respuesta tiene dos características importantes: longitud y variabilidad. Si el tiempo de respuesta es demasiado largo, es inevitable que el usuario sienta frustración y tensión. La *variabilidad* se refiere a la desviación del tiempo de respuesta promedio y, por muchos aspectos, es su característica más importante. La variabilidad baja permite que el usuario establezca un ritmo de interacción, aun si el tiempo de respuesta fuera relativamente largo. Por ejemplo, un tiempo de respuesta de 1 segundo para un comando resulta con frecuencia preferible a una respuesta que varíe de 0.l a 2.5 segundos. Cuando la variabilidad es significativa, el usuario siempre se sale de balance, se pregunta si tras bambalinas ha ocurrido algo "distinto".

Cita:

"Un error común que cometen las personas cuando tratan de diseñar algo por completo a prueba de tontos es subestimar la ingenuidad de los tontos completos."

Douglas Adams

Herramientas de ayuda. Casi siempre, todo usuario de un sistema interactivo basado en computadora requiere ayuda ocasional. En ciertos casos, una simple pregunta dirigida a un colega conocedor lo resuelve. En otros, la única opción es la búsqueda detallada en muchos "manuales del usuario". Sin embargo, en la mayor parte de los casos, el software moderno brinda herramientas de ayuda en línea que permiten al usuario obtener la respuesta para sus preguntas o resolver un problema sin tener que salir de la interfaz.

Cuando se consideren las herramientas de ayuda, deben tomarse en cuenta varios aspectos del diseño [Rub88]:

- ¿Habrá ayuda para todas las funciones del sistema y en todo momento durante la interacción con éste? Las opciones incluyen ayuda para un solo subconjunto de todas las funciones y acciones o para todas las funciones.
- ¿Cómo pedirá ayuda el usuario? Las opciones incluyen un menú, una tecla de función especial o un comando AYUDA.
- ¿Cómo se presentará la ayuda? Las opciones incluyen una ventana por separado, una referencia sobre un documento impreso (lejos de ser lo ideal) o una sugerencia de uno o dos renglones generados en una posición fija de la pantalla.
- ¿Cómo volverá el usuario a la interacción normal? Las opciones incluyen un botón de regreso que se muestre en la pantalla, una tecla de función o una secuencia de control.
- ¿Cómo ayudaría que la información estuviera estructurada? Las opciones incluyen una estructura "plana" en la que se acceda a la información por medio de un teclado, una jerarquía en capas de la información que provea cada vez más detalles a medida que el usuario avance en la estructura o el uso de hipertexto.

Cita:

"La interfaz del infierno: 'para corregir este error y continuar, introduzca cualquier número primo de 11 dígitos…'".

Autor desconocido

Manejo de errores. Los mensajes de error y las advertencias son "malas noticias" que llegan a los usuarios desde sistemas interactivos cuando algo sale mal. En el peor de los casos, los mensajes de error y advertencias dan información inútil o equívoca y sólo sirven para aumentar la frustración del usuario. Son pocos los usuarios de computadoras que no se han encontrado con un error del tipo siguiente: *"La aplicación XXX ha terminado porque encontró un error del tipo 1023."* En algún lado debe de existir una explicación del error 1023; de otro modo, ¿por qué habrían incluido los diseñadores dicha información? No obstante, el mensaje no da una indicación real de lo que salió mal o dónde buscar más información. Un mensaje de error presentado de esta manera no hace nada para mitigar la ansiedad del usuario o para ayudarlo a corregir el problema.

En general, todo mensaje de error o advertencia producida por un sistema interactivo debería tener las siguientes características:

? ¿Qué características debe tener un "buen" mensaje de error?

- El mensaje debe describir el problema en un lenguaje que entienda el usuario.
- El mensaje debe dar consejos constructivos para corregir el error.
- El mensaje debe indicar cualesquiera consecuencias negativas del error (por ejemplo, archivos de datos potencialmente corrompidos) para que el usuario pueda revisarlas a fin de asegurarse de que no hayan tenido lugar (o corregirlas si las hubo).

- El mensaje debe estar acompañado de una clave audible o visual. Es decir, debe generarse un sonido que acompañe la aparición del mensaje o éste debe cintilar momentáneamente o desplegarse en un color que se reconozca con facilidad como el "color del error".
- El mensaje "no debe juzgar". Es decir, sus palabras nunca deben culpar al usuario.

Como en realidad a nadie le gustan las malas noticias, a pocos usuarios les agradará un mensaje de error, no importa lo bien diseñado que esté. Pero una filosofía eficaz de mensajes de error puede hacer mucho para mejorar la calidad de un sistema interactivo y disminuirá de manera significativa la frustración del usuario cuando sucedan los problemas.

Leyendas del menú y de los comandos. El comando escrito fue alguna vez el modo más común de interacción entre el usuario y el sistema de software, y era normal usarlo para aplicaciones de todo tipo. Hoy día, el uso de interfaces orientadas a las ventanas, y de apuntar y tomar, ha reducido la dependencia de los comandos escritos, pero algunos usuarios avanzados aún prefieren un modo de interacción orientado a los comandos. Cuando se dan comandos o leyendas en el menú como un modo de interactuar, surgen ciertos aspectos relacionados con el diseño:

- ¿Toda opción de menú tiene un comando correspondiente?
- ¿Qué forma tendrán los comandos? Las opciones incluyen una secuencia de control (por ejemplo, alt-P), teclas de función o palabras escritas.
- ¿Cuán difícil será aprender y recordar los comandos? ¿Qué puede hacerse si se olvida un comando?
- ¿Los comandos pueden ser personalizados o abreviados por el usuario?
- ¿Las leyendas del menú se explican por sí mismas en el contexto de la interfaz?
- ¿Son consistentes los submenús con la función implicada por un tema maestro del menú?

Como ya se dijo, en todas las aplicaciones deben establecerse convenciones para el uso de los comandos. Con frecuencia resultan confusos y se facilita que el usuario cometa errores si tiene que escribir alt-D cuando se ha de duplicar un objeto gráfico en una aplicación y alt-D cuando ha de eliminarse en otra. Resulta obvio el potencial que hay para el error.

WebRef

En la dirección **www3.ibm.com/able/guidelines/software/accesssoftware.html** se dan lineamientos para desarrollar software accesible.

Accesibilidad de la aplicación. Conforme las aplicaciones de la computación se hacen ubicuas, los ingenieros de software deben asegurarse de que el diseño de la interfaz incluya mecanismos que permitan el acceso fácil de las personas con necesidades especiales. La *accesibilidad* para los usuarios (e ingenieros de software) que tengan discapacidades físicas es un imperativo por razones éticas, legales y comerciales. Son varios los lineamientos de accesibilidad (por ejemplo [W3C03]) —muchos de ellos diseñados para aplicaciones web pero aplicables con frecuencia a todos los tipos de software— que hacen sugerencias detalladas para el diseño de interfaces que alcancen niveles variables de accesibilidad. Otras (ver [App08], [Mic08]) brindan lineamientos específicos para la "tecnología de ayuda" y se abocan a las necesidades de quienes tienen discapacidades visuales, auditivas, motrices, del habla y de aprendizaje.

Internacionalización. Los ingenieros de software y sus gerentes invariablemente subestiman el esfuerzo y aptitudes que se requieren para crear interfaces de usuario que incluyan las necesidades de lugares e idiomas diferentes. Con demasiada frecuencia, las interfaces se diseñan para una localidad y lenguaje y después se adaptan para funcionar en otros países. El desafío para los diseñadores de interfaces es crear un software "globalizado". Es decir, las interfa-

ces de usuario deben emplearse para que incluyan un núcleo general de funcionalidad que se distribuya a todos aquellos que utilicen el software. Las características de *localización* permiten que la interfaz se personalice para un mercado específico.

Los ingenieros de software disponen de varios lineamientos para la internacionalización (consultar [IBM03]. Éstos abordan aspectos amplios del diseño (por ejemplo, las pantallas difieren en mercados distintos) y de implementación discreta (alfabetos distintos generan requerimientos de escritura y espaciamiento especializados). El estándar *Unicode* [Uni03] se desarrolló para resolver el difícil desafío de manejar decenas de idiomas naturales con cientos de caracteres y símbolos.

HERRAMIENTAS DE SOFTWARE

Desarrollo de la interfaz de usuario

Objetivo: Estas herramientas permiten al ingeniero crear una interfaz de usuario gráfica con relativamente poco desarrollo de software especializado. Las herramientas dan acceso a componentes reutilizables y hacen que la creación de una interfaz se reduzca a seleccionar capacidades predefinidas que se ensamblan con el empleo de la herramienta.

Mecánica: Las interfaces modernas se construyen con el empleo de un conjunto de componentes reutilizables que se acoplan con algunos componentes personalizados desarrollados para obtener características especializadas. La mayor parte de las herramientas para el desarrollo de interfaces de usuario permiten que el ingeniero de software las cree con el empleo de la capacidad de "arrastrar y soltar". Es decir, el desarrollador selecciona entre muchas herramientas predefinidas (como constructores de formas, mecanismos de interacción, capacidad de procesamiento de comandos, etc.) y las coloca dentro del contenido de la interfaz que se va a crear.

Herramientas representativas:[7]

LegalSuite GUI, desarrollada por Seagull Software (**www.seagull-software.com**), permitió la creación de interfaces de usuario gráficas basadas en un navegador y da facilidades para hacer la reingeniería de interfaces anticuadas.

MotifCommon Desktop Environment, desarrollada por The Open Group (**www.osf.org/tech/desktop/cde/**), es una interfaz de usuario gráfica integrada para sistemas abiertos de computación de escritorio. Produce una interfaz gráfica única estandarizada para la administración de datos, archivos (escritorio gráfico) y aplicaciones.

Altia Design 8.0, desarrollada por Altia (**www.altia.com**), es una herramienta para crear interfaces gráficas de usuario en plataformas diferentes (como en un automóvil, portátiles, industriales, etcétera).

11.5 DISEÑO DE UNA INTERFAZ PARA WEBAPPS

Toda interfaz de usuario —diseñada para una *webapp*, aplicación de software tradicional, producto de consumo o dispositivo industrial— debe tener las características de usabilidad que se estudiaron en este capítulo. Dix [Dix99] afirma que debe diseñarse una interfaz de *webapp* de modo que responda tres preguntas principales del usuario final:

Si es probable que los usuarios entren a la webapp por distintos lugares y niveles en la jerarquía del contenido, hay que asegurarse de diseñar cada página con características de navegación que lleven al usuario a otros puntos de interés.

¿Dónde estoy? La interfaz debe: 1) dar una indicación de la *webapp* a la que se ha accedido[8] y 2) informar al usuario de su localización en la jerarquía del contenido.

¿Qué puedo hacer ahora? La interfaz siempre debe ayudar al usuario a entender sus opciones actuales: cuáles funciones están disponibles, qué vínculos están vivos, qué contenido es relevante, etcétera.

¿Dónde he estado, hacia dónde voy? La interfaz debe facilitar la navegación. Para ello, debe disponer un "mapa" (implementado en forma tal que sea fácil de entender) que indique

7 Las herramientas mencionadas aquí no son obligatorias, sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

8 Todos hemos marcado una página web y cuando regresamos tiempo después no tenemos una indicación del sitio web o del contexto para la página (ni la manera de pasar a otra ubicación dentro del sitio).

dónde ha estado el usuario y las trayectorias que pueden tomarse para moverse a cualquier punto de la *webapp*.

Una interfaz eficaz de *webapp* debe dar respuestas a todas estas preguntas cuando el usuario navegue por su contenido y por sus funciones.

11.5.1 Principios y lineamientos del diseño de la interfaz

Cita:

"Si un sitio es perfectamente usable, pero carece de un estilo elegante y apropiado, entonces fallará."

Curt Cloninger

La interfaz de usuario de una *webapp* es la "primera impresión" que se recibe. Sin importar el valor de su contenido, ni la sofisticación de sus capacidades y servicios de procesamiento, así como el beneficio general de la *webapp* en sí, una interfaz mal diseñada decepcionará al usuario potencial y en realidad hará que éste vaya a cualquier otro sitio. Debido al enorme volumen de *webapps* competidoras en virtualmente toda área temática, la interfaz debe "atrapar" de inmediato al usuario potencial.

Bruce Tognozzi [Tog01] define un conjunto de características fundamentales que todas las interfaces deben tener y con ello establece la filosofía que todo diseñador de interfaces de *webapps* debe seguir:

PUNTO CLAVE

Una buena interfaz de *webapps* es entendible y benévola, lo que da al usuario la sensación de tener el control.

> Las interfaces eficaces son atractivas visualmente y perdonan los errores, lo que da a sus usuarios la sensación de tener el control. Los usuarios perciben rápidamente la totalidad de sus opciones, captan cómo lograr sus metas y cómo hacer su trabajo.
>
> Las interfaces eficaces no preocupan al usuario con el funcionamiento interno del sistema. El trabajo se guarda de manera cuidadosa y continua, con opción total para que el usuario deshaga cualquier actividad en cualquier momento.
>
> Las aplicaciones y servicios eficaces realizan un máximo de trabajo, al tiempo que requieren un mínimo de información de parte de los usuarios.

A fin de diseñar interfaces de *webapps* con estas características, Tognozzi [Tog01] identifica un conjunto de principios generales de diseño:[9]

¿Hay un conjunto de principios básicos que se apliquen al diseño de una interfaz de usuario gráfica?

Previsión. *Una webapp debe diseñarse de modo que prevea el siguiente movimiento del usuario.* Por ejemplo, considere una *webapp* de ayuda al cliente desarrollada por un fabricante de impresoras para computadora. Un usuario solicita un objeto de contenido que presenta información sobre el controlador de la impresora para un sistema operativo reciente. El diseñador de la *webapp* debe prever que el usuario tal vez pida descargar el controlador y debe brindar facilidades de navegación que lo permitan, sin requerir que el usuario busque esta capacidad.

Comunicación. *La interfaz debe comunicar el estado de cualquier actividad iniciada por el usuario.* La comunicación puede ser obvia (por ejemplo, un mensaje de texto) o sutil (como la imagen de una hoja de papel que se mueva a través de una impresora para indicar que hay una impresión en curso). La interfaz también debe comunicar el estado del usuario (como su identificación) y su ubicación dentro de la jerarquía del contenido de la *webapp*.

Consistencia. *El uso de controles de navegación, menús, iconos y estética (color, forma y distribución) debe ser consistente en la webapp*. Por ejemplo, si un texto en color azul y subrayado implica un vínculo de navegación, el contenido nunca debe incorporar texto con dichas características que no impliquen un vínculo. Además, un objeto, digamos un triángulo amarillo, que se utilice para indicar un mensaje de precaución antes de que el usuario invoque una función o acción particular, no debe usarse para otros propósitos en ningún otro lugar de la *webapp*. Por

9 Los principios originales de Tognozzi fueron adaptados y ampliados para su uso en este libro. Véase [Tog01] para un análisis más amplio de ellos.

último, toda característica de la interfaz debe responder de manera consistente con las expectativas del usuario.[10]

Autonomía controlada. *La interfaz debe facilitar el movimiento del usuario a través de la webapp, pero lo debe hacer de manera que obligue a respetar las convenciones que se hayan establecido para la aplicación*. Por ejemplo, debe controlarse la navegación hacia partes seguras de la *webapp* por medio de la identificación y clave del usuario, y no debe haber ningún mecanismo de navegación que permita que un usuario evite dichos controles.

Cita:

"El mejor recorrido es aquel con el menor número de pasos. Hay que acortar la distancia entre el usuario y su meta."

Autor desconocido

Eficiencia. *El diseño de la webapp y su interfaz deben optimizar la eficiencia del trabajo del usuario, no la del desarrollador que la diseña y construye ni del ambiente cliente-servidor que la ejecuta.* Tognozzi [Tog01] aborda esto cuando escribe lo siguiente: "Esta sencilla verdad explica por qué es tan importante que todos los involucrados en un proyecto de software aprecien la importancia de hacer que la productividad del usuario sea la meta número uno, y de entender la diferencia vital entre construir un sistema eficiente y dar poder a un usuario eficiente."

Flexibilidad. *La interfaz debe tener flexibilidad suficiente para permitir que algunos usuarios realicen tareas directamente, y que otros exploren la webapp en forma aleatoria.* En cada caso, debe permitir al usuario entender dónde se encuentra y darle la funcionalidad para deshacer errores y volver a trazar trayectorias de navegación mal elegidas.

WebRef

Una búsqueda en web revelará muchas bibliotecas disponibles, por ejemplo, paquetes de aplicaciones en Java, interfaces y clases, en **java.sun.com** o COM, DCOM y Bibliotecas Tipo en **msdn.Microsoft.com**.

Centrarse. *La interfaz de la webapp (y el contenido que presente) debe mantenerse centrada en las tareas en curso del usuario.* En todos los hipermedios, existe la tendencia a llevar al usuario a contenido poco relacionado. ¿Por qué? Porque es muy fácil hacerlo... El problema es que el usuario puede extraviarse con rapidez en muchas capas de información de apoyo y perder de vista el contenido original que buscaba inicialmente.

Ley de Fitt. *"El tiempo para llegar a un objetivo está en función de la distancia que hay hasta él y del tamaño que tenga"* [Tog01]. Con base en un estudio realizado en la década de 1950 [Fit54], la Ley de Fitt "es un método eficaz para modelar movimientos rápidos e intencionados, donde un apéndice (como una mano) comienza en reposo en una posición específica de arranque y vuelve al reposo dentro de un área objetivo" [Zha02]. Si una secuencia de selecciones o entradas estandarizadas (con muchas opciones diferentes dentro de la secuencia) es definida por una tarea de usuario, la primera selección (con el ratón, por ejemplo) debe estar físicamente cerca de la siguiente selección. Por ejemplo, considere una interfaz de la página inicial de una *webapp* en un sitio de comercio electrónico que vende productos de consumo electrónicos.

Cada opción del usuario implica un conjunto de elecciones o acciones por seguir. Por ejemplo, la opción "comprar un producto" requiere que el usuario introduzca una categoría de producto seguida por el nombre de éste. La categoría del producto (equipo de audio, televisiones, reproductores DVD, etc.) aparece como menú desplegable tan pronto como se elige "comprar un producto". Entonces, la elección siguiente resulta obvia de inmediato (está cerca) y el tiempo para llegar a ella es despreciable. Por el contrario, si la elección apareciera en un menú localizado en el otro lado de la pantalla, el tiempo para que el usuario llegue (y luego elija) será mucho más largo.

Objetos de la interfaz humana. *Se ha desarrollado una vasta biblioteca de objetos reutilizables de interfaces humanas para webapps. Úselas.* Cualquier objeto de interfaz que pueda ser "visto, escuchado, tocado o percibido de otro modo" [Tog01] por un usuario final, puede obtenerse de alguna, entre muchas, librerías de objetos.

10 Tognozzi [Tog01] hace la observación de que la única manera de asegurarse de que se entienden bien las expectativas del usuario es realizando exhaustivamente pruebas en las que participe (véase el capítulo 20).

Reducción de la latencia. *En vez de hacer que el usuario espere a que termine alguna operación interna (como descargar una imagen gráfica compleja), la webapp debe usar tareas múltiples, de manera que permita que el usuario continúe con su trabajo mientras finaliza la operación.* Además de reducir la latencia, los retrasos deben explicarse de modo que el usuario entienda lo que esté pasando. Esto incluye: 1) dar retroalimentación auditiva cuando una selección no dé como resultado una acción inmediata por parte de la *webapp*, 2) desplegar un reloj con animación o una barra de avance que indique que hay un procesamiento en marcha y 3) dar alguna distracción (presentación o texto animado) cuando tenga lugar un procesamiento tardado.

Aprendizaje. *Una interfaz de webapp debe diseñarse para minimizar el tiempo de aprendizaje y, una vez aprendida, minimizar el que se dedique a reaprender cuando se regrese a la webapp.* En general, la interfaz debe hacer énfasis en un diseño sencillo e intuitivo que organice el contenido y funcionalidad en categorías que resulten obvias para el usuario.

Las metáforas son una idea excelente porque reflejan la experiencia del mundo real. Sólo hay que asegurarse de que la metáfora elegida resulte bien conocida para los usuarios finales.

Metáforas. *Una interfaz que use una metáfora de interacción es más fácil de aprender y de usar, en la medida en la que la metáfora sea apropiada para la aplicación y el usuario.* Una metáfora debe recurrir a imágenes y conceptos salidos de la experiencia del usuario, pero no necesita ser una reproducción exacta de una experiencia del mundo real. Por ejemplo, un sitio de comercio electrónico que implemente el pago automatizado en una institución financiera, usa una metáfora de chequera (lo que no sorprende) para ayudar al usuario a especificar y programar los pagos de sus facturas. Sin embargo, cuando un usuario "escribe" un cheque, no necesita introducir el nombre completo del beneficiario, sino que elige de una lista de ellos o hace que el sistema lo seleccione con base en las primeras letras que escriba. La metáfora permanece intacta, pero el usuario recibe ayuda de la *webapp*.

Mantener la integridad de los productos del trabajo. *Un producto del trabajo (por ejemplo, un formato llenado por un usuario o una lista especificada por él) debe guardarse en forma automática, de modo que no se pierda si ocurriera un error.* Todos hemos experimentado la frustración que surge cuando al terminar de llenar un formato extenso en una *webapp*, se pierde su contenido debido a un error (cometido por nosotros, por la *webapp* o al transmitirlo del cliente al servidor). Para evitar esto, la *webapp* debe diseñarse para que guarde todos los datos especificados por el usuario. La interfaz debe apoyar esta función y dar al usuario un mecanismo fácil de recuperación de la información "perdida".

Legibilidad. *Toda la información presentada en la interfaz debe ser legible para jóvenes y viejos.* El diseñador de la interfaz debe hacer énfasis en estilos legibles para las letras, en su tamaño y en el color del fondo, que debe contrastar.

Dar seguimiento al estado. *Cuando resulte apropiado, debe darse seguimiento al estado de la interacción del usuario y guardarlo, de modo que éste pueda salir y volver más tarde para recuperarlo de donde lo haya dejado.* En general, las *cookies* pueden diseñarse para que guarden información del estado. Sin embargo, son una tecnología controvertida y para ciertos usuarios resultan más atractivas otras soluciones de diseño.

Navegación visible. *Una interfaz de webapp bien diseñada da "la ilusión de que los usuarios están en el mismo lugar, con el trabajo llevado a ellos"* [Tog01]. Cuando se emplea este enfoque, la navegación no es asunto del usuario. En vez de ello, éste recupera objetos del contenido y selecciona funciones que se despliegan y ejecutan a través de la interfaz.

Nielsen y Wagner [Nie96] sugieren algunos lineamientos prácticos para el diseño de interfaces (basados en su rediseño de una *webapp* importante), que constituyen un buen complemento de los principios ya sugeridos en esta sección:

CasaSegura

Revisión del diseño de la interfaz

La escena: Oficina de Doug Miller.

Participantes: Doug Miller (gerente del grupo de ingeniería de software de *CasaSegura*) y Vinod Raman, miembro del equipo de ingeniería de software del producto de *CasaSegura*.

La conversación:

Doug: Vinod, ¿han podido revisar tú y el equipo el prototipo de la interfaz de comercio electrónico de **CasaSeguraAsegurada.com**?

Vinod: Sí… todos lo vimos desde un punto de vista técnico, y tengo muchas observaciones. Se las envié ayer por correo electrónico a Sharon [gerente del equipo de la *webapp* para la venta externa del sitio web de comercio electrónico de *CasaSegura*].

Doug: Tú y Sharon pueden reunirse y analizar eso… hazme un resumen de los aspectos importantes.

Vinod: Sobre todo, han hecho un buen trabajo, nada grandioso, pero es una interfaz normal de comercio electrónico, estética aceptable, distribución razonable, atendieron todas las funciones importantes…

Doug (sonríe contrito): ¿Pero?

Vinod: Bueno, hay algunas cosas…

Doug: Cómo…

Vinod (presenta a Doug una secuencia de esquemas del prototipo de interfaz): Aquí está el menú de funciones principales que aparece en la página de inicio:

Aprenda sobre *CasaSegura*.

Describa su casa.

Recomendaciones sobre componentes de *CasaSegura*.

Compre un sistema de *CasaSegura*.

Obtenga ayuda técnica.

El problema no está en estas funciones. Están bien, pero el nivel de abstracción no es el correcto.

Doug: Todas son funciones principales, ¿o no?

Vinod: Lo son, pero ahí hay algo… puedes comprar un sistema sin introducir una lista de componentes… no hay una necesidad real de describir la casa si no quieres hacerlo. Sugiero que sólo haya cuatro opciones de menú en la página de inicio.

Aprenda sobre *CasaSegura*.

Especifique el sistema de *CasaSegura* que necesite.

Compre un sistema de *CasaSegura*.

Obtenga ayuda técnica.

Cuando se seleccione **Especifique el sistema de *CasaSegura* que necesite**, entonces se tendrán las siguientes opciones:

Seleccione los componentes de *CasaSegura*.

Recomendaciones sobre componentes de *CasaSegura*.

Si eres un usuario conocedor, seleccionarás los componentes de un conjunto de menús desplegables clasificados en sensores, cámaras, paneles de control, etc. Si necesitaras ayuda, solicitarás una recomendación y eso requerirá que describas tu casa. Creo que es un poco más lógico.

Doug: Estoy de acuerdo. ¿Has hablado con Sharon de esto?

Vinod: No, primero quiero analizar esto con la gente de mercadotecnia; después la llamaré.

Cita:

"Las personas tienen muy poca paciencia con los sitios web mal diseñados."

Jakob Nielsen y Annette Wagner

- La lectura rápida en un monitor de computadora es aproximadamente 25 por ciento más lenta que la que se hace en un papel. Por tanto, no obligue al usuario a leer grandes cantidades de texto, en particular cuando se explique la operación de la *webapp*, o dé ayuda para la navegación.
- Evite los avisos "en construcción": un vínculo innecesario es un camino seguro a la decepción.
- Los usuarios prefieren no desplazar la pantalla. La información importante debe situarse dentro de las dimensiones de una ventana normal de navegación.
- Los menús de navegación y los encabezados deben diseñarse de manera consistente y deben estar disponibles en todas las páginas a las que tenga acceso el usuario. El diseño no debe basarse en funciones del navegador que ayuden a la navegación.
- La estética nunca debe obstaculizar la funcionalidad. Por ejemplo, un solo botón será una mejor opción de navegación que una imagen agradable pero vaga o que un icono cuyo objetivo no esté claro.
- Las opciones de navegación deben ser obvias, aun para el usuario casual. Éste no debe tener que buscar en la pantalla para determinar cómo entrar a otro contenido o servicios.

Una interfaz bien diseñada mejora la percepción que tenga el usuario del contenido o de los servicios provistos por el sitio. No necesita ser espectacular, pero siempre debe estar bien estructurada y con ergonomía apropiada.

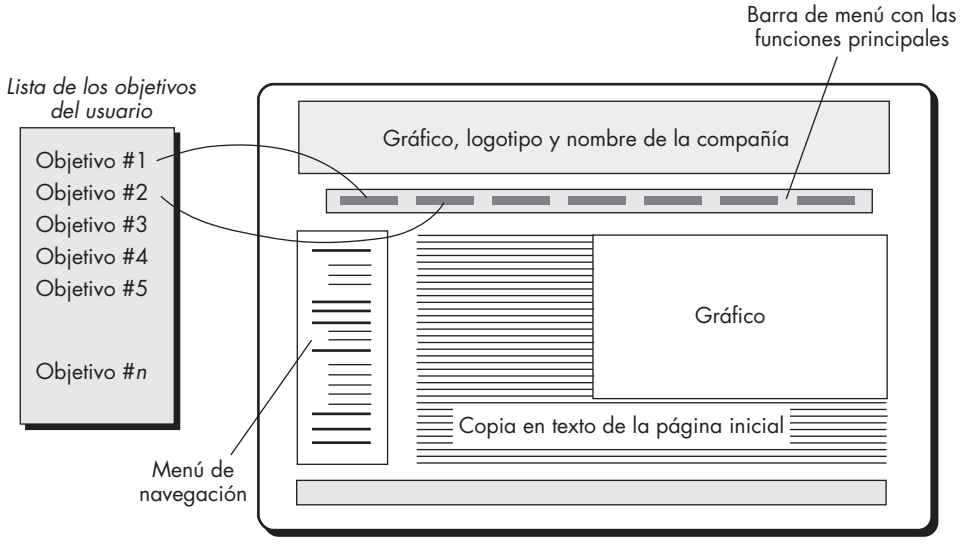
11.5.2 Flujo de trabajos para el diseño de la interfaz de webapp

Al comenzar este capítulo, se dijo que el diseño de la interfaz de usuario comienza con la identificación de los requerimientos del usuario, de la tarea y de los elementos ambientales. Una vez identificadas las tareas del usuario, se crean y analizan los escenarios del usuario (casos de uso) con objeto de definir un conjunto de objetos y acciones de la interfaz.

La información contenida en los formatos del modelo de requerimientos son la base para la creación de una distribución de pantalla que ilustre el diseño gráfico y la ubicación de los iconos, la definición de texto descriptivo en la pantalla, así como la especificación y apilamiento de las ventanas y de los temas mayores y menores del menú. Después se utilizan las herramientas para hacer prototipos e implementar en definitiva el modelo de diseño de la interfaz. Las tareas que siguen representan un flujo de trabajo rudimentario para diseñar una interfaz para *webapp*:

1. **Revisar la información contenida en el modelo de requerimientos y refinarla según se requiera.**
2. **Desarrollar un esquema aproximado de la distribución de la interfaz para la *webapp*.** Como parte de la actividad de modelación de los requerimientos, tal vez se haya desarrollado un prototipo de la interfaz (incluida la distribución). Si ya existe la distribución, debe revisarse y refinarse como se requiera. Si no se hubiera desarrollado la distribución de la interfaz, debe trabajarse con los participantes para hacerlo en este momento. En la figura 11.4 se presenta una primera distribución esquemática.
3. **Mapear los objetivos del usuario en acciones específicas de la interfaz.** Para la gran mayoría de *webapps*, el usuario tendrá un conjunto de objetivos primarios relativamente pequeño. Éstos deben mapearse en acciones específicas de la interfaz, como se muestra en la figura 11.4. En esencia, debe responderse la pregunta siguiente: "¿Cómo hace la interfaz para que el usuario logre cada objetivo?"
4. **Definir un conjunto de tareas de usuario asociadas con cada acción.** Cada acción de la interfaz (por ejemplo, "comprar un producto") se asocia con un conjunto de

FIGURA 11.4
Mapeo de los objetivos del usuario en acciones de la interfaz



tareas de usuario. Éstas se identificaron durante la modelación de los requerimientos. Durante el diseño deben mapearse en interacciones específicas que incluyan los aspectos de navegación, objetos de contenido y funciones de la *webapp*.

5. **Elaborar un guión de las imágenes en la pantalla para cada acción de la interfaz.** A medida que se considera cada acción, debe crearse una secuencia de imágenes del guión (imágenes en la pantalla) a fin de ilustrar la manera en la que responde la interfaz a la interacción con el usuario. Deben identificarse los objetos de contenido (aunque todavía no se hayan diseñado ni desarrollado). Deben mostrarse las funciones de la *webapp* e indicarse los vínculos de navegación.
6. **Refinar la distribución de la interfaz y los guiones con entradas del diseño de la estética.** En la mayor parte de casos, el lector será el responsable de hacer la distribución aproximada y de elaborar los guiones, pero el aspecto estético y la sensación que genere un sitio comercial importante con frecuencia es desarrollada por un artista, más que por profesionales técnicos. El diseño de la estética (véase el capítulo 13) se integra con el trabajo realizado por el diseñador de la interfaz.
7. **Identificar los objetos de la interfaz de usuario requeridos para implementar la interfaz.** Esta tarea quizá requiera buscar en una biblioteca de objetos ya existentes para encontrar aquellos (clases) que sean reutilizables y apropiados para la interfaz de la *webapp*. Además, en este momento se especifican cualesquiera clases personalizadas.
8. **Desarrollar una representación del procedimiento de la interacción del usuario con la interfaz.** Esta tarea opcional utiliza diagramas UML de secuencia o diagramas de actividades (véase el apéndice 1) a fin de ilustrar el flujo de actividades (y decisiones) que tienen lugar cuando el usuario interactúa con la *webapp*.
9. **Desarrollar una representación del comportamiento de la interfaz.** Esta opción de tarea emplea diagramas de estado UML (apéndice 1) para representar transiciones de estado y los eventos que las causan. Se definen los mecanismos de control (tales como los objetos y acciones de que dispone el usuario para modificar el estado de una *webapp*).
10. **Describir la distribución de la interfaz para cada estado.** Con el uso de la información de diseño desarrollada en las tareas 2 y 5, se asocia una distribución específica o imagen de la pantalla a cada estado de la *webapp* descrito en la tarea 8.
11. **Refinar y revisar el modelo del diseño de la interfaz.** La revisión de la interfaz debe centrarse en la usabilidad.

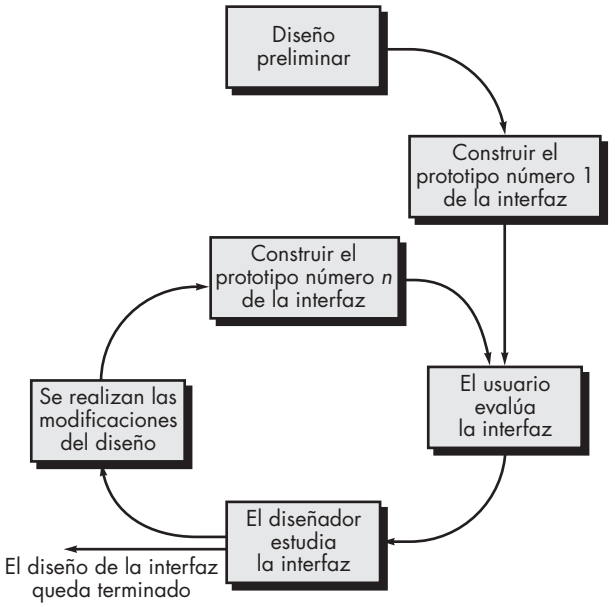
Es importante observar que el conjunto definitivo de tareas que se elijan debe adaptarse a los requerimientos especiales de la aplicación que se va a elaborar.

11.6 Evaluación del diseño

Una vez que se crea un prototipo operativo de la interfaz de usuario, debe evaluarse con objeto de determinar si satisfacen las necesidades de éste. La evaluación abarca un espectro de formalidad que va desde una "prueba de manejo" informal, en la que el usuario da retroalimentación instantánea a un estudio diseñado formalmente que utilice métodos estadísticos para evaluar cuestionarios que respondería una población de usuarios finales.

El ciclo de evaluación de la interfaz de usuario toma la forma que se aprecia en la figura 11.5. Una vez terminado el modelo del diseño, se crea un prototipo de primer nivel. Éste es evaluado

FIGURA 11.5
Ciclo de evaluación del diseño de la interfaz



por el usuario,[11] quien hace comentarios directos acerca de la eficacia de la interfaz. Además, se emplean técnicas formales de evaluación (tales como cuestionarios, hojas de calificación, etc.), de las que se extrae información (por ejemplo: a 80 por ciento de todos los usuarios no le gusta el mecanismo para guardar archivos de datos). Las modificaciones del diseño se hacen con base en las aportaciones de los usuarios, y así se crea el siguiente nivel de prototipo. El ciclo de evaluación continúa hasta que ya no es necesario modificar más el diseño de la interfaz.

El enfoque del prototipo es eficaz, pero ¿es posible evaluar la calidad de una interfaz de usuario antes de que se construya el prototipo? Si se identifican y corrigen a tiempo los problemas potenciales, se reducirá el número de bucles en el ciclo de evaluación y disminuirá el tiempo de desarrollo. Si se hubiera creado un modelo del diseño de la interfaz, se aplicarían los siguientes criterios de evaluación [Mor81] durante las primeras revisiones de éste:

1. La longitud y complejidad del modelo de requerimientos o especificaciones escritas del sistema y su interfaz darán una indicación de la cantidad de aprendizaje requerido por los usuarios del sistema.
2. El número de tareas del usuario especificadas y el número promedio de acciones por tarea indicarán el tiempo de interacción de la eficiencia general del sistema.
3. El número de acciones, tareas y estados del sistema indicados por el modelo del diseño implicarán la carga de memoria para los usuarios del sistema.
4. El estilo de la interfaz, las herramientas de ayuda y el protocolo del manejo de errores darán una indicación general de la complejidad de la interfaz y de su grado de aceptación por parte del usuario.

Una vez construido el prototipo, se reúnen varios datos cualitativos y cuantitativos que ayuden a evaluar la interfaz. Para obtener los datos cualitativos, se distribuyen cuestionarios a los usuarios del prototipo. Las preguntas pueden ser: 1) de respuesta simple, sí o no, 2) de respuesta

11 Es importante observar que los expertos en ergonomía y diseño de interfaces también son quienes revisan éstas. Las revisiones se denominan *evaluaciones heurísticas* o *recorridos cognitivos*.

numérica, 3) de escala (subjetiva), 4) de escalas de aprobación (completamente de acuerdo, de acuerdo), 5) de respuesta porcentual (subjetiva) o 6) de respuesta abierta.

Si se desea obtener datos cuantitativos, puede llevarse a cabo alguna forma de estudio de tiempos. Se observa a los usuarios durante la interacción y se obtienen datos —tales como número de tareas terminadas correctamente en un periodo de tiempo estándar, frecuencia de las acciones, secuencia de éstas, tiempo dedicado a "mirar" la pantalla, número y tipos de errores, tiempo de recuperación del error, tiempo dedicado a usar la ayuda y número de referencias por periodo de tiempo estándar— que se utilizan como guía para la modificación de la interfaz.

El análisis completo de los métodos de evaluación de la interfaz de usuario queda más allá del alcance de este libro. Para mayor información, consulte a [Hac98] y [Sto05].

11.7 Resumen

La interfaz de usuario es presumiblemente el elemento más importante de un sistema o producto basado en computadora. Si la interfaz estuviera mal diseñada, afectaría mucho la capacidad del usuario de aprovechar el poder computacional y el contenido de información de una aplicación. En realidad, una interfaz defectuosa haría que fallara incluso una aplicación bien diseñada y con buena implementación.

Son tres los principios importantes que guían el diseño de interfaces eficaces: 1) dar el control al usuario, 2) reducir la memorización del usuario y 3) hacer que la interfaz sea consistente. Para lograr que una interfaz cumpla estos principios, debe llevarse a cabo un proceso de diseño bien organizado.

El desarrollo de una interfaz de usuario comienza con una serie de tareas de análisis. El análisis del usuario define los perfiles de distintos usuarios finales y proviene de varias fuentes comerciales y técnicas. El análisis de la tarea define las tareas y acciones del usuario por medio de un enfoque de elaboración o bien otro orientado a objetos, la aplicación de casos de uso, elaboración de tareas y objetos, análisis del flujo de trabajo y representaciones jerárquicas de la tarea para entender bien la interacción humano-computadora. El análisis ambiental identifica las estructuras físicas y sociales en las que debe operar la interfaz.

Una vez definidas las tareas, se crean y analizan escenarios de usuario para definir un conjunto de objetos y acciones de la interfaz. Esto da una base para la creación de la distribución de la pantalla que ilustre el diseño gráfico y la colocación de los iconos, la definición de un texto descriptivo en la pantalla, la especificación y apilamiento de las ventanas y la especificación de los temas principales y secundarios del menú. A medida que se refina el modelo del diseño, se consideran aspectos tales como tiempo de respuesta, estructura de comandos y acciones, manejo de errores y herramientas de ayuda. Para construir un prototipo a fin de que lo evalúe el usuario, se utilizan varias herramientas de implementación.

Igual que el diseño de la interfaz para el software convencional, el correspondiente a *webapps* describe la estructura y organización de la interfaz de usuario e incluye una representación de la distribución de la pantalla, la definición de los modos de interacción y la descripción de mecanismos de navegación. Un conjunto de principios de diseño de la interfaz y del flujo de los trabajos respectivos guía al diseñador de *webapp* cuando hace la distribución y cuando diseña los mecanismos de control de la interfaz.

La interfaz de usuario es la ventana hacia el software. En muchos casos, moldea la percepción del usuario respecto de la calidad del sistema. Si la "ventana" está manchada, ondulante o rota, el usuario puede rechazar un sistema basado en computadora que, en lo demás, sería poderoso.

Problemas y puntos por evaluar

11.1. Describa la peor interfaz con la que haya trabajado y haga una crítica de los conceptos introducidos en este capítulo. Describa la mejor interfaz con la que haya laborado y critíquela respecto de los conceptos estudiados aquí.

11.2. Desarrolle dos principios de diseño adicionales que "den el control al usuario".

11.3. Elabore dos principios de diseño adicionales que "reduzcan la necesidad de memorización por parte del usuario".

11.4. Enuncie otros dos principios de diseño que "hagan consistente a la interfaz".

11.5. Considere una de las siguientes aplicaciones interactivas (o una aplicación asignada por su profesor):

a) Sistema de publicación de escritorio
b) Sistema de diseño asistido por computadora
c) Sistema de diseño de interiores (como el descrito en la sección 11.3.2)
d) Sistema automatizado de inscripción a materias para una universidad
e) Sistema de administración de biblioteca
f) Encuestas basadas en internet para elecciones públicas
g) Sistema de banca en casa
h) Aplicación interactiva asignada por su profesor

Desarrolle los modelos de usuario, del diseño, mental y de implementación para cualquiera de dichos sistemas.

11.6. Haga el análisis detallado de las tareas para alguno de los sistemas enlistados en el problema 11.5. Utilice un enfoque de elaboración u orientado a objetos.

11.7. Agregue al menos cinco preguntas adicionales a la lista desarrollada para el análisis del contenido de la sección 11.3.3.

11.8. Continúe el problema 11.5 y defina los objetos y acciones de la interfaz para la aplicación que haya elegido. Identifique cada tipo de objeto.

11.9. Desarrolle un conjunto de distribuciones de pantalla con la definición de los temas principales y secundarios del menú para el sistema que haya escogido en el problema 11.5.

11.10. Desarrolle varias distribuciones de pantalla con la definición de los temas de menú principales y secundarios para el sistema *CasaSegura*. Puede elegir un enfoque distinto del que se muestra en la figura 11.3.

11.11. Describa su enfoque de las herramientas de ayuda para el usuario a fin de hacer el modelo del diseño del análisis de tareas y el análisis de la tarea que haya realizado como parte de los problemas 11.5 a 11.8.

11.12. Dé algunos ejemplos que ilustren por qué la variabilidad del tiempo de respuesta llega a ser importante.

11.13. Desarrolle un enfoque que integre de manera automática los mensajes de error y las herramientas de ayuda para el usuario. Es decir, el sistema reconocerá automáticamente el tipo de error y dará una ventana de ayuda con sugerencias para corregirlo. Realice un diseño del software razonablemente completo que tome en cuenta las estructuras de los datos y algoritmos.

11.14. Elabore un cuestionario de evaluación de la interfaz que contenga 20 preguntas generales que se apliquen a la mayoría de interfaces. Haga que lo respondan 10 de sus compañeros para un sistema interactivo que todos utilicen. Resuma los resultados y haga un informe para su grupo.

Lecturas adicionales y fuentes de información

Aunque este libro no trata específicamente las interfaces humano-computadora, gran parte de lo que afirma Donald Norman (*The Design of Everyday Things*, reimpresión, Currency/Doubleday, 1990) sobre la psicología del diseño eficaz se aplica a la interfaz de usuario. Se recomienda su lectura a cualquier persona que piense hacer el diseño de una interfaz de alta calidad.

Las interfaces de usuario gráficas son ubicuas en el mundo moderno de la computación. Se trate de un cajero automático, teléfono inalámbrico, tablero electrónico de un automóvil, sitio web o aplicación de negocios, la interfaz de usuario constituye una ventana hacia el software. Por esta razón, abundan los libros que se abocan al diseño de la interfaz. Butow (*User Interface Design for Mere Mortals*, Addison-Wesley, 2007), Galitz (*The Essential Guide to User Interface Design*, 3a. ed., Wiley, 2007), Lehikonen *et al.* (*Personal Content Experience: Managing Digital Life in the Mobile Age*, Wiley-Interscience, 2007), Cooper *et al.* (*About Face 3: The Essentials of Interaction Design*, 3a. ed., Wiley, 2007), Ballard (*Designing the Mobile User Experience*, Wiley, 2007), Nielsen (*Coordinating User Interfaces for Consistency*, Morgan-Kauffamn, 2006), Lauesen (*User Interface Design: A Software Engineering Perspective*, Addison-Wesley, 2005), Barfield (*The User Interface: Concepts and Design*, Bosko Books, 2004) son textos que analizan la usabilidad, los conceptos de la interfaz de usuario, principios y técnicas de diseño, y que también contienen ejemplos útiles.

Los libros más antiguos de Beyer y Hontzblatt (*Contextual Design: A Customer Centered Approach to Systems Design*, Morgan-Kauffman, 2002), Raskin (*The Human Interface*, Addison-Wesley, 2000), Constantine y Lockwood (*Software for Use*, ACM Press, 1999) y Mayhew (*The Usability Engineering LifeCycle*, Morgan-Kauffman, 1999) presentan análisis que dan lineamientos y principios de diseño adicionales, así como sugerencias para recabar requerimientos de la interfaz, modelación de su diseño, implementación y prueba.

Johnson (*GUI Bloopers: Don'ts and Do's for Software Developers and Web Designers*, Morgan-Kauffman, 2000) ofrece una guía útil para aquellos que aprenden mejor con el estudio de contraejemplos. Un libro ameno escrito por Cooper (*The Inmates Are Running the Asylum*, Sams Publising, 1999) analiza por qué los productos de alta tecnología nos desconciertan y cómo diseñarlos para que no lo hagan.

Existe una amplia variedad de fuentes de información sobre el diseño de la interfaz disponibles en internet. En el sitio web del libro, hay una lista actualizada de referencias en la red mundial que son relevantes para el diseño de la interfaz de usuario; la dirección es: **www.mhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

CAPÍTULO

12

DISEÑO BASADO EN PATRONES

CONCEPTOS CLAVE

errores de diseño 305
estructuras 299
fuerzas. 296
granularidad 314
interfaz de usuario 310
lenguajes del patrón 300
patrones
arquitectónicos 306
conductuales 298
creacional 298
estructural 298
generativo 297
nivel de componentes . . . 308
webapps 313

Todos hemos encontrado un problema de diseño y pensamos en silencio: *me pregunto si alguien ha desarrollado una solución para esto...* La respuesta casi siempre es *sí*. El problema es encontrarla; luego, estar seguro de que en verdad se ajusta al problema en cuestión, entender las limitaciones que restringen su aplicación y, por último, traducir la solución propuesta al ambiente de diseño.

Pero, ¿qué pasaría si la solución estuviera codificada de algún modo? ¿Qué ocurriría si hubiera una manera estandarizada de describir un problema (de modo que destacara) y un método organizado para representar su solución? Con el empleo de un formato estándar, puede observarse que los problemas de software han sido codificados y descritos, igual que las soluciones (y sus restricciones) propuestas. Este método codificado se denomina *patrones de diseño*, se emplea para describir problemas y sus soluciones y permite que la comunidad de la ingeniería de software aborde el conocimiento de diseño en forma tal que es posible reutilizarlo.

La historia de los patrones de software no comienza con un científico de la computación, sino con un arquitecto constructor, Christopher Alexander, quien reconoció que siempre que se diseñaba un edificio era reconocible un conjunto de problemas recurrentes. Definió éstos y sus soluciones como *patrones*, y los describió del modo siguiente [Ale77]:

> Cada patrón describe un problema que ocurre una y otra vez en nuestro ambiente, y luego describe el núcleo de su solución en forma tal que es posible usarla un millón de veces sin elaborarla dos veces de la misma forma.

UNA MIRADA RÁPIDA

¿Qué es? El diseño basado en patrones crea una aplicación nueva, encontrando un conjunto de soluciones comprobadas para un conjunto de problemas delineado con claridad. Cada problema y su solución está descrito por un patrón de diseño catalogado y analizado por otros ingenieros de software que han encontrado el problema e implantado su solución cuando diseñaban otras aplicaciones. Cada patrón de diseño provee un enfoque demostrado para una parte del problema que debe resolverse.

¿Quién lo hace? Un ingeniero de software estudia cada problema hallado para una nueva aplicación y después trata de encontrar una solución relevante, buscando en un depósito de patrones.

¿Por qué es importante? ¿Ha escuchado el lector la frase "reinventar la rueda"? Eso pasa todo el tiempo en el desarrollo de software y representa una pérdida de tiempo y energía. Al utilizar patrones de diseño existentes, se adquiere una solución probada para un problema específico. A medida que se aplica cada patrón, las soluciones se integran y la aplicación que se va a elaborar se acerca más al diseño final.

¿Cuáles son los pasos? El modelo de requerimientos se estudia con objeto de despejar el conjunto jerárquico de problemas por resolver. Se divide el espacio de problemas de modo que sea posible identificar subconjuntos de problemas asociados con funciones específicas del software. Los problemas también pueden organizarse por tipo: arquitectónicos, en el nivel de componentes, algorítmicos, de interfaz de usuario, etcétera. Una vez definido un subconjunto de problemas, se busca uno o más depósitos de patrones a fin de determinar si existe un patrón de diseño previo representado en un nivel de abstracción apropiado. Los patrones aplicables se adaptan a las necesidades específicas del software que se va a elaborar. La solución específica de problemas se aplica en situaciones en las que no se detectan patrones.

¿Cuál es el producto final? Se desarrolla un modelo del diseño que ilustra la arquitectura del software, la interfaz de usuario y los detalles en el nivel de componentes.

¿Cómo me aseguro de que lo hice bien? A medida que cada patrón de diseño se traduce en cierto elemento del modelo del diseño, se revisan los productos del trabajo respecto de su claridad, corrección, completitud y consistencia con los requerimientos y con los demás patrones.

Las ideas de Alexander fueron traducidas por vez primera al mundo del software en los libros de Gamma [Gam95], Buschmann [Bus96] y muchos de sus colegas.[1] Actualmente, existen decenas de depósitos de patrones y el diseño basado en patrones se emplea en muchos dominios diferentes de aplicación.

12.1 Patrones de diseño

PUNTO CLAVE

Las fuerzas son aquellas características del problema y los atributos de la solución que restringen la forma en la que puede desarrollarse el diseño.

Un *patrón de diseño* se caracteriza como "una regla de tres partes que expresa una relación entre cierto contexto, un problema y una solución" [Ale79]. Para el diseño de software, el *contexto* permite al lector entender el ambiente en el que reside el problema y qué solución sería apropiada en dicho ambiente. Un conjunto de requerimientos, incluidas limitaciones y restricciones, actúan como *sistema de fuerzas* que influyen en la manera en la que puede interpretarse el problema en este contexto y en cómo podría aplicarse con eficacia la solución.

Para entender mejor estos conceptos, piense en una situación[2] en la que una persona debe viajar entre Nueva York y Los Ángeles. En dicho contexto, el viaje tendrá lugar dentro de un país industrializado (Estados Unidos), con el empleo de una infraestructura de transporte (como carreteras, líneas aéreas y ferrocarriles). El sistema de fuerzas que afecta la forma en la que se resuelve el problema de transporte incluye lo siguiente: cuán rápido quiere ir la persona desde Nueva York hasta Los Ángeles, si el viaje incluye paradas en miradores, cuánto dinero puede gastar la persona, si el viaje está previsto para lograr un propósito específico y los vehículos personales que tiene a su disposición el viajero. Dadas estas fuerzas, puede definirse mejor el problema (viajar de Nueva York a Los Ángeles). Por ejemplo, las investigaciones (recabar requerimientos) indican que la persona tiene muy poco dinero, posee sólo una bicicleta (y es un ciclista entusiasta), quiere hacer el viaje a fin de reunir dinero para sus obras de caridad favoritas y dispone de mucho tiempo. La solución al problema, dado el contexto y el sistema de fuerzas, es un viaje en bicicleta a campo traviesa. Si las fuerzas fuesen distintas (debe minimizarse el tiempo de recorrido y su propósito es asistir a una reunión de negocios), será más apropiada otra solución.

Cita:

"Nuestra responsabilidad es hacer lo que podamos, aprender lo que se pueda, mejorar las soluciones y transmitirlas."

Richard P. Feynman

Es razonable afirmar que la mayoría de problemas tienen muchas soluciones, pero sólo es eficaz aquella que resulta apropiada en el contexto del problema existente. Es el sistema de fuerzas el que hace que un diseñador elija una solución específica. El objetivo es proporcionar la que satisfaga mejor al sistema de fuerzas, aun cuando éstas sean contradictorias. Por último, toda solución tiene consecuencias que afectan otros aspectos del software y que se vuelven parte del sistema de fuerzas de otros problemas por resolver en el sistema mayor.

Coplien [Cop05] caracteriza un patrón de diseño eficaz del modo siguiente:

- *Resuelve un problema:* los patrones entrañan soluciones, no sólo principios o estrategias abstractas.
- *Es un concepto probado:* los patrones incluyen soluciones con un historial, no teorías o especulaciones.
- *La solución no es obvia:* muchas técnicas de solución de problemas (como los paradigmas o métodos de diseño de software) tratan de obtener soluciones a partir de sus principios originales. Los mejores patrones *generan* indirectamente una solución a un problema, enfoque necesario para los problemas más difíciles del diseño.
- *Describe una relación:* los patrones no sólo describen módulos, sino estructuras y mecanismos más profundos del sistema.

1 Existen análisis anteriores de los patrones de software, pero estos dos libros clásicos fueron los primeros que dieron un tratamiento cohesivo al tema.

2 Este ejemplo se adaptó de [Cor98].

- *El patrón tiene un componente humano significativo (minimiza la intervención humana).* Todo el software sirve para el confort humano o la calidad de vida; los mejores patrones recurren explícitamente a la estética y a la utilidad.

Dicho en forma más clara, un buen patrón de diseño incorpora el conocimiento de diseño pragmático, ganado con dificultad, en una forma que permite que otros lo reutilicen "un millón de veces sin elaborarla dos veces de la misma forma". Un patrón de diseño evita "reinventar la rueda" o, peor aún, inventar una "nueva rueda" que sea un poco menos redonda, demasiado pequeña para el uso que se pretende y muy angosta para el terreno en el que rodará. Si se usan con eficacia, los patrones de diseño invariablemente harán del lector un buen diseñador de software.

12.1.1 Clases de patrones

Una de las razones por las que los ingenieros de software están interesados (e intrigados) por los patrones de diseño es que los seres humanos son inherentemente buenos para reconocer patrones. Si no fuera así, estaríamos congelados en el tiempo y el espacio: seríamos incapaces de aprender de nuestras experiencias, sin voluntad para ir más lejos debido a que nuestra incapacidad de reconocer situaciones nos haría correr grandes riesgos, estaríamos desarticulados por un mundo que parecería no tener regularidades ni consistencia lógica. Por suerte, nada de esto sucede debido a que reconocemos patrones en virtualmente todos los aspectos de nuestras vidas.

En el mundo real, los patrones que reconocemos los aprendemos en el tiempo que dura nuestra vida. Los reconocemos al instante y comprendemos inherentemente lo que significan y cómo podemos usarlos. Algunos de estos patrones nos permiten detectar fenómenos recurrentes. Por ejemplo, imagine que va camino a casa desde su trabajo, por la autopista, cuando su sistema de navegación (o el radio) le informa que en dirección opuesta ocurrió un serio accidente. Se encuentra a siete kilómetros del accidente, pero ya vio que el tráfico se hace lento, reconoce un patrón que llamamos **CuellodeBotella**. Las personas que están en los carriles en la dirección que sigue usted disminuyen su velocidad para ver lo que pasó en el lado opuesto de la carretera. El patrón **CuellodeBotella** produce resultados predecibles (embotellamiento), pero no hace nada más que describir un fenómeno. En el habla de los patrones, éste sería un patrón *no generativo* debido a que describe un contexto y un problema, pero no ofrece ninguna solución clara.

PUNTO CLAVE

Un patrón "generativo" describe el problema, el contexto y las fuerzas, y también una solución práctica para el problema.

Cuando se toman en cuenta los patrones de diseño de software, se intenta identificar los patrones *generativos*. Es decir, se identifica un patrón que describa un aspecto importante y repetitivo de un sistema, y que provea una manera de construir dicho aspecto dentro de un sistema de fuerzas que son únicas en un contexto determinado. Idealmente, podría usarse un conjunto de patrones de diseño generativos para "generar" una aplicación o sistema basado en computadora cuya arquitectura permita adaptarlo al cambio. En ocasiones se llama *generatividad* a "la aplicación sucesiva de varios patrones, cada uno de los cuales incluye su propio problema y fuerzas, y que despliega una solución más grande que emerge indirectamente como resultado de soluciones más pequeñas" [App00].

Los patrones de diseño abarcan un amplio espectro de abstracción y aplicaciones. Los *patrones arquitectónicos* describen problemas de diseño de base amplia que se resuelven con el empleo de un enfoque estructural. Los *patrones de datos* describen problemas recurrentes orientados a datos y las soluciones de modelado de datos que pueden emplearse para resolverlos. Los *patrones de componentes* (también llamados *patrones de diseño*) se enfocan a problemas asociados con el desarrollo de subsistemas y componentes, así como a la manera en la que se comunican entre sí y su ubicación dentro de una arquitectura mayor. Los *patrones de diseño de la interfaz* describen problemas comunes de interfaz de usuario y su solución con un sistema de fuerzas que incluye las características específicas de los usuarios finales. Los *patrones de webapp*

enfrentan un conjunto de problemas que surgen cuando se elaboran *webapps* y es frecuente que incorporen muchas de las otras categorías de patrones mencionadas. En un nivel de abstracción más bajo, los *idiomas* describen la forma de implementar todo un algoritmo específico o una parte de él, o bien una estructura de datos, para un componente de software en el contexto de un lenguaje de programación específico.

¿Hay alguna forma de clasificar los tipos de patrones?

En su libro fundamental sobre patrones de diseño, Gamma *et al.*[3] [Gam95] se centran en tres tipos de patrones de relevancia especial para el diseño orientado a objetos: patrones creacionales, estructurales y conductuales.

Los *patrones creacionales* se centran en la "creación, composición y representación" de objetos. Gamma *et al.* [Gam95] hacen la observación de que los patrones creacionales "encierran conocimiento acerca de cuáles son las clases concretas que usa el sistema", pero al mismo tiempo "ocultan la forma en la que las instancias de dichas clases se crean y agrupan". Los patrones creacionales ofrecen mecanismos que hacen más fácil la formación de las instancias de los objetos dentro de un sistema y establecen "restricciones en el tipo y número de objetos que es posible crear dentro de un sistema" [Maa07].

INFORMACIÓN

Patrones creacionales, estructurales y conductuales

En la red mundial hay una amplia variedad de patrones de diseño que se ajustan a las categorías creacional, estructural y conductual. En Wikipedia (**www.wikipedia.org**) se encuentran los siguientes ejemplos:

Patrones creacionales

- **Patrón de fábrica abstracta:** centraliza la decisión acerca de para qué fábrica deben hacerse instancias.
- **Patrón de método de fabricación:** centraliza la creación de un objeto de tipo específico para elegir una entre varias implementaciones.
- **Patrón constructor:** separa la construcción de un objeto complejo a partir de su representación, de modo que el mismo proceso de construcción pueda crear representaciones distintas.
- **Patrón prototipo:** se usa cuando el costo inherente que implica crear un nuevo objeto en la forma estándar (como con el empleo de una "nueva" clave) es prohibitivo para una aplicación dada.
- **Patrón de instancia única:** restringe la formación de instancias de una clase a un objeto.

Patrones estructurales

- **Patrón adaptador:** "adapta" una interfaz para una clase en otra que espera un cliente.
- **Patrón agregado:** es una versión del patrón compuesto con métodos para agregar hijos.
- **Patrón de puente:** desacopla una abstracción de su implementación, de modo que las dos puedan variar en forma independiente.
- **Patrón compuesto:** estructura de árbol de objetos en los que cada uno tiene la misma interfaz.
- **Patrón contenedor:** crea objetos con el único propósito de que sostengan a otros y los administren.
- **Patrón de proximidad:** clase que funciona como interfaz respecto de otra cosa.
- **Tubos y filtros:** cadena de procesos en los que la salida de cada uno es la entrada del siguiente.

Patrones conductuales

- **Cadena de patrones de responsabilidad:** objetos de comando que son manejados o pasados a otros objetos por medio de otros que contienen procesamiento lógico.
- **Patrón de comando:** objetos de comando que encierran una acción y sus parámetros.
- **Escucha de eventos:** se distribuyen datos a objetos registrados para recibirlos.
- **Patrón intérprete:** implementa un lenguaje de computadora especializado para resolver con rapidez un conjunto específico de problemas.
- **Patrón iterador:** los iteradores se utilizan para acceder en forma secuencial a los elementos de un agregado sin exponer su representación subyacente.
- **Patrón mediador:** proporciona una interfaz unificada a un conjunto de interfaces en un subsistema.
- **Patrón visitante:** forma de separar un algoritmo de un objeto.
- **Patrón visitante de un solo servicio:** optimiza la implementación de un visitante que se haya asignado, utilizado sólo una vez y luego eliminado.
- **Patrón visitante jerárquico:** brinda una forma de visitar cada nodo en una estructura jerárquica de datos, como un árbol.

Las descripciones detalladas de estos patrones pueden obtenerse por medio de vínculos en **www.wikipedia.org**.

3 En la bibliografía sobre patrones, Gamma *et al.* [Gam95] son llamados con frecuencia "la banda de los cuatro" (GoF, por sus siglas en inglés).

Los *patrones estructurales* se centran en problemas y soluciones asociados con la manera en la que se organizan e integran las clases y objetos para construir una estructura más grande. En esencia, ayudan a establecer relaciones entre entidades dentro de un sistema. Por ejemplo, los patrones estructurales que se centran en aspectos orientados a clases proporcionan mecanismos de herencia que conducen a interfaces de programa más eficaces. Los patrones estructurales que se centran en objetos sugieren técnicas para combinar objetos dentro de otros objetos o para integrarlos en una estructura más amplia.

Los *patrones conductuales* se enfocan a problemas asociados con la asignación de responsabilidad entre los objetos y a la manera en la que se efectúa la comunicación entre ellos.

12.1.2 Estructuras

Una estructura es una "miniarquitectura" reutilizable que sirve como base desde la que se pueden aplicar otros patrones de diseño.

Los patrones en sí mismos podrían no ser suficientes para desarrollar un diseño completo. En ciertos casos, tal vez sea necesario dar el esqueleto de una infraestructura específica para la implementación, llamada *estructura*, para el trabajo de diseño. Es decir, puede seleccionarse una "*miniarquitectura reutilizable* que provea la estructura y comportamiento generales para una familia de abstracciones de software, así como un contexto [...] que especifique su colaboración y uso en un dominio determinado" [Amb98].

Una estructura no es un patrón arquitectónico, sino un esqueleto con varios "puntos de conexión" (también llamados *ganchos* o *ranuras*) que permiten adaptarlo a un dominio de problema específico. Los puntos de conexión permiten integrar clases o funciones específicas de un problema dentro del esqueleto. En un contexto orientado a objetos, una estructura es un conjunto de clases que cooperan.

Gamma *et al.* [Gam95] describen de la manera siguiente las diferencias entre los patrones de diseño y las estructuras:

1. *Los patrones de diseño son más abstractos que las estructuras.* Las estructuras están incrustadas en el código, pero en éste sólo es posible incrustar ejemplos de patrones. Una ventaja de las estructuras es que se escriben en lenguajes de programación y no sólo son estudiadas, sino ejecutadas y reutilizadas directamente [...]
2. *Los patrones de diseño son elementos arquitectónicos más pequeños que las estructuras.* Una estructura normal contiene varios patrones de diseño, pero lo contrario nunca se cumple.
3. *Los patrones de diseño están menos especializados que las estructuras.* Las estructuras siempre tienen un dominio particular de aplicación. En contraste, los patrones de diseño se usan en casi cualquier tipo de aplicación. Si bien es posible tener patrones de diseño más especializados, incluso éstos no imponen la arquitectura de una aplicación.

En esencia, el diseñador de una estructura afirmará que una miniestructura reutilizable es aplicable a todo software por desarrollarse en un dominio limitado de aplicación. Para ser más eficaces, las estructuras se aplican sin cambio. Pueden agregarse otros elementos de diseño, pero sólo a través de los puntos de conexión que permiten que el diseñador modifique el esqueleto de la estructura.

12.1.3 Descripción de un patrón

El diseño basado en patrones comienza con el reconocimiento de patrones en la aplicación que se trata de construir, continúa con una búsqueda para determinar si otros han usado el patrón y termina con la aplicación de un patrón apropiado para el problema de que se trate. Es frecuente que la segunda sea la tarea más difícil. ¿Cómo se encuentran patrones que se ajusten a las necesidades?

Una respuesta a esta pregunta debe basarse en la comunicación eficaz del problema al que se dirige el patrón, el contexto en el que reside ése, el sistema de fuerzas que moldean el con-

texto y la solución propuesta. Para comunicar esta información sin ambigüedades, se requiere un formato o plantilla estándar del documento. Aunque se han propuesto varios formatos diferentes de patrones, casi todos contienen un conjunto importante del contenido sugerido por Gamma *et al.* [Gam95]. En el recuadro se presenta un formato simplificado de patrón.

INFORMACIÓN

Formato de diseño del patrón

Nombre del patrón: describe la esencia del patrón con un nombre corto pero expresivo
Problema: describe el problema al que se dirige el patrón
Motivación: proporciona un ejemplo del problema
Contexto: describe el ambiente en el que reside el problema, incluido el dominio de aplicación
Fuerzas: lista el sistema de fuerzas que afectan la manera en la que debe resolverse el problema; incluye el análisis de las limitaciones y restricciones que deben ser tomadas en cuenta
Solución: hace la descripción detallada de la solución propuesta para el problema
Objetivo: describe el patrón y lo que hace
Colaboraciones: describe la manera en la que otros patrones contribuyen a la solución
Consecuencias: describe los intercambios potenciales que deben de ser considerados cuando se implementa el patrón, y las consecuencias de usar éste
Implementación: identifica los aspectos especiales que deben considerarse cuando se implemente el patrón
Usos conocidos: da ejemplos de usos reales del patrón de diseño en aplicaciones reales
Patrones relacionados: menciona referencias de patrones de diseño relacionados

Cita:

"Los patrones siempre están a medio cocinar, lo que significa que siempre deben ser terminados y adaptados al ambiente específico por el usuario."

Martin Fowler

Los nombres de los patrones de diseño deben escogerse con cuidado. Uno de los problemas técnicos clave en el diseño basado en patrones es la incapacidad de encontrar los ya existentes entre cientos o miles de candidatos. La búsqueda del patrón "correcto" se simplifica muchísimo con un nombre significativo para el patrón.

El formato del patrón es un medio estandarizado para describir un patrón de diseño. Cada una de sus entradas representa características del patrón de diseño que pueden ser buscadas (en una base de datos, por ejemplo) a fin de encontrar el que sea apropiado.

12.1.4 Lenguajes y repositorios de patrones

Cuando se utiliza el término *lenguaje*, lo primero que viene a la mente es un lenguaje natural (como inglés, español o chino) o uno de programación (como C++ o Java). En ambos casos, el lenguaje tiene una sintaxis y semántica que se utiliza para comunicar ideas o instrucciones de procedimiento en forma eficaz.

Cuando se emplea el término *lenguaje* en el contexto de los patrones de diseño, adopta un significado un poco distinto. Un *lenguaje de patrón* agrupa un conjunto de patrones, cada uno de los cuales se describe con el uso de un formato estandarizado (véase la sección 12.1.3) e interrelacionado para mostrar cómo colaboran los patrones para resolver problemas en un dominio de aplicación.[4]

En un lenguaje natural, las palabras están organizadas en oraciones que dan el significado. La estructura de las oraciones es descrita por la sintaxis del lenguaje. En un lenguaje de patrón, los patrones de diseño están organizados en forma tal que proporcionan un "método estructurado para describir las buenas prácticas de diseño en un dominio particular".[5]

4 Christopher Alexander originalmente propuso lenguajes de patrón para construir arquitecturas y hacer planeación urbana. Actualmente, se han desarrollado lenguajes de patrón para todo, desde las ciencias sociales hasta proceso de la ingeniería de software.

5 Esta descripción es de Wikipedia y se encuentra en la dirección http://en.wikipedia.org/wiki/Pattern_language.

CONSEJO

Si no puede encontrar un lenguaje de patrones que se adapte al dominio de su problema, busque analogías en otro conjunto de patrones.

En cierta forma, un lenguaje de patrones es análogo a un manual de instrucciones de hipertexto para resolver problemas en un dominio específico de aplicaciones. El dominio del problema en cuestión primero se describe de manera jerárquica, comenzando con problemas de diseño amplio asociados con el dominio, y luego se refina cada uno de ellos en niveles de abstracción más bajos. En un contexto de software, los problemas de diseño amplio tienden a ser de naturaleza arquitectónica y se abocan a la estructura general de la aplicación y a los datos o contenido que le dan servicio. Los problemas arquitectónicos se mejoran hacia niveles más bajos de abstracción, lo que conduce a patrones de diseño que resuelven los subproblemas y que colaboran entre sí en el nivel de componentes (o clases). En vez de que un lenguaje de patrones represente una lista secuencial de patrones, lo hace con un conjunto interconectado en el que el usuario comienza con un problema de diseño amplio y "lo presta" a problemas específicos no descubiertos, así como a sus soluciones.

WebRef

Para ver una lista de lenguajes de patrones útiles, consulte la dirección **c2.com/ppr/titles.html**. Además, en **hillside.net/patterns/** se encuentra información adicional al respecto.

Se han propuesto decenas de lenguajes de patrones para el diseño de software [Hil08]. En la mayor parte de casos, los patrones de diseño que forman parte de un lenguaje de patrones se almacenan en un repositorio de patrones accesible a través de la web (consulte [Boo08], [Cha03] y [HPR02]). El repositorio proporciona un índice de todos los patrones de diseño y contiene vínculos de hipermedios que permiten al usuario entender las colaboraciones entre patrones.

12.2 DISEÑO DE SOFTWARE BASADO EN PATRONES

Los mejores diseñadores en cualquier campo tienen una aptitud asombrosa para ver los patrones que caracterizan un problema y los que pueden combinarse para generar una solución. Los desarrolladores de software de Microsoft [Mic04] lo dicen del siguiente modo:

> Si bien el diseño basado en patrones es relativamente nuevo en el campo del desarrollo de software, la tecnología industrial lo ha utilizado durante décadas, quizá siglos. Los catálogos de mecanismos y configuraciones estándar proporcionan elementos de diseño utilizados para hacer la ingeniería de automóviles, aviones, máquinas herramienta y robots. La aplicación del diseño basado en patrones al desarrollo de software promete a éste los mismos beneficios que tiene la tecnología industrial: ser predecible, disminuir el riesgo y aumentar la productividad.

A través del proceso de diseño, debe buscarse toda oportunidad para aplicar los patrones de diseño existentes (cuando cumplan las necesidades del diseño), en vez de crear otros nuevos.

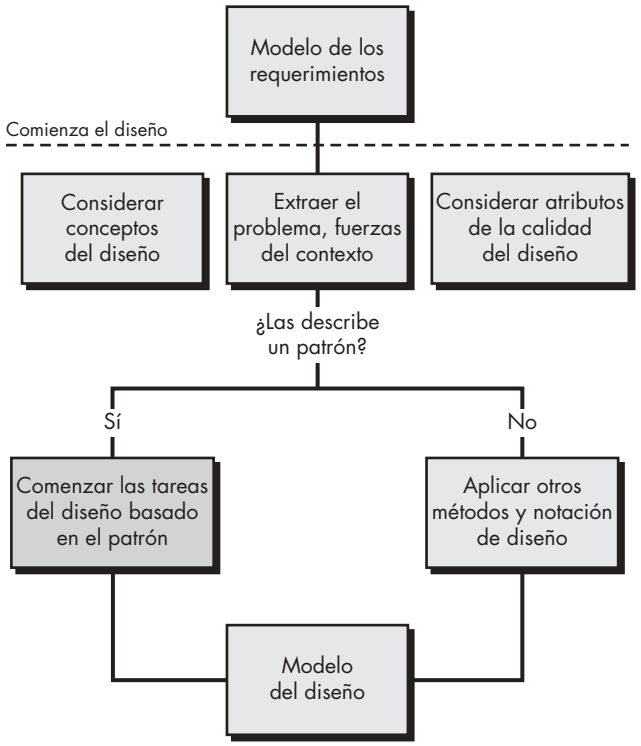
12.2.1 El diseño basado en patrones, en contexto

El diseño basado en patrones no se utiliza en el vacío. Los conceptos y técnicas analizados para el diseño arquitectónico, en el nivel de componentes y de la interfaz de usuario (capítulos 9 a 11), se utilizan junto con un enfoque basado en patrones.

En el capítulo 8 se dijo que un conjunto de lineamientos y atributos de la calidad se emplean como la base para tomar todas las decisiones del diseño de software. Éstas reciben influencia de un conjunto de conceptos fundamentales del diseño (como la separación de problemas, mejora por etapas, independencia funcional, etc.) que se logran con el uso de heurísticos que han evolucionado a lo largo de muchas décadas, así como de las mejores prácticas (tales como las técnicas de notación de modelos) que se han propuesto para hacer que el diseño sea más fácil de realizar y que tenga más eficacia como base para la construcción.

En la figura 12.1 se ilustra el papel que juega en todo esto el diseño basado en patrones. Un diseñador de software comienza con un modelo de requerimientos (explícito o implícito) que muestra una representación abstracta del sistema. El modelo de requerimientos describe el conjunto problema, establece el contexto e identifica el sistema de fuerzas que actúan. Tal vez implique al sistema en forma abstracta, pero el modelo de requerimientos hace poco para representar al diseño explícitamente.

FIGURA 12.1
El diseño basado en patrones, en contexto



Cuando el lector inicie su trabajo como diseñador, es importante que recuerde los atributos de la calidad. Éstos (por ejemplo, un diseño debe implementar todos los requerimientos explícitos establecidos en el modelo de los requerimientos) fijan una forma de evaluar la calidad del software, pero no son de mucha ayuda para lograrlos en la realidad. El diseño que se cree debe tener los conceptos fundamentales del diseño analizados en el capítulo 8. Entonces, deben aplicarse técnicas probadas para traducir las abstracciones contenidas en el modelo de requerimientos en una forma más concreta, que es el diseño del software. Para lograr esto, se usarán los métodos y herramientas de modelado disponibles para el diseño arquitectónico, en el nivel de componentes y de la interfaz, pero sólo cuando se enfrente un problema, contexto y sistema de fuerzas que no se haya resuelto antes. Si ya existe una solución, ¡úsela¡ Y esto quiere decir aplicar un enfoque del diseño basado en patrones.

12.2.2 Pensar en patrones

En un libro excelente sobre diseño basado en patrones, Shalloway y Trott [Sha05] comentan acerca de la "nueva forma de pensar" cuando se utilizan patrones como parte de la actividad de diseño:

> Tuve que abrir mi mente a una nueva forma de pensar. Y cuando lo hice, escuché a [Christopher] Alexander decir que "el buen diseño de software no se logra sólo poniendo juntas las partes ejecutoras".

El buen diseño comienza con la consideración del contexto: el panorama. Cuando se evalúa el contexto, se extrae una jerarquía de problemas que deben resolverse. Algunos de éstos serán de naturaleza global, mientras que otros se abocarán a características y funciones específicas del software. Todo será afectado por las fuerzas del sistema que influirán en la naturaleza de la solución propuesta.

Shalloway y Trott [Sha05] sugieren el siguiente enfoque,[6] que permite que un diseñador piense en patrones:

? El diseño basado en patrones parece interesante para el problema que debo resolver. Pero, ¿por dónde comienzo?

1. Asegurarse de entender el panorama: el contexto en el que se encuentra el software que se va a elaborar. El modelo de requerimientos debe transmitir esa comprensión.
2. Estudiar el panorama, identificar los patrones presentes en ese nivel de abstracción.
3. Comenzar el diseño con patrones del "panorama" que establezcan un contexto o esqueleto para el trabajo de diseño adicional.
4. "Trabajar dentro del contexto" [Sha05] en busca de patrones en niveles más bajos de abstracción que contribuyan a la solución del diseño.
5. Repetir los pasos 1 a 4 hasta que el diseño esté completo.
6. Mejorar el diseño, adaptando cada patrón a las especificidades del software que se trata de elaborar.

Es importante observar que los patrones no son entidades independientes. Los patrones de diseño presentes en un nivel alto de abstracción invariablemente influirán en la manera en la que otros patrones se aplican en niveles más bajos de abstracción. Además, es frecuente que los patrones colaboren entre sí. Esto implica que, cuando se selecciona un patrón arquitectónico, muy bien puede influir en los patrones de diseño en el nivel de componentes que se elijan. Del mismo modo, cuando se seleccione un patrón de diseño de la interfaz, a veces es obligatorio usar patrones que colaboren con él.

Para ilustrar lo anterior, considere la *webapp* **CasaSeguraAsegurada.com**. Si se considera el panorama, la *webapp* debe abordar cierto número de problemas fundamentales:

- Cómo dar información acerca de los productos y servicios de *CasaSegura*
- De qué manera vender los productos y servicios de *CasaSegura* a los clientes
- En qué forma establecer la vigilancia y control de un sistema de seguridad instalado que se base en internet.

Cada uno de estos problemas fundamentales puede desglosarse aún más en un conjunto de subproblemas. Por ejemplo, *Cómo vender por internet* implica un patrón de **comercio electrónico** que en sí mismo acarrea un gran número de patrones en niveles de abstracción más bajos. El patrón **ComercioElectrónico** (probablemente un patrón arquitectónico) implica mecanismos para abrir una cuenta de un cliente, mostrar los productos que se van a vender, seleccionar los que se van a adquirir, etcétera. Entonces, si se piensa en patrones, es importante determinar si existe un patrón para abrir una cuenta. Si se dispone de **AbrirCuenta** como patrón viable para el contexto del problema, puede colaborar con otros patrones, tales como **ElaborarFormatodeEntrada, AdministrarFormatosdeEntrada** y **ValidarFormatosdeEntrada**. Cada uno delinea problemas por resolver y las soluciones que se aplicarán.

12.2.3 Tareas de diseño

Cuando se utiliza una filosofía de diseño basado en patrones, se llevan a cabo las siguientes tareas:

1. **Examinar el modelo de requerimientos y desarrollar una jerarquía del problema.** Describir cada problema y subproblema aislando el problema, el contexto y el sistema de fuerzas que se aplican. Hay que trabajar desde los problemas amplios (nivel

6 Basado en el trabajo de Christopher Alexander [Ale79].

¿Cuáles son las tareas requeridas para crear un diseño basado en patrones?

de abstracción elevado) hasta los subproblemas más pequeños (niveles más bajos de abstracción).

2. **Determinar si se ha desarrollado un lenguaje del patrón confiable para el dominio del problema.** Como se dijo en la sección 12.1.4, un lenguaje del patrón se dirige a problemas asociados con un dominio específico de la aplicación. El equipo de software *CasaSegura* buscaría un lenguaje de patrón desarrollado específicamente para productos para la seguridad del hogar. Si no se encontrara ese nivel de especificidad del lenguaje del patrón, el equipo dividiría el problema del software de *CasaSegura* en una serie de dominios generales del problema (de vigilancia de dispositivos digitales, de interfaz de usuario, de administración de video digital) y en la búsqueda de lenguajes de patrón apropiados.
3. **A partir de un problema amplio, determinar si para el mismo se dispone de uno o más patrones arquitectónicos.** Si existe un patrón arquitectónico, hay que asegurarse de estudiar todos los patrones colaboradores. Si el patrón es apropiado, debe adaptarse la solución del diseño propuesta y elaborar un elemento del modelo del diseño que lo represente en forma adecuada. Como se dijo en la sección 12.2.2, un problema amplio para la *webapp* **CasaSeguraAsegurada.com** se aborda con un patrón **ComercioElectrónico**. Éste sugerirá una arquitectura específica para enfrentar los requerimientos del comercio electrónico.
4. **Con el uso de colaboraciones provistas para el patrón arquitectónico, deben estudiarse los problemas en el nivel de subsistema o componente, y buscar los patrones más apropiados para enfrentarlos.** Tal vez sea necesario buscar en varios depósitos de patrones, así como en la lista de aquellos que correspondan a la solución arquitectónica. Si se encuentra un patrón apropiado, hay que adaptar la solución del diseño propuesta y construir un elemento del modelo del diseño que lo represente de manera adecuada. Hay que asegurarse de aplicar el paso 7.
5. **Repetir los pasos 2 a 5 hasta que se hayan resuelto todos los problemas amplios.** Esto implica comenzar con el panorama general y elaborarlo para resolver problemas en niveles cada vez más detallados.
6. **Si los problemas de diseño de la interfaz de usuario han sido aislados (éste es el caso casi siempre), buscar los muchos depósitos de patrones de diseño de la interfaz de usuario para encontrar patrones apropiados.** Se procede en forma similar a los pasos 3, 4 y 5.
7. **Sin importar su nivel de abstracción, si resulta promisorio un lenguaje de patrón o un depósito de patrones o un patrón individual, hay que comparar el problema por resolver con el patrón o patrones presentados.** Debe asegurarse de estudiar el contexto y las fuerzas para garantizar que el patrón, en efecto, da una solución factible para el problema.
8. **Asegurarse de refinar el diseño a medida que se obtiene de los patrones, con el empleo de criterios de calidad como guía.**

Aunque este enfoque del diseño es de arriba abajo, las soluciones del diseño en la vida real a veces son más complejas. Gillis [Gil06] comenta al respecto lo siguiente:

> Los patrones de diseño que se manejan en la ingeniería de software deben usarse en forma deductiva y racionalista. De modo que cuando se tiene el problema o requerimiento general X y el patrón de diseño Y resuelve X, entonces utilice Y. Pero cuando reflexiono en mi propio proceso —y tengo razones para pensar que no soy el único—, observo que es más orgánico que eso, más inductivo que deductivo, más de abajo hacia arriba que de arriba abajo.

> Es obvio que debe lograrse un balance. Cuando un proyecto se encuentra en la fase inicial y trato de dar el salto de los requerimientos abstractos a una solución concreta del diseño, es frecuente que tome aire para realizar una búsqueda [...] he hallado que los patrones de diseño son útiles, lo que me permite enmarcar con rapidez el problema de diseño en términos concretos.

Además, el enfoque basado en patrones debe usarse junto con otros conceptos y técnicas de diseño de software.

12.2.4 Construcción de una tabla para organizar el patrón

Las entidades que aparecen en la tabla pueden darse con una indicación de la aplicabilidad relativa del patrón.

A medida que avanza el diseño basado en patrones, quizá se encuentren problemas para organizar y clasificar los candidatos que surjan de múltiples lenguajes y repositorios de patrones. Para ayudar a organizar la evaluación de los patrones candidatos, Microsoft [Mic04] sugiere crear una *tabla de organización de patrones* que tenga la forma general que se ilustra en la figura 12.2.

Una tabla organizadora de patrones puede implementarse como modelo de hoja de cálculo con el uso del formato de la figura. La columna de la izquierda (sombreada) está organizada por datos/contenido, arquitectura, nivel de componentes y aspectos de la interfaz de usuario. En el renglón superior se enlistan cuatro tipos de patrón: base de datos, aplicación, implementación e infraestructura. En las celdas de la tabla se anotan los nombres de los patrones que son candidatos.

Para obtener las entradas de la tabla organizadora, se busca en lenguajes y repositorios de patrones que aborden un enunciado particular del problema. Cuando se encuentra uno o más patrones candidatos, se introducen en el renglón correspondiente al enunciado del problema y en la columna que corresponda al tipo de patrón. El nombre del patrón se introduce como hipervínculo hacia la URL o dirección web que contenga la descripción completa del patrón.

12.2.5 Errores comunes en el diseño

El diseño basado en patrones lo hará un mejor diseñador de software, pero *no* es una panacea. Igual que todos los métodos de diseño, debe comenzarse con los primeros principios, con én-

FIGURA 12.2

Tabla de organización de patrones

Fuente: Adaptada de [Mic04].

	Base de datos	Aplicación	Implementación	Infraestructura
Datos/Contenido				
Enunciado del problema ...	Nombre(s) del patrón		Nombre(s) del patrón	
Enunciado del problema ...		Nombre(s) del patrón		Nombre(s) del patrón
Enunciado del problema ...	Nombre(s) del patrón			Nombre(s) del patrón
Arquitectura				
Enunciado del problema ...		Nombre(s) del patrón		
Enunciado del problema ...		Nombre(s) del patrón		Nombre(s) del patrón
Enunciado del problema ...				
Nivel de componentes				
Enunciado del problema ...		Nombre(s) del patrón	Nombre(s) del patrón	
Enunciado del problema ...				Nombre(s) del patrón
Enunciado del problema ...		Nombre(s) del patrón	Nombre(s) del patrón	
Interfaz de usuario				
Enunciado del problema ...		Nombre(s) del patrón	Nombre(s) del patrón	
Enunciado del problema ...		Nombre(s) del patrón	Nombre(s) del patrón	
Enunciado del problema ...		Nombre(s) del patrón	Nombre(s) del patrón	

No fuerce a un patrón, incluso si se aboca al problema en cuestión. Si el contexto y las fuerzas son los equivocados, busque otro patrón.

fasis en los fundamentos de la calidad del software y el aseguramiento de que el diseño sí satisface las necesidades expresadas por el modelo de los requerimientos.

Cuando se emplea el diseño basado en patrones, suelen ocurrir varios errores comunes. En ciertos casos, no se dedica el tiempo suficiente a entender el problema subyacente, su contexto y fuerzas, y en consecuencia se elige un patrón que parece correcto, pero es inapropiado para llegar a la solución que se requiere. Una vez seleccionado el patrón equivocado, se es renuente a reconocer el error y se fuerza el patrón para que se ajuste. En otros casos, el problema tiene fuerzas que no son consideradas por el patrón escogido, lo que da como resultado un ajuste deficiente o erróneo. En ocasiones se aplica demasiado literalmente y no se implementan las adaptaciones requeridas para el espacio del problema.

¿Es posible evitar estos errores? En la mayoría de los casos la respuesta es *sí*. Todo buen diseñador busca una segunda opinión y ve con buenos ojos la revisión de su trabajo. Las técnicas de revisión que se estudian en el capítulo 15 ayudan a garantizar que el diseño basado en patrones que se haya obtenido dé como resultado una solución de alta calidad para el problema de software que debe resolverse.

12.3 Patrones arquitectónicos

Una arquitectura de software puede tener cierto número de patrones arquitectónicos que se aboquen a aspectos tales como concurrencia, persistencia y distribución.

Si un constructor de vivienda decide edificar una de estilo colonial con vestíbulo al centro, sólo hay un estilo arquitectónico que puede aplicarse. Los detalles de éste (como el número de chimeneas, fachada, ubicación de puertas y ventanas, etcétera) variarán en forma considerable; pero, una vez tomada la decisión de la arquitectura general, el estilo será impuesto por el diseño.[7]

Los patrones arquitectónicos son un poco distintos. Por ejemplo, toda vivienda (y todo estilo arquitectónico para ellas) emplea un patrón **Cocina**. Éste y aquéllos con los que colabora abordan problemas asociados con el almacenamiento y preparación de comida, las herramientas que se necesitan para realizar estas tareas y las reglas para situarlas en relación con el flujo de trabajo en dicho espacio. Además, el patrón tal vez se enfrente a problemas asociados con barras, iluminación, interruptores eléctricos, isla central, pisos, etc. Es obvio que hay más de un diseño para la cocina, el que es dictado con frecuencia por el contexto y el sistema de fuerzas. Pero todo diseño se concibe en el contexto de la "solución" sugerida por el patrón **Cocina**.

¿Cuáles son algunos dominios comunes del patrón arquitectónico?

Como ya se dijo, los patrones arquitectónicos para el software definen un enfoque específico para el manejo de algunas características del sistema. Bosch [Bos00] y Booch [Boo08] definen cierto número de dominios del patrón arquitectónico. En los párrafos siguientes se describen ejemplos representativos:

Control de acceso. Hay muchas situaciones en las que el acceso a datos, características y funciones realizadas por una aplicación está limitado a usuarios finales definidos específicamente. Desde un punto de vista arquitectónico, el acceso a cierta parte de la arquitectura del software debe controlarse de manera rigurosa.

Concurrencia. Muchas aplicaciones deben manejar tareas múltiples de manera que simule paralelismo (esto ocurre, por ejemplo, siempre que un solo procesador administra varias tareas o componentes "paralelos"). Hay varias formas distintas en las que una aplicación maneja la concurrencia y cada una puede presentarse con un patrón arquitectónico diferente. Por ejemplo, un enfoque consiste en usar un patrón de **AdministracióndeProcesosdelSistemaOperativo**

7 Esto implica que habrá un atrio y vestíbulo central, que las habitaciones estarán situadas a izquierda y derecha del atrio, que la casa tendrá dos (o más) plantas, que las recámaras de la casa se ubicarán en la planta alta, entre otras características. Estas "reglas" son obligatorias una vez que se toma la decisión de usar un estilo colonial con vestíbulo central.

que proporcione características incrustadas para el SO que permitan que los componentes se ejecuten de manera concurrente. El patrón también incorpora funcionalidad del SO que administra la comunicación entre procesos, programación de tareas y otras capacidades que se requieren para lograr concurrencia. Otro enfoque tal vez defina un programador de tareas en el nivel de la aplicación. Un patrón **ProgramadordeTareas** contiene un conjunto de objetos activos en el que cada uno de estos incluye una operación *tick ()* [Bos00]. El programador invoca en forma periódica *tick ()* para cada objeto, que luego realiza las funciones que debe antes de devolver el control al programador, el que después hace la invocación de la operación *tick ()* para el siguiente objeto concurrente.

Distribución. El problema de la distribución se aboca a la manera en la que los sistemas o componentes de los sistemas se comunican entre sí en un ambiente distribuido. Se toman en cuenta dos subproblemas: 1) la forma en la que se conectan las entidades una con la otra y 2) la naturaleza de la comunicación que tiene lugar. El patrón arquitectónico más común establecido para enfrentar el problema de distribución es el **Negociador**. Un negociador actúa como un "mediador" entre el componente del cliente y el del servidor. El cliente envía un mensaje al negociador (que contiene toda la información apropiada para que se efectúe la comunicación) y éste finaliza la conexión.

Persistencia. Los datos persisten si sobreviven a la ejecución del proceso que los creó. Los datos persistentes se almacenan en una base de datos o archivos que pueden ser leídos o modificados por otros procesos en un momento posterior. En los ambientes orientados a objetos, la idea de un objeto persistente extiende un poco más el concepto de persistencia. Los valores de todos los atributos del objeto, el estado general de éste y otra información complementaria se almacenan para su recuperación y uso futuro. En general, se emplean dos patrones arquitectónicos para lograr la persistencia: un patrón de **SistemadeAdministracióndeBasedeDatos** que aplica la capacidad de almacenamiento y recuperación de un SABD a la arquitectura y un patrón **PersistenciaaNiveldeAplicación** que construye las características de la persistencia

INFORMACIÓN

Repositorios de patrones de diseño

En la red mundial hay muchas fuentes disponibles de patrones de diseño. Algunos se obtienen de lenguajes de patrón publicados en forma individual, mientras que otros forman parte de un portal o repositorio de patrones. Es recomendable consultar los siguientes recursos en web:

Hillside.net **http://hillside.net/patterns/**
Una de las colecciones más amplias de patrones y lenguajes de patrón disponibles en web.

Repositorio de Patrones Portland
http://c2.com/ppr/index.html Contiene apuntadores hacia una amplia variedad de recursos y colecciones de patrones.

Índice de patrones **http://c2.com/cgi/wiki?PatternIndex**
"Colección ecléctica de patrones".

Manual de patrones arquitectónicos de Booch **www.booch.com/architecture/index.jsp** Referencias bibliográficas a cientos de patrones arquitectónicos y de diseño de componentes.

Colecciones de patrones UI

Patrones UI/HCI **www.hcipatterns.org/patterns.html**

Patrones UI de Jennifer Tidwell
www.time-tripper.com/uipatterns/

Patrones de diseño UI móviles
http://patterns.littlespringsdesign.com/wikka.php?wakka=mobile

Patrones

Lenguaje de patrones para el diseño UI
www.maplefish.com/todd/papers/Experiences.html

Biblioteca de diseño interactivo para juegos
www.eelke.com/research/usability.html

Patrones de diseño UI
www.cs.helsinki.fi/u/salaakso/patterns/

Patrones de diseño especializado

Aviónica **http://g.oswego.edu/dl/acs/acs/acs.html**

Sistemas de información para negocios
www.objectarchitects.de/arcus/cookbook/

Procesamiento distribuido **www.cs.wustl.edu/~schmidt/**

Patrones IBM para comercio electrónico
www.128.ibm.com/developerworks/patterns/

Biblioteca de patrones de diseño Yahoo!
http://developer.yahoo.com/ypatterns/

WebPatterns.org **http://webpatterns.org/**

en la arquitectura de la aplicación (por ejemplo, un software de procesamiento de textos que administre su propia estructura de documento).

Antes de que pueda elegirse cualquiera de los patrones arquitectónicos representativos mencionados en los párrafos anteriores, debe evaluarse lo apropiado que es para la aplicación y el estilo arquitectónico general, así como el contexto y sistema de fuerzas que especifiquen.

12.4 PATRONES DE DISEÑO EN EL NIVEL DE COMPONENTES

Los patrones de diseño en el nivel de componentes brindan soluciones comprobadas que se abocan a uno o más subproblemas extraídos del modelo de requerimientos. En muchos casos, los patrones de diseño de este tipo se centran en algún elemento funcional de un sistema. Por ejemplo, la aplicación **CasaSeguraAsegurada.com** debe resolver el siguiente subproblema de diseño: *¿Cómo pueden obtenerse especificaciones del producto e información acerca de cualquier dispositivo de CasaSegura?*

Después de enunciar el subproblema que afecta a la solución, debe considerarse el contexto y el sistema de fuerzas que también la afecten. Al estudiar el modelo de requerimientos apropiados del caso de uso, se observa que el consumidor utiliza la especificación de un dispositivo de *CasaSegura* (como un sensor de seguridad o cámara) con propósitos de información. Sin embargo, cuando se selecciona la función de comercio electrónico, quizá se requiera otro tipo de información relacionada con la especificación (por ejemplo, el precio).

La solución del subproblema involucra una búsqueda. Como buscar es un problema muy común, no es sorprendente que haya muchos patrones relacionados con dicha tarea. Al investigar en varios repositorios de patrones, se encuentran los siguientes, así como el problema que resuelve cada uno:

AdvancedSearch. Los usuarios deben encontrar un objeto específico en una gran colección de ellos.

HelpWizard. Los usuarios necesitan ayuda acerca de cierto tema relacionado con el sitio web o necesitan encontrar una página específica dentro del sitio.

SearchArea. Los usuarios deben encontrar una página.

SearchTips. Los usuarios requieren saber cómo controlar el motor de búsqueda.

SearchResults. Los usuarios tienen que procesar una lista de resultados de una búsqueda.

SearchBox. Los usuarios tienen que encontrar un objeto o información específicos.

Para **CasaSeguraAsegurada.com**, el número de productos es particularmente grande y cada uno tiene una clasificación relativamente sencilla, por lo que es probable que no sean necesarios **AdvanceSearch** y **HelpWizard**. De manera similar, la búsqueda es lo bastante simple como para no requerir **SearchTips**. Sin embargo, la descripción de **SearchBox** se da (en parte) como sigue:

Search Box

(Adaptado de www.welie.com/patterns/showPattern.php?patternID=search.)

Problema: Los usuarios necesitan encontrar un objeto o información específica.

Motivación: Cualquier situación en la que se aplique una búsqueda por medio de una palabra clave a través de una colección de objetos de contenido organizada como páginas web.

Contexto: En vez de navegar para obtener información o contenido, el usuario quiere hacer una búsqueda directa a través del contenido de múltiples páginas web.

Cualquier sitio web que ya tenga navegación primaria. El usuario tal vez desee buscar un objeto en cierta categoría. El usuario quizá quiera especificar una consulta adicional.

Fuerzas: El sitio web ya cuenta con navegación primaria. Los usuarios quieren buscar un objeto en cierta categoría. Los usuarios desean especificar una consulta más profunda con el empleo de operadores booleanos sencillos.

Solución: Ofrecer funciones de búsqueda que consisten en una etiqueta de búsqueda, campo de palabra clave, filtro aplicable y botón de "ir". Oprimir la tecla *return* tiene el mismo efecto que seleccionar el botón *ir*. Asimismo, proveer Sugerencias de Búsqueda y ejemplos en una página distinta. Junto a la función de búsqueda se coloca un vínculo hacia la página. El cuadro de edición para el término de la búsqueda es suficientemente grande como para dar acomodo a tres consultas normales de usuario (alrededor de 20 caracteres). Si el número de filtros es más de 2, se usa un cuadro mayor para seleccionar los filtros, o bien un botón de radio.

Los resultados de la búsqueda se presentan en una página nueva con una etiqueta clara que contiene al menos "resultados de la búsqueda" o algo similar. La función de búsqueda se repite en la parte superior de la página con las palabras clave ingresadas, de modo que los usuarios sepan cuáles fueron.

La descripción continúa con otras entradas, como se describe en la sección 12.1.3.

El patrón continúa para describir cómo acceder, presentar, ajustar, etc. los resultados de la búsqueda. Con base en esto, el equipo de **CasaSeguraAsegurada.com** puede diseñar los componentes requeridos para implementar la búsqueda o (lo que es más probable) adquirir los componentes reutilizables existentes.

CasaSegura

Aplicación de patrones

La escena: Plática informal durante el diseño de un incremento de software que implementa el control de sensores por internet para **CasaSeguraAsegurada.com**

Participantes: Jamie (responsable del diseño) y Vinod (arquitecto jefe del sistema de **CasaSeguraAsegurada.com**).

La conversación:

Vinod: Entonces, ¿cómo va el diseño de la interfaz de control de la cámara?

Jamie: No va mal. Ya diseñé la mayoría de las capacidades para conectarla a los sensores reales sin demasiados problemas. También comencé a pensar acerca de la interfaz para que los usuarios en verdad muevan, abran y acerquen las cámaras desde una página web remota, pero no estoy seguro de haber terminado ya.

Vinod: ¿Qué es lo que tienes?

Jamie: Bueno, los requerimientos son que el control de la cámara sea muy interactivo: cuando el usuario mueva el control, la cámara debe moverse tan pronto como sea posible. Entonces, pensaba en disponer varios botones, como en una cámara normal, para que cuando el usuario controle la cámara haga clic en ellos.

Vinod: Mmm. Sí, eso funcionaría, pero no estoy seguro de que esté bien: cada vez que se haga clic en el control, se necesitará esperar para que ocurra toda la comunicación entre el cliente y el servidor, por lo que no habrá una buena percepción de retroalimentación rápida.

Jamie: Eso es lo que he pensado y por ello no estoy muy conforme con el enfoque, pero no estoy seguro de qué más hacer.

Vinod: Bueno, ¿por qué no usar el patrón **ControldeDispositivoInteractivo**?

Jamie: Mmm… ¿qué es eso? Nunca lo he oído.

Vinod: Básicamente es un patrón para el problema exacto que describes. La solución que propone es crear una conexión de control entre el servidor y el dispositivo, a través del cual pueden enviarse los comandos de control. De esa forma, no necesitas mandar solicitudes HTTP normales. El patrón incluso muestra cómo puedes implementar esto con el uso de algunas técnicas AJAX sencillas. En el lado del cliente tienes algunos scripts de Java que se comunican directamente con el servidor y que envían los comandos tan pronto como el usuario está sin hacer nada.

Jamie: Genial… Eso es justo lo que necesito para resolver el problema. ¿Dónde lo encuentro?

Vinod: Está disponible en un repositorio en línea. Ésta es la URL.

Jamie: Iré a revisarlo.

Vinod: Sí, pero recuerda revisar el campo de consecuencias para el patrón. Creo recordar que había algo acerca de tener cuidado con ciertos aspectos de seguridad. Pienso que es porque se crea un canal de control separado y de ese modo se evaden los mecanismos normales de seguridad de web.

Jamie: Buena observación. Es probable que no se me hubiera ocurrido… Gracias.

12.5 PATRONES DE DISEÑO DE LA INTERFAZ DE USUARIO

En los últimos años, se han propuesto cientos de patrones de interfaz de usuario (IU). La mayoría se ubican en una de las siguientes 10 categorías (se estudian con un ejemplo representativo)[8] según los describen Tidwell [Tid02] y VanWelie[Wel01]:

Whole UI. Proporciona una guía para diseñar la estructura y navegación de alto nivel a través de toda la interfaz.

Patrón: NavegacióndeAltoNivel (TopLevelNavigation)

Descripción breve: Se usa cuando un sitio o aplicación implementa cierto número de funciones importantes. Da un menú de alto nivel, acoplado con frecuencia con un logotipo o gráfico identificador que permite la navegación directa hacia cualquiera de las funciones principales del sistema.

Detalles: Las funciones importantes (por lo general limitadas a tener de cuatro a siete nombres de función) se enlistan en la parte superior de la pantalla (también es posible tener formatos en columnas verticales) en un renglón de texto horizontal. Cada nombre da un vínculo hacia la función o fuente de información apropiada. Es frecuente usarla con el patrón **MigajasdePan** (**BreadCrumbs)** que se estudia más adelante.

Elementos de navegación: Cada nombre de función o contenido representa un vínculo hacia la función o contenido apropiados.

Distribución de la página. Se aboca a la organización general de páginas (para sitios web) o de distintas pantallas (para aplicaciones interactivas).

Patrón: ApilarTarjetas (CardStack)

Descripción breve: Se utiliza cuando deben seleccionarse aleatoriamente cierto número de subfunciones o categorías de contenido específicas relacionadas con una característica o función. Tiene la apariencia de una pila de tarjetas con "pestaña", cada una seleccionable con un clic del ratón, que representan subfunciones o categorías de contenido específicas.

Detalles: Las tarjetas con pestañas son una metáfora bien entendida y son fáciles de manipular por parte del usuario. Cada tarjeta con su pestaña (divisor) tiene un formato ligeramente diferente. Algunas requieren de entradas y tienen botones u otros mecanismos de navegación; otras más son informativas. Pueden combinarse con otros patrones, tales como **ListaDesplegable** y **LlenarLosEspacios**, entre otros.

Elementos de navegación: Un clic en una pestaña hace que aparezca la tarjeta apropiada. También están presentes características de navegación dentro de la tarjeta, pero en general éstas deben iniciar una función relacionada con los datos de la tarjeta, no establecer un vínculo real hacia otra pantalla.

8 Aquí se utiliza un formato del patrón abreviado. En [Tid02] y [Wel01] se encuentran descripciones del patrón completo (así como decenas de otros patrones).

Formatos y entrada. Considera varias técnicas de diseño para llenar las entradas en el nivel de formato.

Patrón: Llenar los espacios

Descripción breve: Permite introducir datos alfanuméricos en un "cuadro de texto".

Detalles: Los datos entran en un cuadro de texto. En general, se validan y procesan después de pulsar algún indicador de texto o gráfico (como un botón que diga "ir", "enviar", "siguiente", etc.). En muchos casos, este patrón se combina con una lista desplegable u otros patrones (por ejemplo, BUSCAR <*lista desplegable*> PARA <*llenar los espacios* del cuadro de texto>).

Elementos de navegación: Indicador de texto o gráfico que inicia la validación y el procesamiento.

Tablas. Dan una guía para el diseño a fin de crear y manipular datos tabulares de todo tipo.

Patrón: OrdenarTabla (SortableTable)

Descripción breve: Despliega una lista larga de registros que pueden ordenarse por medio de un mecanismo de cambio para cualquier etiqueta de columna.

Detalles: Cada renglón de la tabla representa un registro completo. Cada columna representa un campo del registro. Cada encabezado de columna en realidad es un botón seleccionable que se pulsa para iniciar un ordenamiento ascendente o descendente en el campo asociado con la columna para todos los registros desplegados. Por lo general, la tabla es ajustable y tiene algún mecanismo de desplazamiento para el caso de que el número de registros sea más grande que el espacio disponible en la ventana.

Elementos de navegación: Cada encabezado de columna inicia el ordenamiento de todos los registros. No se da otro elemento de navegación, aunque en ciertos casos cada registro contiene vínculos de navegación hacia otro contenido o funciones.

Manipulación directa de los datos. Se aboca a la edición, modificación y transformación de los datos.

Patrón: MigajasdePan (BreadCrumbs)

Descripción breve: Brinda una ruta completa de navegación cuando el usuario trabaja con una jerarquía compleja de páginas o pantallas.

Detalles: Se da a cada página o pantalla un identificador único. La ruta de navegación hacia la ubicación actual se especifica en una ubicación predefinida para cada pantalla. La ruta tiene la forma siguiente: **página inicial>página del tema principal>página del subtema>página específica>página actual.**

Elementos de navegación: Cualquiera de las entradas en la pantalla de las *migajas de pan* puede usarse como un apuntador hacia el vínculo de regreso hacia un nivel más alto de la jerarquía.

Navegación. Ayuda al usuario a navegar a través de menús jerárquicos, páginas web y pantallas interactivas.

Patrón: Editar (EditInPlace)

Descripción breve: Brinda capacidades de edición de texto sencillo para ciertos tipos de contenido en la ubicación que se muestra en la pantalla. No es necesario que el usuario introduzca explícitamente alguna función de edición de texto o algún modo.

Detalles: El usuario observa en la pantalla el contenido que debe modificarse. Con un doble clic en el contenido, se indica al sistema que se desea editar. El contenido se resalta para indicar que el modo de edición está activado para que el usuario haga los cambios apropiados.

Elementos de navegación: Ninguno.

Búsqueda. Permite hacer búsquedas de contenido específico a través de información conservada en un sitio web o que está contenida en almacenamientos persistentes de datos accesibles a través de una aplicación interactiva.

Patrón: BúsquedaSimple (SimpleSearch)

Descripción breve: Da la capacidad de buscar un sitio web o fuente persistente de datos para un concepto simple de datos descritos por una cadena alfanumérica.

Detalles: Brinda la capacidad de hacer una búsqueda local (una página o un archivo) o global (todo el sitio o la base de datos completa) para la cadena de búsqueda. Genera una lista de "aciertos" ordenados según su probabilidad de satisfacer las necesidades del usuario. No hace búsquedas de conceptos múltiples o con operaciones booleanas especiales (véase *patrón de búsqueda avanzada*).

Elementos de navegación: Cada entrada de la lista representa un vínculo de navegación hacia los datos a los que se hace referencia con la entrada.

Elementos de página. Implanta elementos específicos de una página web o de una pantalla del monitor.

Patrón: Mago (Wizard)

Descripción breve: Lleva al usuario paso a paso a través de una tarea compleja y lo guía para que la termine por medio de ventanas sencillas en la pantalla.

Detalles: El ejemplo clásico es un proceso de registro de cuatro etapas. El patrón mago genera una ventana en cada una de ellas, en las que solicita información del usuario paso a paso.

Elementos de navegación: La navegación hacia delante y atrás permite que el usuario vuelva a cada etapa en el proceso mago.

Comercio electrónico. Específicos para sitios web, estos patrones implementan elementos recurrentes de las aplicaciones de comercio electrónico.

Patrón: CarritodeCompras (ShoppingCart)

Descripción breve: Da una lista de artículos seleccionados para compra.

Detalles: Enlista artículos, cantidad, código del producto, disponibilidad (en inventario, fuera de inventario), precio, información para la entrega, costos de envío y otra información relevante para la compra. También da la facilidad de editar (por ejemplo, retirar, cambiar la cantidad, etcétera).

Elementos de navegación: Contiene la capacidad de continuar la compra o salir.

Varios. Son patrones que no se ajustan fácilmente a ninguna de las categorías anteriores. En ciertos casos, dependen del dominio u ocurren sólo para clases específicas de usuarios.

Patrón: IndicadordeAvance (ProgressIndicator)

Descripción breve: Proporciona una indicación del avance cuando una operación dura más de *n* segundos.

Detalles: Se representa con un icono animado o cuadro de mensaje que contiene alguna indicación visual (por ejemplo, una "barra de peluquero", barra de avance con indicador de porcentaje, etc.) de que el procesamiento está en curso. También contiene una indicación de texto acerca del estado del procesamiento.

Elementos de navegación: Es frecuente que contenga un botón que permita al usuario hacer una pausa o cancelar el procesamiento.

Cada uno de los ejemplos de patrones anteriores (y todos los de cada categoría) también pueden tener un diseño completo en el nivel de componentes, incluso clases de diseño, atributos, operaciones e interfaces.

El estudio exhaustivo de los patrones de interfaz de usuario se encuentra más allá del alcance de este libro. Si el lector está interesado, se le recomienda consultar a [Duy02], [Bor01], [Tid02] y [Wel01].

12.6 PATRONES DE DISEÑO DE WEBAPP

En este capítulo aprendimos que hay cuatro tipos diferentes de patrones y muchas formas de clasificarlos. Cuando se consideran los problemas de diseño que deben resolverse para construir una *webapp*, es bueno considerar categorías de patrones en dos dimensiones: centrarse en el diseño del patrón y en el nivel de granularidad. *Centrarse en el diseño* identifica cuál aspecto del modelo del diseño es relevante (por ejemplo, arquitectura de la información, navegación e interacción). La *granularidad* determina el nivel de abstracción que se considera (¿el patrón se aplica a toda la *webapp*, a una sola página web, a un subsistema o a un componente individual de la *webapp*?)

12.6.1 Centrarse en el diseño

PUNTO CLAVE
La atención se hace "más fina" a medida que se avanza en el diseño.

En los capítulos anteriores se hizo énfasis en un avance del diseño que comienza por tomar en cuenta la arquitectura, aspectos en el nivel del componente y representaciones de la interfaz de usuario. En cada paso se consideran los problemas y soluciones propuestos para comenzar en un nivel alto de abstracción a fin de pasar poco a poco a otro más detallado y específico. En otras palabras, el diseño se "angosta" a medida que avanza. Los problemas (y soluciones) que se encontrarán cuando se diseñe una arquitectura de información para una *webapp* serán diferentes de aquellos que aparecen cuando se diseña una interfaz. Por tanto, no debe sorprender que los patrones para el diseño de *webapps* se desarrollen para distintos niveles de atención, de modo que se aborden los problemas (y sus soluciones) únicos que se encuentren en cada nivel. Los patrones de *webapps* se clasifican con el empleo de los siguientes niveles de atención en el diseño:

- **Patrones de arquitectura de la información**: se relacionan con la estructura general del espacio de información y con las formas en las que los usuarios interactúan con ésta.
- **Patrones de navegación:** definen estructuras de los vínculos de navegación, tales como jerarquías, anillos, recorridos, etcétera.
- **Patrones de interacción**: contribuyen al diseño de la interfaz de usuario. Los patrones en esta categoría se enfrentan al modo en el que la interfaz informa al usuario de las consecuencias de una acción específica, cómo expande el usuario el contenido con base en el empleo del contexto y sus deseos, la mejor manera de describir el destino implícito por un vínculo, la manera de informar al usuario acerca del estado de una interacción en curso y aspectos relacionados con la interfaz.
- **Patrones de presentación**: ayudan a presentar el contenido al usuario a través de la interfaz. Los patrones en esta categoría se abocan al modo de organizar las funciones de control de la interfaz de usuario para mejorar su uso, a mostrar la relación entre una acción de la interfaz y los objetos de contenido a los que afecta y a la forma de establecer jerarquías eficaces del contenido.
- **Patrones funcionales**: definen los flujos de trabajo, comportamientos, procesamiento, comunicación y otros elementos algorítmicos dentro de una *webapp*.

En la mayoría de casos, sería inútil explorar la colección de patrones de arquitectura de la información cuando se encuentra un problema en el diseño de la interacción. Se estudiarían los

patrones de interacción porque es la atención en el diseño lo que es relevante para el trabajo que se está ejecutando.

12.6.2 Granularidad del diseño

Cuando un problema involucra aspectos del "panorama", debe tratarse de desarrollar soluciones (y los patrones de uso relevantes) que se centren en éste. A la inversa, cuando la atención es muy estrecha (como cuando se selecciona únicamente un aspecto de un conjunto reducido de cinco o menos de ellos), la solución (y el patrón correspondiente) se busca con más estrechez. En términos del nivel de granularidad, los patrones se describen en los niveles siguientes:

- **Patrones arquitectónicos.** Este nivel de abstracción se relacionará por lo común con patrones que definen la estructura general de la *webapp*, que indican las relaciones entre diferentes componentes o incrementos y que definen las reglas para especificar las relaciones entre los elementos (páginas, paquetes, componentes y subsistemas) de la arquitectura.
- **Patrones de diseño.** Éstos se abocan a un elemento específico del diseño, como un agrupamiento de componentes, a fin de resolver algún problema de diseño, relaciones entre los elementos de una página, o mecanismos para efectuar la comunicación entre componentes. Ejemplo de esto sería el patrón **Broadsheet** para la distribución de la página inicial de una *webapp*.
- **Patrones de componentes.** Este nivel de abstracción se relaciona con elementos individuales de pequeña escala de una *webapp*. Algunos ejemplos son los elementos de interacción individual (botones de radio), de navegación (¿cómo dar formato a los vínculos?) o funcionales (algoritmos específicos).

También es posible definir la relevancia de distintos patrones para diferentes clases de aplicaciones o dominios. Por ejemplo, una colección de patrones (en diferentes niveles de atención al diseño y granularidad) puede tener relevancia particular para los negocios electrónicos (*e-business)*.

INFORMACIÓN

Repositorios de patrones para el diseño de hipermedios

El sitio web IAWiki (**http://iawiki.net/WebsitePatterns**) es un espacio de colaboración para los arquitectos de la información y contiene muchos recursos útiles. Entre ellos hay vínculos hacia varios catálogos y repositorios útiles de patrones de hipermedios. Ahí están representados cientos de patrones de diseño:

Repositorios de patrones de diseño de hipermedios
www.designpattern.lu.unisi.ch/
Patrones de interacción, por Tom Erickson
www.pliant.org/personal/Tom_Erickson/InteractionPatterns.html
Patrones de diseño web, por Martijn van Welie
www.welie.com/patterns/
Patrones web para el diseño de la IU
http://harbinger.sims.berkeley.edu/ui_designpatterns/webpatterns2/webpatterns/home.php
Patrones para sitios web personales
www.rdrop.com/%7ehalf/Creations/Writings/Web.patterns/index.html
Mejora de sistemas de información web con patrones de navegación
http://www8.org/w8-papers/5b-hypertext-media/improving/improving.html
Un lenguaje de patrón HTML 2.0
www.anamorph.com/docs/patterns/default.html
Campos comunes. Un lenguaje de patrón para el diseño HCI
www.mit.edu/~jtidwell/interaction_patterns.html
Patrones para sitios web personales
www.rdrop.com/~half/Creations/Writings/Web.patterns/index.html
Lenguaje de patrón de indización
www.cs.brown.edu/~rms/InformationStructures/Indexing/Overview.html

12.7 Resumen

Los patrones de diseño dan un mecanismo codificado para describir problemas y su solución en forma tal que permiten que la comunidad de ingeniería de software diseñe el conocimiento para que sea reutilizado. Un patrón describe un problema, indica el contexto y permite que el usuario entienda el ambiente en el que sucede el problema, y enlista un sistema de fuerzas que indican cómo puede interpretarse el problema en su contexto, y el modo en el que se aplica la solución. En el trabajo de la ingeniería de software se identifican y documentan patrones generativos que describen un aspecto importante y repetible de un sistema para después proporcionar la manera de elaborar dicho aspecto dentro de un sistema de fuerzas que es único para un contexto determinado.

Los patrones arquitectónicos describen problemas de diseño amplios que se resuelven con un enfoque estructural. Los patrones de datos describen problemas recurrentes orientados a datos y las soluciones para modelar éstos que se utilizan para resolverlos. Los patrones de componentes (también conocidos como patrones de diseño) se abocan a problemas asociados con el desarrollo de subsistemas y componentes, la manera en la que se comunican entre sí y su ubicación en una arquitectura mayor. Los patrones de diseño de interfaces describen problemas comunes de la interfaz de usuario y su solución con un sistema de fuerzas que incluye las características específicas de los usuarios finales. Los patrones de *webapps* se enfrentan a un conjunto de problemas que surgen cuando se construyen *webapps*, y es frecuente que incorporen muchas categorías de los patrones mencionados. Una estructura proporciona el marco en el que residen los patrones y los idiomas describen los detalles de implementación específicos del lenguaje de programación para un algoritmo, o parte de él, o para una estructura de datos específica. Para hacer las descripciones de patrones, se emplea un formato o plantilla estándar. Un lenguaje de patrón incluye un conjunto de patrones, cada uno de los cuales es descrito con el empleo de una plantilla estándar e interrelacionada para que muestre cómo colaboran los patrones para resolver problemas en el dominio de la aplicación.

El diseño basado en patrones se utiliza junto con métodos de diseño arquitectónico, en el nivel de componentes y de interfaz de usuario. El enfoque del diseño comienza con el estudio del modelo de requerimientos a fin de detectar los problemas, definir el contexto y describir el sistema de fuerzas. A continuación se buscan los lenguajes de patrón para el dominio del problema con objeto de determinar si existen patrones para los problemas detectados. Una vez que se han encontrado los patrones apropiados, se usan como guía para el diseño.

Problemas y puntos por evaluar

12.1. Analice las tres "partes" de un patrón de diseño y dé un ejemplo concreto de cada uno en algún campo distinto del software.

12.2. ¿Cuál es la diferencia entre un patrón no generativo y uno generativo?

12.3. ¿En qué difieren los patrones arquitectónicos de los patrones de componentes?

12.4. ¿Qué es estructura y en qué difiere de un patrón? ¿Qué es un idioma y en qué se diferencia de un patrón?

12.5. Con el uso de la plantilla de diseño de patrones presentada en la sección 12.1.3, desarrolle la descripción completa de un patrón sugerido por su profesor.

12.6. Desarrolle un lenguaje de esqueleto de patrón para un deporte con el que esté familiarizado. Comience por abordar el contexto, el sistema de fuerzas y los problemas amplios que deban resolver un entrenador y su equipo. Sólo necesita especificar los nombres de los patrones y hacer la descripción frase por frase de cada uno.

12.7. Encuentre cinco repositorios de patrones y presente la descripción abreviada de los tipos de patrones que contenga cada uno.

12.8. Cuando Christopher Alexander afirma que "un buen diseño no se logra con sólo reunir las partes ejecutantes", ¿qué cree usted que quiere decir?

12.9. Con el uso de las tareas de diseño basado en patrones mencionadas en la sección 12.2.3, desarrolle un diseño de esqueleto para el "sistema de diseño de interiores" descrito en la sección 11.3.2.

12.10. Elabore una tabla de organización de patrones para aquellos que haya utilizado en el problema 12.9.

12.11. Con el uso de la plantilla para diseñar patrones que se presentó en la sección 12.1.3, desarrolle la descripción completa para el patrón **Cocina** mencionado en la sección 12.3.

12.12. La banda de los cuatro [Gam95] ha propuesto varios patrones de componentes aplicables a sistemas orientados a objetos. Seleccione uno de ellos (disponibles en web) y analícelo.

12.13. Encuentre tres repositorios de patrones de interfaz de usuario. Seleccione uno de cada repositorio y haga una descripción breve de él.

12.14. Encuentre tres repositorios de patrones para *webapps*. Seleccione uno de cada repositorio y descríbalos brevemente.

Lecturas adicionales y fuentes de información

En la última década se han escrito muchos libros sobre el diseño de patrones destinados a los ingenieros de software. Gamma *et al.* [Gam95] escribieron un libro fundamental sobre dicho tema. Las contribuciones más recientes incluyen los textos de Lasater (*Design Patterns*, Wordware Publishing, Inc., 2007), Holzner (*Design Patterns for Dummies*, For Dummies, 2006), Freeman *et al.* (*Head First Design Patterns*, O'Reilly Media, Inc., 2005) y Shalloway y Trott (*Design Patterns Explained*, 2a. ed., Addison-Wesley, 2004). Una edición especial de *IEEE Software* (julio/agosto, 2007) estudia una amplia variedad de temas sobre patrones de software. Kent Beck (*Implementation Patterns*, Addison-Wesley, 2008) estudia patrones para codificar e implementar aspectos que se encuentran durante la actividad de construcción.

Otros libros se centran en patrones de diseño según se encuentran en el desarrollo de aplicaciones específicas y ambientes de lenguajes. Las contribuciones en esta área incluyen las siguientes: Bowers (*Pro CSS and HTML Design Patterns*, Apress, 2007), Tropashko y Burleson (*SQL Design Patterns: Expert Guide to SQL Programming*, Rampant Techpress, 2007), Mahemoff (*Ajax Design Patterns*, O'Reilly Media, Inc., 2006), Metsker y Wake (*Design Patterns in Java*, Addison-Wesley, 2006), Nilsson (*Applying Domain-Driven Design and Patterns: With Examples en C# and .NET*, Addison-Wesley, 2006), Sweat (*PHPArchitect's Guide to PHP Design Patterns*, Marco Tabini & Associates, Inc., 2005), Metsker (*Design Patterns C#*, Addison-Wesley, 2004), Grand y Merrill (*Visual Basic .NET Design Patterns*, Wiley, 2003), Crawford y Kaplan (*J2EE Design Patterns*, O'Reilly Media, Inc., 2003), Juric *et al*. (*J2EE Design Patterns Applied*, Wrox Press, 2002), y Marinescu y Roman (*EJB Design Patterns*, Wiley, 2002).

Otros libros se abocan a dominios de aplicaciones específicas. Entre éstos están las contribuciones de Kuchana (*Software Architecture Design Patterns in Java*, Auerbach, 2004), Joshi (C++ *Design Patterns and Derivatives Pricing*, Cambridge University Press, 2004), Douglass (*Real-Time Design Patterns*, Addison-Wesley, 2002) y Schmidt y Rising (*Design Patterns in Communication Software*, Cambridge University Press, 2001).

La lectura de los libros clásicos escritos por el arquitecto Christopher Alexander (*Notes on the Synthesis or Form,* Harvard University Press, 1964, y *A Pattern Language: Towns, Builidings, Construction*, Oxford University Press, 1977) es útil para el diseñador de software que trata de entender por completo los patrones de diseño.

En internet hay una amplia variedad de fuentes de información sobre el diseño de patrones. En el sitio web del libro se encuentra la lista actualizada de referencias en la red mundial que son relevantes para el diseño basado en patrones: **www.mhhe.com/engsc/compsci/pressman/professional/olc/ser.htm**.

CAPÍTULO

13

DISEÑO DE WEBAPPS

CONCEPTOS CLAVE

arquitectura CVM 328
arquitectura de webapps . . . 328
arquitectura del contenido . . 326
objetos. 324
diseño
arquitectónico 326
calidad 374
contenido 324
en el nivel de componentes. 331
estética 323
gráfico 324
metas. 320
navegación 329
pirámide. 321
MDHOO 332

En su autorizado libro sobre diseño web, Jakob Nielsen [Nie00] afirma lo siguiente: "En esencia, hay dos enfoques fundamentales del diseño: el ideal artístico de expresarse a sí mismo y el ideal de la ingeniería de resolver un problema para un cliente." En la primera década del desarrollo de la web, el enfoque que elegían muchos diseñadores era el ideal artístico. El diseño se desarrollaba de manera *ad hoc* y por lo general se efectuaba a medida que se generaba HTML. Después evolucionó a partir de la visión artística que surgió de la construcción de *webapps*.

Incluso hoy, muchos desarrolladores web utilizan *webapps* como cartel infantil para un "diseño limitado". Afirman que la inmediatez y volatilidad de una *webapp* palidecen ante el diseño formal, que éste evoluciona conforme se elabora (se codifica) una aplicación y que debe dedicarse relativamente poco tiempo a crear un modelo detallado del diseño. Este argumento tiene algo de verdad, pero sólo para *webapps* relativamente sencillas. Cuando el contenido y las funciones son complejos o cuando el tamaño de la *webapp* incluye cientos o miles de objetos de contenido, funciones y clases de análisis y cuando el éxito de la *webapp* tenga influencia directa en el éxito del negocio, el diseño no puede ni debe tomarse a la ligera.

Esta realidad conduce al segundo enfoque de Nielsen: "el ideal de la ingeniería es resolver un problema para un cliente". La ingeniería web[1] adopta esta filosofía, y un enfoque más riguroso del diseño de *webapps* permite que los desarrolladores la hagan realidad.

UNA MIRADA RÁPIDA

¿Qué es? El diseño de *webapps* incluye actividades técnicas y no técnicas que incluyen lo siguiente: establecer la vista y sensación de la *webapp*, creando la distribución estética de la interfaz de usuario, definiendo la estructura arquitectónica general, desarrollando el contenido y la funcionalidad que residen en la arquitectura y planeando la navegación que ocurre dentro de la *webapp*.

¿Quién lo hace? En la creación del modelo del diseño de una *webapp*, intervienen ingenieros web, diseñadores gráficos, desarrolladores de contenido y varios participantes más.

¿Por qué es importante? El diseño permite crear un modelo que se evalúe respecto de su calidad para mejorarlo antes de la generación de contenido y código, de la realización de las pruebas y del involucramiento de un gran número de usuarios. El diseño es el lugar donde se establece la calidad de la *webapp*.

¿Cuáles son los pasos? El diseño de una *webapp* incluye seis etapas principales que son orientadas por la información obtenida durante la modelación de los requerimientos. El diseño del contenido utiliza el contenido del modelo (desarrollado durante el análisis) como la base para establecer el diseño de los objetos del contenido. El diseño estético (también llamado diseño gráfico) establece la vista y sensación que el usuario final percibe. El diseño arquitectónico se centra en la estructura general de hipermedios de todos los objetos y funciones del contenido. El diseño de la interfaz establece la distribución y mecanismos de distribución que definen a la interfaz de usuario. El diseño de la navegación define la forma en la que el usuario final navega a través de la estructura de hipermedios. Y el diseño de los componentes representa la estructura interna detallada de los elementos funcionales de la *webapp*.

¿Cuál es el producto final? El principal producto que se genera durante el diseño de la *webapp* es un modelo del diseño que incluye aspectos del diseño del contenido, de la estética, de la arquitectura, de la interfaz, de la navegación y en el nivel de componentes.

¿Cómo me aseguro de que lo hice bien? Cada elemento del modelo del diseño se revisa para descubrir errores, inconsistencias u omisiones. Además, se toman en cuenta soluciones alternativas y se evalúa el grado en el que el modelo actual del diseño llevará a una implementación eficaz.

1 La *ingeniería web* [Pre08] es una versión adaptada del enfoque de ingeniería de software que se presenta en todo este libro. Propone una estructura ágil, pero disciplinada, para construir sistemas y aplicaciones basados en web con calidad industrial.

13.1 Calidad del diseño de webapps

El diseño es la actividad de la ingeniería que genera un producto de alta calidad. Esto lleva a una pregunta recurrente que surge en todas las disciplinas de la ingeniería: ¿qué es calidad? En esta sección estudiaremos la respuesta en el contexto del desarrollo de *webapps*.

Toda persona que haya navegado en la red mundial o que haya utilizado una intranet corporativa se ha formado una opinión sobre lo que constituye una "buena" *webapp*. Los puntos de vista individuales varían mucho. A algunos usuarios les gustan los gráficos brillantes, otros prefieren el texto simple, algunos más demandan mucha información, mientras los hay que desean una presentación abreviada. A algunos les agradan las herramientas analíticas sofisticadas o tener acceso a bases de datos y a otros les gusta lo sencillo. En realidad, la percepción del usuario acerca de lo que es "bueno" (y en consecuencia la aceptación o rechazo de la *webapp*) puede ser un aspecto más importante que cualquier otro de índole técnica sobre la calidad de las *webapps*.

Cita:

"Si los productos se diseñan para que se ajusten mejor a las tendencias naturales del comportamiento humano, entonces las personas estarán más satisfechas, más complacidas y serán más productivas."

Susan Weinschenk

Pero, ¿cómo se percibe la calidad de una *webapp*? ¿Qué atributos debe tener para que haya bondad ante los ojos de los usuarios finales y a la vez existan las características técnicas de calidad que permitan corregir, adaptar, mejorar y dar apoyo a la aplicación en el largo plazo?

En realidad, a las *webapps* se aplican todas las características técnicas de calidad del diseño estudiadas en el capítulo 8 y los atributos generales de calidad que se vieron en el capítulo 14. Sin embargo, los más relevantes de estos atributos generales —usabilidad, funcionalidad, confiabilidad, eficiencia y susceptibilidad de recibir mantenimiento— brindan una base útil para evaluar la calidad de los sistemas basados en web.

Olsina *et al.* [Ols99] han preparado un "árbol de requerimientos de calidad" que identifica un conjunto de atributos técnicos —usabilidad, funcionalidad, confiabilidad, eficiencia y susceptibilidad de recibir mantenimiento— que generan la calidad en las *webapps*.[2] La figura 13.1 resume su trabajo. Los criterios que ahí aparecen tienen interés especial si el lector diseña, construye y da mantenimiento a *webapps* a largo plazo.

Offutt [Off02] agrega los siguientes a los cinco atributos principales de la calidad que se mencionan en la figura 13.1:

¿Cuáles son los principales atributos de la calidad de las *webapps*?

Seguridad. Las *webapps* se han integrado mucho con bases de datos críticas, corporativas y gubernamentales. Las aplicaciones de comercio electrónico extraen y después almacenan información delicada para el cliente. Por estas y muchas otras razones, la seguridad de las *webapps* tiene importancia capital en muchas situaciones. La medida clave de la seguridad de una *webapp* y de su ambiente de servidor es su capacidad para rechazar los accesos no autorizados o para detener un ataque proveniente del exterior. El análisis detallado de la seguridad de las *webapps* está más allá del alcance de este libro. Si el lector está interesado en este tema, puede consultar [Vac06], [Kiz05] o [Kal03].

Disponibilidad. Aun la mejor *webapp* será incapaz de satisfacer las necesidades de los usuarios si no se encuentra disponible. En sentido técnico, la disponibilidad es la medida porcentual del tiempo que una *webapp* puede utilizarse. El usuario final común espera que las webapps se hallen disponibles las 24 horas de los 365 días del año. Algo menos que eso es tomado como inaceptable.[3] Pero el "tiempo arriba" no es el único indicador de la disponibilidad. Offutt [Off02] sugiere que "el empleo de características que sólo se encuentren disponibles en un navegador o plataforma" hace que quienes tengan una configuración diferente de navegador o plataforma no puedan utilizar la *webapp*. Invariablemente, el usuario irá a otro sitio.

---

2 Estos atributos de la calidad son similares a los que se mencionan en los capítulos 8 y 14. Esto implica que las características de la calidad son universales para todo el software.

3 Por supuesto, esta expectativa no es realista. Para las *webapps* importantes, deben programarse "tiempos fuera" a fin de que reciban arreglos y actualizaciones.

FIGURA 13.1
Árbol de requerimientos de la calidad.
Fuente: [Ols99].

Escalabilidad. ¿Una *webapp* y su ambiente de servidor pueden crecer para que manejen 100, 1 000, 10 000 o 100 000 usuarios? ¿La *webapp* y los sistemas con los que tiene interfaz son capaces de manejar una variación significativa del volumen o su respuesta se desplomará (o cesará)? No basta construir una *webapp* exitosa. También es importante que pueda asimilar la carga del éxito (muchos más usuarios) y que tenga aún más éxito.

Tiempo para llegar al mercado. Aunque el tiempo que toma llegar al mercado en realidad no es un atributo de la calidad en el sentido técnico, sí lo es desde el punto de vista de la empresa. Es frecuente que la primera *webapp* que llega a un segmento específico del mercado obtenga un número desproporcionado de usuarios finales.

INFORMACIÓN

Diseño de una webapp. Lista de revisión de la calidad

La siguiente es una lista adaptada de la información contenida en **Webreference.com**, donde se plantean preguntas que ayudarán a los diseñadores y a los usuarios finales de la *webapp* a evaluar la calidad de la aplicación:

- ¿Es posible adaptar las opciones de contenido, función o navegación a las preferencias del usuario?
- ¿Puede personalizarse el contenido o la funcionalidad al ancho de banda con el que se comunica el usuario?
- ¿Se han utilizado de manera apropiada las imágenes y otros medios distintos del texto? ¿Es posible adaptar el tamaño de los archivos de imagen para mejorar la eficiencia de la pantalla?
- ¿Las tablas están organizadas y tienen un tamaño tal que se entienden y despliegan de modo eficiente?
- ¿El HTML está optimizado a fin de eliminar las ineficiencias?
- ¿El diseño general de la página tiene lectura y navegación fáciles?
- ¿Todos los vínculos llevan a información de interés para los usuarios?
- ¿Es probable que la mayor parte de vínculos persistan en la red mundial?
- ¿La *webapp* tiene herramientas de administración del sitio, tales como historial del uso, prueba de vínculos, búsqueda local y seguridad?

Aquellos que buscan información disponen de miles de millones de páginas web. Aun las búsquedas bien dirigidas en la red mundial generan una avalancha de contenidos. Con tantas fuentes de información entre las cuales elegir, ¿cómo evalúa el usuario la calidad (es decir, la veracidad, exactitud, completitud, oportunidad, etc.) del contenido que presenta una *webapp*? Tillamn [Til00] sugiere el siguiente conjunto de criterios útiles para ello:

¿Qué debe considerarse cuando se evalúa la calidad del contenido?

- ¿Es fácil determinar el alcance y la profundidad del contenido a fin de estar seguros de que satisface las necesidades del usuario?
- ¿Puede identificarse fácilmente la formación y la autoridad de los autores del contenido?
- ¿Es posible determinar la actualidad del contenido, la fecha de su última actualización y en qué consistió ésta?
- ¿El contenido y su ubicación son estables (permanecerán en la URL de referencia)?

Además de estas preguntas relacionadas con el contenido, deben agregarse las siguientes:

- ¿Es creíble el contenido?
- ¿El contenido es único?, es decir, ¿la *webapp* brinda algún beneficio único a quienes la emplean?
- ¿Es valioso el contenido para la comunidad de usuarios a la que se dirige?
- ¿Está bien organizado el contenido? ¿Está indizado? ¿Se accede a él con facilidad?

Las listas de comprobación citadas en esta sección representan sólo una muestra pequeña de los aspectos que deben estudiarse a medida que el diseño de la *webapp* evoluciona.

13.2 METAS DEL DISEÑO

En su columna periódica sobre el diseño web, Jean Kaiser [Kai02] sugiere un conjunto de metas para el diseño que son aplicables virtualmente a toda *webapp*, sin importar su dominio de aplicación, tamaño o complejidad.

Cita:

"Que algo pueda hacerse, no significa que deba hacerse."

Jean Kaiser

Simplicidad: Aunque parezca algo pasado de moda, el aforismo "todo con moderación" es aplicable a las *webapps*. Existe una tendencia entre ciertos diseñadores a dar al usuario final "demasiado": contenido exhaustivo, extremos visuales, animaciones intrusas, páginas web enormes... y la lista sigue. Es mejor moderación y simplicidad.

El contenido debe ser informativo pero sucinto y debe utilizar un modo de entrega (texto, imágenes, video o audio) que resulte apropiado para la información que se envíe. La estética debe ser agradable pero no abrumadora (demasiados colores tienden a distraer al usuario en vez de mejorar la interacción). La arquitectura debe lograr los objetivos de la *webapp* de la manera más sencilla posible. La navegación debe ser directa y sus mecanismos, obvios para la intuición del usuario final. Las funciones deben ser fáciles de utilizar y más fáciles de entender.

Cita:

"Para ciertas personas, el diseño web se centra en el aspecto visual y en la percepción… Para otras, se trata de estructurar la información y la navegación a través del espacio del documento. Otras más consideran incluso que el diseño web es tecnología… En realidad, el diseño incluye todo esto y tal vez más."

Thomas Powell

Consistencia. Esta meta del diseño se aplica virtualmente a todo elemento del modelo del diseño. El contenido debe construirse de modo congruente (formato y tipografía del texto deben ser los mismos en todos los documentos de texto; las imágenes deben tener coherencia en su aspecto, color y estilo). El diseño gráfico (estética) debe presentar una vista consistente en todas las partes de la *webapp*. El diseño arquitectónico debe establecer plantillas que generen una estructura de hipermedios constante. El diseño de la interfaz debe definir modos consistentes de interacción, navegación y despliegue del contenido. Los mecanismos de navegación deben usarse de manera consistente en todos los elementos de la *webapp*. Como dice Kaiser [Kai02]: "recuerde que para un visitante, un sitio web es un lugar físico. Si sus páginas no tienen un diseño consistente, son fuente de confusión".

Identidad. El diseño de la estética, la interfaz y la navegación de una *webapp* deben ser consistentes con el dominio de la aplicación para la que se va a elaborar. Un sitio web para un grupo de *hip-hop* sin duda tendrá un aspecto y sensación distintos que una *webapp* diseñada para una compañía de servicios financieros. La arquitectura de la *webapp* será diferente por completo, las

interfaces se construirán para que reciban a distintas categorías de usuarios, la navegación se organizará para que cumpla objetivos diferentes. Usted (y quienes contribuyan al diseño) debe trabajar para establecer la identidad de la *webapp* por medio del diseño.

Robustez. Con base en la identidad que se haya establecido, es frecuente que una *webapp* haga una "promesa" implícita al usuario. Éste espera contenido y funciones robustas que sean relevantes para sus necesidades. Si no existen o son insuficientes, es probable que la *webapp* fracase.

Navegabilidad. Ya se dijo que la navegación debe ser sencilla y consistente. También debe estar diseñada en forma tal que sea intuitiva y predecible. Es decir, el usuario debe comprender cómo moverse por la *webapp* sin tener que buscar vínculos o instrucciones para la navegación. Por ejemplo, si un campo de iconos gráficos o de imágenes contiene algunos que serán usados como mecanismos de navegación, deben identificarse visualmente. Nada es más frustrante que intentar encontrar el vínculo vivo apropiado entre muchas imágenes.

También es importante colocar los vínculos hacia el contenido y las funciones de la *webapp* en una ubicación predecible en cada página web. Si se requiere desplazar la página (lo que sucede con frecuencia), los vínculos situados en las partes superior e inferior de la página hacen que las tareas de navegación del usuario sean más fáciles.

Atractivo visual. De todas las categorías de software, las aplicaciones web son indiscutiblemente las más visuales, dinámicas y estéticas. La belleza (atractivo visual) radica sin lugar a dudas en los ojos del espectador, pero muchas características del diseño (aspecto y sensación del contenido, distribución de la interfaz, coordinación del color, balance del texto, imágenes y otros medios) aumentan el atractivo visual.

Compatibilidad. Una *webapp* se usará en varios ambientes (distinto hardware, tipos de conexión, sistemas operativos, navegadores, etcétera) y debe diseñarse para que sea compatible con cada uno.

13.3 PIRÁMIDE DEL DISEÑO DE WEBAPPS

Cita:

"Si un sitio es perfectamente utilizable, pero carece de un estilo elegante y apropiado, fracasará."

Curt Cloninger

¿Qué es el diseño de una *webapp*? Esta sencilla pregunta es más difícil de responder de lo que se creería. En nuestro libro [Pre08] de ingeniería web, David Lowe y yo lo analizamos del modo siguiente:

> La creación de un diseño eficaz requerirá por lo general de un conjunto diversificado de aptitudes. En ocasiones, para proyectos pequeños, un desarrollador único necesitará tener varias habilidades. Para los proyectos grandes, es aconsejable o factible confiar en la experiencia de especialistas: ingenieros web, diseñadores gráficos, desarrolladores de contenido, programadores, especialistas de bases de datos, arquitectos de la información, ingenieros de redes, expertos en seguridad y probadores. Depender de estas distintas aptitudes permite la creación de un modelo cuya calidad puede evaluarse a fin de mejorar su contenido y su código *antes* de que se generen contenido y código, de que se realicen pruebas y de que se involucre un gran número de usuarios. Si el análisis reside en donde *se establece la calidad de la webapp*, entonces el diseño está donde *la calidad está en verdad incrustada*.

La mezcla apropiada de habilidades de diseño variará en función de la naturaleza de la *webapp*. La figura 13.2 ilustra la pirámide del diseño de las *webapps*. Cada nivel representa una acción del diseño que se describe en las siguientes secciones.

13.4 DISEÑO DE LA INTERFAZ DE LA WEBAPP

Cuando el usuario interactúa con un sistema basado en computadora, se aplica un conjunto de principios fundamentales y lineamientos generales de diseño. Éstos se estudiaron en el capítulo

FIGURA 13.2
Pirámide del diseño de las webapps

11.[4] Aunque las *webapps* plantean algunas dificultades especiales en el diseño de la interfaz de usuario, los principios y lineamientos básicos son aplicables.

Uno de los retos del diseño de la interfaz de las *webapps* es la naturaleza indeterminada del punto en el que entra el usuario. Es decir, éste puede ingresar por una ubicación "inicial" de la *webapp* (la página de arranque, por ejemplo) o por algún vínculo en cierto nivel inferior de la arquitectura de aquélla. En algunos casos, la *webapp* se diseña de modo que redirija al usuario a una ubicación inicial, pero si esto es algo indeseable, entonces el diseño debe dar características de navegación en la interfaz que acompañen a todos los objetos de contenido y de las cuales se disponga sin importar el modo en el que el usuario ingrese al sistema.

Los objetivos de la interfaz de una *webapp* son los siguientes: 1) establecer una ventana congruente en el contenido y las funciones que brinda, 2) guiar al usuario a través de una serie de interacciones con la *webapp* y 3) organizar las opciones de navegación y contenido disponibles para el usuario. Para lograr una interfaz consistente, primero debe usarse un diseño estético (véase la sección 13.5) a fin de establecer un "aspecto" coherente. Esto incluye muchas características, pero debe ponerse énfasis en la distribución y la forma de los mecanismos de navegación. Para guiar la interacción del usuario, debe establecerse una metáfora[5] apropiada que permita al usuario tener una comprensión intuitiva de la interfaz. A fin de implementar las opciones de navegación, puede seleccionarse alguno de los siguientes mecanismos:

? ¿De qué mecanismos de interacción disponen los diseñadores de *webapps*?

- *Menús de navegación*: contienen palabras clave (organizadas en forma vertical u horizontal) que enlistan contenido o funciones clave. Estos menús se implementan de modo que el usuario pueda elegir entre una jerarquía de subtemas que se despliegan al seleccionar la opción principal en el menú.
- *Iconos gráficos*: botones, interruptores y otras imágenes similares que permiten que el usuario seleccione alguna propiedad o que especifique una decisión.
- *Imágenes*: cierta representación gráfica que el usuario selecciona para establecer un vínculo hacia un objeto de contenido o función de la *webapp*.

4 La sección 11.5 está dedicada al diseño de la interfaz de la *webapp*. Si aún no se ha leído, es el momento de hacerlo.

5 En este contexto, una metáfora es la representación (establecida por la experiencia del usuario con el mundo real) que se modela en el contexto de la interfaz. Un ejemplo sencillo sería un interruptor deslizable que se utilice para controlar el volumen auditivo de un archivo .mpg.

Es importante observar que en cada nivel de la jerarquía del contenido debe proporcionarse uno o varios de estos mecanismos de control.

13.5 DISEÑO DE LA ESTÉTICA

? No todo ingeniero web (o de software) tiene talento artístico (estético). Si el lector se encuentra en esta categoría, contrate un diseñador gráfico experimentado para que haga el trabajo de diseño estético.

El diseño estético, también llamado *diseño gráfico*, es una actividad artística que complementa los aspectos técnicos del diseño de las *webapps*. Sin estética, una *webapp* tal vez sea funcional pero no atractiva. Con estética, una *webapp* lleva a sus usuarios a un mundo que los sitúa en un nivel tanto visceral como intelectual.

Pero, ¿qué es la estética? Hay un viejo refrán que dice que "la belleza está en los ojos del espectador". Esto es particularmente cierto cuando se trata del diseño estético de las *webapps*. Para llevar a cabo éste con eficacia, hay que volver a la jerarquía del usuario desarrollada como parte del modelo de requerimientos (véase el capítulo 5) y preguntar: *¿quiénes son los usuarios de la webapp y qué "vista" desean tener?*

13.5.1 Aspectos de la distribución

Toda página web tiene una cantidad limitada de "superficie" que se utiliza para dar apoyo a la estética no funcional, características de navegación, contenido de información y funciones dirigidas al usuario. El desarrollo de dicha superficie se planea durante el diseño estético.

Cita:

"Descubrimos que las personas evalúan rápidamente un sitio tan sólo por su diseño visual."

Lineamientos Stanford para la credibilidad en web

Igual que todos los temas de la estética, cuando se diseña la distribución de la pantalla no hay reglas absolutas. Sin embargo, es útil considerar varios lineamientos de la distribución general:

No tema al espacio en blanco. No es aconsejable ocupar con información cada centímetro cuadrado de una página web. El amasijo resultante hará difícil que el usuario identifique la información o las características que necesita y creará un caos visual que no será agradable a los ojos.

Hacer énfasis en el contenido. Después de todo, ésta es la razón de que el usuario esté ahí. Nielsen [Nie00] sugiere que la página web común debe tener 80 por ciento de contenido y destinar el resto a la navegación y otras características.

Organizar los elementos con una distribución que vaya desde arriba a la izquierda hacia abajo a la derecha. La gran mayoría de usuarios de una página web la recorrerán en forma muy parecida a como lo hacen con las hojas de un libro: desde arriba a la izquierda hacia abajo a la derecha.[6] Si los elementos de la distribución tienen prioridades específicas, aquellos que sean prioritarios deben colocarse en la parte superior izquierda de la superficie de la página.

Agrupar la navegación, el contenido y la función en forma geográfica dentro de la página. Los humanos buscamos patrones virtualmente en todas las cosas. Si en una página web no hay patrones discernibles, es probable que la frustración del usuario aumente (debido a la búsqueda innecesaria de la información requerida).

CONSEJO

Los usuarios tienden a tolerar el desplazamiento vertical mejor que el horizontal. Evite los formatos anchos para la página.

No aumente la superficie con la barra de desplazamiento. Aunque es frecuente que se necesite el desplazamiento, la mayor parte de estudios indican que los usuarios preferirían no hacerlo. Es mejor reducir el contenido de la página o presentar en varias páginas el que sea necesario.

Cuando se diseñe la distribución hay que considerar la resolución y tamaño de la ventana del navegador. En vez de definir tamaños fijos dentro de una plantilla, el diseño debe especificar todos los parámetros en términos de porcentaje del espacio disponible [Nie00].

6 Hay excepciones culturales y lingüísticas, pero esta regla se aplica a la mayor parte de usuarios.

13.5.2 Aspectos del diseño gráfico

El diseño gráfico toma en cuenta cada aspecto de la vista y sensación de la *webapp*. El proceso de diseño gráfico comienza con la distribución (véase la sección 13.5.1) y avanza hacia la consideración de los esquemas de color globales; tipos, tamaños y estilos del texto; uso de medios complementarios (audio, video y animación) y todos los demás elementos estéticos de una aplicación.

El análisis exhaustivo de los temas del diseño gráfico de *webapps* está más allá del alcance de este libro. El lector puede obtener recomendaciones y lineamientos en muchos sitios web dedicados a dicha materia (como **www.graphic-design.com**, **www.grantasticdesigns.com** y **www.wpdfd.com**) o en uno o más documentos impresos (como [Roc06] y [Gor02]).

INFORMACIÓN

Sitios web bien diseñados

A veces, la mejor manera de entender lo que es un buen diseño de *webapps* es ver algunos ejemplos. En su artículo "Las veinte mejores recomendaciones para el diseño web", Marcelle Toor (**www.graphic-design.com/Web/feature/tips.html**) recomienda los siguientes sitios como ejemplos de lo que constituye un buen diseño gráfico:

www.creativepro.com/designresource/home/787.html: empresa de diseño dirigida por Primo Angeli

www.workbook.com: este sitio presenta los portafolios de varios ilustradores y diseñadores

www.pbs.org/riverofsong: serie de televisión y radio públicas acerca de la música estadounidense

www.RKDINC.com: empresa de diseño con un portafolio en línea y buenas recomendaciones de diseño

www.creativehotlist.com/index.html: buena fuente de sitios bien diseñados desarrollados por agencias, empresas de artes gráficas y otros especialistas de la comunicación

www.btdnyc.com: compañía de diseño encabezada por Beth Toudreau

13.6 DISEÑO DEL CONTENIDO

Cita:

"Los buenos diseñadores generan la regularidad a partir del caos; comunican sus ideas con claridad, organizando y manipulando palabras e imágenes."

Jeffery Veen

El diseño del contenido se centra en dos tareas diferentes del diseño, cada una de las cuales es dirigida por individuos que poseen habilidades distintas. En primer lugar, se desarrolla una representación del diseño para los objetos del contenido y los mecanismos requeridos para establecer una relación entre ellos. Además, se crea la información dentro de un objeto de contenido específico. El trabajo posterior es llevado a cabo por escritores, diseñadores gráficos y otros actores que generan el contenido que se usará en la *webapp*.

13.6.1 Objetos de contenido

La relación entre los objetos de contenido definidos como parte del modelo de requerimientos para la *webapp* y los objetos de diseño que representan el contenido es análoga a la relación que existe entre las clases de análisis y los componentes del diseño que se describió en capítulos anteriores. En el contexto del diseño de la *webapp*, un objeto de contenido se parece más a un objeto de datos del software tradicional. Un objeto de contenido tiene atributos que incluyen información de contenido específico (normalmente definido durante el modelado de los requerimientos de la *webapp*) y atributos de implementación específica que se establecen como parte del diseño.

Por ejemplo, piense en una clase de análisis, **ComponentedelProducto**, desarrollada para el sistema de comercio electrónico de *CasaSegura*. El atributo de la clase de análisis, **descripción**, se representa como clase de análisis llamada **DescripcióndeComponente** y está compuesta por cinco objetos de contenido: **DescripcióndelMercado, Fotografía, DescripciónTécnica, Esquema** y **Video**, que se muestran como objetos sombreados en la figura 13.3. La información

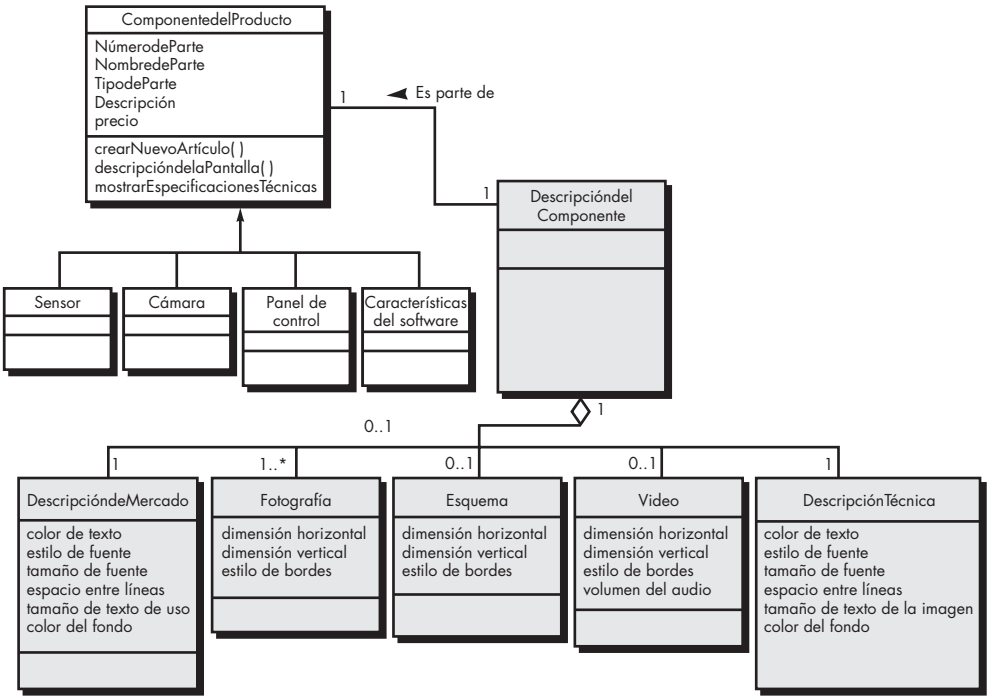


FIGURA 13.3
Representación del diseño de objetos de contenido

contenida dentro del objeto de contenido se etiqueta como atributos. Por ejemplo, **Fotografía** (imagen de tipo .jpg) tiene los atributos **dimensión horizontal, dimensión vertical** y **estilo de bordes**.

Puede usarse una asociación UML y un agregado[7] para representar relaciones entre los objetos de contenido. Por ejemplo, la asociación UML que se ilustra en la figura 13.3 indica que se usa una **DescripcióndeComponente** para cada instancia de la clase **ComponentedelProducto**. La **DescripcióndeComponente** está integrada sobre los cinco objetos de contenido ilustrados. Sin embargo, la notación de multiplicidad que se aprecia indica que **Esquema** y **Video** son opcionales (son posibles 0 ocurrencias), que se requiere una **DescripcióndelMercado** y una **DescripciónTécnica**, y que se emplean una o varias instancias de **Fotografía**.

13.6.2 Aspectos de diseño del contenido

Una vez modelados los objetos del contenido, la información que va a entregar cada objeto debe registrar al autor y después editarse para que satisfaga del mejor modo posible las necesidades del consumidor. La autoría del contenido es trabajo de especialistas en el área relevante de quien diseña el objeto de contenido, dando un bosquejo de la información que se va a entregar y una indicación de los tipos de objetos de contenido general (por ejemplo, texto descriptivo, imágenes, fotografías, etc.) que se usarán para entregar la información. El diseño estético (véase la sección 13.5) también puede aplicarse para representar la vista y sensación apropiadas para el contenido.

Los objetos se "cortan" [Pow02] a medida que se diseñan para formar las páginas de la *webapp*. El número de objetos de contenido incorporado en una página individual está en función de las necesidades del usuario, de las restricciones impuestas por la velocidad de descarga de la conexión de internet y de las restricciones impuestas por la cantidad de desplazamiento vertical que el usuario tolerará.

7 En el apéndice 1 se estudian ambas representaciones.

13.7 Diseño arquitectónico

Cita:

"…la estructura arquitectónica de un sitio bien diseñado no siempre es visible para el usuario: no debe serlo."

Thomas Powell

El diseño arquitectónico está ligado con las metas establecidas para una *webapp*, con el contenido que se va a presentar, con los usuarios que la visitarán y con la filosofía de navegación adoptada. Como diseñador de la arquitectura, el lector debe identificar la arquitectura del contenido y la de la *webapp*. La *arquitectura del contenido*[8] se centra en la manera en la que los objetos de contenido (o compuestos, como páginas web) se estructuran para la presentación y la navegación. La *arquitectura de la webapp* se aboca a la forma en la que la aplicación queda estructurada para administrar la interacción con el usuario, manejar tareas de procesamiento interno, navegar con eficacia y presentar el contenido.

En la mayoría de los casos, el diseño arquitectónico se lleva a cabo en paralelo con el de la interfaz, el estético y el del contenido. Como la arquitectura de la *webapp* tal vez esté muy influida por la navegación, las decisiones que se tomen durante esta acción del diseño influirán en el trabajo realizado durante el diseño de aquélla.

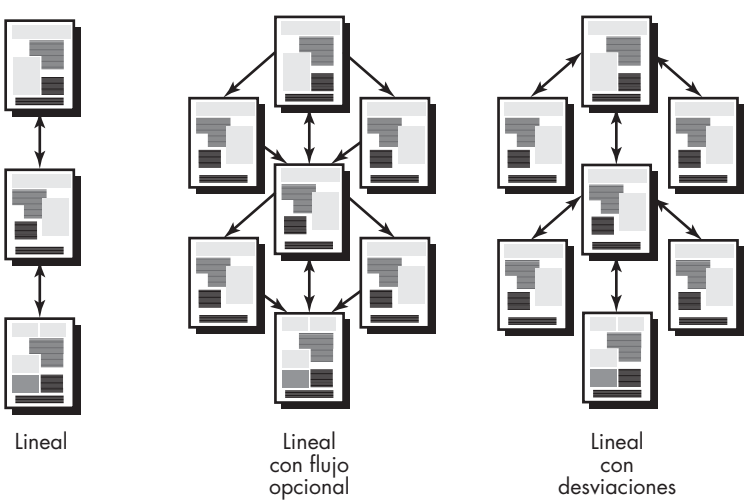
13.7.1 Arquitectura del contenido

El diseño del contenido se centra en la definición de la estructura general de los hipermedios de la *webapp*. Aunque en ocasiones se crean arquitecturas personalizadas, siempre se tiene la opción de elegir entre cuatro distintas estructuras de contenido [Pow00]:

? ¿Qué tipos de arquitectura del contenido es común encontrar?

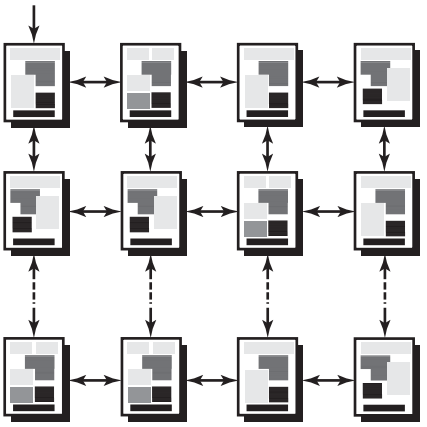
Las *estructuras lineales* (véase la figura 13.4) se encuentran cuando es común una secuencia predecible de interacciones (con cierta variación o diferencia). Un ejemplo clásico es la presentación de *tutoriales* en los que se despliegan páginas de información junto con imágenes relacionadas, videos cortos o audio, sólo después de haber mostrado cierta información de prerrequisitos. La secuencia de la presentación del contenido es predefinida y por lo general es lineal. Otro ejemplo sería una secuencia de entrada para ordenar un producto en la que debe proporcionarse información específica en un orden determinado. En tales casos, resultan apropiadas las estructuras mostradas en la figura 13.4. A medida que el contenido y el procesamiento se hacen más complejos, el flujo exclusivamente lineal que se aprecia a la izquierda de la figura da origen a estructuras lineales más complejas en las que puede invocarse contenido alternativo o

Figura 13.4 Estructuras lineales



8 El término *arquitectura de la información* también se utiliza para denotar estructuras que produzcan una mejor organización, etiquetado, navegación y búsqueda de objetos de contenido.

FIGURA 13.5
Estructura de malla



en las que sucede una desviación para adquirir contenido complementario (estructura que aparece en el lado derecho de la figura 13.4).

Las *estructuras de malla* (véase la figura 13.5) son una opción arquitectónica que se aplica cuando es posible organizar el contenido de una *webapp* en forma categórica en dos (o más) dimensiones. Por ejemplo, considere una situación en la que un sitio de comercio electrónico vende palos de golf. La dimensión horizontal de la malla representa el tipo de palo (madera, metal, cuña, mazo, etc.). La dimensión vertical representa las ofertas que hacen los distintos fabricantes de palos de golf. Entonces, un usuario podría navegar por la malla en forma horizontal a fin de encontrar la columna de mazos y después en forma vertical para examinar las ofertas de los fabricantes que los venden. Esta arquitectura de *webapps* es útil sólo cuando se encuentra contenido muy regular [Pow00].

Las *estructuras jerárquicas* (véase la figura 13.6) son sin duda la arquitectura más común de las *webapps*. A diferencia de las jerarquías de software divididas que se estudiaron en el capítulo 9 y que motivan a controlar el flujo sólo a lo largo de las ramas verticales de la jerarquía, la estructura jerárquica de las *webapps* se diseña en forma tal que permite (por medio de la ramificación del hipertexto) que el flujo del control sea en el sentido horizontal a través de las ramas verticales de la estructura. Así, el contenido presentado en la última rama del lado izquierdo de la jerarquía puede tener vínculos de hipertexto que llevan directamente al contenido que existe en la parte media de la rama del lado derecho de la estructura. Sin embargo, debe observarse

FIGURA 13.6
Estructura jerárquica

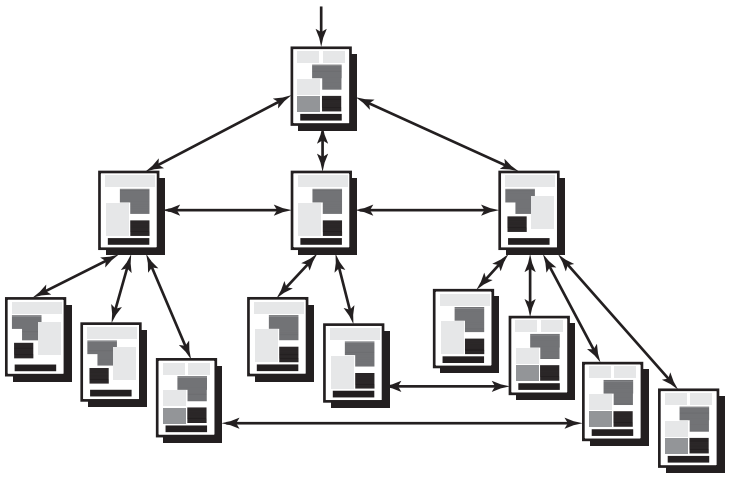
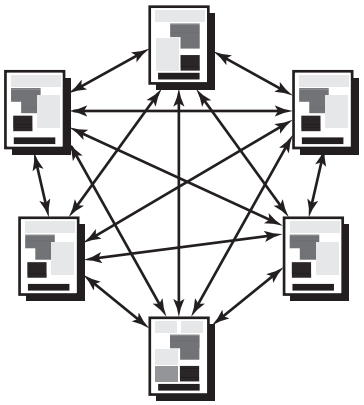


FIGURA 13.7
Estructura de red



que aunque dicha ramificación permite una navegación rápida por el contenido de la *webapp*, genera confusión para el usuario.

Una *estructura de red o "telaraña pura"* (véase la figura 13.7) es similar en muchos sentidos a la arquitectura que evoluciona a partir de sistemas orientados a objetos. Los componentes arquitectónicos (páginas web, en este caso) se diseñan de modo que pasan virtualmente el control (por medio de vínculos de hipertexto) a cada componente del sistema. Este enfoque permite una flexibilidad considerable de navegación, pero al mismo tiempo confunde al usuario.

Las estructuras arquitectónicas estudiadas en los párrafos anteriores se combinan para formar *estructuras compuestas*. La arquitectura general de una *webapp* puede ser jerárquica, pero una parte de la estructura puede tener características lineales y otra, forma de red. La meta del diseñador arquitectónico es ajustar la estructura de la *webapp* con el contenido que va a presentarse y con el procesamiento que va a efectuarse.

13.7.2 Arquitectura de las webapps

La arquitectura de una *webapp* describe una infraestructura que permite que un sistema o aplicación basados en web alcance sus objetivos empresariales. Jacyntho *et al.* [Jac02b] describe las características básicas de esta infraestructura del modo siguiente:

> Las aplicaciones deben construirse con el empleo de capas en las que se tomen en cuenta distintas preocupaciones; en particular, deben separarse los datos de la aplicación de los contenidos de ésta (nodos de navegación), y éstos, a su vez, deben separarse con toda claridad del aspecto y la sensación de la interfaz (páginas).

Los autores sugieren una arquitectura del diseño en tres capas que desacopla la interfaz de la navegación y del comportamiento de la aplicación. Plantean que mantener separadas la interfaz, la aplicación y la navegación, simplifica la implementación y mejora la reutilización.

PUNTO CLAVE
La arquitectura CVM desacopla la interfaz de usuario de las funciones de la *webapp* y del contenido de información.

La arquitectura de *controlador de la vista del modelo* (CVM) [Kra88] [9] es uno de varios modelos sugeridos para la infraestructura de *webapps* que desacoplan la interfaz de usuario de sus funciones y contenido informativo. El *modelo* (a veces denominado "objeto de modelo") contiene todo el contenido y la lógica de procesamiento específicos de la aplicación, incluso todos los objetos de contenido, acceso a fuentes de datos o información externos y todas las funciones de procesamiento que son específicas de la aplicación. La *vista* contiene todas las funciones específicas de la interfaz y permite la presentación de contenido y lógica de procesamiento, incluidos

9 Debe observarse que el CVM es en realidad un patrón de diseño arquitectónico desarrollado para el ambiente Smalltalk (véase **www.cetus-links.org/oo_smalltalk.html**) que puede usarse para cualquier aplicación interactiva.

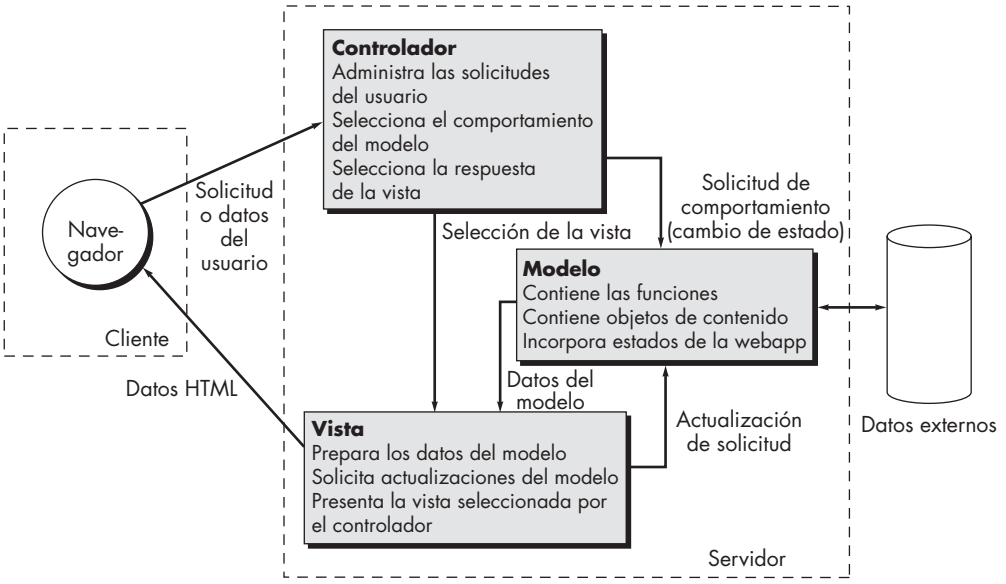


FIGURA 13.8
La arquitectura CVM
Fuente: Adaptado de [Jac02].

todos los objetos de contenido, el acceso a fuentes de datos o información del exterior y todas las funciones de procesamiento que requiere el usuario final. El *controlador* administra el acceso al modelo y la vista, y coordina el flujo de datos entre ellos. En una *webapp*, "la vista es actualizada por el controlador con datos del modelo, basándose en las entradas que da el usuario" [WMT02]. En la figura 13.8 se muestra una representación de la arquitectura CVM.

En relación con la figura, el controlador maneja las solicitudes o datos del usuario. El controlador también selecciona el objeto de vista que sea aplicable con base en la solicitud del usuario. Una vez determinado el tipo de solicitud, se transmite al modelo un pedido de comportamiento, que implementa la funcionalidad o recupera el contenido requerido para dar acomodo a la solicitud. El objeto de modelo accede a los datos almacenados en una base de datos corporativa, como parte de un almacén de datos locales o como una colección de archivos independientes. El objeto de vista apropiado debe dar formato y organizar los datos desarrollados por el modelo para luego trasmitirlos desde el servidor de la aplicación hacia el navegador del cliente para que se desplieguen en la máquina de éste.

En muchos casos, la arquitectura de la *webapp* se define en el contexto del ambiente de desarrollo en el que va a implementarse la aplicación. Si el lector está interesado, puede consultar en [Fow03] el análisis de los ambientes de desarrollo y el papel que juegan en el diseño de arquitecturas de aplicaciones web.

13.8 DISEÑO DE LA NAVEGACIÓN

Cita:

"Gretel, sólo espera que salga la luna y veremos las migajas del pan que desmenucé; ellas nos mostrarán el camino de regreso a casa."

Hansel y Gretel

Una vez que la arquitectura de la *webapp* ha sido establecida y se han identificado sus componentes (páginas, textos, subprogramas y otras funciones de procesamiento), deben definirse las rutas de navegación que permitan a los usuarios acceder al contenido y a las funciones de la *webapp*. Para lograr esto, debe hacerse lo siguiente: 1) identificar la semántica de navegación para los distintos usuarios del sitio y 2) definir la mecánica (sintaxis) para efectuar la navegación.

13.8.1 Semántica de la navegación

Como muchas acciones del diseño de *webapps*, el diseño de la navegación comienza con la consideración de la jerarquía del usuario y los casos de uso relacionados (véase el capítulo 5),

desarrollados para cada categoría de usuario (actor). Cada actor puede usar la *webapp* en forma algo diferente, por lo que tendrán distintos requerimientos de navegación. Además, los casos de uso desarrollados por cada actor definirán un conjunto de clases que incluirán uno o más objetos de contenido o funciones de la webapp. A medida que cada usuario interactúe con la *webapp*, encuentra una serie de *unidades semánticas de navegación* (USN): "conjunto de estructuras de información y navegación relacionadas que colaboran para el cumplimiento de un subconjunto de requerimientos del usuario relacionados" [Cac02].

PUNTO CLAVE

Una USN describe los requerimientos de navegación para cada caso de uso. En esencia, la USN muestra la forma en la que un actor avanza entre los objetos de contenido o entre las funciones de una *webapp*.

Una USN está compuesta por un conjunto de elementos de navegación llamados [Gna99] *formas de navegar* (FdN). Una FdN representa la mejor ruta de navegación a fin de lograr una meta para un tipo de usuario específico. Cada FdN está organizada como un conjunto de *nodos de navegación* (NN) conectados por vínculos. En ciertos casos, un vínculo navegable es otra USN. Entonces, la estructura de navegación general de una *webapp* está organizada como jerarquía de USN.

Para ilustrar el desarrollo de una USN, considere el caso de uso **SeleccionarComponentes deCasaSegura:**

> **Caso de uso: Seleccionar Componentes de CasaSegura**
>
> La *webapp* recomendará componentes del producto (como paneles de control, sensores, cámaras, etc.) y otras características (como funciones con base en PC implementadas en el software) para cada habitación y entrada exterior. Si se piden alternativas, la *webapp* las dará, en caso de que existan. Podré obtener información descriptiva y de precios de cada componente del producto. La *webapp* creará y mostrará una cuenta de los materiales conforme seleccione distintos componentes. Podré dar un nombre a la cuenta de los materiales y guardarla para futuras referencias (véase el caso de uso **Guardar Configuración**).

Los conceptos subrayados en la descripción del caso de uso representan clases y objetos de contenido que se incorporarán en una o más USN que permitirán que un cliente experimente el escenario descrito en el caso de uso **Seleccionar Componentes de CasaSegura**.

Cita:

"El problema de la navegación en un sitio web es conceptual, técnico, espacial, filosófico y logístico. En consecuencia, las soluciones tienden a reclamar combinaciones complejas e improvisadas de arte, ciencia y psicología organizacional."

Tim Horgan

La figura 13.9 ilustra un análisis parcial de la semántica de la navegación que implica el caso de uso **Seleccionar Componentes de CasaSegura**. Con el empleo de la terminología ya mencionada, la figura también representa una forma de navegación (FdeN) para la *webapp* **CasaSeguraAsegurada.com**. Las clases importantes de dominio del problema se muestran junto con objetos seleccionados de contenido (en este caso, el paquete de objetos de contenido llamado **DescripcióndeComponentes**, atributo de la clase **ComponentedelProducto**). Estos conceptos son nodos de navegación. Cada flecha representa un vínculo de navegación[10] y tiene la leyenda con la acción iniciada por el usuario que hace que el vínculo tenga lugar.

Es posible crear una USN para cada caso de uso asociado con cada rol de usuario. Por ejemplo, un **cliente nuevo** de **CasaSeguraAsegurada.com** puede tener tres diferentes casos de uso, los cuales dan como resultado el acceso a distintas funciones de información y de *webapp*. Se crea entonces una USN para cada meta.

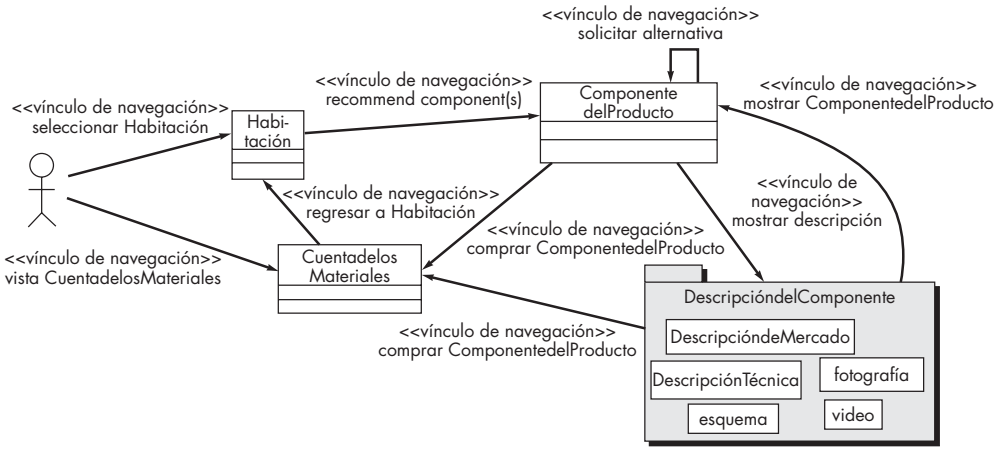
Durante las etapas iniciales del diseño de la navegación, se evalúa la arquitectura del contenido de la *webapp* a fin de determinar una o más FdN para cada caso de uso. Como ya se dijo, una FdN identifica los nodos de navegación (por ejemplo, contenido) y después los vínculos que permiten navegar entre ellos. Entonces, las FdN están organizadas en USN.

13.8.2 Sintaxis de navegación

Al avanzar en el diseño, la tarea siguiente es definir la mecánica de la navegación. Se dispone de varias opciones para desarrollar un enfoque de implementación para cada USN:

10 Éstas se denominan a veces *vínculos semánticos de navegación* (VSN) [Cac02].

FIGURA 13.9
Creación de una USN



CONSEJO

En la mayoría de situaciones, elija mecanismos de navegación horizontales o verticales, pero no ambos.

- *Vínculo de navegación individual*: incluye vínculos basados en texto, iconos, botones e interruptores, así como metáforas gráficas. Deben elegirse vínculos que sean apropiados para el contenido y consistentes con la heurística que conduzca al diseño de una interfaz de alta calidad.
- *Barra de navegación horizontal*: enlista las categorías principales de contenido o de funciones en una barra que contiene vínculos apropiados. En general, se enlistan de cuatro a siete categorías.
- *Columna de navegación vertical*: 1) enlista las principales categorías de contenido o funciones o 2) enlista virtualmente todos los principales objetos de contenido que hay dentro de la *webapp*. Si se elige la segunda opción, las columnas de navegación pueden "expandirse" para que presenten objetos de contenido como parte de una jerarquía (seleccionar una entrada en la columna original ocasiona una expansión que enlista una segunda capa de objetos de contenido relacionados).

CONSEJO

El mapa del sitio debe ser accesible desde cualquier página. El mapa mismo debe estar organizado de modo que la estructura de la información de la webapp se vea fácilmente.

- *Pestañas*: metáfora que no es más que una variación de la barra o columna de navegación y representa categorías de contenido o funciones como pestañas que se seleccionan cuando se requiere un vínculo.
- *Mapas del sitio*: dan una tabla de contenido que incluye todo el contenido a fin de navegar hacia todos los objetos y funciones contenidas dentro de la *webapp*.

Además de elegir la mecánica de navegación, también deben establecerse las convenciones y ayudas apropiadas para navegar. Por ejemplo, los iconos y vínculos gráficos deben invitar a hacer "clic" en ellos, desvaneciendo las aristas a fin de darles una apariencia tridimensional. Debe diseñarse retroalimentación auditiva o visual con objeto de dar al usuario una indicación de que se ha escogido cierta opción de navegación. Para la navegación basada en texto debe utilizarse color que indique los vínculos de navegación y que señale aquéllos ya recorridos. Éstas son unas cuantas convenciones entre las decenas que hay para el diseño y que hacen que la navegación sea amigable para el usuario.

13.9 DISEÑO EN EL NIVEL DE COMPONENTES

Las *webapps* modernas dan funciones de procesamiento cada vez más complejas que: 1) realizan un procesamiento localizado para generar contenido y capacidad de navegación en forma dinámica, 2) proporcionan capacidad de cómputo o de procesamiento de datos que resultan apropiados para el dominio del negocio de la *webapp*, 3) dan consulta y acceso complejos a

bases de datos y 4) establecen interfaces de datos con sistemas corporativos externos. Para lograr estas capacidades (y muchas otras) deben diseñarse y construirse componentes de programas con forma idéntica a los componentes del software tradicional.

Los métodos de diseño estudiados en el capítulo 10 se aplican a los componentes de las *webapps* con poca, o ninguna, modificación. El ambiente de implementación, los lenguajes de programación, los patrones de diseño, estructuras y software, tal vez varíen un poco, pero el enfoque general del diseño es el mismo.

13.10 MÉTODO DE DISEÑO DE HIPERMEDIOS ORIENTADO A OBJETOS (MDHOO)

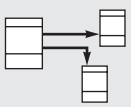
En la última década, se han propuesto varios métodos de diseño para aplicaciones web. Hasta hoy, ninguno de ellos es el dominante.[11] En esta sección se presenta un panorama breve de uno de los métodos de diseño de *webapps* más estudiado.

Daniel Schwabe *et al.* [Sch95, Sch98b] propusieron por primera vez el *Método de Diseño de Hipermedios Orientado a Objetos* (MDHOO), que está compuesto de cuatro distintas actividades de diseño: diseño conceptual, diseño de navegación, diseño abstracto de la interfaz e implementación. En la figura 13.10 se presenta un resumen de estas actividades de diseño y en las secciones que siguen se analizan brevemente.

13.10.1 Diseño conceptual del MDHOO

El *diseño conceptual* del MDHOO genera una representación de los subsistemas, clases y relaciones que definen el dominio de aplicación para la *webapp*. Se puede utilizar UML[12] para crear diagramas de clase apropiados, agregaciones y representaciones compuestas de clase, diagramas de colaboración y otra clase de información que describa el dominio de la aplicación.

FIGURA 13.10
Resumen del MDHOO
Fuente: Adaptado de [Sch95].

	Diseño conceptual	 Diseño de la navegación	Diseño abstracto de la interfaz	Implementación
Productos del trabajo	Clases, subsistemas, relaciones, atributos	Vínculos de nodos, estructuras de acceso, contextos de navegación, transformaciones de navegación	Objetos abstractos de la interfaz, respuestas a eventos externos, transformaciones	Webapp ejecutable
Mecanismos de diseño	Clasificación, composición, agregación, generalización, especialización	Mapeo entre objetos conceptuales y de navegación	Mapeo entre la navegación y los objetos perceptibles	Recurso proporcionado por el ambiente meta
Preocupaciones del diseño	Semántica de modelado del dominio de la aplicación	Toma en cuenta el perfil del usuario y la tarea. Hace énfasis en aspectos cognitivos	Modelado de los objetos perceptibles, implementación de las metáforas escogidas. Descripción de la interfaz para objetos de navegación	Corrección; desempeño de la aplicación; completitud

11 En realidad, son relativamente pocos los desarrolladores web que usan un método específico cuando trabajan en una *webapp*. Hay la esperanza de que este enfoque ad-hoc del diseño cambie a medida que transcurra el tiempo.

12 El MDHOO no prescribe una notación específica; sin embargo, su empleo es común cuando se aplica este método.

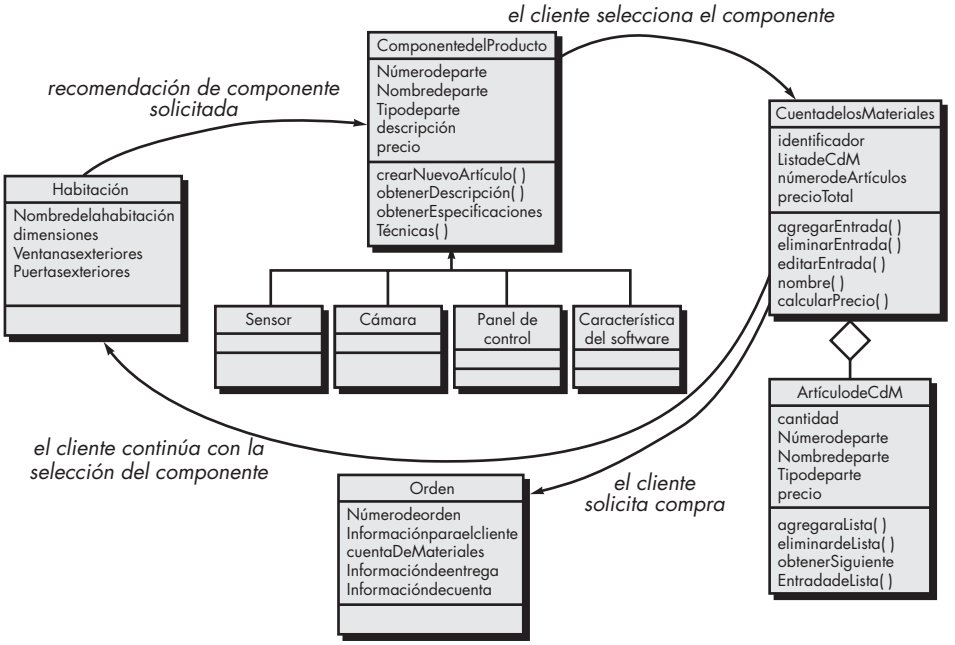


FIGURA 13.11
Esquema conceptual parcial para CasaSeguraAsegurada.com

Como ejemplo sencillo del diseño conceptual del MDHOO, piense en la aplicación de comercio electrónico **CasaSeguraAsegurada.com**. En la figura 13.11, se presenta un "esquema conceptual". Durante el diseño conceptual se reutilizan los diagramas de clase, agregaciones e información desarrollada como parte del análisis de la *webapp*, con objeto de representar las relaciones entre clases.

13.10.2 Diseño de la navegación para el MDHOO

El *diseño de la navegación* identifica un conjunto de "objetos" que se derivan de las clases definidas en el diseño conceptual. Para incluir éstos, se define una serie de "clases de navegación" o "nodos". Se utiliza UML para crear casos de uso, tablas de estado y diagramas de secuencia apropiados; todas éstas son representaciones que ayudan a entender mejor los requerimientos de la navegación. Además, conforme se desarrolla el diseño, se utilizan patrones para el diseño de la navegación. El MDHOO emplea un conjunto predefinido de clases de navegación: nodos, vínculos, anclas y estructuras de acceso [Sch98b]. Estas últimas son más elaboradas e incluyen mecanismos tales como un índice de la *webapp*, mapa del sitio o recorrido guiado.

Una vez definidas las clases de navegación, el MDHOO "estructura el espacio de navegación, agrupando los objetos de navegación en conjuntos llamados contextos" [Sch98b]. Un *contexto* incluye la descripción de la estructura de navegación local, la restricción impuesta al acceso de los objetos de contenido y los métodos (operaciones) requeridos para acceder a los objetos de contenido. Se desarrolla una plantilla contextual (análoga a las tarjetas CRC estudiadas en el capítulo 6) que se emplea para dar seguimiento a los requerimientos de navegación de cada categoría de usuario a través de los distintos contextos definidos en el MDHOO. Al hacer esto, surgen trayectorias específicas de navegación (que en la sección 13.8.1 llamamos FdN).

13.10.3 Diseño abstracto de la interfaz y su implementación

La acción de *diseño abstracto de la interfaz* especifica los objetos de la interfaz que el usuario ve cuando ocurre una interacción con la *webapp*. Se emplea un modelo formal de objetos de inter-

faz, llamado *vista de datos abstractos* (VDA), para representar la relación entre objetos de interfaz y de navegación, así como las características de comportamiento de los objetos de interfaz.

El modelo VDA define una "plantilla estática" [Sch98b] que representa la metáfora de la interfaz e incluye una representación de los objetos de navegación dentro de la interfaz y la especificación de los objetos de ésta (como menús, botones e iconos) que ayudan a la navegación y a la interacción. Además, el modelo VDA contiene un componente de comportamiento (similar al diagrama de estado UML) que indica la forma en la que los eventos "disparan la navegación y cuáles son las transformaciones de la interfaz que ocurren cuando el usuario interactúa con la aplicación" [Sch01a].

La actividad de *implementación* del MDHOO representa una iteración del diseño específica del ambiente en el que opera la *webapp*. Las clases, navegación e interfaz se caracterizan cada una en forma tal que pueden construirse para el ambiente cliente-servidor, sistemas operativos, software de apoyo, lenguajes de programación, y otras características ambientales que son relevantes para el problema.

13.11 Resumen

La calidad de una *webapp* —definida en términos de usabilidad, funcionalidad, confiabilidad, eficiencia, facilidad de mantenimiento, seguridad, escalabilidad y tiempo para llegar al mercado— se introduce durante la etapa de diseño. Para lograr estos atributos de calidad, un buen diseño de la *webapp* debe tener las siguientes características: sencillez, consistencia, identidad, robustez, navegabilidad y atractivo visual. Para lograrlo, la actividad de diseño de la *webapp* se centra en seis distintos elementos del diseño.

El diseño de la interfaz describe la estructura y organización de la interfaz de usuario e incluye una representación de la distribución de la pantalla, una definición de los modos de interacción y una descripción de los mecanismos de navegación. Un conjunto de principios de diseño de la interfaz y el flujo de trabajo del diseño guían el trabajo de diseño de la distribución y los mecanismos de control de la interfaz.

El diseño estético, llamado también *diseño gráfico*, describe el "aspecto y sensación" de la *webapp*, e incluye esquemas de color; distribución geométrica; tamaño del texto, de las fuentes y su colocación; empleo de imágenes y otras decisiones relacionadas con la estética. Un conjunto de lineamientos de diseño gráfico da la base para el enfoque de diseño.

El diseño del contenido define distribución, estructura y bosquejo de todo el contenido que se presenta como parte de la *webapp*, y establece las relaciones entre los objetos del contenido. El diseño del contenido comienza con la representación de sus objetos, así como las asociaciones y relaciones entre ellos. Un conjunto de primitivas de navegación establece la base para el diseño de ésta.

El diseño arquitectónico identifica la estructura general de los hipermedios para la *webapp*, e incluye la arquitectura del contenido y de la *webapp*. Los estilos arquitectónicos para el contenido incluyen estructuras lineales, de malla, jerárquicas y de red. La arquitectura de la *webapp* describe una infraestructura que permite que un sistema o aplicación basado en web cumpla con sus objetivos de negocios.

El diseño de la navegación representa el flujo de ésta entre los objetos de contenido y todas las funciones de la *webapp*. La semántica de la navegación se define, describiendo un conjunto de unidades semánticas de navegación. Cada unidad está compuesta por formas de navegación, así como vínculos y nodos para ello. La sintaxis de navegación ilustra los mecanismos utilizados para navegar descritos como parte de la semántica.

El diseño de los componentes desarrolla la lógica de procesamiento detallada que se requiere para implementar componentes funcionales que desarrollen una función completa de la *web-*

app. Las técnicas de diseño descritas en el capítulo 10 son aplicables para la ingeniería de los componentes de la *webapp*.

El Método de Diseño de Hipermedios Orientado a Objetos (MDHOO) es una de varias propuestas para hacer el diseño de *webapps*. El MDHOO sugiere un proceso que incluye diseño conceptual, diseño de la navegación, diseño abstracto de la interfaz y la implementación.

Problemas y puntos por evaluar

13.1. ¿Por qué es insuficiente para elaborar *webapps* la filosofía de diseño del "ideal artístico"? ¿Hay algún caso en el que ésa sea la filosofía por seguir?

13.2. En este capítulo se selecciona un conjunto amplio de atributos de la calidad de las *webapps*. Seleccione las tres que crea que son las más importantes y construya un argumento que explique por qué debe hacerse énfasis en cada una durante el trabajo de diseño de *webapps*.

13.3. Agregue al menos cinco preguntas adicionales a la Lista de Verificación del Diseño de *webapps* que se presentó en la sección 13.1.

13.4. El lector es un diseñador de *webapps* de *Corporación de Aprendizaje del Futuro*, compañía de aprendizaje a distancia. Trata de implementar un "motor de aprendizaje" basado en internet que permita entregar el contenido de un curso a los estudiantes. El motor de aprendizaje brinda la infraestructura básica para entregar el contenido del aprendizaje sobre cualquier tema (los diseñadores del contenido prepararán el que sea apropiado). Desarrolle el diseño de un prototipo de interfaz para el motor de aprendizaje.

13.5. ¿Cuál es el sitio web de estética más agradable que usted haya visitado y por qué?

13.6. Considere el objeto de contenido **Orden**, generado una vez que un usuario de **CasaSeguraAsegurada.com** haya terminado la selección de todos los componentes y esté listo para finalizar su compra. Desarrolle una descripción UML para **Orden**, así como todas las representaciones del diseño que sean apropiadas.

13.7. ¿Cuál es la diferencia entre la arquitectura del contenido y la de una *webapp*?

13.8. Reconsidere el "motor de aprendizaje" de *Aprendizaje del Futuro* que se describió en el problema 13.4, seleccione una arquitectura del contenido que resulte apropiada para la *webapp*. Analice el porqué de su selección.

13.9. Utilice UML para desarrollar tres o cuatro representaciones del diseño de objetos de contenido que se encontrarían al diseñar el "motor de aprendizaje" descrito en el problema 13.4.

13.10. Investigue un poco acerca de la arquitectura de controlador de vista del modelo (CVM) y decida si sería apropiada para la *webapp* del "motor de aprendizaje" del problema 13.4.

13.11. ¿Cuál es la diferencia entre la sintaxis de navegación y la semántica de ésta?

13.12. Defina dos o tres USN para la *webapp* **CasaSeguraAsegurada.com**. Describa con detalle cada una.

13.13. Escriba un texto breve sobre un método de diseño de hipermedios que no sea MDHOO.

Lecturas adicionales y fuentes de información

Van Duyne *et al.* (*The Design of Sites*, 2a. ed., Prentice Hall, 2007) escribieron un libro exhaustivo que cubre la mayoría de aspectos importantes del proceso de diseño de *webapps*. Cubre con detalle los modelos del proceso de diseño y los patrones de diseño. Wodtke (*Information Architecture*, New Riders Publishing, 2003), Rosenfeld y Morville (*Information Architecture for the World Wide Web*, O'Reilly & Associates, 2002), y Reiss (*Practical Information Architecture*, Addison-Wesley, 2000) abordan la arquitectura del contenido y otros temas.

Aunque se han escrito cientos de libros sobre el "diseño web", son muy pocos los que estudian métodos técnicos significativos para hacer el trabajo de diseño. En el mejor de los casos, presentan varios lineamientos útiles para el diseño de *webapps*, dan ejemplos de páginas web y de programación en Java y analizan los detalles técnicos importantes para implementar *webapps* modernas. Entre los representantes de esta cate-

goría están los libros de Sklar (*Principles of Web Design*, 4a. ed., Course Technology, 2008), McIntire (*Visual Design for the Modern Web*, New Riders Press, 2007), Niederst (*Web Design in a Nutshell*, 3a. ed., O-Reilly, 2006), Eccher (*Advanced Professional Web Design*, Charles River Media, 2006), Cederholm (*Bulletproof Web Design*, New Riders Press, 2005) y Shelly *et al.* (*Web Design*, 2a. ed., Course Technology, 2005). El estudio enciclopédico de Powell [Pow02] y el profundo análisis de Nielsen [Nie00] sobre el diseño también son útiles en cualquier biblioteca.

Los libros de Beaird (*The Principles of Beautiful Web Design*, SitePoint, 2007), Clarke y Holzschlag (*Transcending CSS: The Fine Art of Web Design*, New Riders Press, 2006) y Golbeck (*Art Theory for Web Design*, Addison Wesley, 2005), hacen énfasis en el diseño estético y son una lectura benéfica para los profesionales con poca experiencia en el tema.

El punto de vista ágil del diseño (y otros temas) de *webapps* es presentado por Wallace *et al.* (*Extreme Programming for Web Projects*, Addison-Wesley, 2003). Conallen (*Building Web Applications with UML,* 2a. ed., Addison-Wesley, 2002) y Rosenberg y Scott (*Applying Use-Case Driven Object Modeling with UML*, Addison-Wesley, 2001) presentan ejemplos detallados de *webapps* modeladas con el empleo de UML.

En el contexto de libros escritos acerca de ambientes específicos de desarrollo, también se mencionan técnicas de diseño. Los lectores interesados en ello deben estudiar textos sobre HTML, CSS, J2EE, Java, .NET, XML, Perl, Ruby on Rails, Ajax y varias aplicaciones empleadas para crear *webapps* (*Dreamweaver, HomePage, Frontpage, GoLive, MacroMedia Flash,* etc.) con trucos de diseño útiles.

En internet hay una amplia variedad de fuentes de información sobre diseño de *webapps*. En el sitio web del libro, se encuentra una lista actualizada de referencias en la red mundial que son relevantes para el diseño de *webapps*: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

PARTE

Tres

ADMINISTRACIÓN DE LA CALIDAD

En esta parte de *Ingeniería del software*, aprenderá los principios, conceptos y técnicas que se aplican para administrar y controlar la calidad del software. En los próximos capítulos se responderán preguntas como las siguientes:

- ¿Cuáles son las características generales del software de alta calidad?
- ¿Cómo se revisa la calidad y de qué manera se llevan a cabo revisiones eficaces?
- ¿En qué consiste el aseguramiento de la calidad del software?
- ¿Qué estrategias son aplicables para probar el software?
- ¿Qué métodos se utilizan para diseñar casos de prueba eficaces?
- ¿Hay métodos realistas que aseguren que el software es correcto?
- ¿Cómo pueden administrarse y controlarse los cambios que siempre ocurren cuando se elabora el software?
- ¿Qué medidas y unidades de medición se usan para evaluar la calidad de los modelos de requerimientos y diseño, código fuente y casos de prueba?

Una vez respondidas estas preguntas, el lector estará mejor preparado para asegurar que se ha producido software de alta calidad.

CAPÍTULO

14 Conceptos de calidad

Conceptos clave

acciones de administración . . 349
calidad 339
costo de la calidad. 346
dilema de la calidad. 345
dimensiones de la calidad. . . 341
factores de la calidad. 342
punto de vista cuantitativo . 344
responsabilidad. 348
riesgos. 348
seguridad 349
suficientemente bueno. 345

El redoble de tambores para mejorar la calidad del software comenzó tan luego que éste empezó a integrarse en cada faceta de nuestras vidas. En la década de 1990, las principales corporaciones reconocieron que cada año se desperdiciaban miles de millones de dólares en software que no tenía las características ni la funcionalidad que se habían prometido. Lo que era peor, tanto el gobierno como la industria se preocupaban por la posibilidad de que alguna falla de software pudiera afectar infraestructura importante y provocara pérdidas de decenas de miles de millones de dólares. Al despuntar el nuevo siglo, *CIO Magazine* [Lev01] dio la alerta: "Dejemos de desperdiciar $78 mil millones de dólares al año", y lamentaba el hecho de que "las empresas estadounidenses gastan miles de millones de dólares en software que no hace lo que se supone que debe hacer". *InformationWeek* [Ric01] se hizo eco de la misma preocupación:

> A pesar de las buenas intenciones, el código defectuoso sigue siendo el duende de la industria del software, es responsable hasta de 45% del tiempo que están fuera los sistemas basados en computadoras y costó a las empresas estadounidenses alrededor de $100 mil millones de dólares el último año en pérdidas de productividad y reparaciones, afirma Standish Group, empresa de investigación de mercados. Eso no incluye el costo que implica perder a los clientes disgustados. Como los productores de tecnologías de información escriben aplicaciones que se basan en software empacado en infraestructura, el código defectuoso también puede inutilizar aplicaciones personalizadas...
>
> Pero, ¿cuán malo es el software defectuoso? Las respuestas varían, mas los expertos dicen que sólo se requiere de tres a cuatro defectos por cada 1 000 líneas de código para que un programa tenga mal desempeño. Hay que pensar que la mayoría de los programadores cometen un error en cada 10 líneas de código que escriben, lo que, multiplicado por los millones de líneas que hay en muchos productos comerciales, permite imaginar que la corrección de los errores cuesta a los vendedores de software al menos la mitad de sus presupuestos de desarrollo durante las pruebas. ¿Comprende lo que esto significa?

Una mirada rápida

¿Qué es? La respuesta no es tan fácil como quizá se piense. La calidad se reconoce cuando se ve, por lo que puede ser algo difícil de definir. Pero para el software de computadora, la calidad es algo que debe definirse, y eso es lo que haremos en este capítulo.

¿Quién lo hace? Los involucrados en el proceso del software —ingenieros, gerentes y todos los participantes— son los responsables de la calidad.

¿Por qué es importante? Puede hacerse bien o puede repetirse. Si un equipo de software hace énfasis en la calidad de todas las actividades de la ingeniería de software, se reduce el número de repeticiones que deben hacerse. Esto da como resultado menores costos y, lo que es más importante, mejora el tiempo de llegada al mercado.

¿Cuáles son los pasos? Para lograr software de alta calidad, deben ocurrir cuatro actividades: usar procesos y prácticas probados de la ingeniería de software, administrar bien el proyecto, realizar un control de calidad exhaustivo y contar con infraestructura de aseguramiento de la calidad.

¿Cuál es el producto final? Software que satisface las necesidades del consumidor, con un desempeño apropiado y confiable, y que agrega valor para todos los que lo utilizan.

¿Cómo me aseguro de que lo hice bien? Hay que dar seguimiento a la calidad, estudiando los resultados de todas las actividades de control de calidad y midiendo ésta con el estudio de los errores antes de la entrega y de los defectos detectados en el campo.

En 2005, *ComputerWorld* [Hil05] se quejaba de que "el mal software es una plaga en casi todas las organizaciones que emplean computadoras, lo que ocasiona horas de trabajo perdidas por el tiempo que están fuera de uso las máquinas, por datos perdidos o corrompidos, oportunidades de venta perdidas, costos elevados de apoyo y mantenimiento, y poca satisfacción del cliente. Un año después, *InfoWorld* [Fos06] escribió acerca del "lamentable estado de la calidad del software" e informaba que el problema de la calidad no había mejorado.

Actualmente, la calidad del software es preocupante, pero, ¿de quién es la culpa? Los clientes culpan a los desarrolladores, pues afirman que sus prácticas descuidadas producen software de mala calidad. Los desarrolladores culpan a los clientes (y a otros participantes) con la afirmación de que las fechas de entrega irracionales y un flujo continuo de cambios los obligan a entregar software antes de haber sido validado por completo. ¿Quién tiene la razón? *Ambos*, y ése es el problema. En este capítulo se analiza el concepto de calidad del software y por qué es útil estudiarlo con seriedad siempre que se apliquen prácticas de ingeniería de software.

14.1 ¿Qué es calidad?

En su libro místico, *El zen y el arte del mantenimiento de la motocicleta*, Robert Persig [Per74] comenta lo siguiente acerca de lo que llamamos *calidad*:

> Calidad... sabes lo que es, pero no sabes lo que es. Pero eso es una contradicción. Algunas cosas son mejores que otras; es decir, tienen más calidad. Pero cuando tratas de decir lo que es la calidad, además de las cosas que la tienen, todo se desvanece... No hay nada de qué hablar. Pero si no puede decirse qué es Calidad, ¿cómo saber lo que es, o incluso saber que existe? Si nadie sabe lo que es, entonces, para todos los propósitos prácticos, no existe en absoluto. Pero para todos los propósitos prácticos, en realidad sí existe. ¿En qué otra cosa se basan las calificaciones? ¿Por qué paga fortunas la gente por algunos artículos y tira otros a la basura? Es obvio que algunas cosas son mejores que otras... pero, ¿en qué son mejores? Y así damos vueltas y más vueltas, ruedas de metal que patinan sin nada en lo que hagan tracción. ¿Qué demonios es la Calidad? ¿Qué es?

Es cierto: ¿qué es?

¿Cuáles son las diferentes maneras en las que puede verse la calidad?

En un nivel algo pragmático, David Garvin [Gar84], de Harvard Business School, sugiere que "la calidad es un concepto complejo y de facetas múltiples" que puede describirse desde cinco diferentes puntos de vista. El *punto de vista trascendental* dice (como Persig) que la calidad es algo que se reconoce de inmediato, pero que no es posible definir explícitamente. El *punto de vista del usuario* concibe la calidad en términos de las metas específicas del usuario final. Si un producto las satisface, tiene calidad. El *punto de vista del fabricante* la define en términos de las especificaciones originales del producto. Si éste las cumple, tiene calidad. El *punto de vista del producto* sugiere que la calidad tiene que ver con las características inherentes (funciones y características) de un producto. Por último, el *punto de vista basado en el valor* la mide de acuerdo con lo que un cliente está dispuesto a pagar por un producto. En realidad, la calidad incluye todo esto y más.

Cita:

"La gente olvida cuán rápido hiciste un trabajo, pero siempre recuerda cuán bien lo realizaste."

Howard Newton

La *calidad del diseño* se refiere a las características que los diseñadores especifican para un producto. El tipo de materiales, tolerancias y especificaciones del desempeño, todo contribuye a la calidad del diseño. Si se utilizan mejores materiales, tolerancias más estrictas y se especifican mayores niveles de desempeño, la calidad del diseño de un producto se incrementa si se fabrica de acuerdo con las especificaciones.

En el desarrollo del software, la *calidad del diseño* incluye el grado en el que el diseño cumple las funciones y características especificadas en el modelo de requerimientos. La *calidad de la conformidad* se centra en el grado en el que la implementación se apega al diseño y en el que el sistema resultante cumple sus metas de requerimientos y desempeño.

Pero, ¿son la calidad del diseño y de la conformidad los únicos aspectos que deben considerar los ingenieros de software? Robert Glass [Gla98] afirma que es mejor plantear una relación más intuitiva:

satisfacción del usuario = producto que funciona + buena calidad + entrega dentro del presupuesto y plazo

En última instancia, Glass sostiene que la calidad es importante, pero que si el usuario no está satisfecho, nada de lo demás importa. DeMarco [DeM98] refuerza esta opinión al decir que "la calidad de un producto está en función de cuánto cambia al mundo para bien". Este punto de vista de la calidad afirma que si un producto de software beneficia mucho a los usuarios finales, éstos se mostrarán dispuestos a tolerar problemas ocasionales de confiabilidad o desempeño.

14.2 Calidad del software

Incluso los desarrolladores de software más experimentados estarán de acuerdo en que obtener software de alta calidad es una meta importante. Pero, ¿cómo se define la calidad del *software*? En el sentido más general se define[1] como: *Proceso eficaz de software que se aplica de manera que crea un producto útil que proporciona valor medible a quienes lo producen y a quienes lo utilizan.*

Hay pocas dudas acerca de que la definición anterior podría modificarse o ampliarse en un debate sin fin. Para propósitos de este libro, la misma sirve a fin de enfatizar tres puntos importantes:

1. Un proceso *eficaz de software* establece la infraestructura que da apoyo a cualquier esfuerzo de elaboración de un producto de software de alta calidad. Los aspectos de administración del proceso generan las verificaciones y equilibrios que ayudan a evitar que el proyecto caiga en el caos, contribuyente clave de la mala calidad. Las prácticas de ingeniería de software permiten al desarrollador analizar el problema y diseñar una solución sólida, ambas actividades críticas de la construcción de software de alta calidad. Por último, las actividades sombrilla, tales como administración del cambio y revisiones técnicas, tienen tanto que ver con la calidad como cualquier otra parte de la práctica de la ingeniería de software.
2. Un *producto útil* entrega contenido, funciones y características que el usuario final desea; sin embargo, de igual importancia es que entrega estos activos en forma confiable y libre de errores. Un producto útil siempre satisface los requerimientos establecidos en forma explícita por los participantes. Además, satisface el conjunto de requerimientos (por ejemplo, la facilidad de uso) con los que se espera que cuente el software de alta calidad.
3. Al *agregar valor para el productor y para el usuario* de un producto, el software de alta calidad proporciona beneficios a la organización que lo produce y a la comunidad de usuarios finales. La organización que elabora el software obtiene valor agregado porque el software de alta calidad requiere un menor esfuerzo de mantenimiento, menos errores que corregir y poca asistencia al cliente. Esto permite que los ingenieros de software dediquen más tiempo a crear nuevas aplicaciones y menos a repetir trabajos mal hechos. La comunidad de usuarios obtiene valor agregado porque la aplicación provee una capacidad útil en forma tal que agiliza algún proceso de negocios. El resultado final es 1) mayores utilidades por el producto de software, 2) más rentabilidad cuando una

1 Esta definición ha sido adaptada de [Bes04] y sustituye aquélla más orientada a la manufactura presentada en ediciones anteriores de este libro.

aplicación apoya un proceso de negocios y 3) mejor disponibilidad de información, que es crucial para el negocio.

14.2.1 Dimensiones de la calidad de Garvin

David Garvin [Gar87] sugiere que la calidad debe tomarse en cuenta, adoptando un punto de vista multidimensional que comience con la evaluación de la conformidad y termine con una visión trascendental (estética). Aunque las ocho dimensiones de Garvin de la calidad no fueron desarrolladas específicamente para el software, se aplican a la calidad de éste:

Calidad del desempeño. ¿El software entrega todo el contenido, las funciones y las características especificadas como parte del modelo de requerimientos, de manera que da valor al usuario final?

Calidad de las características. ¿El software tiene características que sorprenden y agradan la primera vez que lo emplean los usuarios finales?

Confiabilidad. ¿El software proporciona todas las características y capacidades sin fallar? ¿Está disponible cuando se necesita? ¿Entrega funcionalidad libre de errores?

Conformidad. ¿El software concuerda con los estándares locales y externos que son relevantes para la aplicación? ¿Concuerda con el diseño *de facto* y las convenciones de código? Por ejemplo, ¿la interfaz de usuario está de acuerdo con las reglas aceptadas del diseño para la selección de menú o para la entrada de datos?

Durabilidad. ¿El software puede recibir mantenimiento (cambiar) o corregirse (depurarse) sin la generación inadvertida de eventos colaterales? ¿Los cambios ocasionarán que la tasa de errores o la confiabilidad disminuyan con el tiempo?

Servicio. ¿Existe la posibilidad de que el software reciba mantenimiento (cambios) o correcciones (depuración) en un periodo de tiempo aceptablemente breve? ¿El equipo de apoyo puede adquirir toda la información necesaria para hacer cambios o corregir defectos? Douglas Adams [Ada93] hace un comentario irónico que parece pertinente: "La diferencia entre algo que puede salir mal y algo que posiblemente no salga mal es que cuando esto último sale mal, por lo general es imposible corregirlo o repararlo."

Estética. No hay duda de que todos tenemos una visión diferente y muy subjetiva de lo que es estético. Aun así, la mayoría de nosotros estaría de acuerdo en que una entidad estética posee cierta elegancia, un flujo único y una "presencia" obvia que es difícil de cuantificar y que, no obstante, resulta evidente. El software estético tiene estas características.

Percepción. En ciertas situaciones, existen prejuicios que influirán en la percepción de la calidad por parte del usuario. Por ejemplo, si se introduce un producto de software elaborado por un proveedor que en el pasado ha demostrado mala calidad, se estará receloso y la percepción de la calidad del producto tendrá influencia negativa. De manera similar, si un vendedor tiene una reputación excelente se percibirá buena calidad, aun si ésta en realidad no existe.

Las dimensiones de la calidad de Garvin dan una visión "suave" de la calidad del software. Muchas de estas dimensiones (aunque no todas) sólo pueden considerarse de manera subjetiva. Por esta razón, también se necesita un conjunto de factores "duros" de la calidad que se clasifican en dos grandes grupos: 1) factores que pueden medirse en forma directa (por ejemplo, defectos no descubiertos durante las pruebas) y 2) factores que sólo pueden medirse indirectamente (como la usabilidad o la facilidad de recibir mantenimiento). En cada caso deben hacerse mediciones: debe compararse el software con algún dato para llegar a un indicador de la calidad.

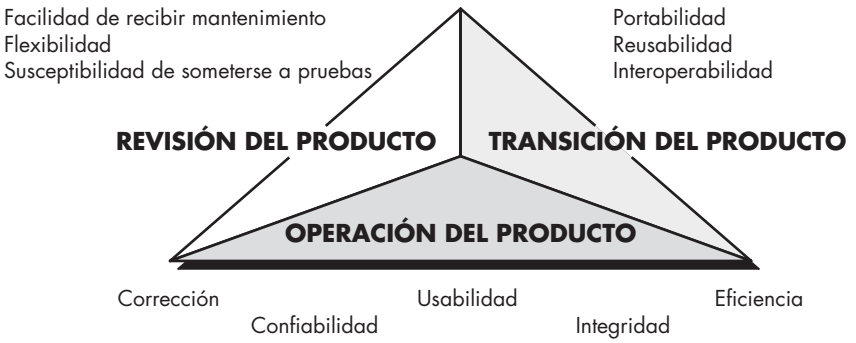


FIGURA 14.1
Factores de la calidad de McCall

14.2.2 Factores de la calidad de McCall

McCall, Richards y Walters [McC77] proponen una clasificación útil de los factores que afectan la calidad del software. Éstos se ilustran en la figura 14.1 y se centran en tres aspectos importantes del producto de software: sus características operativas, su capacidad de ser modificado y su adaptabilidad a nuevos ambientes.

En relación con los factores mencionados en la figura 14.1, McCall *et al.*, hacen las descripciones siguientes:

Corrección. Grado en el que un programa satisface sus especificaciones y en el que cumple con los objetivos de la misión del cliente.

Confiabilidad. Grado en el que se espera que un programa cumpla con su función y con la precisión requerida [debe notarse que se han propuesto otras definiciones más completas de la confiabilidad (véase el capítulo 25)].

Eficiencia. Cantidad de recursos de cómputo y de código requeridos por un programa para llevar a cabo su función.

Integridad. Grado en el que es posible controlar el acceso de personas no autorizadas al software o a los datos.

Usabilidad. Esfuerzo que se requiere para aprender, operar, preparar las entradas e interpretar las salidas de un programa.

Facilidad de recibir mantenimiento. Esfuerzo requerido para detectar y corregir un error en un programa (ésta es una definición muy limitada).

Flexibilidad. Esfuerzo necesario para modificar un programa que ya opera.

Susceptibilidad de someterse a pruebas. Esfuerzo que se requiere para probar un programa a fin de garantizar que realiza la función que se pretende.

Portabilidad. Esfuerzo que se necesita para transferir el programa de un ambiente de sistema de hardware o software a otro.

Reusabilidad. Grado en el que un programa (o partes de uno) pueden volverse a utilizar en otras aplicaciones (se relaciona con el empaque y el alcance de las funciones que lleva a cabo el programa).

Interoperabilidad. Esfuerzo requerido para acoplar un sistema con otro.

Cita:

"La amargura de la mala calidad permanece mucho tiempo después de que ya se ha olvidado la dulzura de haber cumplido el plazo programado."

Karl Weigers (cita sin acreditación)

Es difícil — y, en ciertos casos, imposible— desarrollar mediciones directas[2] de estos factores de la calidad. En realidad, muchas de las unidades de medida definidas por McCall *et al.*, sólo

2 Una *medición directa* implica que hay un solo valor cuantificable que da una indicación directa del atributo en estudio. Por ejemplo, el "tamaño" de un programa se mide directamente, contando el número de sus líneas de código.

pueden obtenerse de manera indirecta. Sin embargo, la evaluación de la calidad de una aplicación por medio de estos factores dará un indicio sólido de ella.

14.2.3 Factores de la calidad ISO 9126

El estándar ISO 9126 se desarrolló con la intención de identificar los atributos clave del software de cómputo. Este sistema identifica seis atributos clave de la calidad:

Funcionalidad. Grado en el que el software satisface las necesidades planteadas según las establecen los atributos siguientes: adaptabilidad, exactitud, interoperabilidad, cumplimiento y seguridad.

Confiabilidad. Cantidad de tiempo que el software se encuentra disponible para su uso, según lo indican los siguientes atributos: madurez, tolerancia a fallas y recuperación.

Usabilidad. Grado en el que el software es fácil de usar, según lo indican los siguientes subatributos: entendible, aprendible y operable.

Eficiencia. Grado en el que el software emplea óptimamente los recursos del sistema, según lo indican los subatributos siguientes: comportamiento del tiempo y de los recursos.

Facilidad de recibir mantenimiento. Facilidad con la que pueden efectuarse reparaciones al software, según lo indican los atributos que siguen: analizable, cambiable, estable, susceptible de someterse a pruebas.

Portabilidad. Facilidad con la que el software puede llevarse de un ambiente a otro según lo indican los siguientes atributos: adaptable, instalable, conformidad y sustituible.

Aunque resulta tentador desarrollar mediciones cuantitativas para los factores de calidad mencionados aquí, también puede crearse una lista de comprobación de atributos que den una indicación sólida de la presencia del factor.

Igual que otros factores de la calidad del software estudiados en las subsecciones anteriores, los factores ISO 9126 no necesariamente conducen a una medición directa. Sin embargo, proporcionan una base útil para hacer mediciones indirectas y una lista de comprobación excelente para evaluar la calidad del sistema.

14.2.4 Factores de calidad que se persiguen

"Cualquier actividad se vuelve creativa cuando a quien la realiza le importa hacerla bien, o mejor."

John Updike

Las dimensiones y factores de la calidad presentados en las secciones 14.2.1 y 14.2.2 se centran en el software como un todo y pueden utilizarse como indicación general de la calidad de una aplicación. Un equipo de software puede desarrollar un conjunto de características de la calidad y las preguntas asociadas correspondientes que demuestren[3] el grado en el que se satisface cada factor. Por ejemplo, McCall identifica la *usabilidad* como un factor importante de la calidad. Si se pidiera revisar una interfaz de usuario para evaluar su usabilidad, ¿cómo se haría? Se comenzaría con los subatributos propuestos por McCall —entendible, aprendible y operable— pero en un sentido práctico: ¿qué significan éstos?

Para hacer la evaluación, se necesita determinar atributos específicos y medibles (o al menos reconocibles) de la interfaz. Por ejemplo [Bro03]:

Intuitiva. Grado en el que la interfaz sigue patrones esperados de uso, de modo que hasta un novato la pueda utilizar sin mucha capacitación.

- ¿La interfaz lleva hacia una comprensión fácil?
- ¿Todas las operaciones son fáciles de localizar e iniciar?
- ¿La interfaz usa una metáfora reconocible?
- ¿La entrada está especificada de modo que economiza el uso del teclado o del ratón?

3 Estas características y preguntas se plantearían como parte de la revisión del software (véase el capítulo 15).

- ¿La entrada sigue las tres reglas de oro? (véase el capítulo 11)
- ¿La estética ayuda a la comprensión y uso?

Eficiencia. Grado en el que es posible localizar o iniciar las operaciones y la información.

- ¿La distribución y estilo de la interfaz permite que un usuario introduzca con eficiencia las operaciones y la información?
- ¿Una secuencia de operaciones (o entrada de datos) puede realizarse con economía de movimientos?
- ¿Los datos de salida o el contenido están presentados de modo que se entienden de inmediato?
- ¿Las operaciones jerárquicas están organizadas de manera que minimizan la profundidad con la que debe navegar el usuario para hacer que alguna se ejecute?

Robustez. Grado en el que el software maneja entradas erróneas de datos o en el que se presenta interacción inapropiada por parte del usuario.

- ¿El software reconocerá el error si entran datos en el límite de lo permitido o más allá y, lo que es más importante, continuará operando sin fallar ni degradarse?
- ¿La interfaz reconocerá los errores cognitivos o de manipulación y guiará en forma explícita al usuario de vuelta al camino correcto?
- ¿La interfaz da un diagnóstico y guía útiles cuando se descubre una condición de error (asociada con la funcionalidad del software)?

Riqueza. Grado en el que la interfaz provee un conjunto abundante de características.

- ¿Puede personalizarse la interfaz según las necesidades específicas del usuario?
- ¿La interfaz tiene gran capacidad para permitir al usuario identificar una secuencia de operaciones comunes con una sola acción o comando?

A medida que se desarrolla el diseño de la interfaz, el equipo del software revisa el prototipo del diseño y plantea las preguntas anteriores. Si la respuesta a la mayor parte de éstas es "sí", es probable que la interfaz de usuario sea de buena calidad. Para cada factor de la calidad que se desee evaluar se desarrollan preguntas similares.

14.2.5 Transición a un punto de vista cuantitativo

En las subsecciones anteriores se presentaron varios factores cualitativos para la "medición" de la calidad del software. La comunidad de la ingeniería de software trata de obtener mediciones precisas de la calidad de éste y a veces se ve frustrada por la naturaleza subjetiva de la actividad. Cavano y McCall [Cav78] analizan esta situación:

> La determinación de la calidad es un factor clave en los eventos cotidianos: concursos para catar vinos, eventos deportivos [como gimnasia], competencias de talento, etc. En estas situaciones se juzga la calidad del modo más fundamental y directo: la comparación directa de objetos en condiciones idénticas y con conceptos predeterminados. El vino se juzga de acuerdo con su claridad, color, buqué, sabor, etc. Sin embargo, este tipo de juicio es muy subjetivo; para que tenga algún valor, debe ser hecho por un experto.
>
> La subjetividad y la especialización también se aplican a la determinación de la calidad del software. Para ayudar a resolver este problema, es necesario tener una definición más precisa de la calidad del software, así como una forma de realizar mediciones cuantitativas de la calidad a fin de hacer análisis objetivos... Como no existe algo parecido al conocimiento absoluto, no debe esperarse medir con toda exactitud la calidad del software, porque toda medición es imperfecta. Jacob Bronkowski

> describió esta paradoja del conocimiento del modo siguiente: "Año con año desarrollamos instrumentos más precisos para observar la naturaleza con más nitidez. Y cuando vemos las observaciones, nos decepcionamos porque son borrosas y sentimos que son tan inciertas como siempre".

En el capítulo 23 se presenta un conjunto de unidades de medida aplicables a la evaluación cuantitativa de la calidad del software. En todos los casos, las unidades representan mediciones indirectas, es decir, nunca miden realmente la *calidad*, sino alguna manifestación de ella. El factor que complica todo es la relación precisa entre la variable que se mide y la calidad del software.

14.3 El dilema de la calidad del software

Cuando se enfrente al dilema de la calidad (y todos lo hacen en un momento u otro), trate de alcanzar el balance: suficiente esfuerzo para producir una calidad aceptable sin que sepulte al proyecto.

En una entrevista [Ven03] publicada en la web, Bertrand Meyer analiza lo que se denomina el *dilema de la calidad:*

> Si produce un sistema de software de mala calidad, usted pierde porque nadie lo querrá comprar. Por otro lado, si dedica un tiempo infinito, demasiado esfuerzo y enormes sumas de dinero para obtener un elemento perfecto de software, entonces tomará tanto tiempo terminarlo y será tan caro de producir que de todos modos quedará fuera del negocio. En cualquier caso, habrá perdido la ventana de mercado, o simplemente habrá agotado sus recursos. De modo que las personas de la industria tratan de situarse en ese punto medio mágico donde el producto es suficientemente bueno para no ser rechazado de inmediato, no en la evaluación, pero tampoco es un objeto perfeccionista ni con demasiado trabajo que lo convierta en algo que requiera demasiado tiempo o dinero para ser terminado.

Es correcto afirmar que los ingenieros de software deben tratar de producir sistemas de alta calidad. Es mejor aplicar buenas prácticas al intento de lograrlo. Pero la situación descrita por Meyer proviene de la vida real y representa un dilema incluso para las mejores organizaciones de ingeniería de software.

14.3.1 Software "suficientemente bueno"

En palabras sencillas, si damos por válido el argumento de Meyer, ¿es aceptable producir software "suficientemente bueno"? La respuesta a esta pregunta debe ser "sí", porque las principales compañías de software lo hacen a diario. Crean software con errores detectados y lo distribuyen a una gran población de usuarios finales. Reconocen que algunas de las funciones y características de la versión 1.0 tal vez no sean de la calidad más alta y planean hacer mejoras en la versión 2.0. Hacen esto, sabiendo que algunos clientes se quejarán; reconocen que el tiempo para llegar al mercado actúa contra la mejor calidad, y liberan el software, siempre y cuando el producto entregado sea "suficientemente bueno".

Exactamente, ¿qué significa "suficientemente bueno"? El software suficientemente bueno contiene las funciones y características de alta calidad que desean los usuarios, pero al mismo tiempo tiene otras más oscuras y especializadas que contienen errores conocidos. El vendedor de software espera que la gran mayoría de usuarios finales perdone los errores gracias a que estén muy contentos con la funcionalidad de la aplicación.

Esta idea resulta familiar para muchos lectores. Si usted es uno de ellos, le pido que considere algunos de los argumentos contra lo "suficientemente bueno".

Es verdad que lo "suficientemente bueno" puede funcionar en ciertos dominios de aplicación y para unas cuantas compañías grandes de software. Después de todo, si una empresa tiene un presupuesto enorme para mercadotecnia y convence a suficientes personas de que compren la versión 1.0, habrá tenido éxito en capturarlos. Como ya se dijo, puede sostener que en las versiones posteriores mejorará la calidad. Al entregar la versión 1.0 suficientemente buena, habrá capturado al mercado.

Si el lector trabaja para una compañía pequeña, debe tener cuidado con esta filosofía. Al entregar un producto suficientemente bueno (defectuoso), corre el riesgo de causar un daño permanente a la reputación de su compañía. Tal vez nunca tenga la oportunidad de entregar una versión 2.0 porque los malos comentarios quizá ocasionen que las ventas se desplomen y que la empresa desaparezca.

Si trabaja en ciertos dominios de aplicación (por ejemplo, software incrustado en tiempo real) o si construye software de aplicación integrado con hardware (como el software automotriz o de telecomunicaciones), entregar software con errores conocidos es una negligencia y deja expuesta a su compañía a litigios costosos. En ciertos casos, incluso, puede ser un delito. ¡Nadie quiere tener software suficientemente bueno en los aviones!

Así que proceda con cautela si piensa que lo "suficientemente bueno" es un atajo que puede resolver los problemas de calidad de su software. Tal vez funcione, pero sólo para unos cuantos y en un conjunto limitado de dominios de aplicación.[4]

14.3.2 El costo de la calidad

El argumento es algo parecido a esto: *sabemos que la calidad es importante, pero cuesta tiempo y dinero —demasiado tiempo y dinero— lograr el nivel de calidad en el software que en realidad queremos.* Visto así, este argumento parece razonable (véanse los comentarios anteriores de Meyer en esta sección). No hay duda de que la calidad tiene un costo, pero la mala calidad también lo tiene —no sólo para los usuarios finales que deban vivir con el software defectuoso, sino también para la organización del software que lo elaboró y que debe darle mantenimiento—. La pregunta real es ésta: *¿por cuál costo debemos preocuparnos?* Para responder a esta pregunta debe entenderse tanto el costo de tener calidad como el del software de mala calidad.

El *costo de la calidad* incluye todos los costos en los que se incurre al buscar la calidad o al realizar actividades relacionadas con ella y los costos posteriores de la falta de calidad. Para entender estos costos, una organización debe contar con unidades de medición que provean el fundamento del costo actual de la calidad, que identifiquen las oportunidades para reducir dichos costos y que den una base normalizada de comparación. El costo de la calidad puede dividirse en los costos que están asociados con la prevención, la evaluación y la falla.

No tema incurrir en costos significativos por la prevención. Esté seguro de que su inversión tendrá un rendimiento excelente.

Los *costos de prevención* incluyen lo siguiente: 1) el costo de las actividades de administración requeridas para planear y coordinar todas las actividades de control y aseguramiento de la calidad, 2) el costo de las actividades técnicas agregadas para desarrollar modelos completos de los requerimientos y del diseño, 3) los costos de planear las pruebas y 4) el costo de toda la capacitación asociada con estas actividades.

Los *costos de evaluación* incluyen las actividades de investigación de la condición del producto la "primera vez" que pasa por cada proceso. Algunos ejemplos de costos de evaluación incluyen los siguientes:

- El costo de efectuar revisiones técnicas (véase el capítulo 15) de los productos del trabajo de la ingeniería de software.
- El costo de recabar datos y unidades de medida para la evaluación (véase el capítulo 23)
- El costo de hacer las pruebas y depurar (véanse los capítulos 18 a 21)

Los *costos de falla* son aquellos que se eliminarían si no hubiera errores antes o después de enviar el producto a los consumidores. Los costos de falla se subdividen en internos y externos. Se incurre en *costos internos de falla* cuando se detecta un error en un producto antes del envío. Los costos internos de falla incluyen los siguientes:

4 Un análisis útil de los pros y contras del software "suficientemente bueno" se encuentra en [Bre02].

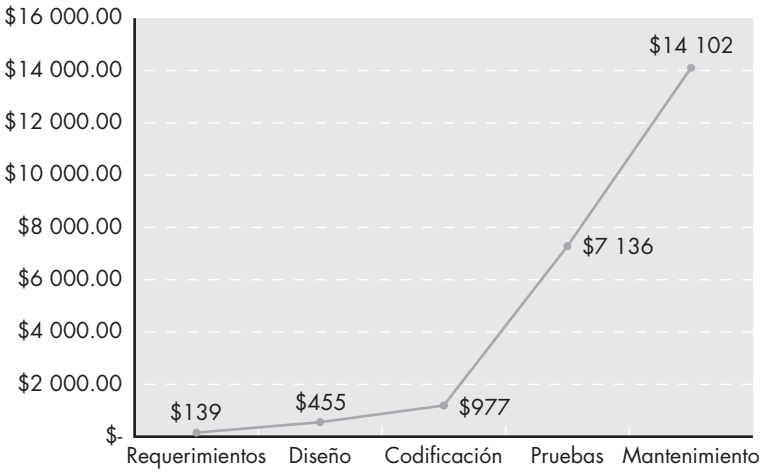


FIGURA 14.2
Costo relativo de corregir errores y defectos (cifras en dólares estadounidenses)
Fuente: Adaptado de [Boe01b].

- El costo requerido por efectuar repeticiones (reparaciones para corregir un error).
- El costo en el que se incurre cuando una repetición genera inadvertidamente efectos colaterales que deban mitigarse.
- Los costos asociados con la colección de las unidades de medida de la calidad que permitan que una organización evalúe los modos de la falla.

Cita:

"Toma menos tiempo hacer algo bien que explicar por qué se hizo mal."

H. W. Longfellow

Los *costos externos de falla* se asocian con defectos encontrados después de que el producto se envió a los consumidores. Algunos ejemplos de costos externos de falla son los de solución de quejas, devolución y sustitución del producto, ayuda en línea y trabajo asociado con la garantía. La mala reputación y la pérdida resultante de negocios es otro costo externo de falla que resulta difícil de cuantificar y que, sin embargo, es real. Cuando se produce software de mala calidad, suceden cosas malas.

En lo que constituye una acusación contra los desarrolladores de software que se rehúsan a considerar los costos de falla externos, Cem Kaner [Kan95] afirma lo siguiente:

> Muchos de los costos de falla externos, tales como los fondos de comercialización, son difíciles de cuantificar, por lo que muchas compañías los ignoran cuando calculan sus relaciones costo-beneficio. Otros costos externos de falla pueden reducirse (al dar un apoyo barato debido a la mala calidad después de hacer la venta, o al cobrar el apoyo a los consumidores) sin que se incremente la satisfacción del cliente. Al ignorar los costos que los malos productos generan a nuestros compradores, los ingenieros de la calidad estimulan una toma de decisiones que los hace víctimas en lugar de satisfacerlos.

Como es de esperar, los costos relacionados con la detección y la corrección de errores o defectos se incrementan en forma abrupta cuando se pasa de la prevención a la detección, a la falla interna y a la externa. La figura 14.2, basada en datos obtenidos por Boehm y Basili [Boe01b] y elaborada por Cigital, Inc. [Cig07], ilustra este fenómeno.

El costo promedio de la industria por corregir un defecto durante la generación de código es aproximadamente de US$977 por error. El promedio del costo en el que incurre la industria por corregir el mismo error si se descubre durante las pruebas del sistema es de US$7 136. Cigital, Inc. [Cig07] tome en cuenta que una aplicación grande contiene 200 errores introducidos durante la codificación.

> De acuerdo con datos promedio, el costo de encontrar y corregir defectos durante la fase de codificación es de US$977 por defecto. Entonces, el costo total por corregir los 200 errores "críticos" durante esta fase es de (200 × US$977) US$195 400, aproximadamente.

> Los datos promedio de la industria indican que el costo de encontrar y corregir defectos durante la fase de pruebas del sistema es de US$7 136 por cada uno. En este caso, si se supone que en dicha fase se descubren aproximadamente 50 defectos críticos (tan sólo 25% de los descubiertos por Cigital en la fase de codificación), el costo de encontrarlos y corregirlos (50 × US$7 136) sería aproximadamente de US$356 800. Esto también habría resultado en 150 errores críticos no detectados ni corregidos. El costo de encontrar y corregir estos 150 defectos en la fase de mantenimiento (150 × US$14 102) habría sido de US$2 115 300. Entonces, el costo total de encontrar y corregir los 200 defectos (US$2 115 300 + US$356 800) después de la fase de codificación habría sido de US$2 472 100.

Aun si la organización de software tuviera costos que fueran la mitad del promedio de la industria (la mayor parte de compañías no tiene idea de cuáles son sus costos), los ahorros asociados con el control de calidad temprano y las actividades para su aseguramiento (efectuadas durante el análisis de los requerimientos y el diseño) serían notables.

14.3.3 Riesgos

En el capítulo 1 de este libro se dijo que "la gente basa su trabajo, confort, seguridad, entretenimiento, decisiones y su propia vida, en software de cómputo. Más vale que esté bien hecho". La implicación es que el software de mala calidad aumenta los riesgos tanto para el desarrollador como para el usuario final. En la subsección anterior se analizó uno de dichos riesgos (el costo). Pero lo perjudicial de las aplicaciones mal diseñadas e implementadas no siempre se mide en dólares y tiempo. Un ejemplo extremo [Gag04] servirá para ilustrar esto.

En el mes de noviembre de 2000, en un hospital de Panamá, 28 pacientes recibieron dosis masivas de rayos gama durante su tratamiento contra diversos tipos de cáncer. En los meses que siguieron, 5 de estos pacientes murieron por envenenamiento radiactivo y 15 más sufrieron complicaciones serias. ¿Qué fue lo que ocasionó esta tragedia? Un paquete de software, desarrollado por una compañía estadounidense, que fue modificado por técnicos del hospital para calcular las dosis de radiación para cada paciente.

Los tres médicos panameños que "pellizcaron" el software para que diera capacidad adicional fueron acusados de asesinato en segundo grado. La empresa de Estados Unidos enfrentó litigios serios en los dos países. Gage y McCormick comentan lo siguiente:

> Éste no es un relato para prevenir a los médicos, aun cuando luchen por estar fuera de la cárcel si no entienden o hacen mal uso de la tecnología. Tampoco es la narración de cómo pueden salir heridos, o algo peor, los seres humanos a causa del software mal diseñado o poco explicado, aunque hay muchos ejemplos al respecto. Ésta es la alerta para cualquier creador de programas de cómputo: la calidad del software importa, las aplicaciones deben ser a prueba de tontos y el código mal desplegado —ya sea incrustado en el motor de un automóvil, un brazo robótico o un dispositivo de curación en un hospital— puede matar.

La mala calidad conlleva riesgos, algunos muy serios.

14.3.4 Negligencia y responsabilidad

La historia es muy común. Una entidad gubernamental o corporativa contrata a una compañía importante de desarrollo de software o a una consultoría para que analice los requerimientos y luego diseñe y construya un "sistema" basado en software para apoyar alguna actividad de importancia. El sistema debe auxiliar a una función corporativa principal (como la administración de pensiones) o a alguna función gubernamental (por ejemplo, la administración del cuidado de la salud o los créditos hipotecarios).

El trabajo comienza con las mejores intenciones por ambas partes, pero en el momento en el que el sistema se entrega, las cosas han marchado mal. El sistema va retrasado, no da los resultados y funciones deseadas, comete errores y no cuenta con la aprobación del cliente. Comienzan los litigios.

En la mayor parte de los casos, el cliente afirma que el desarrollador ha sido negligente (en cuanto a la manera en la que aplicó las prácticas del software), por lo que no merece el pago. Es frecuente que el desarrollador diga que el cliente ha cambiado repetidamente sus requerimientos y trastornado de diversas maneras los acuerdos para el trabajo. En cualquier caso, es la calidad del sistema lo que está en entredicho.

14.3.5 Calidad y seguridad

A medida que aumenta la importancia crítica de los sistemas y aplicaciones basados en web, la seguridad de las aplicaciones se ha vuelto más importante. En pocas palabras, el software que no tiene alta calidad es fácil de penetrar por parte de intrusos y, en consecuencia, el software de mala calidad aumenta indirectamente el riesgo de la seguridad, con todos los costos y problemas que eso conlleva.

En una entrevista para *ComputerWorld*, el autor y experto en seguridad Gary McGraw comenta lo siguiente [Wil05]:

> La seguridad del software se relaciona por completo con la calidad. Debe pensarse en seguridad, confiabilidad, disponibilidad y dependencia, en la fase inicial, en la de diseño, en la de arquitectura, pruebas y codificación, durante todo el ciclo de vida del software [proceso]. Incluso las personas conscientes del problema de la seguridad del software se centran en las etapas finales del ciclo de vida. Entre más pronto se detecte un problema en el software, mejor. Y hay dos clases de problemas. Uno son los errores, que son problemas de implementación. El otro son las fallas del software: problemas de arquitectura en el diseño. La gente presta demasiada atención a los errores pero no la suficiente a las fallas.

Para construir un sistema seguro hay que centrarse en la calidad, y eso debe comenzar durante el diseño. Los conceptos y métodos analizados en la parte 2 del libro llevan a una arquitectura del software que reduce las "fallas". Al eliminar las fallas de arquitectura (con lo que mejora la calidad del software) será más difícil que intrusos penetren en el software.

14.3.6 El efecto de las acciones de la administración

Es frecuente que la calidad del software reciba influencia tanto de las decisiones administrativas como de las tecnológicas. Incluso las mejores prácticas de la ingeniería de software pueden ser arruinadas por malas decisiones gerenciales y por acciones cuestionables de la administración del proyecto.

En la parte 4 de este libro se analiza la administración del proyecto en el contexto del proceso del software. Al iniciar toda tarea del proyecto, el líder de éste tomará decisiones que tienen un efecto significativo en la calidad del producto.

Decisiones de estimación. Como se dice en el capítulo 26, un equipo de software rara vez puede darse el lujo de dar una estimación para el proyecto *antes* de que se hayan establecido las fechas de entrega y especificado un presupuesto general. En vez de ello, el equipo realiza un "filtro sanitario" para garantizar que las fechas de entrega y puntos de revisión son racionales. En muchos casos, hay una presión enorme del tiempo para entrar al mercado que fuerza al equipo a aceptar fechas de entrega irreales. En consecuencia, se toman atajos, se pasan por alto las actividades que elevan la calidad del software y disminuye la calidad del producto. Si una fecha de entrega es irracional, es importante poner los pies sobre la tierra. Explique por qué se necesita más tiempo o, alternativamente, sugiera un subconjunto de funciones que puedan entregarse (sin demasiada calidad) en el tiempo programado.

Decisiones de programación. Cuando se establece un programa de desarrollo de un proyecto de software (véase el capítulo 27), se establece la secuencia de las tareas con base en dependencias. Por ejemplo, como el componente **A** depende del procesamiento que ocurra

dentro de los componentes **B**, **C** y **D**, el componente **A** no puede programarse para ser probado hasta que los componentes **B**, **C** y **D** no hayan sido probados por completo. La programación del proyecto reflejaría esto. Pero si el tiempo es demasiado escaso y debe disponerse de **A** para realizar pruebas de importancia crítica, puede decidirse a probar **A** sin sus componentes subordinados (que están un poco retrasados) a fin de que esté disponible para otras pruebas que se realicen antes de la entrega. Después de todo, el plazo final se acerca. En consecuencia, **A** podría tener defectos ocultos que sólo se descubrirían mucho tiempo después. La calidad bajaría.

Decisiones orientadas al riesgo. La administración del riesgo (véase el capítulo 28) es uno de los atributos clave de un proyecto exitoso de software. En realidad se necesita saber lo que puede salir mal y establecer un plan de contingencia para ese caso. Demasiados equipos de software prefieren un optimismo ciego y establecen un programa de desarrollo con la suposición de que nada saldrá mal. Lo que es peor, no tienen manera de manejar las cosas que salgan mal. En consecuencia, cuando un riesgo se convierte en realidad, reina el caos y aumenta el grado de locuras que se cometen, con lo que invariablemente la calidad se desploma.

El dilema de la calidad del software se resume mejor con el enunciado de la Ley de Meskimen: *Nunca hay tiempo para hacerlo bien, pero siempre hay tiempo para hacerlo otra vez.* Mi consejo es: tomarse el tiempo para hacerlo bien casi nunca es la decisión equivocada.

14.4 Lograr la calidad del software

La calidad del software no sólo se ve. Es el resultado de la buena administración del proyecto y de una correcta práctica de la ingeniería de software. La administración y práctica se aplican en el contexto de cuatro actividades principales que ayudan al equipo de software a lograr una alta calidad en éste: métodos de la ingeniería de software, técnicas de administración de proyectos, acciones de control de calidad y aseguramiento de la calidad del software.

14.4.1 Métodos de la ingeniería de software

? ¿Qué necesito hacer para influir en la calidad de manera positiva?

Si espera construir software de alta calidad, debe entender el problema que se quiere resolver. También debe ser capaz de crear un diseño que esté de acuerdo con el problema y que al mismo tiempo tenga características que lleven al software a las dimensiones y factores de calidad que se estudiaron en la sección 14.2.

En la parte 2 de este libro se presentó una amplia variedad de conceptos y métodos que conducen a una comprensión razonablemente completa del problema y al diseño exhaustivo que establece un fundamento sólido para la actividad de construcción. Si el lector aplica estos conceptos y adopta métodos apropiados de análisis y diseño, se eleva sustancialmente la probabilidad de crear software de alta calidad.

14.4.2 Técnicas de administración de proyectos

El efecto de las malas decisiones de administración sobre la calidad del software se estudió en la sección 14.3.6. Las implicaciones son claras: si 1) un gerente de proyecto usa estimaciones para verificar que las fechas pueden cumplirse, 2) se comprenden las dependencias de las actividades programadas y el equipo resiste la tentación de usar atajos, 3) la planeación del riesgo se lleva a cabo de manera que los problemas no alienten el caos, entonces la calidad del software se verá influida de manera positiva.

Además, el plan del proyecto debe incluir técnicas explícitas para la administración de la calidad y el cambio. Las técnicas que llevan a buenas prácticas de administración de proyectos se estudian en la parte 4 de este libro.

14.4.3 Control de calidad

? ¿Qué es el control de calidad del software?

El control de calidad incluye un conjunto de acciones de ingeniería de software que ayudan a asegurar que todo producto del trabajo cumpla sus metas de calidad. Los modelos se revisan para garantizar que están completos y que son consistentes. El código se inspecciona con objeto de descubrir y corregir errores antes de que comiencen las pruebas. Se aplica una serie de etapas de prueba para detectar los errores en procesamiento lógico, manipulación de datos y comunicación con la interfaz. La combinación de mediciones con retroalimentación permite que el equipo del software sintonice el proceso cuando cualquiera de estos productos del trabajo falla en el cumplimiento de las metas de calidad. Las actividades de control de calidad se estudian en detalle en lo que resta de la parte 3 de este libro.

14.4.4 Aseguramiento de la calidad

WebRef

Pueden encontrarse vínculos útiles acerca de técnicas de aseguramiento de la calidad en la dirección **www.niwotridge.com/Resources/PM-SWEResources/SoftwareQualityAssurance.htm**

El aseguramiento de la calidad establece la infraestructura de apoyo a los métodos sólidos de la ingeniería de software, la administración racional de proyectos y las acciones de control de calidad, todo de importancia crucial si se trata de elaborar software de alta calidad. Además, el aseguramiento de la calidad consiste en un conjunto de funciones de auditoría y reportes para evaluar la eficacia y completitud de las acciones de control de calidad. La meta del aseguramiento de la calidad es proveer al equipo administrativo y técnico los datos necesarios para mantenerlo informado sobre la calidad del producto, con lo que obtiene perspectiva y confianza en que las acciones necesarias para lograr la calidad del producto funcionan. Por supuesto, si los datos provistos a través del aseguramiento de la calidad identifican los problemas, es responsabilidad de la administración enfrentarlos y aplicar los recursos necesarios para resolver los correspondientes a la calidad. En el capítulo 16 se estudia en detalle el aseguramiento de la calidad del software.

14.5 Resumen

La preocupación por la calidad de los sistemas basados en software ha aumentado a medida que éste se integra en cada aspecto de nuestras vidas cotidianas. Pero es difícil hacer la descripción exhaustiva de la calidad del software. En este capítulo se define la calidad como un proceso eficaz del software aplicado de modo que crea un producto útil que da un valor medible a quienes lo generan y a quienes lo utilizan.

Con el tiempo se han propuesto varias dimensiones y factores de calidad del software. Todos ellos tratan de definir un conjunto de características que, si se logran, llevarán a un software de alta calidad. McCall y los factores de calidad de la norma ISO 9126 establecen características tales como confiabilidad, usabilidad, facilidad de dar mantenimiento, funcionalidad y portabilidad, como indicadores de la existencia de calidad.

Toda organización de software se enfrenta al dilema de la calidad del software. En esencia, todos quieren elaborar sistemas de alta calidad, pero en un mundo dirigido por el mercado, sencillamente no se dispone del tiempo y el esfuerzo requeridos para producir software "perfecto". La cuestión es la siguiente: ¿debe elaborarse software que sea "suficientemente bueno"? Aunque muchas compañías hacen eso, hay una desventaja notable que debe tomarse en cuenta.

Sin importar el enfoque que se elija, la calidad tiene un costo que puede estudiarse en términos de prevención, evaluación y falla. Los costos de prevención incluyen todas las acciones de la ingeniería de software diseñadas para prevenir los defectos. Los costos de evaluación están asociados con aquellas acciones que evalúan los productos del trabajo de software para determinar su calidad. Los costos de falla incluyen el precio interno de fallar y los efectos externos que precipitan la mala calidad.

La calidad del software se consigue por medio de la aplicación de métodos de ingeniería de software, prácticas adecuadas de administración y un control de calidad exhaustivo, todo lo cual es apoyado por la infraestructura de aseguramiento de la calidad. En los capítulos que siguen se estudian con cierto detalle el control y aseguramiento de la calidad.

Problemas y puntos por evaluar

14.1. Describa cómo evaluaría la calidad de una universidad antes de inscribirse. ¿Cuáles factores serían importantes? ¿Cuáles tendrían importancia crítica?

14.2. Garvin [Gar84] describe cinco puntos de vista distintos sobre la calidad. Dé un ejemplo de cada uno con el uso de uno o más productos electrónicos conocidos con los que esté familiarizado.

14.3. Con el uso de la definición de calidad del software propuesta en la sección 14.2, diga si cree posible crear un producto útil que genere valor medible sin el uso de un proceso eficaz. Explique su respuesta.

14.4. Agregue dos preguntas adicionales a cada una de las dimensiones de la calidad de Garvin presentadas en la sección 14.2.1.

14.5. Los factores de calidad de McCall se desarrollaron en la década de 1970. Casi todos los aspectos de la computación han cambiado mucho desde entonces, no obstante lo cual aún se aplican al software moderno. ¿Qué conclusiones saca con base en ello?

14.6. Con el empleo de los subatributos mencionados en la sección 14.2.3 para el factor de calidad llamado "facilidad de recibir mantenimiento", de la ISO 9126, desarrolle preguntas que exploren si estos atributos existen o no. Continúe el ejemplo presentado en la sección 14.2.4.

14.7. Describa con sus propias palabras el dilema de la calidad del software.

14.8. ¿Qué es un software "suficientemente bueno"? Mencione una compañía dada y productos específicos que crea que fueron desarrollados con el uso de la filosofía de lo suficientemente bueno.

14.9. Considere cada uno de los cuatro aspectos de la calidad y diga cuál piensa que es el más caro y por qué.

14.10. Haga una búsqueda en web y encuentre otros tres ejemplos de "riesgos" para el público que puedan atribuirse directamente a la mala calidad de un software. Comience la búsqueda en **http://catless.ncl.ac.uk/risks.**

14.11. ¿Son lo mismo *calidad* y *seguridad*? Explique su respuesta.

14.12. Explique por qué es que muchos de nosotros utilizamos la ley de Meskimen. ¿Qué ocurre con el software de negocios que causa esto?

Lecturas y fuentes de información adicionales

Los conceptos básicos de calidad del software se estudian en los libros de Henry y Hanlon (*Software Quality Assurance*, Prentice-Hall, 2008), Kahn *et al.* (*Software Quality: Concepts and Practice*, Alpha Science International, Ltd., 2006), O'Regan (*A Practical Approach to Software Quality*, Springer, 2002) y Daughtrey (*Fundamental Concepts for the Software Quality Engineer*, ASQ Quality Press, 2001).

Duvall *et al.* (*Continuous Integration: Improving Software Quality and Reducing Risk*, Addison-Wesley, 2007), Tian (*Software Quality Engineering*, Wiley-IEEE Computer Society Press, 2005), Kandt (*Software Engineering Quality Practices*, Auerbach, 2005), Godbole (*Software Quality Assurance: Principles and Practice*, Alpha Science International, Ltd., 2004) y Galin (*Software Quality Assurance: From Theory to Implementation*, Addison-Wesley, 2003) presentan estudios detallados del aseguramiento de la calidad del software. Stamelos y Sfetsos (*Agile Software Development Quality Assurance*, IGI Global, 2007) estudian el aseguramiento de la calidad en el contexto del proceso ágil.

El diseño sólido conduce a una alta calidad del software. Jayasawal y Patton (*Design for Trustworthy Software*, Prentice-Hall, 2006) y Ploesch (*Contracts, Scenarios and Prototypes*, Springer, 2004) analizan las herramientas y técnicas para desarrollar software "robusto".

La medición es un componente importante de la ingeniería de calidad del software. Ejiogu (*Software Metrics: The Discipline of Software Quality*, BookSurge Publishing, 2005), Kan (*Metrics and Models in Software*

Quality Engineering, Addison-Wesley, 2002) y Nance y Arthur (*Managing Software Quality*, Springer, 2002) estudian unidades de medida y modelos importantes relacionados con la calidad. Los aspectos de la calidad del software orientados al equipo los estudia Evans (*Achieving Software Quality through Teamwork*, Artech House Publishers, 2004).

En internet existe una amplia variedad de fuentes de información acerca de la calidad del software. En el sitio web del libro, en la dirección: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, se encuentra una lista actualizada de referencias existentes en la red mundial y que son relevantes para la calidad del software.

CAPÍTULO

15 TÉCNICAS DE REVISIÓN

CONCEPTOS CLAVE

amplificación del defecto . . . 356
defectos 355
densidad del error 358
errores 355
registro 363
métricas
  revisión 357
  reporte 363
revisiones
  eficacia del costo de las . . . 358
  informales 361
  orientadas al muestreo . . . 365
  técnicas 362

Las revisiones son como filtros en el flujo del trabajo del proceso de software. Si son muy pocas, el flujo queda "sucio". Si son demasiadas, se hace lento hasta detenerse. Utilice métricas para determinar cuáles son las revisiones que funcionan y haga énfasis en ellas. Elimine del flujo las revisiones ineficaces, con objeto de acelerar el proceso.

Las revisiones del software son un "filtro" para el proceso del software. Es decir, se aplican en varios puntos durante la ingeniería de software y sirven para descubrir errores y defectos a fin de poder eliminarlos. Las revisiones del software "purifican" los productos del trabajo de la ingeniería de software, incluso los modelos de requerimientos y diseño, código y datos de prueba. Freedman y Weinberg [Fre90] analizan del modo siguiente la necesidad de hacer revisiones:

> El trabajo técnico necesita las revisiones por la misma razón que los lápices necesitan borradores: *errar es humano*. La segunda razón por la que son necesarias las revisiones técnicas es porque, si bien las personas son buenas para detectar algunos de sus propios errores, muchas clases de ellos pasan desapercibidos con más facilidad para quien los comete que para otras personas. Por tanto, este proceso de revisión es la respuesta a la oración de Robert Burns:
>
> *Oh, quiera algún Dios el regalo darnos*
>
> *de vernos a nosotros como los demás nos ven*
>
> Una revisión —cualquiera — es una forma de utilizar la diversidad de un grupo para lo siguiente:
>
> 1. Resaltar las mejoras necesarias en el producto que elaboró una sola persona o equipo;
> 2. Confirme aquellas partes de un producto en las que no se desea o no se necesita hacer una mejora;
> 3. Realice el trabajo técnico de calidad más uniforme, o al menos más predecible, que pueda lograrse sin hacer revisiones, a fin de que el trabajo técnico sea más manejable.

Como parte de la ingeniería de software, pueden realizarse muchos diferentes tipos de revisiones. Cada uno tiene su lugar. Una reunión informal alrededor de la máquina del café es una forma de revisión si se analizan problemas técnicos. La presentación formal de la arquitectura del software a un público de clientes, administradores y técnicos también es una forma de revi-

UNA MIRADA RÁPIDA

¿Qué es? Conforme se desarrollen los productos del trabajo de la ingeniería de software se cometerán errores. No es vergonzoso, mientras se trate de detectarlos y corregirlos con ahínco —con mucho ahínco— antes de que lleguen a los usuarios finales. Las revisiones técnicas son el mecanismo más eficaz para detectar los errores en una etapa temprana del proceso de software.

¿Quién lo hace? Son los ingenieros de software quienes realizan una revisión técnica, también llamada revisión de pares, con sus colegas.

¿Por qué es importante? Si encuentra un error al principio del proceso, es menos caro corregirlo. Además, los errores tienen un modo de amplificarse a medida que avanza el proceso. Por ello, un error relativamente pequeño que se deje sin atender al comenzar el proceso se amplifica en un conjunto más grande de errores en una etapa posterior del proyecto. Finalmente, las revisiones ahorran tiempo, reduciendo la cantidad de repeticiones que se requerirán hacia el final del proyecto.

¿Cuáles son los pasos? El enfoque de las revisiones variará en función del grado de formalidad que se elija. En general, se utilizan seis etapas, aunque no todas se emplean siempre: planeación, preparación, estructurar la reunión, resaltar los errores, hacer las correcciones (fuera de la revisión) y verificar que las correcciones se hayan hecho en forma apropiada.

¿Cuál es el producto final? El resultado de una revisión es una lista de conceptos o errores descubiertos. Además, también se indica el estado técnico del producto final.

¿Cómo me aseguro de que lo hice bien? En primer lugar, seleccione el tipo de revisión que sea apropiada para su cultura de desarrollo. Siga los lineamientos que lleven a ejecutar revisiones exitosas. Si éstas conducen a un software de alta calidad, lo habrá hecho bien.

sión. Sin embargo, en este libro nos centramos en las *revisiones técnicas o por pares*, ejemplificadas por las *revisiones casuales,* walkthroughs e *inspecciones*. Desde el punto de vista del control de calidad, una revisión técnica (RT) es el filtro más eficaz. Realizado por ingenieros de software (y de otro tipo) para ingenieros de software, la RT es un medio eficaz para detectar errores y mejorar la calidad.

15.1 Efecto de los defectos del software en el costo

En el contexto del proceso del software, los términos *defecto* y *falla* son sinónimos. Los dos implican un problema de calidad descubierto *después* de haberse liberado el software a los usuarios finales (o a otra actividad estructural del proceso del software). En capítulos anteriores se empleó el término *error* para denotar un problema de calidad descubierto por ingenieros de software (o de otra clase) *antes* de entregar el software al usuario final (o a alguna actividad estructural del proceso del software).

Información

Equivocaciones, errores y defectos

La meta del control de calidad del software, y en un sentido más amplio de la administración de la calidad en general, es eliminar los problemas de calidad que se encuentren en el software. Se hace referencia a estos problemas con diferentes nombres: *equivocaciones*, *fallas*, *errores* o *defectos*, por mencionar algunos. ¿Son sinónimos estos términos o hay diferencias sutiles entre ellos?

En este libro se hace una distinción clara entre un error (problema de calidad que se detecta antes de que el software se entregue a los usuarios finales) y un defecto (problema de calidad que se encuentra después de haber entregado el software a los usuarios finales[1]). Esta distinción se hace porque los errores y defectos tienen muy distinto efecto económico, empresarial, sicológico y humano. Como ingenieros de software, queremos encontrar y corregir tantos errores como sea posible antes de que el consumidor o el usuario final los encuentren. Queremos evitar los defectos porque hacen (justificadamente) que el personal de software se vea mal.

Sin embargo, es importante observar que la distinción temporal entre errores y defectos que se hace en este libro no constituye la principal forma de pensar. El consenso general de la comunidad de ingeniería de software es que defectos, errores, fallas y equivocaciones son sinónimos. Es decir, el punto en el tiempo en el que se encontró el problema no tiene que ver con el término que se usa para describirlo. Parte de la argumentación a favor de este punto de vista es que en ocasiones es difícil hacer una distinción clara entre el antes y el después de la liberación (por ejemplo, considere un proceso incremental en un desarrollo ágil).

Sin que importe el modo en el que se elija interpretar estos términos, hay que reconocer que el momento en el que se descubre un problema sí importa, y que los ingenieros de software deben tratar de detectar con ahínco —con mucho ahínco— los problemas antes de que sus clientes y usuarios finales los encuentren. Si el lector está más interesado en este tema, puede hallar un análisis razonablemente completo de la terminología acerca de las "equivocaciones" en la dirección **www.softwaredevelopment.ca/bugs.shtml**

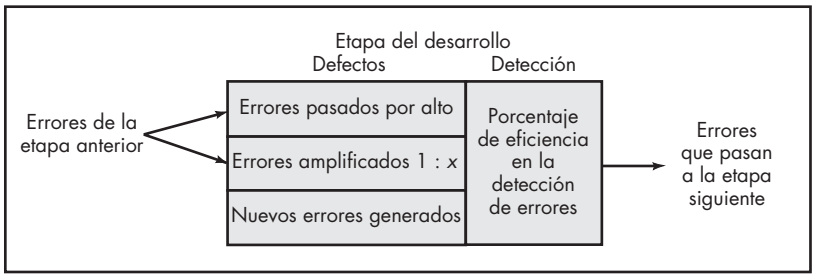
El objetivo principal de una revisión técnica formal es detectar los errores antes de que pasen a otra actividad de la ingeniería de software o de que se entreguen al usuario final.

El objetivo principal de las revisiones técnicas es encontrar errores durante el proceso a fin de que no se conviertan en defectos después de liberar el software. El beneficio obvio de las revisiones técnicas es el descubrimiento temprano de los errores, de modo que no se propaguen a la siguiente etapa del proceso del software.

Varios estudios de la industria indican que las actividades de diseño introducen de 50 a 65 por ciento de todos los errores (y en realidad de todos los defectos) durante el proceso del software. Sin embargo, las técnicas de revisión han demostrado tener una eficacia de hasta 75 por ciento [Jon86] para descubrir fallas del diseño. Al detectar y eliminar un gran porcentaje de estos

1 Si se considera una mejora en el proceso del software, un problema de calidad que se propague de una actividad estructural del proceso (como el **modelado**) a otra (como la **construcción**) también se llama "defecto", porque debe encontrarse el problema antes de que un producto del trabajo (como un modelo del diseño) se "libere" a la siguiente actividad.

FIGURA 15.1
Modelo de amplificación del defecto



errores, el proceso de revisión reduce de manera sustancial el costo de las actividades posteriores en el proceso del software.

15.2 AMPLIFICACIÓN Y ELIMINACIÓN DEL DEFECTO

Cita:

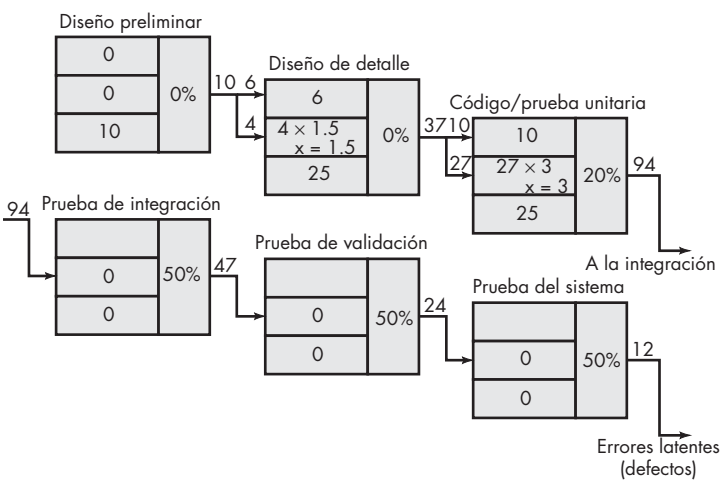
"Dicen los médicos que en sus inicios algunas enfermedades son fáciles de curar pero difíciles de reconocer… mas con el paso del tiempo, si no se detectaron y trataron al principio, se vuelven fáciles de reconocer pero difíciles de curar."

Nicolás Maquiavelo

Para ilustrar la generación y detección de errores durante las acciones de diseño y generación de código de un proceso de software, puede usarse un *modelo de amplificación del defecto* [IBM81]. En la figura 15.1 se ilustra esquemáticamente el modelo. Un cuadro representa una acción de la ingeniería de software. Durante la acción, los errores se generan de manera inadvertida. La revisión puede fracasar en descubrir los errores nuevos que se generan y los cometidos en etapas anteriores, lo que da como resultado cierto número de errores pasados por alto. En ciertos casos, los errores de etapas anteriores ignorados son amplificados (en un factor *x* de amplificación) por el trabajo en curso. Las subdivisiones de los cuadros representan a cada una de estas características y al porcentaje de eficiencia de la detección de errores, que es una función de la profundidad de la revisión.

La figura 15.2 ilustra un ejemplo hipotético de amplificación del defecto para un proceso de software en el que no se hacen revisiones. En la figura, se supone que en cada etapa de prueba se detecta y corrige 50 por ciento de todos los errores de entrada sin que se introduzcan nuevos errores (suposición optimista). Diez defectos preliminares de diseño se amplifican a 94 errores antes de que comiencen las pruebas. Se liberan al campo 12 errores latentes (defectos). La figura 15.3 considera las mismas condiciones, excepto porque se efectúan revisiones del diseño y código como parte de cada acción de la ingeniería de software. En este caso, son 10 los errores

FIGURA 15.2
Amplificación del defecto. Sin revisiones



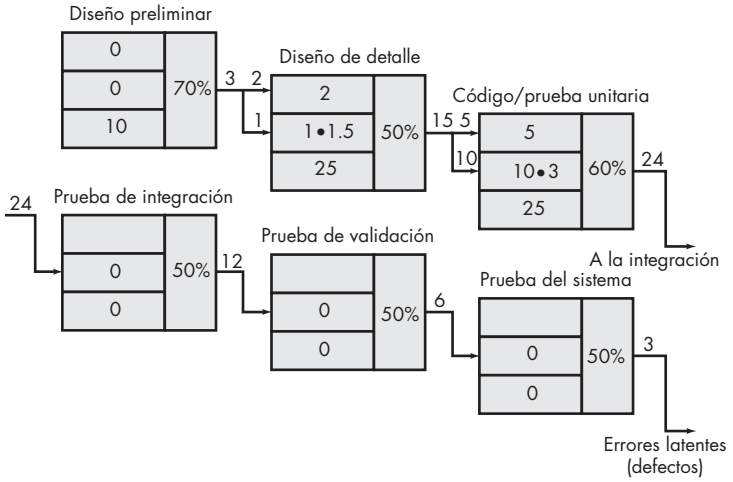


FIGURA 15.3
Amplificación del defecto. Se efectúan revisiones

iniciales de diseño preliminar (arquitectura) que se amplifican a 24 antes de comenzar las pruebas. Sólo existen tres errores latentes. Pueden establecerse los costos relativos asociados con el descubrimiento y corrección de errores, así como el costo general (con y sin revisión para nuestro ejemplo hipotético). El número de errores detectados durante cada una de las etapas citadas en las figuras 15.2 y 15.3 se multiplica por el costo que implica eliminar un error (1.5 unidades de costo para el diseño, 6.5 unidades de costo antes de las pruebas, 15 unidades de costo durante las pruebas y 67 unidades de costo después de la entrega).[2] Con estos datos, el costo total del desarrollo y mantenimiento cuando se efectúan revisiones es de 783 unidades de costo. Cuando no se hacen revisiones, el costo total es de 2 177 unidades, casi tres veces más caro.

Debe dedicarse tiempo y esfuerzo a la realización de revisiones y su organización de desarrollo debe destinar el dinero para ello. Sin embargo, los resultados del ejemplo anterior dejan pocas dudas acerca de lo que puede pagar ahora o de que después deberá pagar mucho más.

15.3 MÉTRICAS DE REVISIÓN Y SU EMPLEO

Las revisiones técnicas son una de las muchas acciones que se requieren como parte de las buenas prácticas de la ingeniería de software. Cada acción requiere un esfuerzo humano dirigido. Como el esfuerzo disponible para el proyecto es finito, es importante que una organización de software comprenda la eficacia de cada acción, definiendo un conjunto de métricas (véase el capítulo 23) que puedan utilizarse para evaluar esa eficacia.

Aunque se han definido muchas métricas para las revisiones técnicas, un conjunto relativamente pequeño da una perspectiva útil. Las siguientes métricas para la revisión pueden obtenerse conforme se efectúe ésta:

- *Esfuerzo de preparación, E_p*: esfuerzo (en horas-hombre) requerido para revisar un producto del trabajo antes de la reunión de revisión real.
- *Esfuerzo de evaluación, E_a*: esfuerzo requerido (en horas-hombre) que se dedica a la revisión real.
- *Esfuerzo de la repetición, E_r*: esfuerzo (en horas-hombre) que se dedica a la corrección de los errores descubiertos durante la revisión.

2 Estos multiplicadores son algo diferentes de los datos presentados en la figura 14.2, que es más actual. Sin embargo, sirven para ilustrar los costos de la amplificación del defecto.

- *Tamaño del producto del trabajo, TPT*: medición del tamaño del producto del trabajo que se ha revisado (por ejemplo, número de modelos UML o número de páginas de documento o de líneas de código).
- *Errores menores detectados,* $Err_{menores}$: número de errores detectados que pueden clasificarse como menores (requieren menos de algún esfuerzo especificado para corregirse).
- *Errores mayores detectados,* $Err_{mayores}$: número de errores encontrados que pueden clasificarse como mayores (requieren más que algún esfuerzo especificado para corregirse).

Estas métricas pueden mejorarse, asociando el tipo de producto del trabajo que se revisó con las métricas obtenidas.

15.3.1 Análisis de las métricas

Antes de comenzar el análisis deben hacerse algunos cálculos sencillos. El esfuerzo total de revisión y el número total de errores descubiertos se definen como sigue:

$$E_{\text{revisión}} = E_p + E_a + E_r$$

$$\text{Err}_{\text{tot}} = \text{Err}_{\text{menores}} + \text{Err}_{\text{mayores}}$$

La *densidad del error* representa los errores encontrados por unidad de producto del trabajo revisada.

$$\text{Densidad del error} = \frac{\text{Err}_{\text{tot}}}{\text{TPT}}$$

Por ejemplo, si se revisa un modelo de requerimientos con objeto de encontrar errores, inconsistencias y omisiones, es posible calcular la densidad del error en varias formas diferentes. El modelo de requerimientos contiene 18 diagramas UML como parte de 32 páginas de materiales descriptivos. La revisión detecta 18 errores menores y 4 mayores. Por tanto, $\text{Err}_{\text{tot}} = 22$. La densidad del error es 1.2 errores por diagrama UML o 0.68 errores por página del modelo de requerimientos.

Si las revisiones se llevan a cabo para varios tipos distintos de productos del trabajo (por ejemplo, modelo de requerimientos, modelo del diseño, código, casos de prueba, etc.), el porcentaje de errores no descubiertos por cada revisión se confronta con el número total de errores detectados en todas las revisiones. Además, puede calcularse la densidad del error para cada producto del trabajo.

Una vez recabados los datos para muchas revisiones efectuadas en muchos proyectos, los valores promedio de la densidad del error permiten estimar el número de errores por hallar en un nuevo documento (aún no revisado). Por ejemplo, si la densidad promedio de error para un modelo de requerimientos es de 0.6 errores por página, y un nuevo modelo de requerimientos tiene una longitud de 32 páginas, una estimación gruesa sugiere que el equipo de software encontrará alrededor de 19 o 20 errores durante la revisión del documento. Si sólo encuentra 6 errores, habrá hecho un trabajo extremadamente bueno al desarrollar el modelo de requerimientos o su enfoque de la revisión no fue tan profundo.

Una vez llevada a cabo la prueba (véanse los capítulos 17 a 20), es posible obtener datos adicionales del error, incluso el esfuerzo requerido para detectar y corregir errores no descubiertos durante las pruebas y la densidad del error del software. Los costos asociados con la detección y corrección de un error durante las pruebas pueden compararse con los de las revisiones. Esto se analiza en la sección 15.3.2.

15.3.2 Eficacia del costo de las revisiones

Es difícil medir en tiempo real la eficacia del costo de cualquier revisión técnica. Una organización de ingeniería de software puede evaluar la eficacia de las revisiones y su relación costo-

beneficio sólo después de que éstas han terminado, de que las unidades de medida de la revisión se han recabado, de que los datos promedio han sido calculados y de que la calidad posterior del software ha sido medida (mediante pruebas).

Si regresamos al ejemplo presentado en la sección 15.3.1, se determinó que la densidad promedio del error para los modelos de requerimientos era de 0.6 errores por página. Se reveló que el esfuerzo requerido para corregir un error menor en el modelo era de 4 horas-hombre. Se vio que el esfuerzo necesario para un error mayor en los requerimientos era de 18 horas-hombre. Al estudiar los datos recabados se observa que los errores menores ocurrieron con una frecuencia cercana a 6 veces más que los errores mayores. Por tanto, puede estimarse que el esfuerzo promedio para detectar y corregir un error en los requerimientos durante la revisión es alrededor de 6 horas-hombre.

Los errores relacionados con los requerimientos no detectados durante las pruebas requieren un promedio de 45 horas-hombre para encontrarse y corregirse (no hay datos disponibles acerca de la severidad relativa del error). Con estos promedios se obtiene lo siguiente:

$$\text{Esfuerzo ahorrado por error} = E_{\text{pruebas}} - E_{\text{revisiones}}$$
$$45 - 6 = 30 \text{ horas-hombre/error}$$

Como durante la revisión del modelo de requerimientos se encontraron 22 errores, se tendrá un ahorro cercano a 660 horas-hombre en el esfuerzo dedicado a las pruebas. Y esto se refiere sólo a los errores relacionados con los requerimientos. Al beneficio general se suman aquellos asociados con el diseño y el código. El esfuerzo total conduce a ciclos de entrega más cortos y a un mejor tiempo para llegar al mercado.

En su libro sobre la revisión por pares, Karl Wiegers [Wie02] analiza datos procedentes de anécdotas de compañías grandes que han utilizado *inspecciones* (un tipo relativamente formal de revisión técnica) como parte de sus actividades de control de calidad del software. Hewlett Packard reportó un rendimiento de 10 a 1 sobre la inversión gracias a las inspecciones y afirmó que la entrega real del producto se aceleró en un promedio de 1.8 meses-calendario. AT&T indicaba que las inspecciones habían reducido el costo general de los errores de software en un factor de 10, que la calidad había mejorado en un orden de magnitud y que la productividad se había incrementado 14 por ciento. Otras empresas reportaban beneficios similares. Las revisiones técnicas (en diseño y otras actividades) generan una buena relación costo-beneficio y en verdad ahorran tiempo.

Pero para muchos profesionales del software, esta afirmación va contra la intuición. "Las revisiones toman tiempo", dicen, "y no tenemos tiempo que perder…". Afirman que el tiempo es precioso en cada proyecto de software y que la actividad de revisar "todo producto del trabajo con detalle" absorbe demasiado.

Los ejemplos presentados en esta sección indican otra cosa. Lo más importante es que los datos de la industria sobre revisiones del software se han recabado durante más de dos décadas y se resumen cualitativamente en las gráficas que aparecen en la figura 15.4

En la figura, el trabajo efectuado cuando se utilizan revisiones se refleja pronto en el desarrollo de un incremento de software, pero esta inversión temprana paga dividendos debido a que se reduce el esfuerzo necesario para hacer pruebas y correcciones. De igual importancia es que la fecha de entrega del desarrollo con revisiones ocurre antes que la que se hace sin revisiones. ¡Las revisiones no quitan tiempo, lo ahorran!

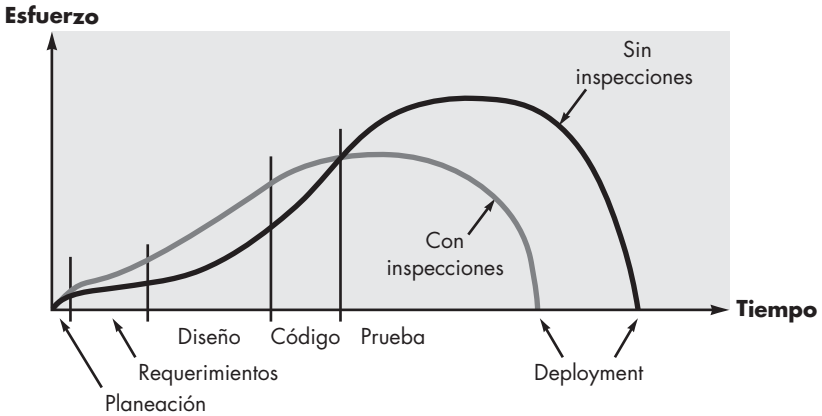
15.4 Revisiones: espectro de formalidad

Las revisiones técnicas deben aplicarse con un nivel de formalidad apropiado para el producto que se va a elaborar, para el plazo que tiene el proyecto y para el personal que realice el trabajo.

FIGURA 15.4

Esfuerzo realizado, con y sin revisiones

Fuente: adaptado de [Fog86].



La figura 15.5 ilustra un modelo de referencia para las revisiones técnicas [Lai02] que identifica cuatro características que contribuyen a la formalidad con la que se efectúa una revisión.

Cada una de las características del modelo de referencia ayuda a definir el nivel de formalidad de la revisión. La formalidad de una revisión se incrementa cuando: 1) se definen explícitamente roles distintos para los revisores, 2) hay suficiente cantidad de planeación y preparación para la revisión, 3) se define una estructura distinta para la revisión (incluso tareas y productos internos del trabajo) y 4) el seguimiento por parte de los revisores tiene lugar para cualesquiera correcciones que se efectúen.

Para entender el modelo de referencia, supongamos que el lector decidió revisar el diseño de la interfaz para **CasaSeguraAsegurada.com**. Esto puede hacerse de varias maneras diferentes, que van de lo relativamente casual a lo riguroso en extremo. Si decide que el enfoque casual es más apropiado, se pide a algunos colegas (pares) que examinen el prototipo de la interfaz en un esfuerzo por descubrir problemas potenciales. Todos deciden que no habrá preparación previa, pero que evaluarán el prototipo en una forma razonablemente estructurada: primero verán la distribución, luego la estética, después las opciones de navegación, etc. Como diseñador que es, el lector decide tomar algunas notas, pero nada formales.

Pero, ¿qué pasa si la interfaz es crucial para el éxito de todo el proyecto? ¿Qué sucede si de lo acertado de su ergonomía dependen vidas humanas? Debió concluirse que era necesario un enfoque más riguroso. Se forma entonces el equipo de revisión. Cada integrante de éste tendrá

FIGURA 15.5

Modelo de referencia para hacer revisiones técnicas

un rol específico: dirigir el equipo, registrar las reuniones, presentar el material, etc. Cada revisor tendrá acceso al producto del trabajo (en este caso, el prototipo de la interfaz) antes de que la revisión tenga lugar y dedicará tiempo a la búsqueda de errores, inconsistencias y omisiones. Se realizará un conjunto de tareas específicas con base en una agenda que se desarrollará antes de que ocurra la revisión. Los resultados de ésta serán registrados de manera formal y el equipo decidirá sobre el estado del producto del trabajo con base en el resultado de la revisión. Los miembros del equipo también verificarán que las correcciones se hagan de manera adecuada.

En este libro se consideran dos grandes categorías de revisiones técnicas: revisiones informales y revisiones técnicas más formales. Dentro de cada una de ellas se escogen varios enfoques diferentes. Éstos se presentan en las secciones que siguen.

15.5 Revisiones informales

Las revisiones informales incluyen una simple verificación de escritorio de un trabajo de ingeniería de software, hecha con algún colega, o una reunión casual (con más de dos personas) con objeto de revisar un producto o aspectos orientados a la revisión de programación por pares (véase el capítulo 3).

Una *verificación de escritorio* simple o una *reunión casual* realizada con un colega constituye una revisión. Sin embargo, como no hay una planeación o preparación por adelantado, ni agenda o estructura de la reunión, y no se da seguimiento a los errores descubiertos, la eficacia de tales revisiones es mucho menor que la de los enfoques más formales. Pero una verificación de escritorio sencilla descubre errores que de otro modo se propagarían en el proceso del software.

Una forma de mejorar la eficacia de una verificación de escritorio es desarrollar un conjunto de listas de revisión para cada producto grande del trabajo generado por el equipo de software. Las preguntas que se plantean en la lista son generales, pero servirán para guiar a los revisores en la verificación del producto. Por ejemplo, veamos una verificación de escritorio del prototipo de la interfaz de **CasaSeguraAsegurada.com**. En vez de sólo jugar con el prototipo en la estación de trabajo del diseñador, éste y un colega lo examinan con el empleo de una lista para interfaces:

- ¿La distribución está diseñada con el empleo de convenciones estándar? ¿De izquierda a derecha? ¿De arriba abajo?
- ¿La presentación necesita ser desplazada verticalmente?
- ¿Se usan con eficacia el color y la ubicación, la tipografía y el tamaño?
- ¿Todas las opciones o funciones de navegación están representadas en el mismo nivel de abstracción?
- ¿Están etiquetadas con claridad todas las elecciones de navegación?

y así sucesivamente. Cualesquiera errores o aspectos señalados por los revisores son registrados por el diseñador para resolverlos tiempo después. Las verificaciones de escritorio se programan en forma *ad hoc* o son obligatorias como parte de las buenas prácticas de la ingeniería de software. En general, la cantidad de material por revisar es relativamente pequeña y el tiempo total dedicado a una revisión de escritorio es de poco más de una hora o dos.

En el capítulo 3 se describió la *programación por pares* en la forma siguiente: "La XP recomienda que dos personas trabajen juntas en una estación de trabajo con objeto de crear el código de una narración. Esto proporciona un mecanismo para resolver problemas y asegurar la calidad en tiempo real (dos cabezas piensan más que una)."

La programación por pares se caracteriza por una verificación de escritorio continua. En vez de programar una revisión en algún momento dado, la programación por pares invita a hacer

una revisión continua a medida que se crea el producto (diseño o código). El beneficio es el inmediato descubrimiento de los errores y, en consecuencia, la mejora de la calidad del producto.

En su estudio sobre la eficacia de la programación por pares, Williams y Kessler [Wil00] afirman lo siguiente:

> Las evidencias anecdóticas e iniciales señalan que la programación por pares es una técnica poderosa para generar productivamente trabajos de software de alta calidad. Los elementos de la pareja laboran y comparten sus ideas para resolver las complejidades del desarrollo del software. Realizan de manera continua inspecciones de lo que hace cada quien, lo que conduce a una forma de eliminación de defectos más rápida y eficiente. Además, se mantienen centrados intensamente en la tarea uno del otro.

Algunos ingenieros de software dicen que la redundancia inherente construida en la programación por parejas es un desperdicio de recursos. Después de todo, ¿por qué asignar dos personas a un trabajo que podría ejecutar sólo una? La respuesta a esta pregunta se encuentra en la sección 15.3.2. Si la calidad del producto del trabajo generado como consecuencia de la programación en parejas es mucho mejor que el trabajo de un individuo, los ahorros relacionados con la calidad justifican de sobra la "redundancia" implícita en la programación por parejas.

INFORMACIÓN

Listas de verificación para revisión

Aun cuando las revisiones estén bien organizadas y se lleven a cabo de manera apropiada, no es mala idea dar a los revisores una "criba". Es decir, es útil tener una lista de verificación que dé a cada revisor las preguntas que debe plantear acerca del producto específico del trabajo que se revisa.

Una de las listas más completas es la desarrollada por la NASA en el Centro Goddard de Vuelos Espaciales, disponible en la dirección **http://sw-assurance.gsfc.nasa.gov/disciplines/quality/index.php**

Hay otras listas útiles de revisión técnica que han sido propuestas por las siguientes entidades:

Process Impact (**www.processimpact.com/pr_goodies.shtml**)

Software Dioxide (**www.softwaredioxide.com/Channels/ConView.asp?id=6309**)

Macadamian (**www.macadamian.com**)

The Open Group Architecture Review Checklist (**www.opengroup.org/architecture/togaf7-doc/arch/p4/comp/clists/syseng.htm**)

DFAS (puede descargarse, **www.dfas.mil/technology/pal/ssps/docstds/spm036.doc**)

15.6 REVISIONES TÉCNICAS FORMALES

Cita:

"No hay nada más urgente para alguien que corregir el trabajo de los demás."

Mark Twain

Una *revisión técnica formal* (RTF) es una actividad del control de calidad del software realizada por ingenieros de software (y otras personas). Los objetivos de una RTF son: 1) descubrir los errores en funcionamiento, lógica o implementación de cualquier representación del software; 2) verificar que el software que se revisa cumple sus requerimientos; 3) garantizar que el software está representado de acuerdo con estándares predefinidos; 4) obtener software desarrollado de manera uniforme y 5) hacer proyectos más manejables. Además, la RTF sirve como método de capacitación, pues permite que los ingenieros principiantes observen distintos enfoques de análisis, diseño e implementación del software. La RTF también funciona para estimular el respaldo y la continuidad debido a que varias personas se familiarizan con software que de otra manera no hubieran visto.

La RTF en realidad es una clase que incluye *walkthroughs* e *inspecciones*. Cada RTF se realiza como una reunión y tendrá éxito sólo si se planea, controla y ejecuta en forma apropiada. En las secciones que siguen se presentan lineamientos similares a aquellos usados para un walkthrough, como representativos de la revisión técnica formal. Si el lector tiene interés en las ins-

pecciones de software y en obtener más información sobre walkthroughs, consulte [Rad02], [Wie02] o [Fre90].

15.6.1 La reunión de revisión

WebRef

El documento *Formal Inspection Guidebook*, de la NASA, puede descargarse del sitio **satc.gsfc.nasa.gov/Documents/fi/gdb/fi.pdf**

Sin importar cuál formato de RTF se elija, cualquiera de ellos debe cumplir las restricciones siguientes:

- En la revisión deben involucrarse de tres a cinco personas (normalmente).
- Debe haber preparación previa, pero no debe exigir más de dos horas de trabajo de cada persona.
- La duración de la reunión de revisión debe ser de al menos dos horas.

Dadas estas restricciones, debe resultar obvio que una RTF se centra en una parte específica (y pequeña) del software general. Por ejemplo, en vez de tratar de revisar todo el diseño, se hacen walkthrougs para cada componente o grupo pequeño de componentes. Al reducir el alcance, la RTF tiene mayor probabilidad de detectar errores.

Una RTF se centra en una parte relativamente pequeña de un producto del trabajo.

La atención de la RTF se dirige a un producto (por ejemplo, una parte del modelo de requerimientos, el diseño detallado de un componente o su código fuente, etc.). El individuo que haya desarrollado el producto —el *productor*— informa al líder del proyecto que ha terminado y que se requiere hacer una revisión. El líder del proyecto contacta al *líder de la revisión*, quien evalúa el producto en cuanto a su conclusión, genera copias de los materiales del producto y las distribuye a dos o tres *revisores* para la preparación previa. Se espera que cada revisor dedique de una a dos horas a la inspección del producto, tome notas y se familiarice con el trabajo. Al mismo tiempo, el líder del proyecto también revisa el producto y establece una agenda para la reunión de revisión, que por lo general se programa para el día siguiente.

En ciertas situaciones, es buena idea que alguien distinto del productor haga el walkthroug del producto que se revisa. Esto lleva a una interpretación literal del producto y a mejorar el reconocimiento de errores.

A la reunión de revisión acuden el líder de ésta, todos los revisores y el productor. Uno de los revisores adopta el rol de *secretario*, es decir, quien registra (por escrito) todos los acontecimientos importantes que surjan durante la revisión. La RTF comienza con el análisis de la agenda y una introducción breve por parte del productor. Después, éste procede a "recorrer" el producto del trabajo, explicando el material, mientras los revisores hacen sus comentarios con base en la preparación que hicieron. Cuando se descubren problemas o errores válidos, el secretario toma nota de ellos.

Al terminar la revisión, todos los asistentes deben decidir si: 1) aceptan el producto sin modificaciones, 2) lo rechazan debido a errores graves (una vez corregidos, se realiza otra revisión) o 3) aceptan el producto de manera provisional (se encontraron errores menores que deben corregirse, pero no se necesita otra revisión). Una vez tomada la decisión, todos los asistentes a la RTF firman el acta que indica su participación y su acuerdo con los descubrimientos del equipo de revisión.

15.6.2 Reporte y registro de la revisión

Durante la RTF, un revisor (el secretario) registra activamente todos los asuntos que se planteen. Éstos se resumen al final de la reunión y se produce la *lista de pendientes de la revisión*. Además se elabora un *reporte técnico formal de la revisión*. Éste responde tres preguntas:

1. ¿Qué fue lo que se revisó?
2. ¿Quién lo revisó?
3. ¿Cuáles fueron los descubrimientos y las conclusiones?

El resumen del reporte de la revisión es una sola página (quizá con anexos) que se vuelve parte del registro histórico del proyecto y se entrega al líder del proyecto y a otras partes interesadas.

La lista de pendientes de la revisión tiene dos propósitos: 1) identificar áreas de problemas en el producto y 2) servir como lista de verificación de acciones que guíe al productor cuando se hagan las correcciones. La lista de pendientes normalmente se anexa al reporte técnico.

Debe establecerse un procedimiento de seguimiento para garantizar que los pendientes de la lista se corrijan de manera apropiada. A menos que esto se haga, es posible que los pendientes anotados "se pierdan en el camino". Un enfoque consiste en asignar la responsabilidad del seguimiento al líder del proyecto.

15.6.3 Lineamientos para la revisión

No señale los errores en forma grosera. Una manera amable de hacerlo es plantear preguntas que lleven al productor a descubrir el error.

Los lineamientos para efectuar revisiones técnicas formales deben establecerse por adelantado, distribuirse a todos los revisores, llegar al consenso y, después, seguirse. Una revisión sin control con frecuencia es peor que si no se hiciera ninguna. Los siguientes representan un conjunto mínimo de lineamientos para hacer revisiones técnicas formales:

1. *Revise el producto, no al productor.* Una RTF involucra personas y sus egos. Si se lleva a cabo en forma adecuada, la RTF debe dejar en todos los participantes una sensación de logro. Si se efectúa de modo inapropiado, adopta un aire inquisitorial. Los errores deben señalarse en forma amable; el tono de la reunión debe ser relajado y constructivo; el trabajo no debe apenar o menospreciar a nadie. El líder de la revisión debe conducir la reunión en tono y actitud apropiados y debe detenerla de inmediato si se sale de control.
2. *Establezca una agenda y sígala*. Una de las fallas clave de las reuniones de todo tipo es la dispersión. Una RTF debe mantenerse encarrilada y dentro del programa. El líder de la revisión tiene la responsabilidad de que así sea y no debe sentir temor de llamar al orden a las personas cuando se dispersen.
3. *Limite el debate y las contestaciones.* Cuando el revisor plantee un asunto, quizá no haya acuerdo universal acerca de su efecto. En vez de perder tiempo en debatir la cuestión, ésta debe registrarse para discutirla después.
4. *Enuncie áreas de problemas, pero no intente resolver cada uno.* Una revisión no es una sesión para resolver problemas. Es frecuente que la solución de un problema la obtenga el productor, solo o con ayuda de otra persona. La solución de los problemas debe posponerse para después de la reunión de revisión.
5. *Tome notas por escrito.* A veces es buena idea que el secretario tome notas en un pizarrón a fin de que la redacción y prioridades sean evaluadas por los demás revisores a medida que la información se registra. De manera alternativa, pueden tomarse notas directamente en una computadora.
6. *Limite el número de participantes e insista en la preparación previa.* Dos cabezas piensan más que una, pero 14 no son necesariamente mejor que 4. Mantenga limitado el número de personas involucradas. Sin embargo, todos los miembros del equipo de revisión deben prepararse. El líder de la revisión tiene que solicitar comentarios por escrito (lo que proporciona un indicador de que el revisor ha inspeccionado el material).
7. *Desarrolle una lista de verificación para cada producto que sea probable que se revise.* Una lista de verificación ayuda al líder del proyecto a estructurar la RTF y a cada revisor a centrarse en los aspectos importantes. Deben desarrollarse listas para los productos del análisis, el diseño, el código e incluso las pruebas.
8. *Asigne recursos y programe tiempo para las RTF.* Para que las revisiones sean eficaces, deben programarse como tareas del proceso de software. Además, debe programarse tiempo para hacer las inevitables modificaciones que ocurrirán como resultado de la RTF.

Cita:

"Una reunión es muy frecuentemente un evento en el cual los minutos son tomados y las horas son gastadas."

Autor desconocido

Cita:

"Es una de las más hermosas compensaciones de la vida, que ningún hombre pueda sinceramente ayudar a otro sin ayudarse a sí mismo."

Ralph Waldo Emerson

9. *Dé una capacitación significativa a todos los revisores.* Para que una revisión sea eficaz, todos los revisores deben recibir cierta capacitación formal. Ésta debe hacer énfasis tanto en aspectos relacionados con el proceso como en el lado de la sicología humana de la revisión. Freedman y Weinberg [Fre90] estiman en un mes la curva de aprendizaje para que 20 personas participen de modo eficaz en una revisión.
10. *Revise las primeras revisiones.* Volver a revisar puede ser benéfico para descubrir problemas con el proceso de revisión en sí mismo. El primer producto por revisar deben ser los lineamientos de la revisión.

Debido a que son muchas las variables (número de participantes, tipo de productos del trabajo, tiempo y duración, enfoque específico de la revisión, etc.) que influyen en que una revisión sea exitosa, la organización de software debe experimentar para determinar el enfoque que mejor funcione en el contexto local.

15.6.4 Revisiones orientadas al muestreo

Idealmente, todo producto del trabajo de la ingeniería de software debe pasar por una revisión técnica. En el mundo real de los proyectos de software, los recursos son limitados y el tiempo, escaso. En consecuencia, es frecuente que las revisiones se omitan, aun cuando se reconozca su valor como un mecanismo de control de calidad.

Thelin *et al.* [The01] sugieren un proceso de revisión orientado al muestreo en el que se toman muestras de todos los productos del trabajo de ingeniería de software a fin de inspeccionarlos para determinar cuáles son más susceptibles de tener errores. Después se enfocan todos los recursos de la RTF sólo en aquellos productos en los que sea muy probable encontrar errores (con base en los datos obtenidos durante el muestreo).

Las revisiones toman tiempo, y éste estará bien invertido. Sin embargo, si hay poco tiempo y no hay otra opción, no omita las revisiones. En vez de ello, aplique la revisión orientadas al muestreo.

Para que sea eficaz, el proceso de revisión orientada al muestreo debe tratar de identificar aquellos productos del trabajo que sean objetivos principales para hacer la RTF. Para lograrlo se sugiere seguir las etapas siguientes [The01]:

1. Inspeccionar una fracción a_i de cada producto del trabajo i. Registrar el número de fallas f_i encontradas dentro de a_i.
2. Desarrollar una estimación gruesa del número de fallas en el producto del trabajo i, con la multiplicación de f_i por $1/a_i$.
3. Ordenar los productos del trabajo en orden descendente de acuerdo con la estimación gruesa del número de fallas que hay en cada uno.
4. Dedicar los recursos disponibles para la revisión a aquellos productos que tengan el número estimado más grande de fallas.

La fracción del producto del trabajo de la que se tomen muestras debe ser representativa del producto del trabajo total y suficientemente grande a fin de que tenga significado para todos los revisores que no hagan muestreo. A medida que aumenta a_i se incrementa la probabilidad de que la muestra sea una representación válida del producto del trabajo. Sin embargo, los recursos requeridos para hacer el muestreo también aumentan. El equipo de ingeniería de software debe establecer el mejor valor de a_i para los tipos de productos generados.[3]

3 Thelin *et al.*, realizaron una simulación detallada que puede ayudar a hacer esto. Consulte [The01] para mayores detalles.

CasaSegura

Aspectos de la calidad

La escena: La oficina de Doug Miller, al comenzar el proyecto de software *CasaSegura*.

Participantes: Doug Miller (gerente del equipo de ingeniería de software de *CasaSegura*) y otros miembros del equipo.

La conversación:

Doug: Sé que no hemos dedicado tiempo a desarrollar un plan de calidad para este proyecto, pero ya estamos en él y tenemos que tomar en cuenta la calidad… ¿Correcto?

Jamie: Seguro. Decidimos que conforme desarrollemos el modelo de requerimientos [capítulos 6 y 7], Ed hará un procedimiento para probar cada uno de ellos.

Doug: Eso es realmente bueno, pero no vamos a esperar a las pruebas para evaluar la calidad, ¿verdad?

Vinod: No, por supuesto que no… Hemos programado revisiones en el plan del proyecto para este incremento de software. Comenzaremos el control de calidad con las revisiones.

Jamie: Me preocupa un poco que no tengamos tiempo suficiente para hacer todas las revisiones. En realidad, sé que no lo tendremos.

Doug: Mmm… ¿Qué proponen?

Jamie: Que seleccionemos aquellos elementos del modelo de requerimientos y diseño que tengan más importancia crítica para *CasaSegura* y que los revisemos.

Vinod: Pero, ¿qué pasa si hay algo mal en una parte que no hayamos revisado?

Shakira: Leí algo sobre una técnica de muestreo [sección 15.6.4] que podría ayudarnos a determinar candidatos a la revisión (Shakira explica este enfoque.)

Jamie: Quizá… pero no estoy seguro de que tengamos tiempo incluso para tomar muestras de cada elemento de los modelos.

Vinod: Doug, ¿qué quieres que hagamos?

Doug: Tomemos algo de la programación extrema [véase el capítulo 3]. Desarrollaremos los elementos de cada modelo en parejas —dos personas— y haremos una revisión informal de cada una conforme avancemos. Entonces nos abocaremos a los elementos "críticos" para hacer una revisión más formal en equipo, pero hay que mantener esas revisiones al mínimo. De ese modo, todo será observado por más de un par de ojos y también cumpliremos nuestras fechas de entrega.

Jamie: Eso significa que vamos a tener que revisar la programación de actividades.

Doug: Así es. La calidad altera la programación de este proyecto.

15.7 Resumen

El objetivo de toda revisión técnica es detectar errores y descubrir aspectos que tendrían un efecto negativo en el software que se va a desarrollar. Entre más pronto se descubra y corrija un error, menos probable es que se propague a otros productos del trabajo de la ingeniería de software y que se amplifique, lo que provocaría un mayor esfuerzo para corregirlo.

A fin de determinar si las actividades de control de calidad funcionan, deben determinarse varias métricas. Éstas se centran en el esfuerzo requerido para realizar la revisión y los tipos y severidad de errores descubiertos durante la revisión. Una vez recabadas las métricas, se usan para evaluar la eficacia de las revisiones que se efectúen. Los datos de la industria indican que las revisiones tienen un rendimiento elevado sobre la inversión.

Un modelo de referencia para la formalidad de la revisión identifica roles de las personas, planeación y preparación, estructura de la reunión, enfoque de corrección y verificación como las características que indican el grado de formalidad con el que se realiza una revisión. Las revisiones informales son de naturaleza casual, pero pueden usarse con eficacia para detectar errores. Las revisiones formales son más estructuradas y tienen una probabilidad mayor de dar como resultado un software de alta calidad.

Las revisiones informales se caracterizan por tener una planeación y preparación mínimas y poco registro de su desarrollo. Las verificaciones de escritorio y la programación por parejas forman parte de esta categoría de revisión.

Una revisión técnica formal es una reunión estilizada que ha demostrado ser extremadamente eficaz para detectar errores. Los walkthrougs y las inspecciones establecen roles definidos para cada revisor, estimulan la planeación y la preparación previa, requieren la aplicación de lineamientos de revisión definidos y ordenan llevar registros y hacer reportes. Las revisiones

por muestreo se utilizan cuando no es posible efectuar revisiones técnicas formales para todos los productos del trabajo.

Problemas y puntos por evaluar

15.1. Explique la diferencia entre un *error* y un *defecto*.

15.2. ¿Por qué no puede esperarse a las pruebas para encontrar y corregir todos los errores del software?

15.3. Suponga que en el modelo de requerimientos se han cometido 10 errores y que cada uno se amplificará en un factor de 2:1 en el diseño, y que se cometerán otros 20 errores de diseño adicionales que luego se amplificarán en un factor de 1.5:1 en el código, donde se cometerán otros 30 errores adicionales. Suponga que todas las pruebas unitarias encontrarán 30 por ciento de todos los errores, que la integración descubrirá 30 por ciento de los restantes y que las pruebas de validación hallarán 50 por ciento de los que queden. No se efectuarán revisiones. ¿Cuántos errores saldrán al público?

15.4. Vuelva a considerar la situación descrita en el problema 15.3, pero ahora suponga que se realizan revisiones en los requerimientos, diseño y código, con 60 por ciento de eficacia en el descubrimiento de todos los errores en esa etapa. ¿Cuántos errores saldrán al público?

15.5. Estudie de nuevo la situación descrita en los problemas 15.3 y 15.4. Si cada uno de los errores que salen al público tiene un costo de $4 800 por ser detectado y corregido, y hacer lo mismo para cada error descubierto en la revisión cuesta $240, ¿cuánto dinero se ahorra por efectuar revisiones?

15.6. En sus propias palabras, describa el significado de la figura 15.4.

15.7. ¿Cuál de las características del modelo de referencia piensa usted que tiene el mayor efecto en la formalidad de la revisión? Explique por qué.

15.8. ¿Se le ocurren algunos casos en los que una verificación de escritorio genere problemas en lugar de beneficios?

15.9. Una revisión técnica formal es eficaz sólo si cada quien se prepara por adelantado. ¿Cómo se reconoce a un participante que no se haya preparado? ¿Qué haría si usted fuera el líder de la revisión?

15.10. Al considerar todos los lineamientos para la revisión presentados en la sección 15.6.3, ¿cuál piensa que sea el más importante y por qué?

Lecturas y fuentes de información adicionales

Se han escrito relativamente pocos libros sobre las revisiones de software. Algunas de las ediciones recientes que dan una guía útil incluyen los textos de Wong (*Modern Software Review*, IRM Press, 2006), Radice (*High Quality, Low Cost Software Inspections*, Paradoxicon Publishers, 2002), Wiegers (*Peer Reviews in Software: A Practical Guide*, Addison-Wesley, 2001) y Gilb y Graham (*Software Inspection*, Addison-Wesley, 1993). El de Freedman y Weinberg (*Handbook of Walkthroughs, Inspections and Technical Reviews*, Dorset House, 1990) sigue siendo un texto clásico y todavía proporciona información útil acerca de este tema tan importante.

En internet existe una amplia variedad de fuentes de información acerca de la calidad del software. En el sitio web del libro, se encuentra una lista actualizada de referencias existentes en la red mundial y que son relevantes para las revisiones de software, en la dirección **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.

CAPÍTULO

16 ASEGURAMIENTO DE LA CALIDAD DEL SOFTWARE

CONCEPTOS CLAVE

confiabilidad del software . . 376
elementos del ACS. 370
  estadístico 374
  plan 379
  tareas 371
enfoques formales. 373
estándar ISO 9001-2000 . . 378
metas. 371
seguridad del software 378
Seis Sigma 375

El enfoque de la ingeniería de software descrito en este libro se dirige a una sola meta: producir software a tiempo y de alta calidad. Pero muchos lectores se preguntarán: "¿Qué es calidad del software?".

Philip Crosby [Cro79], en su libro clásico sobre calidad, da una respuesta irónica a esta pregunta:

> El problema de la administración de la calidad no es lo que la gente ignora de ella. El problema es lo que piensan que saben...
>
> En ese sentido, la calidad tiene mucho en común con el sexo. Todo mundo lo busca (en ciertas condiciones, por supuesto). Todos creen que lo entienden (aunque no querrían explicarlo). Todos piensan que su ejecución sólo consiste en seguir las inclinaciones naturales (después de todo, lo hacemos de algún modo). Y, por supuesto, la mayoría de la gente siente que los problemas en esta área los causan las demás personas (si sólo se dieran el tiempo de hacer las cosas bien).

En realidad, la calidad es un concepto difícil (se abordó con cierto detalle en el capítulo 14).[1]

Algunos desarrolladores de software todavía creen que la calidad del software es algo por lo que hay que empezar a preocuparse una vez generado el código. Nada podría estar más lejos de la verdad... El *aseguramiento de la calidad del software* (con frecuencia llamado *administración*

UNA MIRADA RÁPIDA

¿Qué es? No basta hablar por hablar para decir que la calidad del software es importante. Tiene que 1) definirse explícitamente lo que quiere decir "calidad del software", 2) crearse un conjunto de actividades que ayuden a garantizar que todo producto de la ingeniería de software tenga alta calidad, 3) desarrollarse el control de calidad y las actividades para asegurar ésta en todo proyecto de software, 4) usarse métricas para desarrollar estrategias a fin de mejorar el proceso del software y, en consecuencia, la calidad del producto final.

¿Quién lo hace? Todos los involucrados en el proceso de ingeniería de software son los responsables de la calidad.

¿Por qué es importante? Las cosas pueden hacerse bien o pueden volverse a hacer. Si un equipo de software pone el énfasis en la calidad en todas las actividades de la ingeniería de software, se reduce la cantidad de repeticiones que debe hacer. Eso da como resultado costos más bajos y, lo que es más importante, un mejor tiempo para llegar al mercado.

¿Cuáles son las etapas? Antes de iniciar las actividades de aseguramiento de la calidad del software (ACS), es importante definir la *calidad del software* en varios niveles diferentes de abstracción. Una vez que se entiende lo que es la calidad, el equipo de software debe identificar un conjunto de actividades de ACS que filtren los errores de los productos del trabajo antes de que se aprueben.

¿Cuál es el producto final? Se crea un Plan de Aseguramiento de la Calidad del Software para definir una estrategia de ACS del equipo. Durante la modelación y codificación, el producto principal del ACS es la salida de las revisiones técnicas (véase el capítulo 15). Durante las pruebas (capítulos 17 a 20), se generan los planes y procedimientos de prueba, así como otros productos del trabajo asociados con el proceso de mejora.

¿Cómo me aseguro de que lo hice bien? Hay que encontrar los errores antes de que se vuelvan defectos... Es decir, debe trabajarse para mejorar la eficiencia en la eliminación de defectos (capítulo 23), a fin de reducir la cantidad de repeticiones que tenga que hacer el equipo del software.

1 Si no ha leído el capítulo 14, debe leerlo ahora.

de la calidad) es una actividad sombrilla (véase el capítulo 2) que se aplica en todo el proceso del software.

El aseguramiento de la calidad del software (ACS) incluye lo siguiente: 1) un proceso de ACS, 2) tareas específicas de aseguramiento y control de la calidad (incluidas revisiones técnicas y una estrategia de pruebas relacionadas entre sí), 3) prácticas eficaces de ingeniería de software (métodos y herramientas), 4) control de todos los productos del trabajo de software y de los cambios que sufren (véase el capítulo 22), 5) un procedimiento para garantizar el cumplimiento de los estándares del desarrollo de software (cuando sea aplicable) y 6) mecanismos de medición y reporte.

Este capítulo se centra en aspectos de la administración y en las actividades específicas del proceso que permiten a una organización de software garantizar que hace "las cosas correctas en el momento correcto y de la forma correcta".

16.1 Antecedentes

El control y aseguramiento de la calidad son actividades esenciales para cualquier negocio que genere productos que utilicen otras personas. Antes del siglo XX, el control de calidad era responsabilidad única del artesano que elaboraba el producto. Cuando pasó el tiempo y las técnicas de la producción en masa se hicieron comunes, el control de calidad se convirtió en una actividad ejecutada por personas diferentes de aquellas que elaboraban el producto.

Cita:

"Cometes demasiados errores equivocados."

Yogi Berra

La primera función formal de aseguramiento y control de la calidad se introdujo en los laboratorios Bell en 1916 y se difundió con rapidez al resto del mundo de la manufactura. Durante la década de 1940, sugirieron enfoques más formales del control de calidad. Éstos se basaban en la medición y en el proceso de la mejora continua [Dem86] como elementos clave de la administración de la calidad.

Actualmente, toda compañía tiene mecanismos para asegurar la calidad en sus productos. En realidad, en las últimas décadas, las afirmaciones explícitas del compromiso de una compañía con la calidad se han vuelto un mantra de la mercadotecnia.

La historia del aseguramiento de la calidad en el desarrollo del software corre de manera paralela con la historia de la calidad en la manufactura del hardware. En los primeros días de la computación (décadas de 1950 y 1960), la calidad era responsabilidad única del programador. Los estándares para asegurar la calidad del software se introdujeron en los contratos para desarrollar software militar en la década de 1970 y se extendieron con rapidez al desarrollo de software en el mundo comercial [IEE93]. Si se amplía la definición presentada al principio, el aseguramiento de la calidad del software es un "patrón planeado y sistemático de acciones" [Sch98c] que se requieren para garantizar alta calidad en el software. El alcance de la responsabilidad del aseguramiento de la calidad se caracteriza mejor si se parafrasea un comercial de un automóvil popular: "La calidad es el empleo número 1." La implicación para el software es que muchas entidades diferentes tienen responsabilidad en el aseguramiento de la calidad del software: ingenieros de software, gerentes de proyecto, clientes, vendedores y los individuos que trabajan en el grupo de ACS.

El grupo de ACS funciona como representante del cliente en el interior de la empresa. Es decir, la gente que realiza el ACS debe ver al software desde el punto de vista del cliente. ¿El software cumple adecuadamente los factores de calidad mencionados en el capítulo 14? ¿El desarrollo del software se condujo de acuerdo con estándares preestablecidos? ¿Las disciplinas técnicas han cumplido con sus roles como parte de la actividad de ACS? El grupo de ACS trata de responder éstas y otras preguntas para garantizar que se mantenga la calidad del software.

16.2 ELEMENTOS DE ASEGURAMIENTO DE LA CALIDAD DEL SOFTWARE

WebRef

En la dirección **www.swqual.com/newsletter/vol2/no1/vol2no1.html**, se encuentra un análisis profundo del ACS, que incluye una amplia variedad de definiciones.

El aseguramiento de la calidad del software incluye un rango amplio de preocupaciones y actividades que se centran en la administración de la calidad del software. Éstas se resumen como sigue [Hor03]:

Estándares. El IEEE, ISO y otras organizaciones que establecen estándares han producido una amplia variedad de ellos para ingeniería de software y documentos relacionados. Los estándares los adopta de manera voluntaria una organización de software o los impone el cliente u otros participantes. El trabajo del ACS es asegurar que los estándares que se hayan adoptado se sigan, y que todos los productos del trabajo se apeguen a ellos.

Revisiones y auditorías. Las revisiones técnicas son una actividad del control de calidad que realizan ingenieros de software para otros ingenieros de software (véase el capítulo 15). Su objetivo es detectar errores. Las auditorías son un tipo de revisión efectuada por personal de ACS con objeto de garantizar que se sigan los lineamientos de calidad en el trabajo de la ingeniería de software. Por ejemplo, una auditoría del proceso de revisión se efectúa para asegurar que las revisiones se lleven a cabo de manera que tengan la máxima probabilidad de descubrir errores.

Pruebas. Las pruebas del software (capítulos 17 a 20) son una función del control de calidad que tiene un objetivo principal: detectar errores. El trabajo del ACS es garantizar que las pruebas se planeen en forma apropiada y que se realicen con eficiencia, de modo que la probabilidad de que logren su objetivo principal sea máxima.

Colección y análisis de los errores. La única manera de mejorar es medir cómo se está haciendo algo. El ACS reúne y analiza errores y datos acerca de los defectos para entender mejor cómo se cometen los errores y qué actividades de la ingeniería de software son más apropiadas para eliminarlos.

Administración del cambio. El cambio es uno de los aspectos que más irrumpe en cualquier proyecto de software. Si no se administra en forma adecuada, lleva a la confusión y ésta casi siempre genera mala calidad. El ACS asegura que se hayan instituido prácticas adecuadas de administración del cambio (véase el capítulo 22).

Educación. Toda organización de software quiere mejorar sus prácticas de ingeniería de software. Un contribuyente clave de la mejora es la educación de los ingenieros de software, de sus gerentes y de otros participantes. La organización de ACS lleva el liderazgo en la mejora del proceso de software (capítulo 30) y es clave para proponer y patrocinar programas educativos.

Cita:

"La excelencia es la capacidad ilimitada de mejorar la calidad de lo que se tenga para ofrecer."

Rick Petin

Administración de los proveedores. Son tres las categorías de software que se adquieren a proveedores externos: *paquetes contenidos en una caja* (por ejemplo, *Office*, de Microsoft); *un shell personalizado* [Hor03], que da una estructura básica, tipo esqueleto, que se adapta de manera única a las necesidades del comprador; y *software contratado*, que se diseña y construye especialmente a partir de especificaciones provistas por la organización cliente. El trabajo de la organización de ACS es garantizar que se obtenga software de alta calidad a partir de las sugerencias de prácticas específicas de calidad que el proveedor debe seguir (cuando sea posible) y de la incorporación de cláusulas de calidad como parte de cualquier contrato con un proveedor externo.

Administración de la seguridad. Con el aumento de los delitos cibernéticos y de las nuevas regulaciones gubernamentales respecto de la privacidad, toda organización de software debe instituir políticas para proteger los datos en todos los niveles, establecer cortafuegos de protección para las *webapps* y asegurar que el software no va a ser vulnerado in-

ternamente. El ACS garantiza que para lograr la seguridad del software, se utilicen el proceso y la tecnología apropiados.

Seguridad. Debido a que el software casi siempre es un componente crucial de los sistemas humanos (como aplicaciones automotrices o aeronáuticas), la consecuencia de defectos ocultos puede ser catastrófica. El ACS es responsable de evaluar el efecto de las fallas del software y de dar los pasos que se requieren para disminuir el riesgo.

Administración de riesgos. Aunque el análisis y la mitigación de riesgos (véase el capítulo 28) es asunto de los ingenieros de software, la organización del ACS garantiza que las actividades de administración de riesgos se efectúen en forma apropiada y que se establezcan planes de contingencia relacionados con los riesgos.

Además de cada una de estas preocupaciones y actividades, el ACS tiene como preocupación dominante asegurar que las actividades de apoyo del software (como mantenimiento, líneas de ayuda, documentación y manuales) se lleven a cabo o se produzcan con calidad.

INFORMACIÓN

Recursos para la administración de la calidad

En la red mundial existen decenas de recursos para la administración de la calidad, incluidas sociedades profesionales, organizaciones emisoras de estándares y fuentes de información general. Los sitios siguientes constituyen un buen punto de partida:

American Society for Quality (ASQ), División de Software, **www.asq.org/software**
Association for Computer Machinery, **www.acm.org**, Centro de Datos y Análisis del Software (DACS), **www.dacs.dtic.mil/**
International Organization for Standardization (ISO), **www.iso.ch**
ISO SPIECE, **www.isospiece.com**
Malcolm Baldridge National Quality Award, **www.quality.nist.gov**
Software Engineering Institute, **www.sei.cmu.edu/**
Software Testing and Quality Engineering, **www.stickyminds.com**
Six Sigma Resources, **www.isixsigma.com/** **www.asq.org/sixsigma/**
TickIT International, temas sobre certificación de la calidad, **www.tickit.org/international.htm**
Total Quality Management (TQM)
- Información general: **www.gslis.utexas.edu/~rpollock/tqm.html**
- Artículos: **www.work911.com/tqmarticles.htm**
- Glosario: **www.quality.org/TQM-MSI/TQM-glossary.html**

16.3 TAREAS, METAS Y MÉTRICAS DEL ACS

El aseguramiento de la calidad del software se compone de varias tareas asociadas con dos entidades diferentes: los ingenieros de software que hacen el trabajo técnico y un grupo de ACS que tiene la responsabilidad de planear, supervisar, registrar, analizar y hacer reportes acerca de la calidad.

Los ingenieros de software abordan la calidad (y ejecutan actividades para controlarla), aplicando métodos y medidas técnicas sólidos, realizando revisiones técnicas y haciendo pruebas de software bien planeadas.

16.3.1 Tareas del ACS

El objetivo del grupo de ACS es auxiliar al equipo del software para lograr un producto final de alta calidad. El Instituto de Ingeniería de Software recomienda un conjunto de acciones de ACS que se dirigen a la planeación, supervisión, registro, análisis y elaboración de reportes para el aseguramiento de la calidad. Estas acciones son realizadas (o facilitadas) por un grupo independiente de ACS que hace lo siguiente:

¿Cuál es el rol de un grupo de ACS?

Prepara el plan de ACS para un proyecto. El plan se desarrolla como parte de la preparación del proyecto y es revisado por todos los participantes. Las acciones de aseguramiento de la calidad efectuadas por el equipo de ingeniería de software y por el grupo de ACS son dirigidas por el plan. Éste identifica las evaluaciones que se van a realizar, las auditorías y revisiones por efectuar, los estándares aplicables al proyecto, los procedimientos para reportar y dar seguimiento a los errores, los productos del trabajo que genera el grupo de ACS y la retroalimentación que se dará al equipo del software.

Participa en el desarrollo de la descripción del software del proyecto. El equipo de software selecciona un proceso para el trabajo que se va a realizar. El grupo de ACS revisa la descripción del proceso a fin de cumplir con la política organizacional, los estándares internos para el software, los estándares impuestos desde el exterior (como la norma ISO-9001) y otras partes del plan del proyecto de software.

Revisa las actividades de la ingeniería de software a fin de verificar el cumplimiento mediante el proceso definido para el software. El grupo de ACS identifica, documenta y da seguimiento a las desviaciones del proceso y verifica que se hayan hecho las correcciones pertinentes.

Audita los productos del trabajo de software designados para verificar que se cumpla con aquellos definidos como parte del proceso de software. El grupo de ACS revisa productos del trabajo seleccionados; identifica, documenta y da seguimiento a las desviaciones; verifica que se hayan hecho las correcciones necesarias y reporta periódicamente los resultados de su trabajo al gerente del proyecto.

Asegura que las desviaciones en el trabajo de software y sus productos se documenten y manejen de acuerdo con un procedimiento documentado. Las desviaciones pueden encontrarse en el plan del proyecto, la descripción del proceso, los estándares aplicables o los productos del trabajo de la ingeniería de software.

Registra toda falta de cumplimiento y la reporta a la alta dirección. Se da seguimiento a los incumplimientos hasta que son resueltos.

Además de estas acciones, el grupo de ACS coordina el control y administración del cambio (véase el capítulo 22) y ayuda a recabar y analizar métricas para el software.

16.3.2 Metas, atributos y métricas

Cita:

"La calidad nunca es un accidente; siempre es el resultado de una intención clara, un esfuerzo sincero, una dirección inteligente y una ejecución hábil; representa la elección sabia de muchas alternativas".

William A. Foster

Las acciones de ACS descritas en la sección anterior se realizan con objeto de alcanzar un conjunto de metas pragmáticas:

Calidad de los requerimientos. La corrección, completitud y consistencia del modelo de requerimientos tendrá una gran influencia en la calidad de todos los productos del trabajo que sigan. El ACS debe garantizar que el equipo de software ha revisado en forma apropiada el modelo de requerimientos a fin de alcanzar un alto nivel de calidad.

Calidad del diseño. Todo elemento del modelo del diseño debe ser evaluado por el equipo del software para asegurar que tenga alta calidad y que el diseño en sí se apegue a los requerimientos. El ACS busca atributos del diseño que sean indicadores de la calidad.

Calidad del código. El código fuente y los productos del trabajo relacionados (por ejemplo, otra información descriptiva) deben apegarse a los estándares locales de codificación y tener características que faciliten darle mantenimiento. El ACS debe identificar aquellos atributos que permitan hacer un análisis razonable de la calidad del código.

Eficacia del control de calidad. Un equipo de software debe aplicar recursos limitados, en forma tal que tenga la máxima probabilidad de lograr un resultado de alta calidad. El

FIGURA 16.1 **Metas atributos y métricas de la calidad del software**
Fuente: Adaptado de [Hya96].

Meta	Atributo	Métrica
Calidad de los requerimientos	Ambigüedad	Número de modificadores ambiguos (por ejemplo, muchos, grande, amigable, etc.)
	Completitud	Número de TBA y TBD
	Comprensibilidad	Número de secciones y subsecciones
	Volatilidad	Número de cambios por requerimiento
		Tiempo (por actividad) cuando se solicita un cambio
	Trazabilidad	Número de requerimientos no trazables hasta el diseño o código
	Claridad del modelo	Número de modelos UML
		Número de páginas descriptivas por modelo
		Número de errores de UML
Calidad del diseño	Integridad arquitectónica	Existencia del modelo arquitectónico
	Completitud de componentes	Número de componentes que se siguen hasta el modelo arquitectónico
		Complejidad del diseño del procedimiento
	Complejidad de la interfaz	Número promedio de pasos para llegar a una función o contenido normal
		Distribución apropiada
	Patrones	Número de patrones utilizados
Calidad del código	Complejidad	Complejidad ciclomática
	Facilidad de mantenimiento	Factores de diseño (capítulo 8)
	Comprensibilidad	Porcentaje de comentarios internos
		Convenciones variables de nomenclatura
	Reusabilidad	Porcentaje de componentes reutilizados
	Documentación	Índice de legibilidad
Eficacia del control de calidad	Asignación de recursos	Porcentaje de personal por hora y por actividad
	Tasa de finalización	Tiempo de terminación real *versus* lo planeado
	Eficacia de la revisión	Ver medición de la revisión (capítulo 14)
	Eficacia de las pruebas	Número de errores de importancia crítica encontrados
		Esfuerzo requerido para corregir un error
		Origen del error

ACS analiza la asignación de recursos para las revisiones y pruebas a fin de evaluar si se asignan en la forma más eficaz.

La figura 16.1 (adaptada de [Hya96]) identifica los atributos que son indicadores de la existencia de la calidad para cada una de las metas mencionadas. También se presentan las métricas que se utilizan para indicar la fortaleza relativa de un atributo.

16.4 ENFOQUES FORMALES AL ACS

En las secciones anteriores, se dijo que la calidad del software es el trabajo de cada quien y que puede lograrse por medio de una práctica competente de la ingeniería de software, así como de la aplicación de revisiones técnicas, de una estrategia de pruebas con relaciones múltiples, de un mejor control de los productos del trabajo de software y de los cambios efectuados sobre

WebRef

En la dirección **www.gslis.utexas.edu/~rpollock/tqm.html**, se encuentra información útil acerca del ACS y de los métodos formales de la calidad.

ellos, así como de la aplicación de estándares aceptados de la ingeniería de software. Además, la calidad se define en términos de una amplia variedad de atributos de la calidad y se mide (indirectamente) con el empleo de varios índices y métricas.

En las últimas tres décadas, un segmento pequeño pero sonoro de la comunidad de la ingeniería de software ha afirmado que se requiere un enfoque más formal para el ACS. Puede decirse que un programa de cómputo es un objeto matemático. Para cada lenguaje de programación, es posible definir una sintaxis y semántica rigurosas, y se dispone de un enfoque igualmente riguroso para la especificación de los requerimientos del software (véase el capítulo 21). Si el modelo de los requerimientos (especificación) y el lenguaje de programación se representan en forma rigurosa, debe ser posible usar una demostración matemática para la corrección, de modo que se confirme que un programa se ajusta exactamente a sus especificaciones.

Los intentos de demostrar la corrección de un programa no son nuevos. Dijkstra [Dij76a] y Linger, Mills y Witt [Lin79], entre otros, han invocado pruebas de la corrección de programas y las han relacionado con el uso de conceptos de programación estructurada (véase el capítulo 10).

16.5 Aseguramiento estadístico de la calidad del software

El aseguramiento estadístico de la calidad del software refleja una tendencia creciente en la industria para que se vuelva más cuantitativo respecto de la calidad. Para el software, el aseguramiento estadístico de la calidad implica los pasos siguientes:

? ¿Qué pasos se requieren para efectuar el ACS estadístico?

1. Se recaba y clasifica la información acerca de errores y defectos del software.
2. Se hace un intento por rastrear cada error y defecto hasta sus primeras causas (por ejemplo, no conformidad con las especificaciones, error de diseño, violación de los estándares, mala comunicación con el cliente, etc.).
3. Con el uso del Principio de Pareto (80 por ciento de los defectos se debe a 20 por ciento de todas las causas posibles), se identifica 20 por ciento de las causas de errores y defectos (las *pocas vitales*).
4. Una vez identificadas las pocas causas vitales, se corrigen los problemas que han dado origen a los errores y defectos.

Este concepto relativamente simple representa un paso importante hacia la creación de un proceso adaptativo del software en el que se hacen cambios para mejorar aquellos elementos del proceso que introducen errores.

16.5.1 Ejemplo general

Cita:

"Un análisis estadístico, si se realiza en forma apropiada, es una disección delicada de las incertidumbres, una cirugía de las suposiciones."

M. J. Moroney

A fin de ilustrar el uso de los métodos estadísticos para el trabajo de ingeniería de software, suponga que una organización de ingeniería de software recaba información sobre los errores y defectos cometidos en un periodo de un año. Algunos de dichos errores se descubren a medida que se desarrolla el software. Otros (defectos) se encuentran después de haber liberado el software a sus usuarios finales. Aunque se descubren cientos de problemas diferentes, todos pueden rastrearse hasta una (o más) de las causas siguientes:

- Especificaciones erróneas o incompletas (EEI)
- Mala interpretación de la comunicación con el cliente (MCC)
- Desviación intencional de las especificaciones (DIE)
- Violación de los estándares de programación (VEP)
- Error en la representación de los datos (ERD)

FIGURA 16.2
Colección de datos para hacer ACS estadístico

	Total		Serio		Moderado		Menor	
Error	**No.**	**%**	**No.**	**%**	**No.**	**%**	**No.**	**%**
IES	205	22%	34	27%	68	18%	103	24%
MCC	156	17%	12	9%	68	18%	76	17%
IDS	48	5%	1	1%	24	6%	23	5%
VPS	25	3%	0	0%	15	4%	10	2%
EDR	130	14%	26	20%	68	18%	36	8%
ICI	58	6%	9	7%	18	5%	31	7%
EDL	45	5%	14	11%	12	3%	19	4%
IET	95	10%	12	9%	35	9%	48	11%
IID	36	4%	2	2%	20	5%	14	3%
PLT	60	6%	15	12%	19	5%	26	6%
HCI	28	3%	3	2%	17	4%	8	2%
MIS	56	6%	0	0%	15	4%	41	9%
Totales	942	100%	128	100%	379	100%	435	100%

- Interfaz componente inconsistente (ICI)
- Error en el diseño lógico (EDL)
- Pruebas incompletas o erróneas (PIE)
- Documentación inexacta o incompleta (DII)
- Error en la traducción del lenguaje de programación del diseño (LPD)
- Interfaz humano/computadora ambigua o inconsistente (IHC)
- Varios (V)

Cita:

"20 por ciento del código tiene 80 por cierto de los errores. Encuéntrelos, corríjalos".

Lowell Arthur

Para aplicar el ACS estadístico, se elabora la tabla de la figura 16.2. La tabla indica que EEI, MCC y ERD son las pocas causas vitales que originan 53 por ciento de todos los errores. Sin embargo, debe notarse que EEI, ERD, LPD y EDL se habrían seleccionado como las pocas causas vitales si se consideran sólo errores serios. Una vez que las pocas causas vitales han sido determinadas, la organización de ingeniería de software comienza su acción correctiva. Por ejemplo, a fin de corregir el MCC, deben implementarse técnicas para recabar requerimientos (capítulo 5) que mejoren la calidad de la comunicación y las especificaciones con el cliente. Para mejorar el ERD, deben adquirirse herramientas para desarrollar la modelación de casos y realizar datos y revisiones del diseño más significativos.

Es importante notar que la acción correctiva se centra sobre todo en las pocas causas vitales. En tanto éstas se corrigen, nuevas candidatas se van a la cumbre de la pila.

Las técnicas para el aseguramiento correctivo han sido propuestas para dar una mejora sustancial de la calidad [Art97]. En ciertos casos, las organizaciones de software han tenido una reducción anual de 50 por ciento en defectos después de aplicar esta técnica.

La aplicación del ACS estadístico y el Principio de Pareto se resumen en una sola oración: *Pasa tu tiempo viendo las cosas que realmente importan, pero primero asegúrate de que entiendes lo que realmente importa...*

16.5.2 Seis Sigma para la ingeniería de software

Seis Sigma es la estrategia más ampliamente usada hoy para el aseguramiento estadístico de la calidad en la industria. La estrategia Seis Sigma fue popularizada originalmente por Motorola en la década de 1980 y "es una metodología rigurosa y disciplinada que usa datos y análisis estadísticos para medir y mejorar el desempeño operativo de una compañía, identificando y eliminando defectos en procesos de manufactura y servicios" [ISI08]. El término Seis Sigma se deriva

de seis desviaciones estándar —3.4 casos (defectos) por millón de ocurrencias—, lo que implica un estándar de calidad extremadamente alto. La metodología Seis Sigma define tres etapas fundamentales:

¿Cuáles son las etapas fundamentales de la metodología Seis Sigma?

- *Definir* los requerimientos del cliente y los que se le entregan, así como las metas del proyecto a través de métodos bien definidos de comunicación con el cliente.
- *Medir* el proceso existente y su resultado para determinar el desempeño actual de la calidad (recabar métricas para los defectos).
- *Analizar* las métricas de los defectos y determinar las pocas causas vitales.

Si se trata de un proceso de software existente que se requiere mejorar, Seis Sigma sugiere dos etapas adicionales:

- *Mejorar* el proceso, eliminando las causas originales de los defectos.
- *Controlar* el proceso para asegurar que el trabajo futuro no vuelva a introducir las causas de los defectos.

Estas etapas fundamentales y adicionales en ocasiones son conocidas como método DMAMC (definir, medir, analizar, mejorar y controlar).

Si una organización va a desarrollar un proceso de software (en vez de mejorar uno existente), a las etapas fundamentales se agregan las siguientes:

- *Diseñar* el proceso para 1) evitar las causas originales de los defectos y 2) cumplir los requerimientos del cliente.
- *Verificar* que el modelo del proceso en realidad evite los defectos y cumpla los requerimientos del cliente.

Esta variación en ocasiones es denominada método DMADV (definir, medir, analizar, diseñar y verificar).

El estudio detallado de Seis Sigma se deja a fuentes dedicadas a ese tema. Si el lector tiene interés al respecto, consulte [ISI08], [Pyz303] y [Sne03].

16.6 Confiabilidad del software

Cita:

"El precio inevitable de la confiabilidad es la simplicidad."

C. A. R. Hoare

No hay duda de que la confiabilidad de un programa de cómputo es un elemento importante de su calidad general. Si un programa falla repetida y frecuentemente en su desempeño, importa poco si otros factores de la calidad del software son aceptables.

La confiabilidad del software, a diferencia de muchos otros factores de la calidad, se mide y estima directamente mediante el uso de datos históricos del desarrollo. La *confiabilidad del software* se define en términos estadísticos como "la probabilidad que tiene un programa de cómputo de operar sin fallas en un ambiente específico por un tiempo específico" [Mus87]. Para ilustrar lo anterior, digamos que se estima que el programa *X* tiene una confiabilidad de 0.999 durante ocho horas de procesamiento continuo. En otras palabras, si el programa *X* fuera a ejecutarse 1 000 veces y requiriera un total de ocho horas de tiempo de procesamiento continuo (tiempo de procesamiento), es probable que operara correctamente (sin fallas) 999 veces.

Siempre que se trate de la confiabilidad del software, surge una pregunta crucial: ¿qué significa el término *falla*? En el contexto de cualquier análisis de la calidad y confiabilidad del software, la falla significa la falta de conformidad con los requerimientos del software. Pero, incluso con esta definición, hay gradaciones. Las fallas pueden ser leves o catastróficas. Una falla podría corregirse en segundos, mientras que otra tal vez requiera de varias semanas o meses de trabajo para ser corregida. Para complicar más el asunto, la corrección de una falla quizá dé como resultado la introducción de otros errores que a su vez originen otras fallas.

16.6.1 Mediciones de la confiabilidad y disponibilidad

PUNTO CLAVE

Los problemas de confiabilidad del software casi siempre pueden seguirse hasta encontrar defectos en el diseño o en la implementación.

Los primeros trabajos sobre confiabilidad del software trataban de extrapolar la teoría matemática de la confiabilidad del hardware a la predicción de la confiabilidad del software. La mayor parte de modelos relacionados con el hardware se abocan a la falla debida al uso, en lugar de a la que tiene su origen en los defectos de diseño. En el hardware, las fallas debidas al uso físico (por ejemplo, los efectos de temperatura, corrosión y golpes) son más probables que las debidas al diseño. Desafortunadamente, con el software ocurre lo contrario. En realidad, todas las fallas del software pueden rastrearse en problemas de diseño o de implementación; el uso (véase el capítulo 1) no entra en el escenario.

Ha habido un debate permanente acerca de la relación que existe entre los conceptos clave en la confiabilidad del hardware y su aplicabilidad al software. Aunque es posible establecer un vínculo irrefutable, es útil considerar algunos conceptos sencillos que se aplican a ambos elementos del sistema.

PUNTO CLAVE

Es importante observar que el tiempo medio entre fallas y otras medidas relacionadas se basa en tiempo del CPU, no en tiempo de reloj.

Si se considera un sistema basado en computadora, una medida sencilla de su confiabilidad es el *tiempo medio entre fallas* (TMEF):

$$\text{TMEF} = \text{TMPF} + \text{TMPR}$$

donde las siglas TMPF y TMPR significan *tiempo medio para la falla* y *tiempo medio para la reparación*,[2] respectivamente.

Muchos investigadores afirman que el TMEF es una medición más útil que otras relacionadas con la calidad del software que se estudian en el capítulo 23. En pocas palabras, a un usuario final le preocupan las fallas, no la cuenta total de defectos. Como cada defecto contenido en un programa no tiene la misma tasa de fallas, la cuenta total de defectos indica muy poco acerca de la confiabilidad del sistema. Por ejemplo, considere un programa que haya estado en operación durante 3 000 horas de procesador sin falla. Muchos defectos de este programa estarían sin detectar durante decenas de miles de horas antes de ser descubiertos. El TMEF de tales errores oscuros podría ser de 30 000 o hasta 60 000 horas de procesador. Otros defectos, no descubiertos, podrían tener una tasa de fallas de 4 000 a 5 000 horas. Aun si cada uno de los errores en esta categoría (los que tienen un TMEF largo) se eliminara, el efecto que tendrían sobre el software sería despreciable.

CONSEJO

Algunos aspectos de la disponibilidad (que no se estudian aquí) no tienen que ver con las fallas. Por ejemplo, la programación del tiempo fuera de operación (para funciones de apoyo) hace que el software no esté disponible.

Sin embargo, el TMEF puede ser problemático por dos razones: 1) proyecta un tiempo entre fallas, pero no da una tasa de fallas proyectada y 2) puede interpretarse mal, como la vida promedio, cuando *no* es esto lo que implica.

Una medición alternativa de confiabilidad es la de las *fallas en el tiempo* (FET): medición estadística de cuántas fallas tendrá un componente en mil millones de horas de operación. Por tanto, 1 FET es equivalente a una falla en cada mil millones de horas de operación.

Además de una medida de la confiabilidad, también debe desarrollarse otra para la disponibilidad. La *disponibilidad del software* es la probabilidad de que un programa opere de acuerdo con los requerimientos en un momento determinado de tiempo, y se define así:

$$\text{Disponibilidad} = \frac{\text{TMPF}}{\text{TMPF} + \text{TMPR}} \times 100\%$$

La medición del TMEF para la confiabilidad es igualmente sensible al TMPF y al TMPR. La medición de la disponibilidad es un poco más sensible al TMPR, que es una medición indirecta de la facilidad que tiene el software para recibir mantenimiento.

---

2 Aunque tal vez se requiera depurar (y hacer otras correcciones relacionadas) como consecuencia de la falla, en muchos casos el software funcionará de manera apropiada después de reiniciar, sin ningún otro cambio.

16.6.2 Seguridad del software

Cita:

"La seguridad de las personas debe ser la ley máxima."

Cicerón

La *seguridad del software* es una actividad del aseguramiento del software que se centra en la identificación y evaluación de los peligros potenciales que podrían afectarlo negativamente y que podrían ocasionar que falle todo el sistema. Si los peligros se identifican al principio del proceso del software, las características de su diseño se especifican de modo que los eliminen o controlen.

Como parte de la seguridad del software, se lleva a cabo un proceso de modelado y análisis. Inicialmente se identifican los peligros y se clasifican según su riesgo. Por ejemplo, algunos de los peligros asociados con un control de crucero basado en computadora para un automóvil podrían ser los siguientes: 1) ocasionar una aceleración incontrolada que no pudiera detenerse, 2) no responder a la presión en el pedal de frenado (porque se apague), 3) no encender cuando se active el interruptor y 4) perder o ganar velocidad poco a poco. Una vez identificados estos peligros en el nivel del sistema, se utilizan técnicas de análisis para asignar severidad y probabilidad de ocurrencia a cada uno.[3] Para ser eficaz, el software debe analizarse en el contexto de todo el sistema. Por ejemplo, un error sutil en la entrada de un usuario (las personas son componentes del sistema) podría ampliarse por una falla del software y producir datos de control que situaran equivocadamente un dispositivo mecánico. Si y sólo si se encontrara un único conjunto de condiciones ambientales externas, la posición falsa del dispositivo mecánico ocasionaría una falla desastrosa. Podrían usarse técnicas de análisis [Eri05], tales como árbol de fallas, lógica en tiempo real y modelos de red de Petri, para predecir la cadena de eventos que ocasionarían los peligros, así como la probabilidad de ocurrir que tendría cada uno de los eventos para generar la cadena.

Cita:

"No puedo imaginar ninguna condición que hiciera que esta nave se hundiera. La construcción naval moderna ha llegado más allá de eso".

E. I. Smith, capitán del *Titanic*

Una vez identificados y analizados los peligros, pueden especificarse requerimientos relacionados con la seguridad para el software. Es decir, la especificación contendría una lista de eventos indeseables y las respuestas deseadas del sistema ante ellos. Después se indicaría el papel del software en la administración indeseable de los mismos.

Aunque la confiabilidad y la seguridad del software están muy relacionadas, es importante entender la sutil diferencia entre ellas. La primera utiliza técnicas de análisis estadístico para determinar la probabilidad de que ocurra una falla del software. Sin embargo, la ocurrencia de una falla no necesariamente da como resultado un peligro o riesgo. La seguridad del software examina las formas en las que las fallas generan condiciones que llevan a un peligro. Es decir, las fallas no se consideran en el vacío, sino que se evalúan en el contexto de la totalidad del sistema basado en computadora y de su ambiente.

WebRef

En la dirección **www.safeware-eng.com/**, se encuentran varios artículos sobre seguridad del software.

El estudio exhaustivo de la seguridad del software está más allá del alcance de este libro. Si el lector está interesado en la seguridad del software y en otros aspectos relacionados, consulte [Smi05], [Dun02] y [Lev95].

16.7 Las normas de calidad ISO 9000[4]

Un *sistema de aseguramiento de la calidad* se define como la estructura organizacional, responsabilidades, procedimientos, procesos y recursos necesarios para implementar la administración de la calidad [ANS87]. Los sistemas de aseguramiento de la calidad se crean para ayudar a las organizaciones a asegurar que sus productos y servicios satisfagan las expectativas del con-

3 Este enfoque es similar a los métodos de análisis del riesgo descritos en el capítulo 28. La diferencia principal es el énfasis que se pone en aspectos de la tecnología en lugar de en los relacionados con el proyecto.

4 Esta sección, escrita por Michael Stovski, ha sido adaptada a partir de "Fundamentos de ISO 9000", libro de trabajo desarrollado para *Essential Software Engineering*, video desarrollado por R. S. Pressman & Associates, Inc. Se reimprime con su autorización.

sumidor gracias a que cumplan con sus especificaciones. Estos sistemas cubren una amplia variedad de actividades, que contemplan todo el ciclo de vida del producto, incluidos planeación, control, medición, pruebas e informes, así como la mejora de los niveles de calidad en todo el proceso de desarrollo y manufactura. La norma ISO 9000 describe en términos generales los elementos de aseguramiento de la calidad que se aplican a cualquier negocio, sin importar los productos o servicios ofrecidos.

Para registrarse en alguno de los modelos del sistema de aseguramiento de la calidad contenidos en la ISO 9000, por medio de auditores externos se revisan en detalle el sistema y las operaciones de calidad de una compañía, respecto del cumplimiento del estándar y de la operación eficaz. Después de un registro exitoso, el grupo de registro representado por los auditores emite un certificado para la compañía. Auditorías semestrales de supervisión aseguran el cumplimiento continuo de la norma.

WebRef

En la dirección **www.tantara.ab.ca/info.htm**, se encuentran muchos vínculos hacia los recursos de la norma ISO 9000/9001.

Los requerimientos esbozados por la norma ISO 9001:2000 se dirigen a temas tales como responsabilidad de la administración, sistema de calidad, revisión del contrato, control del diseño, documentación y control de datos, identificación del producto y su seguimiento, control del proceso, inspección y pruebas, acciones correctivas y preventivas, registros del control de calidad, auditorías internas de calidad, capacitación, servicio y técnicas estadísticas. A fin de que una organización de software se registre en la ISO 9001:2000, debe establecer políticas y procedimientos que cumplan cada uno de los requerimientos mencionados (y otros más), y después demostrar que sigue dichas políticas y procedimientos. Si el lector desea más información sobre la norma ISO 9001:2000, consulte [Ant06], [Mut03] o [Dob04].

INFORMACIÓN

La norma ISO 9001:2000

La descripción siguiente define los elementos básicos de la norma ISO 9001:2000. Información completa sobre la misma se obtiene en la Organización Internacional de Normas (**www.iso.ch**) y en otras fuentes de internet (como en **www.praxiom.com**).

Establecer los elementos de un sistema de administración de la calidad.
- Desarrollar, implementar y mejorar el sistema.
- Definir una política que ponga el énfasis en la importancia del sistema.

Documentar el sistema de calidad.
- Describir el proceso.
- Producir un manual de operación.
- Desarrollar métodos para controlar (actualizar) documentos.
- Establecer métodos de registro.

Apoyar el control y aseguramiento de la calidad.
- Promover la importancia de la calidad entre todos los participantes.
- Centrarse en la satisfacción del cliente.
- Definir un plan de calidad que se aboque a los objetivos, responsabilidades y autoridad.
- Definir mecanismos de comunicación entre los participantes.

Establecer mecanismos de revisión para el sistema de administración de la calidad.
- Identificar métodos de revisión y mecanismos de retroalimentación.
- Definir procedimientos para dar seguimiento.

Identificar recursos para la calidad, incluidos personal, capacitación y elementos de la infraestructura.
- Establecer mecanismos de control.
- Para la planeación
- Para los requerimientos del cliente
- Para las actividades técnicas (tales como análisis, diseño y pruebas)
- Para la vigilancia y administración del proyecto

Definir métodos de corrección.
- Evaluar datos y métricas de la calidad.
- Definir el enfoque para la mejora continua del proceso y la calidad.

16.8 EL PLAN DE ACS

El *Plan de ACS* proporciona un mapa de ruta para instituir el aseguramiento de la calidad del software. Desarrollado por el grupo de ACS (o por el equipo del software si no existe un grupo de ACS), el plan funciona como plantilla para las actividades de ACS que se instituyen para cada proyecto de software.

La IEEE [IEEE93] ha publicado una norma para el ACS. Ésta recomienda una estructura que identifica lo siguiente: 1) propósito y alcance del plan, 2) descripción de todos los productos del trabajo de ingeniería de software (tales como modelos, documentos, código fuente, etc.) que se ubiquen dentro del ámbito del ACS, 3) todas las normas y prácticas aplicables que se utilicen durante el proceso del software, 4) acciones y tareas del ACS (incluidas revisiones y auditorías) y su ubicación en el proceso del software, 5) herramientas y métodos que den apoyo a las acciones y tareas de ACS, 6) procedimientos para la administración de la configuración del software (véase el capítulo 22), 7) métodos para unificar las salvaguardas y para mantener todos los registros relacionados con el ACS y 8) roles y responsabilidades relacionados con la calidad del producto.

HERRAMIENTAS DE SOFTWARE

Administración de la calidad del software

Objetivos: El objetivo de las herramientas del ACS es ayudar al equipo del proyecto a evaluar y mejorar la calidad del producto del trabajo de software.

Mecánica: La mecánica de las herramientas varía. En general, el objetivo consiste en evaluar la calidad de un producto específico. Nota: Es frecuente que dentro de la categoría de herramientas para el ACS, se incluya una amplia variedad de herramientas para someter a prueba al software (véanse los capítulos 17 a 20).

Herramientas representativas[5]

ARM, desarrollada por la NASA (**state.gsfc.nasa.gov/tools/index.html**), proporciona mediciones que se utilizan para evaluar la calidad de un documento de requerimientos de software.

QPR ProcessGuide and Scorecard, desarrollada por QPR Software (**www.qpronline.com**), da apoyo para establecer Seis Sigma y otros enfoques de administración de la calidad.

Quality Tools and Templates, desarrollada por iSixSigma (**www.isixsigma.com/tt/**), describe un amplio abanico de herramientas y métodos útiles para la administración de la calidad.

NASA Quality Resources, desarrollada por el Centro Coddard de Vuelos Espaciales (**sw-assurance.gsfc.nasa.gov/index.php**), contiene formatos, plantillas, listas de verificación y herramientas que son útiles para el ACS.

16.9 RESUMEN

El aseguramiento de la calidad del software es una actividad sombrilla de la ingeniería de software que se aplica en cada etapa del proceso del software. El ACS incluye procedimientos para la aplicación eficaz de métodos y herramientas, supervisa las actividades de control de calidad, tales como las revisiones técnicas y las pruebas del software, procedimientos para la administración del cambio, y procedimientos para asegurar el cumplimiento de las normas y mecanismos de medición y elaboración de reportes.

Para llevar a cabo el aseguramiento de la calidad del software de manera adecuada, deben recabarse, evaluarse y divulgarse datos sobre el proceso de la ingeniería de software. Los métodos estadísticos aplicados al ACS ayudan a mejorar la calidad del producto y del proceso de software mismo. Los modelos de confiabilidad del software amplían las mediciones, lo que permite que los datos obtenidos acerca de los defectos se extrapolen hacia tasas de falla proyectadas y hacia la elaboración de pronósticos de confiabilidad.

En resumen, deben tomarse en cuenta las palabras de Dunn y Ullman [Dun82]: "El aseguramiento de la calidad del software es el mapeo de los preceptos administrativos y de las disciplinas de diseño del aseguramiento de la calidad, en el ámbito administrativo y tecnológico aplicable a la ingeniería de software." La capacidad de asegurar la calidad es la medida de una

5 Las herramientas mencionadas aquí no son obligatorias, sino una muestra de las que hay en esta categoría. En la mayoría de casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

disciplina madura de la ingeniería. Cuando el mapeo se lleva a cabo con éxito, el resultado es una ingeniería de software madura.

Problemas y puntos por evaluar

16.1. Algunas personas afirman que "el control de la variación es el corazón del control de calidad". Como todo programa que se crea es diferente de cualquier otro programa, ¿cuáles son las variaciones que se buscan y cómo se controlan?

16.2. ¿Es posible evaluar la calidad del software si el cliente cambia continuamente lo que se supone que debe hacerse?

16.3. La calidad y confiabilidad son conceptos relacionados, pero difieren en lo fundamental por varias razones. Analice las diferencias.

16.4. ¿Un programa puede corregirse y aún así ser confiable? Explique su respuesta.

16.5. ¿Un programa puede corregirse y tener buena calidad? Explique lo que responda.

16.6. ¿Por qué es frecuente que haya tensiones entre el grupo de ingeniería de software y el del aseguramiento de la calidad? ¿Es saludable eso?

16.7. El lector tiene la responsabilidad de mejorar la calidad del software en su organización. ¿Qué es lo primero que debe hacer? ¿Qué es lo siguiente?

16.8. Además de contar los errores y defectos, ¿hay otras características cuantificables de software que impliquen calidad? ¿Cuáles son y cómo podrían medirse directamente?

16.9. El concepto del tiempo medio para la falla del software es objeto de críticas. Explique por qué.

16.10. Considere dos sistemas cuya seguridad sea crítica y que estén controlados por computadora. Enliste al menos tres peligros que se relacionen directamente con fallas del software.

16.11. Obtenga una copia de las normas ISO 9001:2000 e ISO 9000-3. Prepare una presentación que analice tres requerimientos de ISO 9001 y la forma en la que se apliquen en el contexto del software.

Lecturas y fuentes de información adicionales

Los libros de Hoyle (*Quality Management Fundamentals*, Butterworth-Heinemann, 2007), Tian (*Software Quality Engineering*, Wiley-IEEE Computer Society Press, 2005), El Emam (*The ROI from Software Quality*, Auerbach, 2005) y Horch (*Practical Guide to Software Quality Management*, Artech House, 2003), y Nance y Arthur (*Managing Software Quality*, Springer, 2002) son presentaciones excelentes en el nivel de administración acerca de los beneficios de los programas formales de aseguramiento de la calidad del software de computadora. Las obras de Deming [Dem86], Juran (*Juran on Quality by Design*, Free Press, 1992) y Crosby ([Cro79], así como *Quality is Still Free*, McGraw-Hill, 1995) no se abocan al software, pero son una lectura obligada para los altos directivos que tengan responsabilidades en el desarrollo del software. Gluckman y Roome (*Everyday Heroes of the Quality Movement*, Dorset House, 1993) humanizan los aspectos de la calidad a través de la historia de los actores participantes en el proceso. Kan (*Metrics and Models in Software Quality Engineering*, Addison-Wesley, 1995) presenta un enfoque cuantitativo de la calidad del software.

Los libros de Evans (*Total Quality: Management, Organization and Strategy*, 4a. ed., South Western College Publishing, 2004), Bru (*Six Sigma for Managers*, McGraw-Hill, 2005) y Dobb (*ISO 9001:2000 Quality Registration Step-by-Step*, 3a. ed., Butterworth-Heinemann, 2004) son representativos de los muchos que se han escrito sobre Seis Sigma e ISO 9001:2000, respectivamente.

Pham (*System Software Reliability,* Springer, 2006), Musa (*Software Reliability Engineering: More Reliable Software, Faster Development and Testing*, 2a. ed., McGraw-Hill, 2004) y Peled (*Software Reliability Methods*, Springer, 2001) proporcionan guías prácticas que describen los métodos para medir y analizar la confiabilidad del software.

Vincoli (*Basic Guide to System Safety*, Wiley 2006), Dhillon (*Engineering Safety*, World Scientific Publishing Co., Inc., 2003), Hermann (*Software Safety and Reliability*, Wiley-IEEE Computer Society Press, 2000), Storey (*Safety-Critical Computer Systems*, Addison-Wesley, 1996) y Leveson [Lev95] aportan los análisis más exhaustivos que se hayan publicado hasta la fecha acerca de la seguridad del software y del sistema. Además, Van

der Meulen (*Definitions for Hardware and Software Safety Engineers*, Springer-Verlag, 2000) ofrece un compendio completo de conceptos y términos importantes para la confiabilidad y la seguridad; Gartner (*Testing Safety-Related Software*, Springer-Verlag, 1999) ofrece una guía especializada para probar sistemas cuya seguridad sea crítica; Friedman y Voas (*Software Assesment: Reliability Safety and Testability*, Wiley, 1995) proveen modelos útiles para evaluar la confiabilidad y la seguridad. Ericson (*Hazard Analysis Techniques for System Safety*, Wiley, 2005) estudia el dominio cada vez más importante del análisis de los peligros.

En internet, hay una amplia variedad de fuentes de información sobre el aseguramiento de la calidad del software y otros temas relacionados. En el sitio web del libro, **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**, existe una lista actualizada de referencias existentes en la red mundial que son relevantes para el ACS.

CAPÍTULO

17

ESTRATEGIAS DE PRUEBA DE SOFTWARE

CONCEPTOS CLAVE

depuración 404
grupo de prueba independiente 386
prueba alfa 400
prueba beta 400
prueba de clase 398
prueba de despliegue 403
prueba de integración 391
prueba de regresión 398
prueba de unidad 389
prueba de validación 399
prueba del sistema 401
revisión de la configuración 400
V&V 387

Una estrategia de prueba de software proporciona una guía que describe los pasos que deben realizarse como parte de la prueba, cuándo se planean y se llevan a cabo dichos pasos, y cuánto esfuerzo, tiempo y recursos se requerirán. Por tanto, cualquier estrategia de prueba debe incorporar la planificación de la prueba, el diseño de casos de prueba, la ejecución de la prueba y la recolección y evaluación de los resultados.

Una estrategia de prueba de software debe ser suficientemente flexible para promover un uso personalizado de la prueba. Al mismo tiempo, debe ser suficientemente rígida para alentar la planificación razonable y el seguimiento de la gestión conforme avanza el proyecto. Shooman [Sho83] analiza estos temas:

> En muchas formas, la prueba es un proceso de individualización, y el número de tipos diferentes de pruebas varía tanto como los diferentes acercamientos para su desarrollo. Durante muchos años, la única defensa contra los errores de programación fue el diseño cuidadoso y la inteligencia natural del programador. Ahora estamos en una era en la que modernas técnicas de diseño (y revisiones técnicas) ayudan a reducir el número de errores iniciales que son inherentes al código. De igual modo, diferentes métodos de prueba comienzan a agruparse en métodos y filosofías distintos.

Estos "enfoques y filosofías" a los que denomino *estrategias* son el tema que se presenta en este capítulo. En los capítulos 18, 19 y 20 se exponen los métodos y técnicas de prueba que permiten desarrollar la estrategia.

UNA MIRADA RÁPIDA

¿Qué es? El software se prueba para descubrir errores que se cometieron de manera inadvertida conforme se diseñó y construyó. Pero, ¿cómo se realizan las pruebas? ¿Debe realizarse un plan formal para las mismas? ¿Debe probarse el programa completo, como un todo, o aplicar pruebas sólo sobre una pequeña parte de él? ¿Debe volverse a aplicar las pruebas que ya se realizaron mientras se agregan nuevos componentes a un sistema grande? ¿Cuándo debe involucrarse al cliente? Éstas y muchas otras preguntas se responden cuando se desarrolla una estrategia de prueba de software.

¿Quién lo hace? El gerente de proyecto, los ingenieros de software y los especialistas en pruebas desarrollan una estrategia para probar el software.

¿Por qué es importante? Con frecuencia, la prueba requiere más esfuerzo que cualquiera otra acción de ingeniería del software. Si se realiza sin orden, se desperdicia tiempo, se emplea esfuerzo innecesario y, todavía peor, es posible que algunos errores pasen desapercibidos. Por tanto, parecería razonable establecer una estrategia sistemática para probar el software.

¿Cuáles son los pasos? La prueba comienza "por lo pequeño" y avanza "hacia lo grande". Es decir que las primeras etapas de prueba se enfocan sobre un solo componente o un pequeño grupo de componentes relacionados y se aplican pruebas para descubrir errores en los datos y en la lógica de procesamiento que se encapsularon en los componentes. Después de probar éstos, deben integrarse hasta que se construya el sistema completo. En este punto, se ejecuta una serie de pruebas de orden superior para descubrir errores en la satisfacción de los requerimientos del cliente. Conforme se descubren, los errores deben diagnosticarse y corregirse usando un proceso que se llama depuración.

¿Cuál es el producto final? Una *Especificación pruebas* documenta la forma en la que el equipo de software prepara la prueba al definir un plan que describe una estrategia global y un procedimiento con pasos de prueba específicos y los tipos de pruebas que se realizarán.

¿Cómo me aseguro de que lo hice bien? Al revisar la *Especificación pruebas* antes de realizar las pruebas, es posible valorar si están completos los casos de prueba y las tareas de la misma. Un plan de prueba y procedimientos efectivos conducirán a la construcción ordenada del software y al descubrimiento de errores en cada etapa del proceso de construcción.

17.1 Un enfoque estratégico para la prueba de software

La prueba es un conjunto de actividades que pueden planearse por adelantado y realizarse de manera sistemática. Por esta razón, durante el proceso de software, debe definirse una plantilla para la prueba del software: un conjunto de pasos que incluyen métodos de prueba y técnicas de diseño de casos de prueba específicos.

En la literatura sobre el tema, se han propuesto algunas estrategias de prueba de software. Todas proporcionan una plantilla para la prueba y tienen las siguientes características genéricas:

- Para realizar una prueba efectiva, debe realizar revisiones técnicas efectivas (capítulo 15). Al hacerlo, eliminará muchos errores antes de comenzar la prueba.
- La prueba comienza en los componentes y opera "hacia afuera", hacia la integración de todo el sistema de cómputo.
- Diferentes técnicas de prueba son adecuadas para distintos enfoques de ingeniería de software y en diferentes momentos en el tiempo.
- Las pruebas las realiza el desarrollador del software y (para proyectos grandes) un grupo de prueba independiente.
- Prueba y depuración son actividades diferentes, pero la depuración debe incluirse en cualquier estrategia de prueba.

WebRef

En **www.mtsu.edu/~storm** pueden encontrarse útiles recursos para la prueba de software.

Una estrategia para la prueba de software debe incluir pruebas de bajo nivel, que son necesarias para verificar que un pequeño segmento de código fuente se implementó correctamente, así como pruebas de alto nivel, que validan las principales funciones del sistema a partir de los requerimientos del cliente. Una estrategia debe proporcionar una guía para el profesional y un conjunto de guías para el jefe de proyecto. Puesto que los pasos de la estrategia de prueba ocurren cuando comienza a aumentar la presión por las fechas límite, el avance debe ser medible y los problemas deben salir a la superficie tan pronto como sea posible.

17.1.1 Verificación y validación

Cita:

"Probar es la parte inevitable de cualquier esfuerzo responsable por desarrollar un sistema de software".

William Howden

La prueba de software es un elemento de un tema más amplio que usualmente se conoce como verificación y validación (V&V). La *verificación* se refiere al conjunto de tareas que garantizan que el software implementa correctamente una función específica. La *validación* es un conjunto diferente de tareas que aseguran que el software que se construye sigue los requerimientos del cliente. Boehm [Boe81] afirma esto de esta forma:

> Verificación: "¿Construimos el producto correctamente?"
>
> Validación: "¿Construimos el producto correcto?"

La definición de V&V abarca muchas actividades de aseguramiento de calidad del software (capítulo 16).[1]

La verificación y la validación incluyen un amplio arreglo de actividades SQA: revisiones técnicas, auditorías de calidad y configuración, monitoreo de rendimiento, simulación, estudio de factibilidad, revisión de documentación, revisión de base de datos, análisis de algoritmos, pruebas de desarrollo, pruebas de usabilidad, pruebas de calificación, pruebas de aceptación y

1 Debe notarse que hay una fuerte divergencia de opinión acerca de qué tipos de pruebas constituyen la "validación". Algunas personas creen que *todas* las pruebas sirven para la verificación y que la validación se lleva a cabo cuando los requerimientos se revisan y aprueban, y, más tarde, por el usuario, cuando el sistema resulta operativo. Otras personas ven las pruebas de unidad y de integración (secciones 17.3.1 y 17.3.2) como verificación y las de orden superior (secciones 17.6 y 17.7) como validación.

pruebas de instalación. Aunque las pruebas juegan un papel extremadamente importante en V&V, también son necesarias muchas otras actividades.

CONSEJO

Es un error pensar que las pruebas son una "red de seguridad" que atrapará todos los errores que ocurran como producto de deficientes prácticas de ingeniería de software. No lo hará. Enfatice la calidad y la detección de errores a lo largo del proceso de software.

Las pruebas representan el último bastión desde donde puede valorarse la calidad y, de manera más pragmática, descubrirse errores. Pero las pruebas no deben verse como una red de seguridad. Como se dice: "no se puede probar la calidad. Si no está ahí antes de comenzar las pruebas, no estará cuando termine de probar". La calidad se incorpora en el software a lo largo de todo el proceso de ingeniería del software. La adecuada aplicación de métodos y herramientas, revisiones técnicas efectivas, y gestión y medición sólidas conducen a la calidad que se confirma durante las pruebas.

Miller [Mil77] relaciona la prueba del software con el aseguramiento de la calidad al afirmar que "la motivación subyacente de las pruebas de los programas es afirmar la claridad del software con métodos que puedan aplicarse de manera económica y efectiva a sistemas a gran y pequeña escala".

17.1.2 Organización de las pruebas del software

Cita:

"El optimismo es el riesgo ocupacional de la programación; la prueba es el tratamiento".

Kent Beck

En todo proyecto de software hay un conflicto inherente de intereses que ocurre conforme comienzan las pruebas. Hoy en día, a las personas que construyen el software se les pide probarlo. En sí, esto parece sencillo; después de todo, ¿quién conoce mejor el programa que sus desarrolladores? Por desgracia, estos mismos desarrolladores tienen mucho interés en demostrar que el programa está libre de errores, que funciona de acuerdo con los requerimientos del cliente y que se completará a tiempo y dentro del presupuesto. Cada uno de estos intereses tienen un efecto negativo sobre las pruebas más cuidadosas.

Desde un punto de vista psicológico, el análisis y diseño de software (junto con la codificación) son tareas constructivas. El ingeniero de software analiza, modela y luego crea un programa de computadora y su documentación. Como cualquier constructor, el ingeniero de software está orgulloso del edificio que construyó y ve con desconfianza a quien intente derrumbarlo. Cuando comienzan las pruebas, hay un sutil, pero definitivo, intento por "romper" lo que construyó el ingeniero de software. Desde el punto de vista del constructor, las pruebas pueden considerarse como (psicológicamente) destructivas. De modo que el constructor actuará con cuidado, y diseñará y ejecutará pruebas que demostrarán que el programa funciona, en lugar de descubrir errores. Desafortunadamente, los errores estarán presentes. Y si el ingeniero de software no los encuentra, ¡el cliente lo hará!

Con frecuencia, existen algunas malas interpretaciones que pueden inferirse de manera errónea a partir de la discusión anterior: 1) que el desarrollador de software no debe hacer pruebas en absoluto, 2) que el software debe "ponerse tras una pared" que lo separe de los extraños que lo probarán sin misericordia, 3) que quienes realicen las pruebas deben involucrarse con el proyecto sólo cuando los pasos de las pruebas estén por comenzar. Cada uno de estos enunciados es incorrecto.

El desarrollador de software siempre es responsable de probar las unidades individuales (componentes) del programa y de asegurarse de que cada una desempeña la función o muestra el comportamiento para el cual se diseñó. En muchos casos, el desarrollador también realiza pruebas de integración, una etapa en las pruebas que conduce a la construcción (y prueba) de la arquitectura completa del software. Sólo después de que la arquitectura de software está completa se involucra un grupo de prueba independiente (GPI).

PUNTO CLAVE

Un grupo de prueba independiente no tiene el "conflicto de intereses" que pueden experimentar los constructores del software.

El papel de un *grupo de prueba independiente* (GPI) es remover los problemas inherentes que están asociados con dejar al constructor probar lo que construyó. Las pruebas independientes remueven el conflicto de intereses que de otro modo puede estar presente. Después de todo, al personal del GPI se le paga por encontrar errores.

Sin embargo, el desarrollador no da el software al GPI y se retira. Él y el GPI trabajan de manera cercana a lo largo del proyecto de software para garantizar que se realizarán pruebas ex-

haustivas. Mientras se realizan éstas, el desarrollador debe estar disponible para corregir los errores que se descubran.

El GPI es parte del equipo de proyecto de desarrollo del software pues se involucra durante el análisis y el diseño, y sigue involucrado (mediante planificación y especificación de procedimientos de prueba) a lo largo de un proyecto grande. No obstante, en muchos casos, el GPI reporta a la organización de aseguramiento de calidad del software, y por tanto logra un grado de independencia que no puede existir si fuese parte de la organización de ingeniería del software.

Cita:

"El primer error que comete la gente es creer que el equipo de prueba es responsable de asegurar la calidad."

Brian Marick

17.1.3 Estrategia de prueba del software. Visión general

El proceso de software puede verse como la espiral que se ilustra en la figura 17.1. Inicialmente, la ingeniería de sistemas define el papel del software y conduce al análisis de los requerimientos del mismo, donde se establecen los criterios de dominio, función, comportamiento, desempeño, restricciones y validación de información para el software. Al avanzar hacia adentro a lo largo de la espiral, se llega al diseño y finalmente a la codificación. Para desarrollar software de computadoras, se avanza en espiral hacia adentro (contra las manecillas del reloj) a lo largo de una línea que reduce el nivel de abstracción en cada vuelta.

? ¿Cuál es la estrategia global para la prueba del software?

Una estrategia para probar el software también puede verse en el contexto de la espiral (figura 17.1). La *prueba de unidad* comienza en el vértice de la espiral y se concentra en cada unidad (por ejemplo, componente, clase o un objeto de contenido de una *webapp*) del software como se implementó en el código fuente. La prueba avanza al moverse hacia afuera a lo largo de la espiral, hacia la *prueba de integración*, donde el enfoque se centra en el diseño y la construcción de la arquitectura del software. Al dar otra vuelta hacia afuera de la espiral, se encuentra la *prueba de validación*, donde los requerimientos establecidos como parte de su modelado se validan confrontándose con el software que se construyó. Finalmente, se llega a la *prueba del sistema*, donde el software y otros elementos del sistema se prueban como un todo. Para probar el software de cómputo, se avanza en espiral hacia afuera en dirección de las manecillas del reloj a lo largo de líneas que ensanchan el alcance de las pruebas con cada vuelta.

WebRef

Quienes prueban software pueden encontrar recursos útiles en **www.SQAtester.com**

Al considerar el proceso desde un punto de vista procedural, las pruebas dentro del contexto de la ingeniería del software en realidad son una serie de cuatro pasos que se implementan de manera secuencial. Éstos se muestran en la figura 17.2. Inicialmente, las pruebas se enfocan en cada componente de manera individual, lo que garantiza que funcionan adecuadamente como unidad. De ahí el nombre de *prueba de unidad*. Esta prueba utiliza mucho de las técnicas de prueba que ejercitan rutas específicas en una estructura de control de componentes para asegurar una cobertura completa y la máxima detección de errores. A continuación, los componentes deben ensamblarse o integrarse para formar el paquete de software completo. La *prueba de integración* aborda los conflictos asociados con los problemas duales de verificación y construcción de programas. Durante la integración, se usan más las técnicas de diseño de casos de

FIGURA 17.1
Estrategia de pruebas

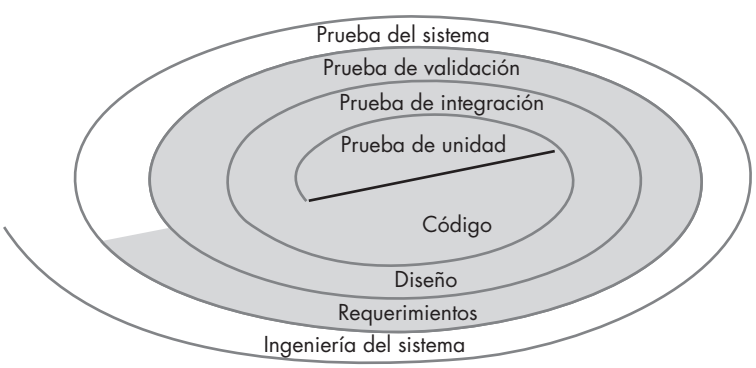
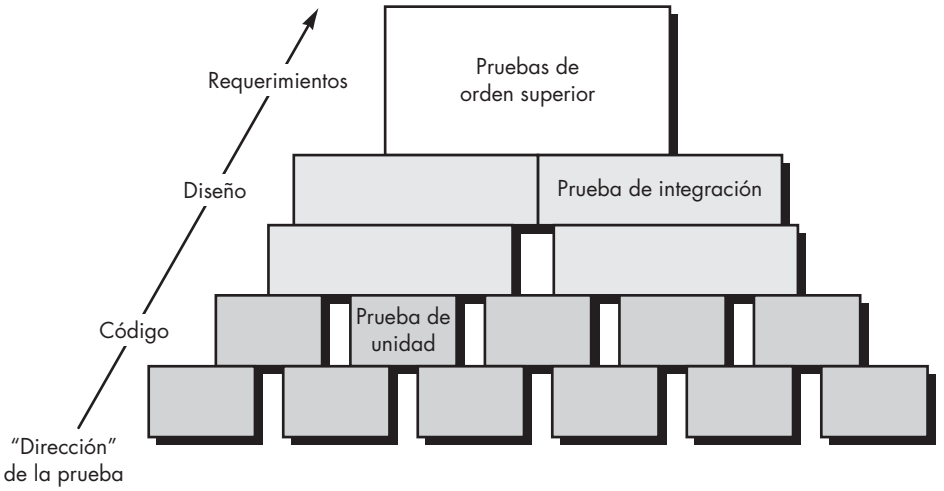


FIGURA 17.2
Pasos de la prueba del software



prueba que se enfocan en entradas y salidas, aunque también pueden usarse técnicas que ejercitan rutas de programa específicas para asegurar la cobertura de las principales rutas de control. Después de integrar (construir) el software, se realiza una serie de *pruebas de orden superior*. Deben evaluarse criterios de validación (establecidos durante el análisis de requerimientos). La *prueba de validación* proporciona la garantía final de que el software cumple con todos los requerimientos informativos, funcionales, de comportamiento y de rendimiento.

El último paso de la prueba de orden superior cae fuera de las fronteras de la ingeniería de software y en el contexto más amplio de la ingeniería de sistemas de cómputo. El software, una vez validado, debe combinarse con otros elementos del sistema (por ejemplo, hardware, personal, bases de datos). La *prueba del sistema* verifica que todos los elementos se mezclan de manera adecuada y que se logra el funcionamiento/rendimiento global del sistema.

CASASEGURA

Preparación para la prueba

La escena: Oficina de Doug Miller, mientras continúa el diseño en el nivel de componentes y comienza la construcción de algunos de ellos.

Participantes: Doug Miller, jefe de ingeniería de software; Vinod, Jamie, Ed y Shakira, miembros del equipo de ingeniería de software de CasaSegura.

La conversación:

Doug: Me parece que no hemos dedicado suficiente tiempo para hablar de las pruebas.

Vinod: Cierto, pero todos hemos estado un poco ocupados. Y además hemos pensado en ello... en realidad, hemos hecho más que pensar.

Doug (sonríe): Lo sé... tenemos exceso de trabajo, pero todavía debemos pensar en las cosas importantes.

Shakira: Me gusta la idea de diseñar pruebas de unidad antes de comenzar a codificar cualquiera de mis componentes, así que eso es lo que he intentado hacer. Tengo un archivo de pruebas bastante grande para aplicar cuando codifique mis componentes por completo.

Doug: Ése es un concepto de programación extrema [proceso de desarrollo de software ágil, véase el capítulo 3], ¿o no?

Ed: Lo es. Aun cuando no usamos programación extrema *per se*, decidimos que sería buena idea diseñar pruebas de unidad antes de construir el componente; el diseño nos dará la información que necesitamos.

Jamie: Yo he hecho lo mismo.

Vinod: Y yo tomé el papel de integrador, así que cada vez que uno de los muchachos me pase un componente, lo integraré y correré una serie de pruebas de regresión sobre el programa parcialmente integrado. He trabajado para diseñar un conjunto de pruebas adecuadas para cada función en el sistema.

Doug (a Vinod): ¿Con qué frecuencia corres las pruebas?

Vinod: Todos los días... hasta que el sistema esté integrado... bueno, quiero decir hasta que esté integrado el incremento de software que planeamos entregar.

Doug: ¡Muchachos, van adelante de mí!

Vinod (ríe): La anticipación lo es todo en el negocio del software, jefe.

17.1.4 Criterios para completar las pruebas

Cada vez que se analiza la prueba del software, surge una pregunta clásica: "¿cuándo terminan las pruebas?, ¿cómo se sabe que se ha probado lo suficiente?". Lamentablemente, no hay una respuesta definitiva a esta pregunta, pero existen algunas respuestas pragmáticas e intentos tempranos a manera de guía empírica.

? ¿Cuándo terminan las pruebas?

Una respuesta a la pregunta es: "nunca se termina de probar; la carga simplemente pasa de usted (el ingeniero de software) al usuario final". Cada vez que el usuario ejecuta un programa de cómputo, el programa se pone a prueba. Este instructivo hecho subraya la importancia de otras actividades a fin de garantizar la calidad del software. Otra respuesta (un tanto cínica, mas no obstante precisa) es: "las pruebas terminan cuando se agota el tiempo o el dinero".

Aunque algunos profesionales usarían estas respuestas, se necesitan criterios más rigurosos para determinar cuándo se han realizado suficientes pruebas. El enfoque de *ingeniería de software de salas limpias* (capítulo 21) sugiere el uso de técnicas estadísticas [Kel00] que ejecutan una serie de pruebas derivadas de una muestra estadística de todas las posibles ejecuciones de programa por parte de todos los usuarios de una población objetivo. Otros (por ejemplo, [Sin99]) abogan por el uso del modelado estadístico y la teoría de confiabilidad del software para predecir cuándo están completas las pruebas.

WebRef

Un glosario amplio de términos de pruebas puede encontrarse en el sitio **www.testingstandards.co.uk/living_glossary.htm**

Al coleccionar estadísticas durante las pruebas del software y usar los modelos existentes de confiabilidad del mismo, es posible desarrollar lineamientos significativos para responder la pregunta: "¿cuándo terminan las pruebas?". Hay poco debate acerca de que todavía queda mucho trabajo por hacer antes de poder establecer reglas cuantitativas para las pruebas, pero los acercamientos empíricos que existen en la actualidad son considerablemente mejores que la intuición pura.

17.2 Aspectos estratégicos

Más adelante en este capítulo, se presenta una estrategia sistemática para probar el software. Pero incluso la mejor estrategia fracasará si no se aborda una serie de aspectos decisivos. Tom Gilb [Gil95] arguye que una estrategia de software triunfará cuando quienes prueban el software:

? ¿Qué lineamientos conducen a una exitosa estrategia de prueba del software?

Especifican los requerimientos del producto en forma cuantificable mucho antes de comenzar con las pruebas. Aunque el objetivo predominante de una prueba es encontrar errores, una buena estrategia de prueba también valora otras características de la calidad, como la portabilidad, el mantenimiento y la facilidad de uso (capítulo 14). Esto debe especificarse en una forma medible, de modo que los resultados de las pruebas no sean ambiguos.

Establecen de manera explícita los objetivos de las pruebas. Los objetivos específicos de las pruebas deben enunciarse en términos medibles. Por ejemplo, la efectividad de las pruebas, su cobertura, el tiempo medio antes de aparecer una falla, el costo por descubrir y corregir defectos, la densidad de defectos restantes o la frecuencia de ocurrencia, y las horas de trabajo de prueba deben enunciarse dentro del plan de la prueba.

WebRef

Una excelente lista de recursos de prueba puede encontrarse en el sitio **www.io.com/~wazmo/qa**

Entienden a los usuarios del software y desarrollan un perfil para cada categoría de usuario. Los casos de uso que describen el escenario de interacción para cada clase de usuario pueden reducir el esfuerzo de prueba global al enfocar las pruebas en el uso real del producto.

Desarrollan un plan de prueba que enfatice "pruebas de ciclo rápido". Gilb [Gil95] recomienda que un equipo de software "aprenda a probar en ciclos rápidos (2 por ciento del esfuerzo del proyecto) de cliente-utilidad al menos la 'comprobabilidad' en campo, los incrementos de funcionalidad y/o la mejora de la calidad". La retroalimentación generada a partir de estas pruebas de ciclo rápido puede usarse para controlar niveles de calidad y las correspondientes estrategias de prueba.

Construyen software "robusto" que esté diseñado para probarse a sí mismo. El software debe diseñarse en forma que use técnicas antierrores (sección 17.3.1), es decir, el software debe poder diagnosticar ciertas clases de errores. Además, el diseño debe incluir pruebas automatizadas y pruebas de regresión.

Usan revisiones técnicas efectivas como filtro previo a las pruebas. Las revisiones técnicas (capítulo 15) pueden ser tan efectivas como probar para descubrir errores. Por esta razón, las revisiones pueden reducir la cantidad del esfuerzo de pruebas que se requieren para producir software de alta calidad.

Cita:

"Probar sólo los **requerimientos** del usuario final es como inspeccionar un edificio con base en el trabajo realizado por el decorador de interiores a costa de cimientos, vigas y plomería."

Boris Beizer

Realizan revisiones técnicas para valorar la estrategia de prueba y los casos de prueba. Las revisiones de prueba pueden descubrir inconsistencias, omisiones y errores evidentes en el abordaje de las pruebas. Esto ahorra tiempo y también mejora la calidad del producto.

Desarrollan un enfoque de mejora continuo para el proceso de prueba. La estrategia de pruebas debe medirse. Las métricas recopiladas durante las pruebas deben usarse como parte de un enfoque de control de proceso estadístico para la prueba del software.

17.3 Estrategias de prueba para software convencional[2]

Existen muchas estrategias que pueden usarse para probar el software. En un extremo, puede esperarse hasta que el sistema esté completamente construido y luego realizar las pruebas sobre el sistema total, con la esperanza de encontrar errores. Este enfoque, aunque atractivo, simplemente no funciona. Dará como resultado software defectuoso que desilusionará a todos los participantes. En el otro extremo, podrían realizarse pruebas diariamente, siempre que se construya alguna parte del sistema. Este enfoque, aunque menos atractivo para muchos, puede ser muy efectivo. Por desgracia, algunos desarrolladores de software son reacios a usarlo. ¿Qué hacer?

Una estrategia de prueba que eligen la mayoría de los equipos de software se coloca entre los dos extremos. Toma una visión incremental de las pruebas, comenzando con la de unidades de programa individuales, avanza hacia pruebas diseñadas para facilitar la integración de las unidades y culmina con pruebas que ejercitan el sistema construido. Cada una de estas clases de pruebas se describe en las secciones que siguen.

17.3.1 Prueba de unidad

La *prueba de unidad* enfoca los esfuerzos de verificación en la unidad más pequeña del diseño de software: el componente o módulo de software. Al usar la descripción del diseño de componente como guía, las rutas de control importantes se prueban para descubrir errores dentro de la frontera del módulo. La relativa complejidad de las pruebas y los errores que descubren están limitados por el ámbito restringido que se establece para la prueba de unidad. Las pruebas de unidad se enfocan en la lógica de procesamiento interno y de las estructuras de datos dentro de las fronteras de un componente. Este tipo de pruebas puede realizarse en paralelo para múltiples componentes.

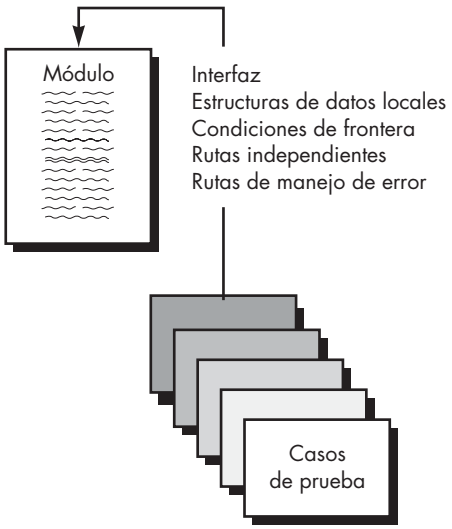
CONSEJO

No es mala idea diseñar casos de prueba de unidad antes de desarrollar el código para un componente. Eso ayuda a garantizar que se desarrollará un código que pasará las pruebas.

Consideraciones de las pruebas de unidad. Las pruebas de unidad se ilustran de manera esquemática en la figura 17.3. La interfaz del módulo se prueba para garantizar que la información fluya de manera adecuada hacia y desde la unidad de software que se está probando. Las estructuras de datos locales se examinan para asegurar que los datos almacenados temporal-

2 A lo largo de este libro, se usan los términos *software convencional* o *software tradicional* para referirse a arquitecturas de software jerárquica común, o de "llamar y regresar", que con frecuencia se encuentran en una variedad de dominios de aplicación. Las arquitecturas de software tradicional *no* son orientadas a objetos y no abarcan *webapps*.

FIGURA 17.3
Prueba de unidad



mente mantienen su integridad durante todos los pasos en la ejecución de un algoritmo. Todas las rutas independientes a través de la estructura de control se ejercitan para asegurar que todos los estatutos en un módulo se ejecuten al menos una vez. Las condiciones de frontera se prueban para asegurar que el módulo opera adecuadamente en las fronteras establecidas para limitar o restringir el procesamiento. Y, finalmente, se ponen a prueba todas las rutas para el manejo de errores.

El flujo de datos a través de la interfaz de un componente se prueba antes de iniciar cualquiera otra prueba. Si los datos no entran y salen de manera adecuada, todas las demás pruebas son irrelevantes. Además, deben ejercitarse las estructuras de datos locales y averiguarse (si es posible) el impacto local sobre los datos globales durante las pruebas de unidad.

? ¿Qué errores se encuentran comúnmente durante las pruebas de unidad?

La prueba selectiva de las rutas de ejecución es una tarea esencial durante la prueba de unidad. Los casos de prueba deben diseñarse para descubrir errores debidos a cálculos erróneos, comparaciones incorrectas o flujo de control inadecuado.

Las pruebas de frontera son una de las tareas de prueba de unidad más importantes. Con frecuencia, el software falla en sus fronteras. Es decir: con frecuencia los errores ocurren cuando se procesa el *en*ésimo elemento de un arreglo *ene*-dimensional, cuando se invoca la *en*ésima repetición de un bucle con *n* pasadas, cuando se encuentra el valor máximo o mínimo permisible. Es muy probable que los casos de prueba que ejercitan la estructura de datos, el flujo de control y los valores de datos justo abajo y arriba de máximos y mínimos descubran errores.

WebRef

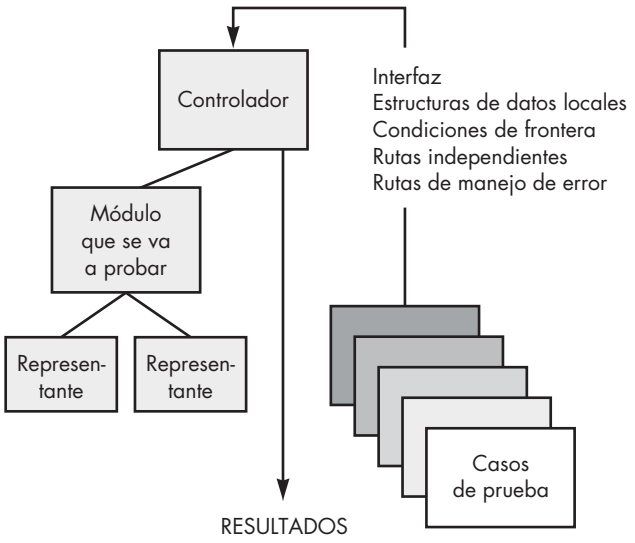
Puede encontrar información útil acerca de una gran variedad de artículos y recursos para "prueba ágil" en **testing.com/agile**

Un buen diseño anticipa las condiciones de error y establece rutas de manejo de errores para enrutar o terminar limpiamente el procesamiento cuando ocurre un error. Yourdon [You75] llama a este enfoque *antierrores*. Desafortunadamente, hay una tendencia a incorporar el manejo de errores en el software y luego nunca probarlo. Una historia verídica puede servir como ilustración:

> Un sistema de diseño asistido por computadora se desarrolló bajo contrato. En un módulo de procesamiento de transacción, un bromista colocó el siguiente mensaje de manejo de error después de una serie de pruebas condicionales que invocaban varias ramas de flujo de control: ¡ERROR! NO HAY FORMA DE QUE PUEDA LLEGAR AQUÍ. ¡Este "mensaje de error" lo descubrió un cliente durante el entrenamiento para usuarios!

Entre los potenciales errores que deben ponerse a prueba cuando se evalúa el manejo de errores están: 1) la descripción de error ininteligible, 2) el error indicado no corresponde con el error que se encuentra, 3) la condición del error causa la intervención del sistema antes de manejar el

FIGURA 17.4
Entorno de prueba de unidad



Asegúrese de diseñar pruebas para ejecutar cada ruta de manejo de error. Si no lo hace, la ruta puede fallar cuando se invoque, lo que agrava una situación de por sí peligrosa.

error, 4) el procesamiento excepción-condición es incorrecto y 5) la descripción del error no proporciona suficiente información para auxiliar en la localización de la causa del error.

Procedimientos de prueba de unidad. Las pruebas de unidad por lo general se consideran como adjuntas al paso de codificación. El diseño de las pruebas de unidad puede ocurrir antes de comenzar la codificación o después de generar el código fuente. La revisión de la información del diseño proporciona una guía para establecer casos de prueba que es probable que descubran errores en cada una de las categorías analizadas anteriormente. Cada caso de prueba debe acoplarse con un conjunto de resultados esperados.

Puesto que un componente no es un programa independiente, con frecuencia debe desarrollarse software controlador y/o de resguardo para cada prueba de unidad. En la figura 17.4 se ilustran los entornos de prueba de unidad. En la mayoría de las aplicaciones, un *controlador* no es más que un "programa principal" que acepta datos de caso de prueba, pasa tales datos al componente (que va a ponerse a prueba) e imprime resultados relevantes. Los *representantes* (en inglés *stubs)* sirven para sustituir módulos que están subordinados al (invocados por el) componente que se va a probar. Un representante o "subprograma tonto" usa la interfaz de módulo subordinado, puede realizar mínima manipulación de datos, imprimir verificación de entradas y regresar el control al módulo sobre el que se realiza la prueba.

Existen algunas situaciones donde no se tienen los recursos para realizar una prueba de unidad amplia. Seleccione los módulos cruciales o complejos y aplique sólo en ellos las pruebas de unidad.

Los controladores y representantes añaden una "sobrecarga" a las pruebas. Es decir: ambos son software que debe escribirse (el diseño formal usualmente no se aplica), pero que no se entrega con el producto de software final. Si los controladores y representantes se mantienen simples, la sobrecarga real es relativamente baja. Por desgracia, muchos componentes no pueden tener prueba de unidad adecuada con un software de sobrecarga simple. En tales casos, la prueba completa puede posponerse hasta el paso de prueba de integración (donde también se usan controladores o representantes).

Las pruebas de unidad se simplifican cuando se diseña un componente con alta cohesión. Cuando un componente aborda una sola función, el número de casos de prueba se reduce y los errores pueden predecirse y descubrirse con mayor facilidad.

17.3.2 Pruebas de integración

Un neófito en el mundo del software podrá plantear una pregunta aparentemente legítima una vez que todos los módulos se hayan probado de manera individual: "si todos ellos funcionan

individualmente, ¿por qué dudan que funcionarán cuando se junten todos?". Desde luego, el problema es "juntarlos todos": conectarlos. Los datos pueden perderse a través de una interfaz; un componente puede tener un inadvertido efecto adverso sobre otro; las subfunciones, cuando se combinan, pueden no producir la función principal deseada; la imprecisión aceptable individualmente puede magnificarse a niveles inaceptables; las estructuras de datos globales pueden presentar problemas. Lamentablemente, la lista sigue y sigue.

Las pruebas de integración son una técnica sistemática para construir la arquitectura del software mientras se llevan a cabo pruebas para descubrir errores asociados con la interfaz. El objetivo es tomar los componentes probados de manera individual y construir una estructura de programa que se haya dictado por diseño.

Tomar el enfoque de "big bang" para la integración es una estrategia perezosa condenada al fracaso. Integre de manera incremental y pruebe conforme avance.

Con frecuencia existe una tendencia a intentar la integración no incremental, es decir, a construir el programa usando un enfoque de *big bang*. Todos los componentes se combinan por adelantado. Todo el programa se prueba como un todo. ¡Y usualmente resulta el caos! Se descubre un conjunto de errores. La corrección se dificulta pues el aislamiento de las causas se complica por la vasta extensión de todo el programa. Una vez corregidos estos errores, otros nuevos aparecen y el proceso continúa en un bucle aparentemente interminable.

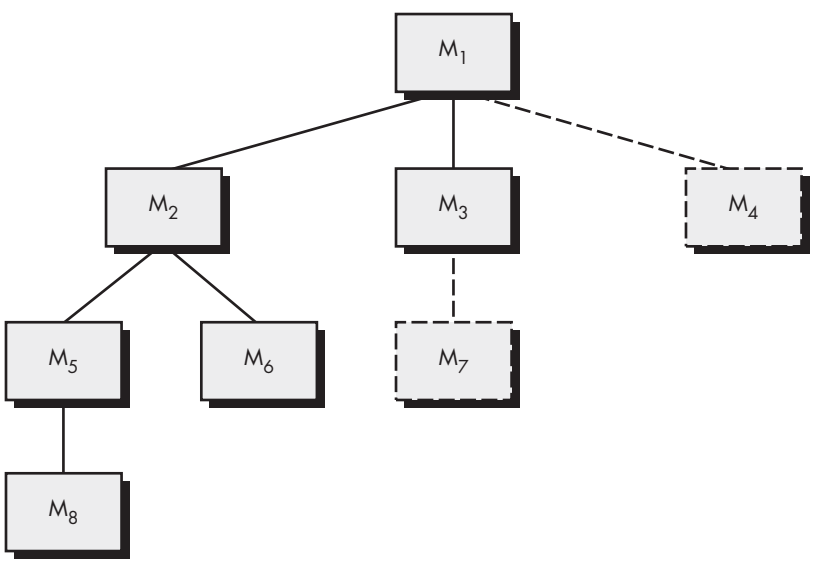
La integración incremental es la antítesis del enfoque *big bang*. El programa se construye y prueba en pequeños incrementos, donde los errores son más fáciles de aislar y corregir; las interfaces tienen más posibilidades de probarse por completo; y puede aplicarse un enfoque de prueba sistemático. En los siguientes párrafos se exponen algunas estrategias diferentes de integración incremental.

Cuando desarrolle un calendario de proyecto, considere la forma en la que ocurrirá la integración, de modo que los componentes estén disponibles cuando se les necesite.

Integración descendente. La *prueba de integración descendente* es un enfoque incremental a la construcción de la arquitectura de software. Los módulos se integran al moverse hacia abajo a través de la jerarquía de control, comenzando con el módulo de control principal (programa principal). Los módulos subordinados al módulo de control principal se incorporan en la estructura en una forma de primero en profundidad o primero en anchura.

Con referencia a la figura 17.5, la *integración primero en profundidad* integra todos los componentes sobre una ruta de control mayor de la estructura del programa. La selección de una ruta mayor es un tanto arbitraria y depende de las características específicas de la aplicación. Por ejemplo, al seleccionar la ruta de la izquierda, los componentes M_1, M_2, M_5 se integrarían primero. A continuación, M_8 o (si es necesario para el adecuado funcionamiento de M_2) se inte-

FIGURA 17.5
Integración descendente



graría M_6. Luego se construyen las rutas de control central y derecha. La *integración primero en anchura* incorpora todos los componentes directamente subordinados en cada nivel, y se mueve horizontalmente a través de la estructura. De la figura, los componentes M_2, M_3 y M_4 se integrarían primero. Le sigue el siguiente nivel de control, M_5, M_6, etc. El proceso de integración se realiza en una serie de cinco pasos:

¿Cuáles son los pasos para la integración descendente?

1. El módulo de control principal se usa como un controlador de prueba y los representantes (*stubs*) se sustituyen con todos los componentes directamente subordinados al módulo de control principal.
2. Dependiendo del enfoque de integración seleccionado (es decir, primero en profundidad o anchura), los representantes subordinados se sustituyen uno a la vez con componentes reales.
3. Las pruebas se llevan a cabo conforme se integra cada componente.
4. Al completar cada conjunto de pruebas, otro representante se sustituye con el componente real.
5. Las pruebas de regresión (que se analizan más adelante en esta sección) pueden realizarse para asegurar que no se introdujeron nuevos errores.

El proceso continúa desde el paso 2 hasta que se construye todo la estructura del programa.

La estrategia de integración descendente verifica los principales puntos de control o de decisión al principio en el proceso de prueba. En una estructura de programa "bien factorizada", la toma de decisiones ocurre en niveles superiores en la jerarquía y, por tanto, se encuentra primero. Si existen grandes problemas de control, el reconocimiento temprano es esencial. Si se selecciona la integración primero en profundidad, es posible implementar y demostrar un funcionamiento completo del software. La demostración temprana de la capacidad funcional es un constructor de confianza para todos los participantes.

¿Qué problemas pueden encontrarse cuando se elige la integración descendente?

Pareciera que la estrategia descendente no tiene complicaciones, pero, en la práctica, pueden surgir problemas logísticos. El más común de éstos ocurre cuando se requiere procesamiento en niveles bajos en la jerarquía a fin de probar de manera adecuada los niveles superiores. Los representantes (*stubs*) sustituyen los módulos de bajo nivel al comienzo de la prueba descendente; por tanto, ningún dato significativo puede fluir hacia arriba en la estructura del programa. A la persona que realiza la prueba le quedan tres opciones: 1) demorar muchas pruebas hasta que los representantes se sustituyan con módulos reales, 2) desarrollar resguardos que realicen funciones limitadas que simulen al módulo real o 3) integrar el software desde el fondo de la jerarquía y hacia arriba.

El primer enfoque (demorar las pruebas hasta que los representantes se sustituyan con módulos reales) puede hacerle perder algo de control sobre la correspondencia entre pruebas específicas y la incorporación de módulos específicos. Esto puede conducir a dificultades para determinar la causa de los errores y tiende a violar la naturaleza enormemente restrictiva del enfoque descendente. El segundo enfoque vale la pena, pero puede conducir a una sobrecarga significativa conforme los representantes se vuelven cada vez más complejos. El tercero, llamado integración ascendente, se analiza en los siguientes párrafos.

Integración ascendente. La *prueba de integración ascendente*, como su nombre implica, comienza la construcción y la prueba con *módulos atómicos* (es decir, componentes en los niveles inferiores dentro de la estructura del programa). Puesto que los componentes se integran de abajo hacia arriba, la funcionalidad que proporcionan los componentes subordinados en determinado nivel siempre está disponible y se elimina la necesidad de representantes (*stubs*). Una estrategia de integración ascendente puede implementarse con los siguientes pasos:

¿Cuáles son los pasos para la integración ascendente?

1. Los componentes en el nivel inferior se combinan en grupos (en ocasiones llamados *construcciones* o *builds)* que realizan una subfunción de software específica.
2. Se escribe un *controlador* (un programa de control para pruebas) a fin de coordinar la entrada y salida de casos de prueba.
3. Se prueba el grupo.
4. Los controladores se remueven y los grupos se combinan moviéndolos hacia arriba en la estructura del programa.

PUNTO CLAVE

La integración ascendente elimina la necesidad de **representantes (*stubs*)** complejos.

La integración sigue el patrón que se ilustra en la figura 17.6. Los componentes se combinan para formar los grupos 1, 2 y 3. Cada uno de ellos se prueba usando un controlador (que se muestra como un bloque rayado). Los componentes en los grupos 1 y 2 se subordinan a M_a. Los controladores D_1 y D_2 se remueven y los grupos se ponen en interfaz directamente con M_a. De igual modo, el controlador D_3 para el grupo 3 se remueve antes de la integración con el módulo M_b. Tanto M_a como M_b al final se integrarán con el componente M_c, y así sucesivamente.

Conforme la integración avanza hacia arriba, se reduce la necesidad de controladores de prueba separados. De hecho, si los dos niveles superiores del programa se integran de manera descendente, el número de controladores puede reducirse de manera sustancial y la integración de grupos se simplifica enormemente.

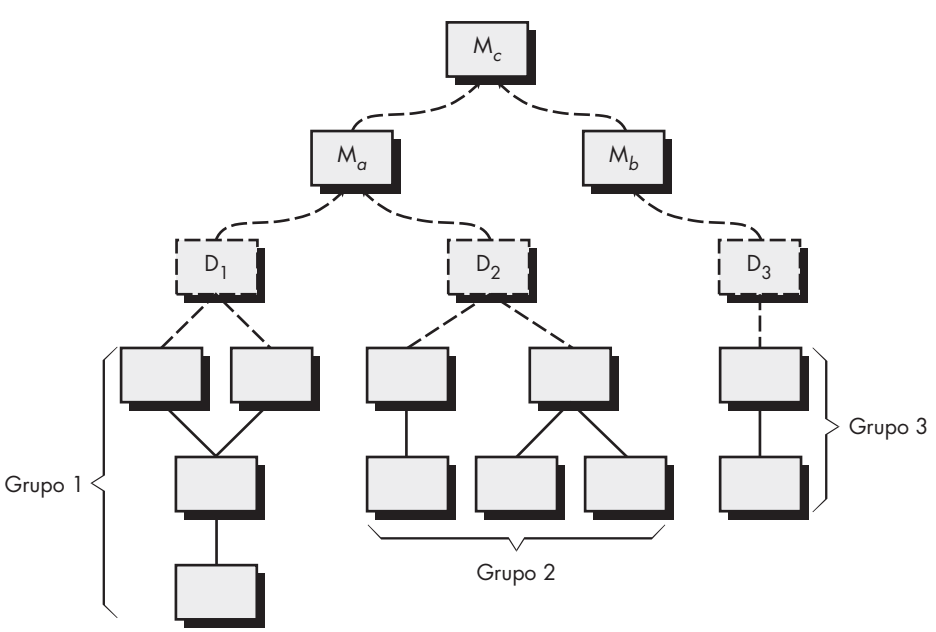
CONSEJO

La prueba de regresión es una importante estrategia para reducir "efectos colaterales". Corra pruebas de regresión cada vez que se realiza un cambio importante al software (incluida la integración de nuevos componentes).

Prueba de regresión. Cada vez que se agrega un nuevo módulo como parte de las pruebas de integración, el software cambia. Se establecen nuevas rutas de flujo de datos, ocurren nuevas operaciones de entrada/salida y se invoca nueva lógica de control. Dichos cambios pueden causar problemas con las funciones que anteriormente trabajaban sin fallas. En el contexto de una estrategia de prueba de integración, la *prueba de regresión* es la nueva ejecución de algún subconjunto de pruebas que ya se realizaron a fin de asegurar que los cambios no propagaron efectos colaterales no deseados.

En un contexto más amplio, las pruebas exitosas (de cualquier tipo) dan como resultado el descubrimiento de errores, y los errores deben corregirse. Siempre que se corrige el software, cambia algún aspecto de la configuración del software (el programa, su documentación o los datos que sustenta). Las pruebas de regresión ayudan a garantizar que los cambios (debidos

FIGURA 17.6
Integración ascendente



a pruebas o por otras razones) no introducen comportamiento no planeado o errores adicionales.

Las pruebas de regresión se pueden realizar manualmente, al volver a ejecutar un subconjunto de todos los casos de prueba o usando herramientas de captura/reproducción automatizadas. Las *herramientas de captura/reproducción* permiten al ingeniero de software capturar casos de prueba y resultados para una posterior reproducción y comparación. La *suite de prueba de regresión* (el subconjunto de pruebas que se va a ejecutar) contiene tres clases diferentes de casos de prueba:

- Una muestra representativa de pruebas que ejercitará todas las funciones de software.
- Pruebas adicionales que se enfocan en las funciones del software que probablemente resulten afectadas por el cambio.
- Pruebas que se enfocan en los componentes del software que cambiaron.

Conforme avanza la prueba de integración, el número de pruebas de regresión puede volverse muy grande. Por tanto, la suite de pruebas de regresión debe diseñarse para incluir solamente aquellas que aborden una o más clases de errores en cada una de las funciones del programa principal. Es impráctico e ineficiente volver a ejecutar toda prueba para cada función del programa cada vez que ocurre un cambio.

Prueba de humo. La *prueba de humo* es un enfoque de prueba de integración que se usa cuando se desarrolla software de producto. Se diseña como un mecanismo de ritmo para proyectos críticos en el tiempo, lo que permite al equipo del software valorar el proyecto de manera frecuente. En esencia, el enfoque de prueba de humo abarca las siguientes actividades:

PUNTO CLAVE

La prueba de humo puede caracterizarse como una estrategia de integración constante. El software se reconstruye (con el agregado de nuevos componentes) y se prueba cada día.

1. Los componentes de software traducidos en código se integran en una *construcción*. Una construcción incluye todos los archivos de datos, bibliotecas, módulos reutilizables y componentes sometidos a ingeniería que se requieren para implementar una o más funciones del producto.
2. Se diseña una serie de pruebas para exponer los errores que evitarán a la construcción realizar adecuadamente su función. La intención debe ser descubrir errores "paralizantes" que tengan la mayor probabilidad de retrasar el proyecto.
3. La construcción se integra con otras construcciones, y todo el producto (en su forma actual) se somete a prueba de humo diariamente. El enfoque de integración puede ser descendente o ascendente.

Cita:

"Trate a la construcción diaria como al latido del proyecto. Si no hay latido, el proyecto está muerto."

Jim McCarthy

La frecuencia diaria de las pruebas de todo el producto puede sorprender a algunos lectores. Sin embargo, las pruebas constantes brindan, tanto a gerentes como a profesionales, una valoración realista del progreso de la prueba de integración. McConnell [McC96] describe la prueba de humo de la forma siguiente:

> La prueba de humo debe ejercitar todo el sistema de extremo a extremo. No tiene que ser exhaustiva, pero debe poder exponer los problemas principales. La prueba de humo debe ser suficientemente profunda para que, si la construcción pasa, pueda suponer que es suficientemente estable para probarse con mayor profundidad.

La prueba de humo proporciona algunos beneficios cuando se aplica sobre proyectos de software complejos y cruciales en el tiempo:

? ¿Qué beneficios pueden derivarse de las pruebas de humo?

- *Se minimiza el riesgo de integración*. Puesto que las pruebas de humo se realizan diariamente, las incompatibilidades y otros errores paralizantes pueden descubrirse tempranamente, lo que reduce la probabilidad de impacto severo sobre el calendario cuando se descubren errores.

- *La calidad del producto final mejora.* Es probable que la prueba de humo descubra errores funcionales así como errores de diseño arquitectónico o en el componente debido a que el enfoque está orientado a la construcción (integración). Si tales errores se corrigen temprano, se tendrá una mejor calidad del producto.
- *El diagnóstico y la corrección de errores se simplifican.* Como todo enfoque de prueba de integración, es probable que los errores descubiertos durante la prueba de humo se asocien con "nuevos incrementos de software"; es decir, el software que se acaba de agregar a la(s) construcción(es) es causa probable de un error recientemente descubierto.
- *El progreso es más fácil de valorar.* Con cada día que transcurre, más software se integra y se demuestra que funciona. Esto incrementa la moral del equipo y brinda a los gerentes un buen indicio de que se está progresando.

WebRef

En **www.qalinks.com** pueden encontrarse enlaces a comentarios acerca de estrategias de pruebas.

Opciones estratégicas. Ha habido mucha discusión (por ejemplo, [Bei84]) acerca de las relativas ventajas y desventajas de las pruebas de integración descendente en comparación con las ascendentes. En general, las ventajas de una estrategia tienden a ser desventajas para la otra. La principal desventaja del enfoque descendente es la necesidad de representantes y las dificultades de prueba que pueden asociarse con ellos. Los problemas asociados con los representantes pueden compensarse con la ventaja de probar tempranamente las principales funciones de control. La principal desventaja de la integración ascendente es que "el programa como entidad no existe hasta que se agrega el último módulo" [Mye79]. Este inconveniente se atempera con la mayor facilidad en el diseño de casos de prueba y la falta de representantes.

La selección de una estrategia de integración depende de las características del software y, en ocasiones, del calendario del proyecto. En general, un enfoque combinado (a veces llamado *prueba sándwich*), que usa pruebas descendentes para niveles superiores de la estructura del programa acopladas con pruebas ascendentes para niveles subordinados, puede ser el mejor arreglo.

? ¿Qué es un "módulo crítico" y por qué debe identificársele?

Conforme se realiza la integración, quien efectúa la prueba debe identificar los módulos críticos. Un *módulo crítico* tiene una o más de las siguientes características: 1) aborda muchos requerimientos de software, 2) tiene un alto nivel de control (reside relativamente alto en la estructura del programa), 3) es complejo o proclive al error o 4) tiene requerimientos de rendimiento definidos. Los módulos críticos deben probarse tan pronto como sea posible. Además, las pruebas de regresión deben enfocarse en la función del módulo crítico.

Productos de trabajo de las pruebas de integración. Un plan global para integración del software y una descripción de las pruebas específicas se documentan en una *Especificación de pruebas.* Este producto de trabajo incorpora un plan de prueba y un procedimiento de prueba, y se vuelve parte de la configuración del software. La prueba se divide en fases y construcciones que abordan características del software funcionales y de comportamiento específicas. Por ejemplo, la prueba de integración para el sistema de seguridad *CasaSegura* puede dividirse en las siguientes fases de prueba:

- *Interacción con el usuario* (entrada y salida de comandos, representación de despliegue, procesamiento y representación de errores)
- *Procesamiento de sensores* (adquisición de salida de sensor, determinación de condiciones del sensor, acciones requeridas como consecuencia de las condiciones)
- *Funciones de comunicación* (capacidad para comunicarse con la estación de monitoreo central)
- *Procesamiento de alarma* (pruebas de acciones del software que ocurren cuando se encuentra una alarma)

Cada una de estas fases de la prueba de integración delinea una amplia categoría funcional dentro del software y por lo general puede relacionarse con un dominio específico dentro de la arquitectura del software. Por tanto, las construcciones de programas (grupos de módulos) se crean para corresponder a cada fase. Los siguientes criterios y pruebas correspondientes se aplican a todas las fases de prueba:

? ¿Qué criterios deben usarse para diseñar pruebas de integración?

Integridad de interfaz. Las interfaces internas y externas se prueban conforme cada módulo (o grupo) se incorpora en la estructura.

Validez funcional. Se realizan pruebas diseñadas para descubrir errores funcionales ocultos.

Contenido de la información. Se realizan pruebas diseñadas para descubrir errores ocultos asociados con las estructuras de datos locales o globales.

Rendimiento. Se realizan pruebas diseñadas para verificar los límites del rendimiento establecidos durante el diseño del software.

Como parte del plan de prueba, también se discute un calendario para la integración, el desarrollo de software de sobrecarga del sistema y temas relacionados. Se establecen las fechas de inicio y fin de cada fase y se definen "ventanas disponibles" para módulos de prueba de unidad. Una breve descripción del software de sobrecarga (representantes y controladores) se concentra en las características que pueden requerir de un esfuerzo especial. Finalmente, se describe el entorno y los recursos de la prueba. Configuraciones inusuales de hardware, simuladores peculiares y herramientas o técnicas de prueba especial son algunos de los muchos temas que también pueden analizarse.

A continuación se describe el procedimiento de prueba detallado que se requiere para lograr el plan de prueba. Se señala el orden de la integración y las pruebas correspondientes en cada paso de ésta. También se incluye una lista de todos los casos de prueba (anotados para referencia posterior) y los resultados esperados.

En un *Reporte de prueba*, que puede anexarse a la *Especificación pruebas* si se desea, se registra una historia de resultados, problemas o peculiaridades de prueba reales. La información contenida en esta sección puede ser vital durante el mantenimiento del software. También se presentan las referencias y apéndices apropiados.

Como todos los demás elementos de una configuración de software, el formato de la especificación pruebas puede adaptarse a las necesidades locales de una organización de ingeniería de software. Sin embargo, es importante señalar que una estrategia de integración (contenida en un plan de prueba) y los detalles de la prueba (descritos en un procedimiento de prueba) son ingredientes esenciales y deben aparecer.

17.4 Estrategias de prueba para software orientado a objeto[3]

Enunciado de manera simple, el objetivo de probar es encontrar el mayor número posible de errores con una cantidad manejable de esfuerzo aplicado durante un lapso realista. Aunque este objetivo fundamental se mantiene invariable para el software orientado a objeto, la naturaleza de este software cambia tanto la estrategia como las tácticas de la prueba (capítulo 19).

17.4.1 Prueba de unidad en el contexto OO

Cuando se considera software orientado a objeto, el concepto de unidad cambia. La encapsulación determina la definición de clases y objetos. Esto significa que cada clase y cada instancia de una clase empaqueta los atributos (datos) y las operaciones que manipulan estos datos. Por lo

3 En el apéndice 2 se presentan conceptos básicos orientados a objeto.

general, una clase encapsulada es el foco de la prueba de unidad. No obstante, las operaciones (métodos) dentro de la clase son las unidades comprobables más pequeñas. Puesto que una clase puede contener algunas operaciones diferentes, y una operación particular puede existir como parte de algunas clases diferentes, las tácticas aplicadas a la prueba de unidad deben cambiar.

Ya no es posible probar una sola operación en aislamiento (la visión convencional de la prueba de unidad) sino más bien como parte de una clase. Para ilustrarlo, considere una jerarquía de clase en la que una operación *X* se define para la superclase y la heredan algunas subclases. Cada subclase usa la operación *X*, pero se aplica dentro del contexto de los atributos y operaciones privados que se definieron para la subclase. Dado que el contexto en el que se usa la operación *X* varía sutilmente, es necesario probar la operación *X* en el contexto de cada una de las subclases. Esto significa que por lo general no es efectivo probar la operación *X* en forma aislada (el enfoque de prueba de unidad convencional) en el contexto orientado a objeto.

PUNTO CLAVE

La prueba de clase para software OO es análoga a la prueba de módulo para software convencional. No es aconsejable probar operaciones en aislamiento.

La prueba de clase para software OO es el equivalente de la prueba de unidad para software convencional. A diferencia de la prueba de unidad del software convencional, que tiende a enfocarse sobre el detalle algorítmico de un módulo y en los datos que fluyen a través de la interfaz de módulo, la prueba de clase para software OO la dirigen las operaciones encapsuladas por la clase y el comportamiento de estado de ésta.

17.4.2 Prueba de integración en el contexto OO

Puesto que el software orientado a objeto no tiene una estructura de control jerárquico obvia, las estrategias tradicionales descendente y ascendente (sección 17.3.2) tienen poco significado. Además, con frecuencia es imposible integrar las operaciones una a la vez en una clase (el enfoque de integración incremental convencional) debido a las "interacciones directa e indirecta de los componentes que constituyen la clase" [Ber93].

Existen dos estrategias diferentes para la prueba de integración de los sistemas OO [Bin94b]. La primera, la *prueba basada en hebra*, integra el conjunto de clases requeridas para responder a una entrada o evento para el sistema. Cada hebra se integra y prueba de manera individual. La prueba de regresión se aplica para asegurar que no ocurran efectos colaterales. El segundo enfoque de integración, la *prueba basada en uso*, comienza la construcción del sistema al probar dichas clases (llamadas *clases independientes*) que usan muy pocas clases *servidor* (si es que usan alguna). Después de probar las clases independientes, se prueba la siguiente capa de clases, llamadas *dependientes*, que usan las clases independientes. Esta secuencia de probar capas de clases dependientes continúa hasta que se construye todo el sistema.

PUNTO CLAVE

Una importante estrategia para la prueba de integración del software OO es la prueba basada en hebra. Las hebras son conjuntos de clases que responden a una entrada o evento. Las pruebas basadas en uso se enfocan en clases que no colaboran fuertemente con otras clases.

El uso de controladores y representantes también cambia cuando se realiza la prueba de integración de los sistemas OO. Los controladores pueden usarse para probar operaciones en el nivel más bajo, y para la prueba de todos los grupos de clases. También puede usarse un controlador para sustituir la interfaz de usuario, de modo que las pruebas de funcionalidad del sistema puedan realizarse antes de la implementación de la interfaz. Los representantes (*stubs*) pueden usarse en situaciones donde se requiere la colaboración entre clases pero donde una o más de las clases colaboradoras todavía no se implementan por completo.

La *prueba de grupo* es un paso en la prueba de integración del software OO. Aquí, un grupo de clases colaboradoras (determinadas al examinar el CRC y el modelo objeto relacional) se ejercita al diseñar casos de prueba que intentan descubrir errores en las colaboraciones.

17.5 Estrategias de prueba para *webapps*

La estrategia para probar *webapps* adopta los principios básicos para todas las pruebas de software y aplica una estrategia y tácticas que se usan para sistemas orientados a objetos. Los siguientes pasos resumen el enfoque:

PUNTO CLAVE

La estrategia global para probar *webapps* puede resumirse en los 10 pasos que se anotan aquí.

1. El modelo de contenido para la *webapp* se revisa para descubrir errores.
2. El modelo de interfaz se revisa para garantizar que todos los casos de uso pueden adecuarse.
3. El modelo de diseño para la *webapp* se revisa para descubrir errores de navegación.
4. La interfaz de usuario se prueba para descubrir errores en los mecanismos de presentación y/o navegación.
5. A cada componente funcional se le aplica una prueba de unidad.
6. Se prueba la navegación a lo largo de toda la arquitectura.
7. La *webapp* se implementa en varias configuraciones ambientales diferentes y se prueba en su compatibilidad con cada configuración.
8. Las pruebas de seguridad se realizan con la intención de explotar vulnerabilidades en la *webapp* o dentro de su ambiente.
9. Se realizan pruebas de rendimiento.
10. La *webapp* se prueba mediante una población de usuarios finales controlada y monitoreada. Los resultados de su interacción con el sistema se evalúan por errores de contenido y navegación, preocupaciones de facilidad de uso, preocupaciones de compatibilidad, así como confiabilidad y rendimiento de la *webapp*.

WebRef

En **www.stickyminds.com/testing.asp** pueden encontrarse excelentes artículos acerca de pruebas de las *webapps*.

Puesto que muchas *webapps* evolucionan continuamente, el proceso de prueba es una actividad siempre en marcha, y se realiza para apoyar al personal que usa pruebas de regresión derivadas de las pruebas desarrolladas cuando se elaboró por primera vez la *webapp*. En el capítulo 20 se consideran métodos para probar la *webapp*.

17.6 PRUEBAS DE VALIDACIÓN

Las pruebas de validación comienzan en la culminación de las pruebas de integración, cuando se ejercitaron componentes individuales, el software está completamente ensamblado como un paquete y los errores de interfaz se descubrieron y corrigieron. En el nivel de validación o de sistema, desaparece la distinción entre software convencional, software orientado a objetos y *webapps*. Las pruebas se enfocan en las acciones visibles para el usuario y las salidas del sistema reconocibles por el usuario.

PUNTO CLAVE

Como todos los demás pasos de las pruebas, la validación intenta descubrir errores, pero el **enfoque se orienta** en **los requerimientos**: sobre las cosas que serán inmediatamente aparentes **para** el usuario final.

La validación puede definirse en muchas formas, pero una definición simple (aunque dura) es que la validación es exitosa cuando el software funciona en una forma que cumpla con las expectativas razonables del cliente. En este punto, un desarrollador de software curtido en la batalla puede protestar: "¿quién o qué es el árbitro de las expectativas razonables?". Si se desarrolló una *Especificación de requerimientos de software*, en ella se describen todos los atributos del software visibles para el usuario; contiene una sección de *Criterios de validación* que forman la base para un enfoque de pruebas de validación.

17.6.1 Criterios de pruebas de validación

La validación del software se logra a través de una serie de pruebas que demuestran conformidad con los requerimientos. Un plan de prueba subraya las clases de pruebas que se van a realizar y un procedimiento de prueba define casos de prueba específicos que se diseñan para garantizar que: se satisfacen todos los requerimientos de funcionamiento, se logran todas las características de comportamiento, todo el contenido es preciso y se presenta de manera adecuada, se logran todos los requerimientos de rendimiento, la documentación es correcta y se

satisfacen la facilidad de uso y otros requerimientos (por ejemplo, transportabilidad, compatibilidad, recuperación de error, mantenimiento).

Después de realizar cada caso de prueba de validación, existen dos posibles condiciones: 1) La característica de función o rendimiento se conforma de acuerdo con las especificaciones y se acepta, o 2) se descubre una desviación de la especificación y se crea una lista de deficiencias. Las desviaciones o errores descubiertos en esta etapa en un proyecto rara vez pueden corregirse antes de la entrega calendarizada. Con frecuencia es necesario negociar con el cliente para establecer un método para resolver deficiencias.

17.6.2 Revisión de la configuración

Un elemento importante del proceso de validación es una *revisión de la configuración*. La intención de la revisión es garantizar que todos los elementos de la configuración del software se desarrollaron de manera adecuada, y que se cataloga y se tiene el detalle necesario para reforzar las actividades de apoyo. La revisión de la configuración, en ocasiones llamada auditoría, se estudia con más detalle en el capítulo 22.

17.6.3 Pruebas alfa y beta

Cita:

"Teniendo los suficientes ojos, todos los errores son superficiales (por ejemplo, con una base suficientemente grande de personas que realizan pruebas beta y codesarrolladores, casi todo problema se caracterizará rápidamente y la corrección será obvia para alguien)."

E. Raymond

Virtualmente, es imposible que un desarrollador de software prevea cómo usará el cliente realmente un programa. Las instrucciones para usarlo pueden malinterpretarse; regularmente pueden usarse combinaciones extrañas de datos; la salida que parecía clara a quien realizó la prueba puede ser ininteligible para un usuario.

Cuando se construye software a la medida para un cliente, se realiza una serie de pruebas de aceptación a fin de permitir al cliente validar todos los requerimientos. Realizada por el usuario final en lugar de por los ingenieros de software, una prueba de aceptación puede variar desde una "prueba de conducción" informal hasta una serie de pruebas planificadas y ejecutadas sistemáticamente. De hecho, la prueba de aceptación puede realizarse durante un periodo de semanas o meses, y mediante ella descubrir errores acumulados que con el tiempo puedan degradar el sistema.

Si el software se desarrolla como un producto que va a ser usado por muchos clientes, no es práctico realizar pruebas de aceptación formales con cada uno de ellos. La mayoría de los constructores de productos de software usan un proceso llamado prueba alfa y prueba beta para descubrir errores que al parecer sólo el usuario final es capaz de encontrar.

¿Cuál es la diferencia entre una prueba alfa y una prueba beta?

La *prueba alfa* se lleva a cabo en el sitio del desarrollador por un grupo representativo de usuarios finales. El software se usa en un escenario natural con el desarrollador "mirando sobre el hombro" de los usuarios y registrando los errores y problemas de uso. Las pruebas alfa se realizan en un ambiente controlado.

La *prueba beta* se realiza en uno o más sitios del usuario final. A diferencia de la prueba alfa, por lo general el desarrollador no está presente. Por tanto, la prueba beta es una aplicación "en vivo" del software en un ambiente que no puede controlar el desarrollador. El cliente registra todos los problemas (reales o imaginarios) que se encuentran durante la prueba beta y los reporta al desarrollador periódicamente. Como resultado de los problemas reportados durante las pruebas beta, es posible hacer modificaciones y luego preparar la liberación del producto de software a toda la base de clientes.

En ocasiones se realiza una variación de la prueba beta, llamada *prueba de aceptación del cliente*, cuando el software se entrega a un cliente bajo contrato. El cliente realiza una serie de pruebas específicas con la intención de descubrir errores antes de aceptar el software del desarrollador. En algunos casos (por ejemplo, un gran corporativo o sistema gubernamental) la prueba de aceptación puede ser muy formal y abarcar muchos días o incluso semanas de prueba.

CasaSegura

Preparación para la validación

La escena: Oficina de Doug Miller, mientras continúan tanto el diseño a nivel de componentes como la construcción de ciertos componentes.

Participantes: Doug Miller, jefe de ingeniería del software, Vinod, Jamie, Ed y Shakira, miembros del equipo de ingeniería del software *CasaSegura*.

La conversación:

Doug: El primer incremento estará listo para validación en... ¿cuánto tiempo? ¿Tres semanas?

Vinod: Más o menos. La integración va bien. Hacemos pruebas de humo todos los días y encontramos algunos *bugs*, pero nada que no podamos manejar. Hasta el momento va bien.

Doug: Háblame de la validación.

Shakira: Bueno, para el diseño de prueba, usaremos todos los casos de uso como base. Todavía no empiezo, pero desarrollaré pruebas para todos los casos de uso de las que sea responsable.

Ed: Igual yo.

Jamie: Yo también, pero debemos actuar juntos para la prueba de aceptación y también para las pruebas alfa y beta, ¿o no?

Doug: Sí. De hecho lo he pensado; podríamos traer un contratista externo para ayudarnos con la validación. Tengo dinero en el presupuesto... y él nos daría un nuevo punto de vista.

Vinod: Creo que lo tenemos bajo control.

Doug: Estoy seguro que sí, pero un GPI nos da un vistazo independiente del software.

Jamie: Estamos apretados de tiempo, Doug. Yo no tengo tiempo para vigilar a alguien que traigan para hacer el trabajo.

Doug: Lo sé, lo sé. Pero si un GPI funciona a partir de los requerimientos y los casos de uso, no necesitará demasiada vigilancia.

Vinod: Yo todavía creo que lo tenemos bajo control.

Doug: Te escuché, Vinod, pero voy a sostener mi opinión en esta ocasión. Más tarde planearemos la reunión con el representante del GPI para esta semana. Dejemos que comiencen y veamos lo que proponen.

Vinod: Muy bien, tal vez eso aligere un poco la carga.

17.7 Pruebas del sistema

Cita:

"Como la muerte y los impuestos, las pruebas son desagradables e inevitables".

Ed Yourdon

Al comienzo de este libro, se resaltó el hecho de que el software sólo es un elemento de un sistema basado en computadora más grande. A final de cuentas, el software se incorpora con otros elementos del sistema (por ejemplo, hardware, personas, información), y se lleva a cabo una serie de pruebas de integración y validación del sistema. Estas pruebas quedan fuera del ámbito del proceso de software y no se llevan a cabo exclusivamente por parte de ingenieros de software. Sin embargo, los pasos que se toman durante el diseño y la prueba del software pueden mejorar enormemente la probabilidad de integración exitosa del software en el sistema más grande.

Un problema clásico en la prueba del sistema es el "dedo acusador". Esto ocurre cuando se descubre un error y los desarrolladores de diferentes elementos del sistema se culpan unos a otros por el problema. En lugar de abandonarse a tal sinsentido, deben anticiparse los potenciales problemas de interfaz y: 1) diseñar rutas de manejo de error que prueben toda la información proveniente de otros elementos del sistema, 2) realizar una serie de pruebas que simulen los datos malos u otros errores potenciales en la interfaz del software, 3) registrar los resultados de las pruebas para usar como "evidencia" si ocurre el dedo acusador, y 4) participar en planificación y diseño de pruebas del sistema para garantizar que el software se prueba de manera adecuada.

En realidad, la *prueba del sistema* es una serie de diferentes pruebas cuyo propósito principal es ejercitar por completo el sistema basado en computadora. Aunque cada prueba tenga un propósito diferente, todo él funciona para verificar que los elementos del sistema se hayan integrado de manera adecuada y que se realicen las funciones asignadas. En las secciones que siguen se estudian los tipos de pruebas del sistema que valen la pena para los sistemas basados en software.

17.7.1 Pruebas de recuperación

Muchos sistemas basados en computadora deben recuperarse de fallas y reanudar el procesamiento con poco o ningún tiempo de inactividad. En algunos casos, un sistema debe ser tole-

rante a las fallas, es decir, las fallas del procesamiento no deben causar el cese del funcionamiento del sistema global. En otros casos, la falla de un sistema debe corregirse dentro de un periodo de tiempo específico u ocurrirán severos daños económicos.

La *recuperación* es una prueba del sistema que fuerza al software a fallar en varias formas y que verifica que la recuperación se realice de manera adecuada. Si la recuperación es automática (realizada por el sistema en sí), se evalúa el reinicio, los mecanismos de puntos de verificación, la recuperación de datos y la reanudación para correcciones. Si la recuperación requiere intervención humana, se evalúa el tiempo medio de reparación (TMR) para determinar si está dentro de límites aceptables.

17.7.2 Pruebas de seguridad

Cualquier sistema basado en computadora que gestione información sensible o cause acciones que puedan dañar (o beneficiar) de manera inadecuada a individuos es un blanco de penetración inadecuada o ilegal. La penetración abarca un amplio rango de actividades: *hackers* que intentan penetrar en los sistemas por deporte, empleados resentidos que intentan penetrar por venganza, individuos deshonestos que intentan penetrar para obtener ganancia personal ilícita.

La *prueba de seguridad* intenta verificar que los mecanismos de protección que se construyen en un sistema en realidad lo protegerán de cualquier penetración impropia. Para citar a Beizar [Bei84]: "La seguridad del sistema debe, desde luego, probarse para ser invulnerable ante ataques frontales; pero también debe probarse su invulnerabilidad contra ataques laterales y traseros."

Durante la prueba de seguridad, quien realiza la prueba juega el papel del individuo que desea penetrar al sistema. ¡Cualquier cosa vale! Quien realice la prueba puede intentar adquirir contraseñas por medios administrativos externos; puede atacar el sistema con software a la medida diseñado para romper cualquier defensa que se haya construido; puede abrumar al sistema, y por tanto negar el servicio a los demás; puede causar a propósito errores del sistema con la esperanza de penetrar durante la recuperación; puede navegar a través de datos inseguros para encontrar la llave de la entrada al sistema.

Con los suficientes tiempo y recursos, las buenas pruebas de seguridad a final de cuentas penetran en el sistema. El papel del diseñador de sistemas es hacer que el costo de la penetración sea mayor que el valor de la información que se obtendrá.

17.7.3 Pruebas de esfuerzo

Los primeros pasos de la prueba del software dieron como resultado una evaluación extensa de las funciones y el rendimiento normales del programa. Las pruebas de esfuerzo se diseñan para enfrentar los programas con situaciones anormales. En esencia, la persona que realiza las pruebas de esfuerzo pregunta: "¿cuánto podemos doblar esto antes de que se rompa?".

Cita:

"Si intenta encontrar verdaderos errores del sistema y no sujeta su software a una verdadera prueba de esfuerzo, entonces es el momento de comenzar."

Boris Beizer

La *prueba de esfuerzo* ejecuta un sistema en forma que demanda recursos en cantidad, frecuencia o volumen anormales. Por ejemplo, pueden 1) diseñarse pruebas especiales que generen diez interrupciones por segundo, cuando una o dos es la tasa promedio, (2) aumentarse las tasas de entrada de datos en un orden de magnitud para determinar cómo responderán las funciones de entrada, 3) ejecutarse casos de prueba que requieran memoria máxima y otros recursos, 4) diseñarse casos de prueba que puedan causar *thrashing* (que es un quebranto del sistema por hiperpaginación) en un sistema operativo virtual, 5) crearse casos de prueba que puedan causar búsqueda excesiva por datos residentes en disco. En esencia, la persona que realiza la prueba intenta romper el programa.

Una variación de la prueba de esfuerzo es una técnica llamada *prueba de sensibilidad*. En algunas situaciones (la más común ocurre en algoritmos matemáticos), un rango muy pequeño

de datos contenidos dentro de las fronteras de los datos válidos para un programa pueden causar procesamiento extremo, e incluso erróneo, o profunda degradación del rendimiento. La prueba de sensibilidad intenta descubrir combinaciones de datos dentro de clases de entrada válidas que puedan causar inestabilidad o procesamiento inadecuado.

17.7.4 Pruebas de rendimiento

Para sistemas en tiempo real y sistemas embebidos , el software que proporcione la función requerida, pero que no se adecue a los requerimientos de rendimiento, es inaceptable. La prueba de rendimiento se diseña para poner a prueba el rendimiento del software en tiempo de corrida, dentro del contexto de un sistema integrado. La prueba del rendimiento ocurre a lo largo de todos los pasos del proceso de prueba. Incluso en el nivel de unidad, puede accederse al rendimiento de un módulo individual conforme se realizan las pruebas. Sin embargo, no es sino hasta que todos los elementos del sistema están plenamente integrados cuando puede determinarse el verdadero rendimiento de un sistema.

Las pruebas de rendimiento con frecuencia se aparean con las pruebas de esfuerzo y por lo general requieren instrumentación de hardware y de software, es decir, con frecuencia es necesario medir la utilización de los recursos (por ejemplo, ciclos del procesador) en forma meticulosa. La instrumentación externa puede monitorear intervalos de ejecución y eventos de registro (por ejemplo, interrupciones) conforme ocurren, y los muestreos del estado de la máquina de manera regular. Con la instrumentación de un sistema, la persona que realiza la prueba puede descubrir situaciones que conduzcan a degradación y posibles fallas del sistema.

17.7.5 Pruebas de despliegue

En muchos casos, el software debe ejecutarse en varias plataformas y bajo más de un entorno de sistema operativo. La *prueba de despliegue*, en ocasiones llamada *prueba de configuración*, ejercita el software en cada entorno en el que debe operar. Además, examina todos los proce-

HERRAMIENTAS DE SOFTWARE

Planeación y administración de pruebas

Objetivo: Estas herramientas ayudan al equipo de software a planificar la estrategia de pruebas que se elija y a administrar el proceso de prueba mientras se lleva a cabo.

Mecánica: Las herramientas en esta categoría abordan la planificación de las pruebas, el almacenamiento, administración y control de las mismas; el seguimiento de los requisitos, la integración, el rastreo de errores y la generación de reportes. Los gestores de proyecto los usan para complementar las herramientas calendarizadas del proyecto. Quienes realizan las pruebas usan estas herramientas para planear actividades de prueba y controlar el flujo de información conforme avanza el proceso de pruebas.

Herramientas representativas:[4]

QaTraq Test Case Management Tool, desarrollada por Traq Software (**www.testmanagement.com**), "alienta un enfoque estructurado de la gestión de pruebas".

QADirector, desarrollada por Compuware Corp. (**www.compuware.com/qacenter**), proporciona un solo punto de control para gestionar todas las fases del proceso de pruebas.

TestWorks, desarrollada por Software Research, Inc. (**www.soft.com/Products/index.html**), contiene una suite completamente integrada de herramientas de prueba, incluidas herramientas para administración y reporte de pruebas.

OpensourceTesting.org (**www.opensourcetesting.org/testmgt.php**), cita varias herramientas de gestión y planificación de pruebas en fuente abierta.

NI TestStand, desarrollada por National Instruments Corp. (**www.ni.com**), le permite "desarrollar, gestionar y ejecutar secuencias de pruebas escritas en cualquier lenguaje de programación".

4 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

dimientos de instalación y el software de instalación especializado (por ejemplo, "instaladores") que usarán los clientes, así como toda la documentación que se usará para introducir el software a los usuarios finales.

Como ejemplo, piense en la versión accesible a internet del software *CasaSegura* que permitiría a un cliente monitorear el sistema de seguridad desde ubicaciones remotas. La *webapp* de *CasaSegura* debe probarse usando todos los navegadores web que es probable que se encuentren. Una prueba de despliegue más profunda puede abarcar combinaciones de navegadores web con varios sistemas operativos (por ejemplo, Linux, Mac OS, Windows). Puesto que la seguridad es un tema principal, un juego completo de pruebas de seguridad se integraría con la prueba de despliegue.

17.8 El arte de la depuración

Cita:

"Para nuestra sorpresa, descubrimos que no fue tan fácil obtener programas justo como los habíamos pensado. Recuerdo el instante exacto en el que me di cuenta de que una gran parte de mi vida, a partir de entonces, la iba a pasar descubriendo los errores en mis propios programas".

Maurice Wilkes, descubre la depuración, 1949

La prueba del software es un proceso que puede planearse y especificarse de manera sistemática. Puede realizarse el diseño de casos de prueba, definir una estrategia y evaluar los resultados comparándolos con las expectativas prescritas.

La *depuración* ocurre como consecuencia de las pruebas exitosas. Es decir, cuando un caso de prueba descubre un error, la depuración es el proceso que da como resultado la remoción del error. Aunque la depuración puede y debe ser un proceso ordenado, todavía en mucho es un arte. Los ingenieros de software con frecuencia se enfrentan con indicios "sintomáticos" de un problema de software mientras evalúan los resultados de una prueba, es decir, la manifestación externa del error y su causa interna pueden no tener relación obvia una con otra. El proceso mental, pobremente comprendido, que conecta un síntoma con una causa se conoce como depuración.

17.8.1 El proceso de depuración

La depuración no es una prueba, pero con frecuencia ocurre como consecuencia de una prueba.[5] De acuerdo con la figura 17.7, el proceso de depuración comienza con la ejecución de un caso de prueba. Los resultados se valoran y se encuentra la falta de correspondencia entre el rendimiento esperado y el real. En muchos casos, la no correspondencia de los datos es un síntoma de una causa subyacente y escondida. El proceso de depuración intenta relacionar síntoma con causa, lo que por tanto conduce a la corrección del error.

CONSEJO

Asegúrese de evitar un tercer resultado: se encuentra la causa, pero la "corrección" no resuelve el problema o incluso introduce otro error.

Por lo general, El proceso de depuración dará como resultado que: 1) la causa se encontrará y corregirá o 2) la causa no se encontrará. En el último caso, la persona que realiza la depuración puede sospechar una causa, diseñar un caso de prueba para auxiliarse en la validación de dicha suposición y trabajar hacia la corrección del error en forma iterativa.

¿Por qué es tan difícil la depuración?

¿Por qué es tan difícil la depuración? Con toda probabilidad, la psicología humana (vea la sección 17.8.2) tiene más que ver con la respuesta que la tecnología del software. Sin embargo, ciertas características de los errores brindan algunas pistas:

1. El síntoma y la causa pueden ser geográficamente remotos. Es decir, el síntoma puede aparecer en una parte de un programa, mientras que la causa en realidad puede ubicarse en un sitio que esté alejado. Los componentes altamente acoplados (capítulo 8) exacerban esta situación.
2. El síntoma puede desaparecer (temporalmente) cuando se corrige otro error.

5 Al hacer esta afirmación, se toma la visión más amplia posible de las pruebas. No sólo el desarrollador prueba el software previo a su liberación, ¡sino que el cliente/usuario prueba el software cada vez que lo usa!

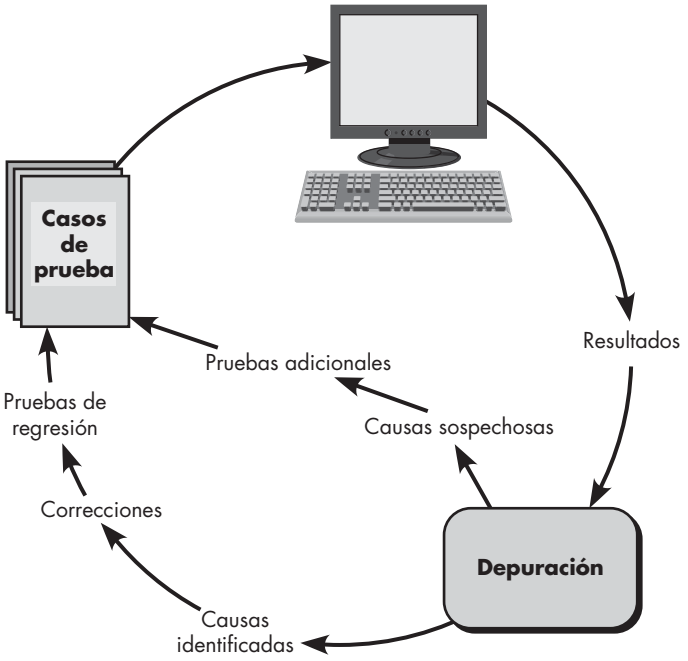


FIGURA 17.7
El proceso de depuración

3. El síntoma en realidad puede no ser causado por errores (por ejemplo, imprecisiones de redondeo).
4. El síntoma puede ser causado por un error humano que no se rastrea con facilidad.
5. El síntoma puede ser resultado de problemas de temporización más que de problemas de procesamiento.
6. Puede ser difícil reproducir con precisión las condiciones de entrada (por ejemplo, una aplicación en tiempo real en la que el orden de la entrada esté indeterminado).
7. El síntoma puede ser intermitente, particularmente común en sistemas embebidos que acoplan hardware y software de manera inextricable.
8. El síntoma puede deberse a causas que se distribuyen a través de algunas tareas que corren en diferentes procesadores.

? "Todo mundo sabe que la depuración es el doble de difícil que escribir un programa por primera vez. De modo que, si se es tan inteligente como se puede ser cuando se escribe el programa, ¿cómo es que se depurará?".

Brian Kernighan

Durante la depuración, encontrará errores que varían desde los ligeramente desconcertantes (por ejemplo, un formato de salida incorrecto) hasta los catastróficos (por ejemplo, la falla del sistema, que provoca serio daño económico o físico). Conforme aumentan las consecuencias de un error, también aumenta la cantidad de presión por encontrar la causa. Con frecuencia, la presión fuerza a algunos de los desarrolladores del software a corregir un error y, al mismo tiempo, introducir dos más.

17.8.2 Consideraciones psicológicas

Por desgracia, parece haber cierta evidencia de que la hazaña de la depuración es un rasgo humano innato. Algunas personas son buenas en ello y otras no lo son. Aunque la evidencia experimental de la depuración está abierta a muchas interpretaciones, se reportan grandes variaciones en la habilidad depuradora para programadores con la misma educación y experiencia. Al comentar acerca de los aspectos humanos de la depuración, Shneiderman [Shn80] afirma:

> La depuración es una de las partes más frustrantes de la programación. Tiene elementos de resolución de problemas o rompecabezas, junto con el desconcertante reconocimiento de que se cometió un error. La elevada ansiedad y la falta de voluntad para aceptar la posibilidad de los errores aumentan la dificultad de la tarea. Por fortuna, hay un gran alivio y la tensión se aligera cuando finalmente el error... se corrige.

Aunque puede ser difícil "aprender" a depurar, es posible proponer algunos enfoques al problema. Revise la sección 17.8.3.

CasaSegura

Depuración

La escena: Cubículo de Ed mientras se realiza la codificación y la prueba de unidad.

Participantes: Ed y Shakira, miembros del equipo de ingeniería de software *CasaSegura*.

La conversación:

Shakira (observa a través de la entrada del cubículo): Hola... ¿dónde estuviste a la hora del almuerzo?

Ed: Aquí... trabajando.

Shakira: Te ves horrible... ¿cuál es el problema?

Ed (suspira): He estado trabajando en este... error desde que lo descubrí a las 9:30 esta mañana y es, ¿qué?, 2:45... No tengo ni idea.

Shakira: Creí que todos estuvimos de acuerdo en no pasar más de una hora en tareas de depuración por cuenta propia; luego pediríamos ayuda, ¿verdad?

Ed: Sí, pero...

Shakira (entra al cubículo): ¿Así que cuál es el problema?

Ed: Es complicado y, además, lo he visto durante, ¿cuánto?, 5 horas. No lo vas a ver en 5 minutos.

Shakira: Permíteme... ¿cuál es el problema?

[Ed explica el problema a Shakira, quien lo observa durante 30 segundos sin hablar, luego...]

Shakira (se asoma una sonrisa en su cara): Oh, justo ahí, la variable llamada *setAlarmCondition*. ¿No debería ponerse en "falso" antes de comenzar el bucle?

[Ed mira la pantalla con incredulidad, se dobla hacia adelante y comienza a golpear su cabeza suavemente contra el monitor. Shakira, quien ahora sonríe abiertamente, se incorpora y sale del cubículo].

17.8.3 Estrategias de depuración

Establezca un límite, por decir, dos horas, en la cantidad de tiempo que empleará al intentar depurar un problema por cuenta propia. Después de eso, ¡pida ayuda!

Sin importar el enfoque que se tome, la depuración tiene un objetivo dominante: encontrar y corregir la causa de un error o defecto de software. El objetivo se realiza mediante una combinación de evaluación sistemática, intuición y suerte. Bradley [Bra85] describe el enfoque de depuración de la siguiente forma:

> La depuración es una aplicación directa del método científico que se ha desarrollado durante más de 2 500 años. La base de la depuración es localizar la fuente del problema [la causa] mediante una partición binaria, a través del trabajo con hipótesis que predicen nuevos valores por examinar.
>
> Tome un ejemplo simple que no sea de software: una lámpara en mi casa no funciona. Si nada en la casa funciona, la causa debe estar en el interruptor principal o en el exterior; observo mi alrededor para ver si el vecindario está a oscuras. Conecto la lámpara sospechosa en un tomacorriente que funcione y un electrodoméstico operativo en el circuito sospechoso. Y así continúo la alternación entre hipótesis y pruebas.

En general, se han propuesto tres estrategias de depuración [Mye79]: 1) fuerza bruta, 2) vuelta atrás (del inglés *backtracking*) y 3) eliminación de causas. Cada una de estas estrategias puede llevarse a cabo de manera manual, pero modernas herramientas de depuración pueden hacer el proceso mucho más efectivo.

Tácticas de depuración. La categoría *fuerza bruta* de la depuración probablemente es el método más común y menos eficiente para aislar la causa de un error de software. Los métodos

Cita:

"El primer paso para reparar un programa descompuesto es hacerlo fallar repetidamente (en el ejemplo más simple posible)."

T. Duff

de depuración de fuerza bruta se aplican cuando todo lo demás falla. Al usar una filosofía de "deje que la computadora encuentre el error", se toman copias de la memoria (*dumps*), se invocan rastreos en el tiempo de corrida y el programa se carga con enunciados de salida. La esperanza es que, en alguna parte del pantano de información que se produzca, se encontrará una pista que pueda conducir a la causa de un error. Aunque la masa de información producida a final de cuentas puede conducir al éxito, con más frecuencia conduce a desperdicio de esfuerzo y tiempo. ¡Piense que primero debe gastarse!

El *seguimiento hacia atrás* o vuelta atrás es un enfoque de depuración bastante común que puede usarse exitosamente en programas pequeños. Al comenzar en el sitio donde se descubrió un síntoma, el código fuente se rastrea hacia atrás (de manera manual) hasta que se encuentra la causa. Por desgracia, conforme aumenta el número de líneas fuente, el número de rutas potenciales hacia atrás puede volverse inmanejable.

El tercer enfoque de la depuración, la *eliminación de la causa*, se manifiesta mediante inducción o deducción, e introduce el concepto de partición binaria. Los datos relacionados con la ocurrencia del error se organizan para aislar las causas potenciales. Se plantea una "hipótesis de causa" y los datos anteriormente mencionados se usan para probar o refutar la hipótesis. De manera alternativa, se desarrolla una lista de las posibles causas y se realizan pruebas para eliminar cada una. Si las pruebas iniciales indican que una hipótesis de causa particular se muestra prometedora, los datos se refinan con la intención de aislar el error.

Depuración automatizada. Cada uno de estos enfoques de depuración puede complementarse con herramientas de depuración que puedan proporcionar apoyo semiautomático conforme se intenten estrategias de depuración. Hailpern y Santhanam [Hai02] resumen el estado de estas herramientas cuando apuntan: "... se han propuesto muchos nuevos enfoques y están disponibles muchos entornos de depuración comerciales. Los entornos de desarrollo integrados (IDE) brindan una forma de capturar algunos de los errores predeterminados específicos del lenguaje (por ejemplo, falta de caracteres de fin de sentencia , variables indefinidas, etc.) sin requerir compilación". Se dispone de una gran variedad de compiladores de depuración, ayudas dinámicas de depuración ("trazadores"), generadores automáticos de casos de prueba y herramientas de mapeo de referencia cruzada. Sin embargo, las herramientas no son un sustituto

HERRAMIENTAS DE SOFTWARE

Depuración

Objetivo: Estas herramientas proporcionan asistencia automatizada para quienes deben depurar problemas de software. La intención es proporcionar conocimiento que puede ser difícil de obtener si se aborda el proceso de depuración de forma manual.

Mecánica: La mayoría de las herramientas de depuración son específicas del lenguaje de programación y del entorno.

Herramientas representativas:[6]

Borland Gauntlet, distribuido por Borland (**www.borland. com**), auxilia tanto en las pruebas como en la depuración.

Coverty Prevent SQS, desarrollada por Coverty (**www.coverty. com**), proporciona asistencia de depuración tanto para C++ como para Java.

C++Test, desarrollada por Parasoft (**www.parasoft.com**), es una herramienta de prueba de unidad que soporta un rango completo de pruebas en código C y C++. Las características de depuración ayudan en el diagnóstico de errores que se encuentren.

CodeMedic, desarrollada por NewPlanet Software (**www. newplanetsfotware.com/medic/**), proporciona una interfaz gráfica para el depurador estándar UNIX, *gdb*, e implementa sus características más importantes. En la actualidad, *gdb* soporta C/C++, Java, PalmOS, varios sistemas incrustados, lenguaje ensamblador, FORTRAN y Modula-2.

GNATS, una aplicación freeware (**www.gnu.org/software/ gnats**), es un conjunto de herramientas para rastrear reportes de error.

6 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

para la evaluación cuidadosa basada en un modelo completo de diseño y en código fuente claro.

El factor humano. Cualquier discusión de los enfoques y herramientas de depuración está incompleta sin mencionar un poderoso aliado: ¡otras personas! Un punto de vista fresco, no empañado por horas de frustración, puede hacer maravillas.[7] Una máxima final para la depuración puede ser: "Cuando todo lo demás falle, ¡consiga ayuda!"

17.8.4 Corrección del error

Una vez encontrado el error, debe corregirse. Pero, como ya se señaló, la corrección de un error puede introducir otros errores y, por tanto, hacer más daño que bien. Van Vleck [Van89] sugiere tres preguntas simples que deben plantearse antes de hacer la "corrección" que remueva la causa de un error:

Cita:

"El mejor examinador no es aquel que encuentra más errores... el mejor es quien consigue la corrección de más errores."

Cem Kaner *et al.*

1. *¿La causa del error se reproduce en otra parte del programa?* En muchas situaciones, un defecto de programa es causado por un patrón de lógica erróneo que puede reproducirse en alguna otra parte. La consideración explícita del patrón lógico puede resultar en el descubrimiento de otros errores.
2. *¿Qué "siguiente error" puede introducirse con la corrección que está a punto de realizar?* Antes de hacer la corrección, debe evaluarse el código fuente (o, mejor, el diseño) para valorar el acoplamiento de las estructuras lógica y de datos. Si la corrección se realizará en una sección altamente acoplada del programa, debe tenerse especial cuidado cuando se realice algún cambio.
3. *¿Qué debió hacerse para evitar este error desde el principio?* Esta pregunta es el primer paso hacia el establecimiento de un enfoque de aseguramiento de calidad estadística del software (capítulo 16). Si se corrigen tanto el proceso como el producto, el error se removerá del programa actual y podrá eliminarse de todos los programas futuros.

17.9 Resumen

Las pruebas de software representan el porcentaje más grande de esfuerzo técnico en el proceso de software. Sin importar el tipo de software que se construya, una estrategia para planificar, ejecutar y controlar pruebas sistemáticas comienza por considerar pequeños elementos del software y moverse hacia afuera, hacia el programa como un todo.

El objetivo de las pruebas del software es descubrir errores. Para software convencional, este objetivo se logra mediante una serie de pasos de prueba. Las pruebas de unidad e integración se concentran en la verificación funcional de un componente y en la incorporación de componentes en una arquitectura de software. Las pruebas de validación demuestran la conformidad con los requerimientos del software y las pruebas del sistema validan el software una vez que se incorporó en un sistema más grande. Cada paso de la prueba se logra a través de una serie de técnicas de prueba sistemáticas que auxilian en el diseño de casos de prueba. Con cada paso de prueba, se amplía el nivel de abstracción con la que se considera el software.

La estrategia para probar software orientado a objeto comienza con pruebas que ejercitan las operaciones dentro de una clase y luego avanzan hacia la prueba basada en hebra para integración. Las hebras son conjuntos de clases que responden a una entrada o evento. Las pruebas basadas en uso se enfocan en clases que no colaboran demasiado con otras clases.

---

7 El concepto de programación por parejas (recomendado como parte del modelo de programación extrema que se estudió en el capítulo 3), proporciona un mecanismo de "depuración" conforme se diseña y codifica el software.

Las *webapps* se prueban en forma muy parecida a los sistemas OO. Sin embargo, las pruebas se diseñan para ejercitar contenido, funcionalidad, interfaz, navegación y aspectos de rendimiento y seguridad de la *webapp*.

A diferencia de las pruebas (una actividad sistemática planificada), la depuración puede verse como un arte. Al comenzar con una indicación sintomática de un problema, la actividad de depuración debe rastrear la causa de un error. De los muchos recursos disponibles durante la depuración, el más valioso es el consejo de otros miembros del equipo de ingeniería del software.

Problemas y puntos por evaluar

17.1. Con sus palabras, describa la diferencia entre verificación y validación. ¿Ambas usan los métodos de diseño de casos de prueba y estrategias de pruebas?

17.2. Mencione algunos problemas que pueden asociarse con la creación de un grupo de prueba independiente. ¿Los GPI y el SQA se integran con las mismas personas?

17.3. ¿Siempre es posible desarrollar una estrategia para probar software que usa la secuencia de pasos de prueba descritos en la sección 17.1.3? ¿Qué posibles complicaciones pueden surgir para sistemas incrustados?

17.4. ¿Por qué un módulo altamente acoplado es difícil para la prueba de unidad?

17.5. El concepto de "antierrores" (sección 17.3.1) es una forma extremadamente efectiva de brindar asistencia de depuración interna cuando se descubre un error:

a) Desarrolle un conjunto de lineamientos para antierror.
b) Analice las ventajas de usar la técnica.
c) Analice las desventajas.

17.6. ¿Cómo puede la calendarización del proyecto afectar la prueba de integración?

17.7. ¿La prueba de unidad es posible o incluso deseable en todas las circunstancias? Proporcione ejemplos para justificar su respuesta.

17.8. ¿Quién debe realizar la prueba de validación: el desarrollador o el usuario del software? Justifique su respuesta.

17.9. Desarrolle una estrategia de prueba completa para el sistema *CasaSegura* que se estudió anteriormente en este libro. Documéntela en una *Especificación de pruebas*.

17.10. Como proyecto de clase, desarrolle una *Guía de depuración* para su instalación. ¡La guía debe brindar lenguaje y sugerencias orientadas a sistemas aprendidos en la escuela de la vida! Comience por destacar los temas que revisarán la clase y el instructor. Publique la guía para otros en su entorno local.

Lecturas y fuentes de información adicionales

Virtualmente todo libro acerca de las pruebas del software analiza estrategias junto con métodos para diseño de casos de prueba. Everett y Raymond (*Software Testing,* Wiley-IEEE Computer Society Press, 2007), Black (*Pragmatic Software Testing,* Wiley, 2007), Spiller *et al*. (*Software Testing Process: Test Management,* Rocky Nook, 2007), Perry (*Effective Methods for Software Testing,* 3a. ed., Wiley, 2005), Lewis (*Software Testing and Continuous Quality Improvement,* 2a. ed., Auerbach, 2004), Loveland *et al*. (*Software Testing Techniques,* Charles River Media, 2004), Burnstein (*Practical Software Testing,* Springer, 2003), Dustin (*Effective Software Testing,* Addison-Wesley, 2002), Craig y Kaskiel (*Systematic Software Testing,* Artech House, 2002), Tamres (*Introducing Software Testing,* Addison-Wesley, 2002), Whittaker (*How to Break Software,* Addison-Wesley, 2002), y Kaner *et al*. (*Lessons Learned in Software Testing,* Wiley, 2001) son sólo una pequeña muestra de muchos libros que estudian los principios, conceptos, estrategias y métodos de las pruebas.

Para aquellos lectores con interés en los métodos de desarrollo de software ágiles, Crispin y House (*Testing Extreme Programming*, Addison-Wesley, 2002) y Beck (*Test Driven Development: By Example*, Addison-Wesley, 2002) presentan estrategias y tácticas de prueba para programación extrema. Kamer *et al*. (*Lessons Learned*

in Software Testing, Wiley, 2001) presentan una colección de más de 300 "lecciones" pragmáticas (lineamientos) que todo examinador de software debe aprender. Watkins (*Testing IT: An off-the-Shelf Testing Process*, Cambridge University Press, 2001) establece un marco conceptual de prueba efectivo para todo tipo de software desarrollado o adquirido. Manges y O'Brien (*Agile Testing with Ruby and Rails*, Apress, 2008) abordan estrategias y técnicas de prueba para el lenguaje de programación Ruby y el marco conceptual web.

Sykes y McGregor (*Practical Guide to Testing Object-Oriented Software,* Addison-Wesley, 2001), Bashir y Goel (*Testing Object-Oriented Software,* Springer-Verlag, 2000), Binder (*Testing Object-Oriented Systems,* Addison-Wesley, 1999), Kung *et. al*. (*Testing Object-Oriented Software,* IEEE Computer Society Press, 1998) y Marick (*The Craft of Software Testing,* Prentice-Hall, 1997) presentan estrategias y métodos para probar sistemas OO.

Lineamientos para depuración se encuentran en los libros de Grötker *et al*. (*The Developer's Guide to Debugging,* Springer, 2008), Agans (*Debugging,* Amacon, 2006), Zeller (*Why Programs Fail: A Guide to Systematic Debugging,* Morgan Kaufmann, 2005), Tells y Hsieh (*The Science of Debugging,* The Coreolis Group, 2001), y Robbins (*Debugging Applications,* Microsoft Press, 2000). Kaspersky (*Hacker Debugging Uncovered,* A-List Publishing, 2005) addresses the technology of debugging tools. Younessi (*Object-Oriented Defect Management of Software,* Prentice-Hall, 2002) aborda la tecnología de las herramientas para depuración. Younessi (*Object-Oriented Defect Management of Software*, Prentice-Hall, 2002) presenta técnicas para manejar defectos que se encuentran en sistemas orientados a objetos. Beizer [Bei84] presenta una interesante "taxonomía de errores" que puede conducir a métodos efectivos para planificación de pruebas.

Los libros de Madisetti y Akgul (*Debugging Embedded Systems,* Springer, 2007), Robbins (*Debugging Microsoft.NET 2.0 Applications,* Microsoft Press, 2005), Best (*Linux Debugging and Performance Tuning,* Prentice-Hall, 2005), Ford y Teorey (*Practical Debugging in C++,* Prentice-Hall, 2002), Brown (*Debugging Perl,* McGraw-Hill, 2000) y Mitchell (*Debugging Java,* McGraw-Hill, 2000) abordan la naturaleza especial de la depuración para los entornos implicados en sus títulos.

Una gran variedad de fuentes de información acerca de estrategias para pruebas está disponible en internet. Una lista actualizada de referencias en la World Wide Web, que son relevantes para las estrategias de prueba de software, puede encontrarse en el sitio web del libro: **www.mhle.com/engcs/compsci/pressman/professional/olc/ser.htm**.

CAPÍTULO

18

PRUEBA DE APLICACIONES CONVENCIONALES

CONCEPTOS CLAVE

análisis de valor de frontera 425
complejidad ciclomática 417
entornos especializados 429
gráfico de flujo 415
matrices de grafo 420
métodos de prueba basados en gráficos 423
partición de equivalencia . . . 425
patrones 433
prueba basada en modelo . . 429
prueba de arreglo ortogonal 426
prueba de caja blanca 414
prueba de caja negra 423
prueba de estructura de control 420
prueba de ruta básica 414

Las pruebas presentan una interesante anomalía para los ingenieros de software, quienes por naturaleza son personas constructivas. Las pruebas requieren que el desarrollador deseche nociones preconcebidas sobre lo "correcto" del software recién desarrollado y luego trabajen duro para diseñar casos de prueba a fin de "romper" el software. Beizer [Bei90] describe esta situación de manera efectiva cuando afirma:

> Existe el mito de que no habría errores que pescar si fuésemos realmente buenos en programación. Si realmente nos pudiéramos concentrar, si todo mundo usara programación estructurada, diseño descendente... entonces no habría errores. Ése es el mito. Hay errores, dice el mito, porque somos malos en lo que hacemos; y si lo somos, deberíamos sentirnos culpables por ello. Por tanto, la aplicación de pruebas y el diseño de casos de prueba es una admisión del fracaso, que inspira una buena dosis de culpa. Y el tedio de las pruebas es un justo castigo por nuestros errores. ¿El castigo por qué? ¿Por ser humanos? ¿Culpa por qué? ¿Por fracasar en lograr la perfección inhumana? ¿Por no distinguir entre lo que otro programador piensa y lo que dice? ¿Por no poder ser telépatas? ¿Por no resolver problemas de comunicación humana a los que se les ha dado la vuelta... durante siglos?

¿Las pruebas deben inspirar culpa? ¿Las pruebas son realmente destructivas? La respuesta a estas preguntas es: "¡No!"

En este capítulo se estudian técnicas para el diseño de casos de prueba de software para aplicaciones convencionales. Este diseño se enfoca en un conjunto de técnicas para la creación de casos de prueba que satisfacen los objetivos de prueba globales y las estrategias de pruebas que se estudiaron en el capítulo 17.

UNA MIRADA RÁPIDA

¿Qué es? Una vez generado el código fuente, el software debe probarse para descubrir (y corregir) tantos errores como sea posible antes de entregarlo al cliente. La meta es diseñar una serie de casos de prueba que tengan una alta probabilidad de encontrar errores; ¿pero cómo? Ahí es donde entran en escena las técnicas de prueba de software. Dichas técnicas proporcionan lineamientos sistemáticos para diseñar pruebas que: 1) revisen la lógica interna y las interfaces de todo componente de software y 2) revisen los dominios de entrada y salida del programa para descubrir errores en el funcionamiento, comportamiento y rendimiento del programa.

¿Quién lo hace? Durante las primeras etapas del proceso, un ingeniero de software realiza todas las pruebas. Sin embargo, conforme avanza el proceso, pueden involucrarse especialistas en pruebas.

¿Por qué es importante? Las revisiones y otras acciones SQA pueden y deben descubrir errores, pero no son suficientes. Cada vez que el programa se ejecuta, ¡el cliente lo prueba! Por tanto, tiene que ejecutarse el programa antes de que llegue al cliente, con la intención específica de encontrar y remover todos los errores. Para encontrar el mayor número posible de éstos, las pruebas deben realizarse de manera sistemática y deben diseñarse casos de prueba usando técnicas sistematizadas.

¿Cuáles son los pasos? Para aplicaciones convencionales, el software se prueba desde dos perspectivas diferentes: 1) la lógica de programa interno se revisa usando técnicas de diseño de casos de prueba de "caja blanca" y 2) los requerimientos de software se revisan usando técnicas de diseño de casos de prueba de "caja negra". El uso de casos auxilia en el diseño de pruebas para descubrir errores de validación del software. En todo caso, la intención es encontrar el máximo número de errores con la mínima cantidad de esfuerzo y tiempo.

¿Cuál es el producto final? Se diseña y documenta un conjunto de casos de prueba elaborados para revisar la lógica interna, las interfaces, las colaboraciones de componentes y los requerimientos externos; se definen los resultados esperados y se registran los resultados reales.

¿Cómo me aseguro de que lo hice bien? Cuando se realizan pruebas, cambia el punto de vista. ¡Intente con ahínco "romper" el software! Diseñe casos de prueba en forma sistemática y revise minuciosamente los casos de prueba creados. Además, puede evaluar la cobertura de la prueba y rastrear las actividades de detección de errores.

18.1 FUNDAMENTOS DE LAS PRUEBAS DEL SOFTWARE

Cita:

"Todo programa hace algo bien, sólo que puede no ser aquello que queremos que haga."

Anónimo

La meta de probar es encontrar errores, y una buena prueba es aquella que tiene una alta probabilidad de encontrar uno. Por tanto, un sistema basado en computadora o un producto debe diseñarse e implementarse teniendo en mente la "comprobabilidad". Al mismo tiempo, las pruebas en sí mismas deben mostrar un conjunto de características que logren la meta de encontrar la mayor cantidad de errores con el mínimo esfuerzo.

Comprobabilidad. James Bach[1] proporciona la siguiente definición de comprobabilidad: "La *comprobabilidad del software* significa simplemente saber con cuánta facilidad puede probarse [un programa de cómputo]." Las siguientes características conducen a software comprobable.

¿Cuáles son las características de la comprobabilidad?

Operatividad. "Mientras mejor funcione, más eficientemente puede probarse." Si un sistema se diseña e implementa teniendo como objetivo la calidad, relativamente pocos errores bloquearán la ejecución de las pruebas, lo que permitirá avanzar en ellas sin interrupciones.

Observabilidad. "Lo que ve es lo que prueba." Las entradas proporcionadas como parte de las pruebas producen distintas salidas. Los estados del sistema y las variables son visibles o consultables durante la ejecución. La salida incorrecta se identifica con facilidad. Los errores internos se detectan y se reportan de manera automática. El código fuente es accesible.

Controlabilidad. "Mientras mejor pueda controlar el software, más podrá automatizar y optimizar las pruebas." Todas las salidas posibles pueden generarse a través de alguna combinación de entradas, y los formatos de entrada/salida (E/S) son consistentes y estructurados. Todo código es ejecutable a través de alguna combinación de entradas. El ingeniero de pruebas puede controlar directamente los estados del software, del hardware y las variables. Las pruebas pueden especificarse, automatizarse y reproducirse convenientemente.

Descomponibilidad. "Al controlar el ámbito de las pruebas, es posible aislar más rápidamente los problemas y realizar pruebas nuevas y más inteligentes." El sistema de software se construye a partir de módulos independientes que pueden probarse de manera independiente.

Cita:

"Los errores son más comunes, más dominantes y más problemáticos en software que en otras tecnologías."

David Parnas

Simplicidad. "Mientras haya menos que probar, más rápidamente se le puede probar." El programa debe mostrar *simplicidad funcional* (por ejemplo, el conjunto característico es el mínimo necesario para satisfacer los requerimientos); *simplicidad estructural* (la arquitectura es modular para limitar la propagación de fallos) y *simplicidad de código* (se adopta un estándar de codificación para facilitar la inspección y el mantenimiento).

Estabilidad. "Mientras menos cambios, menos perturbaciones para probar." Los cambios al software son raros, se controlan cuando ocurren y no invalidan las pruebas existentes. El software se recupera bien de los fallos.

Comprensibilidad. "Mientras más información se tenga, se probará con más inteligencia." El diseño arquitectónico y las dependencias entre componentes internos, externos y compartidos son bien comprendidos. La documentación técnica es accesible al instante, está bien organizada, es específica, detallada y precisa. Los cambios al diseño son comunicados a los examinadores.

Pueden usarse los atributos sugeridos por Bach para desarrollar una configuración de software (es decir, programas, datos y documentos) que sean fáciles de probar.

Características de la prueba. ¿Y qué hay acerca de las pruebas en sí? Kaner, Falk y Nguyen [Kan93] sugieren los siguientes atributos de una "buena" prueba:

1 Los párrafos que siguen se usan con permiso de James Bach (copyright 1994) y se adaptaron de material que originalmente apareció en un comentario en el grupo de noticias comp.software-eng.

? ¿Qué es una "buena" prueba?

Una buena prueba tiene una alta probabilidad de encontrar un error. Para lograr esta meta, el examinador debe comprender el software e intentar desarrollar una imagen mental de cómo puede fallar. De manera ideal, se prueban las clases de fallas. Por ejemplo, una clase de fallas potenciales en una interfaz gráfica de usuario es la falla para reconocer la posición adecuada del ratón. Se diseña entonces un conjunto de pruebas para revisar el ratón con la intención de demostrar un error en el reconocimiento de la posición del ratón.

Una buena prueba no es redundante. El tiempo y los recursos de la prueba son limitados. No se trata de realizar una prueba que tenga el mismo propósito que otra. Cada una debe tener un propósito diferente (incluso si es sutilmente diferente).

Una buena prueba debe ser "la mejor de la camada" [Kan93]. En un grupo de pruebas que tengan una intención similar, las limitaciones de tiempo y recursos pueden mitigar la ejecución de sólo un subconjunto de dichas pruebas. En tales casos, debe usarse la prueba que tenga la mayor probabilidad de descubrir toda una clase de errores.

Una buena prueba no debe ser demasiado simple o demasiado compleja. Aunque en ocasiones es posible combinar una serie de pruebas en un caso de prueba, los efectos colaterales posibles asociados con este enfoque pueden enmascarar errores. En general, cada prueba debe ejecutarse por separado.

CASASEGURA

Diseño de pruebas únicas

La escena: Cubículo de Vinod.

Participantes: Vinod y Ed, miembros del equipo de ingeniería de software CasaSegura.

La conversación:

Vinod: Así que éstos son los casos de prueba que quieres aplicar para la operación *passwordValidation*.

Ed: Sí, deben cubrir muchas de las posibilidades para los tipos de contraseñas que pueda ingresar un usuario.

Vinod: Veamos... observas que la contraseña correcta será 8080, ¿verdad?

Ed: Ajá.

Vinod: ¿Y especificas las contraseñas 1234 y 6789 para probar el error en el reconocimiento de las contraseñas inválidas?

Ed: Exacto, y también pruebo las contraseñas que están cerca de la contraseña correcta, a ver... 8081 y 8180.

Vinod: Ésos están bien, pero no le veo mucho caso aplicar las entradas 1234 y 6789. Son redundantes... prueban la misma cosa, ¿o no?

Ed: Bueno, son valores diferentes.

Vinod: Es cierto, pero si 1234 no descubre un error... en otras palabras... la operación *passwordValidation* detecta que es una contraseña inválida, no es probable que 6789 nos muestre algo nuevo.

Ed: Ya veo lo que dices.

Vinod: No intento ser quisquilloso... es sólo que tenemos tiempo limitado para hacer las pruebas, así que es buena idea aplicar pruebas que tengan una alta probabilidad de encontrar nuevos errores.

Ed: No hay problema... Pensaré en esto un poco más.

18.2 VISIONES INTERNA Y EXTERNA DE LAS PRUEBAS

Cita:

"Sólo hay una regla en el diseño de casos de prueba: cubrir todas las características, mas no hacer demasiados casos de prueba."

Tsuneo Yamaura

Cualquier producto sometido a ingeniería (y la mayoría de otras cosas) pueden probarse en una de dos formas: 1) al conocer la función específica que se asignó a un producto para su realización, pueden llevarse a cabo pruebas que demuestren que cada función es completamente operativa mientras al mismo tiempo se buscan errores en cada función, 2) al conocer el funcionamiento interno de un producto, pueden realizarse pruebas para garantizar que "todos los engranes embonan"; es decir, que las operaciones internas se realizan de acuerdo con las especificaciones y que todos los componentes internos se revisaron de manera adecuada. El primer

enfoque de pruebas considera una visión externa y se llama prueba de caja negra. El segundo requiere una visión interna y se denomina prueba de caja blanca.[2]

PUNTO CLAVE

Las pruebas de caja blanca pueden diseñarse sólo después de que existe el diseño a nivel de componentes (o código fuente). Debe disponerse de los detalles lógicos del programa.

La *prueba de caja negra* se refiere a las pruebas que se llevan a cabo en la interfaz del software. Una prueba de caja negra examina algunos aspectos fundamentales de un sistema con poca preocupación por la estructura lógica interna del software. La *prueba de caja blanca* del software se basa en el examen cercano de los detalles de procedimiento. Las rutas lógicas a través del software y las colaboraciones entre componentes se ponen a prueba al revisar conjuntos específicos de condiciones y/o bucles.

A primera vista, parecería que las pruebas de caja blanca muy extensas conducirían a "programas 100 por ciento correctos". Lo único que se necesita es definir todas las rutas lógicas, desarrollar casos de prueba para revisarlas y evaluar resultados, es decir, generar casos de prueba para revisar de manera exhaustiva la lógica del programa. Por desgracia, las pruebas exhaustivas presentan ciertos problemas logísticos. Hasta para programas pequeños, el número de posibles rutas lógicas puede ser muy grande. Sin embargo, las pruebas de caja blanca no deben descartarse como imprácticas. Puede seleccionarse y revisarse un número limitado de rutas lógicas importantes. Puede probarse la validez de las estructuras de datos importantes.

INFORMACIÓN

Pruebas exhaustivas

Considere un programa de 100 líneas en el lenguaje C. Después de alguna declaración básica de datos, el programa contiene dos bucles anidados que se ejecutan de 1 a 20 veces cada uno, dependiendo de las condiciones especificadas en la entrada. Dentro del bucle interior, se requieren cuatro constructos if-then-else. ¡Existen aproximadamente 10^{14} rutas posibles que pueden ejecutarse en este programa!

Para poner este número en perspectiva, suponga que se desarrolló un procesador de prueba mágico ("mágico" porque no existe tal procesador) para realizar pruebas exhaustivas. El procesador puede desarrollar un caso de prueba, ejecutarlo y evaluar los resultados en un milisegundo. Si trabajara 24 horas al día los 365 días del año, el procesador trabajaría durante 3 170 años para probar el programa. Esto, sin duda alguna, causaría estragos en la mayoría de los calendarios de desarrollo.

Por tanto, es razonable afirmar que la prueba exhaustiva es imposible para sistemas de software grandes.

18.3 PRUEBA DE CAJA BLANCA

Cita:

"Los errores se esconden en las esquinas y se congregan en las fronteras."

Boris Beizer

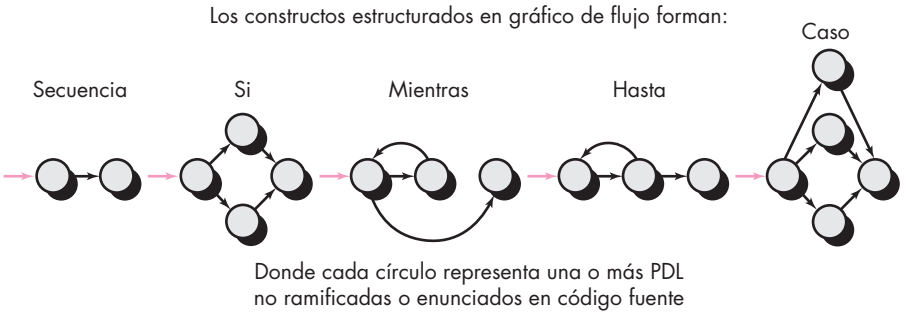
La *prueba de caja blanca*, en ocasiones llamada *prueba de caja de vidrio*, es una filosofía de diseño de casos de prueba que usa la estructura de control descrita como parte del diseño a nivel de componentes para derivar casos de prueba. Al usar los métodos de prueba de caja blanca, puede derivar casos de prueba que: 1) garanticen que todas las rutas independientes dentro de un módulo se revisaron al menos una vez, 2) revisen todas las decisiones lógicas en sus lados verdadero y falso, 3) ejecuten todos los bucles en sus fronteras y dentro de sus fronteras operativas y 4) revisen estructuras de datos internas para garantizar su validez.

18.4 PRUEBA DE RUTA BÁSICA

La *prueba de ruta* o *trayectoria básica* es una técnica de prueba de caja blanca propuesta por primera vez por Tom McCabe [McC76]. El método de ruta básica permite al diseñador de casos de prueba derivar una medida de complejidad lógica de un diseño de procedimiento y usar esta

2 En ocasiones, en lugar de pruebas de caja negra y de caja blanca, se usan, respectivamente, los términos *prueba funcional* y *prueba estructural*.

FIGURA 18.1
Notación de gráfico de flujo



medida como guía para definir un conjunto básico de rutas de ejecución. Los casos de prueba derivados para revisar el conjunto básico tienen garantía para ejecutar todo enunciado en el programa, al menos una vez durante la prueba.

18.4.1 Notación de gráfico o grafo de flujo

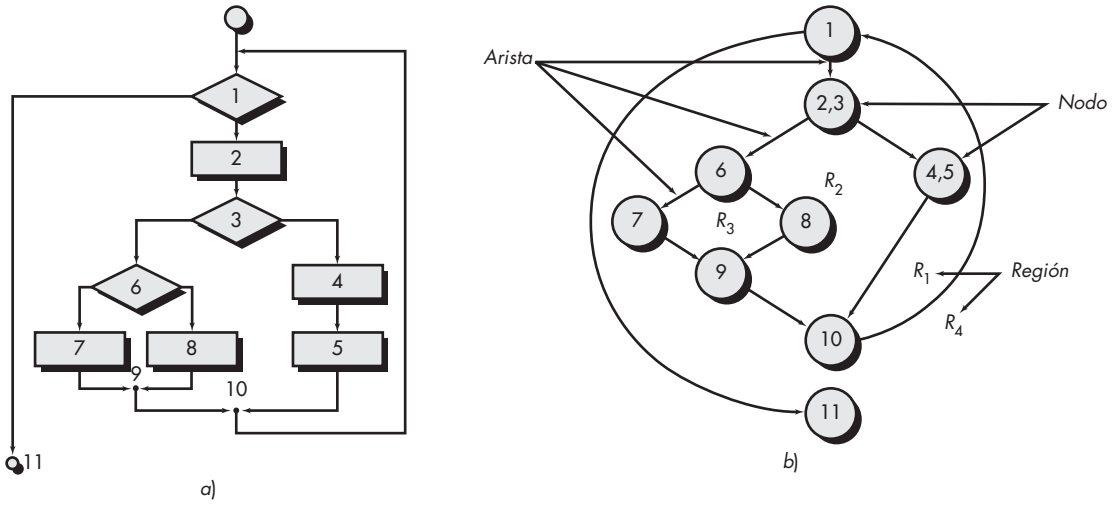
Antes de considerar el método de ruta básica, debe introducirse una notación simple para la representación del flujo de control, llamado *gráfico de flujo* (o *gráfico de programa*).[3] El gráfico de flujo muestra el flujo de control lógico que usa la notación ilustrada en la figura 18.1. Cada constructo estructurado (capítulo 10) tiene un correspondiente símbolo de gráfico de flujo.

CONSEJO

Un gráfico de flujo debe dibujarse sólo cuando la estructura lógica de un componente es compleja. El gráfico de flujo le permite rastrear rutas de programa con más facilidad.

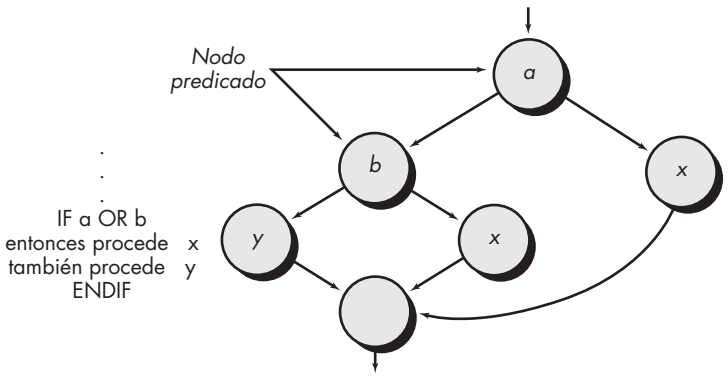
Para ilustrar el uso de un gráfico de flujo, considere la representación del diseño de procedimiento en la figura 18.2*a*). Aquí se usó un diagrama de flujo para mostrar la estructura de control del programa. La figura 18.2*b*) mapea el diagrama de flujo en un gráfico de flujo correspondiente (suponiendo que el diagrama de flujo no contiene condiciones compuestas en los diamantes de decisión). Con referencia a la figura 18.2*b*), cada círculo, llamado *nodo de gráfico de flujo*, representa uno o más enunciados de procedimiento. Una secuencia de cajas de proceso y un diamante de decisión pueden mapearse en un solo nodo. Las flechas en el gráfico de flujo, llamadas *aristas* o *enlaces*, representan flujo de control y son análogas a las flechas en el diagrama de flujo. Una arista debe terminar en un nodo, incluso si el nodo no representa algún enunciado de pro-

FIGURA 18.2
***a*) Diagrama de flujo y *b*) gráfico de flujo**



3 En la actualidad, el método de ruta básica puede realizarse sin el uso de gráficos de flujo. No obstante, sirven como una notación útil para comprender el flujo de control e ilustrar el enfoque.

FIGURA 18.3
Lógica compuesta



cedimiento (por ejemplo, vea el símbolo de gráfico de flujo para el constructo if-then-else). Las áreas acotadas por aristas y nodos se llaman *regiones*. Cuando se cuentan las regiones, el área afuera del gráfico se incluye como región.[4]

Cuando en un diseño de procedimiento se encuentran condiciones compuestas, la generación de un gráfico de flujo se vuelve ligeramente más complicada. Una condición compuesta ocurre cuando uno o más operadores booleanos (OR, AND, NAND, NOR lógicos) se presenta en un enunciado condicional. En la figura 18.3, el segmento en lenguaje de diseño de programa (PDL, por sus siglas en inglés) se traduce en el gráfico de flujo mostrado. Observe que se crea un nodo separado para cada una de las condiciones *a* y *b* en el enunciado IF *a* OR *b*. Cada nodo que contiene una condición se llama *nodo predicado* y se caracteriza por dos o más aristas que emanan de él.

18.4.2 Rutas de programa independientes

Una *ruta independiente* es cualquiera que introduce al menos un nuevo conjunto de enunciados de procesamiento o una nueva condición en el programa. Cuando se establece como un gráfico de flujo, una ruta independiente debe moverse a lo largo de al menos una arista que no se haya recorrido antes de definir la ruta. Por ejemplo, un conjunto de rutas independientes para el gráfico de flujo que se ilustra en la figura 18.2*b*) es

ruta 1: 1-11

ruta 2: 1-2-3-4-5-10-1-11

ruta 3: 1-2-3-6-8-9-10-1-11

ruta 4: 1-2-3-6-7-9-10-1-11

Observe que cada nueva ruta introduce una nueva arista. La ruta

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11

no se considera como independiente porque simplemente es una combinación de rutas ya especificadas y no recorre alguna arista nueva.

Las rutas de la 1 a la 4 constituyen un *conjunto básico* para el gráfico de flujo de la figura 18.2*b*). Es decir, si se pueden diseñar pruebas para forzar la ejecución de estas rutas (un conjunto básico), todo enunciado en el programa tendrá garantizada su ejecución al menos una vez, y cada condición se ejecutará en sus lados verdadero y falso. Debe señalarse que el con-

4 En la sección 18.6.1 se presenta un análisis más detallado de los gráficos y sus usos.

junto básico no es único. De hecho, para un diseño de procedimiento dado, pueden derivarse algunos conjuntos básicos diferentes.

¿Cómo saber cuántas rutas buscar? El cálculo de la complejidad ciclomática proporciona la respuesta. La *complejidad ciclomática* es una medición de software que proporciona una evaluación cuantitativa de la complejidad lógica de un programa. Cuando se usa en el contexto del método de prueba de la ruta básica, el valor calculado por la complejidad ciclomática define el número de rutas independientes del conjunto básico de un programa y le brinda una cota superior para el número de pruebas que debe realizar a fin de asegurar que todos los enunciados se ejecutaron al menos una vez.

CONSEJO

La complejidad ciclomática es una medición útil para predecir aquellos módulos proclives al error. Úsela para planificar las pruebas así como para el diseño de casos de prueba.

? ¿Cómo se calcula la complejidad ciclomática?

La complejidad ciclomática tiene fundamentos en la teoría de gráficos y proporciona una medición de software extremadamente útil. La complejidad se calcula en una de tres formas:

1. El número de regiones del gráfico de flujo corresponde a la complejidad ciclomática.
2. La complejidad ciclomática $V(G)$ para un gráfico de flujo G se define como

   $V(G) = E - N + 2$

   donde E es el número de aristas del gráfico de flujo y N el número de nodos del gráfico de flujo.
3. La complejidad ciclomática $V(G)$ para un gráfico de flujo G también se define como

   $V(G) = P + 1$

   donde P es el número de nodos predicado contenidos en el gráfico de flujo G.

PUNTO CLAVE

La complejidad ciclomática proporciona la cota superior sobre el número de casos de prueba que se requerirán para garantizar que cada enunciado en el programa se ejecuta al menos una vez.

En el gráfico de flujo de la figura 18.2*b*), la complejidad ciclomática puede calcularse usando cada uno de los algoritmos recién indicados:

1. El gráfico de flujo tiene cuatro regiones.
2. $V(G) = 11$ aristas $- 9$ nodos $+ 2 = 4$.
3. $V(G) = 3$ nodos predicado $+ 1 = 4$.

Por tanto, la complejidad ciclomática del gráfico de flujo en la figura 18.2*b*) es 4.

Más importante, el valor para $V(G)$ proporciona una cota superior para el número de rutas independientes que forman el conjunto básico y, por implicación, una cota superior sobre el número de pruebas que deben diseñarse y ejecutarse para garantizar cobertura de todos los enunciados del programa.

CasaSegura

Uso de la complejidad ciclomática

La escena: Cubículo de Shakira.

Participantes: Vinod y Shakira, miembros del equipo de ingeniería del software CasaSegura, quienes trabajan en la planificación de las pruebas para la función de seguridad.

La conversación:

Shakira: Mira... Sé que debemos hacer pruebas de unidad a todos los componentes para la función de seguridad, pero hay muchos de ellos, y si consideras el número de operaciones que se tienen que revisar, no sé... tal vez deberíamos olvidar la prueba de caja blanca, integrar todo y comenzar a aplicar las pruebas de caja negra.

Vinod: ¿Supones que no tenemos tiempo suficiente para hacer pruebas de componentes, revisar las operaciones y luego integrar?

Shakira: La fecha límite para el primer incremento está más cerca de lo que quisiera... sí, estoy preocupada.

Vinod: ¿Por qué al menos no aplicas pruebas de caja blanca sobre las operaciones que tienen probabilidad de ser más proclives a errores?

Shakira (exasperada): ¿Y exactamente cómo sé cuáles son las más proclives a errores?

Vinod: *V* de *G*.

Shakira: ¿Qué?

Vinod: Complejidad ciclomática, *V* de *G*. Sólo calcula *V*(*G*) para cada una de las operaciones dentro de cada uno de los componentes y ve cuáles tienen los valores más altos para *V*(*G*). Ésas son las que tienen más probabilidad de ser proclives a errores.

Shakira: ¿Y cómo calculo *V* de *G*?

Vinod: Realmente es sencillo. Aquí hay un libro que describe cómo hacerlo.

Shakira (hojea el libro): Muy bien, no parece difícil. Lo intentaré. Las operaciones con la *V*(*G*) más altas serán las candidatas para las pruebas de caja blanca.

Vinod: Sólo recuerda que no hay garantías. Un componente con una *V*(*G*) baja puede ser proclive a errores.

Shakira: Bien. Pero al menos esto me ayudará a reducir el número de componentes que tienen que experimentar pruebas de caja blanca.

18.4.3 Derivación de casos de prueba

Cita:

"El cohete Ariane 5 estalló en el despegue debido exclusivamente a un defecto de software (un bug, un error) que involucra la conversión de un valor en punto flotante de 64 bits en un entero de 16 bits. El cohete y sus cuatro satélites no estaban asegurados y valían 500 millones de dólares. [Pruebas de ruta que revisaran la ruta de conversión] habrían descubierto el error, pero se vetaron por razones presupuestarias."

Reporte noticioso

El método de prueba de ruta básica puede aplicarse a un diseño de procedimientos o a un código fuente. En esta sección se presenta la prueba de ruta básica como una serie de pasos. El procedimiento *average* (promedio), que en la figura 18.4 se muestra en PDL, se usará como ejemplo para ilustrar cada paso del método de diseño de caso de prueba. Observe que *average*, aunque es un algoritmo extremadamente simple, contiene condiciones y bucles compuestos. Es posible aplicar los siguientes pasos para derivar el conjunto básico:

1. **Al usar el diseño o el código como cimiento, dibuje el gráfico de flujo correspondiente.** El gráfico de flujo se crea usando los símbolos y reglas de construcción que se presentaron en la sección 18.4.1. En el PDL para *average* de la figura 18.4, el gráfico de flujo se crea al numerar aquellos enunciados PDL que se mapearán en los nodos correspondientes del gráfico de flujo. En la figura 18.5 se muestra el gráfico de flujo correspondiente.
2. **Determine la complejidad ciclomática del gráfico de flujo resultante.** La complejidad ciclomática *V*(*G*) se determina al aplicar los algoritmos descritos en la sección 18.4.2. Debe observarse que *V*(*G*) puede determinarse sin desarrollar un gráfico de flujo,

FIGURA 18.4
PDL con identificación de nodos

```
PROCEDIMIENTO average;

    *  Este procedimiento calcula el promedio de 100 o menos
       números que se encuentran entre valores frontera; también
       calcula la suma y el número total válido.

    INTERFACE RETURNS average, total.input, total.valid;
    INTERFACE ACCEPTS value, minimum, maximum;

    TYPE value[1:100] IS SCALAR ARRAY;
    TYPE average, total.input, total.valid;
       minimum, maximum, sum IS SCALAR;
    TYPE i IS INTEGER;
1   i = 1;
    total.input = total.valid = 0;
    sum = 0;
    DO WHILE value[i] <> –999 AND total.input < 100   3
4      increment total.input by 1;
       IF value[i] > = minimum AND value[i] < = maximum   6
5         THEN increment total.valid by 1;
7                sum = s sum + value[i]
          ELSE skip
8      ENDIF
       increment i by 1;
9   ENDDO
       IF total.valid > 0   10
11        THEN average = sum / total.valid;
12        ELSE average = –999;
13  ENDIF
END average
```

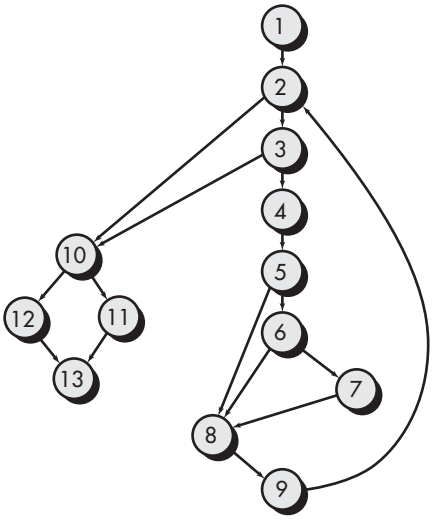


FIGURA 18.5
Gráfico de flujo para el procedimiento *average*

al contar todos los enunciados condicionales en el PDL (para el procedimiento *average*, las condiciones compuestas son dos) y sumar 1. En la figura 18.5,

$V(G)$ = 6 regiones
$V(G)$ = 17 aristas − 13 nodos + 2 = 6
$V(G)$ = 5 nodos predicado + 1 = 6

3. **Determine un conjunto básico de rutas linealmente independientes.** El valor de $V(G)$ proporciona la cota superior sobre el número de rutas linealmente independientes a través de la estructura de control del programa. En el caso del procedimiento *average*, se espera especificar seis rutas:

   ruta 1: 1-2-10-11-13

   ruta 2: 1-2-10-12-13

   ruta 3: 1-2-3-10-11-13

   ruta 4: 1-2-3-4-5-8-9-2-...

   ruta 5: 1-2-3-4-5-6-8-9-2-...

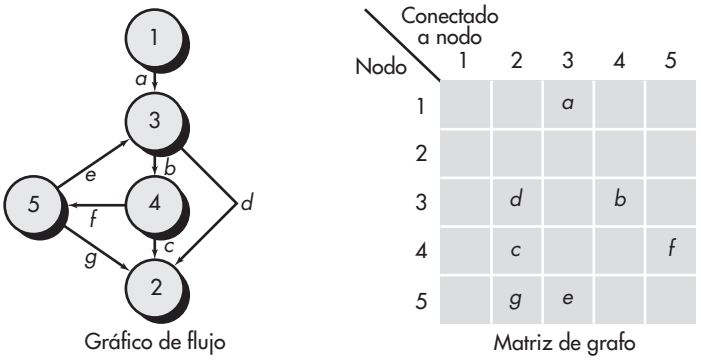
   ruta 6: 1-2-3-4-5-6-7-8-9-2-...

   La elipsis (...) después de las rutas 4, 5 y 6 indica que es aceptable cualquier ruta a través del resto de la estructura de control. Con frecuencia, vale la pena identificar nodos predicado como un auxiliar para la derivación de los casos de prueba. En este caso, los nodos 2, 3, 5, 6 y 10 son nodos predicado.

4. **Prepare casos de prueba que fuercen la ejecución de cada ruta en el conjunto básico.** Los datos deben elegirse de modo que las condiciones en los nodos predicado se establezcan de manera adecuada conforme se prueba cada ruta. Cada caso de prueba se ejecuta y compara con los resultados esperados. Una vez completados todos los casos de prueba, el examinador puede estar seguro de que todos los enunciados del programa se ejecutaron al menos una vez.

Es importante notar que algunas rutas independientes (tomemos por caso la ruta 1 del ejemplo) no pueden probarse en forma individual, es decir, la combinación de datos requerida para recorrer la ruta no puede lograrse en el flujo normal del programa. En tales casos, dichas rutas se prueban como parte de otra prueba de ruta.

FIGURA 18.6
Matriz de grafo



18.4.4 Matrices de grafo

El procedimiento para derivar el gráfico de flujo e incluso determinar un conjunto de rutas básicas es sensible a la mecanización. Una estructura de datos, llamada *matriz de un grafo,* puede ser bastante útil para desarrollar una herramienta de software que auxilie en la prueba de ruta básica.

Una matriz de grafo es una matriz cuadrada cuyo tamaño (es decir, número de filas y columnas) es igual al número de nodos del gráfico de flujo. Cada fila y columna corresponde a un nodo identificado y las entradas de la matriz corresponden a conexiones (una arista) entre nodos. En la figura 18.6 se muestra un ejemplo simple de gráfico de flujo y su correspondiente matriz de grafo [Bei90].

En esa figura, cada nodo en el gráfico de flujo se identifica mediante números, mientras que cada arista se identifica con letras. Una entrada de letra en la matriz corresponde a una conexión entre dos nodos. Por ejemplo, el nodo 3 se conecta con el nodo 4 mediante la arista *b*.

? ¿Qué es una matriz de grafo y cómo se le extiende para su uso en las pruebas?

En este punto, la matriz de grafo no es más que una representación tabular de un gráfico de flujo. Sin embargo, al agregar un enlace ponderado a cada entrada de matriz, la matriz de grafo puede convertirse en una poderosa herramienta para evaluar durante las pruebas la estructura de control del programa. El *enlace ponderado* proporciona información adicional acerca del flujo de control. En su forma más simple, el enlace ponderado es 1 (existe una conexión) o 0 (no existe conexión). Pero a los enlaces ponderados puede asignárseles otras propiedades más interesantes:

- La probabilidad de que un enlace (arista) se ejecutará.
- El tiempo de procesamiento que se emplea durante el recorrido de un enlace.
- La memoria requerida durante el recorrido de un enlace.
- Los recursos requeridos durante el recorrido de una prueba.

Beizer [Bei90] proporciona un tratamiento a fondo de algoritmos matemáticos adicionales que pueden aplicarse a las matrices gráficas. Con estas técnicas, el análisis requerido para diseñar casos de prueba puede ser parcial o completamente automatizado.

18.5 PRUEBA DE LA ESTRUCTURA DE CONTROL

La técnica de prueba de ruta básica descrita en la sección 18.4 es una de varias técnicas para probar la estructura de control. Aunque la prueba de ruta básica es simple y enormemente efectiva, no es suficiente en sí misma. En esta sección se estudian otras variaciones acerca de la prueba de la estructura de control. Esta prueba más amplia cubre y mejora la calidad de la prueba de caja blanca.

Cita:

"Poner más atención a la aplicación de las pruebas que a su diseño es un error clásico."

Brian Marick

18.5.1 Prueba de condición

La *prueba de condición* [Tai89] es un método de diseño de casos de prueba que revisa las condiciones lógicas contenidas en un módulo de programa. Una condición simple es una variable booleana o una expresión relacional, posiblemente precedida de un operador NOT (¬). Una expresión relacional toma la forma

E_1 <operador relacional> E_2

donde E_1 y E_2 son expresiones aritméticas y <operador relacional> es uno de los siguientes: $<$, $\leq$, $=$, $\neq$ (no igualdad), $>$ o $\geq$. Una *condición compuesta* se integra con dos o más condiciones simples, operadores booleanos y paréntesis. Se supone que los operadores booleanos permitidos en una condición compuesta incluyen OR (|), AND (&) y NOT (¬). Una condición sin expresiones relacionales se conoce como expresión booleana.

PUNTO CLAVE

Los errores son mucho más comunes en la cercanía de las condiciones lógicas que en el lugar de los enunciados de procesamiento secuencial.

Si una condición es incorrecta, entonces al menos un componente de la condición es incorrecto. Por tanto, los tipos de errores en una condición incluyen errores de operador booleano (operadores booleanos incorrectos/perdidos/adicionales), de variable booleana, de paréntesis booleanos, de operador relacional y de expresión aritmética. El método de prueba de condición se enfoca en la prueba de cada condición del programa para asegurar que no contiene errores.

Cita:

"Los buenos examinadores son maestros para notar 'algo divertido' y actuar sobre ello."

Brian Marick

18.5.2 Prueba de flujo de datos

El método de prueba de flujo de datos [Fra93] selecciona rutas de prueba de un programa de acuerdo con las ubicaciones de las definiciones y con el uso de variables en el programa. Para ilustrar el enfoque de prueba de flujo de datos, suponga que a cada enunciado en un programa se le asigna un número de enunciado único y que cada función no modifica sus parámetros o variables globales. Para un enunciado con *S* como su número de enunciado,

DEF(*S*) = {*X* | enunciado *S* contiene una definición de *X*}
USE(*S*) = {*X* | enunciado *S* contiene un uso de *X*}

Si el enunciado *S* es un *enunciado if* o *loop*, su conjunto DEF es vacío y su conjunto USE se basa en la condición del enunciado *S*. Se dice que la definición de la variable *X* en el enunciado *S* está *viva* en el enunciado *S*′ si existe una ruta desde el enunciado *S* hasta el enunciado *S*′ que no contiene otra definición de *X*.

CONSEJO

Es irreal suponer que la prueba de flujo de datos se usará de manera extensa cuando se prueba un sistema grande. Sin embargo, puede usarse en forma dirigida para áreas de software que sean sospechosas.

Una *cadena de definición de uso (DU)* de la variable *X* es de la forma [*X, S, S*′], donde *S* y *S*′ son números de enunciado, *X* está en DEF(*S*) y en USE(*S*′), y la definición de *X* en el enunciado *S* está viva en el enunciado *S*′.

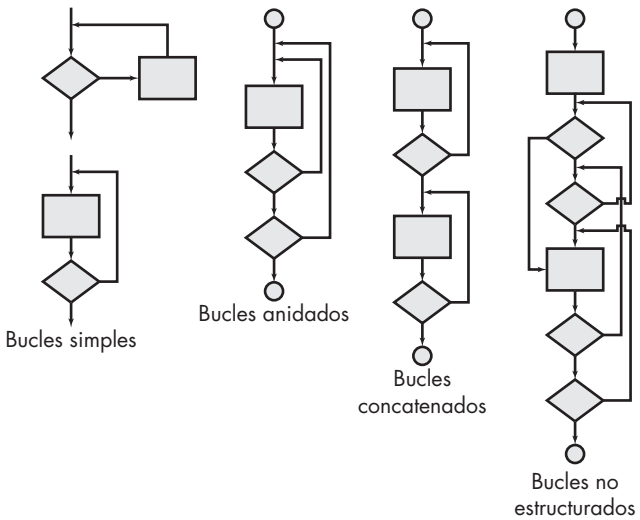
Una estrategia de prueba de flujo de datos simple es requerir que toda cadena DU se cubra al menos una vez. A esta estrategia se le conoce como estrategia de prueba DU. Se ha demostrado que la prueba DU no garantiza la cobertura de todas las ramas de un programa. Sin embargo, la prueba DU no garantiza la cobertura de una rama sólo en raras situaciones, como en los constructos if-then-else en los cuales la *parte then* no tiene definición de alguna variable y la *parte else* no existe. En esta situación, la rama *else* del enunciado *if* no necesariamente se cubre con la prueba DU.

18.5.3 Prueba de bucle

Los bucles son la piedra de toque de la gran mayoría de todos los algoritmos implementados en el software. Y aún así, con frecuencia se les pone poca atención mientras se realizan las pruebas de software.

La *prueba de bucle* es una técnica de prueba de caja blanca que se enfoca exclusivamente en la validez de los constructos bucle. Pueden definirse cuatro clases diferentes de bucles [Bei90]: simples, concatenados, anidados y no estructurados (figura 18.7).

FIGURA 18.7
Clases de bucles



Bucles simples. El siguiente conjunto de pruebas puede aplicarse a los bucles simples, donde *n* es el máximo número de pasadas permisibles a través del bucle.

1. Saltar por completo el bucle.
2. Sólo una pasada a través del bucle.
3. Dos pasadas a través del bucle.
4. *m* pasadas a través del bucle, donde $m < n$.
5. $n - 1$, n, $n + 1$ pasadas a través del bucle.

Bucles anidados. Si tuviera que extender el enfoque de la prueba para bucles simples a los bucles anidados, el número de pruebas posibles crecería geométricamente conforme el nivel de anidado aumenta. Esto daría como resultado un número impráctico de pruebas. Beizer [Bei90] sugiere un acercamiento que ayudará a reducir el número de pruebas:

1. Comience con el bucle más interno. Establezca todos los otros bucles a valores mínimos.
2. Realice pruebas de bucle simple para el bucle más interno mientras mantiene los bucles exteriores en sus valores mínimos de parámetro de iteración (por ejemplo, contador de bucle). Agregue otras pruebas para valores fuera-de-rango o excluidos.
3. Trabaje hacia afuera y realice pruebas para el siguiente bucle, pero mantenga los otros bucles exteriores en valores mínimos y los otros bucles anidados en valores "típicos".
4. Continúe hasta que todos los bucles se hayan probado.

No es posible probar los bucles no estructurados de manera efectiva. Se deben refactorizar.

Bucles concatenados. Los bucles concatenados pueden probarse usando el enfoque definido para bucles simples si cada uno de los bucles es independiente de los otros. No obstante, si dos bucles se concatenan y usa el contador de bucle para el bucle 1 como el valor inicial para el bucle 2, entonces los bucles no son independientes. Cuando los bucles no son independientes, se recomienda el enfoque aplicado a bucles anidados.

Bucles no estructurados. Siempre que sea posible, esta clase de bucles debe rediseñarse para reflejar el uso de los constructos de programación estructurada (capítulo 10).

18.6 PRUEBAS DE CAJA NEGRA

Las *pruebas de caja negra*, también llamadas *pruebas de comportamiento*, se enfocan en los requerimientos funcionales del software; es decir, las técnicas de prueba de caja negra le permiten derivar conjuntos de condiciones de entrada que revisarán por completo todos los requerimientos funcionales para un programa. Las pruebas de caja negra no son una alternativa para las técnicas de caja blanca. En vez de ello, es un enfoque complementario que es probable que descubra una clase de errores diferente que los métodos de caja blanca.

Las pruebas de caja negra intentan encontrar errores en las categorías siguientes: 1) funciones incorrectas o faltantes, 2) errores de interfaz, 3) errores en las estructuras de datos o en el acceso a bases de datos externas, 4) errores de comportamiento o rendimiento y 5) errores de inicialización y terminación.

Cita:

"Errar es humano, pero encontrar un error es divino."

Robert Dunn

A diferencia de las pruebas de caja blanca, que se realizan tempranamente en el proceso de pruebas, la prueba de caja negra tiende a aplicarse durante las últimas etapas de la prueba (vea el capítulo 17). Puesto que, a propósito, la prueba de caja negra no considera la estructura de control, la atención se enfoca en el dominio de la información. Las pruebas se diseñan para responder a las siguientes preguntas:

¿Qué preguntas responden las pruebas de caja negra?

- ¿Cómo se prueba la validez funcional?
- ¿Cómo se prueban el comportamiento y el rendimiento del sistema?
- ¿Qué clases de entrada harán buenos casos de prueba?
- ¿El sistema es particularmente sensible a ciertos valores de entrada?
- ¿Cómo se aíslan las fronteras de una clase de datos?
- ¿Qué tasas y volumen de datos puede tolerar el sistema?
- ¿Qué efecto tendrán sobre la operación del sistema algunas combinaciones específicas de datos?

Al aplicar las técnicas de caja negra, se deriva un conjunto de casos de prueba que satisfacen los siguientes criterios [Mye79]: 1) casos de prueba que reducen, por una cuenta que es mayor que uno, el número de casos de prueba adicionales que deben diseñarse para lograr pruebas razonables y 2) casos de prueba que dicen algo acerca de la presencia o ausencia de clases de errores, en lugar de un error asociado solamente con la prueba específica a mano.

18.6.1 Métodos de prueba basados en gráficos

PUNTO CLAVE

Una gráfica representa las relaciones entre objetos datos y objetos programa, lo que permite derivar casos de prueba que buscan errores asociados con dichas relaciones.

El primer paso en la prueba de caja negra es entender los objetos[5] que se modelan en software y las relaciones que conectan a dichos objetos. Una vez logrado esto, el siguiente paso es definir una serie de pruebas que verifiquen "que todos los objetos tengan la relación mutua esperada" [Bei95]. Dicho de otra forma, la prueba de software comienza con la creación de un gráfico de objetos importantes y sus relaciones, y luego diseña una serie de pruebas que cubrirán el gráfico, de modo que cada objeto y relación se revise y se descubran errores.

Para lograr estos pasos, comience por crear un *gráfico*: una colección de *nodos* que representen objetos, *enlaces* que representen las relaciones entre objetos, *nodos ponderados* que describan las propiedades de un nodo (por ejemplo, un valor de datos o comportamiento de estado específicos) y *enlaces ponderados* que describan alguna característica de un enlace.

En la figura 18.8*a*) se muestra la representación simbólica de un gráfico. Los nodos se representan como círculos conectados mediante ligas que tienen algunas formas diferentes. Un en-

5 En este contexto, debe considerar el término *objetos* en el contexto más amplio posible. Abarca objetos de datos, componentes tradicionales (módulos) y elementos orientados a objeto del software de cómputo.

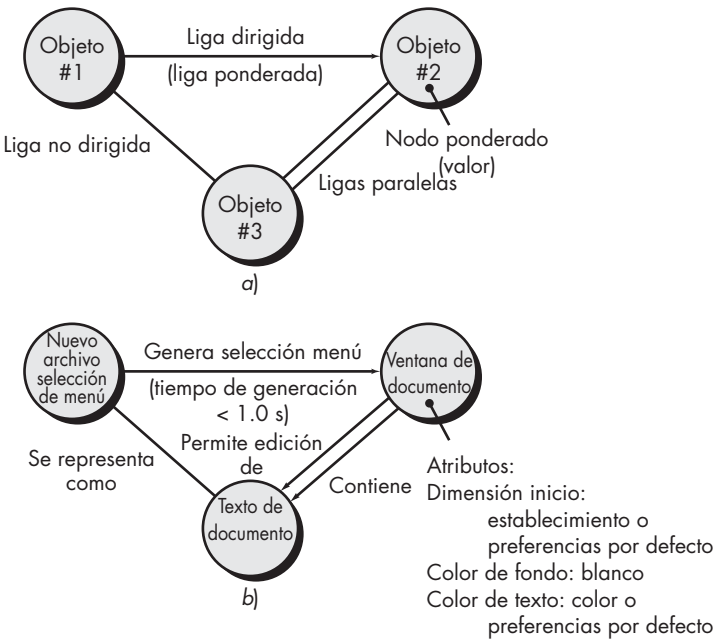


FIGURA 18.8
***a*) Notación de gráfico; *b*) ejemplo simple**

lace dirigido (representado mediante una flecha) indica que una relación sólo se mueve en una dirección. Un *enlace bidireccional*, también llamado *enlace simétrico*, implica que la relación se aplica en ambas direcciones. Los *enlaces paralelos* se usan cuando entre los nodos gráficos se establecen algunas relaciones diferentes.

Como ejemplo simple, considere una porción de un gráfico para una aplicación de un procesador de palabras (figura 18.8*b*) donde

Objeto #1 = **newFile** (selección de menú)

Objeto #2 = **documentWindow**

Objeto #3 = **documentText**

En la figura, una selección de menú en **newFile** genera una ventana de documento. El nodo ponderado de **documentWindow** proporciona una lista de los atributos de ventana que se esperan cuando se genere la ventana. El enlace ponderado indica que la ventana debe generarse en menos de 1.0 segundo. Un enlace no dirigido establece una relación simétrica entre la selección de menú **newFile** y **documentText**, y los enlaces paralelos indican relaciones entre **documentWindow** y **documentText**. En realidad, tendría que generarse un gráfico más detallado como precursor para el diseño de casos de prueba. Entonces podrían derivarse casos de prueba al recorrer el gráfico y cubrir cada una de las relaciones mostradas. Dichos casos de prueba se designan con la intención de encontrar errores en alguna de las relaciones. Beizer [Bei95] describe algunos métodos de prueba de comportamiento que pueden usar gráficos:

Modelado de flujo de transacción. Los nodos representan pasos en alguna transacción (por ejemplo, los pasos requeridos para hacer una reservación en una aerolínea con el uso de un servicio en línea) y los enlaces representan la conexión lógica entre los pasos (por ejemplo, **ingresarInformaciónVuelo** se sigue de *validaciónProcesamientoDisponibilidad*). El diagrama de flujo de datos (capítulo 7) puede usarse para auxiliar en la creación de gráficos de este tipo.

Modelado de estado finito. Los nodos representan diferentes estados del software observables por el usuario (por ejemplo, cada una de las "pantallas" que aparecen cuando un

empleado ingresa información conforme toma una orden telefónica) y los enlaces representan las transiciones que ocurren para moverse de estado a estado (por ejemplo, **pedidoInformación** se verifica durante *inventarioBusquedaDisponibilidad*, y es seguido de la entrada **clienteFacturaInformación**). El diagrama de estado (capítulo 7) puede usarse para auxiliar en la creación de gráficos de este tipo.

Modelado de flujo de datos. Los nodos son objetos datos y los enlaces son las transformaciones que ocurren para traducir un objeto datos en otro. Por ejemplo, el nodo retención de impuesto FICA (**FTW**) se calcula a partir de los ingresos brutos (**IB**), usando la relación **FTW = 0.62 × IB**.

Modelado de temporización. Los nodos son objetos programa y los enlaces son las conexiones secuenciales entre dichos objetos. Los enlaces ponderados se usan para especificar los tiempos de ejecución requeridos conforme se ejecuta el programa.

Un análisis detallado de cada uno de estos métodos de prueba basados en gráfico está más allá del ámbito de este libro. Si se tiene mayor interés, consulte [Bei95] para conocer una cobertura más amplia.

18.6.2 Partición de equivalencia

Las clases de entrada se conocen relativamente pronto en el proceso del software. Por esta razón, se debe pensar acerca de la partición de equivalencia conforme se crea el diseño.

La *partición de equivalencia* es un método de prueba de caja negra que divide el dominio de entrada de un programa en clases de datos de los que pueden derivarse casos de prueba. Un caso de prueba ideal descubre de primera mano una clase de errores (por ejemplo, procesamiento incorrecto de todos los datos carácter) que de otro modo podrían requerir la ejecución de muchos casos de prueba antes de observar el error general.

El diseño de casos de prueba para la partición de equivalencia se basa en una evaluación de las *clases de equivalencia* para una condición de entrada. Con los conceptos introducidos en la sección precedente, si un conjunto de objetos puede vincularse mediante relaciones que son simétricas, transitivas y reflexivas, se presenta una clase de equivalencia [Bei95]. Una clase de equivalencia representa un conjunto de estados válidos o inválidos para condiciones de entrada. Por lo general, una condición de entrada es un valor numérico específico, un rango de valores, un conjunto de valores relacionados o una condición booleana. Las clases de equivalencia pueden definirse de acuerdo con los siguientes lineamientos:

? ¿Cómo se definen las clases de equivalencia para pruebas?

1. Si una condición de entrada especifica un rango, se define una clase de equivalencia válida y dos inválidas.
2. Si una condición de entrada requiere un valor específico, se define una clase de equivalencia válida y dos inválidas.
3. Si una condición de entrada especifica un miembro de un conjunto, se define una clase de equivalencia válida y una inválida.
4. Si una condición de entrada es booleana, se define una clase válida y una inválida.

Al aplicar los lineamientos para la derivación de clases de equivalencia, pueden desarrollarse y ejecutarse los casos de prueba para cada ítem de datos del dominio de entrada. Los casos de prueba se seleccionan de modo que se revise a la vez el número más grande de atributos de una clase de equivalencia.

Cita:

"Una forma efectiva para probar código es revisarlo en sus fronteras naturales."

Brian Kernighan

18.6.3 Análisis de valor de frontera

Un mayor número de errores ocurre en las fronteras del dominio de entrada y no en el "centro". Por esta razón es que el *análisis de valor de frontera* (BVA, del inglés *boundary value analysis*) se desarrolló como una técnica de prueba. El análisis de valor de frontera conduce a una selección de casos de prueba que revisan los valores de frontera.

El análisis de valor de frontera es una técnica de diseño de casos de prueba que complementan la partición de equivalencia. En lugar de seleccionar algún elemento de una clase de equivalencia, el BVA conduce a la selección de casos de prueba en los "bordes" de la clase. En lugar de enfocarse exclusivamente en las condiciones de entrada, el BVA también deriva casos de prueba a partir del dominio de salida [Mye79].

Los lineamientos para el BVA son similares en muchos aspectos a los proporcionados para la partición de equivalencia:

PUNTO CLAVE

El BVA extiende la partición de equivalencia al enfocarse en datos en los "bordes" de una clase de equivalencia.

1. Si una condición de entrada especifica un rango acotado por valores *a* y *b*, los casos de prueba deben designarse con valores *a* y *b*, justo arriba y justo abajo de *a* y *b*.
2. Si una condición de entrada especifica un número de valores, deben desarrollarse casos de prueba que revisen los números mínimo y máximo. También se prueban los valores justo arriba y abajo, mínimo y máximo.
3. Aplicar lineamientos 1 y 2 a condiciones de salida. Por ejemplo, suponga que como salida de un programa de análisis de ingeniería se requiere una tabla de temperatura contra presión. Deben diseñarse casos de prueba para crear un reporte de salida que produzca el número máximo (y mínimo) permisible de entradas de tabla.
4. Si las estructuras de datos de programa internos tienen fronteras prescritas (por ejemplo, una tabla que tenga un límite definido de 100 entradas), asegúrese de diseñar un caso de prueba para revisar la estructura de datos en su frontera.

La mayoría de los ingenieros de software realizan intuitivamente BVA en cierta medida. Al aplicar dichos lineamientos, la prueba de fronteras será más completa y, por tanto, tendrá una mayor probabilidad de detectar errores.

18.6.4 Prueba de arreglo ortogonal

Existen muchas aplicaciones en las cuales el dominio de entrada es relativamente limitado, es decir, el número de parámetros de entrada es pequeño y los valores que cada uno de los parámetros puede tomar están claramente acotados. Cuando dichos números son muy pequeños (por ejemplo, tres parámetros de entrada que toman tres valores discretos cada uno), es posible considerar cada permutación de entrada y probar de manera exhaustiva el dominio de entrada. Sin embargo, conforme crece el número de valores de entrada y el número de valores discretos para cada ítem de datos, la prueba exhaustiva se vuelve impráctica o imposible.

La *prueba de arreglo ortogonal* puede aplicarse a problemas en los que el dominio de entrada es relativamente pequeño pero demasiado grande para alojar la prueba exhaustiva. El método de prueba de arreglo ortogonal es particularmente útil para encontrar los *fallos de región*, una categoría de error asociada con lógica defectuosa dentro de un componente de software.

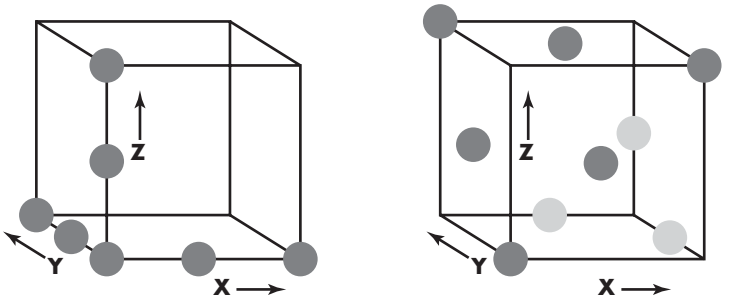
PUNTO CLAVE

La prueba de arreglo ortogonal permite diseñar casos de prueba que proporcionan cobertura máxima de prueba con un número razonable de casos de prueba.

Para ilustrar la diferencia entre prueba de arreglo ortogonal y enfoques más convencionales del tipo "un ítem de entrada a la vez", piense en un sistema que tiene tres ítems de entrada, *X*, *Y* y *Z*. Cada uno tiene tres valores discretos asociados consigo. Existen $3^3 = 27$ posibles casos de prueba. Phadke [Pha97] sugiere una visión geométrica de los posibles casos de prueba asociados con X, Y y Z, que se ilustra en la figura 18.9. En la figura, un ítem de entrada a la vez puede variar en secuencia a lo largo de cada eje de entrada. Esto da como resultado cobertura relativamente limitada del dominio de entrada (representado por el cubo de la izquierda en la figura).

Cuando ocurre la prueba de arreglo ortogonal, se crea un *arreglo ortogonal* L9 de casos de prueba. El arreglo ortogonal L9 tiene una "propiedad de equilibrio" [Pha97]. Es decir, los casos de prueba (representados con puntos oscuros en la figura) se "dispersan de manera uniforme a lo largo de todo el dominio de prueba", como se ilustra en el cubo de la derecha en la figura 18.9. La cobertura de prueba a través del dominio de entrada es más completa.

FIGURA 18.9
Visión geométrica de casos de prueba
Fuente: [Pha97]



Para ilustrar el uso del arreglo ortogonal L9, considere la función *send* para una aplicación de fax. A la función *send* pasan cuatro parámetros: P1, P2, P3 y P4. Cada uno toma tres valores discretos. Por ejemplo, P1 toma los valores:

P1 = 1, enviar ahora
P1 = 2, enviar una hora más tarde
P1 = 3, enviar después de medianoche

P2, P3 y P4 también tomarían los valores de 1, 2 y 3, que significan otras funciones de envío.

Si se eligiera la estrategia de prueba "un ítem de entrada a la vez", la siguiente secuencia de pruebas (P1, P2, P3, P4) se especificaría: (1, 1, 1, 1), (2, 1, 1, 1), (3, 1, 1, 1), (1, 2, 1, 1), (1, 3, 1, 1), (1, 1, 2, 1), (1, 1, 3, 1), (1, 1, 1, 2) y (1, 1, 1, 3). Phadke [Pha97] valora estos casos de prueba al afirmar:

> Tales casos de prueba son útiles sólo cuando uno está seguro de que estos parámetros de prueba no interactúan. Pueden detectar fallas lógicas donde un solo valor de parámetro genere mal funcionamiento del software. Estas fallas se llaman *fallos de modo individual*. Este método no puede detectar fallos lógicos que causen mal funcionamiento cuando dos o más parámetros toman simultáneamente ciertos valores; es decir, no pueden detectar todas las interacciones. Por tanto, su habilidad para detectar fallas es limitada.

Dado el número relativamente pequeño de los parámetros de entrada y de los valores discretos, es posible la prueba exhaustiva. El número de pruebas requeridas es $3^4 = 81$, grande pero manejable. Se encontrarían todos los fallos asociados con la permutación de ítems de datos, pero el esfuerzo requerido es relativamente elevado.

El enfoque de prueba de arreglo ortogonal permite proporcionar una buena cobertura de pruebas con muchos menos casos de prueba que la estrategia exhaustiva. En la figura 18.10 se ilustra un arreglo ortogonal L9 para la función *send* de fax.

Phadke [Pha97] valora el resultado de las pruebas usando el arreglo ortogonal L9 en la siguiente forma:

> **Detectar y aislar todos los fallos de modo individual.** Un fallo de modo individual es un problema congruente con cualquier nivel de cualquier parámetro individual. Por ejemplo, si todos los casos de prueba del factor P1 = 1 causan una condición de error, se trata de una falla de modo individual. En este ejemplo, las pruebas 1, 2 y 3 [figura 18.10] mostrarán errores. Al analizar la información acerca de qué pruebas muestran errores, uno puede identificar cuáles valores de parámetro causan el fallo. En este ejemplo, al notar que las pruebas 1, 2 y 3 causan un error, uno puede aislar [procesamiento lógico asociado con "enviar ahora" (P1 = 1)] la fuente del error. Tal aislamiento del fallo es importante para corregirlo.
>
> **Detectar todos los fallos de modo doble.** Si existe un problema consistente cuando ocurren en conjunto niveles específicos de dos parámetros, se le llama *fallo de modo doble*. De hecho, un fallo

FIGURA 18.10
Un arreglo ortogonal L9

Caso de prueba	Parámetros de prueba			
	P1	P2	P3	P4
1	1	1	1	1
2	1	2	2	2
3	1	3	3	3
4	2	1	2	3
5	2	2	3	1
6	2	3	1	2
7	3	1	3	2
8	3	2	1	3
9	3	3	2	1

de modo doble es indicio de incompatibilidad pareada o de interacciones dañinas entre dos parámetros de prueba.

Fallos multimodo. Los arreglos ortogonales [del tipo mostrado] sólo pueden garantizar la detección de fallos de modo individual y doble. No obstante, muchos fallos multimodo también son detectables por estas pruebas.

En [Pha89] se puede encontrar un análisis detallado de la prueba de arreglo ortogonal.

HERRAMIENTAS DE SOFTWARE

Diseño de casos de prueba

Objetivo: Auxiliar al equipo de software en el desarrollo de un conjunto completo de casos de prueba tanto para prueba de caja negra como de caja blanca.

Mecánica: Estas herramientas se clasifican en dos categorías amplias: las herramientas de prueba estáticas y las herramientas de prueba dinámicas. En la industria se utilizan tres diferentes tipos de herramientas de prueba estáticas: herramientas de prueba basadas en código, lenguajes de prueba especializados y herramientas de prueba basadas en requerimientos. Las primeras aceptan código fuente como entrada y realizan algunos análisis que dan como resultado la generación de casos de prueba. Los lenguajes de prueba especializados (por ejemplo, ATLAS) permiten al ingeniero de software escribir especificaciones de prueba detalladas que describen cada caso de prueba y la logística para su ejecución. Las herramientas de prueba basadas en requerimientos aíslan requerimientos de usuario específicos y sugieren casos de prueba (o clases de pruebas) que revisarán los requerimientos. Las herramientas de prueba dinámicas interactúan con un programa en ejecución, comprueban la cobertura de ruta, prueban las afirmaciones acerca del valor de variables específicas e instrumentan el flujo de ejecución del programa.

Herramientas representativas:[6]

McCabeTest, desarrollada por McCabe & Associates (**www.mccabe.com**), implementa una variedad de técnicas de prueba de trayectoria derivadas de una valoración de complejidad ciclomática y de otras mediciones de software.

TestWorks, desarrollada por Software Research, Inc. (**www.soft.com/Products**), es un conjunto completo de herramientas de prueba automatizadas que auxilia en el diseño de casos de prueba para software desarrollado en C/C++ y Java, y que proporciona apoyo para pruebas de regresión.

T-VEC Test Generation System, desarrollada por T-VEC Technologies (**www.t-vec.com**), es un conjunto de herramientas que soportan pruebas de unidad, integración y validación al asistir en el diseño de casos de prueba, usando la información contenida en una especificación de requerimientos OO.

e-TEST Suite, desarrollada por Empirix, Inc. (**www.empirix.com**), abarca un conjunto completo de herramientas para probar *webapps*, incluidas herramientas que auxilian en el diseño de casos de prueba y planificación de pruebas.

6 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas que existen en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

18.7 PRUEBA BASADA EN MODELO

Cita:

"Es suficientemente difícil encontrar un error en el código cuando se le busca; pero es todavía más difícil cuando se supone que el código está libre de errores."

Steve McConnell

La *prueba basada en modelo* (PBM) es una técnica de prueba de caja negra que usa la información contenida en el modelo de requerimientos como la base para la generación de casos de prueba. En muchos casos, la técnica de prueba basada en modelo usa diagramas de estado UML, un elemento del modelo de comportamiento (capítulo 7), como la base para el diseño de los casos de prueba.[7] La técnica PBM requiere cinco pasos:

1. **Analizar un modelo de comportamiento existente para el software o crear uno.** Recuerde que un *modelo de comportamiento* indica cómo responderá el software a los eventos o estímulos externos. Para crear el modelo, debe realizar los pasos expuestos en el capítulo 7: 1) evaluar todos los casos de uso para comprender por completo la secuencia de interacción dentro del sistema, 2) identificar los eventos que impulsen la secuencia de interacción y entender cómo dichos eventos se relacionan con objetos específicos, 3) crear una secuencia para cada caso de uso, 4) construir un diagrama de estado UML para el sistema (por ejemplo, véase la figura 7.6), y 5) revisar el modelo de comportamiento para verificar precisión y congruencia.
2. **Recorrer el modelo de comportamiento y especificar las entradas que forzarán al software a realizar la transición de estado a estado.** Las entradas dispararán eventos que harán que ocurra la transición.
3. **Revisar el modelo de comportamiento y observar las salidas esperadas, conforme el software realiza la transición de estado a estado.** Recuerde que cada transición de estado se dispara mediante un evento y que, como consecuencia de la transición, se invoca alguna función y se crean salidas. Para cada conjunto de entradas (casos de prueba) especificado en el paso 2, las salidas esperadas se especifican como se caracterizan en el modelo de comportamiento. "Una suposición fundamental de esta prueba es que existe cierto mecanismo, un *oráculo de prueba*, que determinará si los resultados de una prueba de ejecución son o no correctos" [DAC03]. En esencia, un oráculo de prueba establece la base para cualquier determinación de lo correcto de la salida. En la mayoría de los casos, el oráculo es el modelo de requerimientos, pero también podría ser otro documento o aplicación, datos registrados en cualquier otro lado o, incluso, un experto humano.
4. **Ejecutar los casos de prueba.** Las pruebas pueden ejecutarse manualmente o crearse y ejecutarse un guión de prueba usando una herramienta de prueba.
5. **Comparar los resultados reales y esperados y adoptar una acción correctiva según se requiera.**

La PBM ayuda a descubrir errores en el comportamiento del software y, como consecuencia, es extremadamente útil cuando se prueban aplicaciones impulsadas por un evento.

18.8 PRUEBA PARA ENTORNOS, ARQUITECTURAS Y APLICACIONES ESPECIALIZADOS

En ocasiones, los lineamientos y enfoques únicos para pruebas se garantizan cuando se consideran entornos, arquitecturas y aplicaciones especializados. Aunque las técnicas de prueba estudiadas anteriormente en este capítulo, y en los capítulos 19 y 20, con frecuencia pueden

7 La prueba basada en modelo también puede usarse cuando los requerimientos del software se representan con tablas de decisión, gramáticas o cadenas de Markov [DAC03].

adaptarse a situaciones especializadas, vale la pena considerar individualmente sus necesidades únicas.

18.8.1 Pruebas de interfaces gráficas de usuario

Las interfaces gráficas para usuario (GUI, por sus siglas en inglés) presentan interesantes retos de prueba. Puesto que los componentes reutilizables ahora son parte común en los entornos de desarrollo GUI, la creación de la interfaz para el usuario se ha vuelto menos consumidora de tiempo y más precisa (capítulo 11). Pero, al mismo tiempo, la complejidad de las GUI ha crecido, lo que conduce a más dificultad en el diseño y ejecución de los casos de prueba.

Debido a que muchas GUI modernas tienen la misma apariencia y ambiente, puede derivarse una serie de pruebas estándar. Es posible usar las gráficas de modelado de estado finito para derivar una serie de pruebas que aborden objetos de datos y programa específicos que sean relevantes para la GUI. Esta técnica de prueba basada en modelo se estudió en la sección 18.7.

Como producto del gran número de permutaciones asociadas con las operaciones de la GUI, la prueba de GUI debe abordarse usando herramientas automatizadas. Durante los últimos años apareció una amplia gama de herramientas de prueba GUI.[8]

18.8.2 Prueba de arquitecturas cliente-servidor

Cita:

"El tema de las pruebas es un área en la que existe una buena cantidad de comunión entre los sistemas tradicionales y los sistemas cliente-servidor."

Kelley Bourne

La naturaleza distribuida de los entornos cliente-servidor, los conflictos de rendimiento asociados con el procesamiento de transacciones, la potencial presencia de algunas plataformas de hardware diferentes, las complejidades de la comunicación en red, la necesidad de atender a múltiples clientes desde una base de datos centralizada (o en algunos casos, distribuida) y los requerimientos de coordinación impuestos al servidor se combinan para realizar las pruebas de las arquitecturas cliente-servidor y el software que reside dentro de ellas es considerablemente más difícil que las aplicaciones independientes. De hecho, estudios industriales recientes indican un aumento significativo en el tiempo y costo de las pruebas cuando se desarrollan los entornos cliente-servidor.

WebRef

En **www.csst-technologies.com** pueden encontrarse información y recursos útiles para pruebas cliente-servidor.

En general, la prueba del software cliente-servidor ocurre en tres niveles diferentes: 1) las aplicaciones cliente individuales se prueban en un modo "desconectado"; no se considera la operación del servidor ni la red subyacente. 2) El software cliente y las aplicaciones servidor asociadas se prueban en concierto, pero las operaciones de red no se revisan de manera explícita. 3) Se prueba la arquitectura cliente-servidor completa, incluidos la operación de red y el rendimiento.

Aunque en cada uno de estos niveles de detalle se realizan muchos tipos de pruebas diferentes, para las aplicaciones cliente-servidor se encuentran comúnmente los siguientes abordajes de prueba:

? ¿Qué tipos de pruebas se llevan a cabo para los sistemas cliente-servidor?

- **Pruebas de función de aplicación.** La funcionalidad de las aplicaciones cliente se prueba usando los métodos analizados anteriormente en este capítulo y en los capítulos 19 y 20. En esencia, la aplicación se prueba en forma independiente con la intención de descubrir errores en su operación.
- **Pruebas de servidor.** Se prueban las funciones de coordinación y gestión de datos del servidor. También se considera el rendimiento del servidor (tiempo de respuesta global y cantidad de datos transmitidos).
- **Pruebas de base de datos.** Se prueban la precisión y la integridad de los datos almacenados por el servidor. Se examinan las transacciones colocadas por las aplicaciones

8 Cientos, si no miles, de recursos acerca de herramientas de prueba GUI pueden evaluarse en la web. Un buen punto de partida para herramientas de fuente abierta es **www.opensourcetesting.org/functional.php**.

cliente para asegurar que los datos se almacenen, actualicen y recuperen de manera adecuada. También se prueba la forma de archivar.

- **Pruebas de transacción.** Se crea una serie de pruebas para garantizar que cada clase de transacciones se procese de acuerdo con los requerimientos. Las pruebas se enfocan en comprobar lo correcto del procesamiento y también en los conflictos de rendimiento (por ejemplo, tiempos de procesamiento de transacción y volumen de transacción).
- **Pruebas de comunicación de red.** Estas pruebas verifican que la comunicación entre los nodos de la red ocurre de manera correcta y que el mensaje que pasa, las transacciones y el tráfico de red relacionado ocurren sin errores. Como parte de estas pruebas, también pueden realizarse pruebas de seguridad de red.

Para lograr estos abordajes de prueba, Musa [Mus93] recomienda el desarrollo de *perfiles operativos* derivados de escenarios de uso cliente-servidor.[9] Un perfil operativo indica cómo interactúan con el sistema cliente-servidor diferentes tipos de usuarios. Es decir, los perfiles proporcionan un "patrón de uso" que puede aplicarse cuando las pruebas se diseñan y ejecutan. Por ejemplo, para un tipo particular de usuario, ¿qué porcentaje de transacciones serán consultas?, ¿cuántas serán actualizaciones?, ¿cuántos serán pedidos?

Para desarrollar el perfil operativo, es necesario derivar un conjunto de escenarios que sean similares a los casos de uso (capítulos 5 y 6). Cada escenario aborda quién, dónde, qué y por qué. Es decir: quién es el usuario, dónde (en la arquitectura cliente-servidor física) ocurre la interacción del sistema, cuál es la transacción y por qué ocurre. Los escenarios pueden derivarse usando técnicas de respuesta a requerimientos (capítulo 5) o a través de análisis menos formales con los usuarios finales. Sin embargo, el resultado debe ser el mismo. Cada escenario debe proporcionar un indicio de las funciones del sistema que se requerirán para atender a un usuario particular, el orden en el que se requieren dichas funciones, la temporización y la respuesta que se espera, así como la frecuencia con la que se usa cada función. Luego, estos datos se combinan (para todos los usuarios) a fin de crear el perfil operativo. En general, el esfuerzo de prueba y el número de casos de prueba por ejecutar se asignan a cada escenario de uso con base en la frecuencia de uso y en lo crítico de las funciones realizadas.

18.8.3 Documentación de prueba y centros de ayuda

El término *prueba de software* invoca imágenes de gran número de casos de prueba preparados para revisar los programas de cómputo y los datos que manipulan. Al recordar la definición de software que se presentó en el capítulo 1, es importante notar que las pruebas también deben extenderse al tercer elemento de la configuración del software: la documentación.

Los errores en la documentación pueden ser tan devastadores para la aceptación del programa como los errores en los datos o en el código fuente. Nada es más frustrante que seguir con exactitud una guía de usuario o un centro de ayuda en línea y obtener resultados o comportamientos que no coinciden con los predichos por la documentación. Por esta razón, las pruebas de documentación deben ser parte significativa de todo plan de prueba de software.

La prueba de documentación puede abordarse en dos fases. La primera, la revisión técnica (capítulo 15), examina el documento en su claridad editorial. La segunda, prueba en vivo, usa la documentación en conjunto con el programa real.

Sorprendentemente, una prueba en vivo para la documentación puede abordarse usando técnicas que son análogas a muchos de los métodos de prueba de caja negra estudiados anteriormente. La prueba basada en gráfico puede usarse para describir el uso del programa; la partición de equivalencia y el análisis del valor de frontera pueden usarse para definir varias

9 Debe señalarse que los perfiles operativos pueden usarse para probar todo tipo de arquitecturas de sistema, no sólo arquitectura cliente-servidor.

clases de entrada e interacciones asociadas. La PBM puede usarse para garantizar que el comportamiento documentado y el comportamiento real coinciden. Entonces, el uso del programa puede rastrearse a través de la documentación.

INFORMACIÓN

Pruebas de documentación

Las siguientes preguntas deben responderse durante las pruebas de documentación y/o en el centro de ayuda:

- ¿La documentación describe con precisión cómo lograr cada modo de uso?
- ¿La descripción de cada secuencia de interacción es precisa?
- ¿Los ejemplos son precisos?
- ¿La terminología, descripciones de menú y respuestas del sistema son consistentes con el programa real?
- ¿Es relativamente fácil localizar guías dentro de la documentación?
- ¿La solución de problemas puede lograrse con facilidad usando la documentación?
- ¿La tabla de contenido y el índice del documento son consistentes, precisos y completos?
- ¿El diseño del documento (plantilla, fuentes, sangrías, gráficos) contribuye a comprender y asimilar rápidamente la información?
- ¿Todos los mensajes de error del software que se muestran al usuario se describen con más detalle en el documento? ¿Las acciones por tomar como consecuencia de un mensaje de error se delinean con claridad?
- Si se usan enlaces de hipertexto, ¿son precisos y completos?
- Si se usa hipertexto, ¿el diseño de navegación es apropiado para la información requerida?

La única forma viable para responder estas preguntas es hacer que una tercera parte independiente (por ejemplo, usuarios seleccionados) pruebe la documentación en el contexto del uso del programa. Todas las discrepancias se anotan y las áreas de ambigüedad o debilidad en el documento se definen para su potencial reescritura.

18.8.4 Prueba para sistemas de tiempo real

La naturaleza asíncrona, dependiente del tiempo de muchas aplicaciones de tiempo real, agrega un nuevo y potencialmente difícil elemento a la mezcla de pruebas: el tiempo. El diseñador de casos de prueba no sólo debe considerar los casos de prueba convencionales, sino también la manipulación de eventos (es decir, el procesamiento de interrupciones), la temporización de los datos y el paralelismo de las tareas (procesos) que manejan los datos. En muchas situaciones, probar los datos proporcionados cuando un sistema de tiempo real está en un estado dará como resultado un procesamiento adecuado, mientras que los mismos datos proporcionados cuando el sistema está en un estado diferente pueden conducir a error.

Por ejemplo, el software de tiempo real que controla una nueva fotocopiadora acepta interrupciones del operador (es decir, el operador de la máquina presiona teclas de control como RESET o DARKEN) sin error cuando la máquina saca copias (en el estado "copying"). Estas mismas interrupciones del operador, si se ingresan cuando la máquina está en el estado "jammed", generan una pantalla del código de diagnóstico que indica la ubicación del atasco que se tiene que resolver (un error).

Además, la íntima relación que existe entre el software de tiempo real y su entorno de hardware también puede causar problemas en las pruebas. Las pruebas del software deben considerar el impacto de los fallos de hardware en el procesamiento del software. Tales fallos pueden ser extremadamente difíciles de simular de manera realista.

Los métodos amplios de diseño de casos de prueba para sistemas en tiempo real continúan evolucionando. Sin embargo, puede proponerse una estrategia global de cuatro pasos:

? ¿Cuál es una estrategia efectiva para probar un sistema en tiempo real?

- **Prueba de tareas.** El primer paso en la prueba del software en tiempo real es probar cada tarea de manera independiente. Es decir, las pruebas convencionales se diseñan para cada tarea y se ejecutan independientemente durante dichas pruebas. La prueba de tareas descubre errores en lógica y función, mas no en temporización y comportamiento.

- **Prueba de comportamiento.** Con modelos de sistema creados con herramientas automatizadas, es posible simular el comportamiento de un sistema en tiempo real y examinar su comportamiento como consecuencia de eventos externos. Estas actividades de análisis pueden servir de base para el diseño de los casos de prueba que se realizan cuando se construye el software en tiempo real. Al usar una técnica similar a la partición de equivalencia (sección 18.6.2), los eventos (por ejemplo, interrupciones, señales de control) se categorizan para las pruebas. Por ejemplo, los eventos para la fotocopiadora pueden ser interrupciones del usuario (contador de restablecimiento), interrupciones mecánicas (atasco de papel), interrupciones del sistema (baja de tóner) y modos de fallo (sobrecalentamiento del rodillo). Cada uno se prueba de manera individual y el comportamiento del sistema ejecutable se examina para detectar los errores que ocurren como consecuencia del procesamiento asociado con dichos eventos. El comportamiento del modelo del sistema (desarrollado durante la actividad de análisis) y el software ejecutable pueden compararse para asegurar que actúan en conformidad. Una vez que se prueba cada clase de eventos, éstos se presentan al sistema en orden aleatorio y con frecuencia aleatoria. El comportamiento del software se examina para detectar errores de comportamiento.
- **Prueba intertarea.** Una vez aislados los errores en las tareas individuales y en el comportamiento del sistema, las pruebas se cambian a los errores relacionados con el tiempo. Las tareas asíncronas que se sabe que se comunican mutuamente se prueban con diferentes tasas de datos y carga de procesamiento para determinar si ocurrirán errores de sincronización intertarea. Además, las tareas que se comunican vía cola de mensaje o almacenamiento de datos se prueban para descubrir errores en el tamaño de estas áreas de almacenamiento de datos.
- **Prueba de sistema.** Al integrar software y hardware, se lleva a cabo un amplio rango de pruebas del sistema con la intención de descubrir errores en la interfaz software-hardware. La mayoría de los sistemas en tiempo real procesan las interrupciones. Por tanto, probar la manipulación de estos eventos booleanos es esencial. Al usar el diagrama de estado (capítulo 7), el examinador desarrolla una lista de las posibles interrupciones y del procesamiento que ocurre como consecuencia de las interrupciones. Entonces se diseñan pruebas para valorar las siguientes características del sistema:
  - ¿Las prioridades de interrupción se asignan y manejan de manera adecuada?
  - ¿El procesamiento para cada interrupción se maneja de manera correcta?
  - ¿El rendimiento (por ejemplo, tiempo de procesamiento) de cada procedimiento de manejo de interrupción se apega a los requerimientos?
  - ¿Un alto volumen de interrupciones que llegan en momentos críticos crea problemas en el funcionamiento o en el rendimiento?

Además, las áreas de datos globales que se usan para transferir como parte del procesamiento de interrupción deben probarse a fin de valorar el potencial para la generación de efectos colaterales.

18.9 PATRONES PARA PRUEBAS DE SOFTWARE

WebRef

Un catálogo de patrones de prueba de software puede encontrarse en **www.rbsc.com/pages/TestPatternList.htm**

El uso de patrones como un mecanismo para describir soluciones a problemas de diseño específicos se estudió en el capítulo 12. Pero los patrones también pueden usarse para proponer soluciones a otras situaciones de ingeniería de software; en este caso, prueba del software. Los *patrones de prueba* describen problemas y soluciones de prueba comunes que pueden auxiliar en su tratamiento.

Los patrones de prueba no sólo proporcionan lineamientos útiles conforme comienzan las actividades de prueba; también proporcionan tres beneficios adicionales descritos por Marick [Mar02]:

Los patrones de prueba pueden ayudar al equipo de software a comunicarse de manera más efectiva acerca de las pruebas y comprender mejor las fuerzas que conducen a un enfoque de prueba específico.

1. Proporcionan un vocabulario para quienes solucionan problemas. "Oiga, usted sabe, debemos usar un objeto nulo".
2. Enfocan la atención en las fuerzas que hay detrás de un problema. Esto permite que los diseñadores [de caso de prueba] entiendan mejor cuándo y por qué se aplica una solución.
3. Alientan el pensamiento iterativo. Cada solución crea un nuevo contexto en el que pueden resolverse nuevos problemas.

Aunque estos beneficios son "leves", no deben pasarse por alto. Gran parte de las pruebas del software, incluso durante la década pasada, han sido actividades *ad hoc*. Si los patrones de prueba pueden ayudar a un equipo de software a comunicarse de manera más efectiva acerca de las pruebas, a comprender las fuerzas de motivación que conducen a un enfoque específico para las pruebas y a abordar el diseño de las pruebas como una actividad evolutiva en la que cada iteración resulta en una suite más completa de casos de prueba, entonces los patrones lograron mucho.

Los patrones de prueba se describen en forma muy similar a los patrones de diseño (capítulo 12). En la literatura se han propuesto decenas de patrones de prueba (por ejemplo, [Mar02]). Los siguientes tres (presentados sólo en forma resumida) proporcionan ejemplos representativos:

WebRef

Patrones que describen la organización, eficiencia, estrategia y resolución de problemas de las pruebas pueden encontrarse en **www.testing.com/test-patterns/patterns**.

Nombre del patrón: **PairTesting**

Resumen: Patrón orientado a proceso, **PairTesting** describe una técnica que es análoga a la programación por parejas (capítulo 3) en la que dos examinadores trabajan en conjunto para diseñar y ejecutar una serie de pruebas que pueden aplicarse a actividades de prueba de unidad, integración o validación.

Nombre del patrón: **SeparateTestInterface**

Resumen: Hay necesidad de probar cada clase en un sistema orientado a objetos, incluidas "clases internas" (es decir, clases que no exponen alguna interfaz afuera del componente que los usa). El patrón **SeparateTestInterface** describe cómo crear "una interfaz de prueba que puede usarse para describir pruebas específicas sobre clases que son visibles solamente de manera interna en un componente" [Lan01].

Nombre del patrón: **ScenarioTesting**

Resumen: Una vez realizadas las pruebas de unidad e integración, hay necesidad de determinar si el software se desempeñará en forma que satisfaga a los usuarios. El patrón **ScenarioTesting** describe una técnica para revisar el software desde el punto de vista del usuario. Un fallo en este nivel indica que el software fracasó para satisfacer un requisito visible del usuario [Kan01].

Un análisis amplio de los patrones de prueba está más allá del ámbito de este libro. Si tiene más interés, vea [Bin99] y [Mar02] para información adicional acerca de este importante tema.

18.10 Resumen

El objetivo principal para el diseño de casos de prueba es derivar un conjunto de pruebas que tienen la mayor probabilidad de descubrir errores en el software. Para lograr este objetivo, se usan dos categorías diferentes de técnicas de diseño de caso de prueba: pruebas de caja blanca y pruebas de caja negra.

Las pruebas de caja blanca se enfocan en la estructura de control del programa. Los casos de prueba se derivan para asegurar que todos los enunciados en el programa se ejecutaron al me-

nos una vez durante las pruebas y que todas las condiciones lógicas se revisaron. La prueba de ruta o trayectoria básica, una técnica de caja blanca, usa gráficos de programa (o matrices gráficas) para derivar el conjunto de pruebas linealmente independientes que garantizarán la cobertura del enunciado. Las pruebas de condición y de flujo de datos revisan aún más la lógica del programa, y la prueba de bucles complementa otras técnicas de caja blanca al proporcionar un procedimiento para revisar los bucles de varios grados de complejidad.

Hetzel [Het84] describe las pruebas de caja blanca como "pruebas en lo pequeño". Su implicación es que las pruebas de caja blanca que se consideraron en este capítulo por lo general se aplican a pequeños componentes del programa (por ejemplo, módulos o pequeños grupos de módulos). Las pruebas de caja blanca, por otra parte, amplían el foco y pueden llamarse "pruebas en lo grande".

Las pruebas de caja negra se diseñan para validar los requerimientos funcionales sin considerar el funcionamiento interno de un programa. Las técnicas de prueba de caja negra se enfocan en el dominio de información del software, y derivan casos de prueba mediante la partición de los dominios de entrada y salida de un programa en forma que proporciona cobertura de prueba profunda. La partición de equivalencia divide el dominio de entrada en clases de datos que es probable que revisen una función de software específica. El análisis del valor de frontera sondea la habilidad del programa para manejar datos en los límites de lo aceptable. La prueba de arreglo ortogonal proporciona un método sistemático eficiente para probar sistemas con pequeño número de parámetros de entrada. La prueba basada en modelo usa elementos del modelo de requerimientos para probar el comportamiento de una aplicación.

Los métodos de prueba especializados abarcan un amplio arreglo de capacidades de software y áreas de aplicación. La prueba para interfaces gráficas de usuario, arquitecturas cliente-servidor, documentación y centros de ayuda, y los sistemas en tiempo real requieren cada uno lineamientos y técnicas especializadas.

Con frecuencia, los desarrolladores de software experimentados dicen: "las pruebas nunca terminan, sólo se transfieren de uno [el ingeniero de software] al cliente. Cada vez que el cliente usa el programa, se realiza una prueba". Al aplicar el diseño de casos de prueba, pueden lograrse pruebas más completas y, en consecuencia, descubrir y corregir el mayor número de errores antes de comenzar "las pruebas del cliente".

Problemas y puntos para reflexionar

18.1. Myers [Mye79] usa el siguiente programa como una autovaloración de su habilidad para especificar pruebas adecuadas: un programa lee tres valores enteros. Los tres se interpretan como representación de las longitudes de los lados de un triángulo. El programa imprime un mensaje que indica si el triángulo es escaleno, isósceles o equilátero. Desarrolle un conjunto de casos de prueba que crea que probarán este programa de manera adecuada.

18.2. Diseñe e implemente el programa (con manipulación de error donde sea adecuado) que se especifica en el problema 18.1. Derive un gráfico de flujo para el programa y aplique prueba de ruta básica para desarrollar casos de prueba que garanticen la prueba de todos los enunciados en el programa. Ejecute los casos y muestre sus resultados.

18.3. ¿Puede pensar en algunos objetivos de prueba adicionales que no se estudiaron en la sección 18.1.1?

18.4. Seleccione un componente de software que haya diseñado e implementado recientemente. Diseñe un conjunto de casos de prueba que garantice que todos los enunciados se ejecutan, usando prueba de ruta o trayectoria básica.

18.5. Especifique, diseñe e implemente una herramienta de software que calcule la complejidad ciclomática para el lenguaje de programación de su elección. Use la matriz de grafo como la estructura de datos operativa en su diseño.

18.6. Lea Beizer [Bei95] o una fuente en web relacionada (por ejemplo, **www.laynetworks.com/Discret%20Mathematics_1g.htm**) y determine cómo puede extenderse el programa que desarrolló en el problema 18.5 a fin de alojar varias enlaces ponderados. Extienda su herramienta para procesar probabilidades de ejecución o tiempos de procesamiento de liga.

18.7. Diseñe una herramienta automatizada que reconozca bucles y que los clasifique como se indica en la sección 18.5.3.

18.8. Extienda la herramienta descrita en el problema 18.7 a fin de generar casos de prueba para cada categoría de bucle, una vez encontrada. Será necesario realizar esta función de manera interactiva con el examinador.

18.9. Proporcione al menos tres ejemplos en los que la prueba de caja negra puede dar la impresión de que "todo está bien", mientras que las pruebas de caja blanca pueden descubrir un error. Proporcione al menos tres ejemplos en los que las pruebas de caja blanca pueden dar la impresión de que "todo está bien", mientras que las pruebas de caja negra pueden descubrir un error.

18.10. ¿Las pruebas exhaustivas (incluso si es posible para programas muy pequeños) garantizarán que el programa es 100 por ciento correcto?

18.11. Pruebe un manual de usuario (o centro de ayuda) para una aplicación que use con frecuencia. Encuentre al menos un error en la documentación.

Lecturas adicionales y fuentes de información

Virtualmente, todos los libros dedicados a las pruebas de software consideran tanto estrategia como tácticas. Por tanto, las lecturas adicionales anotadas para el capítulo 17 son igualmente aplicables para este capítulo. Everett y Raymond (*Software Testing,* Wiley-IEEE Computer Society Press, 2007), Black (*Pragmatic Software Testing,* Wiley, 2007), Spiller *et al*. (*Software Testing Process: Test Management,* Rocky Nook, 2007), Perry (*Effective Methods for Software Testing,* 3d. ed., Wiley, 2005), Lewis (*Software Testing and Continuous Quality Improvement,* 2a. ed., Auerbach, 2004), Loveland *et al*. (*Software Testing Techniques,* Charles River Media, 2004), Burnstein (*Practical Software Testing,* Springer, 2003), Dustin (*Effective Software Testing,* Addison-Wesley, 2002), Craig y Kaskiel (*Systematic Software Testing,* Artech House, 2002), Tamres (*Introducing Software Testing,* Addison-Wesley, 2002) y Whittaker (*How to Break Software,* Addison-Wesley, 2002) son sólo una pequeña muestra de muchos libros que analizan los principios, conceptos, estrategias y métodos de las pruebas.

Una segunda edición del texto clásico de Myers [Mye79], producido por Myers *et al.* (*The Art of Software Testing*, 2a. ed., Wiley, 2004), cubre con mucho detalle las técnicas de diseño de casos de prueba. Pezze y Young (*Software Testing and Analysis,* Wiley, 2007), Perry (*Effective Methods for Software Testing,* 3a. ed., Wiley, 2006), Copeland (*A Practitioner's Guide to Software Test Design,* Artech, 2003), Hutcheson (*Software Testing Fundamentals,* Wiley, 2003), Jorgensen (*Software Testing: A Craftsman's Approach,* 2a. ed., CRC Press, 2002) proporcionan cada uno presentaciones útiles de los métodos y técnicas del diseño de casos de prueba. El texto clásico de Beizer [Bei90] proporciona una amplia cobertura de las técnicas de caja blanca e introduce un nivel de rigor matemático que con frecuencia falta en otros tratamientos de las pruebas. Su último libro [Bei95] presenta un tratamiento conciso de métodos importantes.

La prueba del software es una actividad que consume muchos recursos. Es por esto que muchas organizaciones automatizan partes del proceso de prueba. Los libros de Li y Wu (*Effective Software Test Automation,* Sybex, 2004); Mosely y Posey (*Just Enough Software Test Automation,* Prentice-Hall, 2002); Dustin, Rashka, y Poston (*Automated Software Testing: Introduction, Management, and Performance,* Addison-Wesley, 1999); Graham *et al*. (*Software Test Automation*, Addison-Wesley, 1999) y Poston (*Automating Specification-Based Software Testing,* IEEE Computer Society, 1996) exponen herramientas, estrategias y métodos para pruebas automatizadas. Nquyen *et al.* (*Global Software Test Automation*, Happy About Press, 2006) presentan un panorama ejecutivo de la automatización de las pruebas.

Thomas *et al.* (*Java Testing Patterns*, Wiley, 2004) y Binder [Bin99] describen patrones de prueba que abarcan pruebas de métodos, clases/grupos, subsistemas, reutilización de componentes, marcos conceptuales y sistemas, así como la automatización de las pruebas y la prueba de bases de datos especializadas.

En internet está disponible una amplia variedad de recursos de información acerca de los métodos de diseño de casos de pruebas. Una lista actualizada de referencias en la World Wide Web que son relevantes para las técnicas de prueba puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

19

PRUEBA DE APLICACIONES ORIENTADAS A OBJETOS

CONCEPTOS CLAVE

prueba aleatoria 447
prueba basada en escenario . 445
prueba basada en fallo 444
prueba basada en hebra. . . . 442
prueba basada en uso 442
prueba de clase. 441
prueba de clase múltiple . . . 449
prueba de grupo 442
prueba de partición 448

En el capítulo 18 se señaló que el objetivo de las pruebas, dicho de manera simple, es encontrar la mayor cantidad posible de errores con una cantidad manejable de esfuerzo aplicado durante un lapso realista. Aunque este objetivo fundamental permanece invariable para el software orientado a objetos (OO), la naturaleza de los programas OO cambia en la estrategia y en las tácticas de las pruebas.

Podría argumentarse que, conforme las bibliotecas de clase reutilizables crecen en tamaño, un reuso mayor mitigará a los sistemas OO en su necesidad de pruebas pesadas. Lo opuesto es exactamente cierto. Binder [Bin94b] analiza esto cuando afirma:

> Cada reuso es un nuevo contexto de uso y es prudente una nueva comprobación. Parece probable que se necesitarán más pruebas, no menos, para obtener alta confiabilidad en los sistemas orientados a objetos.

Para probar adecuadamente los sistemas OO, deben realizarse tres cosas: 1) ampliar la definición de prueba para incluir las técnicas de descubrimiento de error aplicadas al análisis orientado a objetos y a modelos de diseño, 2) cambiar significativamente la estrategia para prueba de unidad e integración y 3) explicar las características únicas del software OO mediante el diseño de casos de prueba.

UNA MIRADA RÁPIDA

¿Qué es? La arquitectura del software orientado a objetos (OO) da como resultado una serie de subsistemas en capas que encapsulan clases colaboradoras. Cada uno de estos elementos de sistema (subsistemas y clases) realiza funciones que ayudan a lograr los requerimientos del sistema. Es necesario probar un sistema OO en varios niveles diferentes con la intención de descubrir errores que puedan ocurrir conforme las clases colaboran unas con otras y conforme los subsistemas se comunican a través de capas arquitectónicas.

¿Quién lo hace? Ingenieros de software y examinadores especializados realizan la prueba orientada a objetos.

¿Por qué es importante? El programa tiene que ejecutarse antes de que llegue al cliente con la intención específica de remover todos los errores, de modo que el cliente no experimente la frustración que produce encontrarse con un producto de calidad pobre. Con la finalidad de encontrar el mayor número posible de errores, las pruebas deben realizarse de manera sistemática y los casos de prueba deben diseñarse usando técnicas disciplinadas.

¿Cuáles son los pasos? Las pruebas OO son estratégicamente análogas a la prueba de sistemas convencionales, pero tácticamente diferentes. Puesto que el análisis OO y los modelos de diseño son similares en estructura y contenido con el programa OO resultante, las "pruebas" se inician con la revisión de dichos modelos. Una vez generado el código, la prueba OO comienza "en lo pequeño", con las pruebas de clase. Se diseña una serie de pruebas que ejercitan las operaciones de clase y que examinan si existen errores conforme una clase colabora con otras clases. En la medida en la que las clases se integran para formar un subsistema, se aplican pruebas basadas en hebra, en uso y de grupo, junto con enfoques basados en fallo, a fin de ejercitar por completo clases colaboradoras. Finalmente, se usan casos de uso (desarrollados como parte del modelo de requerimientos) para descubrir errores de validación del software.

¿Cuál es el producto final? Se diseña y documenta un conjunto de casos de prueba, diseñados para ejercitar clases, sus colaboraciones y comportamientos; se definen los resultados esperados y se registran los resultados reales.

¿Cómo me aseguro de que lo hice bien? Cuando comienzan las pruebas, cambia el punto de vista. ¡Intente "romper" el software! Diseñe casos de prueba en forma disciplinada y revise con minuciosidad los casos de prueba creados.

19.1 AMPLIACIÓN DE LA DEFINICIÓN DE LAS PRUEBAS

La construcción de software orientado a objetos comienza con la creación de modelos de requerimientos (análisis) y de diseño.[1] Debido a la naturaleza evolutiva del paradigma de ingeniería del software OO, dichos modelos comienzan como representaciones relativamente informales de los requisitos de sistema y evolucionan hacia modelos detallados de clases, relaciones de clase, diseño y asignación de sistema, y diseño de objetos (que incorpora un modelo de conectividad de objetos mediante mensajería). En cada etapa, los modelos pueden "probarse" con la intención de descubrir errores previamente a su propagación hacia la siguiente iteración.

Puede argumentarse que la revisión de los modelos de análisis y de diseño OO es especialmente útil, pues los mismos constructos semánticos (por ejemplo, clases, atributos, operaciones, mensajes) aparecen en los niveles de análisis, diseño y código. Por tanto, un problema en la definición de los atributos de clase que se descubra durante el análisis soslayará los efectos colaterales que puedan ocurrir si el problema no se descubriera hasta el diseño o el código (o incluso en la siguiente iteración de análisis).

Aunque la revisión de los modelos de análisis y diseño OO es parte integral de "la prueba" de una aplicación OO, reconozca que ésta no es suficiente en y por sí misma. También debe realizar pruebas ejecutables.

Por ejemplo, considere una clase en la que se define un número de atributos durante la primera iteración de análisis. Un atributo extraño se anexa a la clase (debido a una mala interpretación del dominio del problema). Entonces pueden especificarse dos operaciones para manipular el atributo. Se lleva a cabo una revisión y un experto en dominio puntualiza el problema. Al eliminar el atributo extraño en esta etapa, durante el análisis pueden evitarse los siguientes problemas y un esfuerzo innecesario:

1. Tal vez se generen subclases especiales para alojar el atributo innecesario o las excepciones. Se evita el trabajo involucrado en la creación de subclases innecesarias.
2. Una mala interpretación de la definición de clase puede conducir a relaciones de clase incorrectas o extrañas.
3. El comportamiento del sistema o sus clases puede caracterizarse de manera inadecuada para alojar el atributo extraño.

Si el problema no se descubre durante el análisis y se propaga aún más, podrían ocurrir los siguientes problemas durante el diseño (que se evitarían con la revisión temprana):

1. Durante el diseño del sistema, puede ocurrir la asignación inadecuada de la clase a un subsistema o a algunas tareas.
2. Puede emplearse trabajo de diseño innecesario a fin de crear el diseño de procedimientos para las operaciones que abordan el atributo extraño.
3. El modelo de mensajería será incorrecto (porque los mensajes deben diseñarse para las operaciones que son extrañas).

Cita:

"Las herramientas que usamos tienen profunda (¡y tortuosa!) influencia sobre nuestros hábitos de pensamiento y, por tanto, sobre nuestras habilidades de pensamiento."

Edsger Dijkstra

Si el problema sigue sin detectarse durante el diseño y pasa hacia la actividad de codificación, se empleará esfuerzo considerable para generar código que implemente un atributo innecesario, dos operaciones innecesarias, mensajes que activen la comunicación interobjetos y muchos otros conflictos relacionados. Además, la prueba de la clase absorberá más tiempo que el necesario. Una vez que finalmente se haya descubierto el problema, la modificación del sistema debe realizarse con el potencial siempre presente de efectos colaterales que se generen por el cambio.

Durante las últimas etapas de su desarrollo, los modelos de análisis (AOO) y de diseño (DOO) orientado a objetos proporcionan información sustancial acerca de la estructura y comporta-

1 Las técnicas de modelado de análisis y diseño se presentan en la parte 2 de este libro. Los conceptos básicos de OO se presentan en el apéndice 2.

miento del sistema. Por esta razón, dichos modelos deben sujetarse a una rigurosa revisión, previa a la generación del código.

Todos los modelos orientados a objetos deben probarse (en este contexto, el término *prueba* incorpora revisiones técnicas) en relación con su exactitud, completitud y consistencia dentro del contexto de la sintaxis, la semántica y la pragmática del modelo [Lin94a].

19.2 Modelos de prueba AOO y DOO

Los modelos de análisis y diseño no pueden probarse de la manera convencional porque no pueden ejecutarse. Sin embargo, pueden usarse revisiones técnicas (capítulo 15) para examinar su exactitud y consistencia.

19.2.1 Exactitud de los modelos AOO y DOO

La notación y la sintaxis utilizadas para representar los modelos de análisis y diseño se ligarán a los métodos de análisis y diseño específicos que se elijan para el proyecto. Por tanto, la exactitud sintáctica se juzga mediante el uso adecuado de la simbología; cada modelo se revisa para garantizar que se mantienen las convenciones de modelado adecuadas.

Durante el análisis y el diseño, la exactitud semántica puede valorarse con base en la conformidad del modelo con el dominio de problemas del mundo real. Si el modelo refleja con precisión el mundo real (en un nivel de detalle que sea apropiado para la etapa de desarrollo en la que se revisó el modelo), entonces es semánticamente correcto. Para determinar si el modelo verdaderamente refleja los requerimientos del mundo real, debe presentarse a expertos de dominio de problemas, quienes examinarán las definiciones y jerarquía de clase en busca de omisiones y ambigüedad. Las relaciones de clase (conexiones de instancia) se evalúan para determinar si reflejan con precisión conexiones de objetos en el mundo real.[2]

19.2.2 Consistencia de los modelos orientados a objetos

La consistencia de los modelos orientados a objetos puede juzgarse al "considerar las relaciones entre entidades en el modelo. Un modelo de análisis o diseño inconsistente tiene representaciones en una parte del modelo que no se reflejan de manera correcta en otras porciones" [McG94].

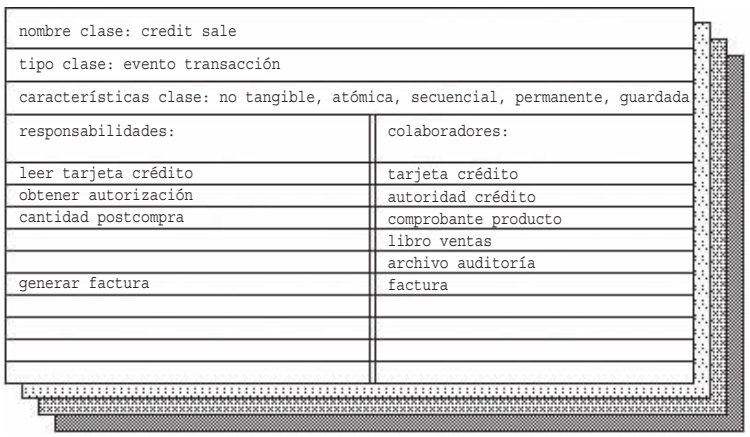
Para valorar la consistencia, debe examinarse cada clase y sus conexiones con otras clases. A fin de facilitar esta actividad, puede usarse el modelo clase-responsabilidad-colaboración (CRC) o un diagrama de objeto-relación. Como se estudió en el capítulo 6, el modelo CRC se compone de tarjetas índice CRC. Cada tarjeta CRC menciona el nombre de la clase, sus responsabilidades (operaciones) y sus colaboradores (otras clases a las que envía mensajes y de las que depende para lograr sus responsabilidades). Las colaboraciones implican una serie de relaciones (es decir, conexiones) entre clases del sistema OO. El modelo objeto-relación proporciona una representación gráfica de las conexiones entre clases. Toda esta información puede obtenerse a partir del modelo de análisis (capítulos 6 y 7).

Para evaluar el modelo de clase, se recomienda seguir los siguientes pasos [McG94]:

1. **Vuelva a consultar el modelo CRC y el modelo objeto-relación**. Realice una comprobación cruzada para garantizar que todas las colaboraciones implicadas por el modelo de requerimientos se reflejan de manera adecuada en ambas.

2 Los casos de uso pueden ser invaluables para cotejar los modelos de análisis y diseño contra escenarios de uso del sistema OO en el mundo real.

FIGURA 19.1
Ejemplo de tarjeta índice CRC utilizada para revisión



2. **Inspeccione la descripción de cada tarjeta índice CRC para determinar si una responsabilidad delegada es parte de la definición del colaborador.** Por ejemplo, considere una clase definida por un sistema de comprobación punto de venta y que se llame **CreditSale**. Esta clase tiene una tarjeta índice CRC como la que se ilustra en la figura 19.1.

   Para esta colección de clases y colaboraciones, pregunte si una responsabilidad (por ejemplo, *leer tarjeta crédito*) se cumple delegándola al colaborador mencionado (**CreditCard**). Es decir, ¿la clase **CreditCard** tiene una operación que le permite leerse? En este caso, la respuesta es "sí". El objeto-relación se recorre para garantizar que tales conexiones son válidas.

3. **Invertir la conexión para garantizar que cada colaborador al que se solicita servicio recibe solicitud de una fuente razonable.** Por ejemplo, si la clase **CreditCard** recibe una solicitud para `purchase amount` (cantidad compra) de la clase **CreditSale**, habría un problema. **CreditCard** no conoce la cantidad de compra.

4. **Al usar las conexiones invertidas que se examinaron en el paso 3, se determina si es posible requerir otras clases o si las responsabilidades se agrupan de manera adecuada entre las clases.**

5. **Determinar si las responsabilidades de amplia solicitud pueden combinarse en una sola responsabilidad**. Por ejemplo, *leer tarjeta de crédito* y *obtener autorización* ocurren en toda situación. Pueden combinarse en una responsabilidad *validar solicitud crédito* que incorpora obtener el número de tarjeta de crédito y conseguir la autorización.

Los pasos del 1 al 5 deben aplicarse de manera iterativa a cada clase y a lo largo de cada evolución del modelo de requerimientos.

Una vez creado el modelo de diseño (capítulos del 9 al 11), también deben realizarse revisiones del diseño del sistema y del diseño del objeto. El diseño del sistema bosqueja la arquitectura de producto global, los subsistemas que abarca el producto, la forma en la que los subsistemas se asignan a los procesadores, la asignación de clases a los subsistemas y el diseño de la interfaz de usuario. El modelo de objetos presenta los detalles de cada clase y las actividades de mensajería que se necesitan para implementar las colaboraciones entre clases.

El diseño del sistema se revisa al examinar el modelo de comportamiento del objeto desarrollado durante el análisis y el mapeo orientado a objetos requerido por el comportamiento del

sistema contra los subsistemas diseñados para lograr este comportamiento. La concurrencia y la asignación de tarea también se revisan en el contexto del comportamiento del sistema. Los estados de comportamiento del sistema se evalúan para determinar cuál existe de manera concurrente. Los casos de uso se utilizan para ejercitar el diseño de la interfaz de usuario.

El modelo de objetos debe ponerse a prueba contra la red de relación de objetos a fin de asegurar que todos los objetos diseñados contienen los atributos y operaciones necesarios para implementar las colaboraciones definidas para cada tarjeta índice CRC. Además, se revisa la especificación minuciosa de los detalles de operación (es decir, los algoritmos que implementan las operaciones).

19.3 Estrategias de pruebas orientadas a objetos

Como se anotó en el capítulo 18, la estrategia clásica de prueba de software comienza "probando en lo pequeño" y funciona hacia afuera, "probando en lo grande". Dicho en el lenguaje de las pruebas de software (capítulo 18), comienza con la *prueba de unidad*, luego avanza hacia la *prueba de integración* y culmina con las *pruebas de validación y sistema*. En aplicaciones convencionales, la prueba de unidad se enfoca en la unidad de programa compatible más pequeña: el subprograma (por ejemplo, componente, módulo, subrutina, procedimiento). Una vez que cada una de estas unidades se prueba de manera individual, se integra en una estructura de programa mientras se aplica una serie de pruebas de regresión para descubrir errores debidos a la puesta en interfaz de los módulos y los efectos colaterales que se generan al sumar nuevas unidades. Finalmente, el sistema como un todo se prueba para garantizar que se descubren los errores en los requerimientos.

19.3.1 Prueba de unidad en el contexto OO

La "unidad" comprobable más pequeña en el software OO es la clase. La prueba de clase se activa mediante las operaciones encapsuladas por la clase y por el comportamiento de estado de la misma.

Cuando se piensa en software orientado a objetos, cambia el concepto de unidad. La encapsulación impulsa la definición de clases y objetos. Esto significa que cada clase y cada instancia de una clase (objeto) encapsulan los atributos (datos) y las operaciones (también conocidas como métodos o servicios) que manipulan dichos datos. En lugar de probar un módulo individual, la unidad comprobable más pequeña es la clase encapsulada. Puesto que una clase puede contener algunas operaciones diferentes y una operación particular puede existir como parte de un número de clases diferentes, el significado de prueba de unidad cambia dramáticamente.

Ya no es posible probar una sola operación aislada (la visión convencional de la prueba de unidad) sino, más bien, como parte de una clase. Para ilustrar lo anterior, considere una jerarquía de clase en la que se define una operación *X( )* para la superclase y la heredan algunas subclases. Cada subclase usa la operación *X( )*, pero se aplica dentro del contexto de los atributos y operaciones privados que se definieron para cada subclase. Puesto que el contexto donde se usa la operación *X( )* varía en formas sutiles, es necesario probarla en el contexto de cada una de las subclases. Esto significa que probar la operación *X( )* en el vacío (el enfoque tradicional de la prueba de unidad) no es efectivo en el contexto orientado a objetos.

La prueba de clase para el software OO es el equivalente de la prueba de unidad para software convencional.[3] A diferencia de la prueba de unidad del software convencional, que tiende a enfocarse en el detalle algorítmico de un módulo y en los datos que fluyen a través de la interfaz de módulo, la prueba de clase para el software OO se activa mediante las operaciones encapsuladas por la clase y por el comportamiento de estado de la misma.

3 En las secciones 19.4 a 19.6 se estudian los métodos de diseño de casos de prueba para clases OO.

19.3.2 Prueba de integración en el contexto OO

Puesto que el software orientado a objetos no tiene una estructura de control jerárquica, las estrategias de integración tradicionales, descendente y ascendente, tienen poco significado. Además, integrar operaciones una a la vez en una clase (el enfoque de integración incremental convencional) con frecuencia es imposible debido a las "interacciones directas e indirectas de los componentes que constituyen la clase" [Ber93].

PUNTO CLAVE

La prueba de integración para software OO examina un conjunto de clases que se requieren para responder a un evento dado.

Existen dos diferentes estrategias para la prueba de integración de los sistemas OO [Bin94a]. La primera, *prueba basada en hebra*, integra el conjunto de clases requeridas para responder a una entrada o evento del sistema. Cada hebra se integra y prueba de manera individual. La prueba de regresión se aplica para asegurar que no ocurran efectos colaterales. El segundo enfoque de integración, *prueba basada en uso*, comienza la construcción del sistema al probar aquellas clases (llamadas *independientes*) que usan muy pocas clases de servidor (si es que emplean alguna). Después de probar las clases independientes, se examina la siguiente capa de clases que usan las clases independientes, llamadas *dependientes*. Esta secuencia de pruebas para las capas de clases dependientes continúa hasta que se construye todo el sistema. A diferencia de la integración convencional, cuando sea posible debe evitarse el uso de controladores y representantes (proxies) (capítulo 18) como operaciones de reemplazo.

La *prueba de grupo* [McG94] es un paso en la prueba de integración del software OO. En ella, se ejercita un grupo de clases colaboradoras (determinadas al examinar el CRC y el modelo objeto-relación) al diseñar casos de prueba que intentan descubrir errores en las colaboraciones.

19.3.3 Prueba de validación en un contexto OO

En el nivel de validación o de sistema, desaparecen los detalles de las conexiones de clase. Como la validación convencional, la del software OO se enfoca en las acciones visibles para el usuario y en las salidas del sistema reconocibles por él mismo. Para auxiliar en la derivación de pruebas de validación, el examinador debe recurrir a casos de uso (capítulos 5 y 6) que sean parte del modelo de requerimientos. El caso de uso proporciona un escenario que tiene una alta probabilidad de descubrir errores en los requerimientos de interacción de usuario.

Los métodos convencionales de prueba de caja negra (capítulo 18) pueden usarse para activar pruebas de validación. Además, puede elegirse derivar casos de prueba del modelo de comportamiento del objeto y de un diagrama de flujo de evento creado como parte del AOO.

19.4 MÉTODOS DE PRUEBA ORIENTADA A OBJETOS

Cita:

"Veo a los examinadores como los guardaespaldas del proyecto. Defendemos del fallo el flanco de nuestros desarrolladores, mientras ellos se enfocan en crear éxito."

James Bach

La arquitectura del software orientado a objetos da como resultado una serie de subsistemas en capas que encapsulan clases colaboradoras. Cada uno de estos elementos de sistema (subsistemas y clases) realiza funciones que ayudan a lograr requerimientos de sistema. Es necesario probar un sistema OO en varios niveles diferentes con la intención de descubrir errores que puedan ocurrir conforme las clases colaboran unas con otras y conforme los subsistemas se comunican a través de las capas arquitectónicas.

Los métodos de diseño de casos de prueba para el software orientado a objetos siguen evolucionando. Sin embargo, Berard [Ber93] sugiere un enfoque global en el diseño de casos de prueba OO:

1. Cada caso de prueba debe identificarse de manera única y explícita asociado con la clase que se va a probar.
2. Debe establecerse el propósito de la prueba.

3. Debe desarrollarse una lista de pasos de prueba para cada una de ellas, que debe contener:
   a. Una lista de estados especificados para la clase que se probará
   b. Una lista de mensajes y operaciones que se ejercitarán como consecuencia de la prueba
   c. Una lista de excepciones que pueden ocurrir conforme se prueba la clase
   d. Una lista de condiciones externas (es decir, con la finalidad de realizar adecuadamente las pruebas, cambios en el entorno externo al software que debe existir)
   e. Información complementaria que ayudará a comprender o a implementar la prueba

A diferencia del diseño convencional de casos de prueba, que se activan mediante una visión entrada-proceso-salida del software o con el detalle algorítmico de módulos individuales, la prueba orientada a objetos se enfoca en el diseño de secuencias apropiadas de operaciones para ejercitar los estados de una clase.

19.4.1 Implicaciones del diseño de casos de prueba de los conceptos OO

WebRef

Se puede encontrar una excelente serie de documentos y recursos sobre pruebas OO en **www.rbsc.com**.

Conforme una clase evoluciona a través de los modelos de requerimientos y diseño, se convierte en un blanco para el diseño de casos de prueba. Puesto que los atributos y las operaciones están encapsulados, por lo general es improductivo probar operaciones afuera de la clase. Aunque la encapsulación es un concepto de diseño esencial para OO, puede crear un obstáculo menor cuando se prueba. Como anota Binder [Bin94a]: "las pruebas requieren reportar el estado concreto y abstracto de un objeto". No obstante, la encapsulación puede hacer que esta información sea un poco difícil de obtener. A menos que se proporcionen operaciones internas a fin de reportar los valores para los atributos de clase, puede ser difícil adquirir una instantánea del estado de un objeto.

La herencia también puede presentar retos adicionales durante el diseño de casos de prueba. Ya se anotó que cada nuevo contexto de uso requiere un nuevo examen, aun cuando se haya logrado el reuso. Además, la herencia múltiple[4] complica la prueba todavía más al aumentar el número de contextos para los cuales se requiere la prueba [Bin94a]. Si dentro del mismo dominio de problema se usan subclases instanciadas de una superclase, es probable que el conjunto de casos de prueba derivados para la superclase pueda usarse cuando se prueba la subclase. Sin embargo, si la superclase se usa en un contexto completamente diferente, los casos de prueba de superclase tendrán poca aplicabilidad y debe diseñarse un nuevo conjunto de pruebas.

19.4.2 Aplicabilidad de los métodos convencionales de diseño de casos de prueba

Los métodos de prueba de caja blanca descritos en el capítulo 18 pueden aplicarse a las operaciones definidas para una clase. Las técnicas de ruta básica, prueba de bucle o flujo de datos pueden ayudar a garantizar que se probaron todos los enunciados en una operación. Sin embargo, la estructura concisa de muchas operaciones de clase hace que algunos argumenten que el esfuerzo aplicado a la prueba de caja blanca puede redirigirse mejor para probar en un nivel de clase.

Los métodos de prueba de caja negra son tan apropiados para los sistemas OO como para los sistemas desarrollados, usando métodos de ingeniería del software convencional. Como se observó en el capítulo 18, los casos de uso pueden proporcionar entrada útil en el diseño de las pruebas de caja negra y en las basadas en estado.

---

4 Un concepto OO que debe usarse con cuidado extremo.

19.4.3 Prueba basada en fallo[5]

PUNTO CLAVE

La estrategia para la prueba basada en fallo es elaborar hipótesis acerca de un conjunto de fallos plausibles y luego derivar pruebas para corroborar o descartar cada hipótesis.

El objeto de la *prueba basada en fallo* dentro de un sistema OO es diseñar pruebas que tengan una alta probabilidad de descubrir fallos plausibles. Puesto que el producto o sistema debe adecuarse a los requerimientos del cliente, la planificación preliminar requerida para realizar alguna prueba basada en fallo comienza con el modelo de análisis. El examinador busca fallos plausibles, es decir, aspectos de la implementación del sistema que pueden resultar en defectos. Para determinar si existen dichos fallos, los casos de prueba se diseñan a fin de ejercitar el diseño o código.

Desde luego, la efectividad de dichas técnicas depende de cómo perciben los examinadores un fallo plausible. Si los fallos reales en un sistema OO se perciben como improbables, entonces este enfoque realmente no es mejor que cualquier técnica de prueba aleatoria. Sin embargo, si los modelos de análisis y diseño pueden proporcionar comprensión acerca de lo que es probable que vaya mal, entonces la prueba basada en fallo puede encontrar un significativo número de errores con gastos de esfuerzo relativamente bajos.

? ¿Qué tipos de fallos se encuentran en los llamados de operación y en las conexiones de mensaje?

La prueba de integración busca fallos plausibles en los llamados de operación y en las conexiones de mensaje. En este contexto se encuentran tres tipos de fallos: resultado inesperado, uso de operación/mensaje equivocado e invocación incorrecta. Para determinar fallos plausibles cuando se invocan funciones (operaciones), debe examinarse el comportamiento de la operación.

La prueba de integración se aplica a los atributos así como a las operaciones. Los "comportamientos" de un objeto están definidos por los valores que le son asignados a sus atributos. Las pruebas deben ejercer los atributos para determinar si los valores adecuados ocurren para los distintos tipos de comportamiento de los objetos.

Es importante observar que la prueba de integración intenta encontrar errores en el objeto cliente, no en el servidor. Dicho en términos convencionales, el foco de la prueba de integración es determinar si existen errores en el código que llama, no en el código llamado. La llamada de operación se usa como pista: es una forma de encontrar requerimientos de prueba que ejerciten el código que llama.

19.4.4 Casos de prueba y jerarquía de clase

La herencia no dispensa la necesidad de pruebas amplias de todas las clases derivadas. De hecho, en realidad puede complicar el proceso de prueba. Considere la siguiente situación. Una clase **Base** contiene operaciones *inherited( )* y *redefined( ).* Una clase **Derived** redefine *redefined( )* para servir en un contexto local. Hay poca duda de que **Derived::redefined( )** tiene que probarse porque representa un nuevo diseño y un nuevo código. Pero, ¿**Derived::inherited( )** debe probarse nuevamente?

PUNTO CLAVE

Aun cuando una clase base se probó ampliamente, todavía tendrá que probar todas las clases derivadas de ella.

Si **Derived::inherited( )** llama a *redefined( )* y el comportamiento de *redefined( )* cambió, **Derived::inherited( )** puede manejar mal el nuevo comportamiento. Por tanto, necesita nuevas pruebas aun cuando el diseño y el código no hayan cambiado. No obstante, es importante observar que es posible que sólo se ejecute un subconjunto de todos las pruebas para **Derived::inherited( )**. Si parte del diseño y código para *inherited( )* no depende de *redefined( )* (es decir, no lo llama ni llama a código alguno que lo llama de manera indirecta), dicho código no necesita probarse de nuevo en la clase derivada.

Base::redefined( ) y **Derived::redefined( )** son dos operaciones diferentes con diferentes especificaciones e implementaciones. Cada una tendrá un conjunto de requerimientos de prueba derivadas de la especificación y la implementación. Dichos requerimientos de prueba sondean fallos plausibles: de integración, de condición, de frontera, etcétera. Pero es probable que las opera-

5 En las secciones 19.4.3 a 19.4.6 se realizó una adaptación de un artículo de Brian Marick publicado en el grupo de noticias de internet llamado comp.testing. Esta adaptación se incluye con el permiso del autor. Para mayor información acerca de estos temas, vea [Mar94]. Debe observarse que las técnicas estudiadas en estas secciones también son aplicables a software convencional.

ciones sean similares. Sus conjuntos de requerimientos de prueba se traslaparán. Mientras mejor sea el diseño OO, mayor es el traslape. Es necesario derivar nuevas pruebas sólo para aquellos requerimientos **Derived::redefined( )** que no se satisfagan con las pruebas **Base::redefined( )**.

Para resumir, las pruebas **Base::redefined( )** se aplican a objetos de la clase **Derived**. Las entradas de prueba pueden ser adecuadas tanto para la clase base como para la derivada, pero los resultados esperados pueden diferir en la clase derivada.

19.4.5 Diseño de pruebas basadas en escenario

La prueba basada en escenario descubrirá errores que ocurren cuando cualquier actor interactúa con el software.

Las pruebas basadas en fallo pierden dos tipos principales de errores: 1) especificaciones incorrectas e 2) interacciones entre subsistemas. Cuando ocurren errores asociados con una especificación incorrecta, el producto no hace lo que el cliente quiere. Puede hacer lo correcto u omitir funcionalidad importante. Pero en cualquier circunstancia, la calidad (conformidad con los requerimientos) se resiente. Los errores asociados con la interacción de subsistemas ocurren cuando el comportamiento de un subsistema crea circunstancias (por ejemplo, eventos, flujo de datos) que hacen que otro subsistema falle.

La prueba basada en escenario se concentra en lo que hace el usuario, no en lo que hace el producto. Esto significa capturar las tareas (por medio de casos de uso) que el usuario tiene que realizar y luego aplicar éstas y sus variantes como pruebas.

Los escenarios descubren errores de interacción. Pero, para lograr esto, los casos de prueba deben ser más complejos y más realistas que las pruebas basadas en fallo. La prueba basada en escenario tiende a ejercitar múltiples subsistemas en una sola prueba (los usuarios no se limitan al uso de un subsistema a la vez).

Como ejemplo, tome en cuenta el diseño de pruebas basadas en escenario para un editor de texto al revisar los casos de uso que siguen:

Caso de uso: corrección del borrador final

Antecedentes: No es raro imprimir el borrador "final", leerlo y descubrir algunos errores desconcertantes que no fueron obvios en la imagen de la pantalla. Este caso de uso describe la secuencia de eventos que ocurren cuando esto sucede.

1. Imprimir todo el documento.
2. Moverse en el documento, cambiar ciertas páginas.
3. Conforme cada página cambia, imprimirla.
4. En ocasiones se imprime una serie de páginas.

"Si quiere y espera que un programa funcione, muy probablemente verá un programa en funcionamiento: no percibirá los errores."

Cem Kaner *et al.*

Este escenario describe dos cosas: una prueba y necesidades específicas del usuario. Las necesidades del usuario son obvias: 1) un método para imprimir páginas solas y 2) un método para imprimir un rango de páginas. Mientras avanzan las pruebas, hay necesidad de probar la edición después de imprimir (así como lo inverso). Por tanto, se trabaja para diseñar pruebas que descubrirán errores en la función de edición que fueron provocados por la función de impresión, es decir, errores que indicarán que las dos funciones de software no son adecuadamente independientes.

Caso de uso: imprimir una nueva copia

Antecedentes: Alguien pide al usuario una copia reciente del documento. Debe imprimirla.

1. Abrir el documento.
2. Imprimirlo.
3. Cerrar el documento.

De nuevo, el enfoque de las pruebas es relativamente obvio. Excepto que este documento no aparece de la nada. Se creó en una tarea anterior. ¿Dicha tarea afecta a la actual?

En muchos editores modernos, los documentos recuerdan cómo se imprimieron la última vez. Por defecto, imprimen de la misma forma la siguiente ocasión. Después del escenario **Corrección del borrador final**, seleccionar solamente "Imprimir" en el menú y dar clic en el botón Imprimir en el recuadro de diálogo hará que la última página corregida se imprima de nuevo. De manera que, de acuerdo con el editor, el escenario correcto debe verse del modo siguiente:

Aunque la prueba basada en escenario tiene méritos, obtendrá un mayor rendimiento en el tiempo invertido al revisar los casos de uso cuando estas pruebas se desarrollen como parte del modelo de análisis.

Caso de uso: imprimir una nueva copia

1. Abrir el documento.
2. Seleccionar "Imprimir" en el menú.
3. Comprobar si se imprime un rango de páginas; si es así, dar *clic* para imprimir todo el documento.
4. Dar *clic* en el botón Imprimir.
5. Cerrar el documento.

Pero este escenario indica una potencial especificación de error. El editor no hace lo que el usuario razonablemente espera que haga. Los clientes con frecuencia pasan por alto la comprobación anotada en el paso 3. Entonces quedarán desconcertados cuando vayan a la impresora y encuentren una página cuando querían 100. Los clientes desconcertados señalan errores de especificación.

Esta dependencia puede perderse cuando se diseñan pruebas, pero es probable que el problema salga a la luz durante las pruebas. Entonces tendría que lidiar con la probable respuesta: "¡así se supone que debe trabajar!"

19.4.6 Pruebas de las estructuras superficial y profunda

Cuando se habla de *estructura superficial* se hace referencia a la estructura observable externamente de un programa OO, es decir, la estructura que es inmediatamente obvia para un usuario final. En lugar de realizar funciones, a los usuarios de muchos sistemas OO se les pueden dar objetos para manipular en alguna forma. Pero, cualquiera que sea la interfaz, las pruebas se basan todavía en tareas de usuario. Capturar estas tareas involucra comprensión, observación y hablar con usuarios representativos (y tantos usuarios no representativos como valga la pena considerar).

Examinar la estructura superficial es análogo a la prueba de caja negra. La prueba de la estructura profunda es similar a la prueba de caja blanca.

Seguramente habrá alguna diferencia en los detalles. Por ejemplo, en un sistema convencional con una interfaz orientada a comandos, el usuario puede usar la lista de todos los comandos como una lista de comprobación de la prueba. Si no existieran escenarios de prueba para ejercitar un comando, la prueba probablemente pasaría por alto algunas tareas (o la interfaz tendría comandos inútiles). En una interfaz orientada a objetos, el examinador puede usar la lista de todos los objetos como una lista de comprobación de prueba.

Las mejores pruebas se derivan cuando el diseñador observa el sistema en una forma nueva o no convencional. Por ejemplo, si el sistema o producto tiene una interfaz basada en comando, se derivarán pruebas más profundas si el diseñador de casos de prueba pretende que las operaciones sean independientes de los objetos. Plantee preguntas como "¿el usuario querrá usar esta operación, que se aplica sólo al objeto **Scanner**, mientras trabaja con la impresora?". Cualquiera que sea el estilo de la interfaz, el diseño de casos de prueba que ejercitan la estructura superficial debe usar objetos y operaciones como pistas que conduzcan a tareas pasadas por alto.

Cita:

"No se avergüence por los errores y, por tanto, no los convierta en crímenes."

Confucio

Cuando se habla de *estructura profunda*, se hace referencia a los detalles técnicos internos de un programa OO, es decir, la estructura que se comprende al examinar el diseño y/o el código. La prueba de estructura profunda se diseña para ejercitar dependencias, comportamientos y mecanismos de comunicación que se establezcan como parte del modelo de diseño para el software OO.

Los modelos de requerimientos y diseño se usan como base para la prueba de la estructura profunda. Por ejemplo, el diagrama de colaboración UML o el modelo de despliegue muestran

colaboraciones entre objetos y subsistemas que pueden no ser visibles de manera externa. Entonces el diseño de caso de prueba pregunta: "¿Se capturó (como prueba) alguna tarea que ejercita la colaboración anotada en el diagrama de colaboración? Si no fue así, ¿por qué no se hizo?"

19.5 MÉTODOS DE PRUEBA APLICABLES EN EL NIVEL CLASE

El número de posibles permutas para la prueba aleatoria puede volverse muy grande. Para mejorar la eficiencia de la prueba, puede usarse una estrategia similar a la prueba de arreglo ortogonal.

La prueba "en lo pequeño" se enfoca en una sola clase y en los métodos que encapsula ésta. La prueba aleatoria y la partición son métodos que pueden usarse para ejercitar una clase durante la prueba OO.

19.5.1 Prueba aleatoria para clases OO

Para ofrecer breves ilustraciones de estos métodos, considere una aplicación bancaria en la que una clase **Account** (cuenta) tiene las siguientes operaciones: *open( )*, *setup( )*, *deposit( )*, *withdraw( )*, *balance( )*, *sumaries( )*, *creditLimit( )* y *close( )* (abrir, configurar, depósito, retiro, saldo, resumen, límite de crédito y cerrar) [Kir94]. Cada una de estas operaciones puede aplicarse a **Account**, pero ciertas restricciones (por ejemplo, la cuenta debe abrirse antes de que otras operaciones puedan aplicarse y debe cerrarse después de que todas las operaciones se completen) están implícitas por la naturaleza del problema. Incluso con estas restricciones, existen muchas permutas de las operaciones. La historia de vida de comportamiento mínima de una instancia de **Account** incluye las siguientes operaciones:

open • setup • deposit • withdraw • close

Esto representa la secuencia de prueba mínima para account. Sin embargo, dentro de esta secuencia puede ocurrir una amplia variedad de otros comportamientos:

open • setup • deposit • [deposit | withdraw | balance | summarize | creditLimit]^{n} • withdraw • close

Varias secuencias diferentes de operaciones pueden generarse al azar. Por ejemplo:

Caso de prueba r_1: **open • setup • deposit • deposit • balance • summarize • withdraw • close**

Caso de prueba r_2: **open • setup • deposit • withdraw • deposit • balance • creditLimit • withdraw • close**

Éstas y otras pruebas de orden aleatorio se realizan para ejercitar diferentes historias de vida de las instancias de clase.

CASASEGURA

Prueba de clase

La escena: Cubículo de Shakira.

Participantes: Jamie y Shakira, miembros del equipo de ingeniería de software CasaSegura, que trabajan en el diseño de casos de prueba para la función seguridad.

La conversación:

Shakira: Desarrollé algunas pruebas para la clase **Detector** [figura 10.4]; tú sabes, la que permite el acceso a todos los objetos **Sensor** para la función seguridad. ¿Estás familiarizado con ella?

Jamie (ríe): Seguro, es la que te permite agregar el sensor "angustia de perrito".

Shakira: La única. De cualquier forma, tiene una interfaz con cuatro operaciones: *read( )*, *enable( )*, *disable( )* y *test( )*. Antes de poder leer un sensor, debe habilitársele. Una vez habilitado, puede leerse y probarse. Puede deshabilitarse en cualquier momento, excepto si se procesa una condición de alarma. Así que definí una secuencia de prueba simple que ejercitará su historia de vida de comportamiento. [Muestra a Jamie la siguiente secuencia].

#1: enable•test•read•disable

Jamie: Eso funcionará, ¡pero tienes que hacer más pruebas que eso!

Shakira: Ya sé, ya sé, aquí hay otras secuencias que encontré. [Muestra a Jamie las siguientes secuencias].

#2: enable•test*[read]^n•test•disable

#3: [read]^n

#4: enable*disable•[test | read]

Jamie: Déjame ver si entiendo la intención de éstos. El número 1 pasa a través de una historia de vida normal, una especie de uso convencional. El número 2 repite la operación leer n veces, y ése es un escenario probable. El número 3 intenta leer el sensor antes de que esté habilitado... eso produciría un mensaje de error de algún tipo, ¿cierto? El número 4 habilita y deshabilita el sensor y luego intenta leerlo. ¿No es lo mismo que la prueba 2?

Shakira: En realidad, no. En el número 4, el sensor se habilitó. Lo que realmente prueba el número 4 es si la operación deshabilitar funciona como debe. Un *read( )* o *test( )* después de *disable( )* generaría el mensaje de error. Si no lo hace, entonces hay un error en la operación deshabilitar.

Jamie: Bien. Sólo recuerda que las cuatro pruebas tienen que aplicarse para cada tipo de sensor, pues todas las operaciones pueden tener diferencias sutiles dependiendo del tipo de sensor.

Shakira: No hay que preocuparse. Ése es el plan.

19.5.2 Prueba de partición en el nivel de clase

? ¿Qué opciones de prueba están disponibles en el nivel de clase?

La *prueba de partición* reduce el número de casos de prueba requeridos para ejercitar la clase, en una forma muy similar a la partición de equivalencia (capítulo 18) para el software tradicional. Las entradas y salidas se categorizan y los casos de prueba se diseñan para ejercitar cada categoría. ¿Pero cómo se derivan las categorías de partición?

La *partición con base en estado* categoriza las operaciones de clase a partir de su capacidad para cambiar el estado de la clase. Considere de nuevo la clase **Account**, las operaciones de estado incluyen *deposit( )* y *withdraw( )*, mientras que las operaciones de no estado incluyen *balance( )*, *sumaries( )* y *creditLimit( )*. Las pruebas se diseñan para que ejerciten por separado las operaciones que cambian el estado y aquellas que no lo cambian. En consecuencia,

Caso de prueba p_1: **open • setup • deposit • deposit • withdraw • withdraw • close**
Caso de prueba p_2: **open • setup • deposit • summarize • creditLimit • withdraw • close**

El caso de prueba p_1 cambia el estado, mientras que el p_2 ejercita las operaciones que no cambian el estado (distintas a las que están en la secuencia de prueba mínima).

La *partición con base en atributo* categoriza las operaciones de clase con base en los atributos que usan. Para la clase **Account**, los atributos **balance** y **creditLimit** pueden usarse para definir particiones. Las operaciones se dividen en tres particiones: 1) operaciones que usan **creditLimit**, 2) operaciones que modifican **creditLimit** y 3) operaciones que no usan ni modifican **creditLimit**. Entonces se diseñan secuencias de prueba para cada partición.

La *partición basada en categoría* jerarquiza las operaciones de clase con base en la función genérica que cada una realiza. Por ejemplo, las operaciones en la clase **Account** pueden categorizarse en operaciones de inicialización (*open*, *setup*), de cálculo (*deposit, withdraw*), consultas (*balance, summarize, creditLimit*) y de terminación (*close*).

19.6 Diseño de casos de prueba interclase

El diseño de casos de prueba se vuelve más complicado conforme comienza la integración del sistema orientado a objetos. En esta etapa debe comenzar la prueba de las colaboraciones entre clases. Para ilustrar "la generación de casos de prueba interclase" [Kir94], se expande el ejemplo bancario presentado en la sección 19.5 a fin de incluir las clases y colaboraciones anotadas en la figura 19.2. La dirección de las flechas en la figura indica la dirección de los mensajes y las etiquetas indican las operaciones que se involucran como consecuencia de las colaboraciones que implican los mensajes.

Cita:

"La frontera que define el ámbito de las pruebas de unidad y de integración es diferente para el desarrollo orientado a objetos. Las pruebas pueden diseñarse y ejercitarse en muchos puntos en el proceso. Por tanto, "diseñe un poco, codifique un poco" se convierte en "diseñe un poco, codifique un poco, pruebe un poco".

Robert Binder

Al igual que la prueba de clases individuales, la de colaboración de clase puede lograrse aplicando métodos aleatorios y de partición, así como pruebas basadas en escenario y pruebas de comportamiento.

19.6.1 Prueba de clase múltiple

Kirani y Tsai [Kir94] sugieren la siguiente secuencia de pasos para generar casos de prueba aleatorios de clase múltiple:

1. Para cada clase cliente, use la lista de operaciones de clase a fin de generar una serie de secuencias de prueba aleatorias. Las operaciones enviarán mensajes a otras clases servidor.
2. Para cada mensaje generado, determine la clase colaborador y la correspondiente operación en el objeto servidor.
3. Para cada operación en el objeto servidor (invocado por los mensajes enviados desde el objeto cliente), determine los mensajes que transmite.
4. Para cada uno de los mensajes, determine el siguiente nivel de operaciones que se invocan e incorpore esto en la secuencia de prueba.

Para ilustrar [Kir94], considere una secuencia de operaciones para la clase **Bank** en relación con una clase **ATM** (figura 19.2):

verifyAcct • verifyPIN • [[verifyPolicy • withdrawReq] | depositReq | acctInfoREQ]^n

Un caso de prueba aleatorio para la clase **Bank** puede ser

Caso de prueba r_3 = **verifyAcct • verifyPIN • depositReq**

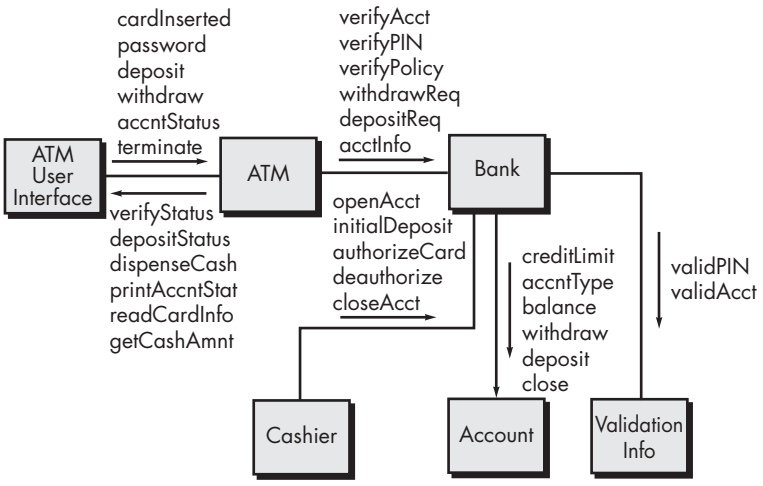
Para considerar los colaboradores involucrados en esta prueba, se consideran los mensajes asociados con cada una de las operaciones anotadas en el caso de prueba r_3. **Bank** debe colaborar con **ValidationInfo** para ejecutar *verifyAcct( )* y *verifyPIN( )*. **Bank** debe colaborar con **Account** para ejecutar *depositReq( )*. Por tanto, un nuevo caso de prueba que ejercita estas colaboraciones es

Caso de prueba r_4 = **verifyAcct [Bank:validAcctValidationInfo] • verifyPIN [Bank: validPinValidationInfo] • depositReq [Bank: depositaccount]**

FIGURA 19.2

Diagrama de colaboración de clases para aplicación bancaria

Fuente: Adaptado de [Kir94].



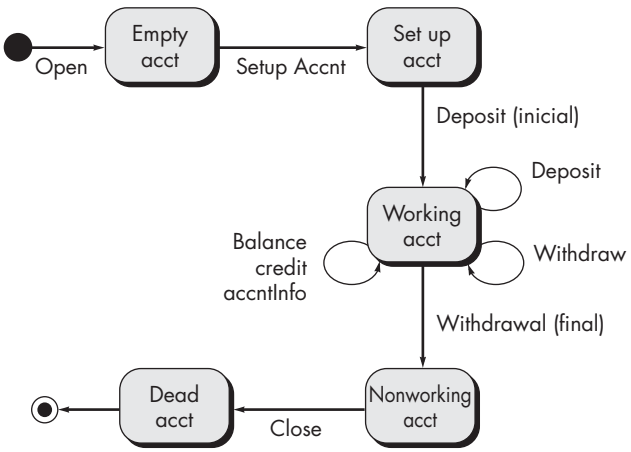


FIGURA 19.3
Diagrama de estado para la clase Account
Fuente: Adaptado de [Kir94].

El enfoque de prueba de partición de clase múltiple es similar al que se usó para la prueba de partición de clases individuales. Una sola clase se divide, como se estudió en la sección 19.5.2. Sin embargo, la secuencia de prueba se expande para incluir aquellas operaciones que se invocan mediante mensajes a clases que colaboran. Un enfoque alternativo divide las pruebas con base en las interfaces en una clase particular. En la figura 19.2, la clase **Bank** recibe mensajes de las clases **ATM** y **Cashier**. Por tanto, los métodos dentro de **Bank** pueden probarse al dividirlos en los que sirven a **ATM** y los que sirven a **Cashier**. La partición con base en estado (sección 19.5.2) puede usarse para refinar aún más las particiones.

19.6.2 Pruebas derivadas a partir de modelos de comportamiento

El uso del diagrama de estado como modelo que representa el comportamiento dinámico de una clase se analiza en el capítulo 7. El diagrama de estado para una clase puede usarse a fin de ayudar a derivar una secuencia de pruebas que ejercitarán el comportamiento dinámico de la clase (y de aquellas clases que colaboran con ella). La figura 19.3 [Kir94] ilustra un diagrama de estado para la clase **Account** estudiada anteriormente. En la figura, las transiciones iniciales se mueven a través de los estados *empty acct* (cuenta vacía) y *setup acct* (configuración de cuenta). La mayoría de los comportamientos para instancias de la clase ocurren mientras está en el estado *working acct* (cuenta operativa). Un retiro final y un cierre de cuenta harán que la clase **Account** realice transiciones hacia los estados *nonworking acct* (cuenta no operativa) y *dead acct* (cuenta muerta), respectivamente.

Las pruebas que se van a diseñar deben lograr cobertura de todos los estados, es decir, las secuencias de operación deben hacer que la clase **Account** realice transiciones a través de todos los estados permisibles:

Caso de prueba s_1: **open • setupAccnt • deposit (initial) • withdraw (final) • close**

Debe observarse que esta secuencia es idéntica a la de prueba mínima que se estudió en la sección 19.5.2. Al agregar secuencias de prueba adicionales a la secuencia mínima,

Caso de prueba s_2: **open • setupAccnt • deposit(initial) • deposit • balance • credit • withdraw (final) • close**
Caso de prueba s_3: **open • setupAccnt • deposit(initial) • deposit • withdraw • accntInfo • withdraw (final) • close**

Es necesario derivar todavía más casos de prueba para garantizar que todos los comportamientos para la clase se ejercitaron adecuadamente. En situaciones en las que el compor-

tamiento de clase da como resultado una colaboración con una o más clases, se usan diagramas de estado múltiple para rastrear el flujo de comportamiento del sistema.

El modelo de estado puede recorrerse en una forma "ancho primero" [McG94]. En este contexto, ancho primero implica que un caso de prueba ejercita una sola transición y que, cuando se prueba una nueva transición, sólo se usan transiciones previamente probadas.

Considere un objeto **CreditCard** que es parte del sistema bancario. El estado inicial de **CreditCard** es *indefinido* (es decir, no se proporcionó número de tarjeta de crédito). Hasta leer la tarjeta de crédito durante una venta, el objeto toma un estado *definido*, es decir, se definen los atributos **card number** y **expiration date**, junto con identificadores específicos del banco. La tarjeta de crédito se somete cuando se envía para autorización y se aprueba cuando se recibe la autorización. La transición de **CreditCard** de un estado a otro puede probarse al derivar casos de prueba que hacen que ocurra la transición. Un enfoque de ancho primero aplicado a este tipo de prueba no ejercitaría *submitted* antes de ejercitar *undefined* y *defined*. Si lo hiciera, usaría transiciones que no se probaron anteriormente y, por tanto, violaría el criterio de ancho primero.

19.7 Resumen

El objetivo global de las pruebas orientadas a objetos (encontrar el número máximo de errores con una cantidad mínima de esfuerzo) es idéntico al de la prueba de software convencional. Pero la estrategia y las tácticas de la prueba OO difieren significativamente. La visión de las pruebas se ensancha para incluir la revisión de los modelos de requerimientos y de diseño. Además, el foco de la prueba se mueve alejándose del componente procedimental (el módulo) y acercándose hacia la clase.

Puesto que los modelos de requerimientos y diseño OO y el código fuente resultante están semánticamente acoplados, la prueba (en la forma de revisiones técnicas) comienza durante la actividad de modelado. Por esta razón, la revisión de los modelos CRC, objeto-relación y objeto-comportamiento puede verse como pruebas de primera etapa.

Una vez disponible el código, la prueba de unidad se aplica para cada clase. El diseño de pruebas para una clase usa varios métodos: prueba basada en fallo, prueba aleatoria y prueba de partición. Cada uno de éstos ejercita las operaciones encapsuladas por la clase. Las secuencias de prueba se diseñan para garantizar que se ejercitan las operaciones relevantes. El estado de la clase, representado por los valores de sus atributos, se examina para determinar si existen errores.

La prueba de integración puede lograrse usando una estrategia basada en hebra o en uso. La prueba basada en hebra integra el conjunto de clases que colaboran para responder a una entrada o evento. La prueba basada en uso construye el sistema en capas, comenzando con aquellas clases que no utilizan clases servidor. La integración de métodos de diseño de caso de prueba también puede usar pruebas aleatorias y de partición. Además, la prueba basada en escenario y las pruebas derivadas de los modelos de comportamiento pueden usarse para probar una clase y a sus colaboradores. Una secuencia de prueba rastrea el flujo de operaciones a través de las colaboraciones de clase.

La prueba de validación del sistema OO está orientada a caja negra y puede lograrse al aplicar los mismos métodos de caja negra estudiados para el software convencional. Sin embargo, la prueba basada en escenario domina la validación de los sistemas OO, lo que hace al caso de uso un impulsor primario para la prueba de validación.

Problemas y puntos por evaluar

19.1. Con sus palabras, describa por qué la clase es la unidad razonable más pequeña para probar dentro de un sistema OO.

19.2. ¿Por qué es necesario volver a probar las subclases que se instancian a partir de una clase existente si ésta ya se probó ampliamente? ¿Puede usarse el diseño de casos de prueba para la clase existente?

19.3. ¿Por qué la "prueba" debe comenzar con el análisis y el diseño orientado a objetos?

19.4. Derive un conjunto de tarjetas índice CRC para *CasaSegura* y realice los pasos anotados en la sección 19.2.2 para determinar si existen inconsistencias.

19.5. ¿Cuál es la diferencia entre las estrategias basadas en hebra y basadas en uso para la prueba de integración? ¿Cómo encaja la prueba de grupo?

19.6. Aplique pruebas aleatorias y de partición a tres clases definidas en el diseño del sistema *CasaSegura*. Produzca casos de prueba que indiquen las secuencias de operación que se invocarán.

19.7. Aplique prueba de clase múltiple y pruebas derivadas del modelo de comportamiento al diseño de *CasaSegura*.

19.8. Derive cuatro pruebas adicionales usando prueba aleatoria y métodos de partición, así como prueba de clase múltiple y pruebas derivadas del modelo de comportamiento, para la aplicación bancaria que se presentó en las secciones 19.5 y 19.6.

Lecturas adicionales y fuentes de información

Muchos de los libros acerca de pruebas mencionados en las secciones *Lecturas adicionales y fuentes de información* de los capítulos 17 y 18 estudian en cierta medida las pruebas de los sistemas OO. Schach (*Object-Oriented and Classical Software Engineering*, McGraw-Hill, 6a. ed., 2004) considera la prueba OO dentro del contexto de una práctica de ingeniería de software más amplia. Sykes y McGregor (*Practical Guide to Testing Object-Oriented Software,* Addison-Wesley, 2001), Bashir y Goel (*Testing Object-Oriented Software,* Springer 2000), Binder (*Testing Object-Oriented Systems,* Addison-Wesley, 1999) y Kung *et al.* (*Testing Object-Oriented Software,* Wiley-IEEE Computer Society Press, 1998) tratan la prueba de OO con significativo detalle.

En internet está disponible gran variedad de fuentes de información acerca de métodos de prueba orientados a objeto. En el sitio del libro **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm** puede encontrarse una lista actualizada de referencias en la World Wide Web que son relevantes para las técnicas de prueba.

CAPÍTULO

20

PRUEBA DE APLICACIONES WEB

CONCEPTOS CLAVE

dimensiones de calidad 454
estrategia. 455
planificación 456
prueba de base de datos . . . 458
prueba de carga 472
prueba de configuración. . . . 469
prueba de contenido 457
prueba de esfuerzo 473
prueba de interfaz. 460
prueba de navegación 467
prueba de rendimiento. 471
prueba de seguridad 470
prueba de usabilidad 463
prueba en el nivel de componente 466
pruebas de compatibilidad . . 465

Existe una urgencia que siempre impregna un proyecto web. Los participantes (intranquilos por la competencia de otras *webapps*, presionados por las demandas del cliente y preocupados porque perderán la ventana de mercado) fuerzan para poner la *webapp* en línea. Como consecuencia, en ocasiones desechan por completo las actividades técnicas que frecuentemente ocurren tarde en el proceso, como las pruebas de la aplicación web. Esto puede ser un error catastrófico. Para evitarlo, los miembros del equipo deben asegurarse de que cada producto resultante muestre alta calidad. Wallace *et al.* [Wal03] observan esto cuando afirman:

> Las pruebas no deben esperar hasta que el proyecto finalice. Comience a probar antes de escribir una línea de código. Pruebe constante y efectivamente, y desarrollará un sitio web mucho más duradero.

Los modelos de requerimientos y de diseño no pueden probarse en el sentido clásico: por ello, el equipo debe realizar revisiones técnicas (capítulo 15) y pruebas ejecutables. La intención es descubrir y corregir errores antes de que la *webapp* esté disponible para sus usuarios finales.

20.1 CONCEPTOS DE PRUEBAS PARA APLICACIONES WEB

Probar es el proceso de ejecución del software con la intención de encontrar (y a final de cuentas corregir) errores. Esta filosofía fundamental, que se expuso por primera vez en el capítulo 17, no cambia para las *webapps*. De hecho, puesto que los sistemas y las aplicaciones basadas en web residen en una red e interactúan con muchos sistemas operativos, navegado-

UNA MIRADA RÁPIDA

¿Qué es? La prueba de una *webapp* es una colección de actividades relacionadas con una sola meta: descubrir errores en el contenido, función, utilidad, navegabilidad, rendimiento, capacidad y seguridad de esa aplicación. Para lograr esto, se aplica una estrategia de prueba que abarca tanto revisiones como pruebas ejecutables.

¿Quién lo hace? En las pruebas de una *webapp* participan ingenieros en web y otros participantes en el proyecto (gestores, clientes y usuarios).

¿Por qué es importante? Si los usuarios finales encuentran errores que derrumben su fe en la *webapp*, irán a algún otro lado en busca del contenido y de la función que necesitan, y la aplicación fracasará. Por esta razón, debe trabajarse para eliminar tantos errores como sea posible antes de poner en línea la *webapp*.

¿Cuáles son los pasos? El proceso de prueba de una *webapp* comienza enfocándose en los aspectos visibles para el usuario de la aplicación y avanza hacia pruebas que ejercitan la tecnología y la infraestructura. Se realizan siete pasos durante la prueba: prueba de contenido, prueba de interfaz, prueba de navegación, prueba de componente, prueba de configuración, prueba de rendimiento y prueba de seguridad.

¿Cuál es el producto final? En algunas ocasiones, se produce un plan de prueba para la *webapp*. En todo caso, se desarrolla una suite de casos de prueba para cada paso de prueba y se mantiene un archivo de los resultados de la prueba para un uso futuro.

¿Cómo me aseguro de que lo hice bien? Aunque nunca se puede estar seguro de que se realizaron todas las pruebas que se necesitan, es posible tener la certeza de que se descubrieron errores (y se corrigieron). Además, si se estableció un plan de prueba, puede realizarse la comprobación para garantizar que todas las pruebas planeadas se llevaron a cabo.

res (residentes en varios dispositivos), plataformas de hardware, protocolos de comunicaciones y aplicaciones "de cuarto trasero" diferentes, la búsqueda de errores representa un reto significativo.

Para entender los objetivos de las pruebas dentro de un contexto de ingeniería web, debe considerar las muchas dimensiones de calidad de la *webapp*.[1] En el contexto de esta discusión, se consideran las dimensiones de calidad que son particularmente relevantes en cualquier análisis de las pruebas de la *webapp*. También se considera la naturaleza de los errores que se encuentran como consecuencia de las pruebas y la estrategia de prueba que se aplica para descubrir dichos errores.

20.1.1 Dimensiones de calidad

La calidad se incorpora en una aplicación web como consecuencia de un buen diseño. Se evalúa aplicando una serie de revisiones técnicas que valoran varios elementos del modelo de diseño y un proceso de prueba que se estudia a lo largo de este capítulo. Tanto las revisiones como las pruebas examinan una o más de las siguientes dimensiones de calidad [Mil00a]:

? ¿Cómo se valora la calidad dentro del contexto de una *webapp* y su entorno?

- El *contenido* se evalúa tanto en el nivel sintáctico como en el semántico. En el primero, se valora vocabulario, puntuación y gramática para documentos basados en texto. En el segundo, se valora la corrección (de la información presentada), la consistencia (a través de todo el objeto de contenido y de los objetos relacionados) y la falta de ambigüedad.
- La *función* se prueba para descubrir errores que indican falta de conformidad con los requerimientos del cliente. Cada función de la *webapp* se valora en su corrección, inestabilidad y conformidad general con estándares de implantación adecuados (por ejemplo, estándares de lenguaje Java o AJAX).
- La *estructura* se valora para garantizar que entrega adecuadamente el contenido y la función de la aplicación, que es extensible y que puede soportarse conforme se agregue nuevo contenido o funcionalidad.
- La *usabilidad* se prueba para asegurar que la interfaz soporta a cada categoría de usuario y que puede aprender y aplicar toda la sintaxis y semántica de navegación requerida.
- La *navegabilidad* se prueba para asegurar que toda la sintaxis y la semántica de navegación se ejecutan para descubrir cualquier error de navegación (por ejemplo, vínculos muertos, inadecuados y erróneos).
- El *rendimiento* se prueba bajo condiciones operativas, configuraciones y cargas diferentes a fin de asegurar que el sistema responde a la interacción con el usuario y que maneja la carga extrema sin degradación operativa inaceptable.
- La *compatibilidad* se prueba al ejecutar la *webapp* en varias configuraciones anfitrión, tanto en el cliente como en el servidor. La intención es encontrar errores que sean específicos de una configuración anfitrión única.
- La *interoperabilidad* se prueba para garantizar que la *webapp* tiene interfaz adecuada con otras aplicaciones y/o bases de datos.
- La *seguridad* se prueba al valorar las vulnerabilidades potenciales e intenta explotar cada una. Cualquier intento de penetración exitoso se estima como un fallo de seguridad.

Cita:

"La innovación es una negociación agridulce para los examinadores de software. Justo cuando parece que se sabe cómo probar una tecnología particular, aparece una nueva [*webapp*] y cualquier cosa puede ocurrir."

James Bach

La estrategia y las tácticas para probar las *webapps* se desarrollaron a fin de ejercitar cada una de estas dimensiones de calidad y se estudian más adelante, en este capítulo.

---

1 Las dimensiones genéricas de la calidad del software, igualmente válidas para las *webapps*, se estudiaron en el capítulo 14.

20.1.2 Errores dentro de un entorno de *webapp*

Los errores que se encuentran como consecuencia de una prueba exitosa de una *webapp* tienen algunas características únicas [Ngu00]:

¿Qué hace que los errores encontrados durante la ejecución de una *webapp* sean un poco diferentes a los que se encuentran para el software convencional?

1. Puesto que muchos tipos de pruebas de *webapps* descubren problemas que se evidencian primero en el lado del cliente (es decir, mediante una interfaz implantada en un navegador específico o en un dispositivo de comunicación personal), con frecuencia se ve un síntoma del error, no el error en sí.
2. Puesto que una *webapp* se implanta en algunas configuraciones distintas y dentro de diferentes entornos, puede ser difícil o imposible reproducir un error afuera del entorno en el que originalmente se encontró.
3. Aunque algunos errores son resultado de diseño incorrecto o codificación HTML (u otro lenguaje de programación) impropia, muchos errores pueden rastrearse en la configuración de la *webapp*.
4. Dado que las *webapps* residen dentro de una arquitectura cliente-servidor, los errores pueden ser difíciles de rastrear a través de tres capas arquitectónicas: el cliente, el servidor o la red en sí.
5. Algunos errores se deben al *entorno operativo estático* (es decir, a la configuración específica donde se realiza la prueba), mientras que otros son atribuibles al entorno operativo dinámico (es decir, a la carga de recurso instantánea o a errores relacionados con el tiempo).

Estos cinco atributos de error sugieren que el entorno juega un importante papel en el diagnóstico de todos los errores descubiertos durante la prueba de *webapps*. En algunas situaciones (por ejemplo, la prueba de contenido), el sitio del error es obvio, pero en muchos otros tipos de prueba de *webapps* (por ejemplo, prueba de navegación, prueba de rendimiento, prueba de seguridad), la causa subyacente del error puede ser considerablemente más difícil de determinar.

20.1.3 Estrategia de las pruebas

La estrategia para probar *webapps* adopta los principios básicos de todas las pruebas de software (capítulo 17) y aplica una estrategia y las tácticas que se recomendaron para los sistemas orientados a objetos (capítulo 19). Los siguientes pasos resumen el enfoque:

PUNTO CLAVE

La estrategia global para probar *webapp* puede resumirse en los 10 pasos que se anotan aquí.

1. El modelo de contenido para la *webapp* a se revisa a fin de descubrir errores.
2. El modelo de interfaz se examina para garantizar que todos los casos de uso pueden alojarse.
3. El modelo de diseño para la *webapp* se revisa para descubrir errores de navegación.
4. La interfaz de usuario se prueba para descubrir errores en la mecánica de presentación y/o navegación.
5. Los componentes funcionales se someten a prueba de unidad.
6. Se prueba la navegación a lo largo de toda la arquitectura.
7. La *webapp* se implanta en varias configuraciones de entorno diferentes y se prueba para asegurar la compatibilidad con cada configuración.
8. Las pruebas de seguridad se realizan con la intención de explotar las vulnerabilidades en la *webapp* o dentro de su entorno.

WebRef

En **www.stickyminds.com/testing.asp** se encuentran excelentes artículos acerca de pruebas de *webapps*

9. Se realizan pruebas de rendimiento.
10. La *webapp* se prueba con una población controlada y monitoreada de usuarios finales; los resultados de su interacción con el sistema se evalúan para detectar errores de contenido y de navegación, preocupaciones de usabilidad y compatibilidad, y seguridad, confiabilidad y rendimiento de la *webapp*.

Puesto que muchas *webapps* evolucionan continuamente, el proceso de prueba es una actividad siempre en marcha que realiza el personal de apoyo web, quien usa pruebas de regresión derivadas de las pruebas desarrolladas cuando comenzó la ingeniería de las *webapps*.

20.1.4 Planificación de pruebas

El uso de la palabra *planificación* (en cualquier contexto) es un anatema para algunos desarrolladores web que no planifican; sólo arrancan, con la esperanza de que surja una *webapp* asesina. Un enfoque más disciplinado reconoce que la planificación establece un mapa de ruta para todo el trabajo que va después. Vale la pena el esfuerzo. En su libro acerca de las pruebas de *webapps*, Splaine y Jaskiel [Spl01] afirman:

> Excepto por el más simple de los sitios web, rápidamente resulta claro que es necesaria alguna especie de planificación de pruebas. Con demasiada frecuencia, el número inicial de errores encontrados a partir de una prueba *ad hoc* es suficientemente grande como para que no todos se corrijan la primera vez que se detectan. Esto impone una carga adicional sobre el personal que prueba sitios y *webapps*. No sólo deben idear nuevas pruebas imaginativas, sino que también deben recordar cómo se ejecutaron las pruebas anteriores con la finalidad de volver a probar de manera confiable el sitio/*webapp*, y garantizar que se removieron los errores conocidos y que no se introdujeron algunos nuevos.

Las preguntas que deben plantearse son: ¿cómo se "idean nuevas pruebas imaginativas" y sobre qué deben enfocarse dichas pruebas? Las respuestas a estas preguntas se integran en un plan de prueba que identifica: 1) el conjunto de tareas[2] que se van a aplicar cuando comiencen las pruebas, 2) los productos de trabajo que se van a producir conforme se ejecuta cada tarea de prueba y 3) la forma en la que se evalúan, registran y reutilizan los resultados de la prueba cuando se realizan pruebas de regresión. En algunos casos, el plan de prueba se integra con el plan del proyecto. En otros, es un documento separado.

PUNTO CLAVE

El plan de prueba identifica el conjunto de tareas de pruebas, los productos de trabajo que se van a desarrollar y la forma en la que deben evaluarse, registrarse y reutilizarse los resultados.

20.2 Un panorama del proceso de prueba

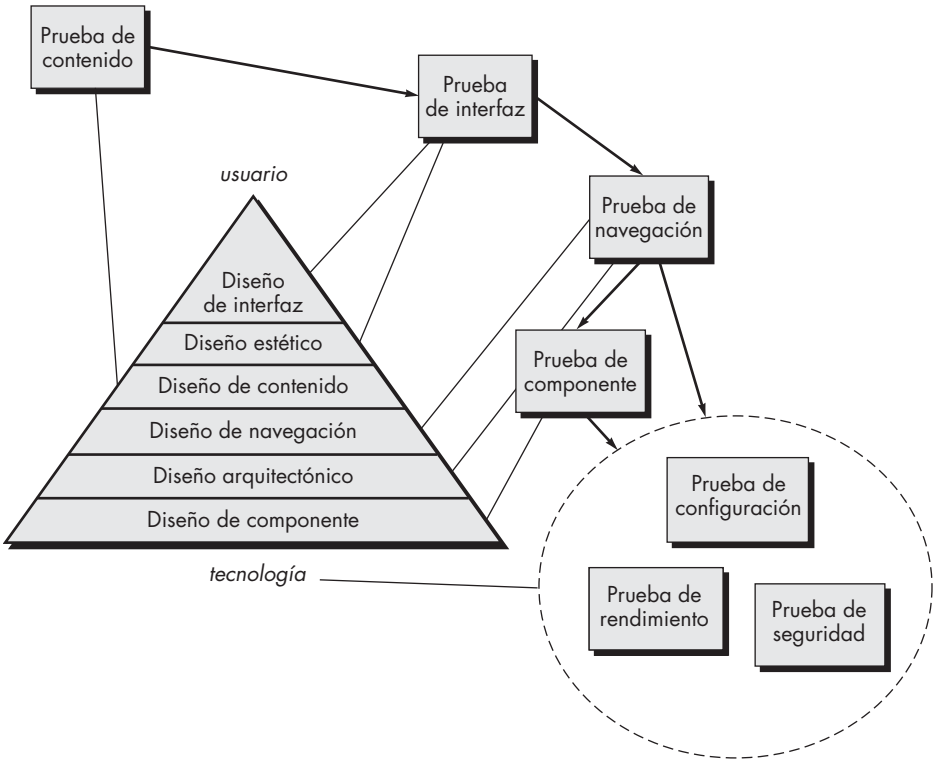
El proceso de prueba de *webapps* comienza con pruebas que ejercitan la funcionalidad del contenido y la interfaz que son inmediatamente visibles para el usuario final. Conforme avanza la prueba, se ejercitan aspectos de la arquitectura del diseño y de la navegación. Finalmente, la atención se centra en las pruebas que examinan las capacidades tecnológicas que no siempre son aparentes para los usuarios finales: los temas de infraestructura e instalación/ implantación de la *webapp*.

La figura 20.1 yuxtapone el proceso de prueba de la *webapp* con la pirámide de diseño para este tipo de aplicaciones (capítulo 13). Observe que, conforme el flujo de la prueba avanza de izquierda a derecha y de arriba abajo, los elementos visibles para el usuario del diseño de la *webapp* (elementos superiores de la pirámide) se prueban primero, seguidos por los elementos de diseño de infraestructura.

---

2 Los conjuntos de tareas se estudian en el capítulo 2. También se usa un término relacionado, *flujo de trabajo*, para describir una serie de tareas requeridas para lograr una actividad de ingeniería del software.

FIGURA 20.1
El proceso de prueba



20.3 PRUEBA DE CONTENIDO

Los errores en el contenido de la *webapp* pueden ser tan triviales como errores tipográficos menores o tan significativos como información incorrecta, organización inadecuada o violación de leyes de la propiedad intelectual. La *prueba de contenido* intenta descubrir éstos y muchos otros problemas antes de que el usuario los encuentre.

CONSEJO

Aunque las revisiones técnicas no son parte de las pruebas, debe realizarse la revisión del contenido para garantizar que éste tiene calidad.

La prueba de contenido combina tanto revisiones como generación de casos de prueba ejecutables. Las revisiones se aplican para descubrir errores semánticos en el contenido (que se estudia en la sección 20.3.1). Las pruebas ejecutables se usan para descubrir errores de contenido que puedan rastrearse a fin de derivar dinámicamente contenido que se impulse por los datos adquiridos de una o más bases de datos.

20.3.1 Objetivos de la prueba de contenido

PUNTO CLAVE

Los objetivos de la prueba de contenido son: 1) descubrir errores sintácticos en el contenido, 2) descubrir errores semánticos y 3) encontrar errores estructurales.

La prueba de contenido tiene tres objetivos importantes: 1) descubrir errores sintácticos (por ejemplo, errores tipográficos o gramaticales) en documentos de texto, representaciones gráficas y otros medios; 2) descubrir errores semánticos (es decir, errores en la precisión o completitud de la información) en cualquier objeto de contenido que se presente conforme ocurre la navegación y 3) encontrar errores en la organización o estructura del contenido que se presenta al usuario final.

Para lograr el primer objetivo, pueden usarse correctores automáticos de vocabulario y gramática. Sin embargo, muchos errores sintácticos evaden la detección de tales herramientas y los debe descubrir un revisor humano (examinador). De hecho, un sitio web grande debe considerar los servicios de un editor profesional para descubrir errores tipográficos, gazapos gramaticales, errores en la consistencia del contenido, errores en las representaciones gráficas y en referencias cruzadas.

La prueba semántica se enfoca en la información presentada dentro de cada objeto de contenido. El revisor (examinador) debe responder las siguientes preguntas:

¿Qué preguntas deben plantearse y responderse para descubrir errores semánticos en el contenido?

- ¿La información realmente es precisa?
- ¿La información es concisa y puntual?
- ¿La plantilla del objeto de contenido es fácil de comprender para el usuario?
- ¿La información incrustada dentro de un objeto de contenido puede encontrarse con facilidad?
- ¿Se proporcionaron referencias adecuadas para toda la información derivada de otras fuentes?
- ¿La información presentada es consistente internamente y con la información presentada en otros objetos de contenido?
- ¿El contenido es ofensivo, confuso o abre la puerta a demandas?
- ¿El contenido infringe derechos de autor o nombres comerciales existentes?
- ¿El contenido incluye vínculos internos que complementan el contenido existente? ¿Los vínculos son correctos?
- ¿El estilo estético del contenido entra en conflicto con el estilo estético de la interfaz?

Cita:

"En general, las técnicas de prueba del software que se emplean en otras aplicaciones son las mismas que las usadas en aplicaciones basadas en web [...] La diferencia [...] es que las variables tecnológicas en el entorno web se multiplican."

Hung Nguyen

Obtener respuestas a cada una de estas preguntas para una gran *webapp* (que contiene cientos de objetos de contenido) puede ser una tarea atemorizante. Sin embargo, el fracaso para descubrir los errores semánticos sacudirá la fe del usuario en la *webapp* y puede conducir al fracaso de la aplicación basada en web.

Los objetos de contenido existen dentro de una arquitectura que tiene un estilo específico (capítulo 13). Durante la prueba de contenido, la estructura y organización de la arquitectura de contenido se prueba para garantizar que el contenido requerido se presente al usuario final en el orden y relaciones adecuados. Por ejemplo, la *webapp* **CasaSeguraAsegurada.com** presenta información variada acerca de los sensores que se utilizan como parte de los productos de seguridad y vigilancia. Los objetos de contenido proporcionan información descriptiva, especificaciones técnicas, una representación fotográfica e información relacionada. Las pruebas de la arquitectura de contenido de **CasaSeguraAsegurada.com** luchan por descubrir errores en la presentación de esta información (por ejemplo, una descripción del sensor X se presenta con una fotografía del sensor Y).

20.3.2 Prueba de base de datos

Las *webapps* modernas hacen mucho más que presentar objetos de contenido estáticos. En muchos dominios de aplicación, la *webapp* tiene interfaz con sofisticados sistemas de gestión de base de datos y construyen objetos de contenido dinámico que se crean en tiempo real, usando los datos adquiridos desde una base de datos.

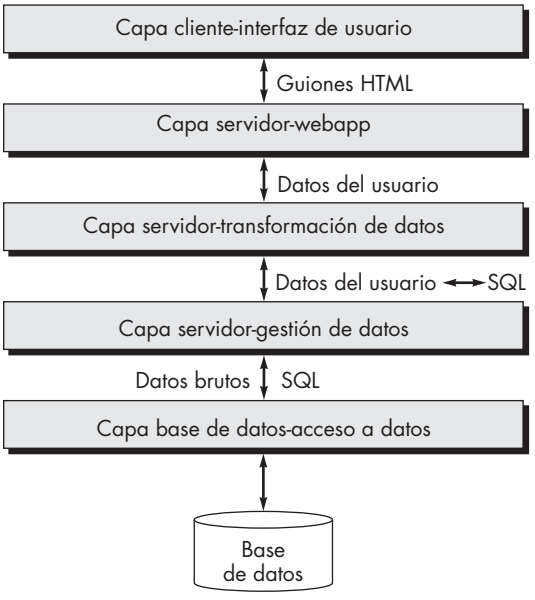
Por ejemplo, una *webapp* de servicios financieros puede producir información compleja basada en texto, tablas tabulares y gráficas acerca de un fondo específico (por ejemplo, una acción o fondo mutualista). El objeto de contenido compuesto que presenta esta información se crea de manera dinámica después de que el usuario hace una solicitud de información acerca de un fondo específico. Para lograrlo, se requieren los siguientes pasos: 1) consulta a una gran base de datos de fondos, 2) extracción de datos relevantes de la base de datos, 3) organización de los datos extraídos como un objeto de contenido y 4) transmisión de este objeto de contenido (que representa información personalizada que requiere un usuario final) al entorno del cliente para su despliegue. Los errores pueden ocurrir, y ocurren, como consecuencia de cada uno de estos pasos. El objeto de la prueba de la base de datos es descubrir dichos errores, pero esta prueba es complicada por varios factores:

¿Qué cosas complican la prueba de base de datos para *webapps*?

1. *El lado cliente original solicita información que rara vez se presenta en la forma [por ejemplo, lenguaje de consulta estructurado (SQL)] en la que puede ingresarse a un sistema de gestión de base de datos (DBMS).* Por tanto, las pruebas deben diseñarse para descubrir errores cometidos al traducir la solicitud del usuario de manera que pueda procesar el DBMS.
2. *La base de datos puede ser remota en relación con el servidor que alberga la webapp.* En consecuencia, deben desarrollarse pruebas que descubran errores en la comunicación entre la *webapp* y la base de datos remota.[3]
3. *Los datos brutos adquiridos de la base de datos deben transmitirse al servidor de la webapp y formatearse de manera adecuada para su posterior transmisión al cliente.* Por tanto, deben desarrollarse pruebas que demuestren la validez de los datos brutos recibidos por el servidor de la *webapp* y también deben crearse pruebas adicionales que demuestren la validez de las transformaciones aplicadas a los datos brutos para crear objetos de contenido válidos.
4. *El objeto de contenido dinámico debe transmitirse al cliente de forma que pueda desplegarse al usuario final.* Por ende, debe diseñarse una serie de pruebas para 1) descubrir errores en el formato del objeto de contenido y 2) probar la compatibilidad con diferentes configuraciones del entorno del cliente.

Al considerar estos cuatro factores, los métodos de diseño de caso de prueba deben aplicarse a cada una de las "capas de interacción" [Ngu01] que se mencionan en la figura 20.2. Las pruebas deben garantizar que 1) información válida pasa entre el cliente y el servidor desde la capa interfaz, 2) la *webapp* procesa los guiones de manera correcta y extrae o formatea adecuadamente los datos del usuario, 3) los datos del usuario pasan correctamente a una función de transformación de datos del lado servidor que formatea consultas adecuadas (por ejemplo, SQL) y 4) las

FIGURA 20.2
Capas de interacción



3 Estos datos pueden volverse complejos cuando se encuentran bases de datos distribuidas o cuando se requiere el acceso a un almacén de datos (capítulo 1).

Cita:

"... es improbable que uno tenga confianza en un sitio web que sufre de constantes periodos de inactividad, que se pasma en medio de una transacción o que tiene una pobre sensación de funcionalidad. Por tanto, las pruebas tienen un papel vital en el proceso de desarrollo global."

Wing Lam

consultas pasan a una capa de gestión de datos[4] que se comunica con las rutinas de acceso a la base de datos (potencialmente ubicadas en otra máquina).

Las capas de transformación de datos, de gestión de datos y de acceso a base de datos que se muestran en la figura 20.2, con frecuencia se construyen con componentes reutilizables que se validaron por separado y como paquete. Si éste es el caso, la prueba de *webapps* se enfoca en el diseño de casos de prueba para ejercitar las interacciones entre la capa cliente y las primeras dos capas servidor (*webapp* y transformación de datos) que se muestran en la figura.

La capa de interfaz de usuario se prueba para garantizar que los guiones se construyeron de manera adecuada para cada consulta de usuario y que transmiten adecuadamente al lado servidor. La capa *webapp* en el lado servidor se prueba para asegurar que los datos de usuario se extraen de manera adecuada de los guiones y que se transmite adecuadamente a la capa de transformación de datos en el lado servidor. Las funciones de transformación de datos se prueban para asegurar que se creó el SQL correcto y que pasó a componentes de gestión de datos adecuados.

Un análisis detallado de la tecnología subyacente que debe comprenderse para diseñar adecuadamente estas pruebas de base de datos está más allá del ámbito de este libro. Si usted tiene interés adicional, vea [Sce02], [Ngu01] y [Bro01].

20.4 Prueba de interfaz de usuario

La verificación y validación de una interfaz de usuario de *webapp* ocurre en tres puntos distintos. Durante el análisis de requerimientos, el modelo de interfaz se revisa para garantizar que se da conformidad a los requerimientos de los participantes y a otros elementos del modelo de requerimientos. Durante el diseño, se revisa el modelo de diseño de interfaz para garantizar que se logran los criterios de calidad genéricos establecidos para todas las interfaces de usuario (capítulo 11) y que los temas de diseño de interfaz específicos de la aplicación se abordaron de manera adecuada. Durante la prueba, la atención se centra en la ejecución de aspectos específicos de la aplicación de la interacción con el usuario, conforme se manifiesten por la sintaxis y la semántica de la interfaz. Además, la prueba proporciona una valoración final de la usabilidad.

CONSEJO

Con excepción de especificaciones orientadas a webapp, la estrategia de interfaz que se anota aquí es aplicable a todo tipo de software cliente-servidor.

20.4.1 Estrategia de prueba de interfaz

La *prueba de interfaz* ejercita los mecanismos de interacción y valida los aspectos estéticos de la interfaz de usuario. La estrategia global para la prueba de interfaz es 1) descubrir errores relacionados con mecanismos de interfaz específicos (por ejemplo, en la ejecución adecuada de un vínculo de menú o en la forma como entran los datos en un formulario) y 2) descubrir errores en la forma como la interfaz implanta la semántica de navegación, la funcionalidad de la *webapp* o el despliegue de contenido. Para lograr esta estrategia, se inician algunos pasos tácticos:

- *Las características de la interfaz se prueban para garantizar que las reglas del diseño, estética y contenido visual relacionado estén disponibles sin error para el usuario.* Las características incluyen tipo de fuente, uso de color, marcos, imágenes, bordes, tablas y características de interfaz relacionadas que se generan conforme avanza la ejecución de la *webapp*.
- *Los mecanismos de interfaz individuales se prueban en forma análoga a la prueba de unidad.* Por ejemplo, las pruebas se diseñan para ejercitar todas las formas, guiones del lado cliente, HTML dinámicos, guiones, contenido de *streaming* (transmisión continua) y

4 La capa de gestión de datos por lo general incorpora una interfaz SQL en el nivel de llamado (SQL-CLI), como Microsoft OLE/ADO o Java Database Connectivity (JDBC).

mecanismos de interfaz específicos de la aplicación (por ejemplo, un carro de mandado para una aplicación de comercio electrónico). En muchos casos, la prueba puede enfocarse exclusivamente en uno de estos mecanismos (la "unidad") y excluir otras características y funciones de interfaz.

- *Cada mecanismo de interfaz se prueba dentro del contexto de un caso de uso o de una unidad semántica de navegación (USN)* (capítulo 13) *para una categoría de usuario específica.* Este enfoque de pruebas es análogo a la prueba de integración porque las pruebas se realizan conforme los mecanismos de interfaz se integran para permitir la ejecución de un caso de uso o USN.
- *La interfaz completa se prueba contra los casos de uso seleccionados y las USN a fin de descubrir errores en la semántica de la interfaz.* Este enfoque de prueba es análogo a la prueba de validación porque el propósito es demostrar conformidad con la semántica de casos de uso o USN específicas. En esta etapa se lleva a cabo una serie de pruebas de usabilidad.
- *La interfaz se prueba dentro de varios entornos (por ejemplo, navegadores) para garantizar que será compatible.* En realidad, esta serie de pruebas también puede considerarse como parte de las pruebas de configuración.

20.4.2 Prueba de mecanismos de interfaz

Cuando un usuario interactúa con una *webapp*, la interacción ocurre a través de uno o más mecanismos de interfaz. En los párrafos que siguen se presenta un breve panorama de las consideraciones de prueba para cada mecanismo de interfaz [Spl01].

La prueba de vínculos externos debe ocurrir durante la vida de la webapp. Parte de una estrategia de apoyo debe ser la calendarización regular de pruebas de vínculos.

Vínculos. Cada vínculo de navegación se prueba para garantizar que se alcanza el objetivo de contenido o función apropiados.[5] Se construye una lista de todos los vínculos asociados con la plantilla de interfaz (por ejemplo, barras de menú e ítems de índice) y luego se ejecuta cada uno individualmente. Además, deben ejercitarse los vínculos dentro de cada objeto de contenido para descubrir URL o vínculos defectuosos con objetos de contenido o funciones inadecuadas. Finalmente, los vínculos con *webapps* externas deben probarse en su precisión y también evaluarse para determinar el riesgo de que se vuelvan inválidos con el tiempo.

Formularios. En un nivel macroscópico, las pruebas se realizan para asegurarse de que 1) las etiquetas identifican correctamente los campos dentro del formulario y los campos obligatorios se identifican visualmente para el usuario, 2) el servidor recibe toda la información contenida dentro del formulario y ningún dato se pierde en la transmisión entre cliente y servidor, 3) se usan valores por defecto adecuados cuando el usuario no selecciona de un menú desplegable o conjunto de botones, 4) las funciones del navegador (por ejemplo, la flecha "retroceso") no corrompen la entrada de datos en un formulario y 5) los guiones que realizan la comprobación de errores en los datos ingresados funcionan de manera adecuada y proporcionan mensajes de error significativos.

En un nivel más dirigido, las pruebas deben garantizar que 1) los campos del formulario tienen ancho y tipos de datos adecuados, 2) el formulario establece salvaguardas adecuadas que prohíben que el usuario ingrese cadenas de texto más largas que cierto máximo predefinido, 3) todas las opciones adecuadas para menús desplegables se especifican y ordenan en forma significativa para el usuario final, 4) las características de "autollenado" del navegador no conducen a errores en la entrada de datos y 5) la tecla de tabulación (o alguna otra) inicia el movimiento adecuado entre los campos del formulario.

5 Estas pruebas pueden realizarse como parte de la prueba de interfaz o de navegación.

Las pruebas de guión en el lado cliente y las pruebas asociadas con HTML dinámico deben repetirse siempre que se libera una nueva versión de un navegador popular.

Guión en el lado cliente. Las pruebas de caja negra se realizan para descubrir cualquier error en el procesamiento conforme se ejecuta el guión. Estas pruebas con frecuencia se acoplan con pruebas de formularios porque la entrada del guión con frecuencia se deriva de los datos proporcionados como parte del procesamiento de formulario. Debe realizarse una prueba de compatibilidad para garantizar que el lenguaje del guión elegido funcionará adecuadamente en las configuraciones de entorno que soporten la *webapp*. Además de probar el guión en sí, Splaine y Jaskiel [Spl01] sugieren que "debe asegurarse de que los estándares [de *webapps*] de la compañía enuncien el lenguaje y versión preferidos del lenguaje de guión que se va a usar para la escritura de guiones en el lado cliente (y en el lado servidor)".

HTML dinámico. Cada página web que contenga HTML dinámico se ejecuta para asegurar que el despliegue dinámico es correcto. Además, debe llevarse a cabo una prueba de compatibilidad para asegurarse que el HTML dinámico funciona adecuadamente en las configuraciones de entorno que soportan la *webapp*.

Ventanas *pop-up*. Una serie de pruebas garantiza que 1) la aparición instantánea tiene el tamaño y posición adecuadas, 2) la aparición no cubre la ventana de la *webapp* original, 3) el diseño estético de la aparición es consistente con el diseño estético de la interfaz y 4) las barras de desplazamiento y otros mecanismos de control anexados a la ventana de aparición se ubican y funcionan de manera adecuada, como se requiere.

Guiones CGI. Las pruebas de caja negra se realizan con énfasis sobre la integridad de los datos (conforme los datos pasan al guión CGI) y del procesamiento del guión (una vez recibidos los datos validados). Además, la prueba de rendimiento puede realizarse para garantizar que la configuración del lado servidor puede alojar las demandas de procesamiento de múltiples invocaciones de los guiones CGI [Spl01].

Contenido de *streaming*. Las pruebas deben demostrar que los datos de *streaming* están actualizados, que se despliegan de manera adecuada y que pueden suspenderse sin error y reanudarse sin dificultad.

***Cookies*.** Se requieren pruebas tanto del lado servidor como del lado cliente. En el primero, las pruebas deben garantizar que una *cookie* se construyó adecuadamente (que contiene datos correctos) y que se transmitió de manera adecuada al lado cliente cuando se solicitó contenido o funcionalidad específico. Además, la persistencia adecuada de la *cookie* se prueba para asegurar que su fecha de expiración es correcta. En el lado cliente, las pruebas determinan si la *webapp* liga adecuadamente las *cookies* existentes a una solicitud específica (enviada al servidor).

Mecanismos de interfaz específicos de aplicación. Las pruebas se siguen conforme una lista de comprobación de funcionalidad y características que se definen mediante el mecanismo de interfaz. Por ejemplo, Splaine y Jaskiel [Spl01] sugieren la siguiente lista de comprobación para la funcionalidad carro de compras definida para una aplicación de comercio electrónico:

- Prueba de frontera (capítulo 18) del número mínimo y máximo de artículos que pueden colocarse en el carro de compras.
- Prueba de una solicitud de "salida" para un carro de compras vacío.
- Prueba de borrado adecuado de un artículo del carro de compras.
- Prueba para determinar si una compra vacía el contenido del carro.
- Prueba para determinar la persistencia del contenido del carro de compras (esto debe especificarse como parte de los requerimientos del cliente).
- Prueba para determinar si la *webapp* puede recordar el contenido del carro de compras en alguna fecha futura (suponiendo que no se realizó compra alguna).

20.4.3 Prueba de la semántica de la interfaz

Una vez que cada mecanismo de interfaz ha sido sometido a prueba de "unidad", la atención de la prueba de interfaz cambia hacia una consideración de la semántica de la interfaz. Esta prueba "evalúa cuán bien cuida el diseño a los usuarios, ofrece instrucciones claras, entrega retroalimentación y mantiene consistencia de lenguaje y enfoque" [Ngu00].

Una revisión profunda del modelo de diseño de interfaz puede proporcionar respuestas parciales a las preguntas implicadas en el párrafo precedente. Sin embargo, cada escenario de caso de uso (para cada categoría de usuario) debe probarse una vez implantada la *webapp*. En esencia, un caso de uso se convierte en la entrada para el diseño de una secuencia de prueba. La intención de la secuencia de prueba es descubrir errores que evitarán que un usuario logre el objetivo asociado con el caso de uso.

Conforme cada caso de uso se prueba, es buena idea mantener una lista de comprobación para asegurar que cada objeto del menú se ejercitó al menos una vez y que se utilizó cada vínculo incrustado dentro de un objeto de contenido. Además, la serie de pruebas debe incluir selección de menú inadecuada y uso de vínculos. La intención es determinar si la *webapp* proporciona manejo y recuperación efectivos del error.

20.4.4 Pruebas de usabilidad

WebRef

En **www.ahref.com/guides/design/199806/0615jef.html** se encuentra una valiosa guía para las pruebas de usabilidad.

La prueba de usabilidad es similar a la de semántica de interfaz (sección 20.4.3) porque también evalúa el grado en el cual los usuarios pueden interactuar efectivamente con la *webapp* y el grado en el que la *webapp* guía las acciones del usuario, proporciona retroalimentación significativa y refuerza un enfoque de interacción consistente. En lugar de enfocarse atentamente en la semántica de algún objetivo interactivo, las revisiones y pruebas de usabilidad se diseñan para determinar el grado en el cual la interfaz de la *webapp* facilita la vida del usuario.[6]

Invariablemente, el ingeniero en software contribuirá con el diseño de las pruebas de usabilidad, pero las pruebas en sí las realizan los usuarios finales. La siguiente secuencia de pasos es aplicable para tal fin [Spl01]:

1. Definir un conjunto de categorías de prueba de usabilidad e identificar las metas de cada una.
2. Diseñar pruebas que permitirán la evaluación de cada meta.
3. Seleccionar a los participantes que realicen las pruebas.
4. Instrumentar la interacción de los participantes con la *webapp* mientras se lleva a cabo la prueba.
5. Desarrollar un mecanismo para valorar la usabilidad de la *webapp*.

La prueba de usabilidad puede ocurrir en varios niveles diferentes de abstracción: 1) puede valorarse la usabilidad de un mecanismo de interfaz específico (por ejemplo, un formulario), 2) puede evaluarse la usabilidad de una página web completa (que abarque mecanismos de interfaz, objetos de datos y funciones relacionadas) y 3) puede considerarse la usabilidad de la *webapp* completa.

El primer paso en la prueba de usabilidad es identificar un conjunto de categorías de usabilidad y establecer los objetivos de la prueba para cada categoría. Las siguientes categorías y objetivos de prueba (escritos en forma de pregunta) ilustran este enfoque:[7]

---

6 En este contexto se ha usado el término *amigable con el usuario*. Desde luego, el problema es que la percepción de un usuario acerca de una interfaz "amigable" puede ser radicalmente diferente a la de otro.

7 Para información adicional acerca de la usabilidad, vea el capítulo 11.

¿Qué características de la usabilidad se convierten en el centro de atención de las pruebas y qué objetivos específicos se señalan?

Interactividad: ¿Los mecanismos de interacción (por ejemplo, menús desplegables, botones, punteros) son fáciles de entender y usar?

Plantilla: ¿Los mecanismos de navegación, contenido y funciones se colocan de forma que el usuario pueda encontrarlos rápidamente?

Legibilidad: ¿El texto está bien escrito y es comprensible?[8] ¿Las representaciones gráficas se entienden con facilidad?

Estética: ¿La plantilla, color, fuente y características relacionadas facilitan el uso? ¿Los usuarios "se sienten cómodos" con la apariencia y el sentimiento de la *webapp*?

Características de despliegue: ¿La *webapp* usa de manera óptima el tamaño y la resolución de la pantalla?

Sensibilidad temporal: ¿Las características, funciones y contenido importantes pueden usarse o adquirir en forma oportuna?

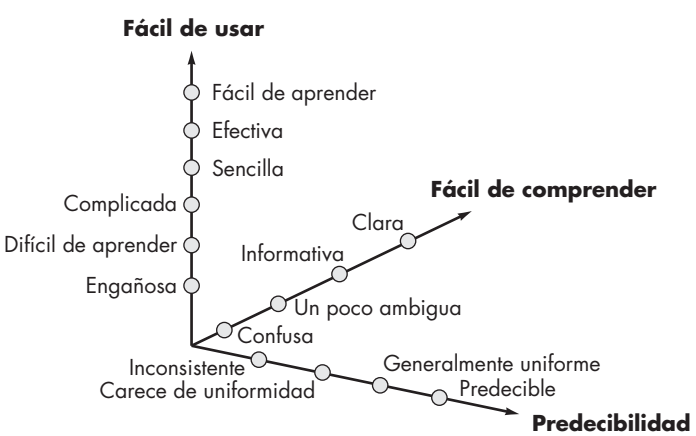
Personalización: ¿La *webapp* se adapta a las necesidades específicas de diferentes categorías de usuario o de usuarios individuales?

Accesibilidad: ¿La *webapp* es accesible a personas que tienen discapacidades?

Dentro de cada una de estas categorías se diseña una serie de pruebas. En algunos casos, la "prueba" puede ser una revisión visual de una página web. En otros, pueden ejecutarse de nuevo pruebas semánticas de la interfaz, pero en esta instancia las preocupaciones por la usabilidad son primordiales.

Como ejemplo, considere la valoración de usabilidad para los mecanismos de interacción e interfaz. Constantine y Lockwood [Con99] sugieren que debe revisarse la siguiente lista de características de interfaz y probar la usabilidad: animación, botones, color, control, diálogo, campos, formularios, marcos, gráficos, etiquetas, vínculos, menús, mensajes, navegación, páginas, selectores, texto y barras de herramientas. Conforme se valora cada característica, es calificada por los usuarios que realizan la prueba sobre una escala cualitativa. La figura 20.3 muestra un posible conjunto de "calificaciones" de valoración que pueden seleccionar los usuarios, mismas que se aplican a cada característica individualmente, a una página web completa o a la *webapp* como un todo.

FIGURA 20.3
Valoración cualitativa de la usabilidad



8 Puede usarse el índice de legibilidad FOG y otros para proporcionar una valoración cuantitativa de la legibilidad. Véase **http://developer.gnome.org/documents/usability/usability-readability.html** para más detalles.

20.4.5 Pruebas de compatibilidad

Diferentes computadoras, dispositivos de despliegue, sistemas operativos, navegadores y velocidades de conexión de red pueden tener influencia significativa sobre la operación de una *webapp*. Cada configuración de cómputo puede dar como resultado diferencias en velocidades de procesamiento en el lado cliente, en resolución de despliegue y en velocidades de conexión. Los caprichos de los sistemas operativos en ocasiones pueden producir conflictos de procesamiento en la *webapp*. En ocasiones, diferentes navegadores producen resultados ligeramente distintos, sin importar el grado de estandarización HTML dentro de la *webapp*. Los plug-ins requeridos pueden o no conseguirse con facilidad para una configuración particular.

PUNTO CLAVE

Las *webapps* se ejecutan dentro de varios entornos en el lado cliente. El objetivo de la prueba de compatibilidad es descubrir errores asociados con un entorno específico (por ejemplo, navegador).

En algunos casos, pequeños conflictos de compatibilidad no representan problemas significativos, pero en otros pueden encontrarse serios errores. Por ejemplo, las velocidades de descarga pueden volverse inaceptables, carecer de un plug-in requerido puede hacer que el contenido no esté disponible, las diferencias de navegador pueden cambiar dramáticamente la plantilla de la página, los estilos de fuente pueden alterarse y volverse ilegibles o los formularios pueden organizarse de manera inadecuada. La *prueba de compatibilidad* busca descubrir dichos problemas antes de que la *webapp* esté en línea.

El primer paso en la prueba de compatibilidad es definir un conjunto de configuraciones de cómputo, y sus variantes, que "se encuentran comúnmente" en el lado cliente. En esencia, se crea una estructura de árbol, identificación de cada plataforma de cómputo, dispositivos de despliegue usuales, sistemas operativos aceptados en la plataforma, navegadores disponibles, probables velocidades de conexión a internet e información similar. A continuación se deriva una serie de pruebas de validación de compatibilidad, con frecuencia adaptadas de pruebas de interfaz existentes, de navegación, de rendimiento y de seguridad. La intención de estas pruebas es descubrir errores o problemas de ejecución que pueden rastrearse para identificar diferencias de configuración.

CASASEGURA

Prueba de webapp

La escena: Oficina de Doug Miller.

Participantes: Doug Miller (gerente del grupo de ingeniería del software *CasaSegura*) y Vinod Raman (miembro del equipo de ingeniería del software del producto).

La conversación:

Doug: ¿Qué piensas de la versión 0.0 de la *webapp* de comercio electrónico para **CasaSeguraAsegurada.com**?

Vinod: El proveedor subcontratado hizo un buen trabajo. Sharon [gerente de desarrollo del proveedor] me dijo que ahora están haciendo pruebas.

Doug: Me gustaría que tú y el resto del equipo hicieran una prueba un poco informal en el sitio de comercio electrónico.

Vinod (hace muecas): Creo que vamos a contratar una compañía externa para validar la *webapp*. Todavía nos estamos matando en el intento de poner el producto en línea.

Doug: Vamos a contratar a un proveedor de pruebas para las pruebas de rendimiento y seguridad, y nuestro proveedor subcontratado ya está haciendo pruebas. Sólo pienso que otro punto de vista sería útil y, además, nos gustaría conservar los costos en línea, de modo que...

Vinod (suspira): ¿Qué buscas?

Doug: Quiero estar seguro de que la interfaz y toda la navegación son sólidas.

Vinod: Supongo que podemos comenzar con los casos de uso para cada una de las principales funciones de interfaz:

Aprenda acerca de *CasaSegura*.
Especifique el sistema *CasaSegura* que necesita.
Compre un sistema *CasaSegura*.
Obtenga soporte técnico.

Doug: Bien. Pero sigan las rutas de navegación durante todo el trayecto hasta su conclusión.

Vinod (observa un cuaderno de casos de uso): Sí, cuando seleccionas Especifique el sistema CasaSegura que necesita, eso te llevará hacia:

Seleccione componentes CasaSegura.

Obtenga recomendaciones de componentes *CasaSegura*

Podemos ejercitar la semántica de cada ruta.

Doug: Mientras estás ahí, verifica el contenido que aparece en cada nodo de navegación.

Vinod: Desde luego... y los elementos funcionales también. ¿Quién prueba la usabilidad?

Doug: Oh... el proveedor examinador coordinará la prueba de usabilidad. Contratamos una firma de investigación de mercado a fin de alinear 20 usuarios comunes para al estudio de usabilidad, pero si tus chicos descubren algún conflicto de usabilidad...

Vinod: Ya sé, pásenlos de largo.

Doug: Gracias, Vinod.

20.5 Prueba en el nivel de componente

La *prueba en el nivel de componente*, también llamada *prueba de función*, se enfoca en un conjunto de pruebas que intentan descubrir errores en funciones de las *webapps*. Cada función de una *webapp* es un componente de software (implantado en uno de varios lenguajes de programación o lenguajes de guiones) y puede probarse usando técnicas de caja negra (y en algunos casos de caja blanca), como se estudió en el capítulo 18.

Los casos de prueba en el nivel de componente con frecuencia se derivan de la entrada a formularios. Una vez definidos los datos de los formularios, el usuario selecciona un botón u otro mecanismo de control para iniciar la ejecución. Son usuales los siguientes métodos de diseño de caso de prueba (capítulo 18):

- *Partición de equivalencia.* El dominio de entrada de la función se divide en categorías o clases de entrada a partir de las cuales se derivan casos de prueba. El formulario de entrada se valora para determinar cuáles clases de datos son relevantes para la función. Los casos de prueba para cada clase de entrada se derivan y ejecutan, mientras que otras clases de entrada se mantienen constantes. Por ejemplo, una aplicación de comercio electrónico puede implantar una función que calcule los cargos de embarque. Entre una variedad de información de embarque proporcionada mediante un formulario, está el código postal del usuario. Los casos de prueba se diseñan con la intención de descubrir errores en el procesamiento del código postal al especificar valores de código postal que puedan descubrir diferentes clases de errores (por ejemplo, un código postal incompleto, un código postal incorrecto, un código postal inexistente, un formato de código postal erróneo).
- *Análisis de valor de frontera.* Los datos de los formularios se prueban en sus fronteras. Por ejemplo, la función de cálculo de embarque anotada anteriormente solicita el número máximo de días requeridos para la entrega del producto. En el formulario se anota un mínimo de 2 días y un máximo de 14. Sin embargo, las pruebas de valor de frontera pueden ingresar valores de 0, 1, 2, 13, 14 y 15 para determinar cómo reacciona la función de datos en y afuera de las fronteras de entrada válida.[9]
- *Prueba de rutas.* Si la complejidad lógica de la función es alta,[10] puede usarse la prueba de rutas (un método de diseño de casos de prueba de caja blanca) para garantizar que se ejercitó cada ruta independiente en el programa.

Además de estos métodos de diseño de casos de prueba, se usa una técnica llamada *prueba de error forzado* [Ngu01] para derivar casos de prueba que a propósito conducen al componente

9 En este caso, un mejor diseño de entrada puede eliminar errores potenciales. El número máximo de días podría seleccionarse de un menú desplegable, lo que impide al usuario especificar entrada fuera de fronteras.

10 La complejidad lógica puede determinarse al calcular la complejidad ciclomática del algoritmo. Vea el capítulo 18 para detalles adicionales.

web a una condición de error. El propósito es descubrir los errores que ocurren durante la manipulación del error (por ejemplo, mensajes de error incorrectos o inexistentes, falla de la *webapp* como consecuencia del error, salida errónea activada por entrada errónea, efectos colaterales que se relacionan con el procesamiento de componentes).

Cada caso de prueba en el nivel componente especifica todos los valores de entrada y salida que se espera que proporcione el componente. La salida real producida como consecuencia de la prueba se registra para futuras referencias durante el soporte y el mantenimiento.

En muchas situaciones, la ejecución correcta de la función de una *webapp* se liga a la interfaz adecuada con una base de datos que puede ser externa a la *webapp*. Por tanto, la prueba de base de datos se convierte en parte integral del régimen de prueba de componente.

20.6 PRUEBA DE NAVEGACIÓN

Un usuario viaja a través de una *webapp* en forma muy parecida a como un visitante camina a través de una tienda o de un museo. Existen muchas rutas que pueden tomarse, muchas paradas que pueden realizarse, muchas cosas que aprender y mirar, actividades por iniciar y decisiones por tomar. Este proceso de navegación es predecible porque cada visitante tiene un conjunto de objetivos cuando llega. Al mismo tiempo, el proceso de navegación puede ser impredecible porque el visitante, influido por algo que ve o aprende, puede elegir una ruta o iniciar una acción que no es usual conforme el objetivo original. La labor de la prueba de navegación es 1) garantizar que son funcionales todos los mecanismos que permiten al usuario de la *webapp* recorrerla y 2) validar que cada unidad semántica de navegación (USN) pueda lograr la categoría de usuario apropiada.

20.6.1 Prueba de sintaxis de navegación

Cita:

"No estamos perdidos. Tenemos un desafío posicional."

John M. Ford

La primera fase de la prueba de navegación en realidad comienza durante la prueba de interfaz. Los mecanismos de navegación se prueban para asegurarse de que cada interfaz realiza la función que se le ha encargado. Splaine y Jaskiel [Spl01] sugieren que debe probarse cada uno de los siguientes mecanismos de navegación:

- *Vínculos de navegación:* estos mecanismos incluyen vínculos internos dentro de la *webapp*, vínculos externos hacia otras *webapps* y anclas dentro de una página web específica. Cada vínculo debe ser probado para asegurarse de que se alcanza el contenido o funcionalidad adecuados cuando se elige el vínculo.
- *Redirecciones:* estos vínculos entran en juego cuando un usuario solicita una URL inexistente o cuando selecciona un vínculo cuyo contenido se removió o cuyo nombre cambió. Se despliega un mensaje para el usuario y la navegación se redirige hacia otra página (por ejemplo, la página de inicio). Los redireccionamientos deben probarse al solicitar vínculos internos incorrectos o URL externas y debe valorarse cómo maneja la *webapp* estas solicitudes.
- *Marcas de página (favoritos, bookmarks):* aunque las marcas de página son función del navegador, la *webapp* debe probarse para garantizar la extracción de un título de página significativo conforme se crea la marca.
- *Marcos y framesets:* cada marco incluye el contenido de una página web específica; un frameset contiene múltiples marcos y habilita el despliegue de múltiples páginas web al mismo tiempo. Puesto que es posible anidar marcos y framesets unos dentro de otros, estos mecanismos de navegación y despliegue deben probarse para que tengan el contenido correcto, la plantilla y tamaño adecuados, rendimiento de descargas y compatibilidad de navegador.

- *Mapas de sitio:* un mapa de sitio proporciona una tabla de contenido completa para todas las páginas web. Cada entrada del mapa de sitio debe probarse para garantizar que los vínculos llevan al usuario al contenido o funcionalidad adecuados.
- *Motores de búsqueda internos:* las *webapps* complejas con frecuencia contienen cientos o incluso miles de objetos de contenido. Un motor de búsqueda interno permite al usuario realizar una búsqueda de palabra clave dentro de la *webapp* para encontrar el contenido necesario. La prueba del motor de búsqueda valida la precisión y completitud de la búsqueda, las propiedades de manejo de error del motor de búsqueda y las características de búsqueda avanzadas (por ejemplo, el uso de operadores booleanos en el campo de búsqueda).

Algunas de las pruebas anotadas pueden realizarse mediante herramientas automatizadas (por ejemplo, comprobación de vínculos), mientras que otras se diseñan y ejecutan manualmente. La intención de principio a fin es garantizar que los errores en la mecánica de navegación se encuentran antes de que la *webapp* entre en línea.

20.6.2 Prueba de la semántica de navegación

? ¿Qué preguntas deben plantearse y responderse conforme se prueba cada USN?

En el capítulo 13, una unidad semántica de navegación (USN) se define como "un conjunto de estructuras de información y navegación relacionada que colaboran en el cumplimiento de un subconjunto de requerimientos de usuario relacionados" [Cac02]. Cada USN se define mediante un conjunto de trayectorias de navegación (llamadas "rutas de navegación") que conectan los nodos de navegación (por ejemplo, páginas web, objetos de contenido o funcionalidad). Considerado como un todo, cada USN permite a un usuario lograr requerimientos específicos definidos por uno o más casos de uso para una categoría de usuario. La prueba de navegación ejercita cada USN para asegurarse de que dichos requerimientos pueden lograrse. Es necesario responder las siguientes preguntas conforme se prueba cada USN:

- ¿La USN se logra en su totalidad sin error?
- ¿Todo nodo de navegación (definido por una USN) se alcanza dentro del contexto de las rutas de navegación definidas por la USN?
- Si la USN puede lograrse usando más de una ruta de navegación, ¿se probó cada ruta relevante?
- Si la interfaz de usuario proporciona una guía para auxiliar en la navegación, ¿las instrucciones son correctas y comprensibles conforme avanza la navegación?
- ¿Existe un mecanismo (distinto a la flecha "retroceso" del navegador) para regresar al nodo de navegación anterior y al comienzo de la ruta de navegación?
- ¿Los mecanismos de navegación dentro de un gran nodo de navegación (es decir, una página web grande) funcionan de manera adecuada?
- Si una función debe ejecutarse en un nodo y el usuario elige no proporcionar entrada, ¿el resto de la USN puede completarse?
- Si una función se ejecuta en un nodo y ocurre un error en el procesamiento de la función, ¿la USN puede completarse?
- ¿Existe alguna forma para descontinuar la navegación antes de que todos los nodos se hayan alcanzado, pero luego regresar a donde se descontinuó la navegación y avanzar desde ahí?
- ¿Todo nodo es alcanzable desde el mapa de sitio? ¿Los nombres de nodo son significativos para los usuarios finales?

Si no se han creado USN como parte del análisis o el diseño de la webapp, puede aplicar casos de uso para el diseño de casos de prueba de navegación. El mismo conjunto de preguntas se plantean y responden.

- Si un nodo dentro de una USN se alcanza desde alguna fuente externa, ¿es posible avanzar hacia el nodo siguiente en la ruta de navegación? ¿Es posible regresar al nodo anterior en la ruta de navegación?
- ¿El usuario entiende su ubicación dentro de la arquitectura de contenido conforme se ejecuta la USN?

La prueba de navegación, como las pruebas de interfaz y usabilidad, debe realizarse por tantos departamentos como sea posible. Usted tiene la responsabilidad durante las primeras etapas de la prueba de navegación, pero las etapas posteriores las deben realizar otros participantes en el proyecto, un equipo de prueba independiente y, a final de cuentas, usuarios no técnicos. La intención es ejercitar la navegación de la *webapp* a profundidad.

20.7 PRUEBA DE CONFIGURACIÓN

La variabilidad y la inestabilidad de la configuración son factores importantes que hacen de la prueba de *webapps* un desafío. El hardware, los sistemas operativos, navegadores, capacidad de almacenamiento, velocidades de comunicación de red y varios otros factores en el lado cliente son difíciles de predecir para cada usuario. Además, la configuración para un usuario dado puede cambiar de manera regular [por ejemplo, actualizaciones del sistema operativo (OS), nuevos ISP y velocidades de conexión]. El resultado puede ser un entorno lado cliente que es proclive a errores sutiles y significativos. La impresión que un usuario tiene de la *webapp* y la forma en la que interactúa con ella pueden diferir significativamente de la experiencia de otro usuario si ambos usuarios no trabajan dentro de la misma configuración en el lado cliente.

La labor de la prueba de configuración no es ejercitar toda configuración posible en el lado cliente. En vez de ello, es probar un conjunto de probables configuraciones en los lados cliente y servidor para garantizar que la experiencia del usuario será la misma en todos ellos y que aislará los errores que puedan ser específicos de una configuración particular.

20.7.1 Conflictos en el lado servidor

En el lado servidor, los casos de prueba de configuración se diseñan para verificar que la configuración servidor proyectada [es decir, servidor *webapp*, servidor de base de datos, sistemas operativos, software de firewall (cortafuegos), aplicaciones concurrentes] pueden soportar la *webapp* sin error. En esencia, la *webapp* se instaló dentro del entorno del lado servidor y se probó para asegurar que opera sin error.

? ¿Qué preguntas deben plantearse y responderse conforme se realiza la prueba de configuración en el lado servidor?

Conforme se diseñan las pruebas de configuración del lado servidor, debe considerarse cada componente de la configuración del servidor. Entre las preguntas que deben plantearse y responderse durante la prueba de configuración del lado servidor se encuentran:

- ¿La *webapp* es completamente compatible con el servidor OS?
- ¿Los archivos de sistema, directorios y datos de sistema relacionados se crean correctamente cuando la *webapp* es operativa?
- ¿Las medidas de seguridad del sistema (por ejemplo, *firewalls* o encriptado) permiten a la *webapp* ejecutarse y atender a los usuarios sin interferencia o degradación del rendimiento?
- ¿La *webapp* se probó con la configuración de servidor distribuido[11] (si existe alguno) que se eligió?

11 Por ejemplo, puede usarse un servidor de aplicación separado de un servidor de base de datos. La comunicación entre las dos máquinas ocurre a través de una conexión de red.

- ¿La *webapp* se integró adecuadamente con el software de base de datos? ¿La *webapp* es sensible a diferentes versiones del software de base de datos?
- ¿Los guiones de la *webapp* en el lado servidor se ejecutan adecuadamente?
- ¿Los errores del administrador del sistema se examinaron en sus efectos sobre las operaciones de la *webapp*?
- Si se usan servidores proxy, ¿las diferencias en su configuración se abordaron con pruebas en sitio?

20.7.2 Conflictos en el lado cliente

En el lado cliente, las pruebas de configuración se enfocan con más peso en la compatibilidad de la *webapp* con las configuraciones que contienen una o más permutas de los siguientes componentes [Ngu01]:

- *Hardware:* CPU, memoria, almacenamiento y dispositivos de impresión
- *Sistemas operativos:* Linux, Macintosh OS, Microsoft Windows, un OS móvil
- *Software navegador:* Firefox, Safari, Internet Explorer, Opera, Chrome y otros
- *Componentes de interfaz de usuario:* Active X, Java applets y otros
- *Plug-ins:* QuickTime, RealPlayer y muchos otros
- *Conectividad:* cable, DSL, módem regular, T1, WiFi

Además de estos componentes, otras variables incluyen software de redes, caprichos de la ISP y aplicaciones que corren de manera concurrente.

Para diseñar pruebas de configuración en el lado cliente, debe reducir el número de variables de configuración a un número manejable.[12] Para lograr esto, cada categoría de usuario se valora para determinar las probables configuraciones que pueden encontrarse dentro de la categoría. Además, pueden usarse datos de participación de mercado para predecir las combinaciones de componentes más probables. Entonces la *webapp* se prueba dentro de estos entornos.

20.8 PRUEBA DE SEGURIDAD

Cita:

"Internet es un lugar riesgoso para realizar negocios o almacenar valores. Hackers, crackers, snoops, spoofers... vándalos, lanzadores de virus y proveedores de programas maliciosos corren a sus anchas."

Dorothy y Peter Denning

La seguridad de la *webapp* es un tema complejo que debe comprenderse por completo antes de que pueda lograrse una prueba de seguridad efectiva.[13] Las *webapps* y los entornos en los lados cliente y servidor donde se albergan representan un blanco atractivo para *hackers* externos, empleados descontentos, competidores deshonestos y para quien quiera robar información sensible, modificar contenido maliciosamente, degradar el rendimiento, deshabilitar la funcionalidad o avergonzar a una persona, organización o negocio.

Las pruebas de seguridad se diseñan para sondear las vulnerabilidades del entorno lado cliente, las comunicaciones de red que ocurren conforme los datos pasan de cliente a servidor y viceversa, y el entorno del lado servidor. Cada uno de estos dominios puede atacarse, y es tarea del examinador de seguridad descubrir las debilidades que puedan explotar quienes tengan intención de hacerlo.

En el lado cliente, las vulnerabilidades con frecuencia pueden rastrearse en errores preexistentes en navegadores, programas de correo electrónico o software de comunicación. Nguyen [Ngu01] describe un hueco de seguridad común:

12 Aplicar pruebas en toda combinación posible de componentes de configuración consume demasiado tiempo.

13 Los libros de Cross y Fischer [Cro07], Andrews y Whittaker [And06] y Trivedi [Tri03] proporcionan información útil acerca del tema.

Si la webapp es crucial para el negocio, mantiene datos sensibles o es un blanco probable de los hackers, es buena idea subcontratar pruebas de seguridad con un proveedor que se especialice en ellas.

Uno de los errores comúnmente mencionados es el desbordamiento de buffer, que permite que código malicioso se ejecute en la máquina cliente. Por ejemplo, ingresar una URL en un navegador que es mucho más larga que el tamaño de buffer asignado para la URL provocará un error de sobreescritura de memoria (desbordamiento de buffer) si el navegador no tiene código de detección de error para validar la longitud de la URL ingresada. Un hacker experimentado puede explotar astutamente este error al escribir una URL larga con código que se va a ejecutar y que puede hacer que el navegador derribe o altere las configuraciones de seguridad (de alto a bajo) o, peor aún, corromper datos del usuario.

Otra vulnerabilidad potencial en el lado cliente es el acceso no autorizado a las *cookies* colocadas dentro del navegador. Los sitios web creados con intenciones maliciosas pueden adquirir información contenida dentro de *cookies* legítimas y usar esta información en formas que ponen en riesgo la privacidad del usuario o, peor aún, que montan el escenario para el robo de identidad.

Los datos comunicados entre el cliente y el servidor son vulnerables al *spoofing* (engaño). El *spoofing* ocurre cuando un extremo de la ruta de comunicación se trastorna por una entidad con intenciones maliciosas. Por ejemplo, un usuario puede ser engañado por un sitio malicioso que actúa como si fuese el servidor de la *webapp* legítimo (apariencia y sensación idénticas). La intención es robar contraseñas, información personal o datos de crédito.

En el lado servidor, las vulnerabilidades incluyen ataques de negación de servicio y guiones maliciosos que pueden pasar hacia el lado cliente o usarse para deshabilitar operaciones del servidor. Además, puede accederse sin autorización a las bases de datos en el lado servidor (robo de datos).

Las pruebas de seguridad deben diseñarse para ejercitar *firewalls*, autenticación, encriptado y autorización.

Para proteger contra éstas (y muchas otras) vulnerabilidades, se implanta uno o más de los siguientes elementos de seguridad [Ngu01]:

- *Firewall:* mecanismo de filtrado, que es una combinación de hardware y software que examina cada paquete de información entrante para asegurarse de que proviene de una fuente legítima y que bloquea cualquier dato sospechoso.
- *Autenticación:* mecanismo de verificación que valida la identidad de todos los clientes y servidores, y permite que la comunicación ocurra solamente cuando ambos lados se verifican.
- *Encriptado:* mecanismo de codificación que protege los datos sensibles al modificarlos de forma que hace imposible leerlos por quienes tienen intenciones maliciosas. El encriptado se fortalece usando *certificados digitales* que permiten al cliente verificar el destino al que se transmiten los datos.
- *Autorización:* mecanismo de filtrado que permite el acceso al entorno cliente o servidor sólo a aquellos individuos con códigos de autorización apropiados (por ejemplo, ID de usuario y contraseña).

Las pruebas de seguridad deben diseñarse para sondear cada una de estas tecnologías de seguridad con la intención de descubrir huecos en la seguridad.

El diseño real de las pruebas de seguridad requiere conocimiento profundo del trabajo interno de cada elemento de seguridad y amplia comprensión de una gran gama de tecnologías de redes. En muchos casos, la prueba de seguridad se subcontrata con firmas que se especializan en dichas tecnologías.

20.9 PRUEBA DE RENDIMIENTO

Nada es más frustrante que una *webapp* que tarda minutos en cargar contenido cuando sitios de la competencia descargan contenido similar en segundos. Nada es más exasperante que

intentar ingresar a una *webapp* y recibir un mensaje de "servidor ocupado", con la sugerencia de que se intente de nuevo más tarde. Nada es más desconcertante que una *webapp* que responde instantáneamente en algunas situaciones y luego en otras parece caer en un estado de espera infinita. Estos eventos suceden en la web todos los días y todos ellos se relacionan con el rendimiento.

Las *pruebas de rendimiento* se usan para descubrir problemas de rendimiento que pueden ser resultado de: falta de recursos en el lado servidor, red con ancho de banda inadecuada, capacidades de base de datos inadecuadas, capacidades de sistema operativo deficientes o débiles, funcionalidad de *webapp* pobremente diseñada y otros conflictos de hardware o software que pueden conducir a rendimiento cliente-servidor degradado. La intención es doble: 1) comprender cómo responde el sistema conforme aumenta la *carga* (es decir, número de usuarios, número de transacciones o volumen de datos global) y 2) recopilar mediciones que conducirán a modificaciones de diseño para mejorar el rendimiento.

20.9.1 Objetivos de la prueba de rendimiento

Algunos aspectos del rendimiento de la webapp, al menos como los percibe el usuario final, son difíciles de poner a prueba. La carga de la red, los caprichos del hardware de interfaz con la red y conflictos similares no son fáciles de poner a prueba en la webapp.

Las pruebas de rendimiento se diseñan para simular situaciones de carga del mundo real. Conforme aumenta el número de usuarios simultáneos de la *webapp* o el número de transacciones en línea o la cantidad de datos (descargados o subidos), las pruebas de rendimiento ayudarán a responder las siguientes preguntas:

- ¿El tiempo de respuesta del servidor se degrada a un punto donde es apreciable e inaceptable?
- ¿En qué punto (en términos de usuarios, transacciones o carga de datos) el rendimiento se vuelve inaceptable?
- ¿Qué componentes del sistema son responsables de la degradación del rendimiento?
- ¿Cuál es el tiempo de respuesta promedio para los usuarios bajo diversas condiciones de carga?
- ¿La degradación del rendimiento tiene impacto sobre la seguridad del sistema?
- ¿La confiabilidad o precisión de la *webapp* resulta afectada conforme crece la carga sobre el sistema?
- ¿Qué sucede cuando se aplican cargas que son mayores que la capacidad máxima del servidor?
- ¿La degradación del rendimiento tiene impacto sobre los ingresos de la compañía?

Para desarrollar respuestas a estas preguntas, se realizan dos tipos diferentes de pruebas de rendimiento: 1) la *prueba de carga* examina la carga del mundo real en varios niveles de carga y en varias combinaciones, y 2) la *prueba de esfuerzo* fuerza a aumentar la carga hasta el punto de rompimiento para determinar cuánta capacidad puede manejar el entorno de la *webapp*. Cada una de estas estrategias de prueba se considera en las secciones siguientes.

20.9.2 Prueba de carga

La intención de la prueba de carga es determinar cómo responderán las *webapps* y su entorno del lado servidor a varias condiciones de carga. Conforme avanzan las pruebas, las permutas de las siguientes variables definen un conjunto de condiciones de prueba:

N, número de usuarios concurrentes

T, número de transacciones en línea por unidad de tiempo

D, carga de datos procesados por el servidor en cada transacción

Si una webapp usa múltiples servidores para proporcionar una capacidad significativa, la prueba de carga debe realizarse en un entorno multiservidor.

En todo caso, dichas variables se definen dentro de fronteras operativas normales del sistema. Conforme se aplica cada condición de prueba, se recopila una o más de las siguientes medidas: respuesta de usuario promedio, tiempo promedio para descargar una unidad estandarizada de datos y tiempo promedio para procesar una transacción. Estas medidas deben examinarse para determinar si una disminución abrupta en el rendimiento puede rastrearse en una combinación específica de *N*, *T* y *D*.

La prueba de carga también puede usarse para valorar las velocidades de conexión recomendadas para los usuarios de la *webapp*. El rendimiento global, *P*, se calcula de la forma siguiente:

$$P = N \times T \times D$$

Tome como ejemplo un sitio popular de noticias deportivas. En un momento dado, 20 000 usuarios concurrentes envían una solicitud (una transacción, *T*) una vez cada 2 minutos en promedio. Cada transacción requiere que la *webapp* descargue un nuevo artículo que promedia 3 Kb de longitud. Por tanto, el rendimiento global puede calcularse como:

$$P = [20\,000 \times 0.5 \times 3\text{Kb}]/60 = 500 \text{ Kbytes/s}$$
$$= 4 \text{ megabits por segundo}$$

Por ende, la conexión de la red para el servidor tendría que soportar esta tasa de datos y debería ponerse a prueba para asegurarse de que lo hace.

20.9.3 Prueba de esfuerzo

El objetivo de la prueba de esfuerzo es comprender de mejor manera como falla un sistema a medida que es forzado más allá de sus límites operacionales.

La *prueba de esfuerzo* es una continuación de la prueba de carga, pero en esta instancia las variables *N*, *T* y *D* se fuerzan a satisfacerse y luego se superan los límites operativos. La intención de estas pruebas es responder a cada una de las siguientes preguntas:

- ¿El sistema se degrada "suavemente" o el servidor se apaga conforme la capacidad se supera?
- ¿El software servidor genera mensajes "servidor no disponible"? De manera más general, ¿los usuarios están conscientes de que no pueden llegar al servidor?
- ¿El servidor pone en cola los recursos solicitados y vacía la cola una vez que disminuye la demanda de capacidad?
- ¿Las transacciones se pierden conforme la capacidad se excede?
- ¿La integridad de los datos resulta afectada conforme la capacidad se excede?
- ¿Qué valores de *N*, *T* y *D* fuerzan el fallo del entorno servidor? ¿Cómo se manifiesta la falla? ¿Se envían notificaciones automáticas al personal de apoyo técnico en el sitio servidor?
- Si el sistema falla, ¿cuánto tiempo tardará en regresar en línea?
- ¿Ciertas funciones de la *webapp* (por ejemplo, funcionalidad de cálculo intenso, capacidades de transmisión de datos) quedan descontinuadas conforme la capacidad alcanza el nivel de 80 o 90 por ciento?

A una variación de las pruebas de esfuerzo en ocasiones se le conoce como *prueba pico/rebote* (*spike/bounce*) [Spl01]. En este régimen de pruebas, la carga alcanza un pico de capacidad, luego se baja rápidamente a condiciones operativas normales y después alcanza de nuevo un pico. Al rebotar la carga del sistema, es posible determinar cuán bien el servidor puede ordenar los recursos para satisfacer una demanda muy alta y entonces liberarlos cuando reaparecen condiciones normales (de modo que esté listo para el siguiente pico).

HERRAMIENTAS DE SOFTWARE

Taxonomía de herramientas para prueba de webapp

En su ensayo acerca de las pruebas de los sistemas de comercio electrónico, Lam [Lam01] presenta una útil taxonomía de herramientas automáticas que tienen aplicabilidad directa para probar en un contexto de ingeniería web. Se anexan las herramientas representativas en cada categoría.[14]

Las **herramientas de configuración y gestión de contenido** gestionan la versión y cambian el control de los objetos de contenido web y los componentes funcionales.
Herramientas representativas: Una lista amplia se encuentra en **www.daveeaton.com/scm/CMTools.html**

Las **herramientas de rendimiento de base de datos** miden el rendimiento de la base de datos, como el tiempo para realizar consultas seleccionadas en bases de datos. Dichas herramientas facilitan la optimización de la base de datos.
Herramientas representativas: BMC Software (**www.bmc.com**)

Los **depuradores** son herramientas de programación usuales que encuentran y resuelven defectos de software en el código. Son parte de la mayoría de los entornos modernos de desarrollo de aplicaciones.
Herramientas representativas:
Accelerated Technology (**www.acceleratedtechnology.com**)
Apple Debugging Tools (**developer.apple.com/tools/performance/**)
IBM VisualAge Environment (**www.ibm.com**)
Microsoft Debugging Tools (**www.microsoft.com**)

Los **sistemas de gestión de defecto** registran los defectos y rastrean su estado y resolución. Algunos incluyen herramientas de reporte para ofrecer información de gestión acerca de la dispersión del defecto y tasas de resolución del mismo.
Herramientas representativas:
EXCEL Quickbigs (**www.excelsoftware.com**)
ForeSoft BugTrack (**www.bugtrack.net**)
McCabe TRUETrack (**www.mccabe.com**)

Las **herramientas de monitoreo de red** vigilan el nivel de tráfico en la red. Son útiles para identificar cuellos de botella en la red y probar el vínculo entre sistemas frontales y traseros.
Herramientas representativas:
Una lista exhaustiva se encuentra en **www.slac.stanford.edu/xorg/nmtf/nmtf-tools.html**

Las **herramientas de prueba de regresión** almacenan casos de prueba y datos de prueba, y pueden volver a aplicar los casos de prueba después de cambios sucesivos de software.
Herramientas representativas:
Compuware QARun (**www.compuware.com/products/qacenter/qarun**)
Rational VisualTest (**www.rational.com**)
Seque Software (**www.seque.com**)

Las **herramientas de monitoreo de sitio** monitorean el rendimiento de un sitio, con frecuencia desde la perspectiva del usuario. Se usan para recopilar estadísticas tales como tiempo de respuesta extremo a extremo y rendimiento global, y para comprobar periódicamente la disponibilidad de un sitio.
Herramientas representativas:
Keynote Systems (**www.keynote.com**)

Las **herramientas de esfuerzo** ayudan a los desarrolladores a explorar el comportamiento del sistema bajo niveles altos de uso operativo y encuentran los puntos de ruptura de un sistema.
Herramientas representativas:
Mercury Interactive (**www.merc-int.com**)
Herramientas de prueba de fuente abierta (**www.opensourcetesting.org/performance .php**)
Web Performance Load Tester (**www.webperformanceinc.com**)

Los **monitores de recursos del sistema** son parte de la mayoría de los servidores OS y software servidor web; monitorean recursos tales como espacio de disco, uso de CPU y memoria.
Herramientas representativas:
Successful Hosting.com (**www.successfulhosting.com**)
Quest Software Foglight (**www.quest.com**)

Las **herramientas de generación de datos de prueba** auxilian a los usuarios en la generación de datos de prueba.
Herramientas representativas:
Una lista exhaustiva se encuentra en **www.softwareqatest.com/qatweb1.html**

Los **comparadores de resultado de prueba** ayudan a comparar los resultados de un conjunto de pruebas con los de otro conjunto. Se usan para comprobar que los cambios de código no han introducido cambios adversos en el comportamiento del sistema.
Herramientas representativas:
Una lista útil se encuentra en **www.aptest.com/resources.html**

Los **monitores de transacción** miden el rendimiento de los sistemas de procesamiento de alto volumen de transacciones.
Herramientas representativas:
QuotiumPro (**www.quotium.com**)
Software Research eValid (**www.soft.com/eValid/index.html**)

Las **herramientas de seguridad website** ayudan a detectar potenciales problemas de seguridad. Con frecuencia puede establecerse un sondeo de seguridad y herramientas de monitoreo para correr sobre una base calendarizada.
Herramientas representativas:
Una lista amplia se encuentra en **www.timberlinetechnologies.com/products/www.html**

14 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas que existen en esta categoría. Además, los nombres de las herramientas son marcas registradas de las compañías señaladas.

20.10 Resumen

La meta de la prueba de las *webapps* es ejercitar cada una de las muchas dimensiones de calidad de la *webapp* con la intención de encontrar errores o descubrir conflictos que puedan conducir a fallas en la calidad. Las pruebas se enfocan en contenido, función, estructura, usabilidad, navegabilidad, rendimiento, compatibilidad, interacción, capacidad y seguridad. Estas pruebas incorporan revisiones que ocurren conforme se diseña la *webapp* y pruebas que se llevan a cabo una vez que se implanta la misma.

La estrategia de prueba de *webapps* ejercita cada dimensión de calidad al examinar inicialmente "unidades" de contenido, funcionalidad o navegación. Una vez validadas las unidades individuales, la atención se cambia hacia pruebas que ejercitan la *webapp* como un todo. Para lograr esto, muchas pruebas se derivan desde la perspectiva del usuario y se activan mediante la información contenida en casos de uso. Se desarrolla un plan de prueba de *webapps* y se identifican los pasos de la prueba, productos de trabajo (por ejemplo, casos de prueba) y mecanismos para la evaluación de los resultados de la prueba. El proceso de prueba abarca siete tipos diferentes de pruebas.

La prueba de contenido (y las revisiones) se enfocan en varias categorías de contenido. La intención es descubrir errores semánticos y sintácticos que afectan la precisión del contenido o la forma en la que se presenta al usuario final. La prueba de interfaz ejercita los mecanismos de interacción que permiten al usuario comunicarse con la *webapp* y valida los aspectos estéticos de la interfaz. La intención es descubrir errores que resultan a partir de mecanismos de interacción pobremente implantados o de omisiones, inconsistencias o ambigüedades en la semántica de la interfaz.

Las pruebas de navegación aplican casos de uso, derivados de la actividad de modelado, en el diseño de casos de prueba que ejercitan cada escenario de uso contra el diseño de navegación. Los mecanismos de navegación se ponen a prueba para garantizar que cualquier error que impida completar un caso de uso se identifique y corrija. La prueba de componente ejercita las unidades de contenido y funcionales dentro de la *webapp*.

La prueba de configuración intenta descubrir errores y/o problemas de compatibilidad que son específicos de un entorno cliente o servidor particular. Entonces se realizan pruebas para descubrir errores asociados con cada posible configuración. Las pruebas de seguridad incorporan una serie de pruebas diseñadas para explotar las vulnerabilidades en la *webapp* y su entorno. La intención es encontrar huecos de seguridad. La prueba de rendimiento abarca una serie de pruebas que se diseñan para valorar el tiempo de respuesta y la confiabilidad de la *webapp* conforme aumenta la demanda en la capacidad de recursos en el lado servidor.

Problemas y puntos por evaluar

20.1. ¿Existen algunas situaciones en las que la prueba de *webapps* deba descartarse por completo?

20.2. Con sus palabras, analice los objetivos de las pruebas en un contexto *webapp*.

20.3. La compatibilidad es una importante dimensión de la calidad. ¿Qué debe probarse para garantizar que existe compatibilidad para una *webapp*?

20.4. ¿Cuáles errores tienden a ser más serios: los que hay en el lado cliente o los del lado servidor? ¿Por qué?

20.5. ¿Qué elementos de la *webapp* pueden recibir "prueba de unidad"? ¿Qué tipos de pruebas deben realizarse sólo después de que se integran los elementos de la *webapp*?

20.6. ¿Siempre es necesario desarrollar un plan de prueba escrito formalmente? Explique.

20.7. ¿Es justo decir que la estrategia de prueba de *webapps* global comienza con los elementos visibles para el usuario y que avanza hacia los elementos tecnológicos? ¿Existen excepciones a esta estrategia?

20.8. ¿La prueba de contenido *realmente* es una prueba en el sentido convencional? Explique.

20.9. Describa los pasos asociados con la prueba de base de datos para una *webapp*. ¿La prueba de base de datos es predominantemente una actividad en el lado cliente o en el lado servidor?

20.10. ¿Cuál es la diferencia entre las pruebas que se asocian con los mecanismos de interfaz y las que abordan la semántica de la interfaz?

20.11. Suponga que desarrolla una farmacia en línea (**FarmaciaDelaEsquina.com**) que abastece a adultos mayores. La farmacia proporciona las funciones usuales, pero también mantiene una base de datos para cada cliente, de modo que puede ofrecer información de medicamentos y advertir de potenciales interacciones medicamentosas. Analice algunas pruebas de usabilidad especiales para esta *webapp*.

20.12. Suponga que establece una función de comprobación de interacción medicamentosa para **FarmaciaDelaEsquina.com** (problema 20.11). Analice los tipos de las pruebas en el nivel de componente que deberían realizarse para garantizar que esta función opera adecuadamente. [Nota: tendría que usar una base de datos para establecer esta función].

20.13. ¿Cuál es la diferencia entre probar la sintaxis de navegación y probar la semántica de navegación?

20.14. ¿Es posible probar toda configuración que una *webapp* probablemente encuentre en el lado servidor? ¿Es posible hacerlo en el lado cliente? Si no, ¿cómo selecciona un conjunto significativo de pruebas de configuración?

20.15. ¿Cuál es el objetivo de la prueba de seguridad? ¿Quién realiza esta prueba?

20.16. **FarmaciaDelaEsquina.com** (problema 20.11) se volvió muy exitosa y el número de usuarios aumentó dramáticamente en los primeros dos meses de operación. Dibuje una gráfica que muestre el probable tiempo de respuesta como función del número de usuarios para un conjunto fijo de recursos en el lado servidor. Etiquete la gráfica para indicar los puntos de interés en la "curva de respuesta".

20.17. En respuesta a su éxito, **FarmaciaDelaEsquina.com** (problema 20.11) implementó un servidor especial exclusivamente para manejar el reabastecimiento de recetas. En promedio, 1 000 usuarios concurrentes envían una solicitud de reabastecimiento una vez cada dos minutos. En respuesta, la *webapp* descarga un bloque de datos de 500 bytes. ¿Cuál es al rendimiento global aproximado requerido para este servidor en megabits por segundo?

20.18. ¿Cuál es la diferencia entre prueba de carga y prueba de esfuerzo?

Lecturas adicionales y fuentes de información

La literatura acerca de pruebas de *webapps* continúa evolucionando. Los libros de Andrews y Whittaker (*How to Break Web Software,* Addison-Wesley, 2006), Ash (*The Web Testing Companion,* Wiley, 2003), Nguyen *et al*. (*Testing Applications for the Web,* 2a. ed., Wiley, 2003), Dustin *et al*. (*Quality Web Systems,* Addison-Wesley, 2002) y Splaine y Jaskiel [Spl01] están entre los tratamientos más completos del tema publicados hasta la fecha. Mosley (*Client-Server Software Testing on the Desktop and the Web*, Prentice Hall, 1999) aborda los temas de prueba en los lados cliente y servidor.

Información útil acerca de las estrategias y métodos de prueba de *webapps*, así como un valioso análisis de las herramientas de prueba automáticas, se presenta en Stottlemeyer (*Automated Web Testing Toolkit,* Wiley, 2001). Graham *et al*. (*Software Test Automation,* Addison-Wesley, 1999) presentan material adicional acerca de herramientas automáticas.

Microsoft (*Performance Testing Guidance for Web Applications,* Microsoft Press, 2008) y Subraya *(Integrated Approach to Web Performance Testing,* IRM Press, 2006) exponen tratamientos detallados acerca de la prueba de rendimiento para *webapps*. Chirillo (*Hack Attacks Revealed,* 2a. ed., Wiley, 2003), Splaine (*Testing Web Security,* Wiley, 2002), Klevinsky *et al*. (*Hack I.T.: Security through Penetration Testing,* Addison-Wesley, 2002) y Skoudis (*Counter Hack,* Prentice Hall, 2001) proporcionan mucha información útil para quienes deben diseñar pruebas de seguridad. Además, los libros que abordan la prueba de seguridad para software en general pueden ofrecer una guía importante para quienes deben poner a prueba *webapps*. Los títulos representativos incluyen: Basta y Halton (*Computer Security and Penetration Testing,* Thomson Delmar Learning, 2007), Wy-

sopal *et al*. (*The Art of Software Security Testing,* Addison-Wesley, 2006) y Gallagher *et al*. (*Hunting Security Bugs,* Microsoft Press, 2006).

En internet está disponible gran variedad de fuentes de información acerca de pruebas de *webapps*. En el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm** puede encontrarse una lista actualizada de referencias que hay en la World Wide Web y que son relevantes para las pruebas de *webapps*.

CAPÍTULO

21 MODELADO Y VERIFICACIÓN FORMAL

CONCEPTOS CLAVE

certificación 487
diseño de cuarto limpio 483
especificación de estructura de caja 479
especificación funcional 480
lenguaje de especificación Z 495
lenguaje de restricción de objeto (OCL) 492
lenguajes de especificación formal 492
modelo de proceso de cuarto limpio 480
refinamiento de diseño 483
verificación de exactitud . . . 479

A diferencia de las revisiones y pruebas que comienzan una vez desarrollados los modelos y códigos de software, el modelado y la verificación formales incorporan métodos de modelado especializados que se integran con enfoques de verificación prescritos. Sin el enfoque de modelado adecuado, la verificación no puede lograrse.

En este capítulo se analizan dos métodos de modelado y verificación formales: el *método de ingeniería del software de cuarto limpio* y los métodos *formales*. Ambos requieren un enfoque de especificación especializado y cada uno aplica un método de verificación único. Los dos son bastante rigurosos y la comunidad de ingeniería del software no usa ninguno de ellos ampliamente. Pero si intenta construir software a prueba de balas, dichos métodos pueden ayudarle de manera inconmensurable. Vale la pena aprenderlos.

La *ingeniería del software de cuarto limpio* es un enfoque que enfatiza la necesidad de construir con exactitud el software conforme éste se desarrolla. En lugar de análisis, diseño, código, prueba y ciclo de depuración clásicos, el enfoque de cuarto limpio sugiere un punto de vista diferente [Lin94b]:

> La filosofía que hay detrás de la ingeniería del software de cuarto limpio consiste en evitar la dependencia de costosos procesos de remoción de defectos, al escribir incrementos de código justo la primera vez y verificar su exactitud antes de examinarlo. Su modelo de proceso incorpora la certificación de calidad estadística de los incrementos de código conforme se acumulan en un sistema.

UNA MIRADA RÁPIDA

¿Qué es? ¿Cuántas veces ha escuchado a alguien decir "Hazlo bien desde la primera vez"? Si esto se lograra en el software, habría considerablemente menos esfuerzo empleado en lo innecesario. Dos métodos avanzados de ingeniería del software (ingeniería del software de cuarto limpio y métodos formales) ayudan al equipo de software a "hacerlo bien desde la primera vez" al proporcionar un enfoque basado en matemáticas para programar el modelado y la capacidad de verificar que el modelo es correcto. La ingeniería del software de cuarto limpio enfatiza la verificación matemática de la exactitud antes de que comience la construcción del programa y la certificación de la confiabilidad del software como parte de la actividad de prueba. Los métodos formales usan teoría de conjuntos y notación lógica para crear un enunciado claro de los hechos (requerimientos) que pueden analizarse para mejorar (o incluso probar) la exactitud y la consistencia. La línea de base para ambos métodos es la creación de software con tasas de falla extremadamente bajas.

¿Quién lo hace? Un ingeniero del software especialmente capacitado.

¿Por qué es importante? Los errores obligan a que haya revisiones. Las revisiones toman tiempo y aumentan los costos. ¿No sería bueno si pudiera reducir dramáticamente el número de errores (bugs) introducidos mientras el software se diseña y construye? Ésa es la premisa del modelado y de la verificación formales.

¿Cuáles son los pasos? Los modelos de requerimientos y de diseño se crean usando notación especializada que es susceptible de verificación matemática. La ingeniería de software de cuarto limpio usa representación de estructura de cajas que encapsulan el sistema (o algún aspecto del mismo) en un nivel específico de abstracción. La verificación de la exactitud se aplica cuando está completo el diseño de la estructura de caja. Una vez verificada la exactitud para cada estructura de caja, comienza la prueba de uso estadístico. Los métodos formales traducen los requerimientos de software en una representación más formal al aplicar la notación y la heurística de conjuntos a fin de definir el invariante de datos, estados y operaciones para una función de sistema.

¿Cuál es el producto final? Se desarrolla un modelo formal especializado de requerimientos. Se registran los resultados de las pruebas de exactitud y de uso estadístico.

¿Cómo me aseguro de que lo hice bien? La prueba de exactitud formal se aplica al modelo de requerimientos. La prueba de uso estadístico ejercita los escenarios de uso para garantizar que se descubren y corrigen los errores en la funcionalidad del usuario.

En muchas formas, el enfoque de cuarto limpio eleva la ingeniería del software a otro nivel, al enfatizar la necesidad de *probar* la exactitud.

Los modelos desarrollados que usan *métodos formales* se describen mediante una sintaxis y semántica formales que especifican el funcionamiento y el comportamiento del sistema. La especificación consiste en matemática en forma (por ejemplo, puede usarse cálculo de predicados como la base para un lenguaje de especificación formal). En su introducción a los métodos formales, Anthony Hall [Hal90] hace un comentario que se aplica igualmente a los métodos de cuarto limpio:

> Los métodos formales [la ingeniería del software de cuarto limpio] son controversiales. Sus defensores afirman que pueden revolucionar el desarrollo [del software]. Sus detractores creen que son imposiblemente difíciles. Mientras tanto, para la mayoría de la gente, los métodos formales [y la ingeniería del software de cuarto limpio] son tan poco corrientes que es difícil juzgar las afirmaciones rivales.

En este capítulo se exploran los métodos de modelado y verificación formales y se examina su impacto potencial sobre la ingeniería del software en los años por venir.

21.1 Estrategia de cuarto limpio

Cita:

"La única forma de que ocurran errores en un programa es que el autor los ponga ahí. No se conocen otros mecanismos [...] La práctica correcta se dirige a evitar la inserción de errores y, al fallar esto, a removerlos antes de probarlo o de cualquier otra forma de poner en marcha el programa."

Harlan Mills

La ingeniería del software de cuarto limpio usa una versión especializada del modelo de software incremental que se introdujo en el capítulo 2. Pequeños equipos de software independientes desarrollan "una tubería de incrementos de software" [Lin94b]. Conforme cada incremento se certifica, se integra en el todo. Por tanto, la funcionalidad del sistema crece con el tiempo.

La secuencia de las tareas de cuarto limpio para cada incremento se ilustra en la figura 21.1. Dentro de la tubería de incrementos de cuarto limpio, ocurren las siguientes tareas:

Planeación del incremento. Se desarrolla un plan de proyecto que adopte la estrategia incremental. Se crea la funcionalidad de cada incremento, su tamaño proyectado y un calendario de desarrollo de cuarto limpio. Debe tenerse especial cuidado para garantizar que los incrementos certificados se integrarán en forma oportuna.

Recopilación de requerimientos. Se desarrolla una descripción más detallada de los requerimientos del cliente (para cada incremento), con el uso de técnicas similares a las introducidas en el capítulo 5.

Especificación de estructura de caja. Se usa un método de especificación que utilice las *estructuras de caja* para describir la especificación funcional. Las estructuras de caja "se aíslan, y separan la definición creativa de comportamiento, datos y procedimientos en cada nivel de refinamiento" [Hev93].

Diseño formal. Al usar el enfoque de estructura de cajas, el diseño de cuarto limpio es una extensión natural y sin costuras de la especificación. Aunque es posible hacer una distinción clara entre las dos actividades, las especificaciones (llamadas *cajas negras*) se refinan iterativamente (dentro de un incremento) para convertirse en análogas de los diseños arquitectónicos y en el nivel de componente (llamados *cajas de estado* y *cajas claras*, respectivamente).

Verificación de exactitud. El equipo de cuarto limpio realiza una serie de rigurosas actividades de verificación de exactitud sobre el diseño y, luego, el código. La verificación (sección 21.3.2) comienza con la estructura de caja (especificación) de nivel más alto y avanza hacia el detalle y el código de diseño. El primer nivel de la verificación de exactitud ocurre al aplicar un conjunto de "preguntas de exactitud" [Lin88]. Si esto no demuestra que la especificación es correcta, se usan métodos más formales (matemáticos) para la verificación.

WebRef

En **www.cleansoft.com** puede encontrarse una excelente fuente de información y recursos para ingeniería del software de cuarto limpio.

Cita:

"La ingeniería del software de cuarto limpio logra control de calidad estadístico sobre el desarrollo del software al separar estrictamente el proceso de diseño del proceso de prueba en una tubería de desarrollo incremental de software."

Harlan Mills

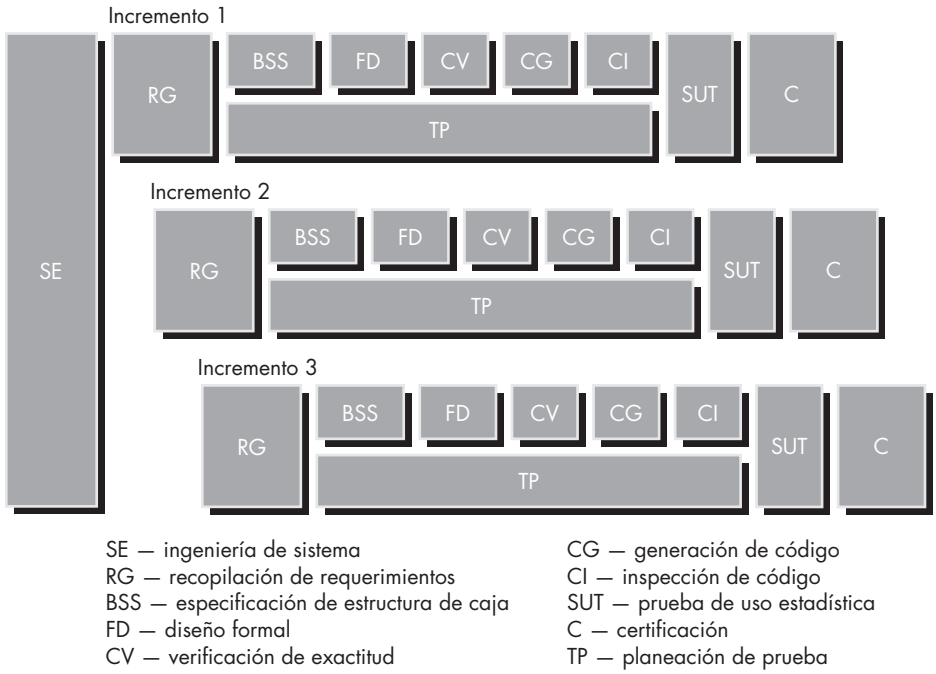


FIGURA 21.1 Modelo de proceso de cuarto limpio

Generación, inspección y verificación de código. Las especificaciones de estructura de caja, representadas en un lenguaje especializado, se traducen al lenguaje de programación adecuado. Las revisiones técnicas (capítulo 15) se usan entonces para asegurar la conformidad semántica del código y las estructuras de código, así como la exactitud sintáctica del código. Luego se realiza la verificación de exactitud para el código fuente.

Planeación de prueba estadística. Se analiza el uso proyectado del software y se planea y diseña (sección 21.4) una suite de casos de prueba que ejercitan una "distribución de probabilidad" de uso. En la figura 21.1, esta actividad de cuarto limpio se realiza en paralelo con la especificación, la verificación y la generación de código.

Prueba de uso estadístico. Si se recuerda que la prueba exhaustiva del software de computadora es imposible (capítulo 18), siempre es necesario diseñar un número finito de casos de prueba. Las técnicas de uso estadístico [Poo88] ejecutan una serie de pruebas derivadas de una muestra estadística (la distribución de probabilidad anotada anteriormente) de todas las posibles ejecuciones de programa efectuadas por todos los usuarios de una población objetivo (sección 21.4).

El cuarto limpio enfatiza las pruebas que ejercitan la manera en la que el software se usa realmente. Los casos de uso proporcionan entrada al proceso de planeación de prueba.

Certificación. Una vez completadas la verificación, inspección y prueba de uso (y todos los errores corregidos), el incremento se certifica como listo para su integración.

Las primeras cuatro actividades en el proceso de cuarto limpio establecen el escenario para las actividades normales de verificación que siguen. Por esta razón, el estudio del enfoque de cuarto limpio comienza con las actividades de modelado, que son esenciales para la verificación formal que se va a aplicar.

21.2 ESPECIFICACIÓN FUNCIONAL

El enfoque de modelado en la ingeniería del software de cuarto limpio usa un método llamado *especificación de estructura de caja*. Una "caja" encapsula el sistema (o algún aspecto del mismo)

Cita:

"Hay algo divertido en la vida: si rechazas aceptar todo menos lo mejor, con mucha frecuencia lo conseguirás."

W. Somerset Maugham

en algún nivel de detalle. A través de un proceso de elaboración o refinamiento por pasos, las cajas se refinan en una jerarquía donde cada caja tiene transparencia referencial. Es decir: "la información contenida en cada especificación de caja es suficiente para definir su refinamiento, sin depender de la implementación de alguna otra caja" [Lin94b]. Esto permite al analista dividir un sistema jerárquicamente y avanzar de la representación esencial en la parte superior al detalle específico de la implementación en el fondo. Para ello, se usan tres tipos de cajas:

¿Cómo se logra el refinamiento como parte de una especificación de estructura de caja?

Caja negra. La caja negra especifica el comportamiento de un sistema o de una parte de un sistema. El sistema (o su parte) responde a estímulos específicos (eventos) al aplicar un conjunto de reglas de transición que mapean el estímulo en una respuesta.

Caja de estado. La caja de estado encapsula los datos y servicios (operaciones) de estado en una forma análoga a los objetos. En esta visión de especificación, se representan las entradas (estímulos) y salidas (respuestas) a la caja de estado. La caja de estado también representa la "historia de estímulos" de la caja negra, es decir, los datos encapsulados en la caja de estado que deben conservarse entre las transiciones implicadas.

Caja clara. Las funciones de transición que se implican mediante la caja de estado se definen en la caja clara. Dicho de manera simple, una caja clara contiene el diseño de procedimientos para la caja de estado.

La figura 21.2 ilustra el enfoque de refinamiento, usando la especificación de estructura de cajas. Una caja negra (BB_1) define las respuestas a un conjunto completo de estímulos. BB_1 puede refinarse en un conjunto de cajas negras, $BB_{1.1}$ a $BB_{1.n}$, cada una de las cuales enfoca una clase de comportamiento. El refinamiento continúa hasta que se identifica una clase cohesiva de comportamiento (por ejemplo, $BB_{1.1.1}$). Entonces se define una caja de estado ($SB_{1.1.1}$) para la caja negra ($BB_{1.1.1}$). En este caso, $SB_{1.1.1}$ contiene todos los datos y servicios requeridos para implementar el comportamiento definido por $BB_{1.1.1}$. Finalmente, $SB_{1.1.1}$ se refina en cajas claras ($CB_{1.1.1.n}$) y se especifican los detalles del diseño de procedimientos.

PUNTO CLAVE

El refinamiento de estructura de cajas y la verificación de exactitud ocurren simultáneamente.

Conforme ocurre cada uno de estos pasos de refinamiento, también se presenta la verificación de la exactitud. Las especificaciones de caja de estado se verifican para asegurar que cada una se conforma de acuerdo con el comportamiento definido por la especificación de la caja negra padre. De igual modo, las especificaciones de caja clara se verifican contra la caja de estado padre.

FIGURA 21.2

Refinamiento de estructura de cajas

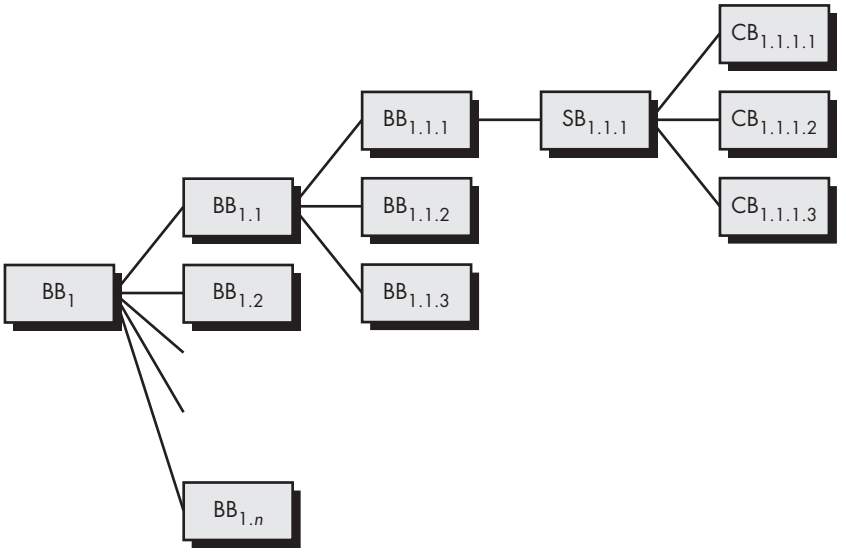
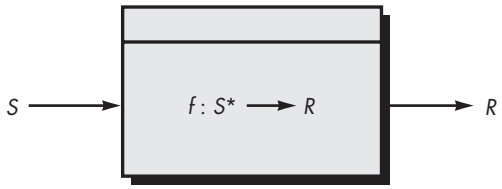


FIGURA 21.3
Especificación de caja negra



21.2.1 Especificación de caja negra

Una especificación de *caja negra* describe una abstracción, estímulos y respuesta, usando la notación que se muestra en la figura 21.3 [Mil88]. La función f se aplica a una secuencia S^* de entradas (estímulos) S y las transforma en una salida (respuesta) R. Para componentes de software simples, f puede ser una función matemática, pero, en general, f se describe usando lenguaje natural (o un lenguaje de especificación formal).

Muchos de los conceptos introducidos para los sistemas orientados a objetos también son aplicables para la caja negra. Las abstracciones de datos y las operaciones que manipulan dichas abstracciones se encapsulan mediante la caja negra. Como una jerarquía de clases, la especificación de caja negra puede mostrar jerarquías de uso en las que las cajas de nivel inferior heredan las propiedades de las cajas superiores que hay en la estructura del árbol.

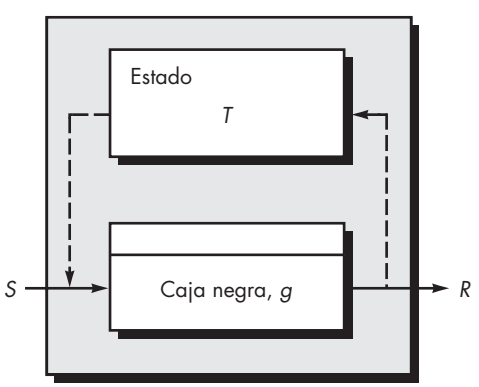
21.2.2 Especificación de caja de estado

La *caja de estado* es "una simple generalización de una máquina de estado" [Mil88]. Si se recuerda el análisis acerca del modelado de comportamiento y de los diagramas de estado estudiados en el capítulo 7, un estado es un modo observable de comportamiento del sistema. Conforme ocurre el procesamiento, un sistema responde a los eventos (estímulos), haciendo una transición desde el estado actual hasta algún estado nuevo. Conforme se realiza la transición, puede ocurrir una acción. La caja de estado usa una abstracción de datos para determinar la transición al siguiente estado y la acción (respuesta) que ocurrirá como consecuencia de la transición.

En la figura 21.4, la caja de estado incorpora una caja negra g. El estímulo S que es entrada a la caja negra llega desde alguna fuente externa y desde un conjunto de estados internos del sistema T. Mills [Mil88] proporciona una descripción matemática de la función f de la caja negra contenida dentro de la caja de estado:

$$g : S^* \times T^* \rightarrow R \times T$$

FIGURA 21.4
Una especificación de caja de estado



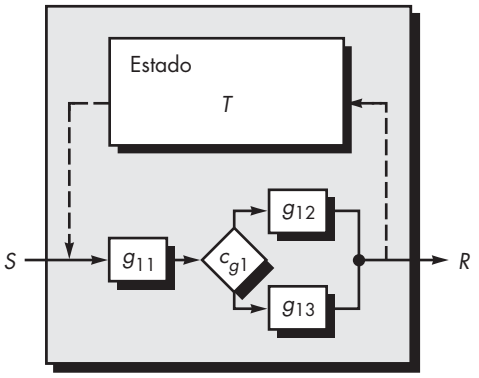


FIGURA 21.5
Una especificación de caja clara

donde *g* es una subfunción que se liga a un estado específico *t*. Cuando se consideran de manera colectiva, los pares de subfunción de estado (*t, g*) definen la función *f* de la caja negra.

21.2.3 Especificación de caja clara

La especificación de caja clara está cercanamente alineada con el diseño procedural y con la programación estructurada. En esencia, la subfunción *g* dentro de la caja de estado se sustituye con los constructos de programación estructurada que implementan *g*.

Como ejemplo, considere la caja clara que se muestra en la figura 21.5. La caja negra *g*, que se muestra en la figura 21.3, se sustituye por un constructo secuencia que incorpora un condicional. Éste, a su vez, puede refinarse en las cajas claras de nivel inferior conforme avanza el refinamiento por pasos.

Es importante observar que puede probarse que la especificación descrita en la jerarquía de caja clara es correcta. Este tema se considera en la sección 21.3.

21.3 DISEÑO DE CUARTO LIMPIO

La ingeniería del software de cuarto limpio utiliza mucho la filosofía de programación estructurada (capítulo 10). Pero, en este caso, la programación estructurada se aplica de manera mucho más rigurosa.

Las funciones de procesamiento básico (descritas durante los primeros refinamientos de la especificación) se refinan usando una "expansión en pasos de funciones matemáticas en estructuras de conectivos lógicos [por ejemplo, if-then-else] y subfunciones, donde la expansión [se] realiza hasta que todas las subfunciones identificadas puedan enunciarse directamente en el lenguaje de programación utilizado para la implementación" [Dye92].

El enfoque de programación estructurada puede usarse de manera efectiva para refinar funciones. Pero ¿qué hay del diseño? Aquí entran en juego algunos conceptos de diseño fundamentales (capítulo 8). Los datos de programa se encapsulan como un conjunto de abstracciones que son atendidas por subfunciones. Los conceptos de encapsulamiento de datos, ocultamiento de información y escritura de datos se usan para crear el diseño de datos.

21.3.1 Refinamiento de diseño

Cada especificación de caja clara representa el diseño de un procedimiento (subfunción) requerido para lograr una transición de caja de estado. Dentro de la caja clara se usan constructos de programación estructurada y refinamiento por pasos para representar detalles procedurales. Por ejemplo, una función de programa *f* se refina en una secuencia de subfunciones *g* y *h*. Éstas a

su vez se refinan en constructos condicionales (por ejemplo, if-then-else y do-while). Un mayor refinamiento continúa hasta que hay suficiente detalle procedural para crear el componente en cuestión.

En cada nivel de refinamiento, el equipo de cuarto limpio[1] realiza una *verificación formal de exactitud*. Para lograr esto, a los constructos de programación estructurada se une un conjunto de condiciones de exactitud genéricas. Si una función *f* se expande en una secuencia *g* y *h*, la condición de exactitud para toda entrada a *f* es

¿Qué condiciones se aplican para probar la exactitud de los constructos estructurados?

- ¿*g* seguida de *h* hace *f*?

Cuando una función *p* se refina en una condicional de la forma "if <*c*> then *q*, else *r*", la condición de exactitud para toda entrada a *p* es

- Siempre que la condición <*c*> es verdadera, ¿*q* hace *p*?; y siempre que <*c*> es falsa, ¿*r* hace *p*?

Cuando la función *m* se refina como un ciclo, las condiciones de exactitud para toda entrada a *m* son

- ¿Está garantizada la finalización?
- Siempre que <*c*> es verdadera, ¿*n* seguida por *m* hace *m*?; y siempre que <*c*> es falsa, ¿saltar el ciclo todavía hace *m*?

Cada vez que una caja clara se refina al siguiente nivel de detalle, se aplican dichas condiciones de exactitud.

21.3.2 Verificación de diseño

CONSEJO

Si se limita sólo a constructos estructurados mientras desarrolla un diseño procedural, la prueba de exactitud es directa. Si viola los constructos, las pruebas de exactitud son difíciles o imposibles.

Debe observar que el uso de los constructos de programación estructurada restringen el número de pruebas de exactitud que deben realizarse. Una sola condición se verifica para secuencias; dos condiciones se prueban para if-then-elseo y tres condiciones se verifican para ciclo.

Para ilustrar la verificación de exactitud para un diseño procedural, use un ejemplo simple, introducido por primera vez por Linger, Mills y Witt [Lin79]. La intención es diseñar y verificar un pequeño programa que encuentre la parte entera *y* de una raíz cuadrada de un entero dado *x*. El diseño procedural se representa usando el diagrama de flujo de la figura 21.6.[2]

Para verificar la exactitud de este diseño, las condiciones de entrada y salida se agregan como se muestra en la figura 21.6. La condición de entrada observa que *x* debe ser mayor que o igual a 0. La condición de salida requiere que *x* permanezca invariable y que *y* satisfaga la expresión anotada en la figura. Para probar que el diseño es correcto, es necesario probar que las condiciones *init*, *loop*, *cont*, *yes* y *exit*, que se muestran en la figura 21.6, son verdaderas en todos los casos. En ocasiones a esto se le conoce como *subpruebas*.

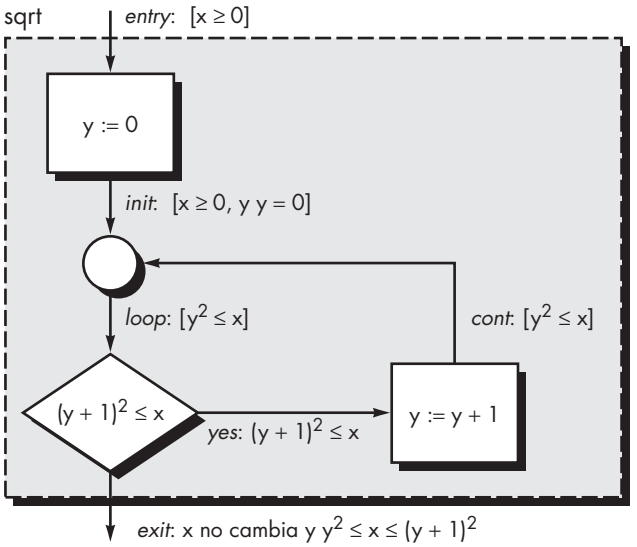
1. La condición *init* demanda que [$x \geq 0$ y $y = 0$]. Con base en los requerimientos del problema, la condición de entrada se supone correcta.[3] Por tanto, se satisface la primera parte de la condición *init*, $x \geq 0$. En el diagrama de flujo, el enunciado inmediatamente anterior a la condición *init* establece $y = 0$. Por tanto, la segunda parte de la condición *init* también se satisface. En consecuencia, *init* es verdadera.
2. La condición *loop* puede encontrarse en una de dos formas: 1) directamente de *init* (en este caso, la condición *loop* se satisface directamente) o por medio del flujo de control

1 Puesto que todo el equipo está involucrado en el proceso de verificación, es menos probable que se cometa un error al realizar la verificación en sí.

2 La figura 21.6 se adaptó de [Lin94]. Utilizada con permiso.

3 En este contexto no tiene validez un valor negativo para la raíz cuadrada.

FIGURA 21.6
Cálculo de la parte entera de una raíz cuadrada
Fuente: [Lin79].



PUNTO CLAVE

Para probar que un diseño es correcto, primero deben identificarse todas las condiciones y luego probar que cada una toma el valor booleano adecuado. A éstas se les llama *subpruebas*.

que pasa a través de la condición *cont*. Dado que la condición *cont* es idéntica a la condición *loop*, ésta es verdadera sin importar la trayectoria de flujo que conduce a ella.

3. La condición *cont* se encuentra solamente después de que el valor de *y* aumenta en 1. Además, la ruta de flujo de control que conduce a *cont* puede invocarse sólo si la condición *yes* también es verdadera. Por tanto, si $(y + 1)^2 \leq x$ se sigue que $y^2 \leq x$. Se satisface la condición *cont*.
4. La condición *yes* se prueba en la lógica condicional que se muestra. Por ende, la condición *yes* debe ser verdadera cuando el flujo de control se mueve a lo largo de la trayectoria mostrada.
5. La condición *exit* demanda primero que *x* permanezca invariable. Un examen del diseño indica que *x* no aparece a la izquierda de un operador de asignación. No hay llamadas de función que usen *x*. En consecuencia, es invariable. Puesto que la prueba condicional $(y + 1)^2 \leq x$ debe fallar para alcanzar la condición *exit*, se sigue que $(y + 1)^2 \leq x$. Además, la condición *loop* debe incluso ser verdadera (es decir, $y^2 \leq x$). Por tanto, $(y + 1)^2 > x$ y $y^2 \leq x$ pueden combinarse para satisfacer la condición *exit*.

Adicionalmente, debe asegurarse de que el ciclo termina. Un examen de la condición *loop* indica que, dado que *y* se incrementa y $x \geq 0$, el ciclo finalmente debe terminar.

Los cinco pasos recién señalados son una prueba de la exactitud del diseño del algoritmo anotado en la figura 21.6. Entonces se está seguro de que el diseño, de hecho, calculará la parte entera de una raíz cuadrada.

Es posible un enfoque matemático más riguroso para verificar el diseño. Sin embargo, un estudio de este tema está más allá del ámbito de este libro. Si tiene interés en ello, consulte [Lin79].

21.4 PRUEBAS DE CUARTO LIMPIO

La estrategia y tácticas de las pruebas de cuarto limpio son fundamentalmente diferentes a las de los enfoques de prueba convencionales (capítulos del 17 al 20). Los métodos convencionales derivan un conjunto de casos de prueba para descubrir errores de diseño y codificación. La meta

Cita:

"La calidad no es un acto, es un hábito."

Aristóteles

de la prueba de cuarto limpio es validar los requerimientos de software al demostrar que una muestra estadística de casos de uso (capítulo 5) se ejecuta exitosamente.

21.4.1 Pruebas de uso estadístico

El usuario de un programa de computadora rara vez necesita entender los detalles técnicos del diseño. El comportamiento del programa visible al usuario se activa con entradas y eventos que con frecuencia son producidos por el usuario. Pero en los sistemas complejos, el posible espectro de entrada y eventos (es decir, los casos de uso) puede ser extremadamente amplio. ¿Qué subconjunto de casos de uso verificará de manera adecuada el comportamiento del programa? Ésta es la primera pregunta que enfoca las pruebas de uso estadístico.

CONSEJO

Incluso si se decide no usar el enfoque de cuarto limpio, vale la pena considerar las pruebas de uso estadístico como parte integral de la estrategia de pruebas.

La prueba de uso estadístico "equivale a examinar el software de la forma en la que los usuarios pretenden usarlo" [Lin94b]. Para lograr esto, los equipos de prueba de cuarto limpio (también llamados *equipos de certificación*) deben determinar una distribución de probabilidad de uso para el software. La especificación (caja negra) para cada incremento del software se analiza a fin de definir un conjunto de estímulos (entradas o eventos) que hacen que el software cambie su comportamiento. En la creación de escenarios de uso y en una comprensión general del dominio de aplicación, a cada estímulo se le asigna una probabilidad de uso con base en entrevistas con usuarios potenciales.

Para cada conjunto de estímulos[4] se generan casos de prueba de acuerdo con la distribución de probabilidad de uso. Para ilustrar lo anterior, considere el sistema *CasaSegura* que se estudió anteriormente en este libro. La ingeniería del software de cuarto limpio se usó para desarrollar un incremento de software que gestiona la interacción del usuario con el teclado del sistema de seguridad. Para este incremento se identificaron cinco estímulos. Los análisis indican la distribución de probabilidad porcentual de cada estímulo. Para seleccionar con facilidad los casos de prueba, dichas probabilidades se mapean en intervalos numerados entre 1 y 99 [Lin94] y se ilustran en la siguiente tabla:

Estímulo del programa	Probabilidad	Intervalo
Armar/desarmar (AD)	50%	1–49
Establecer zona (EZ)	15%	50–63
Consulta (C)	15%	64–78
Prueba (P)	15%	79–94
Alarma de pánico	5%	95–99

Para generar una secuencia de casos de prueba de uso que se ajusten a la distribución de probabilidad de uso, se generan números aleatorios entre 1 y 99. Cada número aleatorio corresponde a un intervalo en la distribución de probabilidad precedente. Por tanto, la secuencia de casos de prueba de uso se define al azar, pero corresponde a la probabilidad adecuada de ocurrencia del estímulo. Por ejemplo, suponga que se generan las siguientes secuencias de números aleatorios:

13-94-22-24-45-56
81-19-31-69-45-9
38-21-52-84-86-4

Al seleccionar los estímulos adecuados con base en el intervalo de distribución que se muestra en la tabla, se derivan los siguientes casos de uso:

4 Puede usar herramientas automatizadas para lograr esto. Para mayor información, consulte [Dye92].

AD–T–AD–AD–AD–ZS
T–AD–AD–AD–Q–AD–AD
AD–AD–ZS–T–T–AD

El equipo de prueba los ejecuta, y verifica el comportamiento del software contrastándolo con la especificación para el sistema. La temporización de las pruebas se registra de modo que puedan determinarse los intervalos de tiempo. Al usar intervalos de tiempo, el equipo de certificación puede calcular el tiempo medio hasta el fallo (TMHF). Si una larga secuencia de pruebas se realiza sin fallas, el TMHF es bajo y la confiabilidad del software puede suponerse alta.

21.4.2 Certificación

Las técnicas de verificación y prueba analizadas anteriormente en este capítulo conducen a componentes de software (e incrementos completos) que pueden certificarse. Dentro del contexto del enfoque de ingeniería del software de cuarto limpio, la *certificación* implica que la confiabilidad (medida por el TMHF) puede especificarse para cada componente.

El impacto potencial de los componentes de software certificables va más allá de un solo proyecto de cuarto limpio. Los componentes de software reutilizables pueden almacenarse junto con sus escenarios de uso, estímulos de programa y distribuciones de probabilidad. Cada componente tendría una confiabilidad certificada bajo el escenario de uso y el régimen de pruebas descritos. Esta información es invaluable para otros que quieran usar los componentes.

? ¿Cómo se certifica un componente de software?

El enfoque de certificación involucra cinco pasos [Woh94]: 1) se crean escenarios de uso, 2) se especifica un perfil de uso, 3) se generan casos de prueba a partir del perfil, 4) las pruebas se ejecutan y los datos de fallo se registran y analizan, y 5) se calcula la confiabilidad y se certifica. Los pasos del 1 al 4 se analizaron en una sección anterior. La certificación para la ingeniería del software de cuarto limpio requiere la creación de tres modelos [Poo93]:

Modelo de muestreo. La prueba de software ejecuta *m* casos de prueba aleatorios y se certifica si no ocurren fallos o un número específico de ellos. El valor de *m* se deriva matemáticamente para asegurar que se logra la confiabilidad requerida.

Modelo de componente. Se certifica un sistema compuesto de *n* componentes. El modelo de componentes permite al analista determinar la probabilidad de que el componente *i* fallará antes de su conclusión.

Modelo de certificación. La confiabilidad global del sistema se proyecta y se certifica.

Al completar las pruebas de uso estadístico, el equipo de certificación tiene la información requerida para entregar el software que tenga un TMHF certificado, usando cada uno de estos modelos. Si tiene más interés, vea [Cur86], [Mus87] o [Poo93], para detalles adicionales.

21.5 Conceptos de métodos formales

The Encyclopedia of Software Engineering [Mar01] define los métodos formales en la forma siguiente:

> Los métodos formales utilizados para desarrollar sistemas de cómputo son técnicas con base matemática para describir las propiedades del sistema. Tales métodos formales proporcionan marcos conceptuales dentro de los cuales las personas pueden especificar, desarrollar y verificar los sistemas en forma sistemática más que *ad hoc*.

Las propiedades deseadas de una especificación formal (consistencia, completitud y falta de ambigüedad) son los objetivos de todos los métodos de especificación. Sin embargo, el lenguaje de especificación con base matemática que se utiliza para los métodos formales da como resultado una probabilidad mucho mayor de lograr dichas propiedades. La sintaxis formal de un

Cita:

"Los métodos formales tienen un tremendo potencial para mejorar la claridad y precisión de las especificaciones de los requerimientos, y para encontrar errores importantes y sutiles."

Steve Easterbrook *et al.*

lenguaje de especificación (sección 21.7) permite que los requerimientos o el diseño se interpreten sólo en una forma, lo que elimina la ambigüedad que con frecuencia ocurre cuando un lector debe interpretar un lenguaje natural (por ejemplo, inglés) o una notación gráfica (por ejemplo, UML). Las facilidades descriptivas de la teoría de conjuntos y la notación lógica permiten un enunciado claro de los requerimientos. Para ser consistente, los requerimientos enunciados en un lugar dentro de una especificación no deben contradecirse en otro lugar. La consistencia se logra[5] al probar matemáticamente que los hechos iniciales pueden mapearse formalmente (usando reglas de inferencia) en los enunciados ulteriores dentro de la especificación.

Para presentar los conceptos de los métodos formales básicos, considere algunos ejemplos simples a fin de ilustrar el uso de la especificación matemática, sin empantanarse en demasiados detalles matemáticos.

Ejemplo 1: una tabla simbólica. Un programa se usa para mantener una tabla simbólica. Dicha tabla se utiliza frecuentemente en muchos tipos diferentes de aplicaciones. Consiste de una colección de ítems sin duplicación alguna. En la figura 21.7 se muestra un ejemplo de una tabla simbólica típica. En ella se representa la tabla que utiliza un sistema operativo para contener los nombres de los usuarios del sistema. Otros ejemplos de tablas incluyen la colección de nombres del personal en un sistema de nómina, la colección de nombres de las computadoras en un sistema de comunicaciones de red y la colección de los destinos en un sistema para producir horarios de transportes.

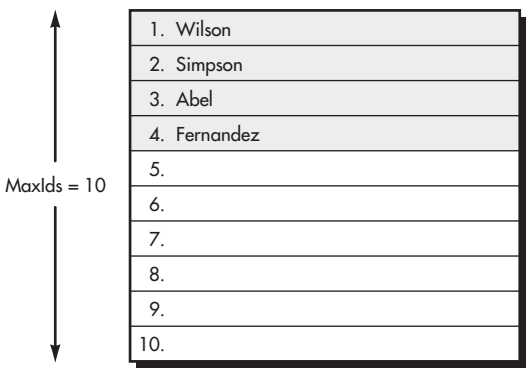
PUNTO CLAVE

Una invariante de datos es un conjunto de condiciones que son verdaderas a lo largo de la ejecución del sistema que contiene una colección de datos.

Suponga que la tabla que se presenta en este ejemplo contiene no más de *MaxIds* nombres. Esta afirmación, que coloca una restricción sobre la tabla, es un componente de una condición conocida como *invariante de datos*: una condición que es verdadera a lo largo de la ejecución de un sistema que contiene una colección de datos. La invariante de datos que se sostiene para la tabla simbólica recién analizada tiene dos componentes: 1) que la tabla contendrá no más de *MaxIds* nombres y 2) que no habrá nombres duplicados en la tabla. En el caso del programa de tabla simbólica, esto significa que en cualquier momento en el que se examine la tabla simbólica durante la ejecución del sistema, siempre contendrá no más de *MaxIds* nombres y no contendrá duplicados.

Otro concepto importante es el de *estado*. Muchos lenguajes formales, como el OCL (sección 21.7.1), usan la noción de estado que se estudió en el capítulo 7, es decir, un sistema puede

FIGURA 21.7
Una tabla simbólica



5 En realidad, la completitud es difícil de garantizar, aun cuando se usen métodos formales. Algunos aspectos de un sistema pueden quedar indefinidos conforme se cree la especificación; otras características pueden omitirse a propósito a fin de permitir a los diseñadores cierta libertad para escoger un enfoque de implementación; y, finalmente, es imposible considerar todo escenario operativo en un sistema grande y complejo. Las cosas pueden omitirse simplemente por equivocación.

Otra forma de apreciar la noción de estado es señalar que los datos determinan el estado. Es decir, puede examinar los datos para ver en qué estado se encuentra el sistema.

estar en uno de muchos estados y cada uno representa un modo de comportamiento observable de manera externa. Sin embargo, una definición diferente para el término *estado* se usa en el lenguaje Z (sección 21.7.2). En Z (y en lenguajes relacionados), el estado de un sistema se representa por los datos almacenados del sistema (por ende, Z sugiere un número mucho mayor de estados, lo que representa cada posible configuración de los datos). Al usar la última definición en el ejemplo del programa de tabla simbólica, el estado es la tabla simbólica.

El concepto final es el de *operación*. Ésta es una acción que tiene lugar dentro de un sistema y que lee o escribe datos. Si el programa de tabla simbólica tiene que ver con agregar y remover nombres de la tabla simbólica, entonces se asociará con dos operaciones: una operación para *add()* (agregar) un nombre específico a la tabla simbólica y otra para *remove()* (remover) un nombre existente de la tabla.[6] Si el programa proporciona la facilidad para comprobar si un nombre específico está contenido en la tabla, entonces habría una operación que regresaría alguna indicación acerca de si el nombre está en la tabla.

Pueden asociarse tres tipos de condiciones con las operaciones: invariantes, precondiciones y poscondiciones. Una *invariante* define lo que se garantiza que no cambia. Por ejemplo, la tabla simbólica tiene una invariante que afirma que el número de elementos siempre es menor que o igual a *MaxIds*. Una *precondición* define las circunstancias en las cuales es válida una operación particular. Por ejemplo, la precondición para una operación que agrega un nombre a una tabla simbólica de identificadores de personal es válida sólo si el nombre que se agrega no está contenido en la tabla y también si hay menos de *MaxIds* identificadores de personal en ella. La *poscondición* de una operación define lo que se garantiza que es verdadero hasta completar una operación. Esto se define por su efecto sobre los datos. Para la operación *add( )*, la poscondición especificaría matemáticamente que la tabla aumentó con el nuevo identificador.

Ejemplo 2: un manipulador de bloques. Una de las partes más importantes de un sistema operativo simple es el subsistema que mantiene los archivos creados por los usuarios. Parte del subsistema de llenado es el *manipulador de bloques*. El almacén de archivos está compuesto de bloques de almacenamiento que se mantienen en un dispositivo de almacenamiento de archivos. Durante la operación de la computadora, se crearán y borrarán archivos, lo que requiere adquisición y liberación de bloques de almacenamiento. Para poder lidiar con esto, el subsistema de llenado mantendrá un reservorio de bloques no utilizados (libres) y seguirá la pista de los bloques que estén en uso actual. Cuando los bloques se liberan de un archivo borrado, por lo general se agregan a una fila de bloques que esperan para incorporarse al reservorio de bloques no utilizados. Esto se muestra en la figura 21.8, donde se presentan algunos componentes: el reservorio de bloques no utilizados, los bloques que en la actualidad constituyen los archivos administrados por el sistema operativo y los bloques que esperan agregarse al reservorio. Los bloques que esperan se mantienen en una fila en la que cada elemento contiene un conjunto de bloques de un archivo borrado.

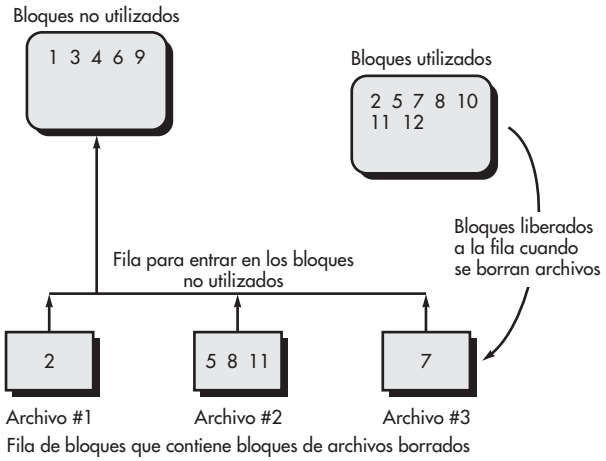
Las técnicas de lluvia de ideas pueden funcionar bien cuando debe desarrollar una invariante de datos para una función razonablemente compleja. Haga que los miembros del equipo de software escriban enlaces, restricciones y limitaciones para la función; luego, combínelas y edítelas.

Para este subsistema, el estado es la colección de bloques libres, la colección de bloques usados y la fila de bloques regresados. La invariante de datos, que se expresa en lenguaje natural, es

- Ningún bloque se marcará como no utilizado y usado al mismo tiempo.
- Todos los conjuntos de bloques que se conservan en la fila serán subconjuntos de la colección de los bloques actualmente utilizados.
- Ningún elemento de la fila contendrá el mismo número de bloque.
- La colección de bloques utilizados y bloques que no se usan será la colección total de bloques que constituyen los archivos.

6 Debe observarse que agregar un nombre no puede ocurrir en el estado *full* (lleno) y borrar un nombre es imposible en el estado *empty* (vacío).

FIGURA 21.8
Manipulador de bloques



- La colección de bloques no utilizados no tendrá números de bloque duplicados.
- La colección de bloques utilizados no tendrá números de bloque duplicados.

Algunas de las operaciones asociadas con estos datos son: *add()* una colección de bloques al final de la fila, *remove()* una colección de bloques usados del frente de la fila y colocarlos en la colección de bloques no utilizados, y *check()* (comprobar) si la fila de bloques está vacía.

La precondición de *add()* es que los bloques que se van a agregar deben estar en la colección de bloques usados. La poscondición es que la colección de bloques ahora se encuentra al final de la fila. La precondición de *remove()* es que la fila debe tener al menos un ítem. La poscondición es que los bloques deben agregarse a la colección de bloques no utilizados. La operación *check()* no tiene precondición. Esto significa que la operación siempre está definida, sin importar qué valor tenga el estado. La poscondición entrega el valor *true* (verdadero) si la fila está vacía y *false* (falso) de otro modo.

En los ejemplos anotados en esta sección, se introducen los conceptos clave de especificación formal, pero sin enfatizar las matemáticas que se requieren para hacer formal la especificación. En la sección 21.6, se considera cómo puede usarse la notación matemática para especificar de manera formal algún elemento del sistema.

21.6 APLICACIÓN DE NOTACIÓN MATEMÁTICA[7] PARA ESPECIFICACIÓN FORMAL

Para ilustrar el uso de la notación matemática en la especificación formal de un componente de software, revise el ejemplo del manipulador de bloques que se presentó en la sección 21.5. A fin de revisarlos, un importante componente del sistema operativo de una computadora mantiene los archivos que crearon los usuarios. El manipulador de bloques mantiene un reservorio de bloques sin utilizar y también seguirá la pista a los bloques que estén en uso en el momento. Cuando los bloques se liberan de un archivo borrado, por lo general se agregan a una fila de bloques que esperan para agregarse al reservorio de bloques no utilizados. Esto se muestra de manera esquemática en la figura 21.8.

7 Esta sección se escribió suponiendo que el lector está familiarizado con la notación matemática asociada con conjuntos y secuencias, y con la notación lógica que se usa en el cálculo de predicados. Si necesita un repaso, en el sitio web de la 7a. edición de este libro se presenta una breve revisión como recurso complementario. Para información más detallada, vea [Jec06] o [Pot04].

¿Cómo pueden representarse estados e invariantes de datos usando un conjunto y operadores lógicos?

Un conjunto llamado *BLOCKS* (bloques) consistirá de todo número de bloques. *AllBlocks* (todos los bloques) es un conjunto de bloques que se encuentra entre 1 y *MaxBlocks* (máximo de bloques). El estado se modelará mediante dos conjuntos y una secuencia. Los dos conjuntos son *used* (utilizado) y *free* (libre). Ambos contienen bloques: el conjunto *used* contiene los que actualmente se usan en los archivos y el conjunto *free* los que están disponibles para nuevos archivos. La secuencia contendrá conjuntos de bloques que están listos para ser liberados de los archivos que se borraron. El estado puede describirse como

used, free: $\mathbb{P}$ *BLOCKS*
BlockQueue: *seq* $\mathbb{P}$ *BLOCKS*

Esto es muy parecido a la declaración de las variables de programa. Se afirma que *used* y *free* serán conjuntos de bloques y que *BlockQueu* (fila de bloques) será una secuencia, cada elemento de la cual será un conjunto de bloques. La invariante de datos puede escribirse como

$$used \cap free = \varnothing \land$$
$$used \cup free = AllBlocks \land$$
$$\forall\, i: \text{dom } BlockQueue \bullet BlockQueue\ i \subseteq used \land$$
$$\forall\, i, j: \text{dom } BlockQueue \bullet i \neq j = BlockQueue\ i \cap BlockQueue\ j = \varnothing$$

WebRef
Amplia información de los métodos formales se puede encontrar en **www.afm.sbu .ac.uk**.

Los componentes matemáticos de la invariante de datos coinciden con cuatro de las características de los componentes de lenguaje natural descritos anteriormente. La primera línea de la invariante de datos afirma que no habrá bloques comunes en la colección utilizada y en las colecciones libres de bloques. La segunda afirma que la colección de bloques utilizados y de bloques libres siempre será igual a la colección completa de bloques en el sistema. La tercera línea indica que el *i*-ésimo elemento en la fila de bloques siempre será un subconjunto de los bloques utilizados. La línea final afirma que, para cualesquiera dos elementos de la fila de bloques que no son el mismo, no habrá bloques comunes en dichos elementos. Los dos componentes de lenguaje natural finales de la invariante de datos se implementan en virtud del hecho de que *used* y *free* son conjuntos y, por tanto, no contendrán duplicados.

La primera operación por definir es aquella que remueve un elemento de la cabeza de la fila de bloques. La precondición es que debe haber al menos un ítem en la fila:

$$\#\text{BlockQueue} > 0,$$

La poscondición es que la cabeza de la fila debe removerse y colocarse en la colección de bloques libres y la fila debe ajustarse para mostrar la remoción:

$$used' = used \setminus head\ BlockQueue \land$$
$$free' = free \cup head\ BlockQueue \land$$
$$BlockQueue' = tail\ BlockQueue$$

Una convención que se usa en muchos métodos formales es que el valor de una variable después de una operación es prima. En consecuencia, el primer componente de la expresión precedente afirma que los nuevos bloques usados (*used'*) serán iguales a los antiguos bloques usados menos los bloques que se removieron. El segundo componente señala que los nuevos bloques libres (*free'*) serán los antiguos bloques libres, con el agregado de la cabeza de la fila de bloques. El tercer componente afirma que la nueva fila de bloques será igual a la fila del valor antiguo de la fila de bloques, es decir, todos los elementos en la fila menos el primero. Una segunda operación agrega una colección de bloques, *Ablocks*, a la fila de bloques. La precondición es que *Ablocks* es en la actualidad un conjunto de bloques utilizados:

¿Cómo se representan las precondiciones y las poscondiciones?

$$Ablocks \subseteq used$$

La poscondición es que el conjunto de bloques se agrega al final de la fila de bloques y el conjunto de bloques usados y libres permanece invariable:

$$BlockQueue' = BlockQueue \frown \langle Ablocks \rangle \wedge$$
$$used' = used \wedge$$
$$free' = free$$

No hay duda de que la especificación matemática de la fila de bloques es considerablemente más rigurosa que una narrativa de lenguaje natural o que un modelo gráfico. El rigor adicional requiere esfuerzo, pero los beneficios obtenidos de la consistencia mejorada y de la completitud pueden justificarse para algunos dominios de aplicación.

21.7 LENGUAJES DE ESPECIFICACIÓN FORMAL

Un lenguaje de especificación formal por lo general se compone de tres componentes primarios: 1) una sintaxis que define la notación específica con la que se representa la especificación, 2) semántica para ayudar a definir un "universo de objetos" [Win90] que se usarán para describir el sistema y 3) un conjunto de relaciones que definen las reglas que indican cuáles objetos satisfacen adecuadamente la especificación.

El dominio sintáctico de un lenguaje de especificación formal con frecuencia se basa en una sintaxis que se deriva de la notación estándar de la teoría de conjuntos y del cálculo de predicados. El *dominio semántico* de un lenguaje de especificación indica cómo representa el lenguaje los requerimientos del sistema.

Es posible usar diferentes abstracciones semánticas para describir el mismo sistema en formas distintas. En los capítulos 6 y 7, se hizo esto de manera menos formal. Se representaron información, función y comportamiento. Para representar el mismo sistema, puede usarse una notación de modelado diferente. La semántica de cada representación proporciona visiones complementarias del sistema. Para ilustrar este enfoque cuando se usan métodos formales, suponga que utiliza un lenguaje de especificación formal para describir el conjunto de eventos que hacen que ocurra un estado particular en un sistema. Otra relación formal muestra todas las funciones que ocurren dentro de un estado determinado. La intersección de estas dos relaciones proporciona un indicio de los eventos que producirán funciones específicas.

En la actualidad se usan varios lenguajes de especificación formales. OCL [OMG03b], Z [ISO02], LARCH [Gut93] y VDM [Jon91] son lenguajes de especificación formal representativos que muestran las características anotadas anteriormente. En este capítulo se presenta un breve estudio de OCL y Z.

21.7.1 Lenguaje de restricción de objeto (OCL)[8]

El *lenguaje de restricción de objeto* (OCL) es una notación formal desarrollada de modo que los usuarios de UML puedan agregar más precisión a sus especificaciones. En el lenguaje está disponible todo el poder de la lógica y de la matemática discreta. Sin embargo, los diseñadores de OCL decidieron que, en los enunciados OCL, sólo deberían usarse caracteres ASCII (en lugar de la notación matemática tradicional). Esto hace que el lenguaje sea más amistoso para las personas que tienen menos inclinación matemática y que la computadora lo procese más fácilmente. Pero también hace al OCL un poco farragoso en algunos lugares.

Para usar OCL, comience con uno o más diagramas UML: los diagramas de clase, estado o actividad más comunes (apéndice 1). Se agregan expresiones OCL y hechos de estado acerca de elementos de los diagramas. Dichas expresiones se llaman *restricciones*; cualquier implementación derivada del modelo debe asegurar que cada una de las restricciones siempre sigue siendo verdadera.

8 Esta sección es aportación del profesor Timothy Lethbridge, de la Universidad de Ottawa, y se presenta aquí con permiso.

TABLA 21.1 RESUMEN DE NOTACIÓN OCL CLAVE

x.y	Obtiene la propiedad y del objeto x. Una propiedad puede ser un atributo, el conjunto de objetos al final de una asociación, el resultado de evaluar una operación y otras cosas, dependiendo del tipo de diagrama UML. Si x es un Conjunto, entonces y se aplica a cada elemento de x; los resultados se recopilan en un nuevo Conjunto.
c−>f()	Aplica la operación f interna de OCL a la Colección c en sí (en oposición a cada uno de los objetos en c). A continuación se mencionan ejemplos de operaciones internas.
and, or, =, <>	And lógica, or lógica, igual, no es igual.
p implica q	Verdadero si q es verdadero o p es falso.
Muestra de operaciones sobre colecciones (incluidos conjuntos y secuencias)	
C−>size()	El número de elementos en la Colección c.
C−>isEmpty()	Verdadero si c no tiene elementos, falso de otro modo.
c1−>includesAll(c2)	Verdadero si cada elemento de c2 se encuentra en c1.
c1−>excludesAll(c2)	Verdadero si ningún elemento de c2 se encuentra en c1.
C−>forAll(elem | boolexpr)	Verdadero si boolexpr es verdadera cuando se aplica a cada elemento de c. Conforme se evalúa un elemento, se enlaza a la variable elem, que puede usarse en boolexpr. Esto implementa cuantificación universal, que se estudió anteriormente.
C−>forAll(elem1, elem2 | boolexpr)	Igual que el anterior, excepto que boolexpr se evalúa para cada posible *par* de elementos tomados de c, incluidos casos donde el par tiene el mismo elemento.
C−>isUnique(elem | expr)	Verdadero si expr evalúa un valor diferente cuando se aplica a cada elemento de c.
Muestra de operaciones específicas para conjuntos	
s1−>intersection(s2)	El conjunto de aquellos elementos que se encuentran en s1 y también en s2.
s1−>union(s2)	El conjunto de aquellos elementos que se encuentran en s1 o en s2.
s1−>excluding(x)	El conjunto s1 con la omisión del objeto x.
Muestra de operación específica a secuencias	
Seq−>first()	El objeto que es el primer elemento en la secuencia seq.

Como un lenguaje de programación orientado a objeto, una expresión OCL involucra operadores que operan sobre objetos. Sin embargo, el resultado de una expresión completa siempre debe ser booleana, es decir, verdadero o falso. Los objetos pueden ser instancias de la clase **Collection** OCL, de la cual **Set** (conjunto) y **Sequence** (secuencia) son dos subclases.

El objeto **self** es el elemento del diagrama UML en cuyo contexto se evaluará la expresión OCL. Al *navegar* usando el símbolo. (punto) del objeto **self** pueden obtenerse otros objetos. Por ejemplo:

- Si **self** es la clase **C**, con atributo a, entonces **self.a** evalúa al objeto almacenado en **a**.
- Si **C** tiene una asociación uno a muchos llamada *assoc* con otra clase **D**, entonces **self.assoc** evalúa un **Set** cuyos elementos son del tipo **D**.
- Finalmente (y un poco más sutilmente), si **D** tiene el atributo **b**, entonces la expresión **self.assoc.b** evalúa al conjunto de todos los b que pertenecen a todos los **D**.

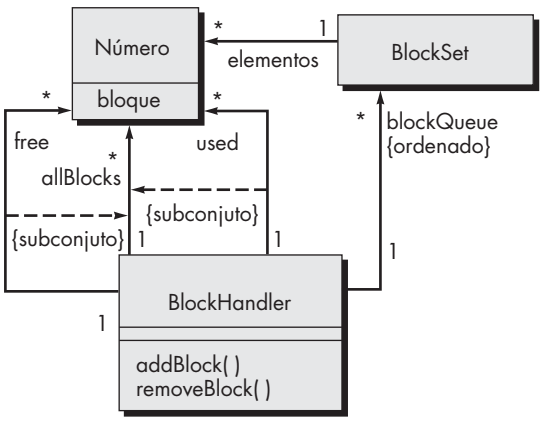


FIGURA 21.9
Diagrama de clase para un manipulador de bloques

OCL proporciona operaciones internas que implementan operadores de conjunto y lógicos, especificación constructiva y matemáticas relacionadas. En la tabla 21.1 se presenta una pequeña muestra de aquéllas.

Para ilustrar el uso de OCL en especificación, se reexamina el ejemplo de manipulador de bloques, que se introdujo en la sección 21.5. El primer paso es desarrollar un modelo UML (figura 21.9). Este diagrama de clase especifica muchas relaciones entre los objetos involucrados. Sin embargo, las expresiones OCL se agregan para que los implementadores del sistema puedan conocer con más precisión que debe permanecer verdadero conforme corre el sistema.

Las expresiones OCL que complementan el diagrama de clase corresponden a las seis partes de la invariante que se estudió en la sección 21.5. En el ejemplo que sigue, la invariante se repite en castellano y luego se escribe la correspondiente expresión OCL. Se considera buena práctica proporcionar texto en lenguaje natural junto con la lógica formal; hacerlo así ayuda a entender la lógica, y también ayuda a los revisores a descubrir errores, por ejemplo, situaciones donde el lenguaje natural y la lógica no corresponden.

1. *Ningún bloque se marcará como no utilizado y usado al mismo tiempo.*

```
context BlockHandler inv:
   (self.used2.intersection(self.free)) 2.isEmpty()
```

Observe que cada expresión comienza con la palabra clave ***context***. Esto indica el elemento del diagrama UML que restringe la expresión. De manera alternativa, podría colocar la restricción directamente en el diagrama UML, encerrada entre llaves. Aquí la palabra clave **self** se refiere a la instancia de **BlockHandler**; en lo que sigue, como es permisible en OCL, se omitirá el **self**.

2. *Todos los conjuntos de bloques que se conservan en la fila serán subconjuntos de la colección de los bloques actualmente utilizados.*

```
context BlockHandler inv:
   blockQueue2.forAll(aBlockSet | used2.includesAll(aBlockSet ))
```

3. *Ningún elemento de la fila contendrá el mismo número de bloque.*

```
context BlockHandler inv:
   blockQueue2.forAll(blockSet1, blockSet2 |
      blockSet1 ,. blockSet2 implies
      blockSet1.elements.number2.excludesAll(blockSet2elements.number))
```

La expresión antes de **implies** es necesaria para garantizar que se ignoran los pares donde ambos elementos son el mismo bloque.

4. *La colección de bloques utilizados y bloques que no se utilizan será la colección total de bloques que constituyen los archivos.*

```
context BlockHandler inv:
   allBlocks 5 used2.union(free)
```

5. *La colección de bloques no utilizados no tendrá números de bloque duplicados.*

```
context BlockHandler inv:
   free2.isUnique(aBlock | aBlock.number)
```

6. *La colección de bloques utilizados no tendrá números de bloque duplicados.*

```
context BlockHandler inv:
   used2.isUnique(aBlock | aBlock.number)
```

OCL también puede usarse para especificar precondiciones y poscondiciones de operaciones. Por ejemplo, lo siguiente describe operaciones que remueven y agregan conjuntos de bloques a la fila. Observe que la notación **x@pre** indica el objeto **x** como existe *antes* de la operación; esto es opuesto a la notación matemática estudiada anteriormente, donde la **x** está *después* de la operación que se designa especialmente (como **x′**).

```
context BlockHandler::removeBlocks()
    pre: blockQueue2.size() .0
    post: used 5 used@pre-blockQueue@pre2.first() and
        free = free@pre2.union(blockQueue@pre2.first()) and
        blockQueue 5 blockQueue@pre2.excluding(blockQueue@pre2.first)

context BlockHandler::addBlocks(aBlockSet :BlockSet)
   pre: used2.includesAll(aBlockSet.elements)
   post: (blockQueue.elements 5 blockQueue.elements@pre
      2. append (aBlockSet.elements) and
      used 5 used@pre and
      free 5 free@pre
```

OCL es un lenguaje de modelado, pero tiene todos los atributos de un lenguaje formal. OCL permite la expresión de varias restricciones, precondiciones y poscondiciones, guardias y otras características que se relacionan con los objetos representados en varios modelos UML.

21.7.2 El lenguaje de especificación Z

Z es un lenguaje de especificación que se usa ampliamente dentro de la comunidad de métodos formales. El lenguaje Z se aplica a conjuntos escritos, relaciones y funciones dentro del contexto de la lógica de predicados de primer orden para construir *esquemas*, un medio para estructurar la especificación formal.

WebRef

En **www.users.cs.york.ac.uk/~susan/abs/z.htm**, puede encontrarse información detallada acerca del lenguaje Z.

Las especificaciones Z se organizan como un conjunto de esquemas, una estructura de lenguaje que introduce variables y que especifica la relación entre dichas variables. Un esquema es en esencia la especificación formal análoga del componente de lenguaje de programación. Los esquemas se usan para estructurar una especificación formal en la misma forma en la que los componentes se usan para estructurar un sistema.

Un esquema describe los datos almacenados a los que accede y que altera un sistema. En el contexto de Z, esto se llama "estado". Este uso del término *estado* en Z es ligeramente diferente

del uso de la palabra en el resto de este libro.[9] Además, el esquema identifica las operaciones que se aplican para cambiar el estado y las relaciones que ocurren dentro del sistema. La estructura genérica de un esquema toma la forma:

esquemaNombre
declaraciones
---
invariante

donde las declaraciones identifican las variables que abarca el estado del sistema y la invariante impone restricciones en la forma en la que puede evolucionar el estado. En la tabla 21.2 se presenta un resumen de la notación del lenguaje Z.

El siguiente ejemplo de esquema describe el estado del manipulador de bloques y la invariante de datos:

BlockHandler
$used, free : \mathbb{P}\ BLOCKS$
$BlockQueue : \text{seq}\ \mathbb{P}\ BLOCKS$
$used \cap free = \varnothing \land$
$used \cup free = AllBlocks \land$
$\forall i: \textbf{dom}\ BlockQueue \bullet BlockQueue\ i \subseteq used \land$
$\forall i, j: \text{dom}\ BlockQueue \bullet i \neq j \Rightarrow BlockQueue\ i \cap BlockQueue\ j = \varnothing$

Como se anotó, el esquema consiste de dos partes. La parte que está arriba de la línea central representa las variables del estado, mientras que la parte de abajo de la línea central describe la invariante de datos. Siempre que el esquema especifique operaciones que cambian al estado, se precede con el símbolo Δ. El siguiente ejemplo de esquema describe la operación que remueve un elemento de la fila de bloques:

RemoveBlocks
$\Delta\ BlockHandler$

$\#BlockQueue > 0,$
$used' = used \setminus head\ BlockQueue \land$
$free' = free \cup head\ BlockQueue \land$
$BlockQueue' = tail\ BlockQueue$

La inclusión de Δ *BlockHandler* da como resultado todas las variables que constituyen el estado disponible para el esquema *RemoveBlocks* y garantiza que la invariante de datos se sostendrá antes y después de ejecutar la operación.

La segunda operación, que agrega una colección de bloques al final de la fila, se representa como

AddBlocks
$\Delta\ BlockHandler$
Ablocks? : BLOCKS

9 Recuerde que, en otros capítulos, *estado* se usó para identificar un modo de comportamiento observable en el exterior para un sistema.

TABLA 21.2 RESUMEN DE NOTACIÓN Z

La notación Z se basa en la teoría de conjuntos descrita y en lógica de primer orden. Z proporciona un constructo, llamado esquema, para describir el espacio y las operaciones de estado de una especificación. Un esquema agrupa declaraciones de variables con una lista de predicados que restringen el posible valor de una variable. En Z, el esquema X se define mediante la forma

———X———————————————
declaraciones
———————————————————
predicados
———————————————————

Las funciones globales y las constantes se definen mediante la forma

declaraciones
———————————————————
predicados

La declaración brinda el tipo de la función o constante, mientras que el predicado proporciona su valor. En esta tabla sólo se presenta un conjunto abreviado de símbolos Z.

Conjuntos:

$S : \mathbb{P}\, X$	*S* se declara como un conjunto de *X*s.
$X \in S$	x es miembro de *S*.
$x \notin S$	x no es miembro de *S*.
$S \subseteq T$	*S* es un subconjunto de *T*: todo miembro de *S* también está en *T*.
$S \cup T$	La unión de *S* y *T*: contiene a todo miembro de *S* o *T* o ambos.
$S \cap T$	La intersección de *S* y *T*: contiene a todo miembro tanto de *S* como de *T*.
$S \setminus T$	La diferencia de *S* y *T*: contiene todo miembro de *S* excepto aquellos que también están en *T*.
$\varnothing$	Conjunto vacío: no contiene miembros.
{x}	Conjunto de un solo elemento: sólo contiene a x.
$\mathbb{N}$	Conjunto de los números naturales 0, 1, 2, ...
$S : \mathbb{F}\, X$	*S* se declara como un conjunto finito de *X*s.
max (*S*)	El máximo del conjunto no vacío de números *S*.

Funciones:

$f : X \rightarrowtail Y$	*f* se declara como una inyección parcial de *X* a *Y*.
dom *f*	El dominio de f: el conjunto de valores *x* para los cuales se define *f*(*x*).
ran *f*	El rango de f: el conjunto de valores tomados por *f*(*x*) conforme *x* varía sobre el dominio de *f*.
$f \oplus \{x \mapsto y\}$	Una función que concuerda con *f* excepto que *x* se mapea a *y*.
$\{x\} \lhd\!\!\!- f$	Una función como *f*, excepto que *x* se remueve de su dominio.

Lógica:

$P \land Q$	*P* y *Q*: es verdadero si tanto *P* como *Q* son verdaderos.
$P \Rightarrow Q$	*P* implica a *Q*: es verdadero si *Q* es verdadero o *P* es falso.
$\theta\, S' = \theta\, S$	Ningún componente del esquema *S* cambia en una operación.

$Ablocks? \subseteq used$
$BlockQueue' = BlockQueue \frown \langle \text{Ablocks?} \rangle \land$
$used' = used \land$
$free' = free$
———————————————————

Por convención en Z, una variable de entrada que se lea, pero que no forme parte del estado, termina con un signo de interrogación. Por tanto, Ablocks?, que actúa como un parámetro de entrada, termina con un signo de interrogación.

HERRAMIENTAS DE SOFTWARE

Métodos formales

Objetivo: El objetivo de las herramientas de métodos formales es auxiliar al equipo de software en la verificación de especificación y de exactitud.

Mecánica: La mecánica de las herramientas varía. En general, las herramientas auxilian a probar la especificación y la exactitud de la automatización, que por lo general se define mediante un lenguaje especializado para probar los teoremas. Muchas herramientas no se comercializan y se desarrollaron con propósitos de investigación.

Herramientas representativas:[10]

ACL2, desarrollada en la Universidad de Texas (**www.cs.utexas.edu/users/moore/acl2/**), es "tanto un lenguaje de programación en el que pueden modelarse sistemas de cómputo como una herramienta para ayudarle a probar las propiedades de dichos modelos".

EVES, desarrollado por ORA Canadá (**www.ora.on.ca/eves.html**), implementa el lenguaje Verdi para especificación formal y un generador de prueba automático.

En **http://vl.fmnet.info/** puede encontrar una extensa lista de más de 90 herramientas de métodos formales.

21.8 RESUMEN

La ingeniería del software de cuarto limpio es un enfoque formal del desarrollo de software que puede conducir a software con una calidad notablemente alta. Usa la especificación de estructura de cajas para el análisis y el modelado del diseño, y enfatiza la verificación de la exactitud, en lugar de las pruebas, como el mecanismo primario para encontrar y remover errores. La prueba de uso estadístico se aplica para desarrollar la información de tasa de fallos necesaria para certificar la confiabilidad del software entregado.

El enfoque de cuarto limpio comienza con los modelos de análisis y diseño que usan una representación de estructura de cajas. Una "caja" encapsula el sistema (o algún aspecto de él) en un nivel específico de abstracción. Las cajas negras se usan para representar el comportamiento externamente observable de un sistema. Las cajas de estado encapsulan los datos y operaciones de estado. Una caja clara se usa para modelar el diseño procedural que se implica mediante los datos y operaciones de una caja de estado.

La verificación de exactitud se aplica una vez que está completo el diseño de estructura de cajas. El diseño procedural para un componente de software se divide en una serie de subfunciones. Para probar la exactitud de las subfunciones, se definen condiciones de salida para cada subfunción y se aplica un conjunto de subpruebas. Si cada condición de salida se satisface, el diseño debe ser correcto.

Una vez que está completa la verificación de exactitud, comienza la prueba de uso estadístico. A diferencia de las pruebas convencionales, la ingeniería del software de cuarto limpio no enfatiza las pruebas de unidad o de integración. En vez de ello, el software se prueba al definir un conjunto de escenarios de uso, al determinar la probabilidad de uso para cada escenario y luego definir pruebas aleatorias que se conformen con las probabilidades. Los registros de error que resultan se combinan con modelos de muestreo, componentes y certificación para habilitar el cálculo matemático de la confiabilidad proyectada para el componente de software.

Los métodos formales usan las facilidades descriptivas de la teoría de conjuntos y la notación lógica para permitir que un ingeniero de software cree un enunciado claro de los hechos (requerimientos). Los conceptos subyacentes que gobiernan los métodos formales son: 1) la invariante de datos, una condición verdadera a lo largo de la ejecución del sistema que contiene una co-

10 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

lección de datos; 2) el estado, una representación del modo de comportamiento observable externamente a un sistema o (en lenguajes Z o relacionados) los datos almacenados a los que un sistema accede o que altera; y 3) la operación, una acción que tiene lugar en un sistema y que lee o escribe datos a un estado. Una operación se asocia con dos condiciones: una precondición y una poscondición.

¿Alguna vez la ingeniería del cuarto limpio o los métodos formales se usarán ampliamente? La respuesta es "probablemente no". Son más difíciles de aprender que los métodos de ingeniería del software convencional y representan un significativo "choque cultural" para algunos profesionales del software. Pero la siguiente vez que escuche a alguien lamentarse: "¿Por qué este software no puede quedar bien desde la primera vez?", sabrá que hay técnicas que le ayudan a hacer exactamente eso.

Problemas y puntos por evaluar

21.1. Si tuviera que elegir un aspecto de la ingeniería del software de cuarto limpio que la haga radicalmente diferente de los enfoques de ingeniería del software convencional o de la orientada a objeto, ¿cuál sería?

21.2. ¿Cómo trabajan en conjunto un modelo de proceso incremental y la certificación para producir software de alta calidad?

21.3. Con la especificación de estructura de cajas, desarrolle modelos de análisis y diseño de "primer paso" para el sistema *CasaSegura*.

21.4. Un algoritmo de ordenamiento en burbujas (*bubble-sort*) se define de la manera siguiente:

```
procedure bubblesort;
var i, t, integer;
begin
repeat until t5a[1]
        t:5a[1];
        for j:5 2 to n do
            if a[j-1] . a[j] then begin
                t:5a[j-1];
                a[j-1]:5a[j];
                a[j]:5t;
                end
endrep
end
```

Divida el diseño en subfunciones y defina un conjunto de condiciones que le permitirían probar que este algoritmo es correcto.

21.5. Documente una prueba de verificación de exactitud para el ordenamiento en burbujas estudiado en el problema 21.4.

21.6. Seleccione un programa que usted use regularmente (por ejemplo, manejador de correo electrónico, procesador de palabra, programa de hoja de cálculo). Cree un conjunto de escenarios de uso para el programa. Defina la probabilidad de uso para cada escenario y luego desarrolle una tabla de estímulos del programa y distribución de probabilidad, similar a la que se muestra en la sección 21.4.1.

21.7. Para la tabla de estímulos del programa y distribución de probabilidad que desarrolló en el problema 21.6, use un generador de números aleatorios para desarrollar un conjunto de casos de prueba para usar en la prueba de uso estadístico.

21.8. Con sus palabras, describa la intención de la certificación en el contexto de la ingeniería del software de cuarto limpio.

21.9. Al lector lo asignan a un equipo que desarrolla software para un fax módem. Su labor es desarrollar la porción de "directorio" de la aplicación. La función directorio permite el almacenamiento de hasta *MaxNombres* personas junto con los nombres de compañía asociados, números de fax y otra información relacionada. Use lenguaje natural para definir

a) La invariante de datos.
b) El estado.
c) Las probables operaciones.

21.10. Al lector se le asigna a un equipo de software que desarrolla software, llamado MemoriaDuplicador, que proporciona a una PC mayor memoria aparente que la memoria física. Esto se logra al identificar, recopilar y reasignar bloques de memoria que se asignaron a una aplicación existente, pero que no se utilizan. Los bloques no utilizados se reasignan a aplicaciones que requieren memoria adicional. Realice las suposiciones adecuadas y use lenguaje natural para definir

a) La invariante de datos.
b) El estado.
c) Las probables operaciones.

21.11. Use la notación OCL o la Z que se presentó en las tabla 21.1 o 21.2, seleccione alguna parte del sistema de seguridad *CasaSegura* descrito anteriormente en este libro e intente especificarla con OCL o con Z.

21.12. Use una o más de las fuentes de información anotadas en las referencias o en *Lecturas adicionales y fuentes de información* de este capítulo, para desarrollar una presentación de media hora acerca de la sintaxis y la semántica básicas de un lenguaje de especificación formal distinto a OCL o a Z.

Lecturas adicionales y fuentes de información

En años recientes, se han publicado relativamente pocos libros acerca de las técnicas de especificación y verificación avanzadas. Sin embargo, vale la pena considerar algunas de las nuevas adiciones a la literatura. Un libro editado por Gabbar (*Modern formal Methods and Applications*, Springer, 2006) presenta tanto fundamentos, como nuevos desarrollos y aplicaciones avanzadas. Jackson (*Software Abstractions*, the MIT Press, 2006) presenta todos los fundamentos y un enfoque que él llama "métodos formales ligeros". Monin y Hinchey (*Understanding formal Methods*, Springer, 2003) ofrecen una excelente introducción a la materia. Butler *et al.* (*Integrated Formal Methods*, Springer, 2002) presentan varios artículos acerca del tema de los métodos formales.

Además de los libros indicados en este capítulo, Prowell *et al.* (*Cleanroom Software Engineering: Technology and Process*, Addison-Wesley, 1999) proporcionan un tratamiento a profundidad de todos los aspectos importantes del enfoque de cuarto limpio. Poore y Trammell (*Cleanroom Software Engineering: A Reader*, Blackweel Publishing, 1996) editaron útiles análisis de temas de cuarto limpio. Becker y Whittaker (*Cleanroom Software Engineering Practies*, Idea Group Publishing, 1996) presentan un excelente panorama para quienes no están familiarizados con las prácticas de cuarto limpio.

El Cleanroom Pamphlet (Software Technology Support Center, Hill AF Base, abril de 1995) contiene reimpresiones de algunos artículos importantes. El Data and Analysis Center for Software (DACS) (**www.dacs.dtic.mil**) proporciona muchos ensayos útiles, manuales y otras fuentes de información acerca de la ingeniería del software de cuarto limpio.

La verificación de diseño mediante prueba de exactitud se encuentra en el centro del enfoque de cuarto limpio. Los libros de Cupillari (*The Nuts and Bolts of Proofs*, 3a ed., Academic Press, 2005), Solow (*How to Read and Do Proofs,* 4a. ed., Wiley, 2004), Eccles (*An Introduction to Mathematical Reasoning,* Cambridge University Press, 1998), proporcionan excelentes introducciones a los principios matemáticos. Stavely (*Toward Zero-Defect Software,* Addison-Wesley, 1998), Baber (*Error-Free Software,* Wiley, 1991) y Schulmeyer (*Zero Defect Software,* McGraw-Hill, 1990) estudian la prueba de exactitud con detalle considerable.

En el dominio de los métodos formales, los libros de Casey (*A Programming Approach to Formal Methods,* McGraw-Hill, 2000), Hinchey y Bowan (*Industrial Strength Formal Methods,* Springer-Verlag, 1999), Hussmann (*Formal Foundations for Software Engineering Methods,* Springer-Verlag, 1997) y Sheppard (*An Introduction to Formal Specification with Z and VDM,* McGraw-Hill, 1995) ofrecen guías útiles. Además, libros específicos sobre lenguaje, como los de Warmer y Kleppe (*Object Constraint Language,* Addison-Wesley, 1998), Jacky (*The Way of Z: Practical Programming with Formal Methods,* Cambridge University Press, 1997), Harry (*Formal Methods Fact File: VDM and Z,* Wiley, 1997) y Cooper y Barden (*Z in Practice,* Prentice-Hall, 1995) proporcionan introducciones útiles a los métodos formales, así como a varios lenguajes de modelado.

En internet está disponible una gran variedad de fuentes de información acerca de ingeniería del software de cuarto limpio y de los métodos formales. Una lista actualizada de referencias en la World Wide Web que son relevantes para el modelado formal y la verificación puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

22

ADMINISTRACIÓN DE LA CONFIGURACIÓN DEL SOFTWARE

CONCEPTOS CLAVE

administración de contenido . 517
auditoría de configuración . . 514
líneas de referencia 504
control de cambio 511
control de versión 510
identificación. 509
ítems de configuración de software (ICS) 505
objeto de configuración 505
objetos de configuración de *webapps* 517
proceso ACS 508
reporte de estado 615
repositorio 506
webapps 515

Cuando se construye software de computadoras, el cambio es inevitable. Y el cambio aumenta el nivel de confusión cuando los miembros de un equipo de software trabajan en un proyecto. La confusión surge cuando los cambios no se analizan antes de que se realicen, cuando no se registran antes de que se implanten, si no se reportan a quienes tienen necesidad de conocerlos o si no se controlan en forma que mejore la calidad y se reduzca el error. Babich [Bab86] analiza esto cuando afirma:

> El arte de coordinar el desarrollo de software para minimizar [...] la confusión se llama administración de la configuración, que es el arte de identificar, organizar y controlar las modificaciones que se hacen al software que construirá un equipo de programación. La meta es maximizar la productividad al minimizar los errores.

La administración de la configuración del software (ACS) es una actividad sombrilla que se aplica a lo largo del proceso de software. Puesto que el cambio puede ocurrir en cualquier momento, se desarrollan actividades ACS para 1) identificar el cambio, 2) controlar el cambio, 3) garantizar que el cambio se implementó de manera adecuada y 4) reportar los cambios a otros que puedan estar interesados.

Es importante hacer una distinción clara entre el apoyo al software y la administración de la configuración del software. El apoyo es un conjunto de actividades de ingeniería de software que ocurren después de que éste se entregó al cliente y de que se puso en operación. La administración de la configuración del software es un conjunto de actividades de rastreo y control que inicia cuando comienza un proyecto de ingeniería de software y sólo termina cuando el software se retira de la operación.

UNA MIRADA RÁPIDA

¿Qué es? Cuando se construye software de computadora, ocurren cambios. Y puesto que ocurren, es necesario administrarlos de manera efectiva. La administración de la configuración del software (ACS), también llamada gestión del cambio, es un conjunto de actividades diseñadas para administrar el cambio mediante la identificación de los productos de trabajo que es probable que cambien, el establecimiento de relaciones entre ellos, la definición de mecanismos para administrar diferentes versiones de dichos productos de trabajo y el control de los cambios impuestos, así como la auditoría y reporte de los cambios realizados.

¿Quién lo hace? Todos los involucrados en el proceso de software se relacionan en cierta medida con la gestión del cambio, pero en ocasiones se crean posiciones de apoyo especializadas para administrar el proceso ACS.

¿Por qué es importante? Si no se controla el cambio, éste lo controla a uno. Y eso nunca es bueno. Es muy fácil que un torrente de cambios descontrolados convierta en caos un proyecto de software bien estructurado. Como consecuencia, la calidad del software se reduce y la entrega se demora. Por dicha razón, la gestión del cambio es parte esencial de la administración de la calidad.

¿Cuáles son los pasos? Puesto que muchos productos de trabajo se realizan cuando el software se construye, cada uno debe identificarse de manera única. Una vez logrado, pueden establecerse mecanismos para control de versión y de cambio. Para garantizar que la calidad se mantiene conforme se realizan cambios, audite el proceso; y para asegurarse que quienes deben conocerlos estén informados acerca de los cambios, realice reportes.

¿Cuál es el producto final? Un Plan de Administración de la Configuración del Software define la estrategia del proyecto para la gestión del cambio. Además, cuando se invoca ACS formal, el proceso de control de cambio produce solicitudes de cambio de software, reportes y órdenes de cambio de ingeniería.

¿Cómo me aseguro de que lo hice bien? Cuando todo producto de trabajo pueda explicarse, rastrearse y controlarse; cuando todo cambio pueda rastrearse y analizarse; cuando todos los que deben saber acerca de un cambio están informados, entonces la gestión del cambio se hizo correctamente.

Una meta principal de la ingeniería de software es mejorar la facilidad con la que los cambios pueden acomodarse y reducir la cantidad de esfuerzo empleado cuando deban realizarse cambios. En este capítulo se estudian las actividades específicas que permiten administrar el cambio.

22.1 Administración de la configuración del software

La salida del proceso de software es información que puede dividirse en tres categorías amplias: 1) programas de cómputo (tanto en el nivel de fuente como en formatos ejecutables), 2) productos de trabajo que describen los programas de cómputo (dirigidos a varios participantes) y 3) datos o contenido (incluidos dentro del programa o externos a él). Los ítems que comprenden toda la información producida como parte del proceso de software se llaman colectivamente *configuración del software*.

Cita:

"No hay nada permanente, excepto el cambio."

Heráclito, 500 a.C.

Conforme avanza el trabajo de ingeniería de software, se crea una jerarquía de *ítems de configuración del software* (ICS): un elemento de información nominado que puede ser tan pequeño como un solo diagrama UML o tan grande como el documento de diseño completo. Si cada ICS simplemente conduce a otros ICS, dará como resultado poca confusión. Por desgracia, en el proceso entra otra variable: *el cambio*, que puede ocurrir en cualquier momento, por cualquier razón. De hecho, la *Primera Ley de la Ingeniería de Sistemas* [Ber80] establece: "Sin importar dónde se esté en el ciclo de vida del sistema, el sistema cambiará, y el deseo por cambiar persistirá a lo largo del ciclo de vida."

¿Cuál es el origen de estos cambios? La respuesta a esta pregunta es tan variada como los mismos cambios. Sin embargo, existen cuatro fuentes fundamentales de cambio:

¿Cuál es el origen de los cambios que se solicitan para el software?

- Nuevas condiciones empresariales o de mercado dictan los cambios en los requerimientos del producto o en las reglas empresariales.
- Nuevas necesidades de los accionistas demandan modificación a los datos producidos por los sistemas de información, a la funcionalidad que entregan los productos o a los servicios que ofrece un sistema basado en computadora.
- La reorganización o crecimiento/reducción de la empresa produce cambios en las prioridades proyectadas o en la estructura del equipo de ingeniería de software.
- Restricciones presupuestales o de calendario causan una redefinición del sistema o del producto.

La administración de la configuración del software es un conjunto de actividades que se desarrollaron para administrar el cambio a lo largo del ciclo de vida del software de computadora. La ACS puede verse como una actividad que garantiza la calidad del software y que se aplica a lo largo del proceso de software. En las siguientes secciones se describen las principales tareas ACS y los conceptos importantes que pueden ayudar a gestionar el cambio.

22.1.1 Un escenario ACS[1]

Un escenario operativo de administración del cambio (AC) típico involucra a un gerente de proyecto que está a cargo de un grupo de software, a un gerente de configuración responsable de los procedimientos y políticas AC, a los ingenieros de software encargados de desarrollar y mantener el producto de software y al cliente que usa el producto. En el escenario, se supone que el producto es pequeño y que involucra a alrededor de 15 000 líneas de código desarrollado por un equipo de seis personas. (Observe que es posible que existan otros escenarios de equipos

1 Esta sección se extrajo de [Dar01]. El permiso especial para reproducir "Spectrum of functionality in CM System", de Susan Dart [Dar01], © 2001 de Carnegie Mellon University, se obtuvo del Software Engineering Institute.

más pequeños o más grandes, pero, en esencia, hay temas genéricos que cada uno de estos proyectos enfrenta con respecto a la AC).

¿Cuáles son las metas de y las actividades realizadas por cada uno de los elementos constituyentes involucrados en la administración del cambio?

En el nivel operativo, el escenario involucra varios roles y tareas. Para el gerente de proyecto, la meta es garantizar que el producto se desarrolla dentro de cierto marco temporal. Por tanto, monitorea el progreso del desarrollo y reconoce y reacciona ante los problemas. Esto se hace mediante la generación y el análisis de reportes acerca del estado del sistema de software y al realizar revisiones al sistema.

Las metas del gerente de configuración son garantizar que se sigan los procedimientos y políticas para crear, cambiar y probar el código, así como hacer accesible la información acerca del proyecto. Para implantar técnicas a fin de mantener el control sobre los cambios de código, este gerente introduce mecanismos para: realizar peticiones oficiales de cambios, evaluarlos (mediante un Consejo de Control de Cambios que sea responsable de aprobar los cambios al sistema de software) y autorizarlos. El gerente elabora y difunde la lista de tareas para los ingenieros y básicamente crea el contexto del proyecto. Además, recopila estadísticas acerca de los componentes que hay en el sistema de software, tales como la información que determina cuáles componentes del sistema son problemáticos.

PUNTO CLAVE

Debe existir un mecanismo para asegurar que los cambios simultáneos hechos al mismo componente se rastreen, gestionen y ejecuten de manera adecuada.

Para los ingenieros de software, la meta es trabajar eficazmente. Esto significa que los ingenieros no deben interferir innecesariamente unos con otros en la creación y prueba del código y en la producción de productos operativos de apoyo. Pero, al mismo tiempo, deben intentar comunicarse y coordinarse de manera eficiente. Específicamente, los ingenieros usan herramientas que ayudan a construir un producto de software consistente. Se comunican y coordinan al notificarse unos con otros las tareas requeridas y las tareas completadas. Los cambios se propagan a través del trabajo mutuo mediante fusión de archivos. Existen mecanismos para garantizar que, para componentes que experimentan cambios simultáneos, hay alguna forma de resolver los conflictos y la fusión de cambios. Se conserva una historia de la evolución de todos los componentes del sistema, una bitácora con las razones de los cambios y un registro de lo que realmente cambió. Los ingenieros tienen su propio espacio de trabajo para crear, cambiar, poner a prueba e integrar código. En cierto punto, el código se convierte en una línea de referencia desde la cual continúan mayores desarrollos y se realizan variantes para otras máquinas objetivo.

El cliente usa el producto. Puesto que éste se encuentra bajo control AC, el cliente sigue procedimientos formales para solicitar cambios y para indicar errores en el producto.

De manera ideal, un sistema AC utilizado en este escenario debe apoyar todos estos roles y tareas, es decir, los roles determinan la funcionalidad requerida de un sistema AC. El gerente de proyecto ve la AC como un mecanismo de auditoría; el gerente de configuración la considera un mecanismo de control, rastreo y generación de políticas; el ingeniero de software, como un mecanismo de control de cambio, construcción y acceso; y el cliente la ve como un camino para garantizar la calidad.

22.1.2 Elementos de un sistema de administración de la configuración

En su exhaustivo artículo acerca de la administración de la configuración del software, Susan Dart [Dar01] identifica cuatro elementos importantes que deben existir cuando se desarrolla un sistema de administración de la configuración:

- *Elementos componentes:* conjunto de herramientas acopladas dentro de un sistema de administración de archivos (por ejemplo, base de datos) que permite el acceso a cada ítem de configuración del software, así como su gestión.
- *Elementos de proceso:* colección de acciones y tareas que definen un enfoque efectivo de la gestión del cambio (y actividades relacionadas) para todos los elementos constituyentes involucrados en la administración, ingeniería y uso del software.

- *Elementos de construcción:* conjunto de herramientas que automatizan la construcción de software al asegurarse de que se ensambló el conjunto adecuado de componentes validados (es decir, la versión correcta).
- *Elementos humanos:* conjunto de herramientas y características de proceso (que abarcan otros elementos AC) utilizados por el equipo de software para implementar ACS efectiva.

Estos elementos (que se estudiarán con más detalle en secciones posteriores) no son mutuamente excluyentes. Por ejemplo, los elementos componentes trabajan en conjunción con los elementos de construcción conforme evoluciona el proceso de software. Los elementos de proceso guían muchas actividades humanas que se relacionan con la ACS y, por tanto, también pueden considerarse como elementos humanos.

22.1.3 Líneas de referencia

La mayoría de los cambios de software se justifican, así que no hay razón para quejarse por su presencia. En su lugar, asegúrese de que tiene los mecanismos para lidiar con ellos.

El cambio es un hecho de vida en el desarrollo de software. Los clientes quieren modificar los requerimientos. Los desarrolladores quieren cambiar el enfoque técnico. Los gerentes quieren modificar la estrategia del proyecto. ¿Por qué todas estas modificaciones? La respuesta realmente es muy simple. Conforme pasa el tiempo, todos los elementos constituyentes saben más (acerca de lo que necesitan, sobre qué enfoque sería mejor, cómo realizarlo y todavía obtener dinero). Este conocimiento adicional es la fuerza motora que hay detrás de la mayoría de los cambios y que conduce a un enunciado de hechos que es difícil de aceptar para muchos profesionales de la ingeniería de software: *¡la mayoría de los cambios se justifican!*

Una *línea de referencia* es un concepto de administración de la configuración del software que le ayuda a controlar el cambio sin impedir seriamente cambios justificados. El IEEE (IEEE Std. No. 610.12-1990) define una línea de referencia como:

> Una especificación o producto que se revisó formalmente y con el que se estuvo de acuerdo, que a partir de entonces sirve como base para un mayor desarrollo y que puede cambiar sólo a través de procedimientos de control de cambio formal.

Antes de que un ítem de configuración del software se convierta en línea de referencia, los cambios pueden realizarse rápida e informalmente. No obstante, una vez establecida la línea de referencia, pueden realizarse cambios, pero debe aplicarse un procedimiento formal específico para evaluar y verificar cada uno de ellos.

En el contexto de la ingeniería de software, una línea de referencia es un hito en el desarrollo del software. Una línea de referencia se marca al entregar uno o más ítems de configuración del software que se aprobaron como consecuencia de una revisión técnica (capítulo 15). Por ejemplo, los elementos de un modelo de diseño se documentaron y revisaron. Se encontraron y corrigieron errores. Una vez que todas las partes del modelo se revisaron, corrigieron y luego aprobaron, el modelo de diseño se convierte en línea de referencia. Los cambios adicionales a la arquitectura del programa (documentada en el modelo de diseño) pueden realizarse sólo después de que cada uno se evalúa y aprueba. Aunque las líneas de referencia pueden definirse en cualquier nivel de detalle, en la figura 22.1 se muestran las líneas de referencia de software más comunes.

Asegúrese de que la base de datos del proyecto se mantenga en una ubicación centralizada controlada.

En la figura 22.1 también se ilustra la progresión de eventos que conducen a una línea de referencia. Las tareas de la ingeniería de software producen uno o más ICS. Después de revisar y aprobar los ICS, se colocan en una *base de datos del proyecto* (también llamada *librería de proyecto* o *repositorio de software*, y que se estudia en la sección 22.2). Cuando un miembro de un equipo de ingeniería de software quiere hacer una modificación a un ICS que se ha convertido en línea de referencia, se copia de la base de datos del proyecto en el espacio de trabajo privado del ingeniero. Sin embargo, este ICS extraído puede modificarse solamente si se siguen controles ACS (que se estudian más adelante en este capítulo). Las flechas de la figura 22.1 ilustran la ruta de modificación de un ICS convertido en línea de referencia.

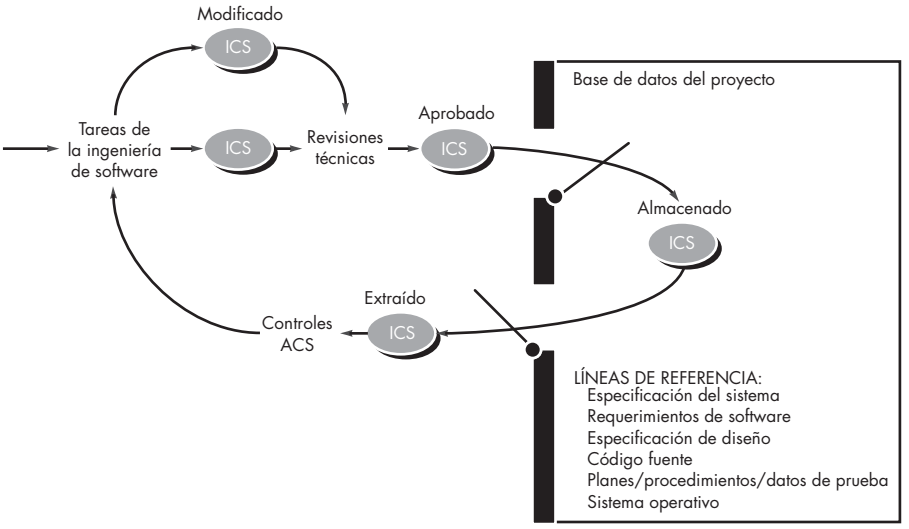


FIGURA 22.1
ICS como línea de referencia y base de datos del proyecto

22.1.4 Ítems de configuración del software

Ya se definió un ítem de configuración del software como la información que se crea como parte del proceso de ingeniería de software. En última instancia, un ICS podría considerarse como una sola sección de una gran especificación o como un caso de prueba en una gran suite de pruebas. De manera más realista, un ICS es todo o parte de un producto de trabajo (por ejemplo, un documento, toda una suite de casos de prueba o un componente de programa nominado).

Además de los ICS que se derivan de los productos de trabajo de software, muchas organizaciones de ingeniería de software también colocan las herramientas de software bajo control de configuración, es decir, versiones específicas de editores, compiladores, navegadores y otras herramientas automatizadas se "congelan" como parte de la configuración del software. Puesto que dichas herramientas se usaron para producir documentación, código fuente y datos, deben estar disponibles cuando tengan que realizarse cambios a la configuración del software. Aunque los problemas son raros, es posible que una nueva versión de una herramienta (por ejemplo, un compilador) pueda producir resultados diferentes que la versión original. Por esta razón, las herramientas, como el software que ayudan a producir, pueden convertirse en líneas de referencia como parte de un proceso amplio de administración de la configuración.

En realidad, los ICS se organizan para formar objetos de configuración que puedan catalogarse con un solo nombre en la base de datos del proyecto. Un *objeto de configuración* tiene un nombre y atributos, y está "conectado" con otros objetos mediante relaciones. En la figura 22.2, los objetos de configuración **DesignSpecification** (especificación de diseño), **DataModel** (modelo de datos), **ComponentN** (componente n), **SourceCode** (código fuente) y **TestSpecification** (especificación de prueba) se definen cada uno por separado. Sin embargo, cada uno de los objetos se relaciona con los demás, como se muestra mediante las flechas. Una flecha curva indica una relación composicional, es decir, **DataModel** y **ComponentN** son parte del objeto **DesignSpecification**. Una flecha con doble punta indica una interrelación. Si se realizara un cambio al objeto **SourceCode**, las interrelaciones permiten determinar qué otros objetos (e ICS) pueden resultar afectados.[2]

2 Esas relaciones se definen dentro de la base de datos. La estructura de la base de datos (repositorio) se estudia con mayor detalle en la sección 22.2.

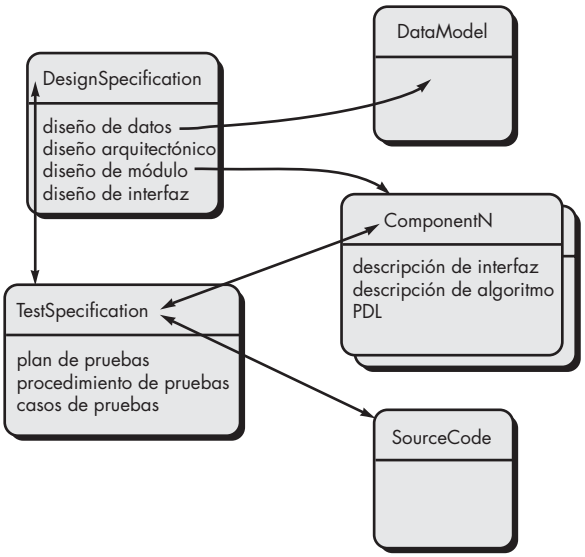


FIGURA 22.2
Objetos de configuración

22.2 EL REPOSITORIO ACS

En los primeros días de la ingeniería de software, los ítems de configuración del software se mantenían como documentos en papel (¡o tarjetas perforadas!), colocadas en carpetas de papel o de anillos, y se almacenaban en archiveros metálicos. Este mecanismo era problemático por muchas razones: 1) con frecuencia era difícil encontrar un ítem de configuración cuando se necesitaba, 2) era muy desafiante determinar cuáles ítems cambiaban, cuándo y por quién, 3) construir una nueva versión de un programa existente consumía mucho tiempo y era proclive al error y 4) describir relaciones detalladas y complejas entre los ítems de configuración era virtualmente imposible.

En la actualidad, los ICS se mantienen en una base de datos del proyecto, o repositorio. El *Diccionario Webster* define la palabra *repositorio* como "cualquier cosa o persona que se considera como centro de acumulación o almacenamiento". Durante la historia temprana de la ingeniería de software, de hecho el repositorio era una persona: el programador que debía recordar la ubicación de toda la información relevante para un proyecto de software, quien debía recuperar la información que nunca se escribió y reconstruir la información perdida. Tristemente, usar a una persona como "el centro para acumulación y almacenamiento" (aun conforme con la definición del Webster) no funciona muy bien. En la actualidad, el repositorio es una "cosa": una base de datos que actúa como el centro de acumulación y de almacenamiento de la información de ingeniería de software. El papel de la persona (el ingeniero del software) es interactuar con el repositorio, usando las herramientas que se integran con él.

22.2.1 El papel del repositorio

El repositorio ACS es el conjunto de mecanismos y estructuras de datos que permiten a un equipo de software administrar el cambio en forma efectiva. Proporciona las funciones obvias de un moderno sistema de administración de base de datos, al asegurar integridad, posibilidad de compartir e integración de datos. Además, el repositorio ACS proporciona un centro para la integración de herramientas de software, es fundamental en el flujo del proceso de software y puede reforzar la estructura y el formato uniforme para los productos que son resultado de la ingeniería de software.

Para lograr estas capacidades, el repositorio se define como un metamodelo. El *metamodelo* determina cómo se almacena la información en el repositorio, cómo pueden acceder las herramientas a los datos y cómo pueden verlas los ingenieros de software, cuán bien pueden mantenerse la seguridad y la integridad de los datos y cuán fácilmente puede extenderse el modelo existente para alojar nuevas necesidades.

22.2.2 Características y contenido generales

Las características y el contenido del repositorio se entienden mejor al observarlo desde dos perspectivas: qué debe almacenarse en él y qué servicios específicos proporciona. En la figura 22.3 se presenta un desglose detallado de los tipos de representaciones, documentos y otros productos de trabajo.

WebRef

En **www.oracle.com/technology/products/repository/index.html** puede obtenerse un ejemplo de repositorio disponible en el mercado.

Un repositorio robusto proporciona dos clases de servicios diferentes: 1) los mismos tipos de servicios que pueden esperarse de cualquier sistema sofisticado de administración de base de datos y 2) los servicios que son específicos del entorno de ingeniería de software.

Un repositorio que sirve a un equipo de ingeniería de software también debe: 1) integrarse con o directamente apoyar las funciones de administración del proceso, 2) apoyar reglas específicas que gobiernan la función ACS y los datos mantenidos dentro del repositorio, 3) proporcionar una interfaz hacia otras herramientas de ingeniería de software y 4) acomodar almacenamiento de objetos de datos sofisticados (por ejemplo, texto, gráficos, video, audio).

PUNTO CLAVE

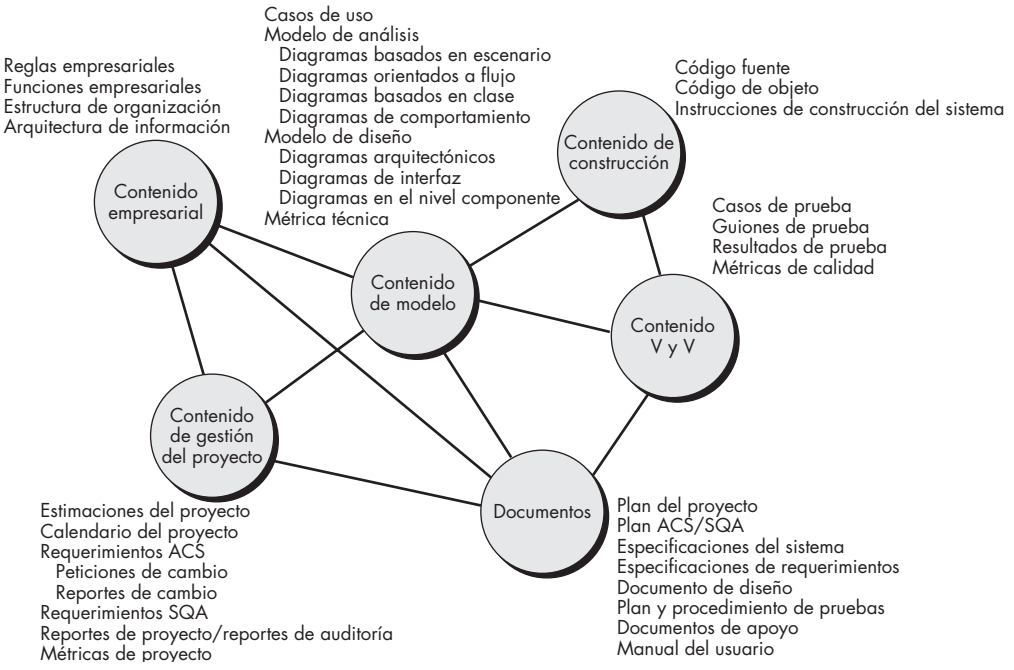
El repositorio debe mantener relacionados los ICS con muchas diferentes versiones del software. Más importante, debe proporcionar los mecanismos para ensamblar dichos ICS en una configuración específica de una versión.

22.2.3 Características ACS

Para apoyar el ACS, el repositorio debe tener un conjunto de herramientas que proporcionan apoyo a las siguientes características:

Versiones. Conforme avanza el proyecto, se crearán muchas versiones (sección 22.3.2) de productos resultantes individuales. El repositorio debe guardar todas estas versiones para permitir la administración efectiva de los productos liberados y, a los desarrolladores, regresar a versiones anteriores durante las pruebas y la depuración.

FIGURA 22.3
Contenido del repositorio



El repositorio debe controlar una amplia variedad de tipos de objeto, incluidos texto, gráficos, mapas de bits, documentos complejos y objetos únicos, como definiciones de pantalla y reportes, archivos objeto, datos de prueba y resultados. Un repositorio maduro rastrea versiones de objetos con niveles arbitrarios de granularidad; por ejemplo, puede rastrearse una sola definición de datos o un grupo de módulos.

Rastreo de dependencia y gestión del cambio. El repositorio administra una amplia variedad de relaciones entre los elementos de datos almacenados en él. En éstos se incluyen relaciones entre entidades y procesos empresariales, entre las partes de un diseño de aplicación, entre componentes de diseño y la arquitectura de información de la empresa, entre elementos de diseño y entregables, etcétera. Algunas de estas relaciones son meras asociaciones, y otras son dependencias o relaciones obligatorias.

La capacidad de seguir la pista de todas estas relaciones es vital para la integridad de la información almacenada en el repositorio y para la generación de entregables con base en él, y es una de las aportaciones más importantes del concepto de repositorio para la mejora del proceso de software. Por ejemplo, si un diagrama de clase UML se modifica, el repositorio puede detectar si clases relacionadas, descripciones de interfaz y componentes de código también requieren modificación y si pueden llevar los ICS afectados a la atención del desarrollador.

Rastreo de requerimientos. Esta función especial depende de la administración de vínculos y ofrece la capacidad de rastrear todos los componentes de diseño y construcción, así como entregables que resulten de una especificación de requerimientos determinada (rastreo hacia adelante). Además, proporciona la capacidad de identificar qué requisito genera algún producto de trabajo determinado (rastreo hacia atrás).

Administración de la configuración. Una instalación de administración de la configuración sigue la pista a una serie de configuraciones que representa hitos de proyecto específicos o liberaciones de producción.

Ensayos de auditoría. Un ensayo de auditoría establece información adicional acerca de cuándo, por qué y quién realiza los cambios. La información acerca de la fuente de los cambios puede ingresarse como atributos de objetos específicos en el repositorio. Un mecanismo de activación de repositorio es útil para que siempre que se modifique un elemento de diseño, se avise al desarrollador, o a la herramienta que se utilice, el inicio de la entrada de la información de auditoría (como la razón para un cambio).

22.3 El proceso ACS

Cita:

"Cualquier cambio, incluso para mejorar, está acompañado de inconvenientes e incomodidades."

Arnold Bennett

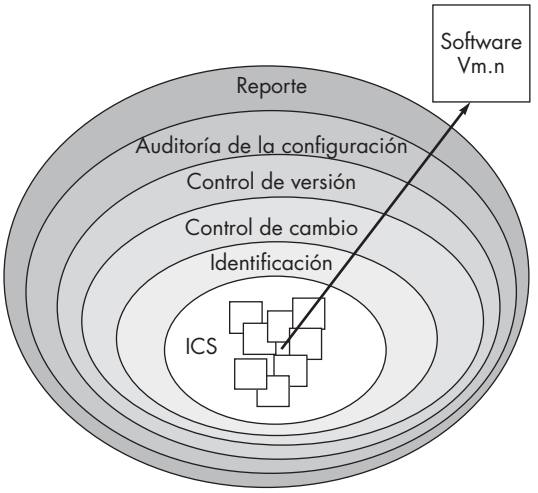
El proceso de administración de la configuración del software define una serie de tareas que tienen cuatro objetivos principales: 1) identificar todos los ítems que de manera colectiva definen la configuración del software, 2) administrar los cambios a uno o más de estos ítems, 3) facilitar la construcción de diferentes versiones de una aplicación y 4) garantizar que la calidad del software se conserva conforme la configuración evoluciona con el tiempo.

Un proceso que logra dichos objetivos no necesita ser burocrático o pesado, pero debe caracterizarse de forma que permita a un equipo de software desarrollar respuestas a un conjunto de preguntas complejas:

¿Qué preguntas debe responder el proceso ACS?

- ¿Cómo identifica un equipo de software los elementos discretos de una configuración de software?
- ¿Cómo gestiona una organización las muchas versiones existentes de un programa (y su documentación) de manera que permita que el cambio se acomode eficientemente?

FIGURA 22.4
Capas del proceso ACS



- ¿Cómo controla una organización los cambios antes y después de que el software se libera a un cliente?
- ¿Quién tiene la responsabilidad de aprobar y clasificar los cambios solicitados?
- ¿Cómo puede garantizarse que los cambios se realizaron adecuadamente?
- ¿Qué mecanismo se usa para enterar a otros acerca de los cambios que se realizaron?

Estas preguntas conducen a la definición de las cinco tareas ACS (identificación, control de versión, control de cambio, auditoría de la configuración y reporte) que se ilustran en la figura 22.4.

En la figura, las tareas ACS pueden visualizarse como capas concéntricas. Los ICS fluyen hacia afuera a través de estas capas a lo largo de su vida útil, y a final de cuentas se vuelven parte de la configuración del software de una o más versiones de una aplicación o sistema. Conforme un ICS se mueve a través de una capa, las acciones que implica cada tarea ACS pueden o no ser aplicables. Por ejemplo, cuando se crea un nuevo ICS, debe identificársele. Sin embargo, si no se solicitan cambios para el ICS, la capa de control de cambio no se aplica. El ICS se asigna a una versión específica del software (entran en juego los mecanismos de control de versión). Un registro del ICS (su nombre, fecha de creación, designación de la versión, etcétera) se conserva con propósitos de auditoría de la configuración y se reporta a quienes tienen necesidad de conocerlo. En las secciones siguientes se examina con más detalle cada una de estas capas del proceso ACS.

22.3.1 Identificación de objetos en la configuración del software

Para controlar y administrar ítems de configuración del software, cada uno debe nombrarse por separado y luego organizarse usando un enfoque orientado a objetos. Es posible identificar dos tipos de objetos [Cho89]: básicos y agregados.[3] Un *objeto básico* es una unidad de información que se crea durante el análisis, el diseño, el código o la prueba. Por ejemplo, un objeto básico puede ser una sección de una especificación de requerimientos, parte de un modelo de diseño, código fuente para un componente o una suite de casos de prueba que se utilice para ejercitar el código. Un *objeto agregado* es una colección de objetos básicos y de otros objetos agregados.

3 El concepto de objeto agregado [Gus89] se ha propuesto como un mecanismo para representar una versión completa de una configuración de software.

Por ejemplo, un **DesignSpecification** es un objeto agregado. Conceptualmente, puede vérsele como una lista nominada (identificada) de punteros que especifican objetos agregados, como **ArchitecturalModel** y **DataModel**, y *objetos básicos*, como **ComponentN** y **UMLClassDiagramN**.

Las interrelaciones establecidas por los objetos de configuración le permiten valorar el impacto del cambio.

Cada objeto tiene un conjunto de características distintivas que lo identifican de manera única: un nombre, una descripción, una lista de recursos y una "realización". El nombre del objeto es una cadena de caracteres que identifica el objeto sin ambigüedades. La descripción del objeto es una lista de ítems de datos que identifican el tipo ICS (por ejemplo, elemento de modelo, programa, datos) representado por el objeto, un identificador de proyecto e información de cambio y/o versión. Los recursos son "entidades que se proporcionan, procesan, referencian o que de algún modo el objeto las requiere" [Cho89]. Por ejemplo, los tipos de datos, las funciones específicas o incluso los nombres de variable pueden considerarse como recursos del objeto. La realización es un puntero hacia la "unidad de texto" para un objeto básico, y nulo para un objeto agregado.

La identificación del objeto de configuración también puede considerar las relaciones que existen entre los objetos nominados. Por ejemplo, al usar la notación simple

```
Diagrama clase <parte de> modelo requerimientos;
Modelo requerimientos <parte de> especificación requerimientos;
```

se puede crear una jerarquía de ICS.

Incluso si la base de datos del proyecto proporciona la capacidad para establecer dichas relaciones, éstas consumen mucho tiempo para su establecimiento y son difíciles de mantener actualizadas. Aunque son muy útiles para el análisis de impacto, no son esenciales para la administración de cambio global.

En muchos casos, los objetos se interrelacionan a través de ramas de la jerarquía de objetos. Estas relaciones transestructurales pueden representarse en la forma siguiente:

```
ModeloDatos <interrelacionado> ModeloFlujoDatos
ModeloDatos <interrelacionado> CasoPruebaClaseM
```

En el primer caso, la interrelación se efectúa entre un objeto compuesto, mientras que la segunda relación es entre un objeto agregado (**ModeloDatos**) y un objeto básico (**CasoPruebaClaseM**).

El esquema de identificación para objetos de software debe reconocer que los objetos evolucionan a lo largo del proceso de software. Antes de que un objeto se convierta en línea de referencia, puede cambiar muchas veces; incluso, después de establecer una línea de referencia, los cambios pueden ser bastante frecuentes.

22.3.2 Control de versión

El control de versión combina procedimientos y herramientas para administrar diferentes versiones de objetos de configuración que se crean durante el proceso de software. Un sistema de control de versión implementa o se integra directamente con cuatro grandes capacidades: 1) una base de datos de proyecto (repositorio) que almacena todos los objetos de configuración relevantes, 2) una capacidad de *administración de versión* que almacena todas las versiones de un objeto de configuración (o que permite la construcción de cualquier versión usando diferencias de las versiones pasadas) y 3) una *facilidad para elaboración* que le permite recopilar todos los objetos de configuración relevantes y construir una versión específica del software. Además, los sistemas de control de versión y de control de cambio con frecuencia implementan una capacidad de *rastreador de conflictos* (también llamado *rastreador de errores*) que permite al equipo registrar y rastrear el estado de todos los conflictos sobresalientes asociados con cada objeto de configuración.

Algunos sistemas de control de versión establecen un *conjunto de cambio*, una colección de todos los cambios (en relación con cierta configuración de referencia) que se requieren para crear una versión específica del software. Dart [Dar91] observa que un conjunto de cambio

"captura todos los cambios habidos en todos los archivos que hay en la configuración, junto con la razón de los cambios y detalles de quién y cuándo hizo los cambios".

Algunos conjuntos de cambio nominados pueden identificarse para una aplicación o sistema. Esto permite construir una versión del software al especificar los conjuntos de cambios (por nombre) que deben aplicarse a la configuración de referencia. Para lograr esto, se aplica un enfoque de *modelado de sistema*. El modelo del sistema contiene: 1) una *plantilla* que incluye una jerarquía de componente y un "orden de construcción" para los componentes que describen cómo debe construirse el sistema, 2) reglas de construcción y 3) reglas de verificación.[4]

Durante las décadas pasadas se propusieron algunos enfoques automatizados diferentes para el control de versión. La diferencia principal en los enfoques es la sofisticación de los atributos que se usan para construir versiones específicas y variantes de un sistema y la mecánica del proceso para su construcción.

HERRAMIENTAS DE SOFTWARE

El Sistema de Versiones Concurrentes (SVC)

El uso de herramientas para lograr el control de versión es esencial para la administración efectiva del cambio. El *sistema de versiones concurrentes* (SVC) es una herramienta ampliamente utilizada para el control de versiones. Originalmente diseñada para código fuente, pero útil para cualquier archivo basado en texto, el SVC 1) establece un repositorio simple, 2) mantiene todas las versiones de un archivo en un solo archivo nominado al almacenar sólo las diferencias entre versiones progresivas del archivo original y 3) protege a un archivo contra los cambios simultáneos al establecer diferentes directorios para cada desarrollador, lo que, por tanto, aísla uno de otros. El SVC mezcla los cambios cuando cada desarrollador completa su trabajo.

Es importante observar que el SVC no es un sistema "de construcción", es decir, no construye una versión específica del software. Deben integrarse otras herramientas (por ejemplo, *Makefile*) con el SVC para lograr esto. El SVC no implanta un proceso de control de cambio (por ejemplo, solicitudes de cambio, reportes de cambio, rastreo de errores).

Incluso con estas limitaciones, el SVC "es un dominante sistema de control de versión, de red transparente y código abierto, [que] es útil para todos, desde desarrolladores individuales hasta grandes equipos distribuidos" [CVS07]. Su arquitectura cliente-servidor permite a los usuarios acceder a los archivos mediante conexiones de internet, y su filosofía de código abierto lo vuelve disponible para la mayoría de las plataformas más usadas.

SVC está disponible sin costo alguno para entornos Windows, Mac OS, LINUX y UNIX. Vea [CVS07] para más detalles.

22.3.3 Control de cambio

Cita:

"El arte de avanzar es preservar el orden en medio del cambio y preservar el cambio en medio del orden."

Alfred North Whitehead

La realidad del control de cambio en un contexto moderno de ingeniería de software la resume bellamente James Bach [Bac98]:

> El control del cambio es vital. Pero las fuerzas que lo hacen necesario también lo hacen desconcertante. Nos preocupamos por el cambio porque una pequeña perturbación en el código puede crear una gran falla en el producto. Pero también puede corregir un gran fallo o permitir maravillosas nuevas capacidades. Nos preocupamos por el cambio porque un solo desarrollador granuja podría hundir el proyecto, aunque en las mentes de dichos granujas se originan ideas brillantes y un abrumador proceso de control del cambio podría efectivamente desalentarlos de hacer trabajo creativo.

Bach reconoce que se está frente a un acto de equilibrio. Mucho control del cambio y se crearán problemas. Muy poco y se crearán otros problemas.

Para un gran proyecto de software, el cambio descontrolado conduce rápidamente al caos. Para tales proyectos, el control del cambio combina procedimientos humanos y herramientas

4 También es posible consultar el modelo del sistema para valorar cómo impactará un cambio en un componente a otros componentes.

FIGURA 22.5
El proceso de control del cambio

automatizadas a fin de proporcionar un mecanismo para el control del cambio. El proceso de control del cambio se ilustra de manera esquemática en la figura 22.5. Una *petición de cambio* se envía y evalúa para valorar el mérito técnico, los potenciales efectos colaterales, el impacto global sobre otros objetos de configuración y funciones del sistema, y el costo proyectado del cambio. Los resultados de la evaluación se presentan como un *reporte de cambio*, que utiliza una *autoridad de control del cambio* (ACC), es decir, una persona o un grupo que toma una decisión final acerca del estatus y la prioridad del cambio. Por cada cambio aprobado se genera una *orden de cambio de ingeniería* (OCI). La OCI describe el cambio que se va a realizar, las restricciones que deben respetarse y los criterios para revisar y auditar.

PUNTO CLAVE
Cabe señalar que un número de peticiones de cambios se pueden combinar para resultar en una OCI única y esas OCI típicamente provocan cambios en los objetos de configuración múltiple.

El objeto que se va a cambiar puede colocarse en un directorio que controlan exclusivamente los ingenieros de software que realizan el cambio. Un sistema de control de versión (vea la barra lateral SVC) actualiza el archivo original una vez que se realiza el cambio. Como alternativa, el objeto que se va a cambiar puede "sacarse" de la base de datos del proyecto (repositorio), realizarse el cambio y aplicarse las actividades adecuadas de SQA. Luego, el objeto "entra" a la base de datos y se usan mecanismos de control de versión adecuados (sección 22.3.2) para crear la siguiente versión del software.

Dichos mecanismos de control de versión, integrados dentro del proceso de control de cambio, implementan dos importantes elementos de la gestión del cambio: control del acceso y control de la sincronización. El *control del acceso* determina qué ingenieros de software tienen la autoridad para acceder y modificar un objeto de configuración particular. El *control de la sincronización* ayuda a garantizar que cambios paralelos, realizados por dos personas diferentes, no se sobreescriban mutuamente.

Acaso se sienta incómodo con el nivel de burocracia que implica la descripción del proceso de control de cambio que se muestra en la figura 22.5. Este sentimiento no es raro. Sin salvaguardas adecuadas, el control del cambio puede retardar el progreso y crear burocracia innecesaria. La mayoría de los desarrolladores de software que tienen mecanismos de control del cambio (por desgracia, muchos no los tienen) han creado algunas capas de control para auxiliarse y evitar los problemas que se mencionan aquí.

Opte por un poco más de control de cambio del que crea que necesitará. Es probable que demasiado sea la cantidad correcta.

Antes de que un ICS se convierta en referencia, sólo es necesario aplicar *control de cambio informal*. El desarrollador del objeto de configuración (ICS) en cuestión puede hacer cualquier cambio que sea justificado por el proyecto y por los requerimientos técnicos, en tanto los cambios no afecten requerimientos más amplios del sistema que se encuentren afuera del ámbito de trabajo del desarrollador. Una vez que el objeto experimenta revisión técnica y se aprueba, puede crearse una línea de referencia.[5] Cuando el ICS se convierte en referencia, se implementa un *control de cambio en el nivel del proyecto*. Entonces, para hacer un cambio, el desarrollador debe obtener la aprobación del gerente del proyecto (si el cambio es "local") o del ACC (si el cambio afecta a otros ICS). En algunos casos, la generación formal de peticiones de cambio, reportes de cambio y OCI se otorgan en conjunto. Sin embargo, se realiza la valoración de cada cambio y todos los cambios se rastrean y revisan.

Cuando el producto de software se libera a los clientes, se instituye el *control de cambio formal*. El procedimiento de control de cambio formal se delineó en la figura 22.5.

Cita:

"El cambio es inevitable, excepto para las máquinas expendedoras."

Letrero en un parachoques

La autoridad de control de cambio juega un papel activo en la segunda y tercera capas del control. Dependiendo del tamaño y carácter de un proyecto de software, la ACC puede componerse de una persona (el gerente de proyecto) o de algunas personas (por ejemplo, representantes de software, hardware, ingeniería de base de datos, apoyo, mercadotecnia). El papel de la ACC es adoptar una visión global, es decir, valorar el impacto del cambio más allá del ICS en cuestión. ¿Cómo afectará el cambio al hardware? ¿Cómo lo hará en el desempeño? ¿Cómo modificará la percepción de los clientes acerca del producto? ¿Cómo afectará la calidad y confiabilidad del producto? La ACC debe abordar éstas y muchas otras preguntas.

CASASEGURA

Conflictos ACS

La escena: Oficina de Doug Miller en el momento de comenzar el proyecto de software *CasaSegura*.

Participantes: Doug Miller (gerente del equipo de ingeniería de software *CasaSegura*), Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería de software del producto.

La conversación:

Doug: Ya sé que es temprano, pero tenemos que hablar acerca de la gestión del cambio.

Vinod (ríe): Difícilmente. Mercadotecnia llamó esta mañana con algunas "segundas opiniones". Nada importante, pero sólo es el comienzo.

5 Una línea de referencia también puede crearse por otras razones. Por ejemplo, cuando se crean "construcciones diarias", todos los componentes verificados en un momento determinado se convierten en la línea de referencia para el trabajo del día siguiente.

Jamie: En proyectos anteriores fuimos bastante informales acerca de la administración del cambio.

Doug: Lo sé, pero éste es más grande y más visible y, según recuerdo...

Vinod (cabecea): Nos matamos con cambios incontrolables en el proyecto de control de iluminación de la casa... recuerdo las demoras que...

Doug (frunce el ceño): Una pesadilla que prefiero no revivir.

Jamie: Así que, ¿qué hacemos?

Doug: Según veo, tres cosas. Primero, tenemos que desarrollar, o pedir prestado, un proceso de control de cambio.

Jamie: Quieres decir: ¿cómo solicitan las personas los cambios?

Vinod: Sí, pero también cómo se evalúa el cambio, cómo se decide cuándo hacerlo (si eso es lo que se decide) y cómo se conservan registros de lo que afecta el cambio.

Doug: Segundo, debemos conseguir una herramienta ACS realmente buena para el control de cambios y de versión.

Jamie: Podemos construir una base de datos para todos los productos de trabajo.

Vinod: En este contexto se llaman ICS, y la mayoría de las buenas herramientas proporcionan cierto soporte para eso.

Doug: Ése es un buen comienzo, ahora tenemos que...

Jamie: Sí, Doug, dijiste que había tres cosas...

Doug (sonríe): Tercero: todos debemos comprometernos en seguir el proceso de administración del cambio y usar las herramientas, sin importar cuáles sean, ¿está claro?

22.3.4 Auditoría de configuración

La identificación, control de versión y control del cambio ayudan a conservar el orden en lo que de otro modo sería una situación caótica y fluida. Sin embargo, incluso los más exitosos mecanismos de control rastrean un cambio sólo hasta que se genera una OCI. ¿Cómo puede un equipo de software asegurarse de que el cambio se implementó adecuadamente? La respuesta es doble: 1) revisiones técnicas y 2) auditoría a la configuración del software.

La revisión técnica (capítulo 15) se enfoca en la exactitud técnica del objeto de configuración que se modificó. Los revisores valoran el ICS para determinar la consistencia con otros ICS, así como omisiones o potenciales efectos colaterales. Una revisión técnica debe realizarse para todos los cambios, salvo los más triviales.

Una *auditoría de configuración del software* complementa la revisión técnica al valorar un objeto de configuración acerca de las características que por lo general no se consideran durante la revisión. La auditoría hace y responde las siguientes preguntas:

? ¿Cuáles son las preguntas principales que se plantean durante una auditoría de configuración?

1. ¿Se realizó el cambio especificado en la OCI? ¿Se incorporó alguna modificación adicional?
2. ¿Se llevó a cabo una revisión técnica para valorar la exactitud técnica?
3. ¿Se siguió el proceso del software y se aplicaron adecuadamente los estándares de ingeniería de software?
4. El cambio se "resaltó" en el ICS? ¿Se especificaron la fecha del cambio y el autor del cambio? ¿Los atributos del objeto de configuración reflejan el cambio?
5. ¿Se siguieron los procedimientos ACS para anotar, registrar y reportar el cambio?
6. ¿Los ICS relacionados se actualizaron adecuadamente?

En algunos casos, las preguntas de la auditoría se plantean como parte de una revisión técnica. No obstante, cuando la ACS es una actividad formal, la auditoría de la configuración la realiza por separado el grupo de aseguramiento de la calidad. Tales auditorías formales de la configuración también garantizan que los ICS correctos (por versión) se incorporan en una construcción específica, y que toda la documentación se actualizó y es consistente con la versión que se construyó.

22.3.5 Reporte de estado

El *reporte del estado de la configuración* (en ocasiones llamado *contabilidad de estado*) es una tarea ACS que responde las siguientes preguntas: 1) ¿Qué ocurrió? 2) ¿Quién lo hizo? 3) ¿Cuándo ocurrió? 4) ¿Qué más se afectará?

Desarrolle una lista de "se necesita saber" para cada objeto de configuración y manténgala actualizada. Cuando se realice un cambio, asegúrese de notificar a todos a través de la lista.

El flujo de información para el reporte del estado de la configuración (REC) se ilustra en la figura 22.5. Cada vez que se le asigna a un ICS una nueva identificación o que se le actualiza, se hace una entrada REC. Cada ocasión que el ACC aprueba un cambio (es decir, emite una OCI), se hace una entrada REC. Cada vez que se lleva a cabo una auditoría de la configuración, los resultados se reportan como parte de la tarea REC. La salida del REC puede colocarse en una base de datos en línea o en un sitio web, de modo que los desarrolladores de software o el personal de apoyo puedan acceder a la información del cambio mediante categorías de palabras clave. Además, regularmente se genera un reporte REC y se tiene la intención de mantener al tanto de los cambios importantes a los gerentes y profesionales.

HERRAMIENTAS DE SOFTWARE

Apoyo a ACS

Objetivo: Las herramientas ACS proporcionan apoyo a una o más de las actividades del proceso que se estudian en la sección 22.3.

Mecánica: La mayoría de las modernas herramientas ACS trabajan en conjunto con un repositorio (un sistema de base de datos) y ofrecen mecanismos para identificación, control de versión y de cambio, auditoría y reportes.

Herramientas representativas:[6]

CCC/Harvest, distribuida por Computer Associates (**www.cai.com**), es un sistema ACS multiplataforma.

ClearCase, desarrollada por Rational, proporciona una familia de funciones ACS (**www-306.ibm.com/software/awdtools/clearcase/index.html**).

Serena ChangeMan ZMF, distribuida por Serena (**www.serena.com/US/products/zmf/index.aspx**), proporciona un juego completo de herramientas ACS que son aplicables tanto para software convencional como para *webapps*.

SourceForge, distribuida por VA Software (**sourceforge.net**), ofrece administración de versión, capacidades de construcción, rastreo de conflictos/errores y muchas otras características de administración.

SurroundSCM, desarrollada por Seapine Software, proporciona capacidades completas de administración del cambio (**www.seapine.com**).

Vesta, distribuida por Compac, es un sistema ACS de dominio público que puede soportar tanto proyectos pequeños (< 10 KLOC) como grandes (10 000 KLOC) (**www.vestasys.org**).

Una gran lista de herramientas y entornos ACS comerciales puede encontrarse en **www.cmtoday.com/yp/commercial.html**.

22.4 ADMINISTRACIÓN DE LA CONFIGURACIÓN PARA *WEBAPPS*

Anteriormente, en este libro, se estudió la naturaleza especial de las *webapps* y los métodos especializados (llamados métodos de *ingeniería web*[7]) que se requieren para construirlas. Entre las muchas características que diferencian a las *webapps* del software tradicional se encuentra la naturaleza siempre presente del cambio.

¿Qué impacto tiene sobre una *webapp* el cambio descontrolado?

Los desarrolladores de *webapps* con frecuencia usan un modelo de proceso iterativo incremental que aplica muchos principios derivados del desarrollo de software ágil (capítulo 3). Al usar este método, un equipo de ingeniería con frecuencia desarrolla un incremento de *webapp* en un periodo muy corto, usando un enfoque impulsado por el cliente. Los incrementos posteriores agregan contenido y funcionalidad adicionales, y es probable que cada uno implemente

6 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

7 Vea [Pre08] para una discusión amplia de los métodos de ingeniería web.

cambios que conduzcan a aumento de contenido, mejor usabilidad, estética mejorada, mejor navegación, desempeño aumentado y seguridad más fuerte. Por tanto, en el mundo ágil de las *webapps*, el cambio se ve de manera un poco diferente.

Si usted es miembro de un equipo *webapp*, debe abrazar el cambio. Más aún, un equipo ágil típico se abstiene de todas las cosas que parecen ser pesadas, burocráticas y formales. La administración de la configuración del software con frecuencia se ve (aunque incorrectamente) como poseedora de estas características. Esta aparente contradicción se remedia no al rechazar los principios, prácticas y herramientas ACS, sino, más bien, al modelarlos para satisfacer las necesidades especiales de los proyectos *webapp*.

22.4.1 Conflictos dominantes

Conforme las *webapps* se vuelven cada vez más importantes para la supervivencia y el crecimiento empresarial, crece la necesidad de la administración de la configuración. ¿Por qué? Porque sin controles efectivos, los cambios inapropiados para una *webapp* (recuerde que la inmediatez y la evolución continua son los atributos dominantes de muchas *webapps*) pueden conducir a: publicación no autorizada de información de productos nuevos, funcionalidad errónea o pobremente probada que frustra a los visitantes de un sitio web, huecos en la seguridad que ponen en peligro los sistemas internos de la compañía y otras consecuencias económicamente desagradables o incluso desastrosas.

Las estrategias generales para la administración de la configuración del software (ACS) descritas en este capítulo son aplicables, pero las tácticas y las herramientas deben adaptarse para conformarse a la naturaleza única de las *webapps*. Cuando se desarrollan tácticas para la administración de la configuración de una *webapp*, deben considerarse cuatro conflictos [Dar99].

Contenido. Una *webapp* típica contiene un arreglo muy amplio de contenido: texto, gráficos, applets, guiones, archivos audio/video, elementos de página activos, tablas, transmisión de datos, y muchos otros. El reto es organizar este mar de contenido en un conjunto racional de objetos de configuración (sección 22.1.4) y luego establecer mecanismos de control de la configuración adecuados para dichos objetos. Un camino es modelar el contenido de la *webapp* usando técnicas de modelado de datos convencionales (capítulo 6), unidas a un conjunto de propiedades especializadas de cada objeto. La naturaleza estática/dinámica de cada objeto y su longevidad proyectada (objeto temporal, de existencia fija o permanente) son ejemplos de propiedades que se requieren para establecer un enfoque ACS efectivo. Por ejemplo, si un ítem de contenido cambia cada hora, tiene longevidad temporal. Los mecanismos de control para este ítem serán diferentes (menos formales) a los aplicados para un componente de formulario, que es un objeto permanente.

Personas. Puesto que un porcentaje significativo de desarrollo *webapp* continúa realizándose en una forma *ad hoc*, cualquier persona involucrada en la *webapp* puede (y con frecuencia lo hace) crear contenido. Muchos creadores de contenido no tienen antecedentes de ingeniería de software y son completamente ajenos a las necesidades de la administración de la configuración. Como consecuencia, la aplicación crece y cambia en forma descontrolada.

Escalabilidad. Las técnicas y los controles aplicados a una *webapp* pequeña no escalan bien hacia arriba. No es raro que una *webapp* simple crezca significativamente conforme se implementan interconexiones con sistemas de información, bases de datos, almacenes de datos y puertas a portales existentes. Conforme crecen el tamaño y la complejidad, pequeños cambios pueden tener efectos de largo alcance y no intencionados que pueden ser problemáticos. Por tanto, el rigor de los mecanismos de control de la configuración debe ser directamente proporcional a la escala de aplicación.

Políticas. ¿Quién "posee" la *webapp* ? Esta pregunta se plantea en compañías grandes y pequeñas, y su respuesta tiene un impacto significativo sobre las actividades de administración y control. En algunas instancias, los desarrolladores web se alojan fuera del área de IT de la organización y crean potenciales dificultades de comunicación. Dart [Dar99] sugiere las siguientes preguntas para ayudar a entender las políticas asociadas con la ingeniería web:

¿Cómo se determina quién tiene la responsabilidad de la AC de la *webapp*?

- ¿Quién asume la responsabilidad por la precisión de la información en el sitio web?
- ¿Quién garantiza que los procesos de control de calidad se siguieron antes de que la información se publique en el sitio?
- ¿Quién es responsable por la realización de cambios?
- ¿Quién asume el costo del cambio?

Las respuestas a estas preguntas ayudan a determinar a las personas que dentro de una organización deben adoptar un proceso de administración de la configuración para *webapps*.

La administración de la configuración para *webapps* continúa evolucionando (por ejemplo, [Ngu06]). Un proceso ACS convencional puede ser demasiado engorroso, pero en años pasados surgió una nueva generación de *herramientas de administración de contenido*, específicamente diseñadas para ingeniería web. Dichas herramientas establecen un proceso que adquiere información existente (de un amplio arreglo de objetos de *webapps*), gestiona los cambios a los objetos, los estructura en forma tal que es posible presentarlos a un usuario final y luego los ofrece al entorno del lado cliente para su despliegue.

22.4.2 Objetos de configuración de *webapps*

Las *webapps* abarcan un amplio rango de objetos de configuración: objetos de contenido (por ejemplo, texto, gráficos, imágenes, video, audio), componentes funcionales (guiones, applets) y objetos de interfaz (COM o CORBA). Los objetos de la *webapp* pueden identificarse (con nombres de archivo asignados) en cualquier forma que sea adecuada para la organización. Sin embargo, se recomiendan las siguientes convenciones para asegurar la conservación de la compatibilidad entre plataformas: los nombres de archivo deben limitarse a 32 caracteres de longitud, deben evitarse nombres con mayúsculas mezcladas o todos con mayúsculas, así como subrayar los nombres de los archivos. Además, las referencias URL (vínculos) dentro de un objeto de configuración siempre deben usar rutas relativas (por ejemplo,../products/alarmsensors.html).

Todo el contenido de la *webapp* tiene formato y estructura. Los formatos de archivo interno los dicta el entorno de computación en el que se almacena el contenido. Sin embargo, el *formato renderizado* (con frecuencia llamado *formato de despliegue*) se define mediante el estilo estético y las reglas de diseño establecidas para la *webapp*. La *estructura del contenido* define una arquitectura de contenido, es decir, la forma en la que se ensamblan los objetos de contenido para presentar información significativa a un usuario final. Boiko [Boi04] define la estructura como "mapas que se tienden sobre un conjunto de trozos de contenido [objetos] para organizarlos y hacerlos accesibles a las personas que los necesitan".

22.4.3 Administración de contenido

Cita:

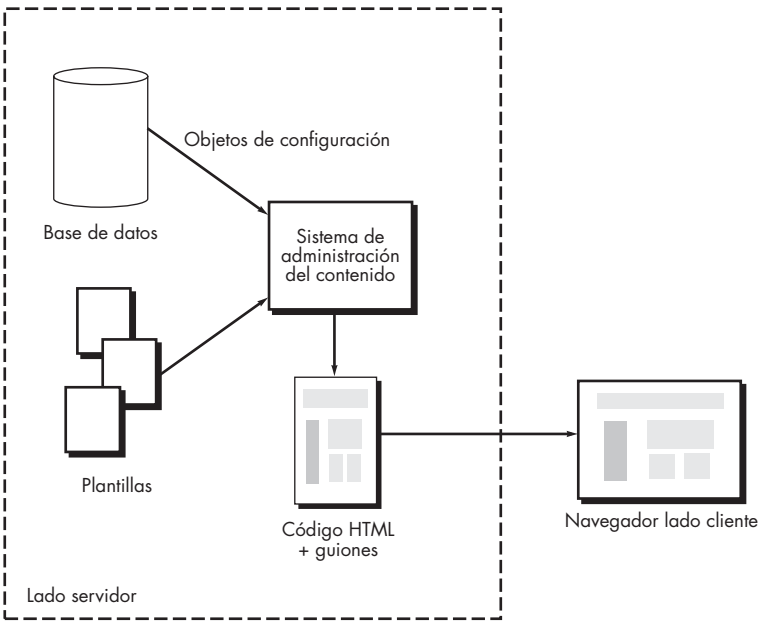
"La administración del contenido es un antídoto al frenesí informativo de la actualidad."

Bob Boiko

La *administración de contenido* se relaciona con la administración de la configuración en tanto un sistema de administración de contenido (SAC) establece un proceso, apoyado por herramientas adecuadas, en el que éstas adquieren contenido existente (de un amplio arreglo de objetos de configuración de la *webapp*), que los estructura en forma que les permite presentarse a un usuario final y luego ofrecerlo al entorno del lado cliente para su despliegue.

El uso más común de un sistema de administración de contenido ocurre cuando se construye una *webapp* dinámica. Las *webapps* dinámicas crean páginas web "al vuelo", es decir, el usuario

FIGURA 22.6
Sistema de administración del contenido



por lo general consulta la *webapp* para requerir información específica. La *webapp* consulta una base de datos, formatea la información en concordancia y la presenta al usuario. Por ejemplo, una compañía musical ofrece una librería de discos compactos para su venta. Cuando un usuario solicita un disco compacto o su equivalente en música electrónica, se consulta una base de datos y se descarga información variada acerca del artista, el disco compacto (por ejemplo, su portada o gráficos), el contenido musical y audio de muestra y se le configura en una plantilla de contenido estándar. La página web resultante se construye en el lado servidor y pasa al navegador en el lado cliente para su examen por parte del usuario final. En la figura 22.6 se muestra una representación genérica de esto.

En el sentido más general, un SAC "configura" el contenido para el usuario final al invocar tres subsistemas integrados: un subsistema de recopilación, un subsistema de gestión y un sistema de publicación [Boi04].

El subsistema de recopilación. El contenido se deriva de datos e información que debe crearse o adquirirse por un desarrollador de contenido. El *subsistema de recopilación* abarca todas las acciones requeridas para crear y/o adquirir contenido y las funciones técnicas que se necesitan para 1) convertir el contenido en una forma que pueda representarse mediante un lenguaje de marcaje (por ejemplo, HTML, XML) y 2) organizar el contenido en paquetes que puedan desplegarse efectivamente en el lado cliente.

PUNTO CLAVE
El subsistema de recopilación abarca todas las acciones requeridas para crear, adquirir y/o convertir el contenido en una forma que pueda presentarse en el lado cliente.

La creación y adquisición de contenido (con frecuencia llamado *autoría*) usualmente ocurre en paralelo con otras actividades de desarrollo de *webapps* y con frecuencia la realizan desarrolladores de contenido no técnicos. Esta actividad combina elementos de creatividad e investigación y se apoya con herramientas que permiten al autor del contenido caracterizar a éste en forma que puede estandarizarse para su uso dentro de la *webapp*.

Una vez que existe el contenido, debe convertirse para adecuarse a los requerimientos de un SAC. Esto implica desnudar el contenido bruto de cualquier información innecesaria (por ejemplo, representaciones gráficas redundantes), formatear el contenido para adecuarse a los requerimientos del SAC y mapear los resultados en una estructura de información que le permitirá manejarse y publicarse.

PUNTO CLAVE
El subsistema de administración implementa un repositorio para todo el contenido. La administración de la configuración se realiza dentro de este subsistema.

El subsistema de administración. Una vez que existe el contenido, debe almacenarse en un repositorio, catalogarse para adquisición y uso posterior y etiquetarse para definir: 1) el estado actual (por ejemplo, ¿el objeto de contenido está completo o en desarrollo?), 2) la versión adecuada del objeto de contenido y 3) los objetos de contenido relacionados. Por tanto, el *subsistema de administración* implementa un repositorio que abarca los siguientes elementos:

- *Base de datos de contenido:* la estructura de información que se establece para almacenar todos los objetos de contenido
- *Capacidades de la base de datos:* funciones que permiten al SAC buscar objetos de contenido específicos (o categorías de objetos), almacenar y recuperar objetos, y administrar la estructura de archivos que se establece para el contenido
- *Funciones de administración de configuración:* los elementos funcionales y el flujo de trabajo asociado que apoyan la identificación del objeto de contenido, el control de la versión, la administración del cambio, la auditoría del cambio y los reportes.

Además de estos elementos, el subsistema de administración implementa una función de administración que abarca los metadatos y las reglas que controlan la estructura global del contenido y la forma en la que se soporta.

PUNTO CLAVE
El subsistema de publicación extrae el contenido del repositorio y lo entrega a navegadores en el lado cliente.

El subsistema de publicación. El contenido debe extraerse del repositorio, convertirse a una forma que sea manejable para su publicación y formatearse de modo que pueda transmitirse a navegadores en el lado cliente. El subsistema de publicación logra estas tareas usando una serie de plantillas. Cada *plantilla* es una función que construye una publicación usando uno de tres componentes diferentes [Boi04]:

- *Elementos estáticos:* texto, gráficos, medios audiovisuales y guiones que no requieren más procesamiento se transmiten directamente al lado cliente.
- *Servicios de publicación:* la función invoca servicios de recuperación y formateo específicos que personalizan el contenido (usando reglas predefinidas), realizan conversión de datos y construyen vínculos de navegación adecuados.
- *Servicios externos:* acceso a infraestructura de información corporativa externa, como datos empresariales o aplicaciones de "cuarto trasero".

Un sistema de administración del contenido que abarque cada uno de estos subsistemas es aplicable para grandes proyectos web. Sin embargo, la filosofía y funcionalidad básicas asociadas con un SAC son aplicables a todas las *webapps* dinámicas.

HERRAMIENTAS DE SOFTWARE

Administración de contenido

Objetivo: Auxiliar a los ingenieros de software y desarrolladores de contenido en la administración del contenido que se incorpora en las *webapps*.

Mecánica: Las herramientas en esta categoría permiten a los ingenieros web y proveedores de contenido actualizar el contenido *webapp* en forma controlada. La mayoría establece un sistema de gestión de archivos simple que asigna permisos de actualización y edición página por página para varios tipos de contenido *webapp*. Algunos mantienen un sistema de versiones, de modo que una versión previa de contenido pueda archivarse con propósitos históricos.

Herramientas representativas:[8]
Vignette Content Management, desarrollado por Vignette (**www.vignette.com/us/Products**), es una suite de herramientas de administración de contenido empresarial.

8 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

ektron-CMS300, desarrollada por ektron (**www.ektron.com**), es una suite de herramientas que proporciona capacidades de administración de contenido y herramientas de desarrollo web.
OmniUpdate, desarrollada por WebsiteASP, Inc. (**www.omniupdate.com**), es una herramienta que permite a proveedores de contenido autorizados desarrollar actualizaciones controladas a contenido web específico.

En los siguientes sitios web puede encontrarse información adicional acerca de ACS y herramientas de administración de contenido para ingeniería web: *Web Developer's Virtual Encyclopedia* (**www.wdlv.com**), *WebDeveloper* (**www.webdeveloper.com**), *Developer Shed* (**www.devshed.com**), *webknowhow.net* (**www.webknowhow.net**), o *WebReference* (**www.webreference.com**).

22.4.4 Administración del cambio

El flujo de trabajo asociado con el control del cambio para software convencional (sección 22.3.3) por lo general es muy pesado para el desarrollo de *webapps*. Es improbable que la secuencia petición de cambio, reporte de cambio y orden de cambio de ingeniería pueda lograrse en forma ágil y aceptable para la mayoría de los proyectos de desarrollo web. Entonces, ¿cómo se gestiona un torrente continuo de cambios solicitados por el contenido y la funcionalidad de la *webapp*?

Para implementar administración de cambio efectiva dentro de la filosofía "codifica y ve" que continúa dominando el desarrollo web debe modificarse el proceso de control de cambio convencional. Cada cambio debe categorizarse en una de cuatro clases:

Clase 1: un cambio de contenido o función que corrige un error o aumenta el contenido o funcionalidad locales

Clase 2: un cambio de contenido o función que tiene un impacto sobre otros objetos de contenido o componentes funcionales

Clase 3: un cambio de contenido o función que tiene un amplio impacto a través de una *webapp* (por ejemplo, extensión de funcionalidad trascendental, significativo aumento o reducción en contenido, grandes cambios requeridos en navegación)

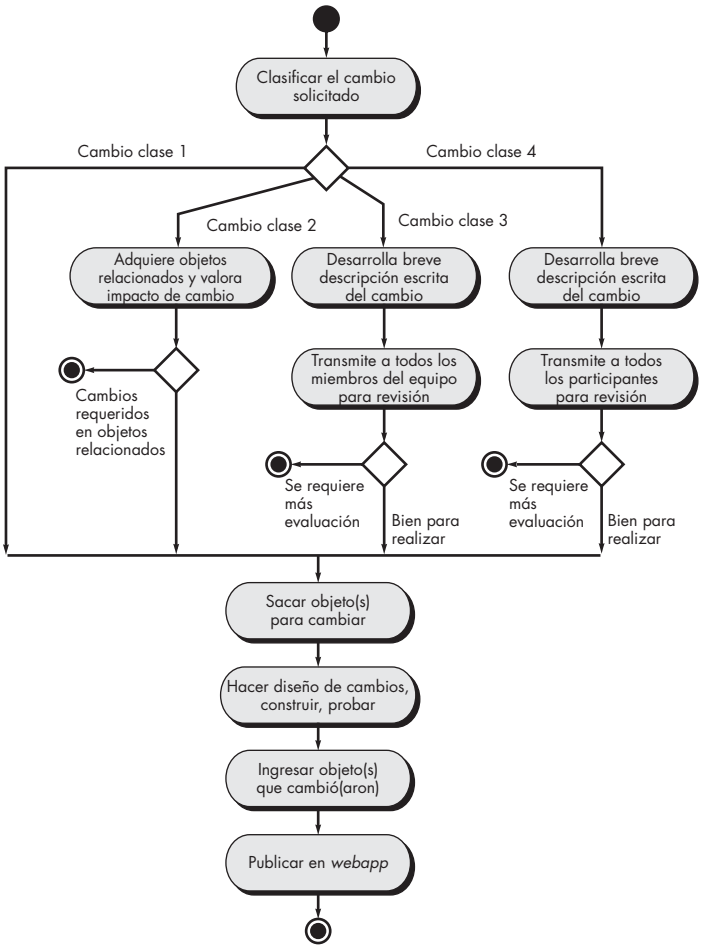
Clase 4: un gran cambio de diseño (por ejemplo, un cambio en diseño de interfaz o enfoque de navegación) que inmediatamente será notable para una o más categorías de usuario

Una vez categorizado el cambio solicitado, puede procesarse en concordancia con el algoritmo que se muestra en la figura 22.7.

En la figura, los cambios en las clases 1 y 2 se tratan de manera informal y se manejan en forma ágil. Para un cambio de clase 1, se evaluaría el impacto del cambio, pero no se requiere revisión o documentación externa. Conforme se realiza el cambio, los procedimientos de entrada y salida estándar se refuerzan mediante herramientas de configuración de repositorio. Para los cambios de clase 2, se revisa el impacto del cambio sobre objetos relacionados (o se pide hacerlo a otros desarrolladores responsables de dichos objetos). Si el el cambio puede hacerse sin requerir cambios significativos a otros objetos, la modificación ocurre sin revisión o documentación adicional. Si se requieren cambios sustantivos, es necesario más evaluación y planificación.

Los cambios de clases 3 y 4 también se tratan en forma ágil, pero se requiere alguna documentación descriptiva y más procedimientos de revisión formal. Para los cambios de clase 3, se desarrolla una *descripción del cambio* que describe el cambio y proporciona una breve valoración del impacto del mismo. La descripción se distribuye a todos los miembros del equipo, quienes revisan el cambio para valorar mejor su impacto. Para los cambios de clase 4, también se desarrolla una descripción del cambio, pero en este caso la realizan todos los participantes.

FIGURA 22.7
Administración de cambios para *webapps*



HERRAMIENTAS DE SOFTWARE

Gestión del cambio

Objetivo: Auxiliar a los ingenieros web y a desarrolladores de contenido a administrar los cambios en objetos de configuración *webapp* conforme estos se realizan.

Mecánica: En esta categoría, las herramientas se desarrollaron originalmente para software convencional, pero pueden adaptarse para su uso por parte de ingenieros web y desarrolladores de contenido a fin de realizar cambios controlados a *webapps*. Soportan entrada y salida automatizadas, control y reconstrucción de versiones, reporte y otras funciones ACS.

Herramientas representativas:[9]

ChangeMan WCM, desarrollada por Serena (**www.serena.com**), es una suite de herramientas de administración del cambio que proporcionan capacidades ACS completas.

ClearCase, desarrollada por Rational (**www-306.ibm.com/software/rational/sw-atoz/indexC.html**), es una suite de herramientas que proporciona todas las capacidades de administración de configuración para *webapps*.

Source Integrity, desarrollada por mks (**www.mks.com**), es una herramienta ACS que puede integrarse con entornos de desarrollo seleccionados.

9 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

22.4.5 Control de versión

Conforme una *webapp* evoluciona a través de una serie de incrementos, pueden existir al mismo tiempo algunas versiones diferentes. Una versión (la *webapp* operativa actual) está disponible mediante internet para usuarios finales; otra (el siguiente incremento de *webapp*) puede estar en las etapas finales de prueba antes de su implementación; una tercera versión está en desarrollo y representa una gran actualización en contenido, estética de interfaz y funcionalidad. Los objetos de configuración deben definirse con claridad, de modo que cada uno pueda asociarse con la versión adecuada. Además, deben establecerse mecanismos de control. Dreilinger [Dre99] analiza la importancia del control de versiones (y de cambios) cuando escribe:

> En un sitio *descontrolado*, donde múltiples autores tienen acceso para editar y contribuir, surge el potencial para conflictos y problemas, más aún si dichos autores trabajan desde diferentes oficinas en diferentes momentos del día y de la noche. Usted puede pasar el día mejorando el archivo *index.html* para un cliente. Después de que usted realiza sus cambios, otro desarrollador que trabaja en casa después de las horas de oficina, o en otra oficina, puede pasar la noche actualizando su propia versión recientemente revisada del archivo *index.html*, ¡y sobreescribir por completo su trabajo sin que haya forma de recuperarlo!

Es probable que usted haya experimentado una situación similar. Para evitarla, se requiere un proceso de control de versiones.

1. *Debe establecerse un repositorio central para el proyecto web*, que contendrá las versiones actuales de todos los objetos de configuración *webapp* (contenido, componentes funcionales y otros).
2. *Cada ingeniero web crea su propia carpeta de trabajo*, que contiene aquellos objetos que se crean o cambian en cualquier momento.
3. *Los relojes en todas las estaciones de trabajo de los desarrolladores deben estar sincronizados* para evitar conflictos de sobreescritura cuando dos desarrolladores realizan actualizaciones que están muy cercanas en el tiempo.
4. *Conforme se desarrollan nuevos objetos de configuración o se cambian los objetos existentes, deben importarse hacia el repositorio central.* La herramienta de control de versión (vea la discusión de SVC en la barra lateral) gestionará todas las funciones de entrada y salida de las carpetas de trabajo de cada desarrollador web. Cuando se hagan cambios al repositorio, la herramienta también proporcionará actualizaciones automáticas por correo electrónico a todas las partes interesadas.
5. *Conforme los objetos se importen al o se exporten del repositorio, se elabora un mensaje de bitácora automático con marca de tiempo.* Esto proporciona información útil para auditar y puede volverse parte de un esquema de reporte efectivo.

La herramienta de control de versión conserva diferentes versiones de la *webapp* y puede revertir una de ellas a una versión más antigua si se requiere.

22.4.6 Auditoría y reporte

Con la intención de obtener agilidad, las funciones de auditoría y reporte no tienen mucho énfasis en el trabajo de ingeniería web.[10] Sin embargo, no se eliminan por completo. Todos los objetos que entran o salen del repositorio se registran en una bitácora que puede revisarse en

10 Esto empieza a cambiar. Hay un creciente énfasis en la ACS como un elemento de la seguridad de la *webapp* [Sar06]. Al proporcionar un mecanismo para rastrear y reportar todo cambio hecho a cada objeto Web, una herramienta de administración del cambio puede proporcionar valiosa protección contra cambios maliciosos.

cualquier momento. Es posible crear un reporte de bitácora completo de modo que todos los miembros del equipo web tengan una cronología de los cambios hechos en un periodo definido. Además, puede enviarse una notificación automatizada por correo electrónico (dirigida a aquellos desarrolladores y participantes que tengan interés) cada vez que un objeto entre o salga del repositorio.

INFORMACIÓN

Estándares ACS

La siguiente lista de estándares ACS (extraída en parte de www.12207.com) es razonablemente exhaustiva:

Estándares IEEE	**standards.ieee.org/catalog/olis/**
IEEE 828	Software para planes de administración de la configuración
IEEE 1042	Software para administración de la configuración
Estándares ISO	**www.iso.ch/iso/en/ISOOnline.frontpage**
ISO 10007-1995	Administración de calidad, guía para AC
ISO/IEC 12207	Software de tecnología de la información-Procesos de ciclo de vida
ISO/IEC TR 15271	Guía para ISO/IEC 12207
ISO/IEC TR 15846	Ingeniería de software-Proceso de ciclo de vida de software-Orden de software para administración de la configuración
Estándares EIA	**www.eia.org/**
EIA 649	Estándar de consenso nacional para administración de la configuración
EIA CMB4-1A	Definiciones de administración de la configuración para programas de cómputo digitales
EIA CMB4-2	Identificación de configuración para programas de cómputo digitales
EIA CMB4-3	Librerías de software de cómputo
EIA CMB4-4	Control de cambio de configuración para programas de cómputo digitales
EIA CMB6-1C	Orden de referencias de administración de configuración y datos
EIA CMB6-3	Identificación de configuración
EIA CMB6-4	Control de la configuración
EIA CMB6-5	Libro de texto para contabilidad de estado de la configuración
EIA CMB7-1	Intercambio electrónico de datos de administración de la configuración
Estándares militares EUA	**Información de estándares MIL: www-library.itsi.disa.mil**
DoD MIL STD-973	Administración de la configuración
MIL-HDBK-61	Guía para administración de la configuración
Otros estándares:	
DO-178B	Lineamientos para el desarrollo de software de aviación
ESA PSS-05-09	Guía para administración de la configuración del software
AECL CE-1001-STD rev.1	Estándar para ingeniería de software crucial para seguridad
DOE SCM checklist:	http://cio.doe.gov/ITReform/sqse/download/cmcklst.doc
BS-6488	British Std., Administración de la configuración de sistemas basados en computadoras
Best Practice—UK	Oficina de comercio gubernamental: **www.ogc.gov.uk**
CMII	Instituto de mejores prácticas AC: **www.icmhq.com**

Configuration Management Resource Guide (*Guía de recursos de administración de la configuración*) proporciona información complementaria para los interesados en procesos y práctica AC. Está disponible en **www.quality.org/config/cm-guide.html**.

22.5 RESUMEN

La administración de la configuración del software es una actividad sombrilla que se aplica a lo largo del proceso de software. La ACS identifica, controla, audita y reporta las modificaciones que invariablemente ocurren mientras el software se desarrolla y después de que se libera a un cliente. Todos los productos de trabajo creados como fases de la ingeniería de software se vuelven parte de una configuración del software. La configuración se organiza de manera que permite el control ordenado del cambio.

La configuración del software se compone de un conjunto de objetos interrelacionados, también llamados ítems de configuración del software (ICS), que se producen como resultado de alguna actividad de ingeniería de software. Además de documentos, programas y datos, el en-

torno de desarrollo que se usa para crear software también puede colocarse bajo control de la configuración. Todos los ICS se almacenan dentro de un repositorio que implementa un conjunto de mecanismos y estructuras de datos para asegurar la integridad de los datos, proporcionar apoyo de integración para otras herramientas de software (información de apoyo que comparte entre todos los miembros del equipo de software) e implementar funciones de apoyo al control de versiones y de cambios.

Una vez desarrollado y revisado el objeto de configuración, se convierte en línea de referencia. Los cambios a un objeto convertido en línea de referencia dan como resultado la creación de una nueva versión de dicho objeto. La evolución de un programa puede rastrearse al examinar la historia de revisión de todos los objetos de configuración. El control de versiones es el conjunto de procedimientos y herramientas que sirven para administrar el uso de dichos objetos.

El control de cambios es una actividad procedimental que garantiza la calidad y la consistencia conforme se realizan cambios a un objeto de configuración. El proceso de control de cambios comienza con una petición de cambios, conduce a una decisión para hacer o rechazar la petición del cambio y culmina con una actualización controlada del ICS que debe cambiarse.

La auditoría de la configuración es una actividad SQA que ayuda a garantizar que la calidad se conserva conforme se realizan cambios. El reporte de estado proporciona información acerca de cada cambio a quienes necesitan conocerla.

La administración de la configuración para *webapps* es similar en muchos aspectos a la ACS para software convencional. Sin embargo, cada una de las tareas núcleo ACS debe dinamizarse para hacerla tan magra como sea posible y deben implementarse provisiones especiales para la administración del contenido.

Problemas y puntos por evaluar

22.1. ¿Por qué es verdadera la primera ley de la ingeniería de sistemas? Ofrezca ejemplos específicos para cada una de las cuatro razones fundamentales para el cambio.

22.2. ¿Cuáles son los cuatro elementos que existen cuando se implementa un sistema ACS efectivo? Analice cada uno brevemente.

22.3. Explique con sus palabras las razones para las líneas de referencia.

22.4. Suponga que usted es el gerente de un proyecto pequeño. ¿Qué líneas de referencia definiría para el proyecto y cómo las controlaría?

22.5. Diseñe un sistema de base de datos de proyecto (repositorio) que permitiría a un ingeniero del software almacenar, poner referencias cruzadas, rastrear, actualizar y cambiar todos los ítems de configuración de software importantes. ¿Cómo manejaría la base de datos diferentes versiones del mismo programa? ¿El código fuente se manejaría de manera diferente a la documentación? ¿Cómo se prohibiría a dos desarrolladores hacer diferentes cambios al mismo tiempo a un mismo ICS?

22.6. Investigue una herramienta ACS existente y describa cómo implementa control para versiones, variantes y objetos de configuración en general.

22.7. Las relaciones <parte de> e <interrelacionado> representan relaciones simples entre objetos de configuración. Describa cinco relaciones adicionales que puedan ser útiles en el contexto de un repositorio ACS.

22.8. Investigue acerca de una herramienta ACS existente y describa cómo implementa la mecánica de control de versiones. De manera alternativa, lea dos o tres ensayos acerca de ACS y describa las diferentes estructuras de datos y mecanismos de referencia que se usan para el control de versiones.

22.9. Desarrolle una lista de verificación para usar durante las auditorías de configuración.

22.10. ¿Cuál es la diferencia entre una auditoría ACS y una revisión técnica? ¿Su función puede plegarse en una revisión? ¿Cuáles son los pros y los contras?

22.11. Describa brevemente las diferencias entre ACS para software convencional y ACS para *webapps*.

22.12. ¿Qué es la administración del contenido? Use la web para investigar las características de una herramienta de administración del contenido y ofrezca un resumen breve.

Lecturas adicionales y fuentes de información

Entre las propuestas de lecturas sobre ACS más recientes se encuentran Leon (*Software Configuration Management Handbook,* 2a. ed., Artech House Publishers, 2005), Maraia (*The Build Master: Microsoft's Software Configuration Management Best Practices,* Addison-Wesley, 2005), Keyes (*Software Configuration Management,* Auerbach, 2004) y Hass (*Configuration Management Principles and Practice,* Addison-Wesley, 2002). Cada uno de estos libros presenta todo el proceso ACS con detalle sustancial. Maraia (*Software Configuration Management Implementation Roadmap*, Wiley, 2004) ofrece una guía única de "cómo hacer" para quienes deben implementar ACS dentro de una organización. Lyon (*Practical CM*, Raven Publishing, 2003, disponible en **www.configuration.org**) escribió una guía exhaustiva para el profesional AC, que incluye lineamientos pragmáticos para implementar cada aspecto de un sistema de administración de la configuración (se actualiza anualmente). White y Clemm (*Software Configuration Management Strategies and Rational ClearCase*, Addison-Wesley, 2000) presentan ACS dentro del contexto de una de las herramientas ACS más populares.

Berczuk y Appleton (*Software Configuration Management Patterns*, Addison-Wesley, 2002) proponen varios patrones útiles que ayudan a comprender la ACS y a implementar de manera efectiva sistemas ACS. Brown *et al.* (*Anti-Patterns and Patterns in Software configuration Management*, Wiley, 1999) estudian las cosas que no se hacen (antipatrones) cuando se implementa un proceso ACS y luego consideran sus remedios. Bays (*Software Release Methodology*, Prentice Hall, 1999) se enfoca en la mecánica de "liberación exitosa de producto", un importante complemento a la ACS efectiva.

Conforme las *webapps* se vuelven más dinámicas, la administración del contenido se ha convertido en un tema esencial para los ingenieros web. Los libros de White (*The Content Management Handbook,* Curtin University Books, 2005), Jenkins *et al*. (*Enterprise Content Management Methods,* Open Text Corporation, 2005), Boiko [Boi04], Mauthe y Thomas (*Professional Content Management Systems,* Wiley, 2004), Addey *et al*. (*Content Management Systems,* Glasshaus, 2003), Rockley (*Managing Enterprise Content,* New Riders Press, 2002), Hackos (*Content Management for Dynamic Web Delivery,* Wiley, 2002), y Nakano (*Web Content Management,* Addison-Wesley, 2001) presentan tratamientos valiosos del tema.

Además de discusiones genéricas del tema, Lim *et al*. (*Enhancing Microsoft Content Management Server with ASP.NET 2.0,* Packt Publishing, 2006), Ferguson (*Creating Content Management Systems in Java,* Charles River Media, 2006), IBM Redbooks (*IBM Workplace Web Content Management for Portal 5.1 and IBM Workplace Web Content Management 2.5,* Vivante, 2006), Fritz *et al*. (*Typo3: Enterprise Content Management,* Packt Publishing, 2005) y Forta (*Reality ColdFusion: Intranets and Content Management,* Pearson Education, 2002) abordan la administración del contenido dentro del contexto de herramientas y lenguajes específicos.

En internet está disponible una gran variedad de fuentes de información acerca de ingeniería de administración de la configuración del software y de administración del contenido. Una lista actualizada de referencias en la World Wide Web que son relevantes para la administración de la configuración del software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compICS/pressman/professional/olc/ser.htm**.

CAPÍTULO

23 MÉTRICAS DE PRODUCTO

Conceptos clave

diseño de interfaz de usuario 545
diseño *webapp* 545
indicador 527
medición 527
medida 527
Meta/Pregunta/Métrica (MPM) 529
métricas
  atributos de 530
  código fuente 547
  diseño arquitectónico 535
  diseño OO 537
  modelo de requerimientos . 531
  orientado a clase 539
  pruebas 548
principios de medición 528
punto de función (PF) 531

Un elemento clave de cualquier proceso de ingeniería es la medición. Pueden usarse medidas para entender mejor los atributos de los modelos que se crean y para valorar la calidad de los productos o sistemas sometidos a ingeniería que se construyen. Pero, a diferencia de otras disciplinas de la ingeniería, la del software no está asentada en las leyes cuantitativas de la física. Mediciones directas, como voltaje, masa, velocidad o temperatura, son raras en el mundo del software. Puesto que las mediciones y métricas del software con frecuencia son indirectas, están abiertas a debate. Fenton [Fen91] aborda este conflicto cuando afirma:

> Medición es el proceso mediante el cual se asignan números o símbolos a los atributos de las entidades en el mundo real, de manera que se les define de acuerdo con reglas claramente determinadas [...] En ciencias físicas, medicina, economía y más recientemente en ciencias sociales, ahora es posible medir atributos que anteriormente se consideraban inmensurables [...] Desde luego, tales mediciones no son tan refinadas como muchas mediciones en las ciencias físicas [...], pero existen [y con base en ellas se toman decisiones importantes]. Sentimos que la obligación de intentar "medir lo inmensurable" para mejorar la comprensión de entidades particulares es tan poderosa en la ingeniería del software como en cualquiera otra disciplina.

Pero algunos miembros de la comunidad del software continúan argumentando que el software es "inmensurable" o que los intentos por medir deben posponerse hasta comprender mejor el software y los atributos que deben usarse para describirlo. Esto es un error.

Una mirada rápida

¿Qué es? Por su naturaleza, la ingeniería es una disciplina cuantitativa. Las métricas de producto ayudan a los ingenieros de software a obtener comprensión acerca del diseño y la construcción del software que elaboran, al enfocarse en atributos mensurables específicos de los productos de trabajo de la ingeniería del software.

¿Quién lo hace? Los ingenieros de software usan métricas de proyecto para auxiliarse en la construcción de software de mayor calidad.

¿Por qué es importante? Siempre habrá un elemento cualitativo en la creación del software para computadoras. El problema es que la valoración cualitativa tal vez no sea suficiente. Se necesitan criterios objetivos que ayuden a guiar el diseño de datos, arquitectura, interfaces y componentes. Cuando se prueban, es necesaria la guía cuantitativa que ayuda en la selección de los casos de prueba y de sus objetivos. Las métricas de producto proporcionan una base desde donde el análisis, el diseño, la codificación y las pruebas pueden realizarse de manera más objetiva y valorarse de modo más cuantitativo.

¿Cuáles son los pasos? El primer paso en el proceso de medición es derivar las mediciones y métricas del software que sean adecuadas para la presentación del software que se está construyendo. A continuación, se recolectan los datos requeridos para derivar las métricas formuladas. Una vez calculadas, las métricas adecuadas se analizan con base en lineamientos preestablecidos y en datos anteriores. Los resultados del análisis se interpretan para obtener comprensión acerca de la calidad del software, y los resultados de la interpretación conducen a modificación de requerimientos y modelos de diseño, código fuente o casos de prueba. En algunas instancias, también puede conducir a modificación del proceso de software en sí.

¿Cuál es el producto final? Las métricas de producto que se calculan a partir de los datos recolectados de los modelos de requerimientos y de diseño, código fuente y casos de prueba.

¿Cómo me aseguro de que lo hice bien? Debe establecer los objetivos de la medición antes de comenzar la recolección de datos y debe definir cada métrica de producto sin ambigüedades. Defina sólo algunas métricas y luego úselas para obtener comprensión acerca de la calidad de un producto de trabajo de ingeniería del software.

Aunque las métricas de producto para el software de computadora son imperfectas, pueden proporcionar una forma sistemática de valorar la calidad con base en un conjunto de reglas claramente definidas. También proporcionan comprensión inmediata, en lugar de hacerlo después de los hechos. Esto permite descubrir y corregir potenciales problemas antes de que se conviertan en defectos catastróficos.

En este capítulo se presentan las mediciones que pueden usarse para valorar la calidad del producto conforme se somete a ingeniería. Estas mediciones de atributos internos del producto ofrecen una indicación en tiempo real de la eficacia de los modelos de requerimientos, diseño y código, así como de la efectividad de los casos de prueba y de la calidad global del software que se va a construir.

23.1 Marco conceptual para las métricas de producto

Como se anotó en la introducción, la medición asigna números o símbolos a atributos de entidades en el mundo real. Para lograr esto, se requiere un modelo de medición que abarque un conjunto consistente de reglas. Aunque la teoría de la medición (por ejemplo, [Kyb84]) y su aplicación al software de computadora (por ejemplo, [Zus97]) son temas que están más allá del ámbito de este libro, vale la pena establecer un marco conceptual fundamental y un conjunto de principios básicos que guíen la definición de las métricas de producto para el software.

Cita:

"Una ciencia es tan madura como sus herramientas de medición."

Louis Pasteur

23.1.1 Medidas, métricas e indicadores

Aunque los términos *medida, medición* y *métrica* con frecuencia se usan de modo intercambiable, es importante observar las sutiles diferencias entre ellos. En el contexto de la ingeniería del software, una *medida* proporciona un indicio cuantitativo de la extensión, cantidad, dimensión, capacidad o tamaño de algún atributo de un producto o proceso. La *medición* es el acto de determinar una medida. El *IEEE Standard Glosary of Software Engineering Terminology* [IEE93b] define *métrica* como "una medida cuantitativa del grado en el que un sistema, componente o proceso posee un atributo determinado".

¿Cuál es la diferencia entre una medida y una métrica?

Cuando se ha recolectado un solo punto de datos (por ejemplo, el número de errores descubiertos dentro de un solo componente de software), se establece una medida. La medición ocurre como resultado de la recolección de uno o más puntos de datos (por ejemplo, algunas revisiones de componente y pruebas de unidad se investigan para recolectar medidas del número de errores de cada uno). Una métrica de software relaciona en alguna forma las medidas individuales (por ejemplo, el número promedio de errores que se encuentran por revisión o el número promedio de errores que se encuentran por unidad de prueba).

PUNTO CLAVE

Un indicador es una o varias métricas que proporcionan comprensión acerca del proceso, el producto o el proyecto.

Un ingeniero de software recolecta medidas y desarrolla métricas de modo que se obtengan indicadores. Un *indicador* es una métrica o combinación de métricas que proporcionan comprensión acerca del proceso de software, el proyecto de software o el producto en sí. Un indicador proporciona comprensión que permite al gerente de proyecto o a los ingenieros de software ajustar el proceso, el proyecto o el producto para hacer mejor las cosas.

23.1.2 El reto de la métrica de producto

Durante las cuatro décadas pasadas, muchos investigadores intentaron desarrollar una sola métrica que proporcionara una medida abarcadora de la complejidad del software. Fenton [Fen94] caracteriza esta investigación como una búsqueda del "imposible Santo Grial". Aunque se han propuesto decenas de medidas de complejidad [Zus90], cada una toma una visión un poco diferente de lo que es la complejidad y de qué atributos de un sistema conducen a la complejidad. Por analogía, considere una métrica para evaluar un automóvil atractivo. Algunos observadores pueden enfatizar el diseño de la carrocería; otros pueden considerar característi-

Cita:

"Así como la medición de temperatura comenzó con un dedo índice... y creció hasta escalas, herramientas y técnicas sofisticadas, de igual modo madura la medición del software."

Shari Pfleeger

cas mecánicas; otros más pueden destacar el costo o el rendimiento o el uso de combustibles alternativos o la capacidad para reciclar cuando el carro sea chatarra. Dado que cualquiera de estas características puede estar en desacuerdo con otras, es difícil derivar un solo valor para el "atractivo". El mismo problema ocurre con el software de computadora.

Aunque existe la necesidad de medir y de controlar la complejidad del software, y si bien un solo valor de esta métrica de calidad es difícil de derivar, es posible desarrollar medidas de diferentes atributos internos de programa (por ejemplo, modularidad efectiva, independencia funcional y otros atributos estudiados en el capítulo 8). Estas medidas y las métricas derivadas de ellas pueden usarse como indicadores independientes de la calidad de los modelos de requerimientos y de diseño. Pero aquí de nuevo surgen problemas. Fenton [Fen94] observa esto cuando afirma: "el peligro de intentar encontrar medidas que caracterizan tantos atributos diferentes es que, inevitablemente, las medidas tienen que satisfacer objetivos en conflicto. Esto es contrario a la teoría representacional de las mediciones". Aunque el enunciado de Fenton es correcto, muchas personas argumentan que la medición de producto realizada durante las primeras etapas del proceso de software brinda a los ingenieros de software un mecanismo consistente y objetivo para valorar la calidad.

WebRef

Horst Zuse compiló voluminosa información acerca de la métrica de producto en **irb.cs.tu-berlin.de/~zuse/**

Sin embargo, es justo preguntar cuán válidas son las métricas de producto, es decir, ¿cuán cercanamente se alinean las métricas de producto con la confiabilidad a largo plazo y con la calidad de un sistema basado en computadora? Fenton [Fen91] aborda esta pregunta en la forma siguiente:

> A pesar de las conexiones intuitivas entre la estructura interna de los productos de software [métricas de producto] y sus atributos externos de producto y proceso, en realidad ha habido pocos intentos científicos para establecer relaciones específicas. Existen algunas razones por las que esto es así; la más comúnmente citada es lo impráctico que resulta realizar experimentos relevantes.

Cada uno de los "retos" anotados aquí es un motivo de precaución, pero no es razón para desechar las métricas de producto.[1] La medición es esencial si debe lograrse la calidad.

23.1.3 Principios de medición

Antes de presentar una serie de métricas de producto que 1) auxilien en la evaluación de los modelos de análisis y diseño, 2) proporcionen un indicio de la complejidad de los diseños procedimentales y del código fuente y 3) faciliten el diseño de pruebas más efectivas, es importante comprender los principios de medición básicos. Roche [Roc94] sugiere un proceso de medición que puede caracterizarse mediante cinco actividades:

? ¿Cuáles son los pasos de un proceso de medición efectivo?

- *Formulación.* La derivación de medidas y métricas de software apropiadas para la representación del software que se está construyendo.
- *Recolección.* Mecanismo que se usa para acumular datos requeridos para derivar las métricas formuladas.
- *Análisis.* El cálculo de métricas y la aplicación de herramientas matemáticas.
- *Interpretación.* Evaluación de las métricas resultantes para comprender la calidad de la representación.
- *Retroalimentación.* Recomendaciones derivadas de la interpretación de las métricas del producto, transmitidas al equipo de software.

1 Aunque las críticas de métricas específicas son comunes en la literatura, muchos críticos se enfocan en conflictos particulares y pierden el objetivo principal de las métricas en el mundo real: ayudar al ingeniero de software a establecer una vía sistemática y objetiva para obtener comprensión de su trabajo y mejorar la calidad del producto resultante.

Las métricas de software serán útiles sólo si se caracterizan efectivamente y si se validan de manera adecuada. Los siguientes principios [Let03b] son representativos de muchos que pueden proponerse para la caracterización y validación de métricas:

En realidad, muchas métricas de producto actualmente en uso no concuerdan con dichos principios tanto como debieran. Pero eso no significa que no tengan valor; sólo tenga cuidado cuando los use y entienda que tienen la intención de proporcionar comprensión, no estricta verificación científica.

- *Una métrica debe tener propiedades matemáticas deseables,* es decir, el valor de la métrica debe estar en un rango significativo (por ejemplo, 0 a 1, donde 0 realmente significa ausencia, 1 indica el valor máximo y 0.5 representa el "punto medio"). Además, una métrica que intente estar en una escala racional no debe constituirse con componentes que sólo se miden en una escala ordinal.
- *Cuando una métrica representa una característica de software que aumenta cuando ocurren rasgos positivos o que disminuye cuando se encuentran rasgos indeseables, el valor de la métrica debe aumentar o disminuir en la misma forma.*
- *Cada métrica debe validarse de manera empírica en una gran variedad de contextos antes de publicarse o utilizarse para tomar decisiones.* Una métrica debe medir el factor de interés, independientemente de otros factores. Debe "escalar" a sistemas más grandes y funcionar en varios lenguajes de programación y dominios de sistema.

Aunque la formulación, caracterización y validación son cruciales, la recolección y el análisis son las actividades que impulsan el proceso de medición. Roche [Roc94] sugiere los siguientes principios para dichas actividades: 1) siempre que sea posible, la recolección y el análisis de datos deben automatizarse; 2) deben aplicarse técnicas estadísticas válidas para establecer relaciones entre atributos de producto internos y características de calidad externas (por ejemplo, si el nivel de complejidad arquitectónica se correlaciona con el número de defectos reportados en el uso de producción), y 3) para cada métrica deben establecerse lineamientos y recomendaciones interpretativos.

23.1.4 Medición de software orientado a meta

WebRef

En **www.thedacts.com/GoldPractices/practices/gqma.html** puede encontrar un útil foro de discusión sobre el MPM.

El paradigma *Meta/Pregunta/Métrica* (MPM) fue desarrollado por Basili y Weiss [Bas84] como una técnica para identificar métricas significativas para cualquier parte del proceso de software. MPM enfatiza la necesidad de: 1) establecer una *meta* de medición explícita que sea específica para la actividad del proceso o para la característica del producto que se quiera valorar, 2) definir un conjunto de *preguntas* que deban responderse con la finalidad de lograr la meta y 3) identificar *métricas* bien formuladas que ayuden a responder dichas preguntas.

Para definir cada meta de medición, puede usarse una *plantilla de definición de meta* [Bas94]. La plantilla toma la forma:

> **Analizar** {el nombre de la actividad o atributo que se va a medir} **con el propósito de** {el objetivo global del análisis[2]} **con respecto a** {el aspecto de la actividad o atributo que se considera} **desde el punto de vista de** {las personas que tienen interés en la medición} **en el contexto de** {el entorno en el que tiene lugar la medición}.

Como ejemplo, considere una plantilla de definición de meta para *CasaSegura*:

> **Analizar** la arquitectura del software *CasaSegura* **con el propósito de** evaluar los componentes arquitectónicos **con respecto a** la capacidad de hacer *CasaSegura* más extensible **desde el punto de vista de** los ingenieros de software que realizan el trabajo **en el contexto de** mejora del producto durante los próximos tres años.

2 Van Solingen y Berghout [Sol90] sugieren que el objeto casi siempre es "comprender, controlar o mejorar" la actividad del proceso o el atributo del producto.

Con una meta de medición definida de manera explícita, se desarrolla un conjunto de preguntas. Las respuestas ayudarán al equipo de software (o a otros participantes) a determinar si se logró la meta de medición. Entre las preguntas que pueden plantearse se encuentran:

P_1: ¿Los componentes arquitectónicos se caracterizan de forma que compartimentalizan la función y los datos relacionados?

P_2: ¿La complejidad de cada componente dentro de las fronteras facilitará la modificación y la extensión?

Cada una de estas preguntas debe responderse de manera cuantitativa, usando una o más medidas y métricas. Por ejemplo, una métrica que proporciona un indicio de la cohesión (capítulo 8) de un componente arquitectónico puede ser útil para responder P_1. Las métricas que se estudian más adelante en este capítulo pueden proporcionar comprensión para P_2. En todo caso, las métricas que se eligen (o derivan) deben corresponderse con los principios de medición analizados en la sección 23.1.3 y con los atributos de medición expuestos en la sección 23.1.5.

23.1.5 Atributos de las métricas de software efectivas

Se han propuesto cientos de métricas para el software de computadora, pero no todas brindan apoyo práctico al ingeniero de software. Algunas demandan medición demasiado compleja, otras son tan particulares que pocos profesionales del mundo real tienen alguna esperanza de entenderlas y otras más violan las nociones intuitivas básicas de lo que realmente es el software de alta calidad.

Ejiogu [Eji91] define un conjunto de atributos que deben abarcar las métricas de software efectivas. La métrica derivada y las medidas que conducen a ella deben ser:

¿Cómo se valora la calidad de una métrica de software propuesta?

- *Simple y calculable.* Debe ser relativamente fácil aprender cómo derivar la métrica y su cálculo no debe demandar esfuerzo o tiempo excesivo.
- *Empírica e intuitivamente convincente.* Debe satisfacer las nociones intuitivas del ingeniero acerca del atributo de producto que se elabora (por ejemplo, una métrica que mide la cohesión del módulo debe aumentar en valor conforme aumenta el nivel de cohesión).
- *Congruente y objetiva.* Siempre debe producir resultados que no tengan ambigüedades. Una tercera parte independiente debe poder derivar el mismo valor de métrica usando la misma información acerca del software.
- *Constante en su uso de unidades y dimensiones.* El cálculo matemático de la métrica debe usar medidas que no conduzcan a combinaciones extrañas de unidades. Por ejemplo, multiplicar personas en los equipos de proyecto por variables de lenguaje de programación en el programa da como resultado una mezcla sospechosa de unidades que no son intuitivamente convincentes.
- *Independiente del lenguaje de programación.* Debe basarse en el modelo de requerimientos, el modelo de diseño o la estructura del programa en sí. No debe depender de los caprichos de la sintaxis o de la semántica del lenguaje de programación.
- *Un mecanismo efectivo para retroalimentación de alta calidad.* Debe proporcionar información que pueda conducir a un producto final de mayor calidad.

CONSEJO

La experiencia indica que una métrica de producto sólo se usará si es intuitiva y fácil de calcular. Si se tienen que hacer decenas de "conteos" y se requieren cálculos complejos, es improbable que la métrica se adopte ampliamente.

Aunque la mayoría de las métricas de software satisfacen estos atributos, algunas métricas de uso común pueden fracasar para satisfacer uno o dos de ellos. Un ejemplo es el punto de función (que se estudia en la sección 23.2.1), una medida de la "funcionalidad" entregada por el software. Puede argumentarse[3] que el atributo de congruente y objetiva fracasa porque una tercera

3 Puede plantearse un contrargumento igualmente vigoroso. Tal es la naturaleza de las métricas del software.

parte independiente no puede ser capaz de derivar el mismo valor del punto de función que un colega que use la misma información acerca del software. Por tanto, ¿debe rechazarse la medida PF? La respuesta es "¡desde luego que no!". El PF proporciona comprensión útil y, en consecuencia, ofrece distinto valor, incluso si falla en satisfacer un atributo a la perfección.

CASASEGURA

Debate acerca de las métricas de producto

La escena: Cubículo de Vinod.

Participantes: Vinod, Jamie y Ed, miembros del equipo de ingeniería del software *CasaSegura*, quienes continúan trabajando en el diseño en el nivel de componentes y en el diseño de casos de prueba.

La conversación:

Vinod: Doug [Doug Miller, gerente de ingeniería del software] me dijo que todos debemos usar métricas de producto, pero fue muy vago. También dijo que no presionaría... que usarlas era asunto nuestro.

Jamie: Eso está bien porque no hay forma de que yo tenga tiempo para comenzar esa cosa de las medidas. Estamos peleando por mantener el calendario como está.

Ed: Estoy de acuerdo con Jamie. Estamos en contra, aquí... no tenemos tiempo.

Vinod: Sí, lo sé, pero probablemente hay algún mérito en usarlas.

Jamie: No lo discuto, Vinod, es cuestión de tiempo... y, en lo que a mí respecta, no tengo para perderlo.

Vinod: Pero, ¿y si las mediciones te ahorran tiempo?

Ed: Estás mal, requieren tiempo, y, como dijo Jamie...

Vinod: No, espera... ¿y si nos ahorra tiempo?

Jamie: ¿Cómo?

Vinod: Volver a trabajar... así es cómo. Si una medida que usemos nos ayuda a evitar un problema grande o incluso moderado, y esto evita que tengamos que volver a trabajar una parte del sistema, ahorramos tiempo. ¿O no?

Ed: Es posible, supongo, ¿pero puedes garantizarnos que alguna métrica de producto nos ayudará a encontrar un problema?

Vinod: ¿Puedes garantizarme que no lo hará?

Jamie: ¿Y qué es lo que propones?

Vinod: Creo que debemos seleccionar algunas métricas de diseño, probablemente orientadas a clase, y usarlas como parte de nuestro proceso de revisión para cada componente que desarrollemos.

Ed: No estoy familiarizado con las métricas orientadas a clase.

Vinod: Yo pasaré algo de tiempo revisándolas y haré una recomendación... ¿está bien para ustedes?

[Ed y Jamie asienten sin mucho entusiasmo.]

23.2 MÉTRICAS PARA EL MODELO DE REQUERIMIENTOS

El trabajo técnico en la ingeniería del software comienza con la creación del modelo de requerimientos. En esta etapa se derivan los requerimientos y se establece un cimiento para el diseño. Por tanto, son deseables métricas de producto que proporcionen comprensión acerca de la calidad del modelo de análisis.

Aunque en la literatura han aparecido relativamente pocas métricas de análisis y especificación, es posible adaptar las métricas que se usan frecuentemente para estimación de proyectos y aplicarlas en este contexto. Dichas métricas examinan el modelo de requerimientos con la intención de predecir el "tamaño" del sistema resultante. En ocasiones (mas no siempre), el tamaño es un indicador de la complejidad del diseño y casi siempre es un indicador creciente de codificación, integración y esfuerzo de pruebas.

23.2.1 Métrica basada en funciones

La *métrica de punto de función* (PF) puede usarse de manera efectiva como medio para medir la funcionalidad que entra a un sistema.[4] Al usar datos históricos, la métrica PF puede entonces usarse para: 1) estimar el costo o esfuerzo requerido para diseñar, codificar y probar el software;

4 Acerca de las métricas PF se han escrito cientos de libros, ensayos y artículos. En [IFP05] puede encontrar una valiosa bibliografía.

WebRef

En **www.ifpug.org** y en **www.functionpoints.com** puede obtenerse mucha información útil acerca de los puntos de función.

2) predecir el número de errores que se encontrarán durante las pruebas, y 3) prever el número de componentes y/o de líneas fuente proyectadas en el sistema implementado.

Los puntos de función se derivan usando una relación empírica basada en medidas contables (directas) del dominio de información del software y en valoraciones cualitativas de la complejidad del software. Los valores de dominio de información se definen en la forma siguiente:[5]

Número de entradas externas (EE). Cada *entrada externa* se origina de un usuario o se transmite desde otra aplicación, y proporciona distintos datos orientados a aplicación o información de control. Con frecuencia, las entradas se usan para actualizar *archivos lógicos internos* (ALI). Las entradas deben distinguirse de las consultas, que se cuentan por separado.

Número de salidas externas (SE). Cada *salida externa* es datos derivados dentro de la aplicación que ofrecen información al usuario. En este contexto, salida externa se refiere a reportes, pantallas, mensajes de error, etc. Los ítems de datos individuales dentro de un reporte no se cuentan por separado.

Número de consultas externas (CE). Una *consulta externa* se define como una entrada en línea que da como resultado la generación de alguna respuesta de software inmediata en la forma de una salida en línea (con frecuencia recuperada de un ALI).

Número de archivos lógicos internos (ALI). Cada *archivo lógico interno* es un agrupamiento lógico de datos que reside dentro de la frontera de la aplicación y se mantiene mediante entradas externas.

Número de archivos de interfaz externos (AIE). Cada *archivo de interfaz externo* es un agrupamiento lógico de datos que reside fuera de la aplicación, pero que proporciona información que puede usar la aplicación.

Una vez recolectados dichos datos, la tabla de la figura 23.1 se completa y un valor de complejidad se asocia con cada conteo. Las organizaciones que usan métodos de punto de función desarrollan criterios para determinar si una entrada particular es simple, promedio o compleja. No obstante, la determinación de complejidad es un tanto subjetiva.

Para calcular puntos de función (PF), se usa la siguiente relación:

$$\text{PF} = \text{conteo total} \times [0.65 + 0.01 \times \Sigma (F_i)] \tag{23.1}$$

donde conteo total es la suma de todas las entradas PF obtenidas de la figura 23.1.

Los F_i (i = 1 a 14) son *factores de ajuste de valor* (FAV) con base en respuestas a las siguientes preguntas [Lon02]:

FIGURA 23.1
Cálculo de puntos de función

Valor de dominio de información	Conteo		Factor ponderado Simple	Promedio	Complejo		
Entradas externas (EE)		×	3	4	6	=	
Salidas externas (SE)		×	4	5	7	=	
Consultas externas (CE)		×	3	4	6	=	
Archivos lógicos internos (ALI)		×	7	10	15	=	
Archivos de interfaz externos (AIE)		×	5	7	10	=	
Conteo total	→						

5 En realidad, la definición de valores de dominio de información y la forma en la que se cuentan son un poco más complejos. Para más detalles, el lector interesado debe consultar [IFP01].

PUNTO CLAVE

Los factores de ajuste de valor se usan para proporcionar un indicio de la complejidad del problema.

1. ¿El sistema requiere respaldo y recuperación confiables?
2. ¿Se requieren comunicaciones de datos especializadas para transferir información hacia o desde la aplicación?
3. ¿Existen funciones de procesamiento distribuidas?
4. ¿El desempeño es crucial?
5. ¿El sistema correrá en un entorno operativo existente enormemente utilizado?
6. ¿El sistema requiere entrada de datos en línea?
7. ¿La entrada de datos en línea requiere que la transacción de entrada se construya sobre múltiples pantallas u operaciones?
8. ¿Los ALI se actualizan en línea?
9. ¿Las entradas, salidas, archivos o consultas son complejos?
10. ¿El procesamiento interno es complejo?
11. ¿El código se diseña para ser reutilizable?
12. ¿La conversión y la instalación se incluyen en el diseño?
13. ¿El sistema se diseña para instalaciones múltiples en diferentes organizaciones?
14. ¿La aplicación se diseña para facilitar el cambio y su uso por parte del usuario?

WebRef

En **irb.cs.uni-mogdeburg.de/sw-eng/us/java/fp/** puede encontrar una calculadora PF en línea.

Cada una de estas preguntas se responde usando una escala que varía de 0 (no importante o aplicable) a 5 (absolutamente esencial). Los valores constantes en la ecuación (23.1) y los factores ponderados que se aplican a los conteos de dominio de información se determinan de manera empírica.

Para ilustrar el uso de la métrica PF en este contexto, considere la representación simple de modelo de análisis que se ilustra en la figura 23.2. En la figura, se representa un diagrama de flujo de datos (capítulo 7) para una función dentro del software *CasaSegura*.

La función gestiona la interacción del usuario, acepta la contraseña de éste para activar o desactivar el sistema y permite consultas sobre el estado de las zonas de seguridad y de varios sensores de seguridad. La función despliega una serie de mensajes de advertencia y envía señales de control adecuadas a varios componentes del sistema de seguridad.

El diagrama de flujo de datos se evalúa para determinar un conjunto de medidas de dominio de información clave que son requeridas para calcular la métrica de punto de función. En la figura se muestran tres entradas externas (**contraseña**, **botón de pánico** y **activar/desactivar**), junto con dos consultas externas (**consulta de zona** y **consulta de sensor**). Se muestra un ALI (**archivo configuración sistema**) y también están presentes dos salidas externas

FIGURA 23.2

Modelo de flujo de datos para el software *CasaSegura*

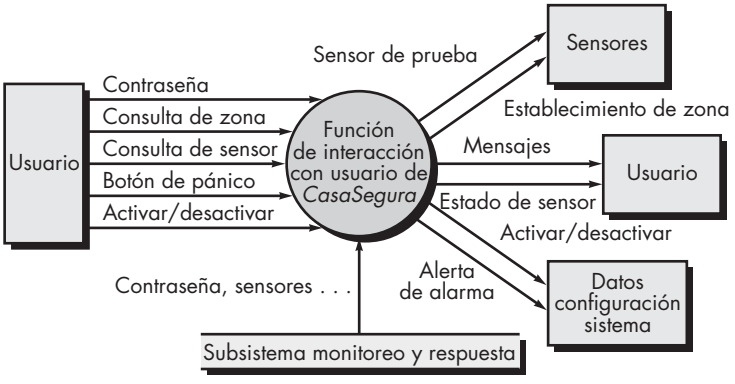


FIGURA 23.3
Cálculo de puntos de función

Valor dominio de información	Conteo		Factor ponderado: Simple	Promedio	Complejo		
Entradas externas (EE)	3	×	3	4	6	=	9
Salidas externas (SE)	2	×	4	5	7	=	8
Consultas externas (CE)	2	×	3	4	6	=	6
Archivos lógicos internos (ALI)	1	×	7	10	15	=	7
Archivos de interfaz externos (AIE)	4	×	5	7	10	=	20
Conteo total							50

(**mensajes** y **estado de sensor**) y cuatro AIE (**sensor de prueba**, **establecimiento de zona**, **activar/desactivar** y **alerta de alarma**). En la figura 23.3 se muestran estos datos, junto con la complejidad adecuada.

El conteo total que se muestra en la figura 23.3 debe ajustarse usando la ecuación (23.1). Para los propósitos de este ejemplo, suponga que $\Sigma(F_i)$ es 46 (un producto moderadamente complejo). Por tanto,

$$PF = 50 \times [0.65 + (0.01 \times 46)] = 56$$

Cita:

"En lugar de sólo meditar acerca de cuál 'nueva métrica' aplicar [...] también debemos plantearnos la pregunta más básica: ¿qué haremos con las métricas?"

Michael Mah y Larry Putnam

Con base en el valor PF proyectado, derivado del modelo de requerimientos, el equipo del proyecto puede estimar el tamaño global implementado de la función de interacción del usuario de *CasaSegura*. Suponga que los datos anteriores indican que un PF se traduce en 60 líneas de código (se usará un lenguaje orientado a objeto) y que se producen 12 PF por cada persona-mes de esfuerzo. Estos datos históricos ofrecen al gerente de proyecto información importante de planificación que se basa en el modelo de requerimientos y no en estimaciones preliminares. Suponga aún más, que en los proyectos pasados se encontró un promedio de tres errores por punto de función durante las revisiones de requerimientos y diseño, y cuatro errores por punto de función durante las pruebas de unidad e integración. A final de cuentas, dichos datos pueden ayudarlo a valorar lo completo de sus actividades de revisión y pruebas.

Uemura *et al.* [Uem99] sugieren que los puntos de función también pueden calcularse a partir de clases UML y diagramas de secuencia. Si tiene más interés, consulte detalles en [Uem99].

23.2.2 Métricas para calidad de la especificación

Davis *et al.* [Dav93] proponen una lista de características que pueden usarse para valorar la calidad del modelo de requerimientos y la correspondiente especificación de requerimientos: *especificidad* (falta de ambigüedad), *completitud*, *corrección*, *comprensibilidad*, *verificabilidad*, *consistencia interna y externa*, *factibilidad*, *concisión*, *rastreabilidad*, *modificabilidad*, *precisión* y *reusabilidad*. Además, los autores observan que las especificaciones de alta calidad se almacenan electrónicamente, son ejecutables o al menos interpretables, se anotan mediante importancia relativa, son estables, tienen versión, se organizan, cuentan con referencia cruzada y se especifican en el nivel correcto de detalle.

Aunque muchas de estas características parecen ser cualitativas por naturaleza, Davis *et al.* [Dav93] sugieren que cada una puede representarse usando una o más métricas. Por ejemplo, se supone que existen n_r requerimientos en una especificación, tales que

$$n_r = n_f + n_{nf}$$

donde n_f es el número de requerimientos funcionales y n_{nf} es el número de requerimientos no funcionales (por ejemplo, rendimiento).

PUNTO CLAVE
Al medir las características de la especificación, es posible obtener comprensión cuantitativa acerca de la especificidad y de la completitud.

Para determinar la *especificidad* (falta de ambigüedad) de los requerimientos, Davis *et al.*, sugieren una métrica que se basa en la consistencia de la interpretación de los revisores de cada requisito:

$$Q_1 = \frac{n_{ui}}{n_r}$$

donde n_{ui} es el número de requerimientos para los cuales todos los revisores tienen interpretaciones idénticas. Mientras más cercano a 1 esté el valor de Q, menor será la ambigüedad de la especificación.

La *completitud* de los requerimientos funcionales puede determinarse al calcular la razón

$$Q_2 = \frac{n_u}{n_i \times n_s}$$

donde n_u es el número de requerimientos funcionales únicos, n_i es el número de entradas (estímulos) definidas o implicadas por la especificación y n_s es el número de estados especificados. La razón Q_2 mide el porcentaje de funciones necesarias que se especificaron para un sistema. Sin embargo, no aborda requerimientos no funcionales. Para incorporar éstos en una métrica global de completitud, se debe considerar el grado en el que se validaron los requerimientos:

$$Q_3 = \frac{n_c}{n_c + n_{nv}}$$

donde n_c es el número de requerimientos que se validaron como correctos y n_{nv} es el número de requerimientos que no se han validado.

Cita:

"Medir lo que es mensurable y lo que no es mensurable, hace [lo] mensurable."

Galileo

23.3 MÉTRICAS PARA EL MODELO DE DISEÑO

PUNTO CLAVE
Las métricas pueden proporcionar comprensión acerca de los datos estructurales y de la complejidad del sistema asociada con el diseño arquitectónico.

Es inconcebible que el diseño de una nueva aeronave, un nuevo chip de computadora o un nuevo edificio de oficinas se realizara sin definir medidas de diseño, ni determinar las métricas para varios aspectos de la calidad del diseño ni usarlos como indicadores para guiar la forma en la que evoluciona el diseño. Y aún así, con frecuencia el diseño de los sistemas complejos basados en software procede virtualmente sin medición. La ironía de esto es que están disponibles métricas del diseño para el software, pero la gran mayoría de los ingenieros del software continúan sin percatarse de su existencia.

Las métricas de diseño para software de computadora, al igual que todas las demás métricas de software, no son perfectas. El debate continúa acerca de su eficacia y sobre la forma en la que deben aplicarse. Muchos expertos argumentan que se requiere más experimentación antes de poder usar las medidas de diseño, aunque el diseño sin medición es una alternativa inaceptable.

En las siguientes secciones se examinan algunas de las métricas de diseño más comunes para software de computadora. Cada una puede proporcionarle comprensión mejorada y todas pueden ayudar a que el diseño evolucione hacia un mayor nivel de calidad.

23.3.1 Métricas del diseño arquitectónico

Las métricas del diseño arquitectónico se enfocan en características de la arquitectura del programa (capítulo 9) con énfasis en la estructura arquitectónica y en la efectividad de los módulos o componentes dentro de la arquitectura. Dichas métricas son "caja negra" en tanto no requieren conocimiento alguno del funcionamiento interior de un componente de software particular.

Card y Glass [Car90] definen tres medidas de complejidad del diseño de software: complejidad estructural, complejidad de datos y complejidad del sistema.

Para arquitecturas jerárquicas (por ejemplo, arquitecturas de petición y retorno), la *complejidad estructural* de un módulo *i* se define de la forma siguiente:

$$S(i) = f^2_{\text{out}}(i)$$

donde $f_{\text{out}}(i)$ es el *fan-out*[6] del módulo *i*.

La *complejidad de datos* ofrece un indicio de la complejidad que hay en la interfaz interna para un módulo *i* y se define como

$$D(i) = \frac{v(i)}{f_{\text{out}}(i) + 1}$$

donde $v(i)$ es el número de variables de entrada y salida que pasan hacia y desde el módulo *i*.

Finalmente, la *complejidad del sistema* se define como la suma de las complejidades estructural y de datos y se especifica como

$$C(i) = S(i) + D(i)$$

Conforme aumenta el valor de cada una de estas complejidades, la complejidad arquitectónica global del sistema también aumenta. Esto conduce a una mayor probabilidad de que también aumenten el esfuerzo de integración y el de pruebas.

Fenton [Fen91] sugiere algunas métricas simples de morfología (es decir, de forma) que permiten la comparación de diferentes arquitecturas de programa usando un conjunto de dimensiones directas. Con respecto a la arquitectura de llamado y retorno de la figura 23.4, puede definirse la siguiente métrica:

Tamaño = $n + a$

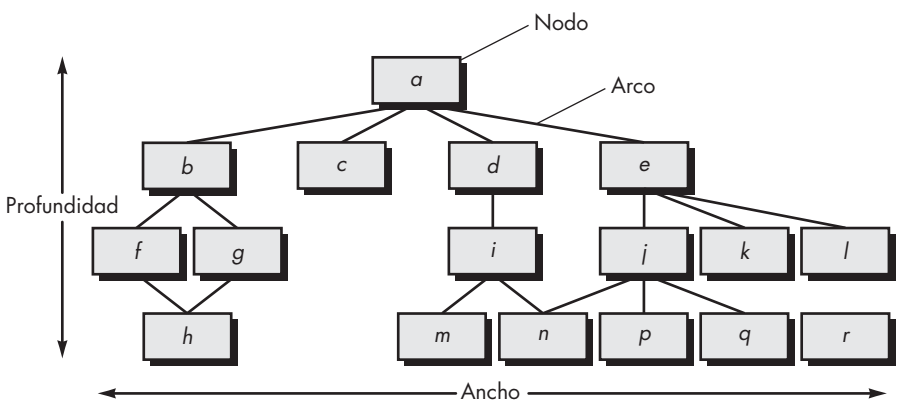
donde *n* es el número de nodos y *a* es el número de arcos. Para la arquitectura que se muestra en la figura 23.4,

Tamaño = 17 + 18 = 35
Profundidad = trayectoria más larga desde el nodo raíz (superior) hasta un nodo hoja. Para la arquitectura que se muestra en la figura 23.4, profundidad = 4.
Ancho = número máximo de nodos en cualquier nivel de la arquitectura. Para la arquitectura que se muestra en la figura 23.4, ancho = 6.

La razón arco a nodo, $r = a/n$, mide la densidad de conectividad de la arquitectura y puede proporcionar un indicio simple del acoplamiento de la arquitectura. Para la arquitectura que se muestra en la figura 23.4, $r = 18/17 = 1.06$.

La comandancia de los sistemas de la fuerza aérea estadounidense [USA87] desarrolló algunos indicadores de calidad del software que se basan en las características de diseño mensura-

FIGURA 23.4
Métricas de morfología



6 *Fan-out* (cargabilidad o abanico de salida) se define como el número de módulos inmediatamente subordinados al módulo *i*, es decir, el número de módulos que invoca directamente el módulo *i*.

bles de un programa de computadora. Al usar conceptos similares a los propuestos en IEEE Std. 982.1-1988 [IEE94], la fuerza aérea usa la información obtenida de los diseños de datos y arquitectónico para derivar un *índice de calidad de la estructura del diseño* (ICED) que varía de 0 a 1. Es necesario averiguar los siguientes valores para calcular el ICED [Cha89]:

S_1 = número total de módulos definidos en la arquitectura del programa
S_2 = número de módulos cuya función correcta depende de la fuente de entrada de datos o que produce los datos que se van a utilizar en alguna otra parte (en general, los módulos de control, entre otros, no se contarían como parte de S_2)
S_3 = número de módulos cuya función correcta depende del procesamiento previo
S_4 = número de ítems de base de datos (incluidos objetos de datos y todos los atributos que definen objetos)
S_5 = número total de ítems de base de datos únicos
S_6 = número de segmentos de base de datos (diferentes registros u objetos individuales)
S_7 = número de módulos con una sola entrada y salida (el procesamiento de excepción no se considera como una salida múltiple)

Cita:

"La medición puede verse como una desviación. Ésta es necesaria porque los humanos básicamente no son capaces de tomar decisiones claras y objetivas [sin apoyo cuantitativo]".

Horst Zuse

Una vez determinados los valores de S_1 a S_7 para un programa de cómputo, pueden calcularse los siguientes valores intermedios:

Estructura del programa: D_1, donde D_1 se define del modo siguiente: si el diseño arquitectónico se desarrolló usando un método distinto (por ejemplo, diseño orientado a flujo de datos o diseño orientado a objeto), entonces $D_1 = 1$, de otro modo $D_1 = 0$.

Independencia de módulo: $D_2 = 1 - \frac{S_2}{S_1}$

Módulos no dependientes del procesamiento previo: $D_3 = 1 - \frac{S_3}{S_1}$

Tamaño de base de datos: $D_4 = 1 - \frac{S_5}{S_4}$

Compartimentalización de base de datos: $D_5 = 1 - \frac{S_6}{S_4}$

Entrada/salida de módulo característico: $D_6 = 1 - \frac{S_7}{S_1}$

Con la determinación de estos valores intermedios, el ICED se calcula en la forma siguiente:

$$\text{ICED} = \Sigma\, w_i D_i$$

donde $i = 1$ a 6, w_i es el peso relativo de la importancia de cada uno de los valores intermedios y $\Sigma w_i = 1$ (si todos los D_i pesan igual, entonces $w_i = 0.167$).

El valor del ICED para diseños anteriores puede determinarse y compararse con un diseño que actualmente esté en desarrollo. Si el ICED es significativamente menor que el promedio, se indican más trabajo de diseño y revisión. De igual modo, si se hacen grandes cambios a un diseño existente puede calcularse el efecto de dichos cambios en el ICED.

23.3.2 Métricas para diseño orientado a objetos

Hay mucho de subjetivo en el diseño orientado a objetos: un diseñador experimentado "sabe" cómo caracterizar un sistema OO de modo que implemente de manera efectiva los requerimientos del cliente. Pero, conforme un modelo de diseño OO crece en tamaño y complejidad, una visión más objetiva de las características del diseño puede beneficiar tanto al diseñador experimentado (quien adquiere comprensión adicional) como al novato (quien obtiene un indicio de la calidad que de otro modo no tendría disponible).

En un tratamiento detallado de las métricas de software para sistemas OO, Whitmire [Whi97] describe nueve características distintas y mensurables de un diseño OO:

¿Qué características pueden medirse cuando se valora un diseño OO?

Tamaño. El tamaño se define en función de cuatro visiones: población, volumen, longitud y funcionalidad. La *población* se mide al realizar un conteo estático de entidades OO, tales como clases u operaciones. Las medidas de *volumen* son idénticas a las medidas de población, pero se recolectan de manera dinámica: en un instante de tiempo determinado. La *longitud* es una medida de una cadena de elementos de diseño interconectados (por ejemplo, la profundidad de un árbol de herencia es una medida de longitud). Las métricas de *funcionalidad* proporcionan un indicio indirecto del valor entregado al cliente por una aplicación OO.

Cita:

"Muchas de las decisiones en las cuales tuve que confiar en el folclore y el mito, ahora se pueden resolver haciendo uso de datos cuantitativos."

Scott Whitmire

Complejidad. Como el tamaño, existen muchas visiones diferentes de la complejidad del software [Zus97]. Whitmire ve la complejidad en términos de características estructurales al examinar cómo se relacionan mutuamente las clases de un diseño OO.

Acoplamiento. Las conexiones físicas entre elementos del diseño OO (por ejemplo, el número de colaboraciones entre clases o el de mensajes que pasan entre los objetos) representan el acoplamiento dentro de un sistema OO.

Suficiencia. Whitmire define *suficiencia* como "el grado en el que una abstracción posee las características requeridas de él o en el que un componente de diseño posee características en su abstracción, desde el punto de vista de la aplicación actual". Dicho de otra forma, se pregunta: "¿Qué propiedades debe poseer esta abstracción (clase) para serme útil?" [Whi97]. En esencia, un componente de diseño (por ejemplo, una clase) es *suficiente* si refleja por completo todas las propiedades del objeto de dominio de aplicación que se modela, es decir, si la abstracción (clase) posee sus características requeridas.

Completitud. La única diferencia entre completitud y suficiencia es "el conjunto de características contra las cuales se compara la abstracción o el componente de diseño" [Whi97]. La suficiencia compara la abstracción desde el punto de vista de la aplicación actual. La *completitud* considera múltiples puntos de vista, y plantea la pregunta: "¿qué propiedades se requieren para representar por completo al objeto de dominio problema?". Puesto que el criterio para completitud considera diferentes puntos de vista, tiene una implicación indirecta en el grado en el que puede reutilizarse la abstracción o el componente de diseño.

Cohesión. Como su contraparte en software convencional, un componente OO debe diseñarse de manera que tenga todas las operaciones funcionando en conjunto para lograr un solo propósito bien definido. La cohesividad de una clase se determina al examinar el grado en el que "el conjunto de propiedades que posee es parte del problema o dominio de diseño" [Whi97].

Primitivismo. Una característica que es similar a la simplicidad, el primitivismo (aplicado tanto a operaciones como a clases), es el grado en el que una operación es atómica, es decir, la operación no puede construirse a partir de una secuencia de otras operaciones contenidas dentro de una clase. Una clase que muestra un alto grado de primitivismo encapsula sólo operaciones primitivas.

Similitud. El grado en el que dos o más clases son similares en términos de su estructura, función, comportamiento o propósito se indica mediante esta medida.

Volatilidad. Como se menciona muchas veces en este libro, los cambios en el diseño pueden ocurrir cuando se modifican los requerimientos o cuando ocurren modificaciones en otras partes de una aplicación, lo que da como resultado la adaptación obligatoria del componente de diseño en cuestión. La volatilidad de un componente de diseño OO mide la probabilidad de que ocurrirá un cambio.

En realidad, las métricas de producto para sistemas OO pueden aplicarse no sólo al modelo de diseño, sino también al de requerimientos. En las siguientes secciones se estudian las métricas

que proporcionan un indicio de la calidad en el nivel de clase OO y en el de operación. Además, también se exploran las métricas aplicables a la administración del proyecto y a las pruebas.

23.3.3 Métricas orientadas a clase: la suite de métricas CK

La clase es la unidad fundamental de un sistema OO. Por tanto, las medidas y métricas para una clase individual, la jerarquía de clase y las colaboraciones de clase serán invaluables cuando se requiera valorar la calidad del diseño OO. Una clase encapsula datos y la función que los manipula. Con frecuencia es el "padre" de las subclases (en ocasiones llamadas hijos) que heredan sus atributos y operaciones. Usualmente colabora con otras clases. Cada una de estas características puede usarse como la base para la medición.[7]

Chidamber y Kemerer propusieron uno de los conjuntos de métricas de software OO de mayor referencia [Chi94]. En ocasiones llamada *suite de métricas CK*, los autores proponen seis métricas de diseño basadas en clase para sistemas OO.[8]

Métodos ponderados por clase (MPC). Suponga que n métodos de complejidad c_1, c_2, ..., c_n se definen para una clase **C**. La métrica de complejidad específica que se elige (por ejemplo, complejidad ciclomática) debe normalizarse de modo que la complejidad nominal para un método tome un valor de 1.0.

$$\text{MPC} = \Sigma c_i$$

para $i = 1$ hasta n. El número de métodos y su complejidad son indicadores razonables de la cantidad de esfuerzo requerido para implementar y probar una clase. Además, mientras más grande sea el número de métodos, más complejo será el árbol de herencia (todas las subclases heredan los métodos de sus padres). Finalmente, conforme el número de métodos crece para una clase determinada, es probable que se vuelva cada vez más específica su aplicación y, por tanto, limite la potencial reutilización. Por todas estas razones, MPC debe mantenerse tan bajo como sea razonable.

Aunque parecería relativamente directo desarrollar un conteo para el número de métodos en una clase, el problema en realidad es más complejo de lo que parece. Debe desarrollarse un enfoque de conteo consistente [Chu95].

Profundidad del árbol de herencia (PAH). Esta métrica es "la máxima longitud desde el nodo hasta la raíz del árbol" [Chi94]. En la figura 23.5, el valor de la PAH para la jerarquía de clase que se muestra es 4. Conforme crece la PAH, es probable que las clases de nivel inferior hereden muchos métodos. Esto conduce a potenciales dificultades cuando se intenta predecir el comportamiento de una clase. Una jerarquía de clase profunda (PAH grande) también conduce a mayor complejidad de diseño. En el lado positivo, grandes valores de PAH implican que muchos métodos pueden reutilizarse.

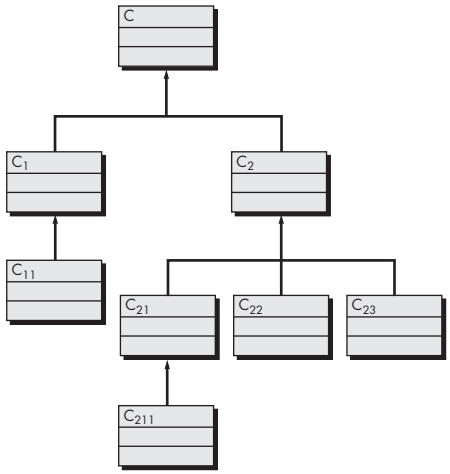
Número de hijos (NDH). Las subclases que son inmediatamente subordinadas a una clase en la jerarquía de clase se denominan hijos. En la figura 23.5, la clase $\mathbf{C_2}$ tiene tres hijos: subclases $\mathbf{C_{21}}$, $\mathbf{C_{22}}$ y $\mathbf{C_{23}}$. Conforme crece el número de hijos, el reuso aumenta, pero también, como el NDH aumenta, la abstracción representada por la clase padre puede diluirse si algunos de los hijos no son miembros adecuados de la clase padre. Conforme el NDH aumenta, la cantidad de pruebas (requeridas para ejercitar cada hijo en su contexto operativo) también aumentará.

---

7 Debe observarse que, en la literatura técnica, actualmente está en debate la validez de algunas de las métricas estudiadas en este capítulo. Quienes defienden la teoría de mediciones demandan un grado de formalismo que algunas métricas OO no proporcionan. Sin embargo, es razonable afirmar que las métricas mencionadas ofrecen comprensión útil para el ingeniero de software.

8 Chidamber, Darcy y Kemerer usan el término *métodos* en lugar de *operaciones*. El uso de este término se refleja en esta sección.

FIGURA 23.5
Una jerarquía de clase



Los conceptos de acoplamiento y cohesión se aplican tanto a software convencional como a OO. Mantenga el acoplamiento de clase bajo y la cohesión de clase y operación alta.

Acoplamiento entre clases de objetos (ACO). El modelo CRC (capítulo 6) puede usarse para determinar el valor para el ACO. En esencia, ACO es el número de colaboraciones citadas para una clase en su tarjeta índice CRC.[9] Conforme el ACO aumenta, es probable que la reusabilidad de una clase disminuya. Valores altos de ACO también complican las modificaciones y las pruebas que sobrevienen cuando se realizan modificaciones. En general, los valores de ACO para cada clase deben mantenerse tan bajos como sea razonable. Esto es consistente con el lineamiento general para reducir el acoplamiento en el software convencional.

Respuesta para una clase (RPC). Respuesta para una clase es "un conjunto de métodos que potencialmente pueden ejecutarse en respuesta a un mensaje recibido por un objeto de dicha clase" [Chi94]. RPC es el número de métodos en el conjunto respuesta. Conforme aumenta la RPC, también lo hace el esfuerzo requerido para probar, pues la secuencia de pruebas (capítulo 19) crece. Igualmente, se sigue que, conforme la RPC aumenta, la complejidad de diseño global de la clase aumenta.

Falta de cohesión en métodos (FCOM). Cada método dentro de una clase **C** accede a uno o más atributos (también llamados variables de instancia). FCOM es el número de métodos que acceden a uno o más de los mismos atributos.[10] Si ningún método accede a los mismos atributos, entonces la FCOM = 0. Para ilustrar el caso donde la FCOM ≠ 0, considere una clase con seis métodos. Cuatro de ellos tienen uno o más atributos en común (es decir, acceden a atributos comunes). Por tanto, la FCOM = 4. Si la FCOM es alta, los métodos pueden acoplarse unos con otros mediante atributos. Esto aumenta la complejidad del diseño de clase. Aunque hay casos en los que un valor alto de la FCOM es justificable, es deseable mantener alta la cohesión, es decir, mantener baja la FCOM.[11]

9 Si las tarjetas índice CRC se desarrollan manualmente, completitud y consistencia deben valorarse antes de que el ACO pueda determinarse de manera confiable.

10 La definición formal es un poco más compleja. Vea [Chi94] para detalles.

11 La métrica FCOM proporciona útil comprensión en algunas situaciones, pero puede confundir en otras. Por ejemplo, mantener en acoplamiento encapsulado dentro de una clase aumenta la cohesión del sistema como un todo. Por tanto, en al menos un sentido importante, la FCOM más alta en realidad sugiere que una clase puede tener mayor cohesión, no menor.

CasaSegura

Aplicación de métricas CK

La escena: Cubículo de Vinod.

Participantes: Vinod, Jamie, Shakira y Ed, miembros del equipo de ingeniería del software *CasaSegura*, quienes continúan trabajando en el diseño en el nivel de componentes y en el diseño de casos de prueba.

La conversación:

Vinod: ¿Alguno de ustedes tuvo oportunidad de leer la descripción de la suite de métricas CK que envié el miércoles e hizo las mediciones?

Shakira: No fue muy complicado. Regresé a mi clase UML y a diagramas de secuencia, como sugeriste, y obtuve conteos burdos para PAH, RPC y FCOM. No pude encontrar el modelo CRC, así que no conté ACO.

Jamie (sonríe): No pudiste encontrar el modelo CRC porque yo lo tengo.

Shakira: Por eso adoro a este equipo: comunicación soberbia.

Vinod: Yo hice mis conteos... ¿ustedes desarrollaron números para las métricas CK?

[Jamie y Ed asienten.]

Jamie: Dado que yo tenía las tarjetas CRC, eché un vistazo al ACO y parecía bastante uniforme a través de la mayoría de las clases. Hubo una excepción, la cual anoté.

Ed: Hay algunas clases donde la RPC es muy alta, en comparación con los promedios... tal vez debamos echar un vistazo para simplificarlos.

Jamie: Quizá sí, quizá no. Todavía estoy preocupado por el tiempo, y no quiero componer cosas que en realidad no están rotas.

Vinod: Estoy de acuerdo. Tal vez debas buscar clases que tengan malos números en al menos dos o más de las métricas CK. Cuestión de dos strikes y estás modificado.

Shakira [observa la lista de clases de Ed con alta RPC]: Mira, ve estas clases, tienen alta FCOM así como alta RPC. ¿Dos strikes?

Vinod: Sí, eso creo... será difícil implementar debido a la complejidad y dificultad para probar por la misma razón. Probablemente valdría la pena diseñar dos clases separadas para lograr el mismo comportamiento.

Jamie: ¿Crees que modificarlo nos ahorrará tiempo?

Vinod: A largo plazo, sí.

23.3.4 Métricas orientadas a clase: La suite de métricas MOOD

Cita:

"Analizar el software OO para evaluar su calidad se ha vuelto cada vez más importante conforme el paradigma [OO] continúa aumentando en popularidad."

Rachel Harrison *et al.*

Harrison, Counsell y Nithi [Har98b] proponen un conjunto de métricas para diseño orientado a objeto que proporciona indicadores cuantitativos para características de diseño OO. A continuación se presenta un muestreo de métricas MOOD.

Factor de herencia de método (FHM). El grado en el que la arquitectura de clase de un sistema OO utiliza la herencia tanto para métodos (operaciones) como para atributos se define como

$$\text{FHM} = \frac{\Sigma M_i (C_i)}{\Sigma M_a (C_i)}$$

donde la suma ocurre sobre $i = 1$ hasta TC. TC se define como el número total de clases en la arquitectura, C_i es una clase dentro de la arquitectura y

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

donde

$M_a(C_i)$ = número de métodos que pueden invocarse en asociación con C_i
$M_d(C_i)$ = número de métodos declarados en la clase C_i
$M_i(C_i)$ = número de métodos heredados (y no invalidados) en C_i

El valor de FHM [el factor de herencia de atributo (FHA) se define de forma análoga] ofrece un indicio del impacto de la herencia sobre el software OO.

Factor de acoplamiento (FA). Anteriormente, en este capítulo, se dijo que el acoplamiento es un indicio de las conexiones entre elementos del diseño OO. La suite de métricas MOOD define el acoplamiento en la forma siguiente:

$$\text{FA} = \Sigma_i \Sigma_j \, is_client \, \frac{(C_i, C_j)}{T_c^2 - T_c}$$

donde las sumas ocurren sobre $i = 1$ hasta T_c y $j = 1$ hasta T_c. La función

$$\begin{aligned} \textit{is_client} \text{ (es cliente)} &= 1, \text{ si y sólo si existe una relación entre la clase cliente } C_c \text{ y la clase servidor } C_s, \text{ y } C_c \neq C_s \\ &= 0, \text{ de otro modo} \end{aligned}$$

Aunque muchos factores afectan la complejidad, comprensibilidad y mantenimiento del software, es razonable concluir que, conforme el valor de FA aumenta, la complejidad del software OO también aumentará, y como resultado pueden sufrir la comprensibilidad, el mantenimiento y el potencial de reuso.

Harrison *et al.* [Har98b] presentan un análisis detallado de FHM y FA junto con otras métricas, y examinan su validez para usarlas en la valoración de la calidad del diseño.

23.3.5 Métricas OO propuestas por Lorenz y Kidd

En su libro acerca de métricas OO, Lorenz y Kidd [Lor94] dividen las métricas basadas en clase en cuatro amplias categorías; cada una tiene una relación en el diseño en el nivel de componentes: tamaño, herencia, internos y externos. Las métricas orientadas a tamaño para una clase de diseño OO se enfocan en conteos de atributos y operaciones para una clase individual y en valores promedio para el sistema OO como un todo. Las métricas basadas en herencia se enfocan en la forma en la que las operaciones se reutilizan a lo largo de la jerarquía de clases. Las métricas para interiores de clase se fijan en la cohesión (sección 23.3.3) y en los conflictos orientados a código. Y las métricas externas examinan el acoplamiento y el reuso. Un ejemplo de las métricas propuestas por Lorenz y Kidd es:

Tamaño de clase (TDC). El tamaño global de una clase puede determinarse usando las siguientes medidas:

- El número total de operaciones (tanto heredadas como operaciones de instancia privada) que se encapsulan dentro de la clase.
- El número de atributos (tanto heredados como de instancia privada) que encapsula la clase.

Durante la revisión del modelo de análisis, las tarjetas índice CRC proporcionarán un indicio razonable de los valores esperados para TDC. Si encuentra una clase con un gran número de responsabilidades, considere dividirla.

La métrica MPC propuesta por Chidamber y Kemerer (sección 23.3.3) también es una medida ponderada del tamaño de clase. Como se indicó anteriormente, grandes valores para TDC indican que una clase puede tener demasiada responsabilidad. Esto reducirá la reutilización de la clase y complicará la implementación y las pruebas. En general, las operaciones y atributos heredados o públicos deben ponderarse más para determinar el tamaño de clase [Lor94]. Las operaciones y atributos privados permiten la especialización y están más localizadas en el diseño. También pueden calcularse promedios para el número de atributos y operaciones de clase. Mientras más bajos sean los valores promedio para el tamaño, hay más probabilidad de que las clases dentro del sistema puedan reutilizarse ampliamente.

23.3.6 Métricas de diseño en el nivel de componente

Las métricas de diseño en el nivel de componente para componentes de software convencional se enfocan en las características internas de un componente de software e incluyen medidas de cohesión de módulo, acoplamiento y complejidad. Dichas medidas pueden ayudarlo a juzgar la calidad de un diseño en el nivel de componente.

Las métricas de diseño en el nivel de componente pueden aplicarse una vez desarrollado el diseño procedural y son "cajas de cristal" en tanto requieren conocimiento del funcionamiento interior del módulo en el que se está trabajando. Alternativamente, pueden demorarse hasta que el código fuente esté disponible.

Es posible calcular medidas de la independencia funcional (acoplamiento y cohesión) de un componente y usarlas para valorar la calidad de un diseño.

Métricas de cohesión. Bieman y Ott [Bie94] definen una colección de métricas que proporcionan un indicio de la cohesión (capítulo 8) de un módulo. Las métricas se definen mediante cinco conceptos y medidas:

Rebanada de datos (*data slice*). Dicho de manera simple, una rebanada de datos es una marcha hacia atrás a través de un módulo que busca valores de datos que afecten la ubicación del módulo donde comenzó la marcha. Debe observarse que es posible definir tanto las rebanadas de programa (que se enfocan en enunciados y condiciones) como las rebanadas de datos.

Símbolos de datos (*data tokens*). Las variables definidas por un módulo pueden definirse como *tokens* de datos para el módulo.

Símbolos pegamento (*glue tokens*). Este conjunto de *tokens* de datos se encuentra en una o más rebanadas de datos.

Símbolos superpegamento (*superglue tokens*). Estos *tokens* de datos son comunes a cada rebanada de datos en un módulo.

Pegajosidad (*stickiness*). La pegajosidad relativa de un *token* pegamento es directamente proporcional al número de rebanadas de datos que enlaza.

Bieman y Ott desarrollan métricas para *cohesión funcional fuerte* (CFF), *cohesión funcional débil* (CFD) y *adhesividad* (el grado relativo en el que los *tokens* pegamento enlazan rebanadas de datos). Un análisis detallado de las métricas de Bieman y Ott, se deja a los autores [Bie94].

Métricas de acoplamiento. El acoplamiento de módulo proporciona un indicio de cuán "conectado" está un módulo con otros módulos, con datos globales y con el entorno exterior. En el capítulo 9 se estudió el acoplamiento en términos cualitativos.

Dhama [Dha95] propuso una métrica para acoplamiento de módulo que abarca acoplamiento de datos y flujo de control, acoplamiento global y acoplamiento ambiental. Las medidas requeridas para calcular acoplamiento de módulo se definen en función de cada uno de los tres tipos de acoplamiento anotados anteriormente.
Para acoplamiento de datos y flujo de control,

d_i = número de parámetros de datos de entrada
c_i = número de parámetros de control de entrada
d_o = número de parámetros de datos de salida
c_o = número de parámetros de control de salida

Para acoplamiento global,

g_d = número de variables globales usadas como datos
g_c = número de variables globales usadas como control

Para acoplamiento de entorno,

w = número de módulos llamados (*fan-out*)
r = número de módulos que llaman al módulo bajo consideración (*fan-in*)

Al usar estas medidas, un indicador de acoplamiento de módulo m_c se define de la forma siguiente:

$$m_c = \frac{k}{M}$$

donde k es una constante de proporcionalidad y

$$M = d_i + (a \times c_i) + d_o + (b \times c_o) + g_d + (c \times g_c) + w + r$$

Los valores para k, a, b y c deben derivarse de manera empírica.

Conforme el valor de m_c crece, el acoplamiento de módulo global disminuye. Con la finalidad de que la métrica de acoplamiento se mueva hacia arriba conforme el grado de acoplamiento aumenta, una métrica de acoplamiento revisada puede definirse como

$$C = 1 - m_c$$

donde el grado de acoplamiento aumenta conforme el valor de *M* aumenta.

Métricas de complejidad. Para determinar la complejidad del flujo de control de programa, pueden calcularse varias métricas de software. Muchas de éstas se basan en el gráfico de flujo. Un gráfico (capítulo 18) es una representación compuesta de nodos y ligas (también llamadas aristas). Cuando las ligas (aristas) se dirigen, el gráfico de flujo es un gráfico dirigido.

McCabe y Watson [McC94] identifican algunos usos importantes para las métricas de complejidad:

> Las métricas de complejidad pueden usarse para predecir información crucial acerca de la confiabilidad y el mantenimiento de los sistemas de software a partir de análisis automáticos de código fuente [o información de diseño procedimental]. Las métricas de complejidad también proporcionan retroalimentación durante el proyecto de software para ayudar a controlar la [actividad de diseño]. Durante las pruebas y el mantenimiento, proporcionan información detallada acerca de los módulos de software para ayudar a destacar áreas de potencial inestabilidad.

La métrica de complejidad para software de computadora más ampliamente usada es la complejidad ciclomática, originalmente desarrollada por Thomas McCabe [McC76] y que se estudió en detalle en el capítulo 18.

PUNTO CLAVE
La complejidad ciclomática es sólo una entre un gran número de métricas de complejidad.

Zuse ([Zus90], [Zus97]) presenta una discusión enciclopédica de no menos de 18 diferentes categorías de métricas de complejidad de software. El autor expone las definiciones básicas para las métricas en cada categoría (por ejemplo, existen algunas variaciones en la métrica de complejidad ciclomática) y luego analiza y critica cada una. El trabajo de Zuse es el más exhaustivo publicado a la fecha.

23.3.7 Métricas orientadas a operación

Puesto que la clase es la unidad dominante en los sistemas OO, se han propuesto menos métricas para operaciones que residen dentro de una clase. Churcher y Shepperd [Chu95] analizan esto cuando afirman: "Los resultados de estudios recientes indican que los métodos tienden a ser pequeños, tanto en términos de número de enunciados como en complejidad lógica [Wil93], lo que sugiere que la estructura de conectividad de un sistema puede ser más importante que el contenido de los módulos individuales." Sin embargo, puede obtenerse algo de comprensión al examinar las características promedio para los métodos (operaciones). Tres métricas simples, propuestas por Lorenz y Kidd [Lor94], son apropiadas:

Tamaño promedio de operación (TO_{prom}). El tamaño puede determinarse al contar el número de líneas de código o el de mensajes enviados por la operación. Conforme aumenta el número de mensajes enviados por una sola operación, es probable que las responsabilidades no se hayan asignado bien dentro de una clase.

Complejidad de la operación (CO). La complejidad de una operación puede calcularse usando cualquiera de las métricas de complejidad propuestas para software convencional [Zus90]. Puesto que las operaciones deben limitarse a una responsabilidad específica, el diseñador debe luchar por mantener la CO tan baja como sea posible.

Número promedio de parámetros por operación (NP_{prom}). Mientras más grande sea el número de parámetros de operación, más compleja es la colaboración entre objetos. En general, el NP_{prom} debe mantenerse tan bajo como sea posible.

23.3.8 Métricas de diseño de interfaz de usuario

Cita:

"Es posible aprender al menos un principio del diseño de interfaz de usuario al cargar una lavadora de platos. Si apila muchos en ella, nada quedará muy limpio."

Autor desconocido

Aunque hay considerable literatura acerca del diseño de interfaces hombre/computadora (capítulo 11), se ha publicado relativamente poca información acerca de las métricas que proporcionarían comprensión de la calidad y de la usabilidad de la interfaz.

Sears [Sea93] sugiere que la *corrección de la plantilla* (CP) es una métrica de diseño valioso para las interfaces hombre/computadora. Una GUI típica usa entidades de plantilla (íconos gráficos, texto, menús, ventanas y similares) para auxiliar al usuario a completar tareas. Para lograr una tarea dada usando una GUI, el usuario debe moverse de una entidad de plantilla a la siguiente. La posición absoluta y relativa de cada entidad de plantilla, la frecuencia con la que se usa y el "costo" de la transición desde una entidad de plantilla a la siguiente contribuirán a la corrección de la interfaz.

Un estudio de métricas de página web [Ivo01] indica que las características simples de los elementos de la plantilla también pueden tener un impacto significativo sobre la calidad percibida del diseño GUI. El número de palabras, vínculos, gráficos, colores y fuentes (entre otras características) contenidas en una página web afectan la complejidad percibida y la calidad de dicha página.

Es importante observar que la selección de un diseño GUI puede guiarse con métricas como CP, pero el árbitro final debe ser la entrada del usuario con base en los prototipos GUI. Nielsen y Levy [Nie94] reportan que "uno tiene una oportunidad razonablemente grande de triunfar si se elige entre [diseños de] interfaz basadas exclusivamente en opiniones de los usuarios. El rendimiento de tarea promedio de los usuarios y su satisfacción subjetiva con una GUI están enormemente correlacionados".

23.4 MÉTRICAS DE DISEÑO PARA *WEBAPPS*

Un útil conjunto de medidas y métricas para *webapps* proporciona respuestas cuantitativas a las siguientes preguntas:

- ¿La interfaz de usuario promueve usabilidad?
- ¿La estética de la *webapp* es apropiada para el dominio de aplicación y agrada al usuario?
- ¿El contenido se diseñó de tal forma que imparte más información con menos esfuerzo?
- ¿La navegación es eficiente y directa?
- ¿La arquitectura de la *webapp* se diseñó para alojar las metas y objetivos especiales de los usuarios de la *webapp*, la estructura de contenido y funcionalidad, y el flujo de navegación requerido para usar el sistema de manera efectiva?
- ¿Los componentes se diseñaron de manera que se reduce la complejidad procedimental y se mejora la exactitud, la confiabilidad y el desempeño?

CONSEJO

Muchas de estas métricas son aplicables a todas las interfaces de usuario y deben considerarse en conjunto con las que se presentaron en la sección 23.3.8.

En la actualidad, cada una de estas preguntas puede abordarse sólo de manera cualitativa, porque todavía no existe una suite validada de métricas que proporcionen respuestas cuantitativas.

En los siguientes párrafos se presenta una muestra representativa de métricas de diseño *webapp* que se han propuesto en la literatura. Es importante observar que muchas de ellas todavía no se validan, por lo que deben usarse juiciosamente.

Métricas de interfaz. Para *webapps* pueden considerarse las siguientes medidas:

Métrica sugerida	*Descripción*
Corrección de plantilla	Ver sección 23.3.8
Complejidad de plantilla	Número de regiones[12] distintas definidas por una interfaz
Complejidad de región de plantilla	Número promedio de distintos vínculos por región
Complejidad de reconocimiento	Número promedio de distintos ítems que el usuario debe buscar antes de realizar una navegación o decidir la entrada de datos
Tiempo de reconocimiento	Tiempo promedio (en segundos) que tarda un usuario en seleccionar la acción adecuada para una tarea determinada
Esfuerzo de escritura	Número promedio de golpes de tecla requeridos para una función específica
Esfuerzo de toma de ratón	Número promedio de tomas de ratón por función
Complejidad de selección	Número promedio de vínculos que pueden seleccionarse por página
Tiempo de adquisición de contenido	Número promedio de palabras de texto por página web
Carga de memoria	Número promedio de distintos ítems de datos que el usuario debe recordar para lograr un objetivo específico

Métricas estéticas (diseño gráfico). Por su naturaleza, el diseño estético se apoya en el juicio cualitativo y por lo general no es sensible a la medición ni a las métricas. Sin embargo, Ivory *et al.* [Ivo01] proponen un conjunto de medidas que pueden ser útiles para valorar el impacto del diseño estético:

Métrica sugerida	*Descripción*
Conteo de palabra	Número total de palabras que aparecen en una página
Porcentaje de texto de cuerpo	Porcentaje de palabras que son cuerpo frente a texto de despliegue (es decir, títulos)
% texto cuerpo enfatizado	Porción de texto de cuerpo que se enfatiza (por ejemplo, negrillas, mayúsculas)
Conteo de posicionamiento de texto	Cambios en posición de texto desde el alineado a la izquierda
Conteo de grupo de texto	Áreas de texto resaltadas con color, regiones con bordes, reglas o listas
Conteo de vínculos	Vínculos totales en una página
Tamaño de página	Bytes totales para la página, así como elementos, gráficos y hojas de estilo
Porcentaje gráfico	Porcentaje de bytes de página que son usados para gráficos
Conteo gráfico	Gráficos totales en una página (no incluye gráficos especificados en guiones, applets y objetos)
Conteo de color	Total de colores empleados
Conteo de fuente	Total de fuentes empleadas (es decir, tipo + tamaño + negrilla + itálica)

Métricas de contenido. Las métricas en esta categoría se enfocan en la complejidad del contenido y en los grupos de objetos de contenido que se organizan en páginas [Men01].

Métrica sugerida	*Descripción*
Espera de página	Tiempo promedio requerido para que una página se descargue a diferentes velocidades de conexión
Complejidad de página	Número promedio de tipos diferentes de medios usados en la página, no incluido el texto

12 Una región distinta es un área dentro de la plantilla de despliegue que logra cierto conjunto específico de funciones relacionadas (por ejemplo, una barra de menú, un despliegue gráfico estático, un área de contenido, un despliegue animado).

Complejidad gráfica	Número promedio de medios gráficos por página
Complejidad de audio	Número promedio de medios de audio por página
Complejidad de video	Número promedio de medios de video por página
Complejidad de animación	Número promedio de animaciones por página
Complejidad de imagen escaneada	Número promedio de imágenes escaneadas por página

Métricas de navegación. Las métricas en esta categoría abordan la complejidad del flujo de navegación [Men01]. En general, son útiles sólo para aplicaciones web estáticas, que no incluyen vínculos y páginas generados de manera dinámica.

Métrica sugerida	*Descripción*
Complejidad de vinculación de página	Número de vínculos por página
Conectividad	Número total de vínculos internos, no incluidos vínculos generados de manera dinámica
Densidad de conectividad	Conectividad dividida por conteo de página

Usar un subconjunto de las métricas sugeridas puede servir para derivar relaciones empíricas que permiten a un equipo de desarrollo *webapp* valorar la calidad técnica y predecir el esfuerzo con base en las estimaciones de complejidad estimadas. En esta área todavía queda mucho trabajo por hacer.

HERRAMIENTAS DE SOFTWARE

Métricas técnicas para webapps

Objetivo: Auxiliar a los ingenieros web a desarrollar métricas *webapp* significativas que proporcionen comprensión acerca de la calidad global de una aplicación.

Mecánica: La mecánica de las herramientas varía.

Herramientas representativas:[13]

Netmechanic Tools, desarrollada por Netmechanic (**www.netmechanic.com**), es una colección de herramientas que ayudan a mejorar el desempeño de sitios web y que se enfocan en conflictos específicos de implementación.

NIST Web Metrics Testbed, desarrollada por The National Institute of Standards and Technology (**zing.ncsl.nist.gov/WebTools/**), abarca la siguiente colección de útiles herramientas que están disponibles para descarga:

Web Static Analyzer Tool (WebSAT): comprueba el HTML de la página web contra lineamientos de usabilidad típicos.

Web Category Analysis Tool (WebCAT): permite que el ingeniero de usabilidad construya y realice un análisis de categoría web.

Web Variable Instrumenter Program (WebVIP): instrumenta un sitio web para capturar una bitácora de interacción con el usuario.

Framework for Logging Usability Data (FLUD): implementa un formateador de archivo y analizador gramatical para representación de las bitácoras de interacción del usuario.

VisVIP Tool: produce una visualización 3D de las rutas de navegación del usuario a través de un sitio web.

TreeDec: agrega auxiliares de navegación a las páginas de un sitio web.

23.5 MÉTRICAS PARA CÓDIGO FUENTE

La teoría de Halstead de la "ciencia del software" [Hal77] propuso las primeras "leyes" analíticas para el software de computadora.[14] Halstead asignó leyes cuantitativas al desarrollo de software

13 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría.

14 Debe observarse que las "leyes" de Halstead generaron gran controversia, y muchos creen que la teoría subyacente tiene fallas. Sin embargo, se ha realizado verificación experimental para lenguajes de programación seleccionados (por ejemplo, [Fel89]).

Cita:

"El cerebro humano sigue un conjunto de reglas más rígido [para desarrollo de algoritmos] del que se tiene conocimiento."

Maurice Halstead

de computadora, usando un conjunto de medidas primitivas que pueden derivarse después de generar el código o de que el diseño esté completo. Las medidas son:

n_1 = número de operadores distintos que aparecen en un programa
n_2 = número de operandos distintos que aparecen en un programa
N_1 = número total de ocurrencias de operador
N_2 = número total de ocurrencias de operando

Halstead usa estas medidas primitivas para desarrollar: expresiones para la longitud de programa global, volumen mínimo potencial para un algoritmo, volumen real (número de bits requeridos para especificar un programa), nivel del programa (una medida de complejidad del software), nivel del lenguaje (una constante para un lenguaje determinado) y otras características, como esfuerzo de desarrollo, tiempo de desarrollo e incluso el número proyectado de fallas en el software.

Halstead muestra que la longitud N puede estimarse

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

y el volumen del programa puede definirse

$$V = N \log_2 (n_1 + n_2)$$

Debe observarse que V variará con el lenguaje de programación y representa el volumen de información (en bits) requerido para especificar un programa.

CONSEJO

Los operadores incluyen todo el flujo de constructos de control, condicionales y operaciones matemáticas. Los operandos abarcan todas las variables y constantes de programa.

Teóricamente, debe existir un volumen mínimo para un algoritmo particular. Halstead define una razón de volumen L como la razón del volumen de la forma más compacta de un programa al volumen del programa real. En realidad, L siempre debe ser menor que 1. En términos de medidas primitivas, la razón de volumen puede expresarse como

$$L = \frac{2}{n_1} \times \frac{n_2}{N_2}$$

El trabajo de Halstead es sensible a la verificación experimental y se ha llevado a cabo un gran trabajo para investigar la ciencia del software. Un análisis de este trabajo está más allá del ámbito de este libro. Para mayor información, vea [Zus90], [Fen91] y [Zus97].

23.6 MÉTRICAS PARA PRUEBAS

PUNTO CLAVE

Las métricas de prueba se ubican en dos amplias categorías: 1) métricas que intentan predecir el número probable de pruebas requeridas en varios niveles de prueba y 2) métricas que se enfocan en la cobertura de pruebas para un componente determinado.

Aunque se ha escrito mucho acerca de las métricas de software para pruebas (por ejemplo, [Het93]), la mayoría de las métricas proponen enfocarse en el proceso de las pruebas, no en las características técnicas de las pruebas en sí. En general, los examinadores deben apoyarse en las métricas de análisis, diseño y código para guiarlos en el diseño y la ejecución de los casos de prueba.

Las métricas del diseño arquitectónico proporcionan información acerca de la facilidad o dificultad asociada con las pruebas de integración (sección 23.3) y de la necesidad de software de pruebas especializado (por ejemplo, resguardos y controladores). La complejidad ciclomática (una métrica de diseño en el nivel de componente) yace en el centro de la prueba de ruta base, un método de diseño de casos de prueba que se presentó en el capítulo 18. Además, la complejidad ciclomática puede usarse para dirigirse a módulos como candidatos para prueba de unidad extensa. Los módulos con alta complejidad ciclomática tienen más probabilidad de ser proclives al error que los módulos donde su complejidad ciclomática es menor. Por esta razón, debe emplear esfuerzo por arriba del promedio para descubrir errores en tales módulos antes de que se integren en un sistema.

23.6.1 Métricas de Halstead aplicadas para probar

El esfuerzo de prueba puede estimarse usando métricas derivadas de las medidas de Halstead (sección 23.5). Al usar las definiciones para volumen de programa *V* y nivel de programa *PL*, el esfuerzo *e* de Halstead puede calcularse como

$$PL = \frac{1}{(n_1/2) \times (N_2/n_2)} \tag{23.2a}$$

$$e = \frac{V}{PL} \tag{23.2b}$$

El porcentaje de esfuerzo de prueba global que se va a asignar a un módulo *k* puede estimarse usando la siguiente relación:

$$\text{Porcentaje de esfuerzo de prueba } (k) = \frac{e(k)}{\Sigma e(i)} \tag{23.3}$$

donde $e(k)$ se calcula para el módulo *k* usando las ecuaciones (23.2), y la suma en el denominador de la ecuación (23.3) es la suma del esfuerzo de Halstead a través de todos los módulos del sistema.

23.6.2 Métricas para pruebas orientadas a objetos

Las métricas del diseño OO anotadas en la sección 23.3 proporcionan un indicio de la calidad del diseño. También ofrecen un indicio general de la cantidad de esfuerzo de prueba requerido para ejercitar un sistema OO. Binder [Bin94b] sugiere un amplio arreglo de métricas de diseño que tienen influencia directa sobre la "comprobabilidad" de un sistema OO. Las métricas consideran aspectos de encapsulación y herencia.

Falta de cohesión en métodos (FCOM).[15] Mientras más alto sea el valor de la FCOM, más estados deben ponerse a prueba para garantizar que los métodos no generan efectos colaterales.

Las pruebas OO pueden ser bastante complejas. Las métricas pueden ayudarle a dirigir los recursos de prueba en hebras, escenarios y paquetes de clases que son "sospechosas" con base en las características medidas. Úselas.

Porcentaje público y protegido (PPP). Los atributos públicos se heredan de otras clases y, por tanto, son visibles para dichas clases. Los atributos protegidos son accesibles a los métodos en las subclases. Esta métrica indica el porcentaje de los atributos de clase que son públicos o protegidos. Valores altos de PPP aumentan la probabilidad de efectos colaterales entre las clases porque los atributos públicos y protegidos conducen a alto potencial para acoplamiento.[16] Las pruebas deben diseñarse para garantizar el descubrimiento de tales efectos colaterales.

Acceso público a miembros de datos (APD). Esta métrica indica el número de clases (o métodos) que pueden acceder a otros atributos de clase, una violación de la encapsulación. Valores altos de APD conducen al potencial de efectos colaterales entre clases. Las pruebas deben diseñarse para garantizar el descubrimiento de tales efectos colaterales.

Número de clases raíz (NCR). Esta métrica es un conteo de las distintas jerarquías de clase que se describen en el modelo de diseño. Deben desarrollarse las suites de prueba para cada clase raíz y la correspondiente jerarquía de clase. Conforme el NCR aumenta, también aumenta el esfuerzo de prueba.

***Fan-in* (FIN).** Cuando se usa en el contexto OO, el *fan-in* (abanico de entrada) en la jerarquía de herencia es un indicio de herencia múltiple. FIN > 1 indica que una clase hereda sus atributos y operaciones de más de una clase raíz. FIN > 1 debe evitarse cuando sea posible.

15 Vea la sección 23.3.3 para una descripción de FCOM.

16 Algunas personas promueven diseños sin que alguno de los atributos sea público o privado, es decir, PPP = 0. Esto implica que todos los atributos deben valorarse en otras clases mediante métodos.

Número de hijos (NDH) y profundidad del árbol de herencia (PAH).[17] Como se mencionó en el capítulo 19, los métodos de superclase tendrán que volverse a probar para cada subclase.

23.7 MÉTRICAS PARA MANTENIMIENTO

Todas las métricas de software presentadas en este capítulo pueden usarse para el desarrollo de nuevo software y para el mantenimiento del software existente. Sin embargo, se han propuesto métricas diseñadas explícitamente para actividades de mantenimiento.

IEEE Std. 982.1-1988 [IEE93] sugiere un *índice de madurez de software* (IMS) que proporcione un indicio de la estabilidad de un producto de software (con base en cambios que ocurran para cada liberación del producto). Para ello, se determina la siguiente información:

M_T = número de módulos en la liberación actual
F_c = número de módulos en la liberación actual que cambiaron
F_a = número de módulos en la liberación actual que se agregaron
F_d = número de módulos de la liberación anterior que se borraron en la liberación actual

El índice de madurez del software se calcula de la forma siguiente:

$$\text{IMS} = \frac{M_T - (F_a + F_c + F_d)}{M_T}$$

Conforme el IMS tiende a 1.0, el producto comienza a estabilizarse. El IMS también puede usarse como una métrica para planificar actividades de mantenimiento de software. El tiempo medio para producir una liberación de un producto de software puede correlacionarse con el IMS, y es posible desarrollar modelos empíricos para esfuerzo de mantenimiento.

HERRAMIENTAS DE SOFTWARE

Métricas de producto

Objetivo: Auxiliar a los ingenieros del software a desarrollar métricas significativas que valoren los productos operativos producidos durante el modelado de análisis y diseño, la generación de código fuente y las pruebas.

Mecánica: Las herramientas en esta categoría abarcan un amplio abanico de métricas y se implementan como una aplicación independiente o (más comúnmente) como funcionalidad que existe dentro de las herramientas para análisis y diseño, codificación o pruebas. En la mayoría de los casos, las herramientas de métricas analizan una representación del software (por ejemplo, un modelo UML o código fuente) y desarrollan como resultado una o más métricas.

Herramientas representativas:[18]

Krakatau Metrics, desarrollada por Power Software (**www.power-software.com/products**), calcula métricas de complejidad, Halstead y otras relacionadas para C/C++ y Java.

Metrics4C, desarrollada por +1 Software Engineering (**www.plus-one.com/Metrics4C_fact_sheet.html**), calcula una variedad de métricas arquitectónicas, de diseño y orientadas a código, así como métricas orientadas a proyecto.

Rational Rose, distribuida por IBM (**www.304.ibm.com/jct03001c/software/awdtools/developer/rose/**), es un amplio conjunto de herramientas para modelado UML que incorpora algunas características de análisis de métricas.

RSM, desarrollada por M-Squared Technologies (**msquaredtechnologies.com/m2rsm/index.html**), calcula una amplia variedad de métricas orientadas a código para C, C++ y Java.

Understand, desarrollada por Scientific Toolworks, Inc. (**www.scitools.com**), calcula métricas orientadas a código para varios lenguajes de programación.

17 Vea la sección 23.3.3 para una descripción del NDH y de la PAH.

18 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

23.8 Resumen

Las métricas de software proporcionan una forma cuantitativa para valorar la calidad de los atributos internos de producto y, por tanto, permiten valorar la calidad antes de construir el producto. Las métricas proporcionan la comprensión necesaria para crear modelos efectivos de requerimientos y diseño, código sólido y pruebas amplias.

Para ser útil en un contexto de mundo real, una métrica de software debe ser simple y calculable, convincente, congruente y objetiva. Debe ser independiente del lenguaje de programación y ofrecer retroalimentación efectiva.

Las métricas para el modelo de requerimientos se enfocan en los tres componentes del modelo: la función, los datos y el comportamiento. Las métricas para diseño consideran arquitectura, diseño en el nivel de componentes y conflictos en el diseño de interfaz. Las métricas de diseño arquitectónico consideran los aspectos estructurales del modelo de diseño. Las métricas de diseño en el nivel de componente proporcionan un indicio de la calidad del módulo al establecer medidas indirectas para cohesión, acoplamiento y complejidad. Las métricas de diseño de interfaz de usuario ofrecen un indicio de la facilidad con la que puede usarse una GUI. Las métricas *webapp* consideran aspectos de la interfaz de usuario, así como estética, contenido y navegación de la *webapp*.

Las métricas para los sistemas OO se enfocan en mediciones que pueden aplicarse a las características de clase y de diseño (localización, encapsulación, ocultamiento de información, herencia y técnicas de abstracción de objeto) que hacen única a la clase. La suite de métricas CK define seis métricas de software orientado a clase que se enfocan en la clase y en la jerarquía de clase. La suite de métricas también desarrolla métricas para valorar las colaboraciones entre clases y la cohesión en métodos que residen dentro de una clase. En un nivel orientado a clase, la suite de métricas CK puede aumentarse con las métricas propuestas por Lorenz y Kidd y con la suite de métricas MOOD.

Halstead proporciona un interesante conjunto de métricas en el nivel del código fuente. Al usar el número de operadores y operandos presentes en el código, la ciencia del software proporciona una variedad de métricas que pueden usarse para valorar la calidad del programa.

Pocas métricas de producto se han propuesto para uso directo en las pruebas de software y en el mantenimiento. Sin embargo, muchas otras métricas de producto pueden usarse para guiar el proceso de pruebas y como mecanismo para valorar la capacidad de mantenimiento de un programa de cómputo. Para valorar la comprobabilidad de un sistema OO, se ha propuesto una gran variedad de métricas OO.

Problemas y puntos por evaluar

23.1. La teoría de la medición es un tema avanzado que tiene un fuerte engranaje con las métricas de software. Con [Zus97], [Fen91], [Zus90] o fuentes en la web, escriba un breve ensayo que resalte las tesis principales de la teoría de la medición. Proyecto individual: desarrolle una presentación acerca del tema y preséntela en clase.

23.2. ¿Por qué no es posible desarrollar una sola métrica exhaustiva para la complejidad de programa o calidad de programa? Intente encontrar una medida o métrica de la vida diaria que viole los atributos de las métricas de software efectivos definidos en la sección 23.1.5.

23.3. Un sistema tiene 12 entradas externas, 24 salidas externas, presenta 30 diferentes consultas externas, gestiona 4 archivos lógicos internos y tiene interfaz con 6 diferentes sistemas legados (6 AIE). Todos estos datos son de complejidad promedio y el sistema global es relativamente simple. Calcule PF para el sistema.

23.4. El software para System X tiene 24 requerimientos funcionales individuales y 14 requerimientos no funcionales. ¿Cuál es la especificidad de los requerimientos y cuál es la completitud?

23.5. Un gran sistema de información tiene 1 140 módulos. Existen 96 módulos que realizan funciones de control y coordinación y 490 módulos cuya función depende del procesamiento previo. El sistema procesa aproximadamente 220 objetos de datos, cada uno de los cuales tiene un promedio de tres atributos. Existen 140 ítems de base de datos únicos y 90 diferentes segmentos de base de datos. Finalmente, 600 módulos tienen puntos de entrada y salida únicos. Calcule el ICED para este sistema.

23.6. Una clase **X** tiene 12 operaciones. La complejidad ciclomática se calcula para todas las operaciones en el sistema OO y el valor promedio de la complejidad de módulo es 4. Para la clase **X**, la complejidad para las operaciones de la 1 a la 12 es 5, 4, 3, 3, 6, 8, 2, 2, 5, 5, 4, 4, respectivamente. Calcule los métodos ponderados por clase.

23.7. Desarrolle una herramienta de software que calcule la complejidad ciclomática para un módulo de lenguaje de programación. Puede elegir el lenguaje.

23.8. Desarrolle una pequeña herramienta de software que realice análisis de Halstead sobre el código fuente del lenguaje de programación de su elección.

23.9. Un sistema legado tiene 940 módulos. La última liberación requirió el cambio de 90 de dichos módulos. Además, se agregaron 40 nuevos módulos y se removieron 12 módulos antiguos. Calcule el índice de madurez de software para el sistema.

Lecturas y fuentes de información adicionales

Existe un número sorprendentemente grande de libros que se dedican a las métricas de software, aunque la mayoría se enfocan en las métricas de proceso y proyecto, con la exclusión de métricas de producto. Lanza *et al.* (*Object-Oriented Metrics in Practice*, Springer, 2006) analizan métricas OO y su uso para valorar la calidad de un diseño. Genero (*Metrics for Software conceptual Models*, Imperial College Press, 2005) y Ejiogu (*Software Metrics*, BookSurge Publishing, 2005) presentan una amplia variedad de métricas técnicas para casos de uso, modelos UML y otras representaciones de modelado. Hutcheson (*Software Testing fundamentals: Methods and Metrics*, Wiley, 2003) presenta un conjunto de métricas para prueba. Kan (*Metrics and Models in Software Quality Engineering,* Addison-Wesley, 2a. ed., 2002), Fenton y Pfleeger (*Software Metrics: A Rigorous and Practical Approach,* Brooks-Cole Publishing, 1998), y Zuse [Zus97] escribieron tratamientos profundos de las métricas de producto.

Los libros de Card y Glass [Car90], Zuse [Zus90], Fenton [Fen91], Ejiogu [Eji91], Moeller y Paulish (*Software Metrics,* Chapman y Hall, 1993), y Hetzel [Het93] abordan métricas de producto con cierto detalle. Oman y Pfleeger (*Applying Software Metrics*, IEEE Computer Society Press, 1997) editaron una antología de importantes ensayos acerca de las métricas de software.

Ebert *et al.* (*Best Practices in Software Measurement*, Springer, 2004) consideran los métodos para establecer un programa de métricas y los principios subyacentes para la medición del software. Shepperd (*Foundations of Software Measurement*, Prentice-Hall, 1996) también aborda la teoría de medición con cierto detalle. La investigación actual se presenta en los *Proceedings of the Symposium on Software Metrics* (IEEE, publicación anual).

En [IEE93] se presenta un amplio resumen de decenas de métricas de software útiles. En general, un análisis de cada métrica se ha separado en las "primitivas" (medidas) esenciales requeridas para calcular la métrica y las relaciones apropiadas para efectuar el cálculo. Se proporciona análisis y muchas referencias en un apéndice.

Whitmire [Whi97] presenta un tratamiento amplio y matemáticamente sofisticado de las métricas OO. Lorenz y Kidd [Lor94] y Hendersen-Sellers (*Object-Oriented Metrics: Measures of Complexity*, Prentice-Hall, 1996) proporcionan tratamientos que se dedican a las métricas OO.

En internet, está disponible una gran variedad de fuentes de información acerca de las métricas del software. Una lista actualizada de referencias en la World Wide Web que son relevantes para la métrica del software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

PARTE

Cuatro

ADMINISTRACIÓN DE PROYECTOS DE SOFTWARE

En esta parte de *Ingeniería del software. Un enfoque práctico,* aprenderá sobre las técnicas de administración requeridas para planificar, organizar, monitorear y controlar proyectos de software. En los capítulos que siguen se abordan preguntas como las siguientes:

- ¿Cómo debe administrarse el personal, el proceso y el problema durante un proyecto de software?
- ¿Cómo pueden usarse las métricas del software para administrar un proyecto y el proceso de software?
- ¿Cómo genera un equipo de software estimaciones confiables de esfuerzo, costo y duración del proyecto?
- ¿Qué técnicas pueden usarse para valorar los riesgos que pueden tener impacto sobre el éxito del proyecto?
- ¿Cómo selecciona un gerente de proyecto de software un conjunto de tareas laborales para los ingenieros del software?
- ¿Cómo se crea un calendario de proyecto?
- ¿Por qué el mantenimiento y la reingeniería son importantes para los gerentes de ingeniería de software y para los profesionales?

Una vez respondidas estas preguntas, estará mejor preparado para administrar proyectos de software en una forma que lo conducirá a la entrega oportuna de un producto de alta calidad.

CAPÍTULO

24 Conceptos de administración de proyecto

Conceptos clave

ámbito del software 562
coordinación y comunicación. 561
descomposición de problema 563
equipo de software 558
equipos ágiles. 561
líderes de equipo. 557
participantes. 557
personal 556
prácticas cruciales 567
principio W^{5}HH 567
producto. 562
proyecto. 566

En el prefacio de su libro acerca de la administración de proyectos de software, Meiler Page-Jones [pág. 85] hace una afirmación que pueden compartir muchos consultores de ingeniería de software:

> He visitado decenas de tiendas comerciales, tanto buenas como malas, y he observado tableros de datos que procesan los administradores, de nuevo, buenos y malos. Con mucha frecuencia, observé con horror cómo estos administradores luchan infructuosamente a través de proyectos de pesadilla, se retuercen bajo fechas límite imposibles o sistemas entregados que contrarían a sus usuarios y que se dedican a devorar enormes trozos de tiempo de mantenimiento.

Lo que Page-Jones describe son síntomas extraídos de una selección de problemas administrativos y técnicos. Sin embargo, si a los proyectos se les realizara un análisis postmortem, es muy probable que se encuentre un tema común: la administración del proyecto era débil.

En éste y en los siguientes capítulos, se presentan los conceptos clave que conducen a la administración efectiva del proyecto de software. Este capítulo considera los conceptos y principios básicos de la administración de proyectos de software. El capítulo 25 presenta las métricas de proceso y proyecto, base para la toma de decisiones administrativas efectivas. Las técnicas que se usan para estimar el costo se analizan en el capítulo 26. El capítulo 27 ayuda a definir un calendario de proyecto realista. Las actividades administrativas que conducen a mo-

Una mirada rápida

¿Qué es? Aunque muchas personas (en sus momentos más oscuros) toman la visión de Dilbert de la "administración", ésta sigue siendo una actividad muy necesaria cuando se construyen sistemas y productos basados en computadora. La administración del proyecto involucra planificación, monitoreo y control del personal, procesos y acciones que ocurren conforme el software evoluciona desde un concepto preliminar hasta su despliegue operativo completo.

¿Quién lo hace? Todo mundo "administra" en cierta medida, pero el ámbito de las actividades administrativas varía entre las personas involucradas en un proyecto de software. Un ingeniero del software administra sus actividades cotidianas, planifica, monitorea y controla las tareas técnicas. Los gerentes de proyecto planifican, monitorean y controlan el trabajo de un equipo de ingenieros de software. Los gerentes ejecutivos coordinan la interfaz entre la empresa y los profesionales del software.

¿Por qué es importante? Construir software de computadora es una labor compleja, particularmente si involucra a muchas personas que trabajan durante un tiempo relativamente largo. Por eso es necesario administrar los proyectos de software.

¿Cuáles son los pasos? Comprender las cuatro P: personal, producto, proceso y proyecto. El personal debe organizarse para realizar el trabajo de software de manera efectiva. La comunicación con el cliente y con otros participantes debe ocurrir de modo que el ámbito del producto y los requerimientos sean comprensibles. Debe seleccionarse un proceso que sea adecuado para el personal y el producto. El proyecto debe planificarse, estimándose el esfuerzo y el cronograma necesarios para concluir las tareas: definición de los productos operativos, establecimiento de los puntos de verificación de calidad, identificación de mecanismos para monitorear y control del trabajo definido por el plan.

¿Cuál es el producto final? En cuanto inician las actividades de administración, se produce un plan de proyecto que define el proceso y las tareas que se van a realizar, el personal que hará el trabajo y los mecanismos que se emplearán para valorar riesgos, controlar el cambio y evaluar la calidad.

¿Cómo me aseguro de que lo hice bien? Nunca se está completamente seguro de que el plan del proyecto es correcto, hasta que se entrega un producto de alta calidad a tiempo y dentro del presupuesto. Sin embargo, un gerente de proyecto lo hace bien cuando alienta al personal del software a trabajar en conjunto, como un equipo efectivo, y cuando enfoca su atención en las necesidades del cliente y en la calidad del producto.

nitoreo, mitigación y administración efectiva del riesgo se presentan en el capítulo 28. Finalmente, el capítulo 29 considera el mantenimiento y la reingeniería, y estudia los conflictos administrativos que se encontrarán cuando lidie con sistemas heredados.

24.1 El espectro administrativo

La administración efectiva de un proyecto de software se enfoca en las cuatro P: personal, producto, proceso y proyecto. El orden no es arbitrario. El gerente que olvida que el trabajo de la ingeniería del software es una empresa intensamente humana nunca triunfará en la administración del proyecto. Un gerente que fracase en alentar una comunicación comprensiva con los participantes durante las primeras etapas de la evolución de un producto se arriesga a construir una solución elegante para el problema equivocado. El gerente que ponga poca atención al proceso corre el riesgo de insertar métodos y herramientas técnicos competentes pero en el vacío. Aquel que se embarque sin un plan sólido pone en peligro el éxito del proyecto.

24.1.1 El personal

Desde la década de 1960 se estudia la formación de personal de software motivado y enormemente calificado. De hecho, el "factor humano" es tan importante que el Software Engineering Institute desarrolló un *Modelo de madurez de capacidades del personal* (People-CMM, por sus siglas en inglés), en reconocimiento al hecho de que "toda organización requiere mejorar continuamente su habilidad para atraer, desarrollar, motivar, organizar y conservar la fuerza de trabajo necesaria a fin de lograr sus objetivos empresariales estratégicos" [Cur01].

El People-CMM define las siguientes áreas prácticas clave para el personal de software: plantilla, comunicación y coordinación, ambiente de trabajo, desempeño administrativo, capacitación, compensación, análisis y desarrollo de competencias, desarrollo profesional, desarrollo de grupo de trabajo y desarrollo de equipo/cultura, entre otros. Las organizaciones que conforme a este modelo logran altos niveles de madurez de capacidades de personal tienen una probabilidad muy elevada de alcanzar la implementación de prácticas administrativas efectivas en los proyectos de software.

El People-CMM es un compañero de la *Integración del modelo de madurez de capacidades del software* (capítulo 30), que guía a las organizaciones en la creación de un proceso de software maduro. Los conflictos asociados con la administración del personal y con la estructura para los proyectos de software se consideran más adelante, en este capítulo.

24.1.2 El producto

Antes de poder planear un proyecto, deben establecerse los objetivos y el ámbito del producto, considerarse soluciones alternativas e identificar las restricciones técnicas y administrativas. Sin esta información, es imposible definir estimaciones razonables (y precisas) del costo, una valoración efectiva del riesgo, una descomposición realista de las tareas del proyecto y un calendario de proyecto manejable que proporcione en cada momento un indicio significativo del progreso.

Como desarrolladores de software, todos los participantes deben reunirse para definir los objetivos y el ámbito del producto. En muchos casos, esta actividad comienza como parte de la ingeniería del sistema o de la ingeniería del proceso empresarial y continúa como el primer paso en la ingeniería de requerimientos del software (capítulo 5). Los objetivos identifican las metas globales para el producto (desde el punto de vista de los participantes) sin considerar cómo se lograrán estas metas. El ámbito identifica los datos, funciones y comportamientos principales que caracterizan al producto y, más importante, intenta ligar dichas características en forma cuantitativa.

Una vez comprendidos los objetivos y el ámbito del producto, se consideran soluciones alternativas. Aunque se analizan muy pocos detalles, las alternativas permiten a los gerentes y profesionales seleccionar un "mejor" enfoque, dadas las restricciones impuestas por fechas de entrega, restricciones presupuestales, disponibilidad de personal, interfaces técnicas y muchos otros factores.

24.1.3 El proceso

Quienes se adhieren a la filosofía de proceso ágil (capítulo 3) argumentan que su proceso es más esbelto que otros. Esto puede ser cierto, pero todavía tienen un proceso, y la ingeniería de software ágil todavía requiere disciplina.

Un proceso de software (capítulos 2 y 3) proporciona el marco conceptual desde el cual puede establecerse un plan completo para el desarrollo de software. Un pequeño número de actividades de marco conceptual se aplica a todos los proyectos de software, sin importar su tamaño o complejidad. Algunos conjuntos de diferentes tareas (tareas, hitos, productos operativos y puntos de aseguramiento de calidad) permiten que las actividades del marco conceptual se adapten a las características del proyecto de software y a los requerimientos del equipo del proyecto. Finalmente, las actividades sombrilla (como el aseguramiento de la calidad del software, la administración de configuración del software y las mediciones) recubren el modelo de proceso. Las actividades sombrilla son independientes de cualquier actividad del marco conceptual y ocurren a lo largo del proceso.

24.1.4 El proyecto

Los proyectos de software se planean y controlan debido a una razón principal: es la única forma conocida para manejar la complejidad. E incluso así, los equipos de software todavía batallan. En un estudio de 250 grandes proyectos de software desarrollados entre 1998 y 2004, Capers Jones [Jon04] encontró que "alrededor de 25 se consideraron exitosos por haber logrado sus objetivos de calendario, costo y calidad. Aproximadamente 50 tuvieron demoras o excesos por abajo de 35 por ciento, mientras que más o menos 175 experimentaron grandes demoras y excesos, o se dieron por concluidos sin completarse". Aunque actualmente la tasa de éxito para los proyectos de software puede haber mejorado un poco, la tasa de falla de proyecto sigue siendo mucho más alta de lo que debiera.[1]

Para evitar el fracaso del proyecto, un gerente de proyecto de software y los ingenieros de software que construyan el producto deben evitar un conjunto de señales de advertencia comunes, entender los factores de éxito cruciales que conducen a una buena administración del proyecto y desarrollar un enfoque de sentido común para planificar, monitorear y controlar el proyecto. Cada uno de estos temas se estudia en la sección 24.5 y en los capítulos que siguen.

24.2 El personal

En un estudio publicado por el IEEE [Cur88], se preguntó a los vicepresidentes de ingeniería de tres grandes compañías tecnológicas cuál era el elemento más importante para el éxito de un proyecto de software. Ellos respondieron de la siguiente manera:

VP 1: Supongo que, si tienes que elegir una cosa que sea la más importante en nuestro ambiente, diría que no son las herramientas que usamos, es el personal.

VP 2: El ingrediente más importante que fue exitoso en este proyecto fue tener gente inteligente [...] en mi opinión, muy pocas cosas más importan [...] La cosa más importante que

1 Con estas estadísticas, es razonable preguntar cómo sigue creciendo exponencialmente el impacto de las computadoras. Parte de la respuesta es que un número sustancial de estos proyectos "fallidos" estuvieron mal concebidos desde el inicio. Los clientes pierden interés rápidamente (porque lo que pidieron en realidad no era tan importante como pensaron la primera vez) y los proyectos se cancelan.

haces para un proyecto es seleccionar al personal [...] El éxito de la organización de desarrollo de software está muy, muy asociada con la habilidad para reclutar buen personal.

VP 3: La única regla que tengo en la administración es asegurarme de que tengo buen personal, gente realmente buena, y que hago crecer gente buena y que proporciono un ambiente en el que la gente buena puede producir.

De hecho, éste es un testimonio convincente acerca de la importancia del personal en el proceso de ingeniería de software. Y aún así, para la mayoría de las personas, desde los vicepresidentes ejecutivos de ingeniería hasta el profesional en el nivel más bajo, con frecuencia dan por hecho al personal. Los administradores argumentan (como lo hizo el grupo anterior) que las personas son lo importante, pero sus acciones en ocasiones contradicen sus palabras. En esta sección se examina a las personas que participan en el proceso de software y la forma en la que se organizan para realizar ingeniería de software efectiva.

24.2.1 Los participantes

El proceso de software (y todo proyecto de software) está poblado de participantes, quienes pueden organizarse en alguna de las siguientes áreas:

1. *Gerentes ejecutivos*, quienes definen los temas empresariales que con frecuencia tienen una influencia significativa sobre el proyecto.
2. *Gerentes de proyecto (técnicos)*, quienes deben planificar, motivar, organizar y controlar a los profesionales que hacen el trabajo de software.
3. *Profesionales* que aportan las habilidades técnicas que se necesitan para someter a ingeniería un producto o aplicación.
4. *Clientes* que especifican los requerimientos para el software que se va a fabricar, así como otros participantes que tienen un interés periférico en el resultado.
5. *Usuarios finales*, quienes interactúan con el software una vez que se libera para su uso productivo.

Todo proyecto de software está poblado con personas que están dentro de esta taxonomía.[2] Para ser efectivo, el equipo de software debe organizarse de manera que maximice las habilidades y capacidades de cada persona. Y ésta es labor del líder del equipo.

24.2.2 Líderes de equipo

La administración del proyecto es una actividad que implica mucho trato con la gente; por esta razón, los profesionales competentes tienen con frecuencia pobre desempeño como líderes de equipo. Simplemente, no tienen la mezcla justa de habilidades personales. Y aún así, como Edgemon afirma: "Por desgracia, y por muy frecuente que parezca, los individuos simplemente se topan con el papel de gerente de proyecto y se convierten en gerentes accidentales de proyecto" [Edg95].

En un excelente libro acerca del liderazgo técnico, Jerry Weinberg [Wei86] sugiere un modelo MOI de liderazgo:

¿Qué se busca cuando se elige a alguien como líder de un proyecto de software?

Motivación. Habilidad para alentar (mediante "empuje o jalón") al personal técnico a producir a su máxima capacidad.

Organización. Habilidad para moldear los procesos existentes (o inventar nuevos) que permitirán que el concepto inicial se traduzca en un producto final.

2 Cuando se desarrollan *webapps*, personal no técnico puede involucrarse en la creación de contenido.

Ideas o innovación. Habilidad para alentar a las personas a crear y sentirse creativas, aun cuando deban trabajar dentro de fronteras establecidas para un producto o aplicación de software particular.

Weinberg sugiere que los líderes de proyecto exitosos aplican un estilo administrativo de resolución de problemas. Es decir, un gerente de proyecto de software debe concentrarse en comprender el problema que se va a resolver, en administrar el flujo de ideas y, al mismo tiempo, en dejar que todos en el equipo sepan (por medio de palabras y, mucho más importante, con acciones) que la calidad cuenta y que no se comprometerá.

Cita:

"En términos más simples, un líder es aquel que sabe a dónde quiere ir, y se levanta y marcha."

John Erskine

Otra visión [Edg95] de las características que definen a un gerente de proyecto eficaz enfatiza cuatro rasgos clave:

Resolución de problemas. Un gerente de proyecto de software eficaz puede diagnosticar los conflictos técnicos y organizativos que son más relevantes, estructura sistemáticamente una solución o motiva adecuadamente a otros profesionales para desarrollarla, aplica lecciones aprendidas de proyectos pasados a situaciones nuevas y sigue siendo suficientemente flexible para cambiar de dirección si los intentos por resolver el problema son infructuosos.

Identidad administrativa. Un buen gerente de proyecto debe hacerse cargo del mismo. Debe tener la confianza para asumir el control cuando sea necesario y asegurarse de permitir que el buen personal técnico siga sus instintos.

Logro. Un gerente competente debe recompensar la iniciativa y el logro para optimizar la productividad de un equipo de proyecto. Debe demostrar mediante sus acciones que no se castigará del correr riesgos de manera controlada.

Influencia y construcción del equipo. Un gerente de proyecto eficaz debe poder "leer" a la gente; debe poder comprender las señales verbales y no verbales, y reaccionar ante las necesidades de las personas que envían estas señales. El gerente debe permanecer bajo control en situaciones de alto estrés.

24.2.3 El equipo de software

Cita:

"No todo grupo es un equipo y no todo equipo es eficaz."

Glenn Parker

Existen casi tantas estructuras organizativas humanas para el desarrollo del software como organizaciones que lo desarrollan. Para bien o para mal, la estructura organizativa no puede modificarse fácilmente. La preocupación por las consecuencias prácticas y por las políticas del cambio organizativo no está dentro del ámbito de responsabilidad del gerente del proyecto de software. Sin embargo, la organización de las personas directamente involucradas en un nuevo proyecto de software está dentro del campo de acción del gerente del proyecto.

La "mejor" estructura de equipo depende del estilo administrativo de la organización, del número de personas que formarán el equipo y de sus niveles de habilidad, así como de la dificultad global del problema. Mantei [Man81] describe siete factores de proyecto que deben considerarse cuando se planee la estructura de los equipos de ingeniería de software:

? ¿Qué factores deben considerarse cuando se elige la estructura de un equipo de software?

- Dificultad del problema que se va a resolver.
- "Tamaño" del programa resultante en líneas de código o puntos de función.
- Tiempo que el equipo permanecerá unido (vida del equipo).
- Grado en el que puede dividirse en módulos el problema.
- Calidad y confiabilidad requeridas por el sistema que se va a construir.
- Rigidez de la fecha de entrega.
- Grado de sociabilidad (comunicación) requerido para el proyecto.

Constantine [Con93] sugiere cuatro "paradigmas organizacionales" para los equipos de ingeniería de software:

¿Qué opciones se tienen cuando se define la estructura de un equipo de software?

1. Un *paradigma cerrado* estructura un equipo conforme a una jerarquía de autoridad tradicional. Tales equipos pueden trabajar bien cuando producen software muy similar al de esfuerzos anteriores, pero será menos probable que sean innovadores cuando trabajen dentro de este paradigma.
2. Un *paradigma aleatorio* estructura un equipo de manera holgada y depende de la iniciativa individual de los miembros del equipo. Cuando se requiere innovación o avance tecnológico, destacarán los equipos que siguen este paradigma, pero pueden batallar cuando se requiera "desempeño ordenado".
3. Un *paradigma abierto* intenta estructurar un equipo de manera que logre algunos de los controles asociados con el paradigma cerrado, pero también mucha de la innovación que ocurre cuando se usa el paradigma aleatorio. El trabajo se realiza de manera colaboradora; la gran comunicación y la toma de decisiones consensuadas constituyen las características de los equipos de paradigma abierto. Las estructuras de equipo de este paradigma son muy adecuadas para la solución de problemas complejos, pero pueden no desempeñarse tan eficazmente como otros equipos.
4. Un *paradigma síncrono* se apoya en la compartimentalización natural de un problema y organiza a los miembros del equipo para trabajar en trozos del problema con poca comunicación activa entre ellos.

Cita:

"Si quieres ser incrementalmente mejor: sé competitivo. Si quieres ser exponencialmente mejor: sé cooperativo."

Autor desconocido

Como acotación histórica, cabe decir que una de las primeras organizaciones de equipo de software fue una estructura de paradigma cerrado originalmente llamado *equipo de programador jefe*. Esta estructura la propuso por primera ocasión Harlan Mills y la describió Baker [Bak72]. El núcleo del equipo estaba compuesto de: un *ingeniero ejecutivo* (el programador jefe), quien planeaba, coordinaba y revisaba todas las actividades técnicas del equipo; *personal técnico* (por lo general de dos a cinco personas), quienes realizaban análisis y desarrollaban actividades; y un *ingeniero de respaldo*, quien apoyaba al ingeniero ejecutivo en sus actividades y podía sustituirlo con mínima pérdida en la continuidad del proyecto. El programador jefe puede auxiliarse con uno o más especialistas (por ejemplo, experto en telecomunicaciones, diseñador de bases de datos), personal de apoyo (por ejemplo, escritores técnicos, oficinistas) y un bibliotecario de software.

Como contrapunto a la estructura del equipo del programador jefe, el paradigma aleatorio de Constantine [Con93] sugiere un equipo de software con independencia creativa cuyo enfoque para trabajar pueda denominarse de mejor manera como *anarquía innovadora*. Aunque el enfoque de espíritu libre en el trabajo de software es atractivo, canalizar la energía creativa hacia un equipo de alto rendimiento debe ser una meta central de una organización de ingeniería de software. Para lograr un equipo de alto rendimiento:

- Los miembros del equipo deben tenerse confianza entre sí.
- La distribución de habilidades debe ser adecuada para el problema.
- Es posible que tenga que excluirse del equipo a los inconformes si debe mantenerse la cohesión del equipo.

Sin importar el tipo de organización del equipo, el objetivo para todo gerente de proyecto es ayudar a crear un equipo que muestre cohesión. En su libro *Peopleware*, DeMarco y Lister [DeM98] analizan este tema:

> Tendemos a usar la palabra equipo con mucha holgura en el mundo empresarial, y llamamos así a cualquier grupo de personas asignadas para trabajar juntas. Pero muchos de estos grupos simple-

¿Qué es un equipo "cuajado"?

mente no parecen equipos. No tienen una definición común de éxito o algún espíritu de equipo identificable. Lo que falta es un fenómeno que llamamos *cuajar*.

Un equipo cuajado es un grupo de personas tan fuertemente unido que el todo es mayor que la suma de las partes [...].

Una vez que un equipo comienza a cuajarse, la probabilidad de éxito va hacia arriba. El equipo puede volverse imparable, una fuerza arrasadora para el éxito [...] No necesita ser administrado en la forma tradicional y, ciertamente, no necesita ser motivado. Adquiere cantidad de movimiento.

DeMarco y Lister sostienen que los miembros de los equipos cuajados son significativamente más productivos y más motivados que el promedio. Comparten una meta común, una cultura común y en muchos casos un "sentido de élite" que los hace únicos.

¿Por qué fallan los equipos cuando deben cuajar?

Pero no todos los equipos cuajan. De hecho, muchos equipos sufren de lo que Jackman [Jac98] llama "toxicidad de equipo". Ella define cinco factores que "fomentan un ambiente de equipo potencialmente tóxico": 1) una atmósfera de trabajo frenético, 2) alta frustración que causa fricción entre los miembros del equipo, 3) un proceso de software "fragmentado o pobremente coordinado", 4) una definición poco clara de los roles en el equipo de software y 5) "continua y repetida exposición al fracaso".

Cita:

"Hacer o no hacer. No hay intento."

Yoda, de *La guerra de las galaxias*

Para evitar un ambiente de trabajo frenético, el gerente del proyecto debe estar seguro de que el equipo tiene acceso a toda la información requerida para hacer el trabajo y de que las metas y objetivos principales, una vez definidos, no deben modificarse a menos que sea absolutamente necesario. Un equipo de software puede evitar la frustración si se le da tanta responsabilidad para la toma de decisiones como sea posible. Un proceso inadecuado (por ejemplo, tareas innecesarias o abrumadoras o productos operativos pobremente elegidos) puede evitarse al entender el producto que se va a construir y a las personas que hacen el trabajo, así como al permitir al equipo seleccionar el modelo de proceso. El equipo mismo debe establecer sus propios mecanismos de responsabilidad (las revisiones técnicas[3] son una excelente forma de lograr esto) y definir una serie de enfoques correctos cuando un miembro del equipo tiene fallos en el desempeño. Finalmente, la clave para evitar una atmósfera de fracaso radica en establecer técnicas basadas en equipo para retroalimentarse y resolver problemas.

Además de las cinco toxinas descritas por Jackman, un equipo de software con frecuencia batalla con los diferentes rasgos humanos de sus miembros. Algunos son extrovertidos; otros, introvertidos. Algunas personas reúnen información de manera intuitiva y separan los conceptos abarcadores de los hechos dispares. Otras procesan la información de manera lineal, y reúnen y organizan detalles minúsculos de los datos proporcionados. Ciertos miembros del equipo se sienten cómodos al tomar decisiones sólo cuando se presenta un argumento lógico y ordenado. Otros son intuitivos y quieren tomar decisiones con base en "corazonadas". Algunos profesionales quieren calendarios detallados poblados de tareas organizadas que les permitan lograr el cierre para algún elemento de un proyecto. Otros prefieren un ambiente más espontáneo en el que los temas abiertos sean bien vistos. Algunos trabajan duro para hacer que las cosas estén listas mucho antes de una fecha final y, por tanto, evitan el estrés conforme la fecha se aproxima, mientras que otros se sienten energizados por la adrenalina que produce una entrega de último minuto. Una discusión detallada de la psicología de estos rasgos y de las formas en las que el líder del equipo habilidoso puede ayudar a las personas con rasgos opuestos para trabajar en conjunto está más allá del ámbito de este libro.[4] Sin embargo, es importante observar que el reconocimiento de las diferencias humanas es el primer paso hacia la creación de equipos que cuajen.

3 Las revisiones técnicas se estudian con detalle en el capítulo 15.

4 Una excelente introducción a estos temas, en su relación con los equipos de proyecto de software, puede encontrarse en [Fer98].

24.2.4 Equipos ágiles

Durante las décadas pasadas, el desarrollo de software ágil (capítulo 3) se ha sugerido como antídoto a muchos de los problemas que plagan el trabajo en un proyecto de software. Cabe recordar que la filosofía ágil alienta la satisfacción del cliente y la entrega incremental temprana del software, así como pequeños equipos de proyecto enormemente motivados, métodos informales, mínimos productos operativos de ingeniería de software y simplicidad de desarrollo global.

El pequeño equipo de trabajo enormemente motivado, también llamado *equipo ágil*, adopta la mayoría de las características de los equipos de proyecto de software exitosos que se estudiaron en la sección anterior y evita muchas de las toxinas que crean problemas. No obstante, la filosofía ágil subraya la competencia individual (miembro de equipo), acoplada con la colaboración grupal como factores de éxito vitales para el equipo. Cockburn y Highsmith [Coc01a] observan esto cuando escriben:

> Si el personal del proyecto es suficientemente bueno, puede usar casi cualquier proceso y lograr esta asignación. Si no lo es, ningún proceso reparará su inadecuación: "personal mata proceso" es una forma de decirlo. Sin embargo, la falta de apoyo de usuarios y ejecutivos puede matar al proyecto: "política mata personal". El apoyo inadecuado puede impedir que incluso el personal bueno logre esta tarea.

PUNTO CLAVE

Un equipo ágil es un equipo autoorganizado que tiene autonomía para planificar y tomar decisiones técnicas.

Para hacer uso efectivo de las competencias de cada miembro del equipo y fomentar la colaboración efectiva a través de un proyecto de software, los equipos ágiles son *autoorganizados*. Un equipo autoorganizado no necesariamente mantiene una sola estructura de equipo, sino que usa elementos de los paradigmas aleatorio, abierto y síncrono de Constantine, estudiados en la sección 24.2.3.

Muchos modelos de proceso ágil (por ejemplo, Scrum) dan al equipo ágil significativa autonomía para tomar las decisiones administrativas y técnicas del proyecto necesarias para hacer que el trabajo se cumpla. La planificación se mantiene al mínimo y al equipo se le permite seleccionar su propio enfoque (por ejemplo, proceso, métodos, herramientas), restringido únicamente por los requerimientos empresariales y los estándares de la organización. Conforme avanza el proyecto, el equipo se autoorganiza para enfocarse en la competencia individual, de manera que ésta sea más benéfica para el proyecto en un momento determinado. Para lograr esto, un equipo ágil puede realizar reuniones grupales diarias para coordinar y sincronizar el trabajo que debe realizarse en ese día.

Con base en la información obtenida durante dichas reuniones, el equipo adapta su enfoque para lograr un incremento de trabajo. Conforme transcurre cada día, la autoorganización y la colaboración continuas mueven al equipo hacia un incremento de software completo.

24.2.5 Conflictos de coordinación y comunicación

Cita:

"La propiedad colectiva no es más que una ilustración de la idea de que los productos deben atribuirse al equipo [ágil], no a los individuos que constituyen el equipo."

Jim Highsmith

Existen muchas razones por las que los proyectos de software tienen problemas. La escala de muchos esfuerzos de desarrollo es grande, lo que conduce a complejidad, confusión y dificultades significativas en la coordinación de los miembros del equipo. La incertidumbre es común, lo que da como resultado un torrente continuo de cambios que detienen al equipo de proyecto. La interoperabilidad se ha convertido en una característica clave de muchos sistemas. El software nuevo debe comunicarse con el software existente y ajustarse a las restricciones predefinidas impuestas por el sistema o por el producto.

Tales características del software moderno (escala, incertidumbre e interoperabilidad) son hechos de la vida. Para lidiar con ellos de manera efectiva, deben implantarse métodos efectivos a fin de coordinar al personal que hace el trabajo. Esto se logra estableciendo mecanismos para la comunicación formal e informal entre los miembros del equipo y entre los distintos equipos. La comunicación formal se consigue mediante "comunicación escrita, reuniones estructuradas y otros canales de comunicación relativamente no interactivos e impersonales" [Kra95]. La co-

municación informal es más personal. Los miembros de un equipo de software comparten ideas sobre una base *ad hoc*, piden ayuda cuando surgen problemas e interactúan unos con otros diariamente.

CASASEGURA

Estructura del equipo

La escena: Oficina de Doug Miller antes de iniciar el proyecto de software *CasaSegura*.

Personajes: Doug Miller (gerente del equipo de ingeniería de software *CasaSegura*) y Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería de software del producto.

La conversación:

Doug: ¿Han tenido oportunidad de consultar la información preliminar que preparó mercadotecnia acerca de *CasaSegura*?

Vinod (asiente y observa a sus compañeros de equipo): Sí. Pero tenemos muchas preguntas.

Doug: Dejemos eso por un momento. Me gustaría hablar acerca de cómo vamos a estructurar el equipo, quién es responsable de qué...

Jamie: Yo estoy totalmente a favor de la filosofía ágil, Doug. Creo que debemos ser un equipo autoorganizado.

Vinod: Estoy de acuerdo. Dada la apretada línea de tiempo y algo de la incertidumbre, así como el hecho de que todos somos realmente competentes [risas], ésta parece ser la forma correcta de avanzar.

Doug: Está bien por mí, pero ustedes conocen las instrucciones.

Jamie (sonríe y habla como si recitara algo): "Tomamos decisiones tácticas acerca de quién hace qué y cuándo, pero es nuestra responsabilidad sacar el producto a tiempo."

Vinod: Y con calidad.

Doug: Exactamente. Pero recuerden que hay restricciones. Mercadotecnia define los incrementos del software que se va a producir... consultándonos, desde luego.

Jamie: ¿Y?

Doug: Y vamos a usar UML como nuestro enfoque de modelado.

Vinod: Pero mantendremos la documentación extraña en un mínimo absoluto.

Doug: ¿Quién es el enlace conmigo?

Jamie: Decidimos que Vinod sea el líder técnico; tiene más experiencia, así que Vinod es tu enlace, pero siéntete en libertad de hablar con cualquiera de nosotros.

Doug (ríe): No se preocupen. Lo haré.

24.3 EL PRODUCTO

Un gerente de proyecto de software se enfrenta con un dilema en el comienzo mismo de un proyecto de software. Se requieren estimaciones cuantitativas y un plan organizado, pero no hay información sólida disponible. Un análisis detallado de los requerimientos del software proporcionaría la información necesaria para las estimaciones, pero el análisis usualmente tarda semanas o incluso meses en completarse. Peor aún, los requerimientos pueden ser fluidos y cambiar con regularidad conforme avanza el proyecto. Y, sin embargo, ¡se necesita un plan "ahora"!

Le guste o no al gerente de proyecto, debe examinar el producto; se pretende que el problema se resuelva desde el principio mismo del proyecto; cuando menos, debe establecer y acotar el ámbito del producto.

24.3.1 Ámbito del software

La primera actividad en la administración del proyecto de software es determinar el *ámbito del software*, que se define al responder las siguientes preguntas:

Si no puede acotar una característica del software que intenta construir, mencione la característica como un riesgo del proyecto (capítulo 25).

Contexto. ¿Cómo encaja en un sistema, producto o contexto empresarial más grande el software que se va a construir y qué restricciones se imponen como resultado del contexto?

Objetivos de información. ¿Qué objetos de datos visibles para el cliente se producen como salida del software? ¿Qué objetos de datos se requieren como entrada?

Función y desempeño. ¿Qué función realiza el software para transformar los datos de entrada en salida? ¿Existe alguna característica de desempeño especial que deba abordarse?

El ámbito del proyecto de software no debe tener ambigüedades ni ser incomprensible en los niveles administrativo y técnico. Debe acotar un enunciado del ámbito del software; es decir, los datos cuantitativos (por ejemplo, número de usuarios simultáneos, entorno objetivo, máximo tiempo de respuesta permisible) se enuncian de manera explícita, se anotan las restricciones y/o limitaciones (por ejemplo, el costo del producto restringe el tamaño de la memoria) y se describen los factores mitigantes (por ejemplo, los algoritmos deseados están bien entendidos y disponibles en Java).

24.3.2 Descomposición del problema

Para desarrollar un plan de proyecto razonable, debe descomponer el problema. Esto puede lograrse usando una lista de funciones o con casos de uso.

La descomposición del problema, en ocasiones llamada *división* o *elaboración del problema*, es una actividad que se asienta en el centro del análisis de requerimientos del software (capítulos 6 y 7). Durante la actividad de determinación del ámbito, no se hacen intentos por descomponer completamente el problema. En vez de ello, la descomposición se aplica en dos áreas principales: 1) la funcionalidad y el contenido (información) que deben entregarse y 2) el proceso que se usará para entregarlo.

Los seres humanos tienden a aplicar una estrategia de "divide y vencerás" cuando se enfrentan a un problema complejo. Dicho de manera simple, un problema complejo se divide en problemas más pequeños que son más manejables. Ésta es la estrategia que se aplica conforme comienza la planeación del proyecto. Las funciones del software, descritas en el enunciado del ámbito, se evalúan y refinan para proporcionar más detalle antes de comenzar la estimación (capítulo 26). Puesto que tanto las estimaciones de costo como las de calendario se orientan funcionalmente, con frecuencia es útil cierto grado de descomposición. De igual modo, los principales objetos de contenido o datos se descomponen en sus partes constituyentes, lo que proporciona una comprensión razonable de la información que se va a producir con el software.

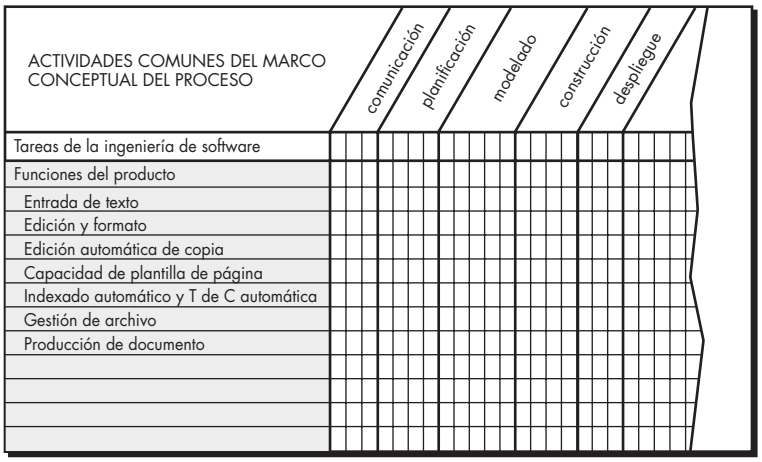
Tome como ejemplo un proyecto que construirá un nuevo producto de procesamiento de palabras. Entre las características únicas del producto, están la entrada continua de voz, así como un teclado virtual a través de una pantalla táctil, características extremadamente sofisticadas de "edición de copia automática", capacidad de plantilla de página, indexado automático y tabla de contenidos automática. El gerente del proyecto debe establecer primero un enunciado del ámbito que acote dichas características (y otras funciones más comunes, tales como edición, gestión de archivos y producción de documentos). Por ejemplo, ¿la entrada continua de voz requiere que el usuario "entrene" al producto? Específicamente: ¿qué capacidades ofrecerá el editor de copia? ¿Cuán sofisticadas serán las capacidades de la plantilla de página?, ¿éstas abarcarán las capacidades que requiere una pantalla táctil?

Conforme avanza la determinación de ámbito, ocurre de manera natural un primer nivel de división. El equipo de proyecto aprende que el departamento de mercadotecnia habló con los clientes potenciales y descubrió que las siguientes funciones deben ser parte de la edición automática de copia: 1) corrector de vocabulario, 2) corrector gramatical, 3) comprobación de referencias para documentos grandes (por ejemplo, ¿una referencia a una entrada bibliográfica se encuentra en la lista de entrada en la bibliografía?), 4) implementación de una característica de hoja de estilo que imponga consistencia a través de un documento y 5) validación de referencias de sección y capítulo para documentos grandes. Cada una de estas características representa una subfunción por implementar en el software. Cada una puede refinarse aún más si la descomposición hace más sencilla la planificación.

24.4 El proceso

Las actividades del marco conceptual (capítulo 2) que caracterizan al proceso de software son aplicables a todos los proyectos de software. El problema es seleccionar el modelo de proceso que sea adecuado para el software que el equipo del proyecto someterá a ingeniería.

FIGURA 24.1
Fusión de problema y proceso



El equipo debe decidir qué modelo de proceso es más adecuado: 1) para los clientes que solicitaron el producto y el personal que hará el trabajo, 2) para las características del producto en sí y 3) para el entorno de proyecto donde trabaja el equipo de software. Cuando se selecciona un modelo de proceso, el equipo define entonces un plan de proyecto preliminar con base en el conjunto de actividades del marco conceptual del proceso. Una vez establecido el plan preliminar, comienza la descomposición del proceso, es decir, debe crearse un plan completo que refleje las tareas laborales requeridas para poblar las actividades del marco conceptual. Estas actividades se exploran brevemente en las secciones que siguen; en el capítulo 26 se presenta una visión más detallada.

24.4.1 Fusión de producto y proceso

La planificación del proyecto comienza con la fusión de producto y proceso. Cada función que se va a someter a ingeniería por parte del equipo debe pasar a través del conjunto de actividades de marco conceptual que defina la organización de software.

Suponga que la organización adoptó las actividades genéricas del marco conceptual que se estudiaron en el capítulo 2: **comunicación**, **planificación**, **modelado**, **construcción** y **despliegue**. Los miembros del equipo que trabajen en una función del producto aplicarán en ella cada una de las actividades del marco conceptual. En esencia, se crea una matriz similar a la que se muestra en la figura 24.1. Cada función de producto principal (las funciones anotadas con números para el software de procesamiento de palabra estudiadas anteriormente) se menciona en la columna izquierda. Las actividades de marco conceptual se mencionan en la fila superior. Las tareas del trabajo de la ingeniería de software (para cada actividad del marco conceptual) ingresarán en la fila siguiente.[5] La labor del gerente de proyecto (y otros miembros del equipo) es estimar los requerimientos de recurso para cada celda de la matriz, fechas de inicio y término de las tareas asociadas con cada celda, y los productos operativos que se van a producir como consecuencia de cada tarea. Dichas actividades se consideran en el capítulo 26.

24.4.2 Descomposición del proceso

Un equipo de software debe tener un grado significativo de flexibilidad al elegir el modelo de proceso de software que es mejor para el proyecto y las tareas de la ingeniería de software que

5 Obsérvese que las tareas deben adaptarse a las necesidades específicas del proyecto, con base en algunos criterios de adaptación.

PUNTO CLAVE

El marco conceptual del proceso establece un esqueleto para la planificación del proyecto y se adapta para abarcar un conjunto de tareas que son adecuadas para el proyecto.

pueblen el modelo de proceso una vez elegido. Un proyecto relativamente pequeño que sea similar a esfuerzos anteriores puede lograrse mejor al usar el enfoque secuencial lineal. Si la fecha límite es tan apretada como para que toda la funcionalidad no pueda entregarse razonablemente, puede ser mejor una estrategia incremental. De igual modo, los proyectos con otras características (por ejemplo, requerimientos de incertidumbre, tecnología innovadora, clientes difíciles, significativo potencial de reuso) conducirán a la selección de otros modelos de proceso.[6]

Una vez elegido el modelo de proceso, el marco conceptual del proceso se adapta a él. En todo caso, puede usarse el marco conceptual genérico de proceso que se estudió anteriormente. Funcionará para los modelos lineales, para modelos iterativos e incrementales, para modelos evolutivos e incluso para modelos concurrentes o de ensamble de componentes. El marco conceptual del proceso es invariante y sirve como la base para todo el trabajo que realiza una organización de software.

Pero las tareas del trabajo real sí varían. La descomposición del proceso comienza cuando el gerente de proyecto pregunta: ¿cómo logramos esta actividad del marco conceptual? Por ejemplo, un proyecto simple y relativamente pequeño puede requerir las siguientes tareas para la actividad de comunicación:

1. Desarrollar lista de clarificación de conflictos.
2. Reunirse con los participantes para abordar la clarificación de conflictos.
3. Desarrollar en conjunto un enunciado del ámbito.
4. Revisar el enunciado del ámbito con todos los interesados.
5. Modificar el enunciado del ámbito según se requiera.

Estos eventos pueden ocurrir durante un periodo de menos de 48 horas. Representan una descomposición de proceso que es adecuada para el pequeño proyecto relativamente simple.

Ahora, considere un proyecto más complejo, que tenga un ámbito más amplio e impacto empresarial más significativo. Tal proyecto puede requerir las siguientes tareas para la **comunicación**:

1. Revisar la solicitud del cliente.
2. Planificar y calendarizar una reunión formal facilitada con todos los participantes.
3. Realizar investigación para especificar la solución propuesta y los enfoques existentes.
4. Preparar un "documento de trabajo" y una agenda para la reunión formal.
5. Realizar la reunión.
6. Desarrollar conjuntamente miniespecificaciones que reflejen las características de datos, funcionales y de comportamiento del software. De manera alternativa, desarrollar casos de uso que describan el software desde el punto de vista del usuario.
7. Revisar cada miniespecificación o usar casos de uso para ver su exactitud, consistencia y falta de ambigüedad.
8. Ensamblar las miniespecificaciones en un documento de ámbito.
9. Revisar el documento de ámbito o colección de casos de uso con todos los interesados.
10. Modificar el documento de ámbito o casos de uso según se requiera.

Ambos proyectos realizan la actividad de marco conceptual que se llama **comunicación**, pero el primer equipo de proyecto realiza la mitad de tareas de trabajo de ingeniería de software que el segundo.

---

6 Recuerde que las características del proyecto también tienen mucho apoyo en la estructura del equipo de software (sección 24.2.3).

24.5 El proyecto

Para administrar un proyecto de software exitoso, se debe comprender qué puede salir mal, de modo que los problemas puedan evitarse. En un excelente ensayo acerca de los proyectos de software, John Reel [Ree99] define 10 señales que indican que un proyecto de sistemas de información está en peligro:

¿Cuáles son las señales de que un proyecto de software está en peligro?

1. El personal del software no entiende las necesidades del cliente.
2. El ámbito del producto está pobremente definido.
3. Los cambios se gestionan pobremente.
4. Cambia la tecnología elegida.
5. Las necesidades empresariales cambian [o están mal definidas].
6. Las fechas límite son irreales.
7. Los usuarios son resistentes.
8. Pérdida de patrocinio [o nunca obtenido adecuadamente].
9. El equipo del proyecto carece de personal con habilidades adecuadas.
10. Los gerentes [y profesionales] evitan mejores prácticas y lecciones aprendidas.

Cita:

"No tenemos tiempo para detenernos por combustible, ya vamos retrasados."

M. Cleron

Los profesionales de la industria, hastiados, con frecuencia se refieren a la regla 90-90 cuando estudian proyectos de software particularmente difíciles: el primer 90 por ciento de un sistema absorbe el 90 por ciento del esfuerzo y tiempo asignados. El último 10 por ciento toma otro 90 por ciento del esfuerzo y tiempo asignados [Zah94]. Las semillas que conducen a la regla 90-90 están contenidas en las señales anotadas en la lista anterior.

Pero, ¡basta de negatividad! ¿Cómo actúa un gerente para evitar los problemas recién anotados? Reel [Ree99] sugiere un enfoque de sentido común de cinco partes en los proyectos de software:

1. *Comenzar con el pie derecho.* Esto se logra al trabajar duro (muy duro) para entender el problema que debe resolverse y luego establecer objetivos y expectativas realistas para todos aquellos que estarán involucrados en el proyecto. Lo anterior se refuerza al construir el equipo correcto (sección 24.2.3) y darle autonomía, autoridad y tecnología necesarias para realizar el trabajo.

Cita:

"Un proyecto es como un viaje en carretera. Algunos son simples y rutinarios, como conducir hacia la tienda a plena luz del día. Pero la mayoría de los proyectos que vale la pena realizar son más parecidos a conducir una camioneta 4×4 en las montañas y de noche."

Cem Kaner, James Bach y Bret Pettichord

2. *Mantener la cantidad de movimiento.* Muchos proyectos parten hacia un buen comienzo y luego lentamente se desintegran. A fin de mantener la cantidad de movimiento, el gerente de proyecto debe proporcionar incentivos para mantener la rotación de personal en un mínimo absoluto, el equipo debe enfatizar la calidad en cada tarea que realice y el administrador ejecutivo debe hacer todo lo posible para permanecer fuera del camino del equipo.[7]
3. *Siga la pista al progreso.* Para un proyecto de software, el progreso se rastrea conforme los productos operativos (por ejemplo, modelos, código fuente, conjuntos de casos de prueba) se producen y aprueban (usando revisiones técnicas) como parte de una actividad que asegure la calidad. Además, pueden recopilarse medidas de proceso de software y proyecto (capítulo 25) y usarse para valorar el progreso contra promedios desarrollados para la organización de desarrollo del software.

7 La implicación de esta afirmación es que la burocracia se reduce al mínimo, las reuniones extrañas se eliminan y se quita el énfasis a la adhesión dogmática a las reglas de proceso y proyecto. El equipo debe ser autoorganizado y autónomo.

4. *Tome decisiones inteligentes.* En esencia, las decisiones del gerente del proyecto y del equipo de software deben "mantenerse simples". Siempre que sea posible, decida usar software comercial de anaquel, o componentes o patrones de software existentes, así como evitar interfaces a la medida cuando estén disponibles enfoques estándar; decida también identificar y luego evitar los riesgos obvios, y asignar más tiempo del que se considere necesario para tareas complejas y riesgosas (necesitará cada minuto).
5. *Realice un análisis postmortem*. Establezca un mecanismo consistente para extraer lecciones aprendidas por cada proyecto. Evalúe los calendarios planeado y real, recopile y analice métricas de proyecto de software, consiga retroalimentación de los miembros del equipo y de los clientes, y registre los hallazgos en forma escrita.

24.6 El principio W^{5}HH

En un excelente ensayo acerca del proceso de software y los proyectos, Barry Boehm [Boe96] afirma: "necesita un principio de organización que reduzca la escala a fin de proporcionar planes [de proyecto] simples para proyectos simples". Boehm sugiere un enfoque que aborda los objetivos del proyecto, hitos y calendarios, responsabilidades, enfoques administrativos y técnicos, y recursos requeridos. Él lo llama *principio W^{5}HH*, por una serie de preguntas que conducen a una definición de las características clave del proyecto y al plan de proyecto resultante:

¿Cómo se definen las características clave del proyecto?

¿Por qué (why) *se desarrollará el sistema?* Todos los participantes deben valorar la validez de las razones empresariales para el trabajo de software. ¿El propósito de la empresa justifica el gasto de personal, tiempo y dinero?

¿Qué (what) *se hará?* Defina el conjunto de tareas requeridas para el proyecto.

¿Cuándo (when) *se hará?* El equipo establece un calendario de proyecto al identificar cuándo se realizarán las tareas del proyecto y cuándo se alcanzarán los hitos.

¿Quién (who) *es responsable de cada función?* Defina el papel y la responsabilidad de cada miembro del equipo de software.

¿Dónde (where) *se ubicarán en la organización?* No todos los roles y responsabilidades residen dentro de los profesionales del software. Clientes, usuarios y otros participantes también tienen responsabilidades.

¿Cómo (how) *se hará el trabajo, técnica y organizativamente?* Una vez establecido el ámbito del producto, debe definirse una estrategia técnica para el proyecto.

¿Cuánto (how much) *se necesita de cada recurso?* La respuesta a esta pregunta se deriva al desarrollar estimaciones (capítulo 26) con base en las respuestas a las preguntas anteriores.

El principio W^{5}HH de Boehm es aplicable sin importar el tamaño o complejidad de un proyecto de software. Las preguntas anotadas ofrecen un excelente esbozo de la planificación.

24.7 Prácticas cruciales

El Airlie Council[8] desarrolló una lista de "prácticas de software cruciales para administración basada en desempeño". Dichas prácticas "las usan consistentemente, y las consideran cruciales,

8 El Airlie Council incluyó un equipo de expertos en ingeniería de software contratados por el Departamento de Defensa estadounidense para ayudar a desarrollar lineamientos para mejores prácticas en administración de proyectos de software e ingeniería de software. Para conocer más acerca de mejores prácticas, vea **www.swqual.com/newsletter/vol1/no3/vol1no3.html**

proyectos y organizaciones enormemente exitosas cuya 'línea base' para el desempeño es consistentemente mucho mejor que el promedio industrial" [Air99].

Las prácticas cruciales[9] incluyen: administración del proyecto basada en métrica (capítulo 25), estimación empírica de costo y calendario (capítulos 26 y 27), rastreo del valor ganado (capítulo 27), rastreo de defecto contra metas de calidad (capítulos del 14 al 16) y administración consciente del personal (sección 24.2). Cada una de estas prácticas cruciales se aborda a lo largo de las partes 3 y 4 de este libro.

HERRAMIENTAS DE SOFTWARE

Herramientas de software para gerentes de proyecto

Las "herramientas" citadas aquí son genéricas y se aplican a un amplio rango de actividades realizadas por los gerentes de proyecto. En capítulos finales se consideran las herramientas específicas de administración de proyecto (por ejemplo, herramientas de calendarización, de estimación, de análisis de riesgo).

Herramientas representativas:[10]

El Software Program Manager's Network (**www.spmn.com**) desarrolló una herramienta simple llamada *Project Control Panel*, que brinda a los gerentes de proyecto un indicio directo del estado del proyecto. La herramienta tiene "calibradores" muy parecidos a un tablero y se implementa con Microsoft Excel. Está disponible para descarga en **www.spmn.com/products_software.html**

Gantthead.com (**www.gantthead.com/**) desarrolló un conjunto de útiles *listas de comprobación para gerentes de proyecto*.

Ittoolkit.com (**www.ittoolkit.com**) proporciona "una colección de guías de planificación, plantillas de proceso y hojas de trabajo inteligentes" disponible en CD-ROM.

24.8 RESUMEN

La administración de proyectos de software es una actividad sombrilla dentro de la ingeniería de software. Comienza antes de iniciar cualquier actividad técnica y continúa a lo largo del modelado, construcción y despliegue del software de cómputo.

Cuatro P tienen influencia sustancial sobre la administración del proyecto de software: personal, producto, proceso y proyecto. El personal debe organizarse en equipos eficaces, motivados para hacer trabajo de software de alta calidad, y coordinarse para lograr comunicación efectiva. Los requerimientos del producto deben comunicarse de cliente a desarrollador, dividirse (descomponerse) en sus partes constitutivas y ubicarse para su trabajo por parte del equipo de software. El proceso debe adaptarse al personal y al producto. Se selecciona un marco conceptual común al proceso, se aplica un paradigma de ingeniería de software adecuado y se elige un conjunto de tareas de trabajo para realizar el trabajo. Finalmente, el proyecto debe organizarse de forma que permita triunfar al equipo de software.

El elemento esencial en todos los proyectos de software es el personal. Los ingenieros del software pueden organizarse en diferentes estructuras de equipo que van desde las jerarquías tradicionales de control hasta los equipos de "paradigma abierto". Para apoyar el trabajo del equipo, pueden aplicarse varias técnicas de coordinación y comunicación. En general, las revisiones técnicas y la comunicación informal persona a persona tienen más valor para los profesionales.

La actividad de administración del proyecto abarca medición y métricas, estimación y calendarización, análisis de riesgos, rastreo y control. Cada uno de estos temas se considera en los capítulos siguientes.

---

9 Aquí sólo se mencionan aquellas prácticas cruciales asociadas con "integridad del proyecto".

10 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

Problemas y puntos por evaluar

24.1. Con base en la información contenida en este capítulo y en su propia experiencia, desarrolle "diez mandamientos" para empoderar a los ingenieros del software, es decir, elabore una lista de 10 lineamientos que conducirán al personal de software a que trabaje a toda su potencia.

24.2. El modelo de madurez de capacidades del personal (People-CMM) de The Software Engineering Institute, People-CMM echa un vistazo organizado a "áreas prácticas clave" que cultivan buen personal de software. Su instructor le asignará una APC para análisis y resumen.

24.3. Describa tres situaciones de la vida real en las que el cliente y el usuario final sean el mismo. Describa tres situaciones en las que sean diferentes.

24.4. Las decisiones tomadas por los administradores ejecutivos pueden tener un impacto significativo sobre la efectividad de un equipo de ingeniería del software. Proporcione cinco ejemplos para ilustrar que esto es cierto.

24.5. Revise el libro de Weinberg [Wei86] y escriba un resumen, con una extensión de dos a tres páginas, de los temas que deben considerarse al aplicar el modelo MOI.

24.6. Al lector se le asigna una gerencia de proyecto dentro de una organización de sistemas de información. Su labor será construir una aplicación que sea muy similar a otras que su equipo construyó, aunque ésta será más grande y más compleja. Los requerimientos se documentaron ampliamente por parte del cliente. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo de proceso de software elegiría y por qué?

24.7. Al lector se le asigna una gerencia de proyecto para una pequeña compañía de productos de software. Su labor será construir un producto innovador que combine hardware de realidad virtual con software de última generación. Puesto que la competencia para el mismo mercado de entretenimiento es intensa, existe una presión significativa para tener listo el trabajo. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo de proceso de software elegiría y por qué?

24.8. Al lector se le asigna una gerencia de proyecto para una gran compañía de productos de software. Su labor será administrar el desarrollo de la versión de siguiente generación de su software de procesamiento de palabras ampliamente usado. Puesto que la competencia es intensa, se establecieron y anunciaron apretadas fechas límite. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo de proceso de software elegiría y por qué?

24.9. Al lector se le asigna una gerencia de proyecto de software para una compañía que atiende al mundo de la ingeniería genética. Su labor será administrar el desarrollo de un nuevo producto de software que acelerará el ritmo de tipificación genética. El trabajo está orientado a investigación y desarrollo, pero la meta es elaborar un producto dentro del próximo año. ¿Qué estructura de equipo elegiría y por qué? ¿Qué modelo de proceso de software elegiría y por qué?

24.10. Al lector se le ha pedido desarrollar una pequeña aplicación que analice cada curso ofrecido en la universidad y reporte las calificaciones promedio obtenidas en el curso (por un determinado periodo). Exponga el alcance y las limitaciones de este trabajo.

24.11. Haga una descomposición funcional de primer nivel de la función de plantilla de página que se estudió brevemente en la sección 24.3.2.

Lecturas y fuentes de información adicionales

El Project Management Institute (*Guide to the Project management body of Knowledge*, PMI, 2001) abarca todos los aspectos importantes de la administración de proyectos. Bechtold (*Essentials of Software Project Management,* 2a. ed., Management Concepts, 2007), Wysocki (*Effective Software Project Management,* Wiley, 2006), Stellman y Greene (*Applied Software Project Management,* O'Reilly, 2005), y Berkun (*The Art of Project Management,* O'Reilly, 2005) enseñan habilidades básicas y ofrecen lineamientos detallados para todas las tareas de administración de proyectos de software. McConnell (*Professional Software Development*, Addison-Wesley, 2004) ofrece consejo pragmático para lograr "calendarios más cortos, productos de mayor calidad y proyectos más exitosos". Henry (*Software Project Management*, Addison-Wesley, 2003) ofrece consejo del mundo real que es útil para todos los gerentes de proyecto.

Tom DeMarco *et al.* (*Adrenaline Junkies and Template Zombies*, Dorset House, 2008) escribieron un tratamiento comprensivo de los patrones humanos que se encuentran en todo proyecto de software. Una excelente serie de cuatro volúmenes, escrito por Weinberg (*Quality Software Management*, Dorset House, 1992, 1993, 1994, 1996), presenta conceptos de sistemas básicos de pensamiento y administración, explica cómo usar efectivamente las mediciones y aborda la "acción congruente", la habilidad para establecer "ajuste" entre las necesidades del gerente, las necesidades del personal técnico y las necesidades de la empresa. Ello proporcionará información útil a los gerentes novatos y a los expertos. Futrell *et al.* (*Quality Software Project Management*, Prentice-Hall, 2002) presentan un voluminoso tratamiento de la administración de proyectos. Brown *et al.* (*Antipatterns in Project Management*, Wiley, 2000) plantean qué no hacer durante la administración de un proyecto de software.

Brooks (*The Mythical Man-Month*, Anniversary Edition, Addison-Wesley, 1995) actualizó su libro clásico para ofrecer nueva comprensión de los temas de proyecto y administración de software. McConnell (*Software Project Survival Guide*, Microsoft Press, 1997) presenta excelente lineamiento pragmático para quienes deben administrar proyectos de software. Purba y Shah (*How to Manage a Successful Software Project*, 2a. ed., Wiley, 2000) presentan diversos estudios de caso que indican por qué algunos proyectos triunfan y otros fracasan. Bennatan (*On Time Within Budget*, 3a. ed., Wiley, 2000) presenta consejos útiles y lineamientos para gerentes de proyecto de software. Weigers (*Practical Project Initiation*, Microsoft Press, 2007) proporciona lineamientos prácticos para poner en marcha exitosamente un proyecto de software.

Puede argumentarse que el aspecto más importante de la administración del proyecto de software es la administración de personas. Cockburn (*Agile Software Development*, Addison-Wesley, 2002) presenta uno de los mejores análisis del personal de software escrito a la fecha. DeMarco y Lister [DeM98] escribieron el libro definitivo acerca del personal de software y los proyectos del software. Además, en años recientes se publicaron los siguientes libros acerca de la materia y vale la pena examinarlos:

Cantor, M., *Software Leadership: A Guide to Successful Software Development,* Addison-Wesley, 2001.

Carmel, E., *Global Software Teams: Collaborating Across Borders and Time Zones,* Prentice Hall, 1999.

Constantine, L., *Peopleware Papers: Notes on the Human Side of Software,* Prentice Hall, 2001.

Garton, C., y K. Wegryn, *Managing Without Walls,* McPress, 2006.

Humphrey, W. S., *Managing Technical People: Innovation, Teamwork, and the Software Process,* Addison-Wesley, 1997.

Humphrey, W. S., *TSP-Coaching Development Teams,* Addison-Wesley, 2006.

Jones, P. H., *Handbook of Team Design: A Practitioner's Guide to Team Systems Development,* McGraw-Hill, 1997.

Karolak, D. S., *Global Software Development: Managing Virtual Teams and Environments,* IEEE Computer Society, 1998.

Peters, L., *Getting Results from Software Development Teams,* Microsoft Press, 2008.

Whitehead, R., *Leading a Software Development Team,* Addison-Wesley, 2001.

Aun cuando no se relacionan específicamente con el mundo del software, y en ocasiones adolecen de sobresimplificación y gran generalización, los muy vendidos libros de "administración" de Kanter (*Confidence,* Three Rivers Press, 2006), Covy (*The 8th Habit,* Free Press, 2004), Bossidy (*Execution: The Discipline of Getting Things Done,* Crown Publishing, 2002), Drucker (*Management Challenges for the 21st Century,* Harper Business, 1999), Buckingham y Coffman (*First, Break All the Rules: What the World's Greatest Managers Do Differently,* Simon and Schuster, 1999), y Christensen (*The Innovator's Dilemma,* Harvard Business School Press, 1997) enfatizan "nuevas reglas" definidas por una economía rápidamente cambiante. Los títulos más antiguos, como *Who Moved My Cheese?* y *The One-Minute Manager* e *In Search of Excellence*, siguen ofreciendo valiosos elementos que pueden ayudar a administrar personal y proyectos de manera más efectiva.

En internet está disponible una gran variedad de fuentes de información acerca de las métricas de la administración de proyectos de software. Una lista actualizada de referencias existentes en la World Wide Web y que son relevantes para la administración de proyectos de software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

25

MÉTRICAS DE PROCESO Y DE PROYECTO

CONCEPTOS CLAVE

eficiencia en la remoción del defecto (DRE) 584
medición 575
métricas 572
  argumentos para 585
  basada en LOC 577
  calidad del software 582
  establecimiento de un programa 588
  línea de referencia 586
  orientada a función 577
  orientada a objeto 579
  orientadas a caso de uso . . 580
  orientadas a tamaño 576
  proceso 572
  productividad 579
  proyecto 574
  público y privado 573
  webapp 580
punto de función 577

La medición permite ganar comprensión acerca del proceso y del proyecto, al proporcionar un mecanismo de evaluación objetiva. Lord Kelvin dijo alguna vez:

> Cuando puedes medir aquello de lo que hablas y expresarlo en números, sabes algo acerca de ello; pero cuando no puedes medir, cuando no puedes expresarlo en números, tu conocimiento es exiguo e insatisfactorio: puede ser el comienzo del conocimiento; sin embargo, apenas habrás avanzado, en tus pensamientos, hacia la etapa de una ciencia.

La comunidad de la ingeniería del software tomó a pecho las palabras de lord Kelvin. ¡Mas no sin frustración ni con poca controversia!

La medición puede aplicarse al proceso de software con la intención de mejorarlo de manera continua. Puede usarse a través de un proyecto de software para auxiliar en estimación, control de calidad, valoración de productividad y control de proyecto. Finalmente, la medición pueden usarla los ingenieros del software para ayudar en la valoración de la calidad de los productos de trabajo y auxiliar en la toma de decisiones tácticas conforme avanza un proyecto (capítulo 23).

Dentro del contexto del proceso de software y de los proyectos que se realizan usando a aquél, un equipo de software está preocupado principalmente por la productividad y por las métricas de calidad: medidas de "salidas" de desarrollo de software como función del esfuerzo y el tiempo aplicado y medidas de la "aptitud para el uso" de los productos operativos que se producen. Con propósitos de planificación y estimación, el interés es histórico. ¿Cuál fue la productividad en el desarrollo de software en proyectos anteriores? ¿Cuál la calidad del software que se produjo? ¿Cómo pueden extrapolarse al presente los datos de productividad y calidad anteriores? ¿Cómo pueden las mediciones ayudar a planificar y a estimar con más precisión?

En su manual acerca de medición del software, Park, Goethert y Florac [Par96] anotan las razones por las que se mide: 1) para *caracterizar* un esfuerzo y obtener comprensión "de los

UNA MIRADA RÁPIDA

¿Qué es? Las métricas de proceso y proyecto de software son medidas cuantitativas que permiten obtener comprensión acerca de la eficacia del proceso del software y de los proyectos que se realizan, usando el proceso como marco conceptual. Se recopilan datos básicos de calidad y productividad. Luego, se analizan, se comparan con promedios anteriores y se valoran para determinar si han ocurrido mejoras en calidad y productividad. Las métricas también se usan para puntualizar áreas problemáticas, de modo que puedan desarrollarse remedios y el proceso de software pueda mejorarse.

¿Quién lo hace? Las métricas del software se analizan y valoran por parte de los gerentes del software. Con frecuencia, los ingenieros del software recopilan las medidas.

¿Por qué es importante? Si no se mide, el juicio puede basarse solamente en la evaluación subjetiva. Con medición, pueden marcarse las tendencias (buenas o malas), hacerse mejores estimaciones y, con el tiempo, lograrse verdadera mejoría.

¿Cuáles son los pasos? Se define un conjunto limitado de medidas de proceso, proyecto y producto, que son fáciles de recopilar. Dichas medidas con frecuencia se normalizan usando métricas de tamaño o de función. El resultado se analiza y compara con promedios anteriores para proyectos similares realizados dentro de la organización. Las tendencias se valoran y se generan conclusiones.

¿Cuál es el producto final? Un conjunto de métricas de software que proporcionan comprensión acerca del proceso y del proyecto.

¿Cómo me aseguro de que lo hice bien? Al aplicar un esquema de medición consistente, aunque simple, que nunca debe usarse para valorar, recompensar o castigar el desempeño individual.

procesos, productos, recursos y entornos, y establecer líneas de referencia para comparar con valoraciones futuras"; 2) para *evaluar* y "determinar el estado de avance con respecto a los planes"; 3) para *predecir* al "obtener comprensión de las relaciones entre procesos y productos, y construir modelos de dichas relaciones", y 4) para *mejorar* al "identificar barricadas, causas raíz, ineficiencias y otras oportunidades para mejorar la calidad del producto y el desempeño del proceso".

La medición es una herramienta administrativa. Si se realiza adecuadamente, ofrece entendimiento al gerente de un proyecto. Y, como resultado, lo auxilia a él y al equipo de software para tomar decisiones que conducirán hacia un proyecto exitoso.

25.1 Métricas en los dominios de proceso y proyecto

PUNTO CLAVE

La métrica de proceso tiene impacto a largo plazo. Su intención es mejorar el proceso en sí. La métrica de proyecto usualmente contribuye al desarrollo de la primera.

Las *métricas de proceso* se recopilan a través de todos los proyectos y durante largos espacios de tiempo. Su intención es proporcionar un conjunto de indicadores de proceso que conduzca a mejorar el proceso de software a largo plazo. Las *métricas de proyecto* permiten al gerente de un proyecto de software: 1) valorar el estado de un proyecto en marcha, 2) rastrear riesgos potenciales, 3) descubrir áreas problema antes de que se vuelvan "críticas", 4) ajustar el flujo de trabajo o las tareas y 5) evaluar la habilidad del equipo del proyecto para controlar la calidad de los productos operativos del software.

Las medidas que recopila un equipo de proyecto y que convierte en métricas para uso durante un proyecto también pueden transmitirse a quienes tienen responsabilidad en la mejora del proceso de software (capítulo 30). Por esta razón, muchas de las métricas se usan tanto en los dominios del proceso como en los del proyecto.

25.1.1 Las métricas del proceso y la mejora del proceso de software

La única forma racional para mejorar cualquier proceso es medir atributos específicos del mismo, desarrollar un conjunto de métricas significativas con base en dichos atributos y luego usarlas para proporcionar indicadores que conducirán a una estrategia para mejorar (capítulo 30). Pero antes de estudiar las métricas del software y su impacto sobre el mejoramiento del proceso de software, es importante observar que el proceso sólo es uno de varios "factores controlables en el mejoramiento de la calidad del software y del desempeño organizativo" [Pau94].

PUNTO CLAVE

La habilidad y motivación del personal del software que hace el trabajo son los factores más importantes que influyen en la calidad del software.

En la figura 25.1, el proceso se asienta en el centro de un triángulo que conecta tres factores que tienen profunda influencia sobre la calidad del software y en el desempeño de la organización. La habilidad y motivación del personal ha demostrado [Boe81] ser el factor individual más influyente en la calidad y el desempeño. La complejidad del producto puede tener un impacto sustancial sobre la calidad y el desempeño del equipo. La tecnología (es decir, los métodos y herramientas de la ingeniería del software) que puebla el proceso también tiene un impacto.

Además, existe el triángulo de proceso dentro de un círculo de condiciones ambientales que incluyen entorno de desarrollo (por ejemplo, herramientas de software integradas), condiciones empresariales (fechas límite, reglas empresariales) y características del cliente (facilidad de comunicación y colaboración).

Cita:

"Las métricas de software le permiten saber cuándo reír y cuándo llorar."

Tom Gilb

La eficacia de un proceso de software sólo puede medirse de manera indirecta. Esto significa que es posible derivar un conjunto de métricas con base en los resultados que pueden derivarse del proceso. Los resultados incluyen medidas de los errores descubiertos antes de liberar el software, defectos entregados a y reportados por usuarios finales, productos operativos entregados (productividad), esfuerzo humano empleado, tiempo calendario consumido, conformidad con la agenda y otras medidas. También pueden derivarse métricas de proceso al medir las características de tareas de ingeniería de software específicas. Por ejemplo, puede medirse el

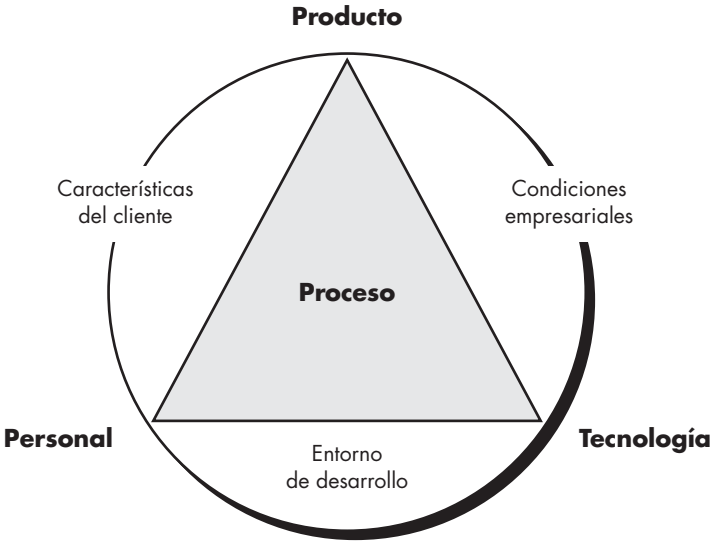


FIGURA 25.1
Determinantes para calidad del software y efectividad organizativa.
Fuente: Adaptado de [Pau94].

esfuerzo y el tiempo empleados al realizar las actividades sombrilla y las actividades genéricas de ingeniería del software descritas en el capítulo 2.

Grady [Gra92] argumenta que existen usos "privados y públicos" para diferentes tipos de datos de proceso. Puesto que es natural que los ingenieros de software individual puedan ser sensibles al uso de las métricas recopiladas de manera individual, dichos datos deben ser privados para el individuo y funcionar sólo como un indicio para él. Los ejemplos de *métricas privadas* incluyen tasas de defecto (por individuo), tasas de defecto (por componente) y errores que se encuentran durante el desarrollo.

? ¿Cuál es la diferencia entre usos privado y público para las métricas del software?

La filosofía de "datos de proceso privados" se conforma bien con el enfoque de proceso de software personal (capítulo 2) propuesto por Humphrey [Hum97], quien reconoce que el mejoramiento en el proceso del software puede y debe comenzar en el nivel individual. Los datos de proceso privado pueden funcionar como un importante motor conforme se trabaja para mejorar el enfoque de la ingeniería del software.

Algunas métricas de proceso son privadas para el equipo de proyecto del software, pero públicas para todos los miembros del equipo. Los ejemplos incluyen defectos reportados por grandes funciones de software (que se desarrollaron por parte de algún número de profesionales), errores encontrados durante las revisiones técnicas y líneas de código o puntos de función por componente o función.[1] El equipo revisa dichos datos para descubrir indicios que puedan mejorar el desempeño del equipo.

Las métricas públicas por lo general asimilan información que originalmente era privada para los individuos y el equipo. Las tasas de defecto en el nivel de proyecto (absolutamente no atribuidas a un individuo), esfuerzo, tiempos calendario y datos relacionados se recopilan y evalúan con la intención de descubrir indicios que puedan mejorar el desempeño del proceso organizativo.

Las métricas de proceso de software pueden proporcionar beneficios significativos conforme una organización trabaja para mejorar su nivel global de madurez de proceso. Sin embargo, como todas las métricas, éstas pueden tener mal uso, lo que crea más problemas de los que resuelven. Grady [Gra92] sugiere una "etiqueta de métrica de software" que sea adecuada tanto para gerentes como para profesionales, conforme instauran un programa de métricas de proceso:

1 Las líneas de código y las métricas de punto de función se estudian en las secciones 25.2.1 y 25.2.2.

- Usar el sentido común y sensibilidad organizacional cuando se interpreten datos de métricas.
- Proporcionar retroalimentación regular a los individuos y equipos que recopilan medidas y métricas.
- No usar métricas para valorar a los individuos.
- Trabajar con los profesionales y con los equipos para establecer metas y métricas claras que se usarán para lograr las primeras.
- Nunca usar métricas para amenazar a los individuos o a los equipos.
- No considerar "negativos" los datos de métricas que indiquen un área problemática. Dichos datos simplemente son un indicio para mejorar el proceso.
- No obsesionarse con una sola métrica ni excluir otras métricas importantes.

¿Qué lineamientos deben aplicarse cuando se recopilan métricas de software?

Conforme una organización se siente más cómoda con la recolección y uso de métricas de proceso, la derivación de los indicadores simples da lugar a un enfoque más riguroso llamado *mejora estadística de proceso de software* (MEPS). En esencia, MEPS usa análisis de falla del software para recopilar información acerca de todos los errores y defectos[2] que se encuentren conforme se desarrolle y use una aplicación, sistema o producto.

25.1.2 Métricas de proyecto

A diferencia de las métricas de proceso de software que se usaron con propósitos estratégicos, las medidas de proyecto de software son tácticas. Es decir, el gerente de proyecto y un equipo de software usan las métricas de proyecto y los indicadores derivados de ellas para adaptar el flujo de trabajo del proyecto y las actividades técnicas.

La primera aplicación de las métricas de proyecto sobre la mayoría de los proyectos de software ocurre durante la estimación. Las métricas recopiladas de proyectos anteriores se usan como la base desde la cual se hacen estimaciones de esfuerzo y tiempo para el trabajo de software nuevo. Conforme avanza un proyecto, las medidas de esfuerzo y tiempo calendario utilizadas se comparan con las estimaciones originales (y con la agenda del proyecto). El gerente del proyecto usa dichos datos para monitorear y controlar el progreso.

Mientras comienza el trabajo técnico, otras métricas del proyecto empiezan a tener significado. Se miden las tasas de producción representadas en términos de modelos creados, horas de revisión, puntos de función y líneas de fuente entregadas. Además, se rastrean los errores descubiertos durante cada tarea de ingeniería del software. Conforme el software evoluciona desde los requerimientos hasta el diseño, se recopilan métricas técnicas (capítulo 23) a fin de valorar la calidad del diseño y proporcionar indicios que influirán en el enfoque tomado para generación y prueba de código.

¿Cómo deben usarse las métricas durante el proyecto en sí?

La intención de las métricas de proyecto es doble. Primero, se usan para minimizar el calendario de desarrollo al hacer los ajustes necesarios para evitar demoras y mitigar potenciales problemas y riesgos. Segundo, se usan para valorar la calidad del producto sobre una base en marcha y, cuando es necesario, modificar el enfoque técnico para mejorar la calidad.

Conforme la calidad mejora, los defectos se minimizan, y conforme el conteo de defectos baja, la cantidad de reelaboración requerida durante el proyecto también se reduce. Esto conduce a una reducción en el costo global del proyecto.

---

2 En este libro, un *error* se define como un fallo en un producto operativo de la ingeniería del software que se descubre *antes* de que el software se entregue al usuario final. Un *defecto* es un fallo que se descubre *después* de entregar el software al usuario final. Debe destacarse que otros no hacen esta distinción.

CASASEGURA

Establecimiento de un enfoque de métricas

La escena: Oficina de Doug Miller cuando el proyecto de software *CasaSegura* está a punto de comenzar.

Participantes: Doug Miller (gerente del equipo de ingeniería del software *CasaSegura*) y Vinod Raman y Jamie Lazar, miembros del equipo de ingeniería de software del producto.

La conversación:

Doug: Antes de empezar a trabajar en este proyecto, me gustaría que definieran y recopilaran un conjunto de métricas simples. Para comenzar, tendrán que definir sus metas.

Vinod (frunce el ceño): Nunca hemos hecho esto antes y...

Jamie (interrumpe): Y con base en la administración de la línea de tiempo de la que hablaste, nunca tendremos el tiempo. De cualquier forma, ¿qué bien hacen las métricas?

Doug (levanta su mano para detener el embate): Cálmense y respiren, chicos. El hecho de que nunca lo hayamos hecho antes es la principal razón para comenzar ahora, y el trabajo de las métricas del que hablo de ninguna manera debe tardar mucho tiempo... de hecho, sólo puede ahorrarnos tiempo.

Vinod: ¿Cómo?

Doug: Miren, vamos a hacer mucho más trabajo interno en ingeniería del software conforme nuestros productos se vuelvan más inteligentes, se habiliten en web, todo eso... y necesitamos entender el proceso que usamos para construir software... y mejorarlo de modo que podamos construir software de mejor manera. La única forma de hacer esto es medir.

Jamie: Pero estamos bajo presión de tiempo, Doug. No estoy a favor de más papeleo... necesitamos el tiempo para hacer nuestro trabajo, no para recolectar datos.

Doug (con calma): Jamie, el trabajo de un ingeniero involucra recopilar datos, evaluarlos y usar los resultados para mejorar el producto y el proceso. ¿Me equivoco?

Jamie: No, pero...

Doug: ¿Y si mantenemos el número de medidas que recopilemos en no más de cinco o seis y nos enfocamos en la calidad?

Vinod: Nadie puede estar en contra de la alta calidad...

Jamie: Cierto... pero, no sé. Todavía creo que no es necesario.

Doug: Voy a pedirte que me complazcas en esto. ¿Cuánto saben acerca de las métricas del software?

Jamie (mira a Vinod): No mucho.

Doug: Aquí hay algunas referencias en la red... pasé algunas horas recuperándolas para avanzar.

Jamie (sonríe): Creo que dijiste que esto no tomaría tiempo.

Doug: El tiempo que se emplea aprendiendo nunca se desperdicia... háganlo y luego establezcan algunas metas, planteen algunas preguntas y definan la métrica que necesitamos recopilar.

25.2 MEDICIÓN DEL SOFTWARE

En el capítulo 23 se indicó que las mediciones en el mundo físico pueden clasificarse en dos formas: medidas directas (por ejemplo, la longitud de un tornillo) y medidas indirectas (por ejemplo, la "calidad" de los tornillos producidos, medidos por conteo de rechazos). Las métricas de software pueden clasificarse de igual modo.

Cita:

"No todo lo que puede contarse cuenta y no todo lo que cuenta puede contarse."

Albert Einstein

Las *medidas directas* del proceso de software incluyen costo y esfuerzo aplicado. Las medidas directas del producto incluyen líneas de código (LOC) producidas, rapidez de ejecución, tamaño de memoria y defectos reportados sobre cierto espacio de tiempo. Las medidas indirectas del producto incluyen funcionalidad, calidad, complejidad, eficiencia, confiabilidad, capacidad de mantenimiento y muchas otras "habilidades" que se estudiaron en el capítulo 14.

El costo y el esfuerzo requeridos para construir software, el número de líneas de código producidas y otras medidas directas son relativamente sencillos de recolectar, en tanto se establezcan por adelantado convenciones específicas para la medición. Sin embargo, la calidad y funcionalidad del software o su eficiencia o capacidad de mantenimiento son más difíciles de valorar y pueden medirse sólo de manera indirecta.

CONSEJO

Puesto que muchos factores afectan el trabajo de software, no use métricas para comparar individuos o equipos.

El dominio de la métrica del software se dividió en métricas de proceso, proyecto y producto, y se dijo que las métricas de producto que son privadas para un individuo con frecuencia se combinan para desarrollar métricas de proyecto que son públicas para un equipo de software. Luego las métricas de proyecto se consolidan para crear métricas de proceso que son públicas para la organización del software como un todo. Pero, ¿cómo combina una organización las métricas que vienen de diferentes individuos o proyectos?

FIGURA 25.2
Métricas orientadas a tamaño

Proyecto	LOC	Esfuerzo	$(000)	Pp. doc.	Errores	Defectos	Personal
alfa	12 100	24	168	365	134	29	3
beta	27 200	62	440	1 224	321	86	5
gamma	20 200	43	314	1 050	256	64	6
•	•	•	•	•	•		
•	•	•	•	•	•		
•	•	•	•	•	•		

Para ilustrar, considere un ejemplo simple. Los individuos que trabajan en dos equipos de proyecto diferentes registran y categorizan todos los errores que encuentran durante el proceso de software. Las medidas individuales luego se combinan para desarrollar medidas de equipo. El equipo A encuentra 342 errores durante el proceso de software antes de la liberación. El equipo B encuentra 184 errores. Si todas las demás cosas permanecen iguales, ¿cuál equipo es más efectivo para descubrir errores a lo largo del proceso? Dado que no se conoce el tamaño o complejidad de los proyectos, no puede responderse esta pregunta. Sin embargo, si las medidas se normalizan, es posible crear métricas de software que permitan la comparación con promedios organizacionales más amplios.

25.2.1 Métricas orientadas a tamaño

Las métricas de software orientadas a tamaño se derivan al normalizar las medidas de calidad y/o productividad para considerar el *tamaño* del software que se produjo. Si una organización de software mantiene registros simples, puede crearse una tabla de medidas orientadas a tamaño, como la que se muestra en la figura 25.2. La tabla menciona cada proyecto de desarrollo de software que se completó durante los años anteriores y que corresponden a medidas para dicho proyecto. En la entrada de la tabla (figura 25.2) para el proyecto alfa: 12 100 líneas de código se desarrollaron con 24 persona-meses de esfuerzo a un costo de US$168 000. Debe observarse que los registros de esfuerzo y código que aparecen en la tabla representan todas las actividades de ingeniería del software (análisis, diseño, código y prueba), no sólo codificación. Más información para el proyecto alfa indica que: se desarrollaron 365 páginas de documentación, se registraron 134 errores antes de liberar el software y se encontraron 29 defectos después de liberarlo al cliente, dentro del primer año de operación. Tres personas trabajaron en el desarrollo del software para el proyecto alfa.

Con la finalidad de desarrollar métricas que puedan asimilarse con métricas similares de otros proyectos, pueden elegirse líneas de códigos como un valor de normalización. A partir de los rudimentarios datos contenidos en la tabla, pueden desarrollarse métricas simples orientadas a tamaño para cada proyecto:

- Errores por KLOC (miles de líneas de código).
- Defectos por KLOC.
- $ por KLOC.
- Páginas de documentación por KLOC.

Además, es posible calcular otras métricas interesantes:

- Errores por persona-mes.
- KLOC por persona-mes.
- $ por página de documentación.

PUNTO CLAVE

Las métricas orientadas a tamaño se usan ampliamente, pero continúa el debate acerca de su validez y aplicabilidad.

Las métricas orientadas a tamaño no se aceptan universalmente como la mejor forma de medir el proceso de software. La mayor parte de la controversia gira en torno del uso de líneas de código como medida clave. Quienes proponen la medida LOC afirman que las LOC son un "artefacto" de todos los proyectos de desarrollo de software y que pueden contarse fácilmente; que muchos modelos existentes de estimación de software usan LOC o KLOC como entrada clave y que ya existe un gran cuerpo de literatura y predicado de datos acerca de LOC. Por otra parte, los opositores argumentan que las medidas LOC dependen del lenguaje de programación; que cuando se considera la productividad, castigan a los programas bien diseñados pero cortos; que no pueden acomodarse con facilidad en lenguajes no procedurales y que su uso en la estimación requiere un nivel de detalle que puede ser difícil de lograr (es decir, el planificador debe estimar las LOC que se van a producir mucho antes de completar el análisis y el diseño).

25.2.2 Métricas orientadas a función

Las métricas de software orientadas a función usan una medida de la funcionalidad entregada por la aplicación como un valor de normalización. La métrica orientada a función de mayor uso es el punto de función (PF). El cálculo del punto de función se basa en características del dominio y de la complejidad de información del software. La mecánica del cálculo del PF se estudió en el capítulo 23.[3]

El punto de función, como la medida LOC, es controvertido. Quienes lo proponen afirman que el PF es independiente del lenguaje de programación, lo que lo hace ideal para aplicaciones que usan lenguajes convencionales y no procedurales, y que se basa en datos que es más probable que se conozcan tempranamente en la evolución de un proyecto, lo que hace al PF más atractivo como enfoque de estimación. Sus opositores afirman que el método requiere cierta "maña", pues dicho cálculo se basa en datos subjetivos más que objetivos, que el conteo del dominio de información (y otras dimensiones) puede ser difícil de recopilar después del hecho y que el PF no tiene significado físico directo: es sólo un número.

25.2.3 Reconciliación de métricas LOC y PF

La relación entre líneas de código y puntos de función depende del lenguaje de programación que se use para implementar el software y la calidad del diseño. Algunos estudios intentan relacionar las medidas PF y LOC. La tabla[4] [QSM02], que se presenta en la página siguiente, ofrece estimaciones burdas del número promedio de líneas de código requeridas para construir un punto de función en varios lenguajes de programación.

Una revisión de estos datos indica que un LOC de C++ proporciona aproximadamente 2.4 veces la "funcionalidad" (como promedio) que un LOC de C. Más aún, un LOC de Smalltalk proporciona al menos cuatro veces la funcionalidad de un LOC para un lenguaje de programación convencional, como Ada, COBOL o C. Al usar la información contenida en la tabla, es posible "retroactivar" [Jon98] el software existente para estimar el número de puntos de función, una vez conocido el número total de enunciados del lenguaje de programación.

---

3 Vea la sección 23.2.1 para un análisis detallado del cálculo de PF.

4 Usado con permiso de Quantitative Software Management (www.qsm.com), copyright 2002.

	LOC por punto de función			
Lenguaje de programación	**Promedio**	**Mediana**	**Bajo**	**Alto**
Access	35	38	15	47
Ada	154	—	104	205
APS	86	83	20	184
ASP 69	62	—	32	127
Ensamblador	337	315	91	694
C	162	109	33	704
C++	66	53	29	178
Clipper	38	39	27	70
COBOL	77	77	14	400
Cool:Gen/IEF	38	31	10	180
Culprit	51	—	—	—
DBase IV	52	—	—	—
Easytrieve+	33	34	25	41
Excel47	46	—	31	63
Focus	43	42	32	56
FORTRAN	—	—	—	—
FoxPro	32	35	25	35
Ideal	66	52	34	203
IEF/Cool:Gen	38	31	10	180
Informix	42	31	24	57
Java	63	53	77	—
JavaScript	58	63	42	75
JCL	91	123	26	150
JSP	59	—	—	—
Lotus Notes	21	22	15	25
Mantis	71	27	22	250
Mapper	118	81	16	245
Natural	60	52	22	141
Oracle	30	35	4	217
PeopleSoft	33	32	30	40
Perl	60	—	—	—
PL/1	78	67	22	263
Powerbuilder	32	31	11	105
REXX	67	—	—	—
RPG II/III	61	49	24	155
SAS	40	41	33	49
Smalltalk	26	19	10	55
SQL	40	37	7	110
VBScript36	34	27	50	—
Visual Basic	47	42	16	158

Las medidas LOC y PF se usan frecuentemente para calcular métricas de productividad. Esto invariablemente conduce a un debate acerca del uso de tales datos. ¿El LOC/persona-mes (o PF/persona-mes) de un grupo debe compararse con datos similares de otro? ¿Los gerentes deben valorar el desempeño individual usando dichas métricas? La respuesta a estas preguntas es un enfático *¡NO!* La razón para esta respuesta es que muchos factores influyen en la producti-

vidad, lo que hace que las comparaciones entre "manzanas y naranjas" se malinterpreten con facilidad.

Es cierto que los puntos de función y las métricas basadas en LOC son predictores relativamente precisos del esfuerzo y del costo en el desarrollo del software. Sin embargo, si se van a usar LOC y PF para estimación (capítulo 26), debe establecerse una línea de referencia de información.

Dentro del contexto de las métricas de proceso y proyecto, la preocupación debe estar centrada principalmente en la productividad y la calidad, medidas de "salida" del desarrollo del software como función del esfuerzo y el tiempo aplicados y medidas de "aptitud para el uso" de los productos operativos que se producen. Con propósitos de mejorar el proceso y la planificación del proyecto, su interés es histórico. ¿Cuál fue la productividad en el desarrollo del software en proyectos anteriores? ¿Cuál fue la calidad del software que se produjo? ¿Cómo pueden extrapolarse al presente los datos de productividad y calidad anteriores? ¿Cómo puede ayudar a mejorar el proceso y la planificación de nuevos proyectos con más precisión?

25.2.4 Métricas orientadas a objeto

Las métricas de proyecto de software convencional (LOC o PF) pueden usarse para estimar proyectos de software orientados a objeto. Sin embargo, dichas métricas no proporcionan suficiente granularidad para los ajustes de calendario y esfuerzo que se requieren conforme se repite a través de un proceso evolutivo o incremental. Lorenz y Kidd [Lor94] sugieren el siguiente conjunto de métricas para proyectos OO:

Número de guiones de escenario. Un guión de escenario (análogo a los casos de uso estudiados a través de la parte 2 de este libro) es una secuencia detallada de pasos que describen la interacción entre el usuario y la aplicación. Cada guión se organiza en tripletas de la forma

{**iniciador,** *acción,* **participante**}

donde **iniciador** es el objeto que solicita cierto servicio (que inicia un mensaje), *acción* es el resultado de la solicitud y **participante** es el objeto servidor que satisface la solicitud. El número de guiones de escenario se relaciona directamente con el tamaño de la aplicación y con el número de casos de prueba que deben desarrollarse para ejercitar el sistema una vez construido.

No es raro para guiones de escenario múltiple mencionar la misma funcionalidad u objetos de datos. Por tanto, tenga cuidado cuando use conteo de guión. Muchos guiones en ocasiones pueden reducirse a una sola clase o conjunto de código.

Número de clases clave. Las *clases clave* son los "componentes enormemente independientes" [Lor94] que se definen tempranamente en el análisis orientado a objeto (capítulo 6).[5] Puesto que las clases clave son centrales en el dominio del problema, el número de tales clases es un indicio de la cantidad de esfuerzo requerido para desarrollar el software y también de la cantidad potencial de reuso por aplicar durante el desarrollo del sistema.

Número de clases de apoyo. Las *clases de apoyo* se requieren para implementar el sistema, pero no se relacionan de inmediato con el dominio del problema. Los ejemplos pueden ser clases de interfaz de usuario (GUI), clases de acceso y manipulación de base de datos y clases de cálculo. Además, es posible desarrollar clases de apoyo para cada una de las clases clave. Las clases de apoyo se definen de manera iterativa a lo largo de un proceso evolutivo. El número de clases de apoyo es un indicio de la cantidad de esfuerzo requerido para desarrollar el software y también de la potencial cantidad de reuso que se va a aplicar durante el desarrollo del sistema.

Las clases pueden variar en tamaño y complejidad. Por tanto, vale la pena considerar la clasificación del conteo de clase por tamaño y complejidad.

Número promedio de clases de apoyo por clase clave. En general, las clases clave se conocen tempranamente en el proyecto. Las clases de apoyo se definen a todo lo largo del mismo. Si el número promedio de clases de apoyo por clase clave se conoce para un dominio de problema determinado, la estimación (con base en el número total de clases) se simplificará

5 En la parte 2 de este libro, se hace referencia a clases clave como *clases de análisis*.

enormemente. Lorenz y Kidd sugieren que las aplicaciones con una GUI tienen entre dos y tres veces más clases de apoyo que clases clave. Las aplicaciones no GUI tienen entre una y dos veces más clases de apoyo que clases clave.

Número de subsistemas. Un *subsistema* es un agregado de clases que apoyan una función que es visible para el usuario final de un sistema. Una vez identificados los subsistemas, es más fácil plantear un calendario razonable en el cual el trabajo sobre los subsistemas se divide entre el personal del proyecto.

Para usarse de manera efectiva en un entorno de ingeniería del software orientado a objeto, es necesario recopilar métricas similares a las anotadas anteriormente, junto con medidas del proyecto, tales como esfuerzo empleado, errores y defectos descubiertos, y modelos o páginas de documentación producidos. Conforme crece la base de datos (después de haber completado algunos proyectos), las relaciones entre las medidas orientadas a objeto y las medidas del proyecto proporcionarán métricas que pueden auxiliar en la estimación del proyecto.

25.2.5 Métricas orientadas a caso de uso

Los casos de uso[6] se utilizan ampliamente como un método para describir los requerimientos en el dominio en el nivel del cliente o empresarial, que implican características y funciones del software. Parecería razonable usar el caso de uso como una medida de normalización similar a LOC o PF. Como los PF, el caso de uso se define al principio del proceso del software, lo que permite emplearlo para estimación antes de iniciar actividades significativas de modelado y construcción. Los casos de uso describen (de manera indirecta, al menos) las funciones y características visibles para el usuario que son requisitos básicos para un sistema. El caso de uso es independiente del lenguaje de programación. Además, el número de casos de uso es directamente proporcional al tamaño de la aplicación en LOC y al número de casos de prueba que tendrán que designarse para ejercitar por completo la aplicación.

Puesto que los casos de uso pueden crearse en niveles de abstracción enormemente diferentes, no hay "tamaño" estándar para un caso de uso. Sin una medida estándar de lo que es un caso de uso, su aplicación como medida de normalización (por ejemplo, esfuerzo empleado por caso de uso) causa suspicacia.

Los investigadores sugieren *puntos de caso de uso* (PCU) como un mecanismo para estimar el esfuerzo del proyecto y otras características. Los PCU son una función del número de actores y transacciones implicados por los modelos de caso de uso y su análogo al PF en algunas formas. Si se tiene más interés en este tema, véase [Cle06].

25.2.6 Métricas de proyecto *webapp*

El objetivo de todos los proyectos *webapp* es entregar al usuario final una combinación de contenido y funcionalidad. Las medidas y métricas usadas para proyectos tradicionales de ingeniería del software son difíciles de traducir directamente a *webapps*. Sin embargo, es posible desarrollar una base de datos que permita el acceso a medidas productivas y calidad internas derivadas de algunos proyectos. Entre las medidas que pueden recopilarse están:

Número de páginas web estáticas. Las páginas web con contenido estático (es decir, el usuario final no tiene control sobre el contenido que se despliega en la página) son las más comunes de todas las características *webapp*. Dichas páginas representan complejidad relativamente baja y por lo general requieren menos esfuerzo para construirlas que las páginas dinámicas. Esta medida proporciona un indicio del tamaño global de la aplicación y del esfuerzo requerido para desarrollarla.

---

6 Los casos de uso se introdujeron en los capítulos 5 y 6.

Número de páginas web dinámicas. Las páginas web con contenido dinámico (es decir, acciones del usuario final u otros factores externos dan como resultado contenido personalizado que se despliega en la página) son esenciales en toda aplicación de comercio electrónico, motores de búsqueda, aplicaciones financieras y muchas otras categorías *webapp*. Dichas páginas representan complejidad relativa más alta y requieren más esfuerzo para construirlas que las páginas estáticas. Esta medida proporciona un indicio del tamaño global de la aplicación y del esfuerzo requerido para desarrollarla.

Número de vínculos de página internos. Los vínculos de página internos son punteros que proporcionan un hipervínculo hacia alguna otra página web dentro de la *webapp*. Esta medida proporciona un indicio del grado de acoplamiento arquitectónico dentro de la *webapp*. Conforme el número de vínculos de página aumenta, el esfuerzo empleado en el diseño y construcción de la navegación también aumenta.

Número de objetos de datos persistentes. Una *webapp* puede tener acceso a uno o más objetos de datos persistentes (por ejemplo, una base de datos o archivo de datos). Conforme crece el número de objetos de datos persistentes, la complejidad de la *webapp* también crece y el esfuerzo para implementarla lo hace de manera proporcional.

Número de sistemas externos puestos en interfaz. Con frecuencia, las *webapps* deben tener interfaces con aplicaciones empresariales "de puerta trasera". Conforme crecen los requerimientos para interfaces, también crecen la complejidad y el desarrollo del sistema.

Número de objetos de contenido estáticos. Los objetos de contenido estáticos abarcan información basada en texto, gráfica, video, animación y audioestática, que se incorporan dentro de la *webapp*. Múltiples objetos de contenido pueden aparecer en una sola página web.

Número de objetos de contenido dinámicos. Los objetos de contenido dinámicos se generan con base en las acciones del usuario final y abarcan información basada en texto, gráfica, video, animación y audio, generada internamente, que se incorpora dentro de la *webapp*. Múltiples objetos de contenido pueden aparecer en una sola página web.

Número de funciones ejecutables. Una función ejecutable (por ejemplo, un guión o applet) proporciona cierto servicio computacional al usuario final. Conforme el número de funciones ejecutables aumenta, también crece el esfuerzo de modelado y construcción.

Cada una de las medidas anteriores puede determinarse en una etapa relativamente temprana. Por ejemplo, puede definirse una métrica que refleje el grado de personalización del usuario final que se requiere para la *webapp* y correlacionarla con el esfuerzo empleado en el proyecto y/o los errores descubiertos conforme se realizan revisiones y se aplican pruebas. Para lograr esto, se define

N_{pe} = número de páginas web estáticas
N_{pd} = número de páginas web dinámicas

Entonces,

$$\text{Índice de personalización, } C = \frac{N_{pd}}{N_{pd} + N_{pe}}$$

El valor de C varía de 0 a 1. Conforme C se hace más grande, el nivel de personalización de la *webapp* se convierte en un problema técnico considerable.

Similares métricas *webapp* pueden calcularse y correlacionarse con medidas del proyecto, como esfuerzo empleado, errores y defectos descubiertos y modelos o páginas de documenta-

ción producidos. Conforme crece la base de datos (después de completar algunos proyectos), las relaciones entre las medidas de la *webapp* y las medidas del proyecto proporcionarán indicadores que pueden auxiliar en la estimación del proyecto.

HERRAMIENTAS DE SOFTWARE

Métricas del proyecto y del proceso

Objetivo: Auxiliar en la definición, recolección, evaluación y reporte de medidas y métricas de software.

Mecánica: Cada herramienta varía en su aplicación, pero todas ofrecen mecanismos para recolectar y evaluar datos que conducen al cálculo de métricas de software.

Herramientas representativas:[7]

Function Point WORKBENCH, desarrollada por Charismatek (**www.charismatek.com.au**), ofrece un amplio arreglo de métricas orientadas a PF.

MetricCenter, desarrollada por Distributive Software (**www.distributive.com**), respalda procesos de automatización para recolección de datos, análisis, formateo de gráficos, generación de reportes y otras tareas de medición.

PSM Insight, desarrollada por Practical Software and Systems Measurement (**www.psmsc.com**), auxilia en la creación y en el análisis posterior de una base de datos de medición de proyecto.

SLIM tool set, desarrollada por QSM (**www.qsm.com**), proporciona un conjunto exhaustivo de métricas y herramientas de estimación.

SPR tool set, desarrollada por Software Productivity Research (**www.spr.com**), ofrece una colección exhaustiva de herramientas orientadas a PF.

TychoMetrics, desarrollada por Predicate Logic, Inc. (**www.predicate.com**), es una suite de herramientas que sirven para recopilar y reportar métricas de administración.

25.3 MÉTRICAS PARA CALIDAD DE SOFTWARE

El software es una entidad compleja. Por tanto, deben esperarse errores conforme se desarrollen productos operativos. Las métricas de proceso tienen la intención de mejorar el proceso del software, de modo que los errores se descubran en la forma más efectiva.

La meta dominante de la ingeniería del software es producir un sistema, aplicación o producto de alta calidad dentro de un marco temporal que satisfaga una necesidad de mercado. Para lograr esta meta, deben aplicarse métodos efectivos acoplados con herramientas modernas dentro del contexto de un proceso de software maduro. Además, un buen ingeniero de software (y los buenos gerentes de ingeniería del software) deben medir si la alta calidad es realizable.

La calidad de un sistema, aplicación o producto sólo es tan buena como los requerimientos que describen el problema, el diseño que modela la solución, el código que conduce a un programa ejecutable y las pruebas que ejercitan el software para descubrir errores. Conforme el software se somete a ingeniería, pueden usarse mediciones para valorar la calidad de los modelos de requerimientos y de diseño, el código fuente y los casos de prueba que se crearon. Para lograr esta valoración en tiempo real, las métricas de producto (capítulo 23) se aplican a fin de evaluar la calidad de los productos operativos de la ingeniería del software en forma objetiva, en lugar de subjetiva.

Un gerente de proyecto también debe evaluar la calidad conforme avanza el proyecto. Las métricas privadas recopiladas individualmente por los ingenieros del software se combinan para proporcionar resultados en el nivel del proyecto. Aunque muchas medidas de calidad pueden recopilarse, el empuje primario en el nivel del proyecto es medir los errores y defectos. Las métricas derivadas de estas medidas proporcionan un indicio de la efectividad de las actividades, individuales y grupales, de aseguramiento y control de la calidad del software.

Métricas como errores de producto operativo por punto de función, errores descubiertos por hora de revisión y errores descubiertos por prueba por hora proporcionan comprensión de la eficacia de cada una de las actividades implicadas por la métrica. Los datos de error también

7 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas existentes en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

pueden usarse para calcular la *eficiencia de remoción de defecto* (ERD) para cada actividad de marco conceptual del proceso. La ERD se estudia en la sección 25.3.3.

25.3.1 Medición de la calidad

Aunque existen muchas medidas de calidad del software,[8] la exactitud, capacidad de mantenimiento, integridad y usabilidad proporcionan útiles indicadores para el equipo del proyecto. Gilb [Gil88] sugiere definiciones y medidas para cada una.

WebRef

Una excelente fuente de información acerca de la calidad del software y sobre temas relacionados (incluidas métricas) puede encontrarse en **www.qualityworld.com**

Exactitud. Un programa debe operar correctamente o proporcionará poco valor a sus usuarios. La exactitud es el grado en el cual el software realiza la función requerida. La medida más común de la exactitud son los defectos por KLOC, donde un defecto se define como una falta verificada de acuerdo con los requerimientos. Cuando se considera la calidad global de un producto de software, los defectos son aquellos problemas reportados por un usuario del programa después de que el programa se liberó para su uso general. Con propósitos de valoración de calidad, los defectos se cuentan sobre un periodo estándar, por lo general un año.

Capacidad de mantenimiento. El mantenimiento y soporte del software representan más esfuerzo que cualquiera otra actividad de ingeniería del software. La capacidad de mantenimiento es la facilidad con la que un programa puede corregirse si se encuentra un error, la facilidad con que se adapta si su entorno cambia o de mejorar si el cliente quiere un cambio en requerimientos. No hay forma de medir directamente la capacidad de mantenimiento; por tanto, deben usarse medidas indirectas. Una métrica simple orientada a tiempo es el *tiempo medio al cambio* (TMC), el tiempo que tarda en analizarse la petición de cambio, diseñar una modificación adecuada, implementar el cambio, probarlo y distribuirlo a todos los usuarios. En promedio, los programas con capacidad de mantenimiento tendrán un TMC más bajo (para tipos de cambios equivalentes) que los que no tienen dicha capacidad.

Integridad. La integridad del software se ha vuelto cada vez más importante en la era de los ciberterroristas y hackers. Este atributo mide la habilidad de un sistema para resistir ataques (tanto accidentales como intencionales) a su seguridad. Los ataques pueden hacerse en los tres componentes de software: programas, datos y documentación.

Para medir la integridad, deben definirse dos atributos adicionales: amenaza y seguridad. *Amenaza* es la probabilidad (que puede estimarse o derivarse de evidencia empírica) de que un ataque de un tipo específico ocurrirá dentro de un tiempo dado. *Seguridad* es la probabilidad (que puede estimarse o derivarse de evidencia empírica) de que el ataque de un tipo específico se repelerá. La integridad de un sistema puede definirse entonces como:

$$\text{Integridad} = \Sigma[1 - (\text{amenaza} \times (1 - \text{seguridad}))]$$

Por ejemplo, si la amenaza (la probabilidad de que un ataque ocurrirá) es de 0.25 y la seguridad (la probabilidad de repeler un ataque) es de 0.95, la integridad del sistema es 0.99 (muy alta). Si, por otra parte, la probabilidad de amenaza es 0.50 y la probabilidad de repeler un ataque es de solamente 0.25, la integridad del sistema es 0.63 (inaceptablemente baja).

Usabilidad. Si un programa no es fácil de usar, con frecuencia está condenado al fracaso, incluso si las funciones que realiza son valiosas. La usabilidad es un intento por cuantificar la facilidad de uso y puede medirse en términos de las características que se presentaron en el capítulo 11.

8 En el capítulo 23 se presentó un análisis detallado de los factores que influyen en la calidad y en las métricas del software que pueden usarse para valorar la calidad del software.

Los cuatro factores recién descritos sólo son una muestra de los que se han propuesto como medidas para la calidad del software. El capítulo 23 considera este tema con detalles adicionales.

25.3.2 Eficiencia en la remoción del defecto

Una métrica de calidad que proporciona beneficio tanto en el nivel del proyecto como en el del proceso es la *eficiencia de remoción del defecto* (ERD). En esencia, la ERD es una medida de la habilidad de filtrado de las acciones de aseguramiento y control de la calidad según se aplican a lo largo de todas las actividades del marco conceptual del proceso.

Cuando se considera para un proyecto como un todo, la ERD se define en la forma siguiente:

$$\text{ERD} = \frac{E}{E + D}$$

donde E es el número de errores que se encontraron antes de entregar el software al usuario final y D es el número de defectos que se encontraron después de la entrega.

Si la ERD es baja conforme se avanza a través del análisis y el diseño, emplee algo de tiempo para mejorar la forma en la que realiza las revisiones técnicas.

El valor ideal para la ERD es 1. Es decir, no se encuentran defectos en el software. En realidad, D será mayor que 0, pero el valor de la ERD todavía puede tender a 1 conforme E aumenta para un valor dado de D. De hecho, conforme E aumenta, es probable que el valor final de D disminuirá (los errores se filtran antes de convertirse en defectos). Si se usa como una métrica que proporciona un indicio de la capacidad de filtrado de las actividades de control y aseguramiento de la calidad, la ERD alienta a un equipo de software a instituir técnicas para encontrar tantos errores como sea posible antes de entregar.

La ERD también puede usarse dentro del proyecto a fin de valorar la habilidad de un equipo para encontrar errores antes de que pasen a la siguiente actividad de marco conceptual o acción de ingeniería del software. Por ejemplo, el análisis de requerimientos produce un modelo de requerimientos que puede revisarse para encontrar y corregir errores. Aquellos que no se encuentran durante la revisión del modelo de requerimientos pasan al diseño (donde pueden o no encontrarse). Cuando se usan en este contexto, la ERD se redefine como

$$\text{ERD}_i = \frac{E_i}{E_i + E_{i+1}}$$

donde E_i es el número de errores encontrados durante la acción i de ingeniería del software y E_{i+1} es el número de errores encontrados durante la acción $i + 1$ de ingeniería del software que son rastreables por errores que no se descubrieron en la acción i de ingeniería del software.

Un objetivo de calidad para un equipo de software (o un ingeniero de software individual) es lograr que la ERD_i tienda a 1. Es decir, los errores deben filtrarse antes de que pasen a la siguiente actividad o acción.

CASASEGURA

Establecimiento de un enfoque de métricas

La escena: Oficina de Doug Miller dos días después de la reunión inicial acerca de las métricas de software.

Participantes: Doug Miller (gerente del equipo de ingeniería del software *CasaSegura*), y Vinod Raman y Jamie Lazar, miembros del equipo de ingeniería del software de producto.

La conversación:

Doug: ¿Los dos tuvieron oportunidad de aprender un poco acerca de las métricas de proceso y proyecto?

Vinod y Jamie: [Ambos afirman con la cabeza.]

Doug: Siempre es buena idea establecer metas cuando se adopta alguna métrica. ¿Cuáles son las suyas?

Vinod: Nuestras métricas deben enfocarse en la calidad. De hecho, nuestra meta global es mantener en un mínimo absoluto el número de errores que pasamos de una actividad de ingeniería del software a la siguiente.

Doug: Y estar muy seguros de mantener el número de defectos liberados con el producto tan cerca de cero como sea posible.

Vinod (afirma con la cabeza): Desde luego.

Jamie: Me gusta la ERD como métrica, y creo que podemos usarla para todo el proyecto, pero también conforme nos movemos de una actividad de marco conceptual a la siguiente. Nos alentará a descubrir errores en cada paso.

Vinod: También quiero recopilar el número de horas que pasamos en las revisiones.

Jamie: Y el esfuerzo global que usamos en cada tarea de ingeniería del software.

Doug: Puedes calcular una razón entre revisión y desarrollo... puede ser interesante.

Jamie: También me gustaría rastrear algunos datos de caso de uso. Como la cantidad de esfuerzo requerido para desarrollar un caso de uso, la cantidad de esfuerzo requerido para construir software para implementar un caso de uso y...

Doug (sonríe): Creí que íbamos a mantener esto simple.

Vinod: Deberíamos, pero una vez que entras en este asunto de las métricas, hay muchas cosas interesantes que observar.

Doug: Estoy de acuerdo, pero caminemos antes de correr y apeguémonos a la meta. Limiten los datos a recopilar cinco o seis ítems, y estamos listos para arrancar.

25.4 Integración de métricas dentro del proceso de software

La mayoría de los desarrolladores de software todavía no miden y, tristemente, la mayor parte tiene poco deseo de comenzar. Como se apuntó anteriormente en este capítulo, el problema es cultural. Intentar recopilar medidas donde nadie las recopiló en el pasado con frecuencia precipita la resistencia. "¿Por qué necesitamos hacer esto?", pregunta un gerente de proyecto asediado. "No le veo el caso", se queja un profesional saturado de trabajo.

En esta sección se consideran algunos argumentos para fomentar el uso de las métricas del software y se presenta un enfoque para instituir un programa de recopilación de métricas dentro de una organización de ingeniería del software. Pero antes de comenzar, algunas palabras de sabiduría (ahora con más de dos décadas de antigüedad) sugeridas por Grady y Caswell [Gra87]:

> Algunas de las cosas que se describen aquí sonarán muy sencillas. Sin embargo, en realidad, establecer un programa de métricas de software exitoso que abarque a toda la compañía es trabajo duro. Cuando se dice que es necesario esperar al menos tres años antes de que tendencias organizacionales amplias estén disponibles, se tiene alguna idea del ámbito de tal esfuerzo.

Vale la pena hacer caso a la advertencia que sugieren los autores, pero los beneficios de la medición son tan atractivos que el trabajo duro bien lo vale.

25.4.1 Argumentos para métricas de software

Cita:

"Mediante números, administramos las cosas en muchos aspectos de nuestras vidas... Dichos números nos dan comprensión y nos ayudan a guiar nuestras acciones."

Michael Mah y Larry Putnam

¿Por qué es tan importante medir el proceso de ingeniería del software y del producto (software) que da como resultado? La respuesta es relativamente obvia. Si no se mide, no hay forma real de determinar si se está mejorando. Y si no se mejora, se está perdido.

Al solicitar y evaluar las medidas de productividad y calidad, un equipo de software (y su administración) puede establecer metas significativas para mejorar el proceso de software. Anteriormente, en este libro, se indicó que el software es un tema empresarial estratégico para muchas compañías. Si puede mejorarse el proceso a través del cual se desarrolla, puede tenerse como resultado un impacto directo sobre la línea de referencia. Pero para establecer metas a fin de mejorar, debe entenderse el estado actual del desarrollo del software. Por tanto, la medición se utiliza para establecer una línea de referencia del proceso desde el cual puedan valorarse las mejoras.

Los rigores diarios del trabajo del proyecto del software dejan poco tiempo para el pensamiento estratégico. Los gerentes de proyecto de software se preocupan por conflictos más

mundanos (pero igualmente importantes): desarrollar estimaciones de proyecto significativas, producir sistemas de alta calidad, sacar el producto a tiempo. Al usar la medición para establecer una línea de referencia del proyecto, cada uno de estos conflictos se vuelve más manejable. Ya se mencionó que la línea de referencia sirve como base para la estimación. Adicionalmente, la colección de métricas de calidad permite a una organización "afinar" su proceso de software para remover las causas de defectos "menos vitales" que tienen mayor impacto sobre el desarrollo del software.[9]

25.4.2 Establecimiento de una línea de referencia

? ¿Qué es una línea de referencia de métricas y qué beneficio proporciona a un ingeniero del software?

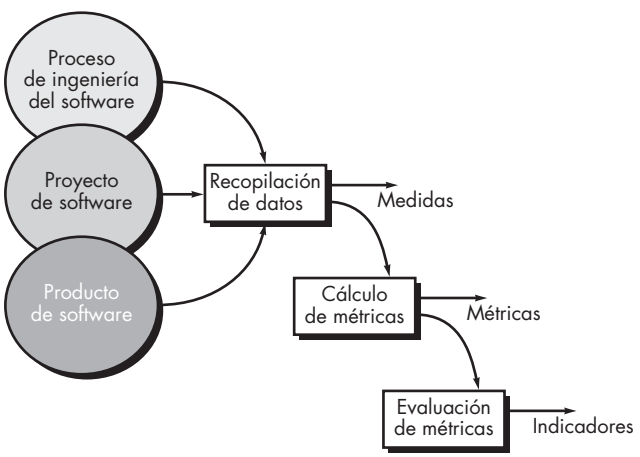
Al establecer una línea de referencia para métricas, pueden obtenerse beneficios en el proceso, el proyecto y el producto (técnico). Aunque la información que se recopila no necesita ser fundamentalmente diferente. Las mismas métricas pueden servir a muchos dominios. La línea de referencia de métricas consiste en los datos recopilados a partir de los proyectos de desarrollo de software y puede ser tan simple como se presenta en la tabla de la figura 25.2 o tan compleja como una base de datos exhaustiva que contiene decenas de medidas de proyecto y las métricas derivadas de ellas.

Para ser un auxiliar efectivo en el mejoramiento del proceso y/o estimación de costo y esfuerzo, los datos de la línea de referencia deben tener los siguientes atributos: 1) deben ser razonablemente precisos y deben evitarse "suposiciones" acerca de los proyectos anteriores, 2) deben recopilarse para tantos proyectos como sea posible, 3) deben ser consistentes (por ejemplo, una línea de código debe interpretarse consistentemente a través de todos los proyectos para los cuales se recopilan datos), 4) las aplicaciones deben ser similares al trabajo que debe estimarse: tiene poco sentido usar una línea de referencia para el trabajo de sistemas de información en lote a fin de estimar una aplicación incrustada en tiempo real.

25.4.3 Recolección, cálculo y evaluación de métricas

En la figura 25.3 se ilustra el proceso que se sigue para establecer una línea de referencia de métricas. De manera ideal, los datos necesarios para establecer una línea de referencia se recolectaron de una forma continua. Tristemente, éste rara vez es el caso. Por tanto, la recopilación de datos requiere una investigación histórica de los proyectos anteriores a fin de reconstruir los datos requeridos. Una vez recopiladas las medidas (sin duda, el paso más difícil), es posible el

FIGURA 25.3
Proceso de recopilación de métricas del software



9 Estas ideas se formalizaron en un enfoque llamado *aseguramiento estadístico de la calidad del software*.

Los datos de métricas de línea de referencia deben recopilarse a partir de un gran muestreo representativo de proyectos de software anteriores.

cálculo de métricas. Dependiendo de la envergadura de las medidas recopiladas, las métricas pueden abarcar un amplio rango de métricas orientadas a aplicaciones (por ejemplo, LOC, PF, orientada a objeto, *webapp*), así como otras orientadas a calidad y proyecto. Finalmente, las métricas deben evaluarse y aplicarse durante la estimación, el trabajo técnico, el control del proyecto y la mejora del proceso. La evaluación de métricas se enfoca en las razones subyacentes para los resultados obtenidos y produce un conjunto de indicadores que guían al proyecto o al proceso.

25.5 Métricas para organizaciones pequeñas

Si apenas comienza a recopilar datos de métricas, recuerde mantenerlas simples. Si se entierra debajo de datos, sus esfuerzos de métricas fallarán.

La gran mayoría de las organizaciones de desarrollo de software tienen menos de 20 personas en sus departamentos de software. No es razonable, y en la mayoría de los casos es irreal, esperar que tales organizaciones desarrollen programas de métricas de software exhaustivos. Sin embargo, sí es razonable sugerir que las organizaciones de software de todos los tamaños midan y luego usen las métricas resultantes para ayudar a mejorar su proceso de software local y la calidad y temporalidad de sus productos.

Un enfoque de sentido común para la implementación de cualquier actividad de software relacionada con el software es: mantenerlo simple, personalizar para satisfacer las necesidades locales y asegurarse de que agrega valor. En los siguientes párrafos se examina la manera como se relacionan dichos lineamientos con las métricas para tiendas pequeñas.[10]

¿Cómo debe derivarse un conjunto de métricas de software "simples"?

"Mantenerlo simple" es un lineamiento que funciona razonablemente bien en muchas actividades. Pero, ¿cómo debe derivarse un conjunto de métricas de software "simples" que aun así proporcionen valor y cómo puede asegurarse de que dichas métricas simples satisfarán las necesidades específicas de su organización de software? Puede comenzarse con un enfoque en los resultados, no en la medición. Se consulta al grupo de software para definir un solo objetivo que requiera mejora. Por ejemplo, "reducir el tiempo para evaluar e implementar las peticiones de cambio". Una organización pequeña puede seleccionar el siguiente conjunto de medidas de fácil recolección:

- Tiempo transcurrido (horas o días) desde el momento en el que se hace una petición hasta que la evaluación está completa, t_{cola}.
- Esfuerzo (persona-horas) para realizar la evaluación, W_{eval}.
- Tiempo transcurrido (horas o días) desde la conclusión de la evaluación hasta la asignación de la orden de cambio al personal, t_{eval}.
- Esfuerzo requerido (persona-horas) para hacer el cambio, W_{cambio}.
- Tiempo requerido (horas o días) para hacer el cambio, t_{cambio}.
- Errores descubiertos durante el trabajo para hacer el cambio, E_{cambio}.
- Defectos descubiertos después de liberar el cambio al cliente base, D_{cambio}.

Una vez recopiladas dichas medidas para algunas peticiones de cambio, es posible calcular el tiempo transcurrido total desde la petición de cambio hasta la implementación del mismo y el porcentaje del tiempo transcurrido absorbido por la cola inicial, la evaluación y la asignación de cambio y su implementación. De igual modo, puede determinarse el porcentaje de esfuerzo requerido para evaluación e implementación. Dichas métricas pueden valorarse en el contexto de datos de calidad, E_{cambio} y D_{cambio}. Los porcentajes brindan comprensión acerca de dónde se frena el proceso de petición de cambio y pueden conducir a definir medidas de mejora del pro-

10 Este análisis es igualmente relevante para los equipos de software que adoptan un proceso de desarrollo de software ágil (capítulo 3).

ceso para reducir t_{cola}, W_{eval}, t_{eval}, W_{cambio} y/o E_{cambio}. Además, la eficiencia de remoción de defecto puede calcularse como

$$\text{ERD} = \frac{E_{cambio}}{E_{cambio} + D_{cambio}}$$

La ERD puede compararse con el tiempo transcurrido y con el esfuerzo total para determinar el impacto de las actividades de aseguramiento de la calidad sobre el tiempo y el esfuerzo requerido para hacer un cambio.

Para grupos pequeños, el costo que representa recopilar medidas y calcular métricas varía de 3 a 8 por ciento del presupuesto del proyecto durante la fase de aprendizaje, y luego disminuye a menos de 1 por ciento del presupuesto del proyecto después de que los ingenieros del software y los gerentes del proyecto se familiarizan con el programa de métricas [Gra99]. Dichos costos pueden mostrar un rendimiento sustancial sobre la inversión si la comprensión de los datos de las métricas conducen a una mejora de proceso significativa para la organización del software.

25.6 Establecimiento de un programa de métricas del software

El Software Engineering Institute (SEI) desarrolló un manual muy completo [Par96b] para establecer un programa de métrica de software "dirigido hacia la meta". El manual sugiere los siguientes pasos:

1. Identificar las metas empresariales.
2. Identificar lo que se quiere conocer o aprender.
3. Identificar las submetas.
4. Identificar las entidades y atributos relacionados con las submetas.
5. Formalizar las metas de medición.
6. Identificar preguntas cuantificables y los indicadores relacionados que se usarán para ayudar a lograr las metas de medición.
7. Identificar los elementos de datos que se recopilarán para construir los indicadores que ayuden a responder las preguntas.
8. Definir las medidas que se van a usar y hacer operativas estas definiciones.
9. Identificar las acciones que se tomarán para implementar las medidas.
10. Preparar un plan para la implantación de las medidas.

WebRef

Puede descargarse de **www.sei.cmu.edu** un *manual para medición de software dirigido hacia las metas.*

Un estudio detallado de estos pasos se aprecia mejor en el manual del SEI. Sin embargo, vale la pena un breve panorama de los puntos clave.

Puesto que el software apoya las funciones empresariales, diferencia los sistemas o productos basados en computadora y actúa como un producto en sí mismo, las metas definidas por la empresa casi siempre pueden rastrearse en las metas específicas de ingeniería del software. Por ejemplo, considere el producto *CasaSegura*. Al trabajar como equipo, los gerentes de ingeniería del software y de la empresa desarrollan una lista de metas empresariales prioritarias:

Las métricas del software que elija deben activarse mediante las metas empresariales y técnicas que quiera lograr.

1. Mejorar la satisfacción de los clientes con los productos.
2. Hacer los productos más fáciles de usar.
3. Reducir el tiempo que tarda en salir un nuevo producto al mercado.
4. Facilitar el apoyo a los productos.
5. Mejorar la rentabilidad global.

La organización de software examina cada meta empresarial y pregunta: ¿qué actividades manejamos, ejecutamos o apoyamos y qué hacemos para mejorar dichas actividades? Para responder estas preguntas, el SEI recomienda la creación de una "lista de entidad-pregunta" en la que se anotan todas las cosas (entidades) existentes dentro del proceso de software que se gestionan o que influyen en la organización de software. Los ejemplos de entidades incluyen recursos de desarrollo, productos operativos, código fuente, casos de prueba, peticiones de cambio, tareas de ingeniería del software y calendarios. Para cada entidad mencionada, el personal de software desarrolla un conjunto de preguntas que valoran las características cuantitativas de la entidad (por ejemplo, tamaño, costo, tiempo para desarrollar). Las preguntas derivadas como consecuencia de la creación de una lista de entidad-pregunta conducen a la derivación de un conjunto de submetas que se relacionan directamente con las entidades creadas y con las actividades realizadas como parte del proceso de software.

Considere la cuarta meta: "Facilitar el apoyo a los productos". La siguiente lista de preguntas puede derivarse de esta meta [Par96b]:

- ¿Las peticiones de cambio de los clientes contienen la información que se requiere para evaluar adecuadamente el cambio y luego implementarlo en una forma oportuna?
- ¿Cuán grande es la acumulación de trabajo debida a la petición de cambio?
- ¿El tiempo de respuesta para corregir errores es aceptable con base en las necesidades del cliente?
- ¿Se sigue el proceso de control de cambio (capítulo 22)?
- ¿Los cambios de alta prioridad se implementan en forma oportuna?

A partir de estas preguntas, la organización de software puede derivar la siguiente submeta: *mejorar el rendimiento del proceso de gestión del cambio*. Entonces, se identifican las entidades y atributos del proceso de software que son relevantes para la submeta y se delinean las metas de medición asociadas con ellas.

El SEI [Par96b] proporciona lineamientos detallados para los pasos del 6 al 10 de su enfoque de medición dirigido a metas. En esencia, las metas de medición se refinan en preguntas que se desglosan aún más en entidades y atributos que entonces se refinan en métricas.

INFORMACIÓN

Establecimiento de un programa de métricas

El Software Productivity Center **(www.spc.ca)** sugiere un enfoque de ocho pasos para establecer un programa de métricas dentro de una organización de software, que puedan usarse como alternativa al enfoque SEI descrito en la sección 25.6. Su enfoque se resume en este recuadro.

1. Comprender el proceso de software existente.
   Identificar actividades de marco conceptual (capítulo 2).
   Describir la información de entrada para cada actividad.
   Definir las tareas asociadas con cada actividad.
   Anotar las funciones de aseguramiento de calidad.
   Mencionar los productos operativos que se producen.
2. Definir las metas por lograr mediante el establecimiento de un programa de métricas.
   Ejemplos: mejorar la precisión de la estimación, mejorar la calidad del producto.
3. Identificar las métricas requeridas para lograr las metas.
   Definir preguntas, por ejemplo, *¿cuántos errores encontrados en una actividad de marco conceptual pueden rastrearse en la actividad de marco conceptual precedente?*
   Crear medidas y métricas que ayudarán a responder dichas preguntas.
4. Identificar las medidas y métricas que se van a recopilar y calcular.
5. Establecer un proceso de recolección de medición al responder estas preguntas:
   ¿Cuál es la fuente de las mediciones?
   ¿Pueden usarse herramientas para recopilar los datos?
   ¿Quién es responsable de recopilar los datos?
   ¿Cuándo se recopilan y registran datos?
   ¿Cómo se almacenan los datos?

¿Qué mecanismos de validación se usan para garantizar que los datos son correctos?

6. Adquirir herramientas adecuadas para auxiliar en la recopilación y valoración.
7. Establecer una base de datos de métricas.
   Establecer la sofisticación relativa de la base de datos.
   Explorar el uso de herramientas relacionadas (por ejemplo, un repositorio SCM, capítulo 26).
   Evaluar los productos de base de datos existentes.
8. Definir mecanismos de retroalimentación adecuados.
   ¿Quién requiere información de métricas actuales?
   ¿Cómo se entregará la información?
   ¿Cuál es el formato de la información?

Una descripción considerablemente más detallada de estos ocho pasos puede descargarse de **www.spc.ca/resources/metrics/**

25.7 Resumen

La medición permite a gerentes y profesionales mejorar el proceso de software; auxiliar en la planificación, rastreo y control de proyectos de software y valorar la calidad del producto (software) que se elabora. Las medidas de atributos específicos del proceso, proyecto y producto se usan para calcular las métricas de software. Dichas métricas pueden analizarse para proporcionar indicadores que guíen las acciones administrativas y técnicas.

Las métricas de proceso permiten que una organización adopte una visión estratégica al proporcionar comprensión acerca de la efectividad de un proceso de software. Las métricas de proyecto son tácticas. Permiten que un gerente de proyecto adapte el flujo de trabajo del proyecto y el enfoque técnico en tiempo real.

Las métricas orientadas a tamaño y a función se usan a lo largo de la industria. Las primeras usan la línea de código como un factor de normalización para otras medidas, como persona-meses o defectos. El punto de función se deriva de las medidas del dominio de información y de una valoración subjetiva de la complejidad del problema. Además, pueden usarse métricas orientadas a objeto y a *webapp*.

Las métricas de calidad del software, como las métricas de productividad, se enfocan en el proceso, el proyecto y el producto. Al desarrollar y analizar una línea de referencia de métricas para la calidad, una organización puede corregir aquellas áreas del proceso de software que sean la causa de los defectos de software.

La medición da como resultado un cambio cultural. La recopilación de datos, cálculo de métricas y análisis de métricas son los tres pasos que deben implementarse para comenzar un programa de métricas. En general, un enfoque dirigido a metas ayuda a una organización a enfocarse en las métricas correctas para su empresa. Al crear una línea de referencia de métricas, una base de datos que contenga mediciones de proceso y producto, los ingenieros de software y sus gerentes pueden obtener mejor comprensión del trabajo que hacen y del producto que elaboran.

Problemas y puntos por evaluar

25.1. Describa con sus palabras la diferencia entre métricas de proceso y de proyecto.

25.2. ¿Por qué algunas métricas de software deben mantenerse "privadas"? Ofrezca cinco ejemplos de tres métricas que deban ser privadas. Brinde ejemplos de tres métricas que deban ser públicas.

25.3. ¿Qué es una media indirecta y por qué tales mediciones son comunes en el trabajo con métricas de software?

25.4. Grady sugiere una etiqueta para las métricas de software. ¿Puede agregar tres reglas más a las anotadas en la sección 25.1.1?

25.5. El equipo A encontró 342 errores durante el proceso de ingeniería del software antes de la liberación. El equipo B encontró 184 errores. ¿Qué medidas adicionales tendrían que realizarse a los proyectos A y B para determinar cuál de los equipos eliminó errores de manera más eficiente? ¿Qué métricas propondría para ayudar a realizar esta determinación? ¿Qué datos históricos pueden ser útiles?

25.6. Presente un argumento contra las líneas de código como medida para la productividad del software. ¿Su caso se sostendría cuando se consideren decenas o cientos de proyectos?

25.7. Calcule el valor de punto de función para un proyecto con las siguientes características de dominio de información:

Número de entradas de usuario: 32
Número de salidas de usuario: 60
Número de consultas de usuario: 24
Número de archivos: 8
Número de interfaces externas: 2

Suponga que todos los valores de ajuste de complejidad son promedios. Use el algoritmo mencionado en el capítulo 23.

25.8. Con la tabla que se presenta en la sección 25.2.3, plantee un argumento contra el uso de lenguaje ensamblador con base en la funcionalidad que entrega por enunciado de código. Nuevamente con la tabla, analice por qué C++ representaría una mejor alternativa que C.

25.9. El software que se usa para controlar una fotocopiadora requiere 32 000 líneas de C y 4 200 líneas de Smalltalk. Estime el número de puntos de función para el software dentro de la fotocopiadora.

25.10. Un equipo de ingeniería web construye una *webapp* de comercio electrónico que contiene 145 páginas individuales. De éstas, 65 son dinámicas, es decir, se generan internamente con base en entrada del usuario final. ¿Cuál es el índice de personalización para esta aplicación?

25.11. Una *webapp* y su entorno de apoyo no están completamente fortificados contra ataques. Los ingenieros web estiman que la probabilidad de repeler un ataque es de sólo 30 por ciento. El sistema no contiene información sensible o controvertida, de modo que la probabilidad de amenaza es de 25 por ciento. ¿Cuál es la integridad de la *webapp*?

25.12. En la conclusión de un proyecto, se determinó que se encontraron 30 errores durante la actividad de modelado y 12 durante la actividad de construcción, que fueron rastreables en errores que no se descubrieron en la actividad de modelado. ¿Cuál es la ERD para la actividad de modelado?

25.13. Un equipo de software entrega un incremento de software a los usuarios finales. Éstos descubren ocho defectos durante el primer mes de uso. Antes de la liberación, el equipo de software encontró 242 errores durante las revisiones técnicas formales y todas las tareas de prueba. ¿Cuál es la ERD global para el proyecto después de un mes de uso?

Lecturas y fuentes de información adicionales

El mejoramiento del proceso de software (MPS) recibió una cantidad significativa de atención durante las dos décadas pasadas. Dado que la medición y las métricas de software son clave para el mejoramiento exitoso del proceso de software, muchos libros acerca de MPS también estudian métricas. Rico (*ROI of Software Process Improvement*, J. Ross Publishing, 2004) ofrece un análisis a profundidad del MPS y de las métricas que pueden ayudar a una organización a lograrlo. Ebert *et al.* (*Best Practices in Software Measurement*, Springer, 2004) abordan el uso de medición dentro del contexto de estándares ISO y CMMI. Kan (*Metrics and Models in Software Quality Engineering*, 2a. ed., Addison-Wesley, 2002) presenta una colección de métricas relevantes.

Ebert y Dumke (*Software Measurement*, Springer, 2007) proporcionan un tratamiento útil de medición y métrica para proyectos IT. McGarry *et al.* (*Practical Software Measurement*, Addison-Wesley, 2001) presentan consejos profundos para valorar el proceso de software. Una valiosa colección de ensayos fueron editados por Haug y colegas (*Software Process Improvement: Metrics, Measurement, and Process Modeling,* Springer-Verlag, 2001). Florac y Carlton (*Measuring the Software Process,* Addison-Wesley, 1999) y Fenton y Pfleeger (*Software Metrics: A Rigorous and Practical Approach,* Revised, Brooks/Cole Publishers, 1998) estudian cómo pueden usarse las métricas del software para proporcionar los indicadores necesarios a fin de mejorar el proceso de software.

Laird y Brennan (*Software Measurement and Estimation*, Wiley-IEEE Computer Society Press, 2006) y Goodman (*Software Metrics: Best practices for Successful IT Management*, Rothstein Associates, Inc., 2004), analizan el uso de métricas de software para administración y estimación de proyectos. Putnam y Myers (*Five Core Metrics*, Dorset House, 2003) recurren a una base de datos de más de 6 000 proyectos de software para demostrar cómo pueden usarse cinco métricas centrales (tiempo, esfuerzo, tamaño, confiabilidad y productividad de proceso) para controlar los proyectos de software. Maxwell (*Applied Statistics for Software Managers*, Prentice-Hall, 2003), presenta técnicas para analizar datos de proyecto de software. Munson (*Software Engineering Measurement*, Auerbach, 2003) estudia una amplia matriz de conflictos de medición en ingeniería del software. Jones (*Software Assessments, Benchmarks and Best Practices*, Addison-Wesley, 2000) describe tanto la medición cuantitativa como los factores cualitativos que ayudan a una organización a valorar sus procesos y prácticas de software.

La medición del punto de función se ha convertido en una técnica ampliamente usada en muchas áreas del trabajo en ingeniería del software. Parthasarathy (*Practical Software Estimation: Function Point Methods for Insourced and Outsourced Projects*, Addison-Wesley, 2007) ofrece una guía exhaustiva. Garmus y Herron (*Function Point Analysis: Measurement Practices for Successful Software Projects*, Addison-Wesley, 2000) analizan las métricas de proceso con énfasis en el análisis de punto de función.

Relativamente poco se ha publicado acerca de las métricas para trabajo en ingeniería web. Sin embargo, Kaushik *(Web Analytics: An Hour a Day,* Sybex, 2007), Stern (*Web Metrics: Proven Methods for Measuring Web Site Success,* Wiley, 2002), Inan y Kean (*Measuring the Success of Your Website,* Longman, 2002), y Nobles y Grady (*Web Site Analysis and Reporting,* Premier Press, 2001) abordan las métricas web desde una perspectiva empresarial y de mercadotecnia.

El IEEE (*Symposium on Software Metrics*, publicación anual) resume las investigaciones más recientes en el área de métricas. En internet, está disponible una gran variedad de fuentes de información acerca de las métricas de procesos y de proyectos. Una lista actualizada de referencias en la World Wide Web que son relevantes para las métricas de procesos y de proyectos puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

26

ESTIMACIÓN PARA PROYECTOS DE SOFTWARE

CONCEPTOS CLAVE

ámbito del software 595
ecuación del software 610
estimación 594
  ágil 612
  basada en PF 602
  basada en problema 600
  basada en proceso. 604
  casos de uso. 605
  modelos empíricos. 608
  proyectos orientados a objetos 611
  reconciliación 607
  webapps 613
factibilidad 595
planificación del proyecto. . . 595

La administración de los proyectos de software comienza con un conjunto de actividades que de manera colectiva se llaman *planificación de proyecto*. Antes de que el proyecto pueda comenzar, el equipo de software debe estimar el trabajo que se va a realizar, los recursos que se requerirán y el tiempo que transcurrirá de principio a fin. Una vez completadas dichas actividades, el equipo de software debe establecer un calendario del proyecto que defina las tareas e hitos de la ingeniería de software, que identifique quién es responsable de realizar cada tarea y especifique las dependencias entre tareas que puedan imponer una fuerte demora sobre el avance.

En una excelente guía para "sobrevivir al proyecto de software", Steve McConnell [McC98] presenta una visión del mundo real de la planificación del proyecto:

> Muchos trabajadores técnicos preferirían realizar trabajo técnico en lugar de pasar tiempo planificando. Muchos gerentes técnicos no tienen suficiente capacitación en administración técnica como para sentirse seguros de que su planificación mejorará el resultado de un proyecto. Dado que ninguna parte quiere hacer planes, con frecuencia nunca se hacen.
>
> Pero fallar en la planificación es uno de los errores más cruciales que un proyecto puede tener... la planificación efectiva es necesaria para resolver problemas corriente arriba [tempranamente en el proyecto] a bajo costo, en lugar de corriente abajo [tardíamente en el proyecto] a alto costo. El proyecto promedio emplea *80 por ciento* de su tiempo en "poner al día": corregir errores que se cometieron anteriormente en el proyecto.

UNA MIRADA RÁPIDA

¿Qué es? Se establece una necesidad real para el software; los participantes están a bordo, los ingenieros de software están listos para comenzar y el proyecto está a punto de iniciar. Pero, ¿cómo se procede? La planificación de proyectos de software abarca cinco grandes actividades: estimación, calendarización, análisis de riesgos, planificación de gestión de la calidad y planificación de gestión del cambio. En el contexto de este capítulo, sólo se considera la estimación, el intento por determinar cuánto dinero, esfuerzo, recursos y tiempo tomará construir un sistema o producto específico basado en software.

¿Quién lo hace? Los gerentes de proyecto de software, con la información solicitada a los participantes del proyecto y datos de métricas de software recopiladas de proyectos anteriores.

¿Por qué es importante? ¿Construiría una casa sin saber más o menos cuánto gastará, las tareas que necesita realizar y el cronograma para el trabajo que se va a realizar? Desde luego que no y, dado que la mayoría de los sistemas y productos basados en computadora cuestan considerablemente más que construir una gran casa, parece razonable realizar una estimación antes de comenzar a crear el software.

¿Cuáles son los pasos? La estimación comienza con una descripción del ámbito del problema. Luego éste se descompone en un conjunto de problemas más pequeños y cada uno de éstos se estima, usando como guías datos históricos y experiencia. La complejidad y el riesgo del problema se consideran antes de realizar una estimación final.

¿Cuál es el producto final? La generación de una tabla simple que delinea las tareas que se van a realizar, las funciones por implementar y el costo, esfuerzo y tiempo involucrados para cada tarea.

¿Cómo me aseguro de que lo hice bien? Eso es difícil, porque en realidad no se sabe hasta que el proyecto se completa. Sin embargo, si se tiene experiencia y se sigue un enfoque sistemático, si genera estimaciones usando datos históricos sólidos, si crea puntos de datos de estimación con al menos dos métodos diferentes, si establece un calendario realista y continuamente lo adapta conforme el proyecto avanza, puede estar seguro de que está haciendo su mejor esfuerzo.

McConnell argumenta que todo equipo puede encontrar el tiempo para planificar (y adaptar el plan a lo largo del proyecto), tomando simplemente un pequeño porcentaje del tiempo que se hubiera empleado en rehacer lo que ocurre porque no se llevó a cabo la planificación.

26.1 Observaciones acerca de las estimaciones

La planificación requiere adoptar un compromiso inicial, aun cuando es probable que este "compromiso" resulte erróneo. Siempre que se hacen estimaciones se mira hacia el futuro y se acepta cierto grado de incertidumbre habitual. En palabras de Frederick Brooks [Bro95]:

> ... nuestras técnicas de estimación están pobremente desarrolladas. Más seriamente, reflejan una suposición no expresada que es muy falsa: que todo irá bien [...] puesto que se tiene incertidumbre acerca de las estimaciones, los gerentes de software con frecuencia carecen de la tenacidad cortés para hacer lo que la gente espera de un buen producto.

Cita:

"Los buenos enfoques de estimación y los datos históricos sólidos ofrecen la mejor esperanza de que realmente se triunfará sobre demandas imposibles."

Caper Jones

Aunque las estimaciones son tanto un arte como una ciencia, esta importante acción no necesita realizarse en forma azarosa. Existen técnicas útiles para estimación de tiempo y esfuerzo. Las métricas de proceso y proyecto pueden proporcionar perspectiva histórica y poderosa entrada para la generación de estimaciones cuantitativas. Las experiencias pasadas (de todas las personas involucradas) pueden auxiliar sin medida conforme se desarrollen y revisen las estimaciones. Puesto que éstas tienden los cimientos de todas las acciones de planificación del proyecto, y la planificación del proyecto ofrece el mapa de caminos para la ingeniería de software exitosa, estaríamos mal aconsejados si nos embarcáramos sin ella.

La estimación de recursos, costo y calendario para un esfuerzo de ingeniería de software requiere experiencia, acceso a buena información histórica (métricas) y coraje para comprometerse con las predicciones cuantitativas cuando todo lo que existe es información cualitativa. La estimación porta un riesgo inherente,[1] y éste conduce a incertidumbre.

La *complejidad del proyecto* tiene un fuerte efecto sobre la incertidumbre inherente a la planificación. Sin embargo, la complejidad es una medida relativa que es afectada por la familiaridad con el esfuerzo pasado. El profesional que por primera vez desarrolla una sofisticada aplicación de comercio electrónico puede considerarla excesivamente compleja. Sin embargo, un equipo de ingeniería web que desarrolla su décima *webapp* de comercio electrónico considerará tal trabajo común y corriente. Se han propuesto [Zus97] algunas medidas cuantitativas de complejidad de software. Éstas se aplican en el nivel de diseño o código y, por tanto, son difíciles de usar durante la planificación del software (antes de una salida de diseño y código). No obstante, es posible establecer otras valoraciones de complejidad más subjetivas (por ejemplo, los factores de ajuste de complejidad de punto de función descritos en el capítulo 23) en las primeras etapas del proceso de planificación.

PUNTO CLAVE

Complejidad del proyecto, tamaño del proyecto y grado de incertidumbre estructural afectan la confiabilidad de las estimaciones.

El *tamaño del proyecto* es otro factor importante que puede afectar la precisión y la eficacia de las estimaciones. Conforme aumenta el tamaño, la interdependencia entre varios elementos del software crece rápidamente.[2] La descomposición del problema, un importante enfoque de la estimación, se vuelve más difícil porque el refinamiento de los elementos del problema todavía puede ser formidable. Para parafrasear la ley de Murphy: "lo que puede salir mal, saldrá mal", y si hay más cosas que pueden fallar, más cosas fallarán.

El *grado de incertidumbre estructural* también tiene un efecto sobre el riesgo de estimación. En este contexto, estructura se refiere al grado en el cual se solidificaron los requisitos, la faci-

1 En el capítulo 28 se presentan técnicas sistemáticas para el análisis de riesgos.

2 Con frecuencia, el tamaño aumenta debido al "lento avance del ámbito", que ocurre cuando cambian los requisitos del problema. El aumento en el tamaño del proyecto puede tener un impacto geométrico sobre el costo y el calendario del proyecto (Michael Mah, comunicación personal).

lidad con la que se dividieron las funciones y la naturaleza jerárquica de la información que debe procesarse.

Cita:

"Es distintivo de una mente instruida descansar satisfecha con el grado de precisión que la naturaleza del sujeto admite, y no buscar exactitud cuando sólo es posible una aproximación a la verdad."

Aristóteles

La disponibilidad de información histórica tiene fuerte influencia sobre el riesgo de estimación. Al mirar hacia atrás, puede emular las cosas que funcionaron y mejorar las áreas donde surgieron problemas. Cuando hay disponibles métricas de software exhaustivas (capítulo 25) para proyectos anteriores, pueden hacerse estimaciones con mayor precisión, así como establecerse calendarios para evitar las dificultades pasadas y el riesgo global se reduce.

El riesgo de estimación se mide por el grado de incertidumbre en las estimaciones cuantitativas establecidas para recursos, costo y calendario. Si el ámbito del proyecto se comprende pobremente o si los requisitos del proyecto están sujetos a cambio, la incertidumbre y el riesgo en la estimación se vuelven peligrosamente altos. Como planificador, usted y el cliente deben reconocer que la variabilidad en los requisitos del software significa inestabilidad en costo y calendario.

Sin embargo, no debe volverse obsesivo acerca de la estimación. Los modernos enfoques de ingeniería de software (por ejemplo, modelos de proceso evolutivos) toman una visión iterativa del desarrollo. En tales enfoques es posible, aunque no siempre políticamente aceptable, revisitar la estimación (conforme se conoce más información) y revisarla cuando el cliente hace cambios a los requisitos.

26.2 El proceso de planificación del proyecto

CONSEJO

Mientas más conozca, mejor estimará. En consecuencia, actualice sus estimaciones conforme avance el proyecto.

El objetivo de la planificación del proyecto de software es proporcionar un marco conceptual que permita al gerente hacer estimaciones razonables de recursos, costo y calendario. Además, las estimaciones deben intentar definir los escenarios de mejor caso y peor caso, de modo que los resultados del proyecto puedan acotarse. Aunque hay un grado inherente de incertidumbre, el equipo de software se embarca en un plan que se haya establecido como consecuencia de dichas tareas. Por tanto, el plan debe adaptarse y actualizarse conforme avanza el proyecto. En las siguientes secciones se estudia cada una de las acciones asociadas con la planificación del proyecto de software.

Conjunto de tareas

Conjunto de tareas para planificación de proyectos

1. Establecer ámbito del proyecto.
2. Determinar la factibilidad.
3. Analizar los riesgos (capítulo 28).
4. Definir recursos requeridos.
   a) Determinar recursos humanos requeridos.
   b) Definir recursos de software reutilizables.
   c) Identificar recursos ambientales.
5. Estimar costo y esfuerzo.
   a) Descomponer el problema.
   b) Desarrollar dos o más estimaciones usando tamaño, puntos de función, tareas de proceso o casos de uso.
   c) Reconciliar las estimaciones.
6. Desarrollar un calendario del proyecto (capítulo 27).
   a) Establecer un conjunto de tareas significativas.
   b) Definir una red de tareas.
   c) Usar herramientas de calendarización para desarrollar un cronograma.
   d) Definir mecanismos de seguimiento de calendario.

26.3 Ámbito y factibilidad del software

El *ámbito del software* describe las funciones y características que se entregan a los usuarios finales; los datos que son entrada y salida; el "contenido" que se presenta a los usuarios como consecuencia de usar el software y el desempeño, las restricciones, las interfaces y la confiabilidad que se *ligan* al sistema. El ámbito se define usando una de dos técnicas:

1. Una descripción narrativa del ámbito del software se desarrolla después de la comunicación con todos los participantes.
2. Los usuarios finales desarrollan un conjunto de casos de uso.[3]

La factibilidad del proyecto es importante, pero una consideración de la necesidad empresarial lo es incluso más. No es bueno construir un sistema o producto de alta tecnología que nadie quiere.

Las funciones descritas en el enunciado del ámbito (o dentro de los casos de uso) se evalúan y en algunos casos se desglosan para proporcionar más detalle, previamente al comienzo de la estimación. Puesto que las estimaciones de costo y calendario están funcionalmente orientadas, con frecuencia es útil cierto grado de descomposición. Las consideraciones de rendimiento abarcan los requisitos de procesamiento y de tiempo de respuesta. Las restricciones identifican los límites colocados en el software por parte de hardware externo, memoria disponible u otros sistemas existentes.

Una vez identificado el ámbito (con la concurrencia del cliente), es razonable preguntar: ¿puede construirse software para satisfacer este ámbito? ¿El proyecto es factible? Con mucha frecuencia, los ingenieros de software rápidamente desechan estas preguntas (o son presionados para desecharlas por gerentes impacientes u otros participantes), sólo para encontrarse enlodados en un proyecto que está condenado desde el principio. Putnam y Myers [Put97a] abordan este conflicto cuando escriben:

> [No] todo lo imaginable es factible, ni siquiera en software, tan evanescente como pueda aparecer a los profanos. Por el contrario, la factibilidad del software tiene cuatro dimensiones sólidas: *Tecnología*: ¿Un proyecto es técnicamente factible? ¿Está dentro del estado del arte? ¿Pueden reducirse los defectos en un nivel que coincida con las necesidades de la aplicación? *Finanzas*: ¿Es financieramente factible? ¿El desarrollo puede completarse a un costo que la organización de software, su cliente o el mercado puede pagar? *Tiempo*: ¿El tiempo del proyecto para llegar al mercado vencerá a la competencia? *Recursos*: ¿La organización tiene los recursos necesarios para triunfar?

Putnam y Myers sugieren de manera correcta que establecer el ámbito no es suficiente. Una vez comprendido éste, debe trabajarse para determinar si puede hacerse dentro de las dimensiones recién anotadas. Ésta es una parte vital, aunque con frecuencia pasada por alto, del proceso de estimación.

26.4 Recursos

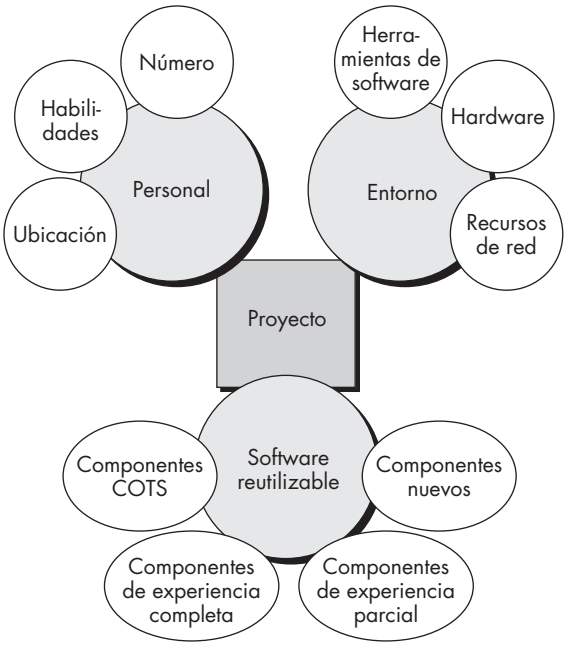
La segunda tarea en la planificación es la estimación de los recursos requeridos para lograr el esfuerzo de desarrollo del software. La figura 26.1 muestra las tres principales categorías de los recursos de la ingeniería de software: personal, componentes de software reutilizables y entorno de desarrollo (herramientas de hardware y software). Cada recurso se especifica con cuatro características: descripción del recurso, un enunciado de disponibilidad, momento en el que se requerirá el recurso y duración del tiempo que se aplicará el recurso. Las últimas dos características pueden verse como una *ventana temporal*. La disponibilidad del recurso para una ventana específica debe establecerse en el tiempo práctico más temprano.

26.4.1 Recursos humanos

El planificador comienza por evaluar el ámbito del software y seleccionando las habilidades requeridas para completar el desarrollo. Se especifican tanto la posición organizacional (por ejemplo, gerente, ingeniero de software ejecutivo) como la especialidad (por ejemplo, telecomunicaciones, base de datos, cliente-servidor). Para proyectos relativamente pequeños (algunos persona-meses), un solo individuo puede realizar todas las tareas de ingeniería de software y

3 Los casos de uso se estudiaron con detalle a lo largo de la parte 2 de este libro. Un caso de uso es una descripción basada en el escenario de la interacción del usuario con el software desde el punto de vista del usuario.

FIGURA 26.1
Recursos del proyecto



consultar especialistas según lo requiera. Para proyectos más grandes, el equipo de software puede dispersarse geográficamente a través de algunas ubicaciones diferentes. Por tanto, se debe especificar la ubicación de cada recurso humano.

El número de personas requeridas para un proyecto de software puede determinarse sólo después de hacer una estimación del esfuerzo de desarrollo (por ejemplo, persona-meses). Más adelante, en este capítulo, se estudian técnicas para estimar el esfuerzo.

26.4.2 Recursos de software reutilizables

La ingeniería de software basada en componentes (ISBC)[4] pone el énfasis en la reusabilidad; es decir, en la creación y reutilización de bloques constructores de software. Tales bloques constructores, con frecuencia llamados *componentes*, deben catalogarse para facilitar su referencia, estandarizarse para facilitar su aplicación y validarse para facilitar su integración. Bennatan [Ben00] sugiere cuatro categorías de recursos de software que deben considerarse conforme avanza la planificación:

Nunca olvide que integrar varios componentes reutilizables puede ser un reto considerable. Peor aún, el problema de la integración sale a la superficie conforme varios componentes se actualizan.

Componentes comerciales. Es el software existente que puede adquirirse de una tercera parte o de un proyecto anterior. Los componentes COTS (por las siglas en inglés de "anaqueles comerciales") se compran de una tercera parte, están listos para su uso en el proyecto actual y están completamente validados.

Componentes de experiencia completa. Son especificaciones, diseños, código o datos de prueba existentes, desarrollados para proyectos anteriores que son similares al software que se va a construir para el proyecto en cuestión. Los miembros del equipo de software tienen mucha experiencia en el área de aplicación representada por dichos componentes. Por tanto, las modificaciones requeridas para los componentes de experiencia completa tendrán un riesgo relativamente bajo.

4 La ISBC se consideró en el capítulo 10.

Componentes de experiencia parcial. Son especificaciones, diseños, código o datos de prueba existentes, desarrollados para proyectos anteriores que se relacionan con el software que se va a construir para el proyecto en cuestión, pero que requerirán modificación sustancial. Los miembros del equipo de software sólo tienen experiencia limitada en el área de aplicación especificada por dichos componentes. Por tanto, las modificaciones requeridas por los componentes de experiencia parcial entrañan un buen grado de riesgo.

Componentes nuevos. Son componentes de software que el equipo de software debe construir específicamente para las necesidades del proyecto en cuestión.

Irónicamente, los componentes de software reutilizables con frecuencia se desprecian durante la planificación, sólo para convertirse en una preocupación suprema más tarde, durante el avance del software. Es mejor especificar tempranamente los requisitos de recurso del software. De esta forma puede realizarse una evaluación técnica de las alternativas, así como la adquisición oportuna.

26.4.3 Recursos ambientales

El entorno que soporta a un proyecto de software, con frecuencia llamado *entorno de ingeniería de software* (EIS), incorpora hardware y software. El hardware proporciona una plataforma que soporta las herramientas (software) requeridas para producir los productos operativos que son resultado de la buena práctica de la ingeniería de software.[5] Puesto que la mayoría de las organizaciones de software tienen múltiples circunscripciones que requieren acceso al EIS, debe prescribirse la ventana temporal requerida para hardware y software y verificar que dichos recursos estarán disponibles.

Cuando un sistema basado en computadora (que incorpore hardware y software especializados) debe someterse a ingeniería, el equipo de software puede requerir acceso a elementos de hardware que se va a desarrollar por otros equipos de ingeniería. Por ejemplo, el software para un dispositivo robótico utilizado dentro de una célula de fabricación puede requerir un robot específico (por ejemplo, un soldador robótico) como parte del paso de prueba de validación; un proyecto de software para plantilla de página avanzada acaso necesite un sistema de impresión digital de alta velocidad en algún momento durante el desarrollo. Cada elemento de hardware debe especificarse como parte de la planificación.

26.5 Estimación de proyectos de software

Cita:

"En una época de outsourcing y creciente competencia, la capacidad para estimar con mayor precisión... ha surgido como un factor crítico de éxito para muchos grupos de TI."

Rob Thomsett

La estimación de costo y esfuerzo del software nunca será una ciencia exacta. Demasiadas variables (humanas, técnicas, ambientales, políticas) pueden afectar el costo final del software y el esfuerzo aplicado para su desarrollo. Sin embargo, la estimación del proyecto de software puede transformarse de un arte oscuro a una serie de pasos sistemáticos que proporcionen estimaciones con riesgo aceptable. Para lograr estimaciones confiables de costo y esfuerzo, surgen algunas opciones:

1. Retrase la estimación hasta avanzado el proyecto (obviamente, ¡puede lograr estimaciones 100 por ciento precisas después de que el proyecto esté completo!).
2. Base las estimaciones en proyectos similares que ya estén completos.
3. Use técnicas de descomposición relativamente simples para generar estimaciones de costo y esfuerzo de proyecto.
4. Use uno o más modelos empíricos para estimación de costo y esfuerzo de software.

5 Otro hardware, el *entorno meta*, es la computadora sobre la cual se ejecutará el software cuando éste se haya liberado al usuario final.

Desafortunadamente, la primera opción, aunque atractiva, no es práctica. Las estimaciones de costo deben proporcionarse por anticipado. No obstante, debe reconocer que mientras más espere, más conocerá, y mientras más conozca, menos probabilidades tendrá de cometer errores serios en sus estimaciones.

Cita:

"Es muy difícil hacer una defensa vigorosa, plausible y arriesgada de una estimación que se infiera sin método cuantitativo, apoyada por pocos datos y certificada principalmente por las corazonadas de los gerentes."

Fred Brooks

La segunda opción puede funcionar razonablemente bien si el proyecto actual es muy similar a esfuerzos anteriores y otros factores que influyen en el proyecto (por ejemplo, el cliente, condiciones del negocio, entorno de ingeniería de software, fechas límite) son aproximadamente equivalentes. Desafortunadamente, la experiencia pasada no siempre es buen indicador de resultados futuros.

Las opciones restantes son enfoques viables para la estimación del proyecto de software. De manera ideal, las técnicas anotadas para cada opción deben aplicarse en cascada y cada una es una comprobación cruzada para las demás. Las técnicas de descomposición tienen un enfoque de "divide y vencerás" para la estimación del proyecto. Al descomponer un proyecto en funciones principales y actividades de ingeniería de software relacionadas, la estimación de costo y esfuerzo puede realizarse en forma escalonada. Los modelos de estimación empírica pueden usarse para complementar las técnicas de descomposición y ofrecer un enfoque de estimación potencialmente valioso por derecho propio. Un modelo se basa en la experiencia (datos históricos) y toma la forma

$$d = f(v_i)$$

donde d es uno de los valores estimados (por ejemplo, esfuerzo, costo, duración del proyecto) y v_i son parámetros independientes seleccionados (por ejemplo, LOC o PF estimadas).

Las herramientas de estimación automatizadas implementan una o más técnicas de descomposición o modelos empíricos y proporcionan una atractiva opción para estimar. En tales sistemas se describen las características de la organización de desarrollo (por ejemplo, experiencia, entorno) y el software que se va a desarrollar. Las estimaciones de costo y esfuerzo se infieren de dichos datos.

Cada una de las opciones de estimación de costo del software viables sólo es tan buena como los datos históricos usados para generar la estimación. Si no existen datos históricos, el cálculo descansa sobre un cimiento muy inseguro. En el capítulo 25 se examinan las características de algunas de las métricas de software que proporcionan la base para los datos de estimación históricos.

26.6 Técnicas de descomposición

La estimación del proyecto de software es una forma de resolución de problemas y, en la mayoría de los casos, el problema por resolver; es decir, desarrollar una estimación de costo y esfuerzo para un proyecto de software es muy complejo como para considerarse en una sola pieza. Por esta razón, debe descomponerse el problema y volver a caracterizarlo como un conjunto de problemas más pequeños (y, esperanzadoramente, más manejables).

En el capítulo 24, el enfoque de descomposición se analizó desde dos puntos de vista diferentes: descomposición del problema y descomposición del proceso. La estimación usa una o ambas formas de división. Pero antes de hacer una estimación, debe entenderse el ámbito del software que se va a construir y generar una estimación de su "tamaño".

PUNTO CLAVE

El "tamaño" del software que se va a construir puede estimarse usando una medida directa, LOC, o indirecta, PF.

26.6.1 Dimensionamiento del software

La precisión de una estimación de proyecto de software se basa en algunas cosas: 1) el grado en el que se estimó adecuadamente el tamaño del producto que se va a construir, 2) la habilidad para traducir la estimación de tamaño en esfuerzo humano, tiempo calendario y dinero (una

función de la disponibilidad de métricas de software confiables de proyectos anteriores), 3) el grado en el que el plan del proyecto refleja las habilidades del equipo de software y 4) la estabilidad de los requisitos del producto y el entorno que soporta el esfuerzo de ingeniería de software.

En esta sección se considera el problema del *dimensionamiento del software*. Puesto que una estimación de proyecto sólo es tan buena como la estimación del tamaño del trabajo que se va a realizar, el dimensionamiento representa el primer gran desafío como planificador. En el contexto de la planificación del software, el tamaño se refiere a un resultado cuantificable del proyecto de software. Si se toma un enfoque directo, el tamaño puede medirse en líneas de código (LOC). Si se elige un enfoque indirecto, el tamaño se representa como puntos de función (PF).

Putnam y Myers [Put92] sugieren cuatro enfoques diferentes para el problema de dimensionamiento:

¿Cómo se dimensiona el software que se planea construir?

- *Dimensionamiento de "lógica difusa"*. Este enfoque usa las técnicas de razonamiento aproximadas que son la piedra angular de la lógica difusa. Para aplicar este enfoque, el planificador debe identificar el tipo de aplicación, establecer su magnitud en una escala cualitativa y luego refinar la magnitud dentro del rango original.
- *Dimensionamiento del punto de función*. El planificador desarrolla estimaciones de las características del dominio de información que se estudiaron en el capítulo 23.
- *Dimensionamiento de componente estándar*. El software está compuesto de algunos "componentes estándares" diferentes que son genéricos a un área de aplicación particular. Por ejemplo, los componentes estándares para un sistema de información son subsistemas, módulos, pantallas, reportes, programas interactivos, programas en lote, archivos, LOC e instrucciones en el nivel objeto. El planificador del proyecto estima el número de ocurrencias de cada componente estándar y luego usa datos de proyecto históricos para estimar el tamaño entregado por componente estándar.
- *Dimensionamiento del cambio*. Este enfoque se usa cuando un proyecto abarca el uso de software existente que debe modificarse en alguna forma como parte de un proyecto. El planificador estima el número y tipo (por ejemplo, reuso, código agregado, cambio de código, código borrado) de las modificaciones que deben lograrse.

Putnam y Myers sugieren que los resultados de cada uno de estos enfoques de dimensionamiento se combinen estadísticamente para crear una estimación de *tres puntos* o *valor esperado*. Esto se logra al desarrollar valores para tamaño optimistas (bajos), más probables y pesimistas (altos), y al combinarlos usando la ecuación 26.1, descrita en la sección 26.6.2.

26.6.2 Estimación basada en problema

En el capítulo 25 se describieron las líneas de código y los puntos de función como medidas a partir de las cuales pueden calcularse métricas de productividad. Los datos LOC y PF se usan en dos formas durante la estimación del proyecto de software: 1) como variables de estimación para "dimensionar" cada elemento del software y 2) como métricas de referencia recopiladas de proyectos pasados y utilizadas en conjunto con variables de estimación para desarrollar proyecciones de costo y esfuerzo.

¿Que tienen en común las estimaciones basadas en LOC y en PF?

Las estimaciones LOC y PF son técnicas de estimación distintas, aunque ambas tienen algunas características en común. Comience con un enunciado acotado del ámbito del software y a partir de este enunciado intente descomponer el enunciado de ámbito en funciones problema que puedan estimarse cada una de manera individual. De modo alternativo, puede elegir otro componente para dimensionamiento, como clases u objetos, cambios o procesos empresariales afectados.

Las métricas de productividad de referencia (por ejemplo, LOC/pm o PF/pm[6]) se aplican entonces a la variable de estimación adecuada y se infiere el costo o esfuerzo para la función. Las estimaciones de función se combinan para producir una estimación global para todo el proyecto.

Cuando recopile métricas de productividad para proyectos, asegúrese de establecer una taxonomía de tipos de proyecto. Esto le permitirá calcular promedios específicos de dominio y hacer estimaciones más precisas.

Sin embargo, es importante observar que con frecuencia existe una sustancial dispersión en las métricas de productividad para una organización, lo que hace sospechoso el uso de una sola métrica de referencia para la productividad. En general, los promedios de LOC/pm o PF/pm deben calcularse por dominio de proyecto. Es decir, los proyectos deben agruparse por tamaño de equipo, área de aplicación, complejidad y otros parámetros relevantes. Luego se calculan los promedios de dominio local. Cuando estime un nuevo proyecto, primero debe asignarlo a un dominio y después debe usar un promedio de dominio adecuado para productividad anterior a la generación de la estimación.

Las técnicas de estimación LOC y PF difieren en el nivel de detalle requerido para descomposición y en la meta de la partición. Cuando se usa LOC como la variable de estimación, la descomposición es absolutamente esencial y con frecuencia lleva a considerables niveles de detalle. Mientras mayor sea el grado de partición, es más probable que puedan desarrollarse estimaciones de LOC razonablemente precisas.

Para estimaciones PF, la descomposición funciona de modo diferente. En lugar de enfocarse en la función, se estima cada una de las características del dominio de información (entradas, salidas, archivos de datos, consultas e interfaces externas), así como los 14 valores de ajuste de complejidad que se estudiaron en el capítulo 23. Entonces las estimaciones resultantes pueden usarse para inferir un valor PF que pueda ligarse a datos pasados y usarse para generar una estimación.

Sin importar la variable de estimación que se utilice, debe comenzar por estimar un rango de valores para cada función o valor de dominio de información. Con el uso de datos históricos o (cuando todo lo demás falle) de la intuición, estime un valor de tamaño optimista, más probable y pesimista para cada función o conteo para cada valor de dominio de información. Cuando se especifica un rango de valores, se proporciona un indicio implícito del grado de incertidumbre.

¿Cómo se calcula el "valor esperado" para el tamaño del software?

Entonces puede calcularse un valor de tres puntos o esperado. El *valor esperado* para la variable de estimación (tamaño) S puede calcularse como un promedio ponderado de las estimaciones optimista (S_{opt}), más probable (S_m) y pesimista (S_{pes}). Por ejemplo,

$$S = \frac{S_{opt} + 4S_m + S_{pes}}{6} \tag{26.1}$$

le da más crédito a la estimación "más probable" y sigue una distribución de probabilidad beta. Se supone que hay una probabilidad muy pequeña de que el resultado de tamaño real se ubicará afuera de los valores optimista o pesimista.

Una vez determinado el valor esperado para la variable de estimación se aplican datos de productividad históricos LOC o PF. ¿Las estimaciones son correctas? La única respuesta razonable a esta pregunta es "no puede estar seguro". Cualquier técnica de estimación, sin importar su sofisticación, debe verificarse con otro enfoque. Incluso así, deben prevalecer el sentido común y la experiencia.

26.6.3 Un ejemplo de estimación basada en LOC

Como ejemplo de técnicas de estimación LOC y PF basadas en problema, considere un paquete de software que se va a desarrollar para una aplicación de diseño asistido por computadora para componentes mecánicos. El software debe ejecutarse en una estación de trabajo de ingeniería y tener interfaz con varios periféricos de gráficos de computadora, incluido un ratón, digitaliza-

6 El acrónimo *pm* significa persona-mes de esfuerzo.

dora, pantalla a color de alta resolución e impresora láser. Es posible desarrollar un enunciado preliminar del ámbito del software:

> El software CAD mecánico aceptará datos geométricos bidimensionales y tridimensionales de un ingeniero. El ingeniero interactuará y controlará el sistema CAD a través de una interfaz de usuario que mostrará características de buen diseño de interfaz hombre/máquina. Todos los datos geométricos y otra información de apoyo se mantendrán en una base de datos CAD. Los módulos de análisis de diseño se desarrollarán para producir la salida requerida, que se desplegará en varios dispositivos gráficos. El software se diseñará para controlar e interactuar con dispositivos periféricos que incluyen un ratón, digitalizadora, impresora láser y plotter.

Muchas aplicaciones modernas residen en una red o son parte de una arquitectura cliente-servidor. Por tanto, asegúrese de que sus estimaciones incluyen el esfuerzo requerido para desarrollar "infraestructura" de software.

Este enunciado de ámbito es preliminar, no está acotado. Cada oración tendría que expandirse para proporcionar detalle concreto y acotamiento cuantitativo. Por ejemplo, antes de comenzar la estimación, el planificador debe determinar qué significa "características de buen diseño de interfaz hombre/máquina" o cuáles serán el tamaño y sofisticación de la "base de datos CAD".

Para los propósitos señalados, suponga que ocurrió mayor refinamiento y que se identifican las principales funciones del software mencionadas en la figura 26.2. Después de la técnica de descomposición para LOC se elabora una tabla de estimación (figura 26.2). Para cada función se desarrollan estimaciones para un rango de LOC. Por ejemplo, el rango de estimaciones LOC para la función de análisis geométrico 3D es optimista, 4 600 LOC; más probablemente, 6 900 LOC, y pesimista, 8 600 LOC. Al aplicar la ecuación 26.1, el valor esperado para la función de análisis geométrico 3D es 6 800 LOC. Otras estimaciones se infieren en forma similar. Al sumar verticalmente en la columna LOC estimada, para el sistema CAD se establece un estimado de 33 200 líneas de código.

No sucumba a la tentación de usar este resultado como su estimación de proyecto. Debe inferir otro resultado usando un enfoque diferente.

Una revisión de los datos históricos indica que la productividad organizacional promedio para los sistemas de este tipo es 620 LOC/pm. Con base en una tarifa de mano de obra sobrecargada de US$8 000 por mes, el costo por línea de código es aproximadamente US$13. Con base en la estimación LOC y los datos de productividad históricos, el costo de proyecto total estimado es US$431 000 y el esfuerzo estimado es 54 persona-meses.[7]

26.6.4 Un ejemplo de estimación basada en PF

La descomposición para estimación basada en PF se enfoca en valores de dominio de información en lugar de en funciones del software. Con base en la tabla que se presentó en la figura 26.3 se estimarían entradas, salidas, consultas, archivos e interfaces externas para el software CAD.

FIGURA 26.2

Tabla de estimación para los métodos LOC

Función	LOC estimadas
Interfaz de usuario y facilidades de control (IUFC)	2 300
Análisis geométrico bidimensional (AG2D)	5 300
Análisis geométrico tridimensional (AG3D)	6 800
Gestión de base de datos (GBD)	3 350
Facilidades de despliegue de gráficos de computadora (FDGC)	4 950
Función de control periférico (FCP)	2 100
Módulos de análisis de diseño (MAD)	8 400
Líneas de código estimadas	*33 200*

7 Las estimaciones se redondean a US$1 000 y persona-mes más cercanos. Mayor precisión es innecesaria e irreal, dadas las limitaciones de la precisión de la estimación.

CasaSegura

Estimación

La escena: Oficina de Doug Miller mientras comienza la planificación del proyecto.

Participantes: Doug Miller (gerente del equipo de ingeniería de software *CasaSegura*) y Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería de software del producto.

La conversación:

Doug: Necesitamos desarrollar una estimación del esfuerzo para el proyecto y luego debemos definir un microcalendario para el primer incremento y un macrocalendario para los incrementos restantes.

Vinod (afirma con la cabeza): Muy bien, pero todavía no hemos definido algún incremento.

Doug: Cierto, pero es por eso por lo que necesitamos estimar.

Jamie (frunce el ceño): ¿Quieres saber cuánto tiempo nos tomará?

Doug: Esto es lo que necesito. Primero, necesitamos descomponer funcionalmente el software *CasaSegura*... en un nivel superior... luego tenemos que determinar el número de líneas de código que tomará cada función... luego...

Jamie: ¡Vaya! ¿Cómo se supone que haremos eso?

Vinod: Yo lo hice en proyectos anteriores. Comienzas con casos de uso, determinas la funcionalidad requerida para implementar cada uno, estimas el conteo de LOC para cada pieza de la función. El mejor enfoque es hacer que todos estimen de manera independiente y luego comparar los resultados.

Doug: O puedes hacer una descomposición funcional de todo el proyecto.

Jamie: Pero eso llevará mucho tiempo y ya teníamos que empezar.

Vinod: No... puede hacerse en pocas horas... esta mañana, de hecho.

Doug: Estoy de acuerdo... no podemos esperar exactitud, sólo una idea aproximada de cuál será el tamaño de *CasaSegura*.

Jamie: Creo que sólo debemos estimar el esfuerzo... es todo.

Doug: También haremos eso. Luego usen ambas estimaciones como comprobación cruzada.

Vinod: Vamos a hacerlo...

FIGURA 26.3
Estimación de información de valores de dominio

Valor de dominio de información	Opt.	Probable	Pes.	Conteo est.	Peso	Conteo PF
Número de entradas externas	20	24	30	24	4	97
Número de salidas externas	12	15	22	16	5	78
Número de consultas externas	16	22	28	22	5	88
Número de archivos lógicos internos	4	4	5	4	10	42
Número de archivos de interfaz externos	2	2	3	2	7	15
Conteo total						*320*

Un valor PF se calcula usando la técnica analizada en el capítulo 23. Para los propósitos de esta estimación se supone que el factor de ponderación de complejidad es el promedio. La figura 26.3 presenta los resultados de esta estimación.

Cada uno de los factores de ponderación de complejidad se estima y el factor de ajuste de valor se calcula como se describe en el capítulo 23:

Factor	Valor
Respaldo y recuperación	4
Comunicaciones de datos	2
Procesamiento distribuido	0
Rendimiento crítico	4
Existencia de entorno operativo	3
Entrada de datos en línea	4

Transacción de entrada sobre múltiples pantallas	5
Archivos maestros actualizados en línea	3
Complejo de valores de dominio de información	5
Complejo de procesamiento interno	5
Código diseñado para reuso	4
Conversión/instalación en diseño	3
Instalaciones múltiples	5
Aplicación diseñada para cambio	5
Factor de ajuste de valor	**1.17**

Finalmente, se infiere el número estimado de PF:

$$FP_{estimado} = \text{conteo total} \times [0.65 + 0.01 \times \Sigma(F_i)] = 375$$

La productividad organizacional promedio para sistemas de este tipo es 6.5 PF/pm. Con base en una tarifa de mano de obra sobrecargada de US$8 000 por mes, el costo por PF es aproximadamente US$1 230. Con base en el PF estimado y los datos de productividad históricos, el costo de proyecto estimado total es US$461 000 y el esfuerzo estimado es 58 personas-meses.

26.6.5 Estimación basada en proceso

La técnica más común para estimar un proyecto es basar la estimación sobre el proceso que se usará. Es decir, el proceso se descompone en un conjunto relativamente pequeño de tareas y se estima el esfuerzo requerido para lograr cada tarea.

Como en las técnicas basadas en problemas, la estimación basada en proceso comienza con un delineado de las funciones de software obtenidas del ámbito del proyecto. Para cada función debe realizarse una serie de actividades de marco conceptual. Las funciones y actividades de marco conceptual relacionadas[8] pueden representarse como parte de una tabla similar a la que se presentó en la figura 26.4.

Una vez fusionadas las funciones del problema y las actividades de proceso, se estima el esfuerzo (por ejemplo, persona-mes) que se requerirá para lograr cada actividad de proceso de

FIGURA 26.4

Tabla de estimación basada en proceso

Actividad →	CC	Planificación	Análisis de riesgo	Ingeniería		Construcción/liberación		CE	Totales
Tarea →				Análisis	Diseño	Código	Prueba		
Función									
↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totales	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% esfuerzo	1%	1%	1%	8%	45%	10%	36%		

CC = comunicación cliente CE = evaluación cliente

8 Las actividades de marco conceptual elegidas para este proyecto difieren un poco de las actividades genéricas estudiadas en el capítulo 2. Son: comunicación con el cliente (CC), planificación, análisis de riesgos, ingeniería y construcción/liberación.

software para cada función del software. Dichos datos constituyen la matriz central de la tabla de la figura 26.4. Las tarifas de mano de obra promedio (es decir, esfuerzo costo/unidad) se aplican entonces al esfuerzo estimado para cada actividad del proceso. Es muy probable que la tarifa de mano de obra varíe para cada tarea. El personal ejecutivo está enormemente involucrado en las primeras actividades de marco conceptual y por lo general son más costosos que el personal no ejecutivo involucrado en la construcción y liberación.

CONSEJO

Si el tiempo lo permite, use granularidad más fina cuando especifique las tareas de la figura 26.4. Por ejemplo, descomponga el análisis en sus principales tareas y estime cada una por separado.

Los costos y esfuerzos para cada función y actividad de marco conceptual se calculan igual que el último paso. Si la estimación basada en proceso se realiza independientemente de la estimación LOC o PF, entonces se tienen dos o tres estimaciones para costo y esfuerzo que pueden compararse y reconciliarse. Si ambos conjuntos de estimaciones muestran concordancia razonable, hay buenas razones para creer que las estimaciones son confiables. Si, por otra parte, los resultados de dichas técnicas de descomposición muestran poca concordancia, debe realizarse mayor investigación y análisis.

26.6.6 Un ejemplo de estimación basada en proceso

Para ilustrar el uso de la estimación basada en proceso, considere el software CAD que se presentó en la sección 26.6.3. La configuración del sistema y todas las funciones de software permanecen sin cambios y se indican por ámbito de proyecto.

Cita:

"Es mejor comprender el fondo de una estimación antes de usarla."

Barry Boehm y Richard Fairley

En la tabla completa basada en proceso que se muestra en la figura 26.4, las estimaciones de esfuerzo (en persona-meses) para cada actividad de ingeniería de software se proporcionan para cada función del software CAD (abreviada por conveniencia). Las actividades de ingeniería y construcción/liberación se subdividen en las principales tareas de ingeniería de software que se muestran. Para comunicación con el cliente, planificación y análisis de riesgo se proporcionan estimaciones burdas de esfuerzo. Las mismas se anotan en la hilera total al fondo de la tabla. Los totales horizontal y vertical proporcionan un indicio del esfuerzo estimado requerido para análisis, diseño, código y prueba. Debe observarse que 53 por ciento de todo el esfuerzo se emplea en tareas de ingeniería frontales (análisis de requisitos y diseño), lo que indica la relativa importancia de este trabajo.

Con base en una tarifa promedio de mano de obra sobrecargada de US$8 000 por mes, el costo total estimado promedio es US$368 000, y el esfuerzo estimado es 46 persona-mes. Si se desea, las tarifas de mano de obra podrían asociarse con cada actividad de marco conceptual o tarea de ingeniería de software y calcularse por separado.

26.6.7 Estimación con casos de uso

Como se señaló a lo largo de la parte 2 de este libro, los casos de uso brindan a un equipo de software comprensión acerca del ámbito y los requisitos del software. Sin embargo, desarrollar un enfoque de estimación con casos de uso es problemático por las siguientes razones [Smi99]:

? ¿Por qué es difícil desarrollar una técnica de estimación usando casos de uso?

- Los casos de uso se describen usando muchos formatos y estilos diferentes; no existe una forma estándar.
- Los casos de uso representan una visión externa (la visión del usuario) del software y, por tanto, pueden escribirse en muchos niveles de abstracción diferentes.
- Los casos de uso no abordan la complejidad de las funciones y de las características que se describen.
- Los casos de uso pueden describir comportamiento complejo (por ejemplo, interacciones) que involucran muchas funciones y características.

A diferencia de una LOC o de un punto de función, el "caso de uso" de una persona puede requerir meses de esfuerzo, mientras que el de otra puede implementarse en un día o dos.

Aunque algunos investigadores consideran los casos de uso como una entrada de estimación, a la fecha ningún método de estimación probado ha surgido.[9] Smith [Smi99] sugiere que los casos de uso pueden usarse para estimación, pero sólo si se consideran dentro del contexto de la "jerarquía estructural" donde se usan para describir.

Smith argumenta que cualquier nivel de esta jerarquía estructural puede describirse mediante no más de 10 casos de uso. Cada uno de éstos abarcaría no más de 30 escenarios distintos. Obviamente, los casos de uso que describen un sistema grande se escriben en un nivel de abstracción mucho más alto (y representan considerablemente más esfuerzo de desarrollo) que los que describen un solo subsistema. En consecuencia, antes de poder usar los casos de uso para estimación, se establece el nivel dentro de la jerarquía estructural, se determina la longitud promedio (en páginas) de cada caso de uso, se define el tipo de software (por ejemplo, tiempo real, empresarial, ingeniería/científico, *webapp*, incrustado) y se considera una arquitectura burda para el sistema. Una vez establecidas dichas características pueden usarse datos empíricos para establecer el número estimado de LOC o PF por caso de uso (por cada nivel de la jerarquía). Entonces se usan datos históricos a fin de calcular el esfuerzo requerido para desarrollar el sistema.

Para ilustrar cómo puede realizarse este cálculo, considere la siguiente relación:[10]

$$\text{LOC estimadas} = N \times \text{LOC}_{\text{prom}} + [(S_a/S_h - 1) + (P_a/P_h - 1)] \times \text{LOC}_{\text{ajuste}} \tag{26.2}$$

donde

N = número real de casos de uso
LOC_{prom} = LOC promedio históricas por caso de uso para este tipo de subsistema
$\text{LOC}_{\text{ajuste}}$ = representa un ajuste con base en n por ciento de LOC_{prom}, donde n se define localmente y representa la diferencia entre este proyecto y los proyectos "promedio"
S_a = escenarios reales por caso de uso
S_h = escenarios promedio por caso de uso para este tipo de subsistema
P_a = páginas reales por caso de uso
P_h = páginas promedio por caso de uso para este tipo de subsistema

La expresión 26.2 podría usarse para desarrollar una estimación burda del número de LOC con base en el número real de casos de uso ajustados por el número de escenarios y la longitud de página de los casos de uso. El ajuste representa hasta n por ciento de las LOC promedio históricas por caso de uso.

26.6.8 Un ejemplo de estimación basada en caso de uso

El software CAD introducido en la sección 26.6.3 se compone de tres grupos de subsistemas: subsistema de interfaz de usuario (incluye UICF), grupo de subsistemas de ingeniería (incluye los subsistemas 2DGA, 3DGA y DAM) y grupo de subsistemas de infraestructura (incluye los subsistemas CGDF y PCF). Seis casos de uso describen el subsistema de interfaz de usuario. Cada uno se describe mediante no más de 10 escenarios y tiene una longitud promedio de seis páginas. El grupo de subsistemas de ingeniería se describe mediante 10 casos de uso (se considera que están en un nivel superior de la jerarquía estructural). Cada uno de estos casos de uso tiene no más de 20 escenarios asociados con él y una longitud promedio de ocho páginas. Finalmente,

9 Trabajo reciente en la inferencia de *puntos de casos de uso* [Cle06] a final de cuentas puede conducir a un enfoque de estimación utilizable, usando casos de uso.

10 Es importante observar que la expresión 26.2 se usa exclusivamente con propósitos ilustrativos. Como todos los modelos de estimación, debe validarse localmente antes de que pueda usarse con confianza.

FIGURA 26.5
Estimación de caso de uso

	casos de uso	escenarios	páginas	escenarios	páginas	LOC	LOC estimadas
Subsistema de interfaz de usuario	6	10	6	12	5	560	3 366
Grupo de subsistemas de ingeniería	10	20	8	16	8	3 100	31 233
Grupo de subsistemas de infraestructura	5	6	5	10	6	1 650	7 970
Total LOC estimadas							42 568

el grupo de subsistemas de infraestructura se describe mediante cinco casos de uso con un promedio de sólo seis escenarios y una longitud promedio de cinco páginas.

Usando la relación anotada en la expresión 26.2, con n = 30 por ciento, se elaboró la tabla que se muestra en la figura 26.5. Observe la primera hilera de la tabla; los datos históricos indican que el software UI requiere un promedio de 800 LOC por caso de uso cuando el caso de uso no tiene más de 12 escenarios y se describe en menos de cinco páginas. Dichos datos se ajustan razonablemente bien para el sistema CAD. Por tanto, la estimación LOC para el subsistema de interfaz de usuario se calcula con la expresión 26.2. Usando el mismo enfoque se hacen estimaciones para los grupos de subsistemas de ingeniería e infraestructura. La figura 26.5 resume las estimaciones e indica que el tamaño global del CAD se estima en 42 500 LOC.

Con 620 LOC/pm como la productividad promedio para sistemas de este tipo y una tarifa de mano de obra sobrecargada de $8 000 por mes, el costo por línea de código es aproximadamente US$13. Con base en la estimación de caso de uso y los datos de productividad históricos, el costo total estimado del proyecto es US$552 000 y el esfuerzo estimado es 68 persona-meses.

26.6.9 Reconciliación de estimaciones

Cita:

"Los métodos complicados pueden no producir una estimación más precisa, en particular cuando los desarrolladores pueden incorporar su propia intuición en la estimación."

Philip Johnson *et al.*

Las técnicas de estimación estudiadas en las secciones anteriores dan como resultado estimaciones múltiples que deben reconciliarse para producir una sola estimación de esfuerzo, duración de proyecto o costo. Para ilustrar este procedimiento de reconciliación, considere de nuevo el software CAD introducido en la sección 26.6.3.

El esfuerzo total estimado para el software CAD varía de uno bajo de 46 persona-meses (inferido con el enfoque de estimación basado en proceso) a uno alto de 68 persona-meses (inferido con estimación de caso de uso). La estimación promedio (usando los cuatro enfoques) es de 56 persona-meses. La variación de la estimación promedio es aproximadamente 18 por ciento en el lado bajo y 21 por ciento en el alto.

¿Qué ocurre cuando es pobre la concordancia entre las estimaciones? La respuesta a esta pregunta requiere una reevaluación de la información usada para hacer las estimaciones. Las estimaciones ampliamente divergentes con frecuencia pueden tener una de dos causas: 1) el ámbito del proyecto no se entiende adecuadamente o el planificador lo malinterpretó o 2) los datos de productividad usados por las técnicas de estimación basadas en problema son inadecuadas para la aplicación, obsoletos (ya no reflejan con precisión la organización de ingeniería de software) o se aplicaron mal. Debe determinar la causa de la divergencia y luego reconciliar las estimaciones.

INFORMACIÓN

Técnicas de estimación automatizada para proyectos de software

Las herramientas de estimación automatizadas permiten al planificador estimar costo y esfuerzo, y realizar análisis "y... si" para variables de proyecto importantes, como fecha de entrega o personal. Aunque existen muchas herramientas automatizadas (vea la barra lateral más adelante en este capítulo), todas muestran las mismas características generales y todas realizan las siguientes seis funciones genéricas [Jon96]:

1. *Dimensionamiento de entregas de proyecto.* Estimación del "tamaño" de uno o más productos operativos de software. Los

productos operativos incluyen la representación externa del software (por ejemplo, pantalla, reportes), el software en sí (por ejemplo, KLOC), funcionalidad entregada (por ejemplo, puntos de función) e información descriptiva (por ejemplo, documentos).
2. *Selección de actividades de proyecto*. Selección del marco conceptual de proceso adecuado y especificación del conjunto de tareas de ingeniería de software.
3. *Predicción de niveles de personal*. Especificación del número de personas que estarán disponibles para hacer el trabajo. Puesto que la relación entre personal disponible y trabajo (esfuerzo predicho) es enormemente no lineal, ésta es una entrada importante.
4. *Predicción de esfuerzo de software*. Las herramientas de estimación usan uno o más modelos (sección 26.7) que relacionan el tamaño de las entregas del proyecto con el esfuerzo requerido para producirlos.
5. *Predicción del costo de software*. Dados los resultados del paso 4, los costos pueden calcularse asignando tarifas de mano de obra a las actividades de proyecto anotadas en el paso 2.
6. *Predicción de calendarios de software*. Cuando se conocen el esfuerzo, el nivel de personal y las actividades del proyecto, puede producirse un calendario tentativo para asignar la mano de obra a través de las actividades de ingeniería de software con base en modelos recomendados para distribución de esfuerzo, lo que se estudia más adelante en este capítulo.

Cuando diferentes herramientas de estimación se aplican a los mismos datos de proyecto, puede encontrarse una variación relativamente grande en los resultados estimados. Más importante, en ocasiones, los valores predichos son significativamente diferentes a los valores reales. Esto refuerza la noción de que la salida de las herramientas de estimación debe usarse como un "punto de datos" de los cuales se derivan estimaciones, no como la única fuente para una estimación.

26.7 Modelos de estimación empíricos

Un modelo de estimación para software de computadora usa fórmulas empíricamente inferidas para predecir el esfuerzo como una función de LOC o PF.[11] Los valores para LOC o PF se estiman usando el enfoque descrito en las secciones 26.6.3 y 26.6.4. Pero en lugar de usar las tablas descritas en dichas secciones, los valores resultantes de LOC o PF se alimentan con el modelo de estimación.

PUNTO CLAVE

Un modelo de estimación refleja la población de proyectos de los cuales se derivó. Por tanto, el modelo es sensible al dominio.

Los datos empíricos que soportan a la mayoría de los modelos de estimación se infieren de una muestra limitada de proyectos. Por esta razón, ningún modelo de estimación es adecuado para todas las clases de software y en todos los entornos de desarrollo. Por tanto, se deben usar juiciosamente los resultados obtenidos de tales modelos.

Un modelo de estimación debe calibrarse para que refleje las condiciones locales. El modelo debe probarse aplicando los datos recopilados de los proyectos completados, alimentando los datos en el modelo y luego comparando los resultados reales con los predichos. Si la concordancia es pobre, el modelo debe afinarse y volverse a probar antes de poder usarse.

26.7.1 La estructura de los modelos de estimación

Un modelo de estimación típico se infiere usando análisis de regresión sobre los datos recopilados de proyectos de software anteriores. La estructura global de tales modelos toma la forma [Mat94]

$$E = A + B \times (e_v)^C \tag{26.3}$$

donde A, B y C son constantes derivadas empíricamente, E es esfuerzo en persona-meses y e_v es la variable de estimación (LOC o PF). Además de la relación anotada en la ecuación 26.3, la mayoría de los modelos de estimación tienen alguna forma de componente de ajuste de proyecto que permite que E se ajuste mediante otras características del proyecto (por ejemplo,

11 En la sección 26.6.6 se sugiere un modelo empírico que use casos de uso como la variable independiente. Sin embargo, a la fecha, en la literatura han aparecido relativamente pocos.

complejidad del problema, experiencia del personal, entorno de desarrollo). Entre los muchos modelos de estimación orientados a LOC propuestos en la literatura, están:

CONSEJO

Ninguno de estos modelos debe usarse sin calibración cuidadosa en su entorno.

$E = 5.2 \times (\text{KLOC})^{0.91}$	Modelo Walston-Felix
$E = 5.5 + 0.73 \times (\text{KLOC})^{1.16}$	Modelo Bailey-Basili
$E = 3.2 \times (\text{KLOC})^{1.05}$	Modelo Boehm simple
$E = 5.288 \times (\text{KLOC})^{1.047}$	Modelo Doty para KLOC > 9

También se han propuesto modelos orientados a PF. En ellos se incluyen:

$E = -91.4 + 0.355\ \text{PF}$	Modelo Albrecht y Gaffney
$E = -37 + 0.96\ \text{PF}$	Modelo Kemerer
$E = -12.88 + 0.405\ \text{PF}$	Pequeño modelo de regresión de proyecto

Un examen rápido a dichos modelos indica que cada uno producirá un resultado diferente para los mismos valores de LOC o PF. La implicación es clara. ¡Los modelos de estimación deben calibrarse para las necesidades locales!

26.7.2 El modelo COCOMO II

WebRef

En **sunset.usc.edu/research/COCOMOII/cocomo_main.html**, puede obtenerse información detallada acerca de COCOMO II, incluido software descargable.

En su libro clásico acerca de "economía de la ingeniería de software", Barry Boehm [Boe81] introdujo una jerarquía de modelos de estimación de software que llevan el nombre COCOMO, por *COnstructive COst MOdel*: modelo constructivo de costos. El modelo COCOMO original se convirtió en uno de los modelos de estimación de costo más ampliamente utilizados y estudiados en la industria. Evolucionó hacia un modelo de estimación más exhaustivo, llamado COCOMO II [Boe00]. Como su predecesor, COCOMO II en realidad es una jerarquía de modelos de estimación que aborda las áreas siguientes:

- *Modelo de composición de aplicación*. Se usa durante las primeras etapas de la ingeniería de software, cuando son primordiales la elaboración de prototipos de las interfaces de usuario, la consideración de la interacción del software y el sistema, la valoración del rendimiento y la evaluación de la madurez de la tecnología.
- *Modelo de etapa temprana de diseño*. Se usa una vez estabilizados los requisitos y establecida la arquitectura básica del software.
- *Modelo de etapa postarquitectónica*. Se usa durante la construcción del software.

Como todos los modelos de estimación para software, los modelos COCOMO II requieren información sobre dimensionamiento. Como parte de la jerarquía del modelo, están disponibles tres diferentes opciones de dimensionamiento: puntos objeto, puntos de función y líneas de código fuente.

? ¿Qué es un punto de objeto?

El modelo de composición de aplicación COCOMO II usa puntos de objeto y se ilustra en los siguientes párrafos. Debe observarse que otros modelos de estimación, más sofisticados (que usan PF y KLOC), también están disponibles como parte de COCOMO II.

Como los puntos de función, el *punto de objeto* es una medida de software indirecta que se calcula usando conteos del número de 1) pantallas (en la interfaz de usuario), 2) reportes y 3)

FIGURA 26.6

Ponderación de complejidad para tipos de objeto.
Fuente: [Boe96]

Tipo de objeto	Peso de complejidad		
	Simple	Medio	Difícil
Pantalla	1	2	3
Reporte	2	5	8
Componente 3GL			10

componentes que probablemente se requieran para construir la aplicación. Cada instancia de objeto (por ejemplo, una pantalla o reporte) se clasifica en uno de tres niveles de complejidad (simple, medio o difícil), usando criterios sugeridos por Boehm [Boe96]. En esencia, la complejidad es una función del número y de la fuente de las tablas de datos de cliente y servidor que se requieren para generar la pantalla o el reporte y el número de vistas o secciones que se presentan como parte de la pantalla o del reporte.

Una vez determinada la complejidad, el número de pantallas, reportes y componentes se ponderan de acuerdo con la tabla que se ilustra en la figura 26.6. Entonces se determina el conteo de puntos de objeto multiplicando el número original de instancias de objeto por el factor de ponderación que hay en la figura y se suman para obtener un conteo total de puntos de objeto. Cuando debe aplicarse desarrollo basado en componente o reuso de software general, se estima el porcentaje de reuso (%reuso) y el conteo de puntos de objeto se ajusta:

$$\text{NOP} = (\text{puntos de objeto}) \times [(100 - \%\text{reuso})/100]$$

donde NOP se define como nuevos puntos de objeto.

Para derivar una estimación del esfuerzo con base en el valor NOP calculado, debe derivarse una "tasa de productividad". La figura 26.7 presenta la tasa de productividad

$$\text{PROD} = \frac{\text{NOP}}{\text{persona-mes}}$$

para diferentes niveles de experiencia del desarrollador y de madurez del entorno de desarrollo.

Una vez determinada la tasa de productividad se calcula una estimación del esfuerzo del proyecto usando

$$\text{Esfuerzo estimado} = \frac{\text{NOP}}{\text{PROD}}$$

En modelos COCOMO II más avanzados,[12] se requieren varios factores de escala, controladores de costo y procedimientos de ajuste. Una discusión completa de éstos está más allá del ámbito de este libro. Si tiene más interés, vea [Boe00] o visite el sitio web de COCOMO II.

26.7.3 La ecuación del software

La *ecuación del software* [Put92] es un modelo dinámico multivariable que supone una distribución de esfuerzo específica durante la vida de un proyecto de desarrollo de software. El modelo

FIGURA 26.7 **Tasa de productividad para puntos de objeto.**
Fuente: [Boe96].

Experiencia/capacidad del desarrollador	Muy baja	Baja	Nominal	Alta	Muy alta
Madurez/capacidad del entorno	Muy baja	Baja	Nominal	Alta	Muy alta
PROD	4	7	13	25	50

12 Como se señaló anteriormente, estos modelos usan conteos PF y KLOC para la variable tamaño.

se infirió a partir de datos de productividad recopilados por más de 4 000 proyectos de software contemporáneos. Con base en dichos datos, se infiere un modelo de estimación de la forma

$$E = \frac{\text{LOC} \times B^{0.333}}{P^3} \times \frac{1}{t^4} \tag{26.4}$$

donde

WebRef

En **www.qsm.com** puede encontrarse información acerca de herramientas de estimación de costo de software que evolucionaron a partir de la ecuación de software.

E = esfuerzo en persona-meses o persona-años
t = duración del proyecto en meses o años
B = "factor de habilidades especiales"[13]
P = "parámetro de productividad" que refleja: madurez global del proceso y prácticas administrativas, la medida en la que se usan buenas prácticas de ingeniería de software, el nivel de lenguajes de programación utilizado, el estado del entorno de software, las habilidades y experiencia del equipo de software y la complejidad de la aplicación.

Valores típicos pueden ser $P = 2\,000$ para desarrollo de un software incrustado en tiempo real, $P = 10\,000$ para software de telecomunicaciones y sistemas y $P = 28\,000$ para aplicaciones de sistemas empresariales. El parámetro de productividad puede inferirse para condiciones locales, usando datos históricos recopilados de esfuerzos de desarrollo anteriores.

Debe observarse que la ecuación de software tiene dos parámetros independientes: 1) una estimación del tamaño y 2) una indicación de la duración del proyecto en meses o años calendario.

Para simplificar el proceso de estimación y usar una forma más común para sus modelos de estimación, Putnam y Myers [Put92] sugieren un conjunto de ecuaciones derivadas de la ecuación de software. El tiempo mínimo de desarrollo se define como

$$t_{\text{mín}} = 8.14 \frac{\text{LOC}}{p^{0.43}} \text{ en meses para } t_{\text{mín}} > 6 \text{ meses} \tag{26.5a}$$

$$E = 180\, Bt^3 \text{ en persona-meses para } E \geq 20 \text{ persona-meses} \tag{26.5b}$$

Observe que t en la ecuación 26.5b se representa en años.

Con la ecuación 26.5, para $P = 12\,000$ (el valor recomendado para software científico) para el software CAD que se estudió anteriormente en este capítulo,

$$t_{\text{mín}} = 8.14 \times \frac{33\,200}{12\,000^{0.43}} = 12.6 \text{ meses calendario}$$

$$E = 180 \times 0.28 \times (1.05)^3 = 58 \text{ persona-meses}$$

Los resultados de la ecuación de software corresponden favorablemente con las estimaciones desarrolladas en la sección 26.6. Como el modelo COCOMO anotado en la sección 26.7.2, la ecuación de software sigue evolucionando. Un análisis más a fondo de una versión extendida de este enfoque de estimación puede encontrarse en [Put97b].

26.8 Estimación para proyectos orientados a objetos

Vale la pena complementar los métodos de estimación de costo de software convencional con una técnica que se diseñó explícitamente para software OO. Lorenz y Kidd [Lor94] sugieren el siguiente enfoque:

13 B aumenta lentamente conforme "crecen la necesidad para integración, pruebas, aseguramiento de la calidad, documentación y habilidades administrativas" [Put92]. Para programas pequeños (KLOC = 5 a 15), $B = 0.16$. Para programas mayores de 70 KLOC, $B = 0.39$.

1. Desarrollar estimaciones usando descomposición de esfuerzo, análisis PF y cualquier otro método que sea aplicable para aplicaciones convencionales.
2. Usar el modelo de requisitos (capítulo 6), desarrollar casos de uso y determinar un conteo. Reconocer que el número de casos de uso puede cambiar conforme avance el proyecto.
3. A partir del modelo de requisitos, determinar el número de clases clave (llamadas clases de análisis en el capítulo 6).
4. Categorizar el tipo de interfaz para la aplicación y desarrollar un multiplicador para clases de apoyo:

Tipo de interfaz	Multiplicador
No GUI	2.0
Interfaz de usuario basada en texto	2.25
GUI	2.5
GUI compleja	3.0

   Multiplique el número de clases clave (paso 3) por el multiplicador a fin de obtener una estimación para el número de clases de apoyo.
5. Multiplicar el número total de clases (clave + apoyo) por el número promedio de unidades de trabajo por clase. Lorenz y Kidd sugieren 15 a 20 persona-días por clase.
6. Comprobación cruzada de la estimación basada en clase, multiplicando el número promedio de unidades de trabajo por caso de uso.

26.9 Técnicas de estimación especializadas

Las técnicas de estimación estudiadas en las secciones 26.6 a 26.8 pueden usarse para cualquier proyecto de software. Sin embargo, cuando un equipo de software encuentra una duración de proyecto extremadamente corta (semanas en lugar de meses) en la que es probable tener un torrente continuo de cambios, la planificación del proyecto en general y la estimación en particular deben abreviarse.[14] En las siguientes secciones se examinan dos técnicas de estimación especializadas.

26.9.1 Estimación para desarrollo ágil

Puesto que los requisitos para un proyecto ágil (capítulo 3) se definen mediante un conjunto de escenarios de usuario (por ejemplo, "historias" en programación extrema), es posible desarrollar un enfoque de estimación que sea informal, razonablemente disciplinado y significativo dentro del contexto de la planificación del proyecto para cada incremento de software. La estimación para proyectos ágiles usa un enfoque de descomposición que abarca los siguientes pasos:

? ¿Cómo se desarrollan las estimaciones cuando se aplica un proceso ágil?

1. Cada escenario de usuario (el equivalente de un minicaso de uso creado al comienzo mismo de un proyecto por los usuarios finales u otros participantes) se considera por separado con propósitos de estimación.
2. El escenario se descompone en el conjunto de tareas de ingeniería de software que será necesario desarrollar.

14 "Abreviar" *no* significa eliminar. Incluso los proyectos de corta duración deben planificarse, y la estimación es el cimiento de la planificación sólida.

3a. El esfuerzo requerido por cada tarea se estima por separado. Nota: La estimación puede basarse en datos históricos, un modelo empírico o la "experiencia".

3b. De manera alternativa, el "volumen" del escenario puede estimarse en LOC, PF o alguna otra medida orientada a volumen (por ejemplo, conteo de casos de uso).

4a. Las estimaciones para cada tarea se suman a fin de crear una estimación para el escenario.

4b. De manera alternativa, la estimación de volumen para el escenario se traduce en esfuerzo, usando datos históricos.

5. Las estimaciones de esfuerzo para todos los escenarios que se implementan para un incremento de software determinado se suman a fin de desarrollar la estimación del esfuerzo para el incremento.

PUNTO CLAVE

En el contexto de la estimación para proyectos ágiles, "volumen" es una estimación del tamaño global de un escenario de usuario en LOC o PF.

Puesto que la duración del proyecto requerido para el desarrollo de un incremento de software es muy corta (por lo general de tres a seis semanas), este enfoque de estimación tiene dos propósitos: 1) asegurarse de que el número de escenarios que se van a incluir en el incremento se ajusta a los recursos disponibles y 2) establecer una base para asignar esfuerzo conforme se desarrolla el incremento.

26.9.2 Estimación para webapp

Los *webapps* adoptan con frecuencia el modelo de proceso ágil. Puede usarse una medida de punto de función modificada, junto con los pasos que se destacan en la sección 26.9.1, a fin de desarrollar una estimación para la *webapp*. Roetzheim [Roe00] sugiere el siguiente enfoque cuando adapta puntos de función para estimación de *webapps*:

- *Entradas* son cada pantalla o formulario de entrada (por ejemplo, CGI o Java), cada pantalla de mantenimiento y, si usa una metáfora de etiquetas de libreta, cualquier etiqueta.
- *Salidas* son cada página web estática, cada guión de página web dinámica (por ejemplo, ASP, ISAPI u otro guión DHTML) y cada reporte (ya sea basado en web o administrativo por naturaleza).
- *Tablas* son cada tabla lógica en la base de datos más, si usa XML para almacenar datos en un archivo, cada objeto XML (o colección de atributos XML).
- Las *interfaces* conservan su definición como archivos lógicos (por ejemplo, formatos de registro único) dentro de las fronteras del sistema.
- *Consultas* son cada una de las publicaciones externas o el uso de una interfaz orientada a mensaje. Un ejemplo usual son las referencias externas DCOM o COM.

Los puntos de función (interpretados en la forma señalada) son un indicador razonable de volumen para una *webapp*.

Mendes *et al.* [Men01] sugieren que el volumen de una *webapp* se determina mejor al recolectar medidas (llamadas "variables predictoras") asociadas con la aplicación (por ejemplo, conteo de página, conteo de medios, conteo de función), características de su página web (por ejemplo, complejidad de página, complejidad de vinculación, complejidad gráfica), características de medios (por ejemplo, duración de los medios) y características funcionales (por ejemplo, longitud de código, longitud de código reutilizado). Dichas medidas pueden usarse para desarrollar modelos de estimación empíricos para esfuerzo de proyecto total, esfuerzo de autoría de página, esfuerzo de autoría de medios y esfuerzo de guiones. Sin embargo, todavía falta trabajo por hacer antes de que tales modelos puedan usarse con confianza.

HERRAMIENTAS DE SOFTWARE

Estimaciones de esfuerzo y costo

Objetivo: El objetivo de las herramientas de estimación de esfuerzo y costo es brindar a un equipo de proyecto estimaciones del esfuerzo requerido, duración del proyecto y costo, de manera que aborde las características específicas del proyecto a mano y el entorno donde se construirá el proyecto.

Mecánica: En general, las herramientas de estimación de costo usan una base de datos histórica inferida de proyectos locales y datos recopilados a través de la industria, y un modelo empírico (por ejemplo, COCOMO II) que se utiliza para derivar estimaciones de esfuerzo, duración y costo. Las características del proyecto y el entorno de desarrollo son entradas y la herramienta proporciona un rango de salidas de estimación.

Herramientas representativas:[15]

Costar, desarrollada por Softstar Systems (**www.softstarsystems.com**), usa el modelo COCOMO II para desarrollar estimaciones de software.

CostXpert, desarrollada por Cost Xpert Group, Inc. (**www.costxpert.com**) integra múltiples modelos de estimación y una base de datos histórica de proyectos.

Estimate Professional, desarrollada por el Software Productivity Centre, Inc. (**www.spc.com**), se basa en el COCOMO II y en el modelo SLIM.

Knowledge Plan, desarrollado por Software Productivity Research (**www.spr.com**), usa entrada de puntos de función como el principal controlador para un paquete de estimación completo.

Price S, desarrollada por Price Systems (**www.pricesystems.com**), es una de las herramientas de estimación más antiguas y de más uso para proyectos de desarrollo de software a gran escala.

SEER/SEM, desarrollada por Galorath, Inc. (**www.galorath.com**), proporciona una amplia capacidad de estimación, análisis de sensibilidad, valoración de riesgo y otras características.

SLIM-Estimate, desarrollada por QSM (**www.qsm.com**), se apoya en una exhaustiva "base de conocimiento industrial" para proporcionar una "comprobación de sanidad" para estimaciones que se infieren usando datos locales.

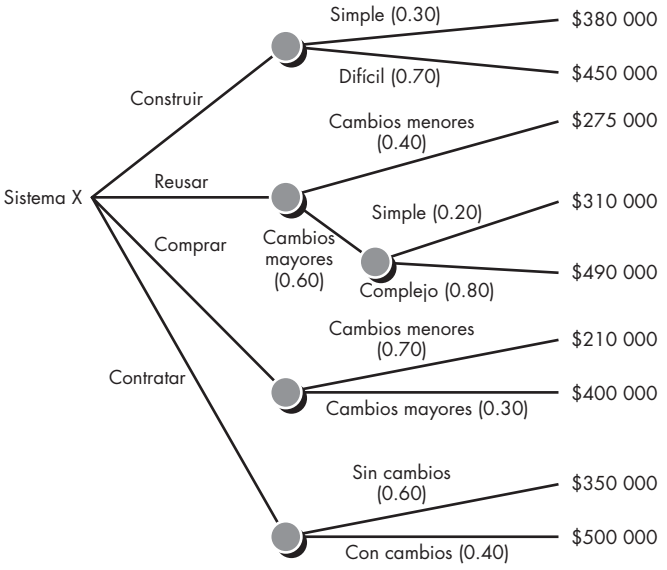
26.10 LA DECISIÓN HACER/COMPRAR

En muchas áreas de aplicación, con frecuencia es más efectivo en costo adquirir, en lugar de desarrollar, software de computadora. Los gerentes de ingeniería de software se enfrentan a la decisión hacer/comprar que puede complicarse todavía más por algunas opciones de adquisición: 1) el software puede comprarse (o licenciarse) de manera comercial, 2) los componentes de software de "experiencia completa" o "experiencia parcial" (vea la sección 26.4.2) pueden adquirirse y luego modificarse e integrarse para satisfacer necesidades específicas o 3) el software puede construirse a la medida por parte de un contratista externo para satisfacer las especificaciones del comprador.

Los pasos involucrados en la adquisición del software se definen por lo crucial del software que se va a comprar y por el costo final. En algunos casos (por ejemplo, software de PC de bajo costo), es menos costoso comprar y experimentar que realizar una larga evaluación de potenciales paquetes de software. En el análisis final, la decisión hacer/comprar se toma con base en las siguientes condiciones: 1) ¿La fecha de entrega del producto de software será más próxima que la del software que se desarrolle internamente? 2) ¿El costo de adquisición más el costo de personalización será menor que el costo que implica desarrollar el software internamente? 3) ¿El costo del apoyo exterior (por ejemplo, un contrato de mantenimiento) será menor que el costo del apoyo interno? Estas condiciones se aplican para cada una de las opciones de adquisición.

15 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

FIGURA 26.8
Árbol de decisiones para apoyar la decisión hacer/comprar



26.10.1 Creación de un árbol de decisión

¿Existe alguna manera sistemática de recorrer las opciones asociadas con la decisión hacer/comprar?

Los pasos recién descritos pueden aumentar usando técnicas estadísticas como el análisis de árbol de decisión.[16] Por ejemplo, la figura 26.8 muestra un árbol de decisión para un sistema X basado en software. En este caso, la organización de ingeniería de software puede: 1) construir el sistema X desde cero, 2) reusar componentes de experiencia parcial existentes para construir el sistema, 3) comprar un producto de software disponible y modificarlo para satisfacer las necesidades locales o 4) contratar el desarrollo del software a un proveedor externo.

Si el sistema se construirá desde cero, hay una probabilidad de 70 por ciento de que el trabajo será difícil. Al usar las técnicas de estimación estudiadas en este capítulo, el planificador de proyecto estima que un esfuerzo de desarrollo difícil costará US$450 000. Un esfuerzo de desarrollo "simple" se estima que cuesta US$380 000. El valor esperado por costo, calculado a lo largo de cualquier rama del árbol de decisión, es

$$\text{Costo esperado} = \Sigma \text{ (probabilidad de ruta)}_i \times \text{(costo de ruta estimado)}_i$$

donde *i* es la ruta del árbol de decisión. Para la ruta de construcción,

$$\text{Costo esperado}_{\text{construir}} = 0.30 \text{ (\$380K)} + 0.70 \text{ (\$450K)} = \$429\text{K}$$

Siguiendo otras rutas del árbol de decisión, también se muestran los costos proyectos para reúso, compra y contrato, bajo varias circunstancias. Los costos esperados para dichas rutas son

$$\text{Costo esperado}_{\text{reuso}} = 0.40 \text{ (\$275K)} + 0.60 \text{ [0.20 (\$310K)} + 0.80 \text{ (\$490K)]} = \$382\text{K}$$
$$\text{Costo esperado}_{\text{comprar}} = 0.70 \text{ (\$210K)} + 0.30 \text{ (\$400K)} = \$267\text{K}$$
$$\text{Costo esperado}_{\text{contratar}} = 0.60 \text{ (\$350K)} + 0.40 \text{ (\$500K)} = \$410\text{K}$$

Con base en la probabilidad y los costos proyectados que se anotaron en la figura 26.8, el costo esperado más bajo es la opción "comprar".

Sin embargo, es importante observar que deben considerarse muchos criterios, no sólo el costo, durante el proceso de toma de decisión. Disponibilidad, experiencia del desarrollador/

16 Una valiosa introducción al análisis del árbol de decisiones puede encontrarse en **http://en.wikipedia.org/wiki/Decision_tree**

proveedor/contratista, conformidad con los requisitos, "políticas" locales y la probabilidad de cambiar son sólo algunos de los criterios que pueden afectar la decisión final de construir, reusar, comprar o contratar.

26.10.2 *Outsourcing*

Tarde o temprano, toda compañía que desarrolla software de computadora plantea una pregunta fundamental: ¿hay alguna forma en la que puedan conseguirse el software y los sistemas necesarios a un precio más bajo? La respuesta a esta pregunta no es simple y las discusiones emocionales que ocurren en respuesta a la pregunta siempre conducen a una sola palabra: *outsourcing*.

Como concepto, el *outsourcing* (la subcontratación) es extremadamente simple. Las actividades de ingeniería de software se contratan a una tercera parte, que hace el trabajo a un costo más bajo y, con un poco de suerte, con mayor calidad. El trabajo de software realizado dentro de una compañía se reduce a una actividad de administración de contrato.[17]

Cita:

"Como regla, el outsourcing requiere una administración incluso más habilidosa que el desarrollo en casa."

Steve McConnell

La decisión por el *outsourcing* puede ser estratégica o táctica. En el nivel estratégico, los gerentes empresariales consideran si una porción significativa de todo el trabajo de software puede contratarse a otros. En el nivel táctico, un gerente de proyecto determina si parte o todo un proyecto puede lograrse mejor al subcontratar el trabajo de software.

Sin importar la amplitud del enfoque, la decisión de *outsourcing* con frecuencia es financiera. Un estudio detallado del análisis financiero para el *outsourcing* está más allá del ámbito de este libro y mejor se deja a otros (por ejemplo, [Min95]). Sin embargo, vale la pena una breve revisión de los pros y los contras de la decisión.

En el lado positivo, el ahorro en costo usualmente puede lograrse reduciendo el número de personal de software y las instalaciones (por ejemplo, computadoras, infraestructura) que lo apoyan. En el lado negativo, una compañía pierde cierto control sobre el software que necesita. Dado que el software es una tecnología que diferencia sus sistemas, servicios y productos, una compañía corre el riesgo de poner la fe de su competitividad en las manos de una tercera persona.

La tendencia hacia el *outsourcing* indudablemente continuará. La única forma de detenerla es reconocer que el trabajo de software es extremadamente competitivo en todos los niveles. La única manera de sobrevivir es volverse tan competitivo como los mismos proveedores de *outsourcing*.

CasaSegura

Outsourcing

La escena: Sala de juntas en CPI Corporation, cuando el proyecto comienza.

Participantes: Mal Golden, gerente ejecutivo, desarrollo de producto; Lee Warren, gerente de ingeniería; Joe Camalleri, VP ejecutivo, desarrollo empresarial; y Doug Miller, gerente de proyecto, ingeniería de software.

La conversación:

Joe: Estamos considerando la subcontratación de la porción de ingeniería de software del producto *CasaSegura*.

Doug (consternado): ¿Cuándo pasó esto?

Lee: Conseguimos una cotización de un desarrollador externo. Viene con 30 por ciento por abajo de lo que tu grupo pare-

17 El *outsourcing* puede verse de manera más general como cualquier actividad que conduce a la adquisición del software o de los componentes de software desde una fuente externa a la organización de ingeniería de software.

ce cree que costará. Vélo. [Extiende la cotización a Doug, quien la lee.]

Mal: Como sabes, Doug, intentamos mantener bajos los costos y 30 por ciento es 30 por ciento. Además, estas personas vienen muy recomendadas.

Doug (toma un respiro e intenta permanecer tranquilo): Me tomaron por sorpresa, pero antes de que tomen una decisión final, ¿puedo comentar algo?

Joe (afirma con la cabeza): Claro, adelante.

Doug: No hemos trabajado con esta compañía de *outsourcing*, ¿cierto?

Mal: Sí, pero...

Doug: Y señalan que cualquier cambio en las especificaciones se facturará con una tarifa adicional, ¿cierto?

Joe (frunce el ceño): Cierto, pero esperamos que las cosas serán razonablemente estables.

Doug: Mala suposición, Joe.

Joe: Bueno...

Doug: Es probable que liberemos nuevas versiones de este producto dentro de algunos años. Y es razonable suponer que el software proporcionará muchas de las nuevas características, ¿cierto?

[Todos afirman con la cabeza.]

Doug: ¿Alguna vez hemos coordinado un proyecto internacional?

Lee (observa preocupado): No, pero me dijeron...

Doug (intenta contener su enojo): Así que lo que me están diciendo es: 1) estamos a punto de trabajar con un proveedor desconocido, 2) los costos por hacer esto no son tan bajos como parecen, 3) nos comprometemos *de facto* para trabajar con ellos durante muchas liberaciones de producto, sin importar lo que hagan la primera vez y 4) vamos a aprender sobre la marcha lo relativo a un proyecto internacional.

[Todos permanecen en silencio.]

Doug: Chicos... Creo que éste es un error, y me gustaría que lo reconsideraran durante un día. Tenemos mucho más control si hacemos el trabajo en casa. Tenemos la experiencia y puedo garantizarles que no quiero que nos cueste mucho más... el riesgo será más bajo y sé que ustedes tienen aversión al riesgo, como yo.

Joe (frunce el ceño): Hiciste buenas observaciones, pero tú tienes un interés personal para mantener este proyecto en casa.

Doug: Es verdad, pero eso no cambia los hechos.

Joe (suspira): Está bien, consideremos esto durante un día o dos; lo pensaremos un poco más y nos reuniremos de nuevo para una decisión final. Doug, ¿puedo hablar contigo en privado?

Doug: Seguro... Realmente quiero estar seguro de que hacemos lo correcto.

26.11 Resumen

Un planificador de proyecto de software debe estimar tres cosas antes de comenzar un proyecto: cuánto tardará, cuánto esfuerzo se requerirá y cuántas personas se involucrarán. Además, el planificador debe predecir los recursos (hardware y software) que se requerirán y el riesgo involucrado.

El enunciado del ámbito ayuda al planificador a desarrollar estimaciones usando una o más técnicas que se clasifican en dos amplias categorías: descomposición y modelado empírico. Las técnicas de descomposición requieren una delineación de las principales funciones del software, seguidas por estimaciones de: 1) el número de LOC, 2) valores seleccionados dentro del dominio de información, 3) el número de casos de uso, 4) el número de persona-meses requeridos para implementar cada función o 5) el número de persona-meses requeridos para cada actividad de ingeniería de software. Las técnicas empíricas usan expresiones derivadas empíricamente para esfuerzo y tiempo a fin de predecir dichas cantidades de proyecto. Las herramientas automatizadas pueden usarse para implementar un modelo empírico específico.

Las estimaciones de proyecto precisas por lo general usan al menos dos de las tres técnicas recién anotadas. Al comparar y reconciliar las estimaciones desarrolladas usando diferentes técnicas, el planificador tiene más probabilidad de derivar una estimación precisa. La estimación de proyecto de software nunca puede ser una ciencia exacta, pero una combinación de buenos datos históricos y técnicas sistemáticas pueden mejorar la precisión de la estimación.

Problemas y puntos por evaluar

26.1. Suponga que usted es el gerente de proyecto de una compañía que construye software para robots caseros. Se le contrata a fin de construir el software para un robot que pode el césped para el propietario de

una casa. Escriba un enunciado para el ámbito que describa el software. Asegúrese de que su enunciado de ámbito esté acotado. Si no está familiarizado con los robots, haga un poco de investigación antes de comenzar a escribirlo. Además, establezca sus suposiciones acerca del hardware que se requerirá. Alternativa: Sustituya el robot podadora con otro problema que sea de su interés.

26.2. La complejidad del proyecto de software se analiza brevemente en la sección 26.1. Elabore una lista de características de software (por ejemplo, operación concurrente, salida gráfica) que afecten la complejidad de un proyecto. Priorice la lista.

26.3. El rendimiento es una importante consideración durante la planificación. Analice cómo puede interpretarse de manera diferente el rendimiento, dependiendo del área de aplicación del software.

26.4. Haga una descomposición funcional del software de robot que describió en el problema 26.1. Estime el tamaño de cada función en LOC. Si supone que su organización produce 450 LOC/pm con una tarifa de mano de obra sobrecargada de US$7 000 por persona-mes, estime el esfuerzo y el costo requeridos para construir el software, usando la técnica de estimación basada en LOC descrita en este capítulo.

26.5. Use el modelo COCOMO II para estimar el esfuerzo requerido para construir software para un simple ATM que produce 12 pantallas, 10 reportes y que requerirá aproximadamente 80 componentes de software. Suponga complejidad promedio y madurez desarrollador/entorno promedio. Use el modelo de composición de aplicación con puntos de objeto.

26.6. Use la ecuación de software para estimar el software del robot podadora. Suponga que la ecuación 26.4 es aplicable y que $P = 8\,000$.

26.7. Compare las estimaciones de esfuerzo inferidas en los problemas 26.4 y 26.6. ¿Cuál es la desviación estándar y cómo afecta esto a su grado de certidumbre acerca de la estimación?

26.8. Usando los resultados obtenidos en el problema 26.7, determine si es razonable esperar que el software pueda construirse dentro de los siguientes seis meses y cuántas personas tendrían que emplearse para realizar el trabajo.

26.9. Desarrolle un modelo de hoja de cálculo que implemente una o más de las técnicas de estimación descritas en este capítulo. De manera alternativa, adquiera uno o más modelos en línea para estimación de fuentes en la web.

26.10. Para un equipo de proyecto: desarrolle una herramienta de software que implemente cada una de las técnicas de estimación desarrolladas en este capítulo.

26.11. Parece extraño que las estimaciones de costo y calendario se desarrollen durante la planificación del proyecto de software, antes del análisis detallado de los requisitos de software o de realizar el diseño. ¿Por qué cree que se haga esto? ¿Existen circunstancias para no hacerlo?

26.12. Vuelva a calcular los valores esperados que se anotaron para el árbol de decisión en la figura 26.8 y suponga que cada rama tiene una probabilidad 50-50. ¿Esto cambiaría su decisión final?

Lecturas y fuentes de información adicionales

La mayoría de los libros de administración de proyectos de software contienen análisis acerca de la estimación del proyecto. The Project Management Institute (*PMBOK Guide,* PMI, 2001), Wysoki *et al*. (*Effective Project Management,* Wiley, 2000), Lewis (*Project Planning Scheduling and Control,* 3a. ed., McGraw-Hill, 2000), Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques,* 3a. ed., Wiley, 2000) y Phillips [Phi98] proporcionan útiles lineamientos de estimación.

McConnell (*Software Estimation: Demystifying the Black Art*, Microsoft Press, 2006) escribió una guía pragmática que proporciona valiosos lineamientos para todos aquellos que deban estimar el costo del software. Parthasarathy (*Practical Software Estimation*, Addison-Wesley, 2007) enfatiza los puntos de función como una métrica de estimación. Laird y Brennan (*Software Measurement and Estimation: A Practical Approach*, Wiley-IEEE Computer Society Press, 2006) aborda las mediciones y su uso en la estimación del software. Pfleeger (*Software Cost Estimation and Sizing Methods, Issues, and Guidelines*, RAND Corporation, 2005) desarrolló un manual abreviado que aborda muchos fundamentos de estimación. Jones (*Estimating Software Costs*, 2a. ed., McGraw-Hill, 2007) escribió uno de los tratamientos más exhaustivos de los modelos y los datos que son aplicables a la estimación del software en todo dominio de aplicación. Coombs (*IT Project Estimation*, Cambridge University Press, 2002) y Roetzheim y Beasley (*Software Project Cost and Schedule Estimating: Best*

Practices, Prentice-Hall, 1997) presentan muchos modelos útiles y sugieren lineamientos, paso a paso, para generar las mejores estimaciones posibles.

En internet, está disponible una gran variedad de fuentes de información acerca de la estimación del software. Una lista actualizada de referencias en la World Wide Web que son relevantes para la estimación del software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

27 CALENDARIZACIÓN DEL PROYECTO

CONCEPTOS CLAVE

cronogramas 629
distribución de esfuerzo 625
distribución del trabajo 629
personal y esfuerzo 624
principios de calendarización para webapps 633
red de tareas 628
ruta crítica 622
seguimiento 631
time-boxing 632
valor ganado 635

A finales de los años sesenta del siglo pasado, se eligió a un entusiasta joven ingeniero para que "escribiera" un programa de computadora para una aplicación de fabricación automatizada. La razón para su selección fue simple. Él era la única persona en su grupo técnico que asistió a un seminario de programación de computadoras. Sabía los pros y los contras del lenguaje ensamblador y de FORTRAN, pero nada conocía acerca de ingeniería de software incluso menos acerca de la calendarización y el seguimiento de proyectos.

Su jefe le dio los manuales apropiados y una descripción verbal de lo que tenía que hacer. Se le informó que el proyecto debía estar terminado en dos meses.

Leyó los manuales, consideró su enfoque y comenzó a escribir el código. Después de dos semanas, el jefe lo llamó a su oficina y le preguntó sobre cómo iban las cosas.

"Realmente grandiosas", dijo el ingeniero con entusiasmo juvenil. "Esto fue mucho más simple de lo que pensé. Probablemente tenga ya un avance de 75 por ciento".

El jefe sonrió y alentó al joven ingeniero a seguir con el buen trabajo. Planearon reunirse de nuevo en una semana.

Una semana después, el jefe llamó al ingeniero a su oficina y le preguntó: "¿Dónde estamos?"

"Todo está bien", dijo el joven, "pero encontré algunos tropiezos. Los allanaré y pronto estaré de vuelta en el camino".

"¿Qué te parece la fecha límite?", preguntó el jefe.

"No hay problema", dijo el ingeniero. "tengo un avance de cerca de 90 por ciento".

UNA MIRADA RÁPIDA

¿Qué es? Ya seleccionó un modelo de proceso adecuado, identificó las tareas de ingeniería de software que deben realizarse, estimó la cantidad de trabajo y el número de personas, conoce la fecha límite e incluso consideró los riesgos. Ahora es momento de unir los puntos. Es decir, tiene que crear una red de tareas de ingeniería de software que le permitirán concluir el trabajo a tiempo. Una vez creada la red, tiene que asignar responsables para cada tarea, asegurarse de que se realizan todas ellas y adaptar la red conforme los riesgos que habrá cuando se convierta en realidad. En pocas palabras, eso es la calendarización y el seguimiento del proyecto de software.

¿Quién lo hace? En el nivel del proyecto, los gerentes de proyecto de software que usan la información solicitada a los ingenieros de software. En lo individual, los mismos ingenieros de software.

¿Por qué es importante? Para construir un sistema complejo, muchas tareas de ingeniería de software ocurren en paralelo, y el resultado del trabajo realizado durante una tarea puede tener un profundo efecto sobre el trabajo que se va a realizar en otra tarea. Estas interdependencias son muy difíciles de comprender sin un calendario. También es virtualmente imposible valorar el avance en un proyecto de software regular o grande sin un calendario detallado.

¿Cuáles son los pasos? Las tareas de ingeniería de software dictadas por el modelo de proceso del software se refinan en función de la funcionalidad que se va a construir. Se asigna esfuerzo y duración determinados a cada tarea y se crea una red de tareas (también llamada "red de actividad") de manera que permita al equipo de software alcanzar la fecha de entrega establecida.

¿Cuál es el producto final? Se produce el calendario del proyecto y la información relacionada.

¿Cómo me aseguro de que lo hice bien? La calendarización adecuada requiere que: 1) todas las tareas aparezcan en la red, 2) el esfuerzo y la calendarización se asignen de manera inteligente a cada tarea, 3) las interdependencias entre tareas se indiquen de manera adecuada, 4) se asignen los recursos para el trabajo que se va a realizar y 5) se proporcionen hitos cercanamente espaciados de modo que pueda darse seguimiento al progreso.

Si el lector ha trabajado en el mundo del software durante algunos años, puede terminar la historia. No es de sorprender que el joven ingeniero[1] se quedara en 90 por ciento de avance durante todo el proyecto y terminara (con la ayuda de otros) sólo un mes más tarde.

Esta historia se ha repetido decenas de miles de veces entre los desarrolladores de software durante las pasadas cinco décadas. La gran pregunta es por qué.

27.1 Conceptos básicos

Aunque existen muchas razones por las que el software se entrega tardíamente, la mayoría pueden rastrearse en una o más de las siguientes causas fundamentales:

- Una fecha límite irreal establecida por alguien externo al equipo de software y que fuerza a los gerentes y profesionales.
- Requerimientos del cliente variables que no se reflejan en cambios del calendario.
- Una honesta subestimación de la cantidad de esfuerzo y/o número de recursos que se requerirán para hacer el trabajo.
- Riesgos predecibles y/o impredecibles que no se consideraron cuando comenzó el proyecto.
- Dificultades humanas que no podían preverse por anticipado.
- Falta de comunicación entre el personal del proyecto que da como resultado demoras.
- Falta de comunicación entre el equipo de trabajo que se traduce en retrasos.
- Una falla por parte de la administración del proyecto para reconocer que el proyecto tiene retrasos en el calendario y una falta de acción para corregir el problema.

Cita:

"Los calendarios excesivos o irracionales son probablemente la influencia más destructiva en todo el software."

Capers Jones

Las fechas límite agresivas (léase "irreales") son un hecho de la vida en el negocio del software. En ocasiones, tales fechas límite se demandan por razones legítimas, desde el punto de vista de la persona que las establece. Pero el sentido común dice que la legitimidad también debe percibirla el personal que hace el trabajo.

Napoleón dijo una vez: "Cualquier comandante que se comprometa a llevar a cabo un plan que considere defectuoso está equivocado; debe plantear sus razones, insistir en que se cambie el plan y finalmente ofrecer formalmente su renuncia en lugar de ser el instrumento de la derrota de su ejército". Éstas son duras palabras que muchos gerentes de proyecto de software deberían ponderar.

Las actividades de estimación estudiadas en el capítulo 26 y las técnicas de calendarización descritas en este capítulo con frecuencia se implementan bajo la restricción de una fecha límite definida. Si las mejores estimaciones indican que la fecha límite es irreal, un gerente de proyecto competente debe "proteger a su equipo contra la presión excesiva [calendario]... [y] devolver la presión a quienes la originaron" [Pag85].

Cita:

"Me encantan las fechas límite. Me gusta el zumbido que producen cuando pasan volando."

Douglas Adams

Para ilustrar lo anterior, suponga que a su equipo de software se le pide construir un controlador en tiempo real para un instrumento de diagnóstico médico que debe introducirse en el mercado en nueve meses. Después de realizar la estimación y el análisis de riesgo cuidadosamente (capítulo 28), llega a la conclusión de que el software, como se solicitó, requerirá 14 meses para su creación con el personal que se tiene disponible. ¿Cómo procedería?

Es irreal marchar hacia la oficina del cliente (en este caso el probable cliente es mercadotecnia/ventas) y demandar que se cambie la fecha de entrega. Las presiones del mercado externo dictaron la fecha y el producto debe liberarse. Es igualmente temerario rechazar el compromiso de realizar el trabajo (desde el punto de vista profesional). De modo que, ¿qué hacer? Ante esta situación, los autores recomiendan los siguientes pasos:

1 En caso de que el lector se lo pregunte, esta historia es autobiográfica.

1. Realice una estimación detallada, usando datos históricos de proyectos anteriores. Determine el esfuerzo y la duración estimados para el proyecto.
2. Con el modelo de proceso incremental (capítulo 2), desarrolle una estrategia de ingeniería de software que entregue funcionalidad crucial hacia la fecha límite impuesta, pero desarrolle otra estrategia para otra entrega de software con funcionalidad hasta más tarde. Documente el plan.

? ¿Qué debe hacer cuando los gerentes demandan una fecha límite que es imposible de cumplir?

3. Reúnase con el cliente y (con la estimación detallada) explique por qué la fecha límite es irreal. Asegúrese de señalar que todas las estimaciones se basan en el rendimiento de proyectos anteriores. También asegúrese de indicar el porcentaje de mejora que se requeriría para lograr cumplir en la fecha límite, como se plantea originalmente.[2] El siguiente comentario puede ser apropiado como respuesta:

   Creo que podemos tener problemas con la fecha de entrega para el software controlador XYZ. A cada uno de ustedes le entrego una distribución abreviada de las tasas de desarrollo para proyectos de software anteriores y una estimación que se hizo de maneras distintas. Observarán que supuse una mejora de 20 por ciento en tasas de desarrollo anteriores, pero todavía tenemos una fecha de entrega que es de 14 meses en lugar de nueve.

4. Ofrezca la estrategia de desarrollo incremental como una alternativa:

   Tenemos algunas opciones, y me gustaría que tomaran una decisión con base en ellas. Primero, podemos aumentar el presupuesto y conseguir recursos adicionales, de modo que podremos tener listo este trabajo en nueve meses. Pero entiendo que esto aumentará el riesgo de empobrecer la calidad debido a las fechas tan apretadas.[3] Segundo, podemos remover algunas funciones y capacidades del software que se solicitan. Esto hará que la versión preliminar del producto sea un poco menos funcional, pero puede anunciarse toda la funcionalidad y entregarla durante el periodo de 14 meses. Tercero, podemos prescindir de la realidad y querer que el proyecto esté completo en nueve meses. Acabaremos sin tener algo que pueda entregarse al cliente. La tercera opción, y creo que estarán de acuerdo, es inaceptable. La historia pasada y nuestras mejores estimaciones dicen que es irreal y que representa una receta para el desastre.

Habrá algunos gruñidos, pero si se presenta una estimación sólida con base en buenos datos históricos, es probable que se elijan las versiones negociadas de las opciones 1 o 2. La fecha límite irreal se evapora.

27.2 Calendarización del proyecto

Alguna vez se le preguntó a Fred Brooks cómo es que los proyectos de software se atrasan en su calendario. Su respuesta fue tan simple como profunda: "un día a la vez".

La realidad de un proyecto técnico (ya sea que implique construir una hidroeléctrica o desarrollar un sistema operativo) es que cientos de pequeñas tareas deben ocurrir para lograr una meta más grande. Algunas de esas tareas yacen fuera de la corriente principal y pueden completarse sin preocuparse acerca de su impacto sobre la fecha de conclusión del proyecto. Otras se encuentran en la "ruta crítica". Si las tareas "críticas" se retrasan en el calendario, la fecha de conclusión de todo el proyecto se pone en riesgo.

CONSEJO

Las tareas requeridas para lograr el objetivo de un gerente de proyecto no deben realizarse manualmente. Existen muchas excelentes herramientas de calendarización. Úselas.

Como gerente de proyecto, su objetivo es definir todas las tareas del proyecto, construir una red que muestre sus interdependencias, identificar las que son cruciales dentro de la red y luego monitorear su progreso para asegurar que la demora se reconoce "en el momento". Para lograr

2 Si la mejora requerida es de 10 a 25 por ciento, en realidad puede tener listo el trabajo. Pero, muy probablemente, la mejora requerida en el rendimiento del equipo será mayor a 50 por ciento. Ésta es una expectativa irreal.

3 También puede argumentar que aumentar el número de personas no reduce el tiempo de manera proporcional.

esto, debe tener un calendario que se haya definido en un grado de resolución que permita monitorear el progreso y controlar el proyecto.

La *calendarización del proyecto de software* es una acción que distribuye el esfuerzo estimado a través de la duración planificada del proyecto, asignando el esfuerzo a tareas específicas de ingeniería del software. Sin embargo, es importante observar que el calendario evoluciona con el tiempo. Durante las primeras etapas de la planificación del proyecto se desarrolla un calendario macroscópico. Este tipo de calendario identifica las principales actividades de marco conceptual de proceso y las funciones de producto a las cuales se aplican. Conforme el proyecto avanza, cada entrada en el calendario macroscópico se desglosa en un calendario detallado. Aquí, acciones y tareas de software específicas (requeridas para lograr una actividad) se identifican y calendarizan.

Cita:

"La calendarización demasiado optimista no da como resultado calendarios reales más cortos; da como resultado unos más largos."

Steve McConnell

La calendarización para proyectos de ingeniería de software puede verse desde dos perspectivas más bien diferentes. En la primera, ya se estableció una fecha final (e irrevocable) para liberar un sistema basado en computadora. La organización de software se restringe para distribuir el esfuerzo dentro del marco temporal prescrito. La segunda visión de la calendarización del software supone que se han discutido límites cronológicos burdos, pero que la fecha final la establece la organización de ingeniería del software. El esfuerzo se distribuye para hacer mejor uso de los recursos y se define una fecha final después de un cuidadoso análisis del software. Por desgracia, la primera situación se encuentra con mucha más frecuencia que la segunda.

27.2.1 Principios básicos

Como otras áreas de la ingeniería de software, algunos principios básicos guían la calendarización del proyecto de software:

PUNTO CLAVE

Cuando desarrolle un calendario, divida el trabajo, anote las interdependencias entre tareas, asigne esfuerzo y tiempo a cada tarea, y defina responsabilidades, resultados e hitos.

Compartimentalización. El proyecto debe compartimentalizarse en algunas actividades y tareas manejables. Para lograr la compartimentalización se desglosan tanto el producto como el proceso.

Interdependencia. Debe determinarse la interdependencia de cada actividad o tarea compartimentalizada. Algunas tareas deben sucederse en secuencia, mientras que otras pueden ocurrir en paralelo. Algunas actividades no pueden comenzar hasta que esté disponible el producto operativo producido por otra. Otras pueden realizarse de manera independiente.

Asignación de tiempo. A cada tarea por calendarizar debe asignársele cierto número de unidades de trabajo (por ejemplo, persona-días de esfuerzo). Además, a cada tarea debe asignársele una fecha de comienzo y una de conclusión, en función de las interdependencias y de si el trabajo se realizará sobre una base de tiempo completo o parcial.

Validación de esfuerzo. Todo proyecto tiene un número definido de personas en el equipo de software. Conforme ocurre la asignación de tiempo, debe asegurarse de que, en un momento determinado, no se ha calendarizado más que el número de personal asignado. Por ejemplo, considere un proyecto que tiene tres ingenieros de software asignados (tres personas-días están disponibles por día de esfuerzo asignado[4]). En un día determinado deben lograrse siete tareas concurrentes. Cada tarea requiere 0.50 persona-días de esfuerzo. Hay más esfuerzo por asignar que personas disponibles para hacer el trabajo.

Responsabilidades definidas. Cada tarea por calendarizar debe asignarse a un miembro de equipo específico.

4 En realidad, están disponibles menos de tres personas-días de esfuerzo debido a juntas no relacionadas, enfermedad, vacaciones y varias otras razones. Sin embargo, para nuestros propósitos, se supone 100 por ciento de disponibilidad.

Resultados definidos. Cada tarea que se calendarice debe tener un resultado definido. Para proyectos de software, el resultado usualmente es un producto operativo (por ejemplo, el diseño de un componente) o una parte de un producto operativo. Los productos operativos con frecuencia se combinan con productos operativos entregables.

Hitos definidos. Cada tarea o grupo de tareas debe asociarse con un hito del proyecto. Un hito se logra cuando uno o más productos operativos se revisan en su calidad (capítulo 15) y se aprueban.

Cada uno de estos principios se aplica conforme evoluciona el calendario del proyecto.

27.2.2 Relación entre personal y esfuerzo

En un pequeño proyecto de desarrollo de software, una sola persona puede analizar requerimientos, elaborar diseño, generar código y realizar pruebas. Conforme el tamaño de un proyecto aumenta, más personas deben involucrarse. (¡Rara vez uno puede darse el lujo de abordar un esfuerzo de 10 persona-años con una persona que trabaje durante 10 años!).

Si debe agregar personal a un proyecto retrasado, asegúrese de que se les asigna trabajo que esté muy compartimentalizado.

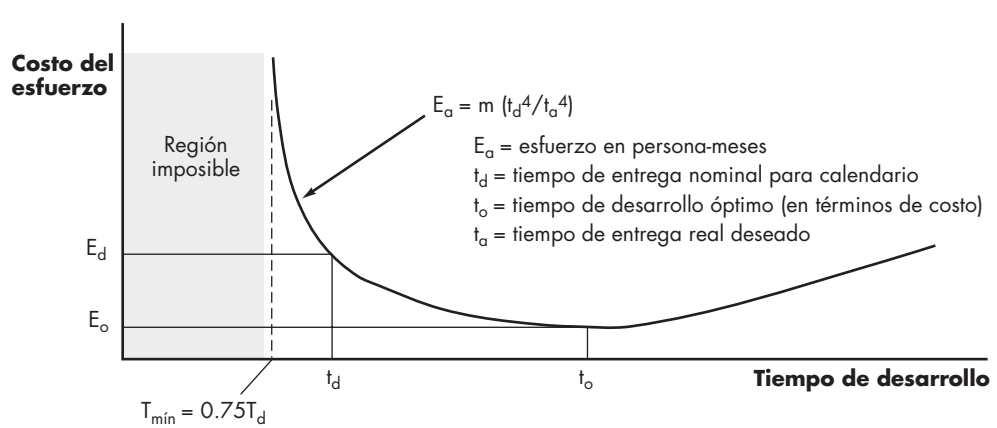
Existe un mito común que todavía creen muchos gerentes responsables de proyectos de desarrollo de software: "Si nos atrasamos en el calendario, siempre podemos agregar más programadores y ponernos al corriente en el proyecto más adelante". Por desgracia, agregar personal tardíamente en un proyecto con frecuencia tiene efectos perturbadores sobre el proyecto, lo que hace que el calendario se deteriore todavía más. Las personas que se agregan deben aprender el sistema y las que les enseñan son las mismas personas que hacían el trabajo. Mientras enseñan no trabajan y el proyecto se atrasa aún más.

Además del tiempo que tardan en aprender el sistema, más personas aumentan el número de rutas de comunicación y la complejidad de la comunicación a lo largo de un proyecto. Aunque la comunicación es absolutamente esencial para el desarrollo de software exitoso, toda nueva ruta de comunicación requiere esfuerzo adicional y, por tanto, tiempo adicional.

Con los años, los datos empíricos y el análisis teórico han demostrado que los calendarios de proyecto son elásticos. Es decir: es posible comprimir en cierta medida la fecha de conclusión de un proyecto deseado (al agregar recursos adicionales). También lo es extender una fecha de conclusión (al reducir el número de recursos).

La *curva Putnam-Norden-Rayleigh (PNR)*[5] proporciona un indicio de la relación entre esfuerzo aplicado y tiempo de entrega para un proyecto de software. En la figura 27.1 se muestra una

FIGURA 27.1
Relación entre esfuerzo y tiempo de entrega



5 La investigación original puede encontrarse en [Nor70] y [Put78].

Si la entrega puede retrasarse, la curva PNR indica que los costos del proyecto pueden reducirse sustancialmente.

versión de la curva, que representa el esfuerzo del proyecto como función del tiempo de entrega. La curva indica un valor mínimo t_o que señala el costo mínimo para entrega (es decir, el tiempo de entrega que resultará en el menor esfuerzo empleado). Conforme avanza hacia la izquierda de t_o (es decir, conforme se intenta acelerar la entrega), la curva se eleva de manera no lineal.

Como ejemplo, suponga que un equipo de proyecto estimó que se requerirá un nivel de esfuerzo E_d para lograr un tiempo de entrega nominal t_d que es óptimo en términos de calendario y recursos disponibles. Aunque es posible acelerar la entrega, la curva se eleva de manera muy pronunciada hacia la izquierda de t_d. De hecho, la curva PNR indica que el tiempo de entrega del proyecto no puede comprimirse mucho más allá de $0.75t_d$. Si se intenta mayor compresión, el proyecto se mueve hacia la "región imposible" y el riesgo de fracaso se vuelve muy alto. La curva PNR también indica la opción de entrega de costo más bajo, $t_o = 2t_d$. Lo que se observa aquí es que la entrega demorada del proyecto puede reducir los costos significativamente. Desde luego, esto debe ponderarse contra el costo empresarial asociado con la demora.

La ecuación de software [Put92] que se introdujo en el capítulo 26 se infirió a partir de la curva PNR y demuestra la relación enormemente no lineal entre el tiempo cronológico para completar un proyecto y el esfuerzo humano que se aplica al mismo. El número de líneas de código entregadas (enunciados fuente), L, se relaciona con el esfuerzo y el tiempo de desarrollo mediante la ecuación:

$$L = P \times E^{1/3} t^{4/3}$$

donde E es el esfuerzo de desarrollo en persona-meses, P es un parámetro de productividad que refleja varios factores que conducen a trabajo de ingeniería de software de alta calidad (por lo general valores de P que varían entre 2 000 y 12 000) y t es la duración del proyecto en meses calendario.

Conforme la fecha límite del proyecto se acerca, se llega a un punto donde el trabajo no puede completarse en calendario, sin importar el número de personas que hagan el trabajo. Enfrente la realidad y defina una nueva fecha de entrega.

Al reordenar esta ecuación de software puede llegarse a una expresión para el esfuerzo de desarrollo E:

$$E = \frac{L^3}{P^3 t^4} \tag{27.1}$$

donde E es el esfuerzo empleado (en persona-años) durante todo el ciclo de vida para el desarrollo y mantenimiento del software y t es el tiempo de desarrollo en años. La ecuación para esfuerzo de desarrollo puede relacionarse con el costo de desarrollo mediante la inclusión de un factor de tarifa de mano de obra sobrecargada ($/persona-año).

Esto conduce a algunos resultados interesantes. Considere un complejo proyecto de software en tiempo real estimado en 33 000 LOC, 12 persona-años de esfuerzo. Si ocho personas se asignan al equipo del proyecto, éste puede completarse en aproximadamente 1.3 años. Sin embargo, la fecha final se extiende a 1.75 años; la naturaleza enormemente no lineal del modelo descrito en la ecuación 27.1 produce:

$$E = \frac{L^3}{P^3 t^4} \sim 3.8 \text{ persona-años}$$

Esto implica que, al extender la fecha final por seis meses, ¡es posible reducir el número de personas de ocho a cuatro! La validez de tales resultados está abierta a debate, pero la implicación es clara: pueden obtenerse beneficios usando menos personal durante un lapso un poco más largo para lograr el mismo objetivo.

27.2.3 Distribución de esfuerzo

Cada una de las técnicas de estimación del proyecto de software que se estudian en el capítulo 26 conducen a estimaciones de unidades de trabajo (por ejemplo, persona-meses) requeridas para completar el desarrollo de software. Una distribución de esfuerzo recomendada a través

del proceso de software con frecuencia se conoce como la *regla 40-20-40*. Cuarenta por ciento de todo el esfuerzo se asigna a análisis frontal y diseño. Un porcentaje similar se aplica a pruebas traseras. De ahí se infiere correctamente que la codificación pierde el énfasis (20 por ciento de esfuerzo).

? ¿Cómo debe distribuirse el esfuerzo a través del flujo de trabajo del proceso de software?

Esta distribución de esfuerzo debe usarse solamente como guía.[6] Las características de cada proyecto dictan la distribución del esfuerzo. El trabajo empleado en la planificación del proyecto rara vez representa más de 2 a 3 por ciento de esfuerzo, a menos que el plan comprometa a una organización a realizar más gastos con alto riesgo. La comunicación con el cliente y el análisis de requerimientos pueden comprender de 10 a 25 por ciento del esfuerzo del proyecto. El esfuerzo que se emplea en análisis o creación de prototipos debe aumentar en proporción directa con el tamaño y la complejidad del proyecto. Por lo general, al diseño de software se aplica un rango de 20 a 25 por ciento del esfuerzo. También debe considerarse el tiempo que se emplea para revisión del diseño y su posterior iteración.

Debido al esfuerzo aplicado al diseño de software, el código debe seguir relativamente con poca dificultad. Es posible lograr un esfuerzo global de 15 a 20 por ciento. Las pruebas y la posterior depuración pueden representar de 30 a 40 por ciento del esfuerzo de desarrollo del software. Lo crucial del software con frecuencia dicta la cantidad de pruebas que se requieren. Si el software se clasifica humanamente (es decir, si errores en el software pueden resultar en pérdida de la vida), incluso son usuales porcentajes más altos.

27.3 Definición de un conjunto de tareas para el proyecto de software

Sin importar el modelo de proceso que se elija, el trabajo que realiza un equipo de software se logra a través de un conjunto de tareas que permiten definir, desarrollar y, a final de cuentas, apoyar el software de computadora. Ningún conjunto de tareas es adecuado para todos los proyectos. Un conjunto de tareas adecuado para un sistema grande y complejo probablemente se percibirá como excesivo para un producto de software pequeño y relativamente simple. En consecuencia, un proceso de software efectivo debe definir una colección de conjuntos de tareas, cada una diseñada para satisfacer las necesidades de diferentes tipos de proyectos.

Como se anotó en el capítulo 2, un conjunto de tareas es una colección de tareas de trabajo de ingeniería del software, hitos, productos operativos y filtros de aseguramiento de la calidad que deben lograrse para completar un proyecto particular. El conjunto de tareas debe proporcionar suficiente disciplina a fin de lograr software de alta calidad. Pero, al mismo tiempo, no debe abrumar al equipo del proyecto con trabajo innecesario.

Con la finalidad de desarrollar un calendario del proyecto, en la línea de tiempo del proyecto debe distribuirse un conjunto de tareas. El conjunto de tareas variará dependiendo del tipo de proyecto y el grado de rigor con el que el equipo de software decide hacer su trabajo. Aunque es difícil desarrollar una taxonomía exhaustiva de los tipos de proyecto de software, la mayoría de las organizaciones de software encuentran los siguientes proyectos:

WebRef

Para auxiliar en la definición de los conjuntos de tareas para varios proyectos de software, se desarrolló un modelo de proceso adaptable (APM, por sus siglas en inglés). En **www.rspa.com/apm** puede encontrar una descripción completa del MPA en inglés.

1. *Proyectos de desarrollo de concepto* que inician para explorar algún concepto empresarial nuevo o la aplicación de alguna nueva tecnología.
2. Los *proyectos de desarrollo de nueva aplicación* que se realizan como consecuencia de la solicitud de un cliente específico.

6 En la actualidad, la regla 40-20-40 está bajo ataque. Algunos creen que más de 40 por ciento de esfuerzo global debe emplearse durante análisis y diseño. Por otro lado, quienes proponen el desarrollo ágil (capítulo 3) argumentan que debe emplearse menos tiempo "frontal" y que un equipo debe avanzar rápidamente hacia la construcción.

3. *Proyectos de mejora de aplicación* que ocurren cuando un software existente experimenta grandes modificaciones a funciones, rendimiento o interfaces que son observables por el usuario final.
4. *Proyectos de mantenimiento de aplicación* que corrigen, adaptan o extienden el software existente de maneras que pueden no ser inmediatamente obvias para el usuario final.
5. *Proyectos de reingeniería* que se llevan a cabo con la intención de reconstruir un sistema existente (heredado) en todo o en parte.

Incluso dentro de un solo tipo de proyecto, muchos factores influyen en el conjunto de tareas por elegir. Los factores incluyen [Pre05]: tamaño del proyecto, número de usuarios potenciales, vitalidad de la misión, longevidad de la aplicación, estabilidad de requerimientos, facilidad de comunicación cliente/desarrollador, madurez de tecnología aplicable, restricciones de rendimiento, características incrustadas y no incrustadas, personal del proyecto y factores de reingeniería. Cuando se combinan, dichos factores proporcionan un indicio del *grado de rigor* con el que debe aplicarse el proceso de software.

27.3.1 Un ejemplo de conjunto de tareas

Los proyectos de desarrollo de concepto inician cuando debe explorarse el potencial para alguna nueva tecnología. No hay certeza de que la tecnología será aplicable, pero un cliente (por ejemplo, mercadotecnia) cree que existen beneficios potenciales. Los proyectos de desarrollo de concepto se abordan al aplicar las siguientes acciones:

1.1 El **ámbito del concepto** determina el ámbito global del proyecto.

1.2 La **planificación preliminar del concepto** establece la habilidad de la organización para llevar a cabo el trabajo implicado por el ámbito del proyecto.

1.3 La **valoración del riesgo tecnológico** evalúa el riesgo asociado con la tecnología que se va a implementar como parte del ámbito del proyecto.

1.4 La **prueba del concepto** demuestra la viabilidad de una nueva tecnología en el contexto del software.

1.5 La **implementación del concepto** constituye la representación del concepto de manera que pueda revisarse por parte de un cliente y se usa con propósitos de "mercadotecnia" cuando debe venderse un concepto a otros clientes o gerentes.

1.6 La **reacción del cliente** al concepto solicita retroalimentación acerca de un nuevo concepto tecnológico y se dirige a aplicaciones de cliente específicas.

Una rápida exploración a dichas acciones debe producir pocas sorpresas. De hecho, el flujo de ingeniería de software para proyectos de desarrollo de concepto (y también para todos los otros tipos de proyectos) no es mucho más que sentido común.

27.3.2 Refinamiento de acciones de ingeniería del software

Las acciones de ingeniería de software descritas en la sección anterior pueden usarse para definir un calendario macroscópico para un proyecto. No obstante, éste debe refinarse para crear un calendario de proyecto detallado. El refinamiento comienza tomando cada acción y descomponiéndola en un conjunto de tareas (con productos operativos e hitos relacionados).

Como ejemplo de descomposición de tarea, considere la acción 1.1, ámbito del concepto. El refinamiento de las tareas puede lograrse usando un formato de bosquejo, pero en este libro se usará un enfoque de lenguaje de diseño de proceso para ilustrar el flujo de la acción de determinación del ámbito del concepto:

Task definition: Acción 1.1 Ámbito del concepto
1.1.1 Identificar necesidad, beneficios y clientes potenciales;
1.1.2 Definir salida/control deseado y eventos de entrada que impulsen la aplicación;
Begin Task 1.1.2
1.1.2.1 RT: Revisar descripción escrita de necesidad[7]
1.1.2.2 Inferir una lista de salidas/entradas visibles para el cliente
1.1.2.3 RT: revisar salidas/entradas con cliente y revisar según se requiera; endtask Task 1.1.2
1.1.3 Definir la funcionalidad/comportamiento para cada función principal;
Begin Task 1.1.3
1.1.3.1 RT: Revisar salida y entrada de objetos de datos inferidos en la tarea 1.1.2;
1.1.3.2 Inferir un modelo de funciones/comportamientos;
1.1.3.3 RT: Revisar funciones/comportamientos con cliente y revisar según se requiera;
endtask Task 1.1.3
1.1.4 Aislar aquellos elementos de la tecnología que se va a implementar en el software;
1.1.5 Investigar disponibilidad de software existente;
1.1.6 Definir factibilidad técnica;
1.1.7 Hacer estimación rápida de tamaño;
1.1.8 Crear una definición de ámbito;
endtask definition: Acción 1.1

Las tareas y subtareas anotadas en el refinamiento del lenguaje de diseño de proceso forman la base de un calendario detallado para la determinación del ámbito del concepto.

27.4 Definición de una red de tareas

PUNTO CLAVE
La red de tareas es un mecanismo útil para mostrar las dependencias intertarea y determinar la ruta crítica.

Las tareas y subtareas individuales tienen interdependencias en función de su secuencia. Además, cuando más de una persona está involucrada en un proyecto de ingeniería del software, es probable que las actividades y tareas de desarrollo se realicen en paralelo. Cuando esto ocurre, las tareas concurrentes deben coordinarse de modo que se completen en el momento en el que las tareas posteriores requieran sus productos operativos.

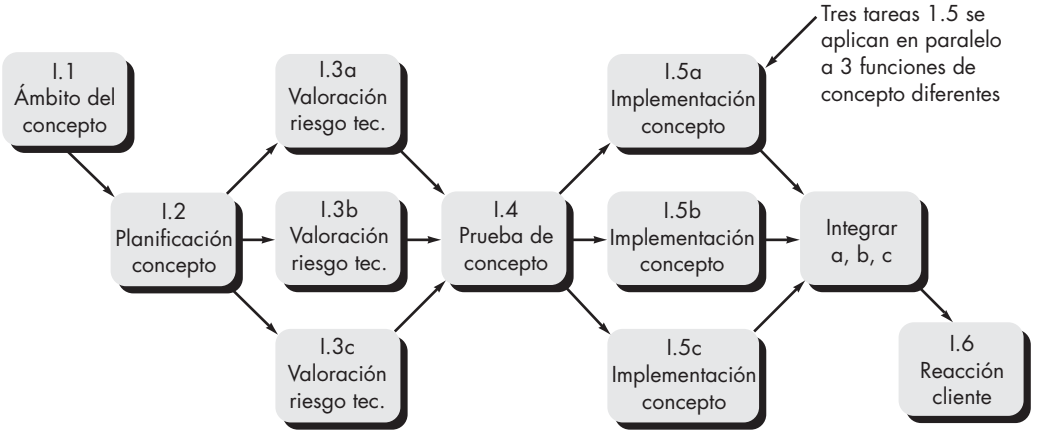
Una *red de tareas*, también llamada *red de actividad*, es una representación gráfica del flujo de tareas para un proyecto. En ocasiones se usa como el mecanismo mediante el cual la secuencia y las dependencias de tareas se integran en una herramienta automatizada de calendarización de proyecto. En su forma más simple (usada cuando se crea un calendario macroscópico), la red de tareas muestra las principales acciones de la ingeniería del software. La figura 27.2 presenta una red de tareas esquemática para un proyecto de desarrollo de concepto.

La naturaleza concurrente de las acciones de ingeniería de software conduce a algunos importantes requerimientos de calendarización. Puesto que las tareas paralelas ocurren de manera asíncrona, debe determinar las dependencias intertarea para asegurar el progreso continuo hacia la conclusión. Además, debe estar al tanto de aquellas tareas que se encuentren en la *ruta crítica*, es decir, aquellas que deben concluirse conforme al calendario si el proyecto como un todo debe completarse de acuerdo con ese calendario. Estos temas se estudian con más detalle más adelante, en este capítulo.

Es importante observar que la red de tareas que se muestra en la figura 27.2 es macroscópica. En una red de tareas detallada (un precursor de un calendario detallado), cada acción que se muestra en la figura se expandirá. Por ejemplo, la tarea 1.1 se expandirá para mostrar todas las tareas detalladas en el refinamiento de las acciones 1.1 que se muestran en la sección 27.3.2.

7 RT indica que debe realizarse una revisión técnica (capítulo 15).

FIGURA 27.2
Red de tareas para desarrollo de concepto



27.5 CALENDARIZACIÓN

Cita:

"Todo lo que debemos decidir es qué hacer con el tiempo que nos dan."

Gandalf en *El señor de los anillos: La comunidad del anillo*

La calendarización de un proyecto de software no difiere enormemente de la de cualquier esfuerzo de ingeniería multitarea. Por tanto, las herramientas y técnicas generalizadas de calendarización de proyecto pueden aplicarse con pocas modificaciones para proyectos de software.

La *evaluación del programa y la técnica de revisión* (PERT, por sus siglas en inglés) y el *método de ruta crítica* (CPM, por sus siglas en inglés) son dos métodos de calendarización de proyecto que pueden aplicarse en el desarrollo de software. Ambas técnicas se impulsan mediante información ya desarrollada en actividades de planificación de proyectos anteriores: estimaciones de esfuerzo, una descomposición de la función del producto, la selección del modelo de proceso y el conjunto de tareas adecuadas, así como la descomposición de las tareas que se seleccionan.

Las interdependencias entre tareas pueden definirse usando una red de tareas. Las tareas, en ocasiones llamadas *estructura de distribución del trabajo* del proyecto (EDT), se definen para el producto como un todo o para funciones individuales.

Tanto la PERT como el CPM proporcionan herramientas cuantitativas que permiten: 1) determinar la ruta crítica (la cadena de tareas que determina la duración del proyecto), 2) establecer estimaciones de tiempo "más probables" para tareas individuales aplicando modelos estadísticos y 3) calcular "tiempos frontera" que definen una "ventana" de tiempo para una tarea particular.

27.5.1 Cronogramas

Cuando se crea un calendario de proyecto de software, se comienza con un conjunto de tareas (la estructura de distribución del trabajo). Si se usan herramientas automatizadas, la distribución del trabajo se ingresa como una red o esbozo de tareas. Luego se ingresan esfuerzo, duración y fecha de inicio para cada tarea. Además, las tareas pueden asignarse a individuos específicos.

PUNTO CLAVE
Un cronograma le permite determinar qué tareas se realizarán en un momento determinado.

Como consecuencia de esta entrada se genera un *cronograma*, también llamado *gráfico de Gantt*. Es posible desarrollar un cronograma para todo el proyecto. Alternativamente, pueden generarse gráficos separados para cada función del proyecto o para cada individuo que trabaje en el proyecto.

La figura 27.3 ilustra el formato de un cronograma. Muestra una parte de un calendario de proyecto de software que enfatiza la tarea de formación del ámbito del concepto para un producto de software procesador de palabras (PP). Todas las tareas del proyecto (para el ámbito del

HERRAMIENTAS DE SOFTWARE

Calendarización del proyecto

Objetivo: El objetivo de las herramientas de calendarización del proyecto es permitir a un gerente de proyecto definir las tareas del trabajo, establecer sus dependencias, asignar recursos humanos a las tareas y desarrollar varios gráficos, cuadros y tablas que ayuden a monitorear y controlar el proyecto de software.

Mecánica: En general, las herramientas de calendarización de proyecto requieren la especificación de una estructura de distribución de trabajo de tareas o la generación de una red de tareas. Una vez definida la distribución o red de tareas (un esbozo), a cada una se le confieren fechas de inicio y de término, recursos humanos, fechas límite y otros datos. Entonces la herramienta genera una variedad de cronogramas y otras tablas que permiten a un gerente valorar el flujo de tareas de un proyecto. Dichos datos pueden actualizarse de manera continua conforme el proyecto avance.

Herramientas representativas:[8]

AMS Realtime, desarrollada por Advanced Management Systems (**www.amsusa.com**), proporciona capacidades de calendarización para proyectos de todos los tamaños y tipos.

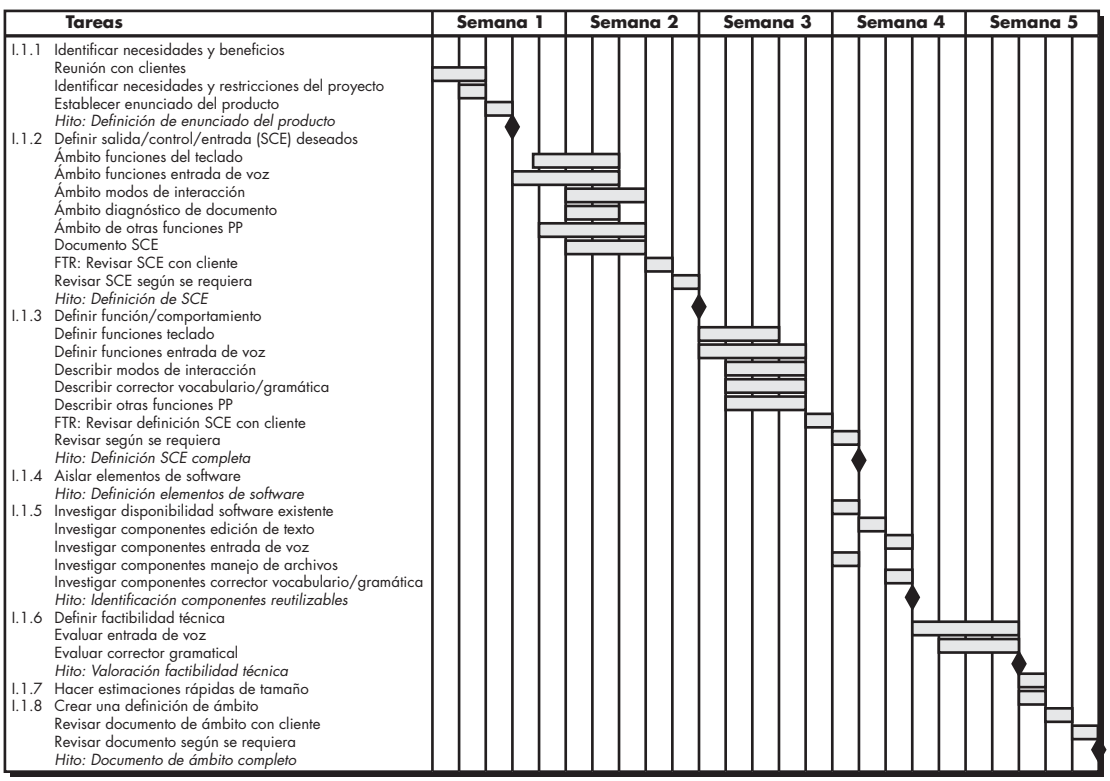
Microsoft Project, desarrollada por Microsoft (**www.microsoft.com**), es la herramienta de calendarización de proyecto basada en PC más ampliamente usada.

4C, desarrollada por 4C Systems (**www.4csys.com**), soporta todos los aspectos de planificación de proyecto, incluida la calendarización.

Una lista exhaustiva de proveedores y productos de software de gestión de proyectos puede encontrarse en **www.infogoal.com/pmc/pmcswr.htm**

concepto) se mencionan en la columna de la izquierda. Las barras horizontales indican la duración de cada tarea. Cuando muchas barras ocurren al mismo tiempo en el calendario, implican la concurrencia de tareas. Los diamantes indican hitos.

FIGURA 27.3
Ejemplo de cronograma



8 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas que hay en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

FIGURA 27.4
Ejemplo de tabla de proyecto

Tareas	Inicio planeado	Inicio real	Conclusión planeada	Conclusión real	Persona asignada	Esfuerzo asignado	Notas
I.1.1 Identificar necesidades y beneficios							Definir el ámbito requerirá más esfuerzo/tiempo
Reunión con clientes	sem1, d1	sem1, d1	sem1, d2	sem1, d2	BLS	2 p-d	
Identificar necesidades y restricciones del proyecto	sem1, d2	sem1, d2	sem1, d2	sem1, d2	JPP	1 p-d	
Establecer enunciado del producto	sem1, d3	sem1, d3	sem1, d3	sem1, d3	BLS/JPP	1 p-d	
Hito: Definición de enunciado del producto	sem1, d3	sem1, d3	sem1, d3	sem1, d3			
I.1.2 Definir salida/control/entrada (SCE) deseados							
Ámbito funciones del teclado	sem1, d4	sem1, d4	sem2, d2		BLS	1.5 p-d	
Ámbito funciones entrada de voz	sem1, d3	sem1, d3	sem2, d2		JPP	2 p-d	
Ámbito modos de interacción	sem2, d1		sem2, d3		MLL	1 p-d	
Ámbito diagnóstico de documento	sem2, d1		sem2, d2		BLS	1.5 p-d	
Ámbito de otras funciones PP	sem1, d4	sem1, d4	sem2, d3		JPP	2 p-d	
Documento SCE	sem2, d1		sem2, d3		MLL	3 p-d	
FTR: Revisar SCE con cliente	sem2, d3		sem2, d3		all	3 p-d	
Revisar SCE según se requiera	sem2, d4		sem2, d4		all	3 p-d	
Hito: Definición de SCE	sem2, d5		sem2, d5				
I.1.3 Definir función/comportamiento							

Una vez ingresada la información necesaria para la generación del cronograma, la mayoría de las herramientas de calendarización de proyecto de software producen *tablas de proyecto*, un listado tabular de todas las tareas del proyecto, sus fechas de inicio y término, planeadas y reales, y otra información relacionada (figura 27.4). Utilizadas en conjunto con el cronograma, las tablas de proyecto permiten monitorear el progreso.

27.5.2 Seguimiento del calendario

Cita:

"La regla básica del reporte de estado del software puede resumirse en una sola frase: 'sin sorpresas'."

Capers Jones

Si se desarrolló de manera adecuada, el calendario del proyecto se convierte en un mapa de caminos que define las tareas e hitos que se van a monitorear y controlar conforme el proyecto avance. El seguimiento puede lograrse en varias formas diferentes:

- Realizar reuniones periódicas del estado del proyecto, en las que cada miembro del equipo reporte avances y problemas
- Evaluar los resultados de todas las revisiones realizadas a través del proceso de ingeniería del software
- Determinar si los hitos formales del proyecto (los diamantes que se muestran en la figura 27.3) se lograron en la fecha prevista
- Comparar la fecha de inicio real con la fecha de inicio planeada para cada tarea de proyecto mencionada en la tabla de recursos (figura 27.4)
- Reunirse informalmente con los profesionales para obtener su valoración subjetiva del avance a la fecha y los problemas en el horizonte
- Usar análisis de valor ganado (sección 27.6) para valorar cuantitativamente el avance

CONSEJO

El mejor indicio de avance es la conclusión y la revisión exitosa de un producto operativo de software definido.

En realidad, todas estas técnicas de seguimiento las usan los gerentes de proyecto experimentados.

El control lo emplea un gerente de proyecto de software para administrar los recursos del proyecto, enfrentar los problemas y dirigir al personal del proyecto. Si las cosas van bien (es decir, si el proyecto avanza conforme el calendario y dentro de presupuesto, las revisiones indican que se realiza progreso real y que se alcanzan los hitos), el control es ligero. Pero cuando ocurren problemas, debe ejercerse el control para reconciliar los elementos discordantes tan rápidamente como sea posible. Después de diagnosticar un problema pueden enfocarse recur-

sos adicionales sobre el área problemática: puede reasignarse al personal o redefinirse el calendario del proyecto.

Cuando se enfrentan a severa presión debido a la fecha límite, los gerentes experimentados en ocasiones usan un calendario de proyecto y técnica de control llamado *time-boxing* (encuadre temporal) [Jal04]. La estrategia *time-boxing* reconoce que el producto completo no puede entregarse en la fecha límite preestablecida. Por tanto, se elige un paradigma de software incremental (capítulo 2) y se establece un calendario para cada entrega incremental.

PUNTO CLAVE

Cuando se alcanza la fecha de conclusión definida de una tarea encuadrada en el tiempo, el trabajo cesa para dicha tarea y comienza la siguiente.

Las tareas asociadas con cada incremento se encuadran en el tiempo. Esto significa que el calendario para cada tarea se ajusta trabajando en reversa desde la fecha de entrega hasta el momento del incremento. Alrededor de cada tarea se pone un "recuadro". Cuando una tarea llega a la frontera de su encuadre temporal (más o menos 10 por ciento), el trabajo se detiene y comienza la siguiente tarea.

Con frecuencia, la reacción inicial ante el enfoque *time-boxing* es negativa: "Si el trabajo no se termina, ¿cómo puedo avanzar?". La respuesta se encuentra en la forma en la que se logra el trabajo. Para cuando se llega a la frontera del encuadre temporal, es probable que 90 por ciento de la tarea esté completa.[9] El restante 10 por ciento, aunque importante, puede 1) demorarse hasta el siguiente incremento o 2) completarse más tarde si se requiere. En lugar de quedarse "atascado" en una tarea, el proyecto avanza hacia la fecha de entrega.

27.5.3 Seguimiento del progreso para un proyecto OO

Aunque un modelo iterativo es el mejor marco conceptual para un proyecto OO, el paralelismo de tareas hace difícil el seguimiento del proyecto. Acaso se tengan dificultades al establecer hitos significativos para un proyecto OO, debido a que algunas cosas diferentes ocurren a la vez. En general, los siguientes hitos importantes pueden considerarse "completos" cuando se satisfacen los criterios anotados.

Hitos técnicos: análisis OO completo

- Definición y revisión de todas las clases y de la jerarquía de clases.
- Definición y revisión de los atributos de clase y de las operaciones asociadas.
- Establecimiento y revisión de las relaciones de clase (capítulo 6).
- Creación y revisión de un modelo de comportamiento (capítulo 7).
- Anotación de las clases reutilizables.

Hitos técnicos: diseño OO completo

- Definición y revisión del conjunto de subsistemas.
- Asignación de clases a subsistemas y su revisión.
- Establecimiento y revisión de la asignación de tareas.
- Identificación de responsabilidades y colaboraciones.
- Diseño y revisión de atributos y operaciones.
- Creación y revisión del modelo de comunicación.

Hitos técnicos: programación OO completa

- Implementación en código de cada nueva clase, a partir del modelo de diseño.
- Implementación de las clases extraídas (a partir de la librería de reutilización).
- Construcción de prototipo o incremento.

9 Un cínico puede recordar el dicho: "el primer 90 por ciento del sistema requiere 90 por ciento del tiempo; el restante 10 por ciento del sistema requiere 90 por ciento del tiempo".

Depuración y pruebas ocurren en concierto mutuo. El estado de la depuración frecuentemente se valora considerando el tipo y número de errores "abiertos" (bugs).

Hitos técnicos: prueba de OO

- Revisión de la exactitud y completitud de los modelos de análisis y diseño OO.
- Desarrollo y revisión de una red clase-responsabilidad-colaboración (capítulo 6).
- Diseño de casos de prueba y realización de pruebas en el nivel de clase (capítulo 19) para cada clase.
- Diseño de casos de prueba, conclusión de pruebas de grupo (capítulo 19) e integración de clases.
- Conclusión de pruebas en el nivel de sistema.

Recuerde que el modelo de proceso OO es iterativo: cada uno de estos hitos puede revisarse nuevamente conforme diferentes incrementos se entreguen al cliente.

27.5.4 Calendarización para proyectos webapp

La *calendarización de proyectos webapp* distribuye el esfuerzo estimado a través de la línea temporal planeada (duración) para construir cada incremento de la *webapp*. Esto se logra asignando el esfuerzo a tareas específicas. Sin embargo, es importante observar que el calendario *webapp* global evoluciona con el tiempo. Durante la primera iteración se desarrolla un calendario macroscópico. Este tipo de calendario identifica todos los incrementos de la *webapp* y proyecta las fechas en las que se desplegará cada una. Conforme el desarrollo de un incremento está en marcha, la entrada para el incremento en el calendario macroscópico se refina en un calendario detallado. Aquí se identifican y calendarizan tareas de desarrollo específicas (requeridas para lograr una actividad).

Como ejemplo de calendarización macroscópica, considere la *webapp* **CasaSeguraAsegurada.com**. Si recuerda las discusiones anteriores acerca de **CasaSeguraAsegurada.com**, es posible identificar siete incrementos para el componente del proyecto basado en web:

Incremento 1: Información básica de compañía y producto

Incremento 2: Información detallada de producto y descargas

Incremento 3: Citas de producto y procesamiento de pedidos de producto

Incremento 4: Plantilla espacial y diseño de sistema de seguridad

Incremento 5: Servicios de información y solicitud de monitoreo

Incremento 6: Control en línea del equipo de monitoreo

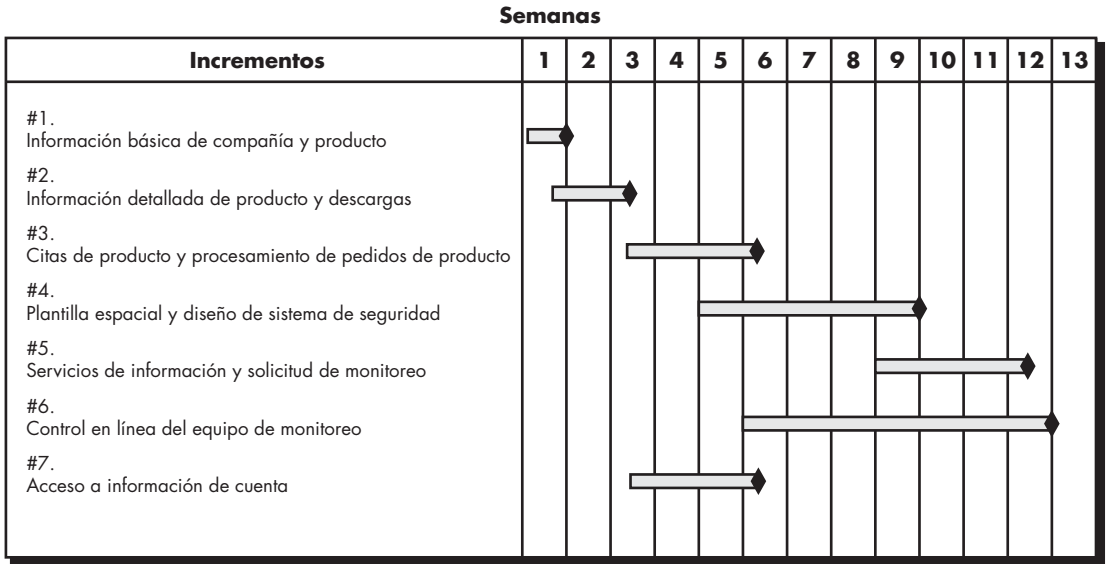
Incremento 7: Acceso a información de cuenta

El equipo consulta y negocia con los participantes y desarrolla un calendario de despliegue *preliminar* para los siete incrementos. En la figura 27.5 se muestra un cronograma para este calendario.

Es importante observar que las fechas de despliegue (representadas mediante diamantes en el cronograma) son preliminares y pueden cambiar conforme ocurran calendarizaciones más detalladas de los incrementos. No obstante, este calendario macroscópico brinda a la administración un indicio acerca de cuándo estará disponible el contenido y la funcionalidad, así como cuándo estará completo todo el proyecto. Como estimación preliminar, el equipo trabajará para desplegar todos los incrementos con un cronograma de 12 semanas. También vale la pena observar que algunos de los incrementos se desarrollarán en paralelo (por ejemplo, los incrementos 3, 4, 6 y 7). Esto supone que el equipo tendrá suficiente personal para hacer este trabajo paralelo.

Una vez desarrollado el calendario macroscópico, el equipo está listo para calendarizar las tareas del trabajo para un incremento específico. Para lograr esto, puede usar un marco conceptual de proceso genérico que sea aplicable a todos los incrementos de la *webapp*. Con las tareas

FIGURA 27.5
Cronograma para calendario de proyecto macroscópico



genéricas inferidas como parte del marco conceptual, se crea una *lista de tareas* como punto de partida y luego se adapta considerando el contenido y las funciones que se van a inferir para un incremento específico de la *webapp*.

Cada acción de marco conceptual (y sus tareas relacionadas) pueden adaptarse en una de cuatro formas: 1) una tarea se aplica como es, 2) una tarea se elimina porque no es necesaria para el incremento, 3) se agrega una nueva tarea (a la medida) y 4) una tarea se refina (elabora) en algunas subtareas nominales y cada una se vuelve parte del calendario.

Para ilustrar, considere una acción de *modelado de diseño* genérica para *webapps* que puede lograrse al aplicar alguna o todas las tareas siguientes:

- Diseño de la estética para la *webapp*.
- Diseño de la interfaz.
- Diseño del esquema de navegación.
- Diseño de la arquitectura de la *webapp*.
- Diseño del contenido y la estructura que lo soporta.
- Diseño de componentes funcionales.
- Diseño de mecanismos de seguridad y privacidad adecuados.
- Revisión del diseño.

Como ejemplo, considere la tarea genérica *Diseño de la interfaz* como se aplica al cuarto incremento de **CasaSeguraAsegurada.com**. Recuerde que el cuarto incremento implementa el contenido y la función para describir el espacio habitable o empresarial que se va a asegurar con el sistema de seguridad *CasaSegura*. En la figura 27.5, el cuarto incremento comienza al principio de la quinta semana y termina al final de la novena.

No hay duda de que la tarea *Diseño de la interfaz* debe realizarse. El equipo reconoce que el diseño de la interfaz es crucial para el éxito del incremento y decide refinar (elaborar) la tarea. Las siguientes subtareas se infieren para la tarea *Diseño de la interfaz* para el cuarto incremento:

- Desarrollo de un bosquejo de la plantilla de la página para la página de diseño del espacio.

- Revisión de plantilla con participantes.
- Diseño de mecanismos de navegación en la plantilla espacial.
- Diseño de plantilla "tablero de dibujo".[10]
- Desarrollo de detalles de procedimiento para la función de plantilla de pared gráfica.
- Desarrollo de detalles de procedimiento para el cálculo de longitud de pared y función de despliegue.
- Desarrollo de detalles de procedimiento para la función de plantilla de ventana gráfica.
- Desarrollo de detalles de procedimiento para la función de plantilla de puerta gráfica.
- Diseño de mecanismos para seleccionar componentes del sistema de seguridad (sensores, cámaras, micrófonos, etc.).
- Desarrollo de detalles de procedimiento para la plantilla gráfica de los componentes del sistema de seguridad.
- Realización de un par de recorridos según se requiera.

Estas tareas se vuelven parte del calendario de incrementos para el cuarto incremento de la aplicación Web y se asignan en el calendario de desarrollo de incrementos. Pueden ingresarse a software de calendarización y usarse para seguimiento y control.

CasaSegura

Seguimiento de calendario

La escena: Oficina de Doug Miller antes de iniciar el proyecto de software CasaSegura.

Participantes: Doug Miller (gerente del equipo de ingeniería de software CasaSegura) y Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería de software del producto.

La conversación:

Doug (observa una diapositiva powerpoint): El calendario para el primer incremento de CasaSegura parece razonable, pero vamos a tener problemas para monitorear el progreso.

Vinod (con una mirada preocupada): ¿Por qué? Tenemos tareas calendarizadas diariamente, llenas de productos operativos, y nos aseguramos de no asignar demasiados recursos.

Doug: Todo está bien, ¿pero cómo sabemos cuándo está completo el modelo de requerimientos para el primer incremento?

Jamie: Las cosas son iterativas, así que no es difícil.

Doug: Lo entiendo, pero... bueno, por ejemplo, considera la "definición de clases de análisis". Indicaste esto como un hito.

Vinod: Sí.

Doug: ¿Quién determina eso?

Jamie (agravado): Se hace cuando esté lista.

Doug: Eso no es suficientemente bueno, Jamie. Tenemos que calendarizar las RT [revisiones técnicas, capítulo 15] y no lo has hecho. La conclusión exitosa de una revisión en el modelo de análisis, por ejemplo, es un hito razonable. ¿Entendido?

Jamie (frunce el ceño): Está bien, de vuelta al tablero de dibujo.

Doug: No debería tomar más de una hora hacer las correcciones... todos los demás pueden comenzar ahora.

27.6 Análisis de valor ganado

PUNTO CLAVE

El valor ganado proporciona un indicio cuantitativo del progreso.

En la sección 27.5 se estudiaron algunos enfoques cualitativos sobre el seguimiento del proyecto. Cada uno proporciona al gerente del proyecto un indicio del progreso, pero una valoración de la información proporcionada es un poco subjetiva. Es razonable preguntar si existe una técnica cuantitativa para valorar el progreso conforme el equipo de software avanza a través de

10 En esta etapa, el equipo vislumbra la creación del espacio al literalmente dibujar las paredes, ventanas y puertas usando funciones gráficas. Las líneas de pared "encajarán" en puntos de agarre. Las dimensiones de la pared se desplegarán de manera automática. Ventanas y puertas se colocarán gráficamente. El usuario final también podrá seleccionar sensores, cámaras y otros elementos específicos, y colocarlos una vez definido el espacio.

las tareas asignadas en el calendario del proyecto. De hecho, sí existe una técnica para realizar el análisis cuantitativo del progreso. Se llama *análisis de valor ganado* (AVG). Humphrey [Hum95] analiza el valor ganado en los siguientes términos:

> El sistema de valor ganado ofrece una escala de valor común para toda tarea [de proyecto de software], sin importar el tipo de trabajo que se va a realizar. Se estiman las horas totales para hacer todo el proyecto y a cada tarea se le da un valor ganado con base en su porcentaje estimado del total.

Dicho en forma todavía más simple, el valor ganado es una medida de progreso. Le permite valorar el "porcentaje de completitud" de un proyecto usando análisis cuantitativo en lugar de apoyarse en una corazonada. De hecho, Fleming y Koppleman [Fle98] argumentan que el análisis de valor ganado "proporciona lecturas precisas y confiables del rendimiento tan tempranamente como con 15 por ciento del proyecto". Para determinar el valor ganado se realizan los siguientes pasos:

? ¿Cómo se calcula el valor ganado y cómo se le usa para valorar el progreso?

1. El *costo presupuestado del trabajo calendarizado* (CPTC) se determina para cada tarea representada en el calendario. Durante la estimación se planifica el trabajo (en persona-horas o persona-días) de cada tarea de ingeniería del software. Por tanto, $CPTC_i$ es el esfuerzo planificado para la tarea *i*. Para determinar el progreso en un punto determinado a lo largo del calendario del proyecto, el valor de CPTC es la suma de los valores $CPTC_i$ para todas las tareas que deben estar completas en dicho punto en el tiempo señalado en el calendario del proyecto.
2. Los valores CPTC para todas las tareas se suman para inferir el *presupuesto al concluir* (PAC). Por tanto,

   PAC = Σ ($CPTC_k$) para todas las tareas *k*
3. A continuación, se calcula el *costo presupuestado del trabajo realizado* (CPTR). El valor de CPTR es la suma de los valores CPTC para todas las tareas que realmente se concluyeron en un punto en el tiempo sobre el calendario del proyecto.

Wilkens [Wil99] observa que "la distinción entre el CPTC y el CPTR es que el primero representa el presupuesto de las actividades que se planea completar y el último representa el presupuesto de las actividades que realmente se completaron". Determinados los valores para CPTC, PAC y CPTR, es posible calcular importantes indicadores del progreso:

$$\text{Índice de rendimiento en calendario, IRS} = \frac{\text{CPTR}}{\text{CPTC}}$$

$$\text{Variación en calendario, VC} = \text{CPTR} - \text{CPTC}$$

WebRef

En **www.acq.osd.mil/pm/** puede encontrarse una amplia variedad de recursos para análisis de valor ganado.

El IRS es un indicio de la eficiencia con la que el proyecto utiliza los recursos calendarizados. Un valor IRS cercano a 1.0 indica ejecución eficiente del calendario del proyecto. VC es simplemente un indicio absoluto de la variación con respecto al calendario planeado.

$$\text{Porcentaje calendarizado para conclusión} = \frac{\text{CPTC}}{\text{PAC}}$$

proporciona un indicio del porcentaje de trabajo que debe concluirse hacia el tiempo *t*.

$$\text{Porcentaje completo} = \frac{\text{CPTC}}{\text{PAC}}$$

proporciona un indicio cuantitativo del porcentaje de completitud del proyecto en un punto dado en el tiempo *t*.

También es posible calcular el *costo real del trabajo realizado* (CRTR). El valor para CRTR es la suma del esfuerzo realmente utilizado en las tareas que se completaron en un punto en el tiempo en el calendario del proyecto. Entonces es posible calcular

$$\text{Índice de rendimiento de costo, IRC} = \frac{\text{CPTR}}{\text{CRTR}}$$

$$\text{Variación en costo, VC} = \text{CPTR} - \text{CRTR}$$

Un valor IRC cercano a 1.0 proporciona un fuerte indicio de que el proyecto está dentro de su presupuesto definido. La VC es un indicio absoluto de ahorros en costo (contra los costos planificados) o déficits en una etapa particular de un proyecto.

Como otros radares en el horizonte, el análisis de valor ganado ilumina las dificultades en la calendarización antes de que puedan ser aparentes de otro modo. Esto permite tomar acciones correctivas antes de que se desarrolle una crisis en el proyecto.

27.7 Resumen

La calendarización es la culminación de una actividad de planificación que es un componente principal de la administración de proyectos de software. Cuando se combina con los métodos de estimación y análisis de riesgos, establece un mapa de caminos para el gerente del proyecto.

La calendarización comienza con la descomposición del proceso. Las características del proyecto se usan para adaptar un conjunto de tareas adecuado para el trabajo que se va a realizar. Una red de tareas muestra cada tarea de ingeniería, su dependencia de otras tareas y su duración proyectada. La red de tareas se usa para calcular la ruta crítica, un cronograma y otra información del proyecto. Al usar el calendario como guía, puede monitorearse y controlar cada paso en el proceso de software.

Problemas y puntos por evaluar

27.1. Las fechas límite "irracionales" son un hecho de la vida en el negocio del software. ¿Cómo debe proceder si se enfrenta con una?

27.2. ¿Cuál es la diferencia entre un calendario macroscópico y uno detallado? ¿Es posible administrar un proyecto si sólo se desarrolla un calendario macroscópico? Explique su respuesta.

27.3. ¿Puede existir un caso donde un hito de proyecto de software no se ligue a una revisión? Si es así, ofrezca uno o más ejemplos.

27.4. El "gasto en comunicación" puede ocurrir cuando múltiples personas trabajan en un proyecto de software. El tiempo que se emplea para comunicarse con los demás reduce la productividad individual (LOC/mes) y el resultado puede ser menos productividad para el equipo. Ilustre (cuantitativamente) cómo usan los ingenieros versados en las buenas prácticas de ingeniería de software revisiones técnicas que pueden aumentar la tasa de producción de un equipo (cuando se compara con la suma de las tasas de producción individuales) y qué tipo de revisiones técnicas usan. Sugerencia: Puede suponer que las revisiones reducen el reproceso y que el reproceso puede representar de 20 a 40 por ciento del tiempo de una persona.

27.5. Aunque agregar personas a un proyecto de software retrasado puede hacer que se retrase aún más, existen circunstancias en las que esto no es cierto. Descríbalas.

27.6. La relación entre personal y tiempo es enormemente no lineal. Con la ecuación de software de Putnam (descrita en la sección 27.2.2), desarrolle una tabla que relacione el número de personas con la duración del proyecto para un proyecto de software que requiere 50 000 LOC y 15 personas-años de esfuerzo (el parámetro de productividad es 5 000 y $B = 0.37$). Suponga que el software debe entregarse en más o menos 24 meses con una posibilidad de prórroga de 12 meses.

27.7. Suponga el lector que una universidad lo contrata para desarrollar un sistema para registrarse en línea a los cursos (OLCRS, según sus siglas en inglés). Primero, actúe como el cliente (si es estudiante, ¡debe resultarle sencillo!) y especifique las características de un buen sistema. (De manera alternativa, su instructor le proporcionará un conjunto de requerimientos preliminares para el sistema.) Con los métodos de estimación estudiados en el capítulo 26 desarrolle una estimación de esfuerzo y duración para OLCRS. Sugiera cómo:

a) Definiría actividades de trabajo paralelas durante el proyecto OLCRS.

b) Distribuiría el esfuerzo a través del proyecto.

c) Establecería hitos para el proyecto.

27.8. Seleccione un conjunto de tareas adecuado para el proyecto OLCRS.

27.9. Defina una red de tareas para el OLCRS descrito en el problema 27.7 o, alternativamente, para otro proyecto de software que sea de su interés. Asegúrese de mostrar tareas e hitos y de unir estimaciones de esfuerzo y duración a cada tarea. Si es posible, use una herramienta de calendarización automatizada para realizar este trabajo.

27.10. Si está disponible una herramienta de calendarización automatizada determine la ruta crítica para la red definida en el problema 27.9.

27.11. Con la herramienta de calendarización (si está disponible), o con papel y lápiz (si es necesario), desarrolle un cronograma para el proyecto OLCRS.

27.12. Suponga que usted es un gerente de proyecto de software y que se le pide calcular estadísticas de valor ganado para un pequeño proyecto de software. El proyecto tiene 56 tareas planeadas que se estima que requieren 582 personas-días para completarlas. En el momento en el que se le pide hacer el análisis de valor ganado, se han completado 12 tareas. Sin embargo, el calendario del proyecto indica que deberían estar completas 15. Están disponibles los siguientes datos de calendarización (en persona-días):

Tarea	Esfuerzo planeado	Esfuerzo real
1	12.0	12.5
2	15.0	11.0
3	13.0	17.0
4	8.0	9.5
5	9.5	9.0
6	18.0	19.0
7	10.0	10.0
8	4.0	4.5
9	12.0	10.0
10	6.0	6.5
11	5.0	4.0
12	14.0	14.5
13	16.0	—
14	6.0	—
15	8.0	—

Calcule IRS, variación en calendario, porcentaje calendarizado para conclusión, porcentaje completo, IRC y variación en costo para el proyecto.

Lecturas y fuentes de información adicionales

Virtualmente, todo libro escrito acerca de administración de proyectos de software contiene un análisis de la calendarización. Wysoki (*Effective Project Management,* Wiley, 2006), Lewis (*Project Planning Scheduling and Control,* 4a. ed., McGraw-Hill, 2006), Luckey y Phillips (*Software Project Management for Dummies,* For Dummies, 2006), Kerzner (*Project Management: A Systems Approach to Planning, Scheduling, and Controlling,* 9a. ed., Wiley, 2005), Hughes (*Software Project Management,* McGraw-Hill, 2005), The Project Management Institute (*PMBOK Guide,* 3a. ed., PMI, 2004), Lewin (*Better Software Project Management,* Wiley, 2001) y Bennatan (*On Time, Within Budget: Software Project Management Practices and Techniques,* 3a. ed., Wiley, 2000) contienen valiosos análisis sobre la materia. Aunque es específico para aplicación, Harris (*Planning and Scheduling Using Microsoft Office Project 2007*, Eastwood Harris Pty Ltd., 2007) proporciona un útil estudio sobre cómo pueden usarse las herramientas de calendarización para seguimiento y control exitosos de un proyecto de software.

Fleming y Koppelman (*Earned Value Project Management,* 3a ed., Project Management Institute Publications, 2006), Budd (*A Practical Guide to Earned Value Project Management,* Management Concepts, 2005) y

Webb y Wake (*Using Earned Value: A Project Manager's Guide,* Ashgate Publishing, 2003) analizan con detalle considerable el uso de las técnicas de valor ganado para planificación, seguimiento y control de proyectos.

En internet está disponible una gran variedad de fuentes de información acerca de calendarización de proyectos de software. Una lista actualizada de referencias en la World Wide Web que son relevantes para la calendarización de proyectos de software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

28 ADMINISTRACIÓN DEL RIESGO

CONCEPTOS CLAVE

categorías de riesgo 642
estrategias 641
  proactiva 641
  reactiva 641
exposición al riesgo 648
identificación. 642
lista de verificación de ítem de riesgo 643
MMMR. 651
proyección 644
refinamiento 649
seguridad y riesgos 651
tabla de riesgo 645
valoración. 643

En su libro acerca de administración y análisis de riesgos, Robert Charette [Cha89] presenta una definición conceptual de riesgo:

> Primero, el riesgo se preocupa por los acontecimientos futuros. Ayer y hoy están más allá de la preocupación activa, pues ya cosechamos lo que previamente se sembró por nuestras acciones pasadas. La cuestión tiene que ver, por tanto, con si podemos, al cambiar nuestras acciones de hoy, crear una oportunidad para una situación diferente y esperanzadoramente mejor para nosotros en el mañana. Esto significa, segundo, que el riesgo involucra cambio, como en los cambios de mentalidad, de opinión, de acciones o de lugares [...] [Tercero,] el riesgo involucra elección y la incertidumbre que ella conlleva. En consecuencia, paradójicamente, el riesgo, como la muerte y los impuestos, es una de las pocas certezas de la vida.

Cuando se considera el riesgo en el contexto de la ingeniería del software, los tres fundamentos conceptuales de Charette siempre están presentes. El futuro es su preocupación: ¿qué riesgos pueden hacer que el proyecto de software salga defectuoso? El cambio es lo que preocupa: ¿cómo afectan en los cronogramas y en el éxito global los cambios que puede haber en los requisitos del cliente, en las tecnologías de desarrollo, en los entornos meta y en todas las otras entidades conectadas con el proyecto? Por último, se debe lidiar con las opciones: ¿qué métodos y herramientas deben usarse, cuántas personas deben involucrarse, cuánto énfasis es "suficiente" poner en la calidad?

Peter Drucker [Dru75] dijo alguna vez: "aunque sea fútil intentar eliminar el riesgo, y cuestionable intentar minimizarlo, es esencial que los riesgos tomados sean los riesgos correctos".

UNA MIRADA RÁPIDA

¿Qué es? El análisis y la administración del riesgo son acciones que ayudan al equipo de software a entender y manejar la incertidumbre. Muchos problemas pueden plagar un proyecto de software. Un riesgo es un problema potencial: puede ocurrir, puede no ocurrir. Pero, sin importar el resultado, realmente es una buena idea identificarlo, valorar su probabilidad de ocurrencia, estimar su impacto y establecer un plan de contingencia para el caso de que el problema realmente ocurra.

¿Quién lo hace? Todos los involucrados en el proceso de software (gerentes, ingenieros de software y otros interesados) participan en el análisis y la administración del riesgo.

¿Por qué es importante? Piense en la consigna de los boy scouts: "estar preparados". El software es una empresa difícil. Muchas cosas pueden salir mal y, francamente, muchas con frecuencia lo hacen. Por esta razón es que estar preparado, comprender los riesgos y tomar medidas proactivas para evitarlos o manejarlos son elementos clave de una buena administración de proyecto de software.

¿Cuáles son los pasos? Reconocer qué puede salir mal es el primer paso, llamado "identificación de riesgos". A continuación, cada riesgo se analiza para determinar la probabilidad de que ocurra y el daño que causará si ocurre. Una vez establecida esta información se clasifican los riesgos, por probabilidad e impacto. Finalmente, se desarrolla un plan para manejar aquellos que tengan alta probabilidad y alto impacto.

¿Cuál es el producto final? Se produce un plan para mitigar, monitorear y manejar el riesgo (MMMR) o un conjunto de hojas de información de riesgo.

¿Cómo me aseguro de que lo hice bien? Los riesgos que se analizan y manejan deben inferirse a partir de un estudio del personal, el producto, el proceso y el proyecto. El MMMR debe revisarse conforme avance el proyecto para asegurarse de que los riesgos se mantienen actualizados. Los planes de contingencia para administración del riesgo deben ser realistas.

Antes de poder identificar los "riesgos correctos" que se van a tomar durante un proyecto de software, es importante identificar todos los que son obvios para gerentes y profesionales.

28.1 Estrategias reactivas de riesgo frente a estrategias proactivas de riesgo

Cita:

"Si no atacas de manera activa los riesgos, ellos te atacarán de manera activa."

Tom Gilb

Las estrategias *reactivas* de riesgo se han llamado irrisoriamente la "escuela de gestión de riesgo de Indiana Jones" [Tho92]. En las películas que llevan su nombre, Indiana Jones, cuando enfrenta una dificultad abrumadora, invariablemente dice: "No te preocupes, ¡pensaré en algo!". Al nunca preocuparse por los problemas hasta que suceden, Indy reaccionará en alguna forma heroica.

Tristemente, el gerente promedio de proyectos de software no es Indiana Jones y los miembros del equipo del proyecto de software no son sus fieles ayudantes. Sin embargo, la mayoría de los equipos de software se apoyan exclusivamente en estrategias reactivas de riesgo. Cuando mucho, una estrategia reactiva monitorea el proyecto para riesgos probables. Los recursos se hacen a un lado para lidiar con los riesgos, hasta que se convierten en problemas reales. De manera más común, el equipo de software no hace nada acerca de los riesgos hasta que algo sale mal. Entonces el equipo se apresura a entrar en acción con la intención de corregir el problema rápidamente. Con frecuencia esto se llama *modo bombero*. Cuando falla, la "administración de crisis" [Cha92] toma el control y el proyecto está en un peligro real.

Una estrategia considerablemente más inteligente para la administración del riesgo es ser proactivo. Una estrategia *proactiva* comienza mucho antes de iniciar el trabajo técnico. Los riesgos potenciales se identifican, su probabilidad e impacto se valoran y se clasifican por importancia. Luego, el equipo de software establece un plan para gestionar el riesgo. El objetivo principal es evitarlo, pero, dado que no todos los riesgos son evitables, el equipo trabaja para desarrollar un plan de contingencia que le permitirá responder en forma controlada y efectiva. A lo largo del resto de este capítulo se estudia una estrategia proactiva de gestión del riesgo.

28.2 Riesgos de software

Aunque hay un considerable debate acerca de la definición adecuada de riesgo de software, existe un acuerdo general en que los riesgos siempre involucran dos características: *incertidumbre* (el riesgo puede o no ocurrir; es decir, no hay riesgos 100 por ciento probables[1]) y *pérdida* (si el riesgo se vuelve una realidad, ocurrirán consecuencias o pérdidas no deseadas [Hig95]). Cuando se analizan los riesgos es importante cuantificar el nivel de incertidumbre y el grado de pérdidas asociados con cada riesgo. Para lograr esto, se consideran diferentes categorías de riesgos.

? ¿Qué tipos de riesgos es probable encontrar conforme se construye el software?

Los *riesgos del proyecto* amenazan el plan del proyecto, es decir, si los riesgos del proyecto se vuelven reales, es probable que el calendario del proyecto se deslice y que los costos aumenten. Los riesgos del proyecto identifican potenciales problemas de presupuesto, calendario, personal (tanto técnico como en la organización), recursos, participantes y requisitos, así como su impacto sobre un proyecto de software. En el capítulo 26, la complejidad, el tamaño y el grado de incertidumbre estructural del proyecto también se definieron como factores de riesgos para el proyecto (y la estimación).

Los *riesgos técnicos* amenazan la calidad y temporalidad del software que se va a producir. Si un riesgo técnico se vuelve una realidad, la implementación puede volverse difícil o imposible. Los riesgos técnicos identifican potenciales problemas de diseño, implementación, interfaz,

1 Un riesgo que es 100 por ciento probable es una restricción sobre el proyecto de software.

verificación y mantenimiento. Además, la ambigüedad en la especificación, la incertidumbre técnica, la obsolescencia técnica y la tecnología "de punta" también son factores de riesgo. Los riesgos técnicos ocurren porque el problema es más difícil de resolver de lo que se creía.

Los *riesgos empresariales* amenazan la viabilidad del software que se va a construir y con frecuencia ponen en peligro el proyecto o el producto. Los candidatos para los cinco principales riesgos empresariales son: 1) construir un producto o sistema excelente que realmente no se quiere (riesgo de mercado), 2) construir un producto que ya no encaje en la estrategia empresarial global de la compañía (riesgo estratégico), 3) construir un producto que el equipo de ventas no sabe cómo vender (riesgo de ventas), 4) perder el apoyo de los administradores debido a un cambio en el enfoque o en el personal (riesgo administrativo) y 5) perder apoyo presupuestal o de personal (riesgos presupuestales).

Es extremadamente importante observar que la categorización simple de riesgos no siempre funciona. Algunos de ellos son simplemente impredecibles por adelantado.

Otra categorización general de los riesgos es la propuesta por Charette [Cha89]. Los *riesgos conocidos* son aquellos que pueden descubrirse después de una evaluación cuidadosa del plan del proyecto, del entorno empresarial o técnico donde se desarrolla el proyecto y de otras fuentes de información confiables (por ejemplo, fecha de entrega irreal, falta de requisitos documentados o ámbito de software, pobre entorno de desarrollo). Los *riesgos predecibles* se extrapolan de la experiencia en proyectos anteriores (por ejemplo, rotación de personal, pobre comunicación con el cliente, disolución del esfuerzo del personal conforme se atienden las solicitudes de mantenimiento). Los *riesgos impredecibles* son el comodín en la baraja. Pueden ocurrir y lo hacen, pero son extremadamente difíciles de identificar por adelantado.

Cita:

"Los proyectos sin riesgos reales son perdedores. Casi siempre están desprovistos de beneficio; es por esto por lo que no se hicieron años atrás."

Tom DeMarco y Tim Lister

INFORMACIÓN

Siete principios de la administración de riesgos

El Software Engineering Institute (SEI) **(www.sei.cmu.edu)** identifica siete principios que "ofrecen un marco conceptual para lograr una administración de riesgo efectiva". Éstos son:

Mantener una perspectiva global: ver los riesgos del software dentro del contexto de un sistema donde el riesgo es un componente y el problema empresarial que se pretende resolver.

Tomar una visión de previsión: pensar en los riesgos que pueden surgir en el futuro (por ejemplo, debido a cambios en el software); establecer planes de contingencia de modo que los eventos futuros sean manejables.

Alentar la comunicación abierta: si alguien enuncia un riesgo potencial, no lo ignore. Si un riesgo se propone de manera informal, considérelo. Aliente a todos los participantes y usuarios a sugerir riesgos en cualquier momento.

Integrar: una consideración de riesgo debe integrarse en el proceso del software.

Enfatizar un proceso continuo: el equipo debe vigilar a lo largo del proceso de software, modificar los riesgos identificados conforme se conozca más información y agregar unos nuevos conforme se logre mejor comprensión.

Desarrollar una visión de producto compartida: si todos los participantes comparten la misma visión del software, es probable que haya mejor identificación y valoración del riesgo.

Alentar el trabajo en equipo: los talentos, habilidades y conocimientos de todos los participantes deben reunirse cuando se realicen actividades de administración de riesgos.

28.3 IDENTIFICACIÓN DE RIESGOS

La identificación de riesgos es un intento sistemático por especificar amenazas al plan del proyecto (estimaciones, calendario, carga de recursos, etc.). Al identificar los riesgos conocidos y predecibles, el gerente de proyecto da un primer paso para evitarlos cuando es posible y para controlarlos cuando es necesario.

Existen dos tipos distintos de riesgos para cada una de las categorías que se presentaron en la sección 28.2: riesgos genéricos y riesgos específicos del producto. Los *riesgos genéricos* son una amenaza potencial a todo proyecto de software. Los *riesgos específicos del producto* pueden

Aunque es importante considerar los riesgos genéricos, son los riesgos específicos del producto los que provocan más dolores de cabeza. Asegúrese de emplear tiempo para identificar tantos riesgos específicos del producto como sea posible.

identificarse solamente por quienes tienen clara comprensión de la tecnología, el personal y el entorno específico del software que se construye. Para identificar los riesgos específicos del producto, examine el plan del proyecto y el enunciado de ámbito del software, y desarrolle una respuesta a la siguiente pregunta: ¿qué características especiales de este producto pueden amenazar el plan del proyecto?

Un método para identificar riesgos es crear una lista de verificación de ítem de riesgo. La lista de verificación puede usarse para identificación del riesgo y así enfocarse sobre algún subconjunto de riesgos conocidos y predecibles en las siguientes subcategorías genéricas:

- *Tamaño del producto:* riesgos asociados con el tamaño global del software que se va a construir o a modificar.
- *Impacto empresarial:* riesgos asociados con restricciones impuestas por la administración o por el mercado.
- *Características de los participantes:* riesgos asociados con la sofisticación de los participantes y con la habilidad de los desarrolladores para comunicarse con los participantes en forma oportuna.
- *Definición del proceso:* riesgos asociados con el grado en el que se definió el proceso de software y la manera como se sigue por parte de la organización desarrolladora.
- *Entorno de desarrollo:* riesgos asociados con la disponibilidad y calidad de las herramientas por usar para construir el producto.
- *Tecnología por construir:* riesgos asociados con la complejidad del sistema que se va a construir y con lo "novedoso" de la tecnología que se incluye en el sistema.
- *Tamaño y experiencia del personal:* riesgos asociados con la experiencia técnica y de proyecto global de los ingenieros de software que harán el trabajo.

La lista de verificación de ítem de riesgo puede organizarse en diferentes formas. Las preguntas relevantes en cada uno de los temas pueden responderse para cada proyecto de software. Las respuestas a dichas preguntas permiten estimar el impacto del riesgo. Un formato diferente de lista de verificación de ítem de riesgo simplemente menciona las características que son relevantes en cada subcategoría genérica. Finalmente, se menciona un conjunto de "componentes y promotores de riesgo" [AFC88] junto con sus probabilidades de ocurrencia. Los promotores de desempeño, apoyo, costo y calendario se analizan en respuesta a las preguntas anteriores.

Algunas listas de verificación exhaustivas para riesgo de proyecto de software están disponibles en la red (por ejemplo, [Baa07], [NAS07], [Wor04]). Puede usar dichas listas de verificación para comprender los riesgos genéricos para proyectos de software.

28.3.1 Valoración del riesgo de proyecto global

Las siguientes preguntas se infirieron de los datos de riesgo obtenidos al entrevistar en diferentes partes del mundo a gerentes de proyectos de software experimentados [Kei98]. Las preguntas se ordenan por su importancia relativa para el éxito del proyecto.

¿El proyecto de software en el que trabaja está en serio peligro?

1. ¿Los gerentes de software y de cliente se reunieron formalmente para apoyar el proyecto?
2. ¿Los usuarios finales se comprometen de manera entusiasta con el proyecto y con el sistema/producto que se va a construir?
3. ¿El equipo de ingeniería del software y sus clientes entienden por completo los requisitos?
4. ¿Los clientes se involucraron plenamente en la definición de los requisitos?
5. ¿Los usuarios finales tienen expectativas realistas?

6. ¿El ámbito del proyecto es estable?
7. ¿El equipo de ingeniería del software tiene la mezcla correcta de habilidades?
8. ¿Los requisitos del proyecto son estables?
9. ¿El equipo de proyecto tiene experiencia con la tecnología que se va a implementar?
10. ¿El número de personas que hay en el equipo del proyecto es adecuado para hacer el trabajo?
11. ¿Todas las divisiones de cliente/usuario están de acuerdo en la importancia del proyecto y en los requisitos para el sistema/producto que se va a construir?

WebRef

Risk radar es una base de datos y herramientas que ayudan a los gerentes a identificar, clasificar y comunicar riesgos de proyecto. Puede encontrarse en **www.spmn.com**

Si alguna de estas preguntas se responde de manera negativa deben establecerse sin falta pasos de mitigación, monitoreo y gestión. El grado en el que el proyecto está en riesgo es directamente proporcional al número de respuestas negativas a dichas preguntas.

28.3.2 Componentes y promotores de riesgo

La fuerza aérea estadounidense [AFC88] publicó un escrito que contiene excelentes lineamientos para la identificación y reducción de los riesgos de software. El enfoque de la fuerza aérea requiere que el gerente del proyecto identifique los promotores de riesgo que afectan los componentes de riesgo de software: rendimiento, costo, apoyo y calendario. En el contexto de este análisis, los componentes de riesgo se definen en la forma siguiente:

Cita:

"La administración del riesgo es administración de proyecto para adultos."

Tim Lister

- *Riesgo de rendimiento:* grado de incertidumbre de que el producto satisfará sus requisitos y se ajustará al uso pretendido.
- *Riesgo de costo:* grado de incertidumbre de que el presupuesto del proyecto se mantendrá.
- *Riesgo de apoyo:* grado de incertidumbre de que el software resultante será fácil de corregir, adaptar y mejorar.
- *Riesgo de calendario:* grado de incertidumbre de que el calendario del proyecto se mantendrá y de que el producto se entregará a tiempo.

El impacto de cada promotor de riesgo sobre el componente de riesgo se divide en una de cuatro categorías de impacto: despreciable, marginal, crítico o catastrófico. En la figura 28.1 [Boe89] se describe una caracterización de las potenciales consecuencias de errores (hileras con etiqueta 1) o de un fallo para lograr el resultado deseado (hileras con etiqueta 2). La categoría de impacto se elige con base en la caracterización que se ajusta mejor a la descripción en la tabla.

28.4 Proyección del riesgo

La *proyección del riesgo*, también llamada *estimación del riesgo*, intenta calificar cada riesgo en dos formas: 1) la posibilidad o probabilidad de que el riesgo sea real y 2) las consecuencias de los problemas asociados con el riesgo, en caso de que ocurra. Usted trabaja junto con otros gerentes y personal técnico para realizar cuatro pasos de proyección de riesgo:

1. Establecer una escala que refleje la probabilidad percibida de un riesgo.
2. Delinear las consecuencias del riesgo.
3. Estimar el impacto del riesgo sobre el proyecto y el producto.
4. Valorar la precisión global de la proyección del riesgo de modo que no habrá malos entendidos.

FIGURA 28.1
Valoración de impacto.
Fuente: [Boe89].

Componentes / Categoría		Rendimiento	Apoyo	Costo	Calendario
Catastrófico	1	La falla para satisfacer el requisito resultaría en fallo en la misión		La falla da como resultado aumento de costos y demoras en el calendario, con valores esperados en exceso de US$500K	
	2	Degradación significativa para no lograr el rendimiento técnico	Software que no responde o no puede tener apoyo	Significativos recortes financieros, probable agotamiento de presupuesto	IOC inalcanzable
Crítico	1	Falla para satisfacer el requisito degradaría el rendimiento del sistema hasta un punto donde el éxito de la misión sería cuestionable		La falla da como resultado demoras operativas y/o aumento de costos con valor esperado de US$100K a US$500K	
	2	Cierta reducción en rendimiento técnico	Demoras menores en modificaciones de software	Cierto recorte de recursos financieros, posible agotamiento	Posible deterioro en IOC
Marginal	1	Falla para satisfacer los requisitos resultaría en degradación de misión secundaria		Costos, impactos y/o calendario recuperable se deterioran con valor esperado de US$1K a US$100K	
	2	Reducción mínima a pequeña en rendimiento técnico	Apoyo de software receptivo	Suficientes recursos financieros	Calendario realista, alcanzable
Despreciable	1	Falla para satisfacer requisitos crearía inconvenientes o impacto no operativo		Error da como resultado costo menor y/o impacto en calendario con valor esperado de menos de US$1K	
	2	No reducción en rendimiento técnico	Software fácilmente soportable	Posible subejercicio de presupuesto	IOC alcanzable con facilidad

Nota: 1) La consecuencia potencial de errores o fallos de software no detectados.
2) La consecuencia potencial si el resultado deseado no se alcanza.

Piense duro acerca del software que está a punto de construir y pregúntese: ¿qué puede salir mal? Cree su propia lista y pida a otros miembros del equipo que hagan lo mismo.

La intención de estos pasos es considerar los riesgos de manera que conduzcan a una priorización. Ningún equipo de software tiene los recursos para abordar todo riesgo posible con el mismo grado de rigor. Al priorizar los riesgos es posible asignar recursos donde tendrán más impacto.

28.4.1 Elaboración de una tabla de riesgos

Una tabla de riesgos proporciona una técnica simple para proyección de riesgos.[2] Una tabla de muestra de riesgo se ilustra en la figura 28.2.

Comience por elaborar una lista de todos los riesgos (sin importar cuán remotos sean) en la primera columna de la tabla. Esto puede lograrse con la ayuda de las listas de verificación de ítem de riesgo mencionadas en la sección 28.3. Cada riesgo se clasifica en la segunda columna (por ejemplo, TP implica un riesgo de tamaño de proyecto, EMP implica un riesgo empresarial). La probabilidad de ocurrencia de cada riesgo se ingresa en la siguiente columna de la tabla. El valor de probabilidad para cada riesgo puede estimarse individualmente por los miembros del equipo. Una forma de lograr esto es encuestar a todos los miembros del equipo hasta que su valoración colectiva de la probabilidad del riesgo comience a convergir.

A continuación, se valora el impacto de cada riesgo. Cada componente de riesgo se valora usando la caracterización que se presenta en la figura 28.1 y se determina una categoría de

2 La tabla de riesgos puede implementarse como un modelo de hoja de cálculo. Esto permite fácil manipulación y ordenamiento de las entradas.

FIGURA 28.2
Ejemplo de tabla de riesgo previo al ordenamiento

Riesgos	Categoría	Probabilidad	Impacto	RMMM
Estimación de tamaño puede ser significativamente baja	PS	60%	2	
Mayor número de usuarios que el planificado	PS	30%	3	
Menos reuso que el planificado	PS	70%	2	
Usuarios finales que se resisten al sistema	BU	40%	3	
Fecha de entrega será apretada	BU	50%	2	
Pérdida de fondos	CU	40%	1	
Cliente cambiará requisitos	PS	80%	2	
Tecnología no satisfará las expectativas	TE	30%	1	
Falta de capacitación en herramientas	DE	80%	3	
Personal inexperto	ST	30%	2	
Alta rotación de personal	ST	60%	2	
Σ				
Σ				
Σ				

Valores de impacto:
1—catastrófico
2—crítico
3—marginal
4—despreciable

impacto. Las categorías para cada uno de los cuatro componentes de riesgo (rendimiento, apoyo, costo y calendario) se promedian[3] para determinar un valor de impacto global.

Una vez completadas las primeras cuatro columnas de la tabla de riesgos, la tabla se ordena por probabilidad y por impacto. Los riesgos de alta probabilidad y alto impacto se ubican en la parte superior de la tabla y los riesgos de baja probabilidad se ubican en el fondo. Esto logra una priorización de riesgo de primer orden.

Es posible estudiar la tabla ordenada resultante y definir una línea de corte. La *línea de corte* (dibujada horizontalmente en algún punto de la tabla) implica que sólo los riesgos que se encuentran por arriba de la línea recibirán mayor atención. Los riesgos que caen por abajo de la línea se vuelven a valorar para lograr una priorización de segundo orden. En la figura 28.3, el impacto y la probabilidad del riesgo tienen una influencia distinta sobre la preocupación de la administración. Un factor de riesgo que tenga un alto impacto pero una muy baja probabilidad de ocurrencia no debe absorber una cantidad significativa de tiempo administrativo. Sin embargo, los riesgos de alto impacto con probabilidad moderada alta y los riesgos de bajo impacto con alta probabilidad deben someterse a los siguientes pasos del análisis de riesgos.

PUNTO CLAVE
Una tabla de riesgos se ordena por probabilidad e impacto para clasificar riesgos.

Todos los riesgos que se encuentran por arriba de la línea de corte deben manejarse. La columna marcada MMMR contiene un apuntador al *plan de mitigación, monitoreo y manejo de riesgo* o, alternativamente, en una colección de hojas de información de riesgo desarrolladas para todos los riesgos que se encuentran arriba del corte. El plan MMMR y las hojas de información de riesgo se estudian en las secciones 28.5 y 28.6.

Cita:

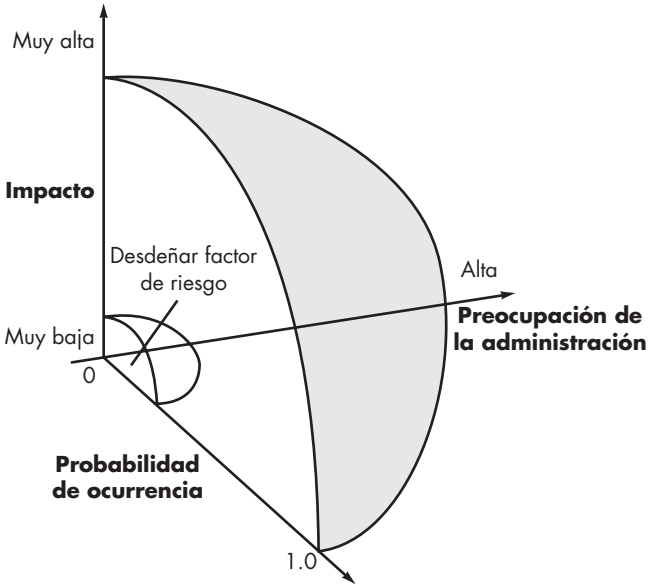
"[Hoy] nadie tiene el lujo de poder conocer una tarea tan bien como para que no contenga sorpresas, y las sorpresas significan riesgo."

Stephen Grey

La probabilidad del riesgo puede determinarse al hacer estimaciones individuales y luego desarrollar un solo valor de consenso. Aunque dicho enfoque es factible, se han desarrollado técnicas más sofisticadas para determinar la probabilidad del riesgo [AFC88]. Los promotores de riesgo pueden valorarse sobre una escala de probabilidad cualitativa que tenga los siguientes valores: imposible, improbable, probable y frecuente. Entonces puede asociarse probabilidad

3 Puede usar un promedio ponderado si un componente de riesgo tiene más significado para un proyecto.

FIGURA 28.3
Riesgo y preocupación de la administración



matemática con cada valor cualitativo (por ejemplo, una probabilidad de 0.7 a 0.99 implica un riesgo enormemente probable).

28.4.2 Valoración de impacto de riesgo

Tres factores afectan las probables consecuencias si ocurre un riesgo: su naturaleza, su ámbito y su temporización. La naturaleza del riesgo indica los problemas probables si ocurre. Por ejemplo, una interfaz externa pobremente definida en el hardware cliente (un riesgo técnico) impedirá el diseño y las pruebas tempranas, y probablemente conducirá más tarde a problemas de integración de sistema en un proyecto. El ámbito de un riesgo combina la severidad (¿cuán serio es?) con su distribución global (¿cuánto del proyecto se afectará o cuántos participantes se dañarán?). Finalmente, la temporización de un riesgo considera cuándo y por cuánto tiempo se sentirá el impacto. En la mayoría de los casos se quiere que las "malas noticias" ocurran tan pronto como sea posible, pero en algunos, mientras más se demoren, mejor.

? ¿Cómo se valoran las consecuencias de un riesgo?

Regrese una vez más al enfoque de análisis de riesgos que propuso la fuerza aérea estadounidense [AFC88]; puede aplicar los siguientes pasos para determinar las consecuencias globales de un riesgo: 1) determine la probabilidad promedio del valor de ocurrencia para cada componente de riesgo; 2) con la figura 28.1, determine el impacto para cada componente con base en los criterios mostrados, y 3) complete la tabla de riesgos y analice los resultados como se describe en las secciones anteriores.

La *exposición al riesgo* global, ER, se determina usando la siguiente relación [Hal98]:

$$\text{ER} = P \times C$$

donde P es la probabilidad de ocurrencia para un riesgo y C es el costo para el proyecto si ocurre el riesgo.

Por ejemplo, suponga que el equipo de software define un riesgo de proyecto en la forma siguiente:

Identificación de riesgo. De hecho, sólo 70 por ciento de los componentes de software calendarizados para reuso se integrarán en la aplicación. La funcionalidad restante tendrá que desarrollarse a la medida.

Probabilidad del riesgo. Un 80 por ciento (probable).

Impacto del riesgo. Se planificaron 60 componentes de software reutilizables. Si sólo puede usarse 70 por ciento, tendrán que desarrollarse 18 componentes desde cero (además de otro software a la medida que se calendarizó para desarrollo). Dado que el componente promedio es de 100 LOC y que los datos locales indican que el costo de la ingeniería del software para cada LOC es US$14.00, el costo global (impacto) para desarrollar los componentes sería $18 \times 100 \times 14 =$ US$25 200.

Exposición al riesgo. ER = $0.80 \times 25\ 200 \sim$ US$20 200.

Compare la ER para todos los riesgos con la estimación de costo para el proyecto. Si ER es mayor a 50 por ciento del costo del proyecto, debe evaluarse la viabilidad de éste.

La exposición al riesgo puede calcularse para cada riesgo en la tabla de riesgo, una vez hecha la estimación del costo del riesgo. La exposición al riesgo total para todos los riesgos (arriba del corte en la tabla de riesgos) puede proporcionar los medios para ajustar la estimación del costo final para un proyecto. También puede usarse para predecir el aumento probable en recursos de personal requeridos en varios puntos durante el calendario del proyecto.

La proyección del riesgo y las técnicas de análisis descritas en las secciones 28.4.1 y 28.4.2 se aplican de manera iterativa conforme avanza el proyecto de software. El equipo del proyecto debe revisar la tabla de riesgos a intervalos regulares, reevaluar cada riesgo para determinar cuándo nuevas circunstancias cambian su probabilidad e impacto. Como consecuencia de esta actividad, acaso sea necesario agregar nuevos riesgos a la tabla, eliminar algunos riesgos que ya no son relevantes e incluso cambiar las posiciones relativas de otros.

CasaSegura

Análisis de riesgos

La escena: Oficina de Doug Miller antes de comenzar el proyecto de software CasaSegura.

Participantes: Doug Miller (gerente del equipo de ingeniería del software CasaSegura) y Vinod Raman, Jamie Lazar y otros miembros del equipo de ingeniería de software del producto.

La conversación:

Doug: Me gustaría usar algo de tiempo en una lluvia de ideas para el proyecto CasaSegura.

Jamie: ¿Acerca de que puede salir mal?

Doug: Sip. Aquí hay algunas categorías donde las cosas pueden salir mal. [Muestra a todos las categorías anotadas en la introducción a la sección 28.3.]

Vinod: Hmmm... quieres que sólo las mencionemos o...

Doug: No. Esto es lo que creo que debemos hacer. Todo mundo haga una lista de riesgos... ahora...

[Transcurren diez minutos, todos escriben].

Doug: Muy bien, deténganse.

Jamie: ¡Pero no he terminado!

Doug: Está bien. Revisaremos la lista de nuevo. Ahora, para cada ítem en su lista, asignen un porcentaje de probabilidad de que ocurrirá el riesgo. Luego, asignen un impacto al proyecto sobre una escala de 1 (menor) a 5 (catastrófico).

Vinod: Si creo que el riesgo es un volado, especifico una probabilidad de 50 por ciento y, si creo que tendrá un impacto de proyecto moderado, especifico un 3, ¿cierto?

Doug: Exactamente.

[Transcurren cinco minutos, todos escriben].

Doug: Muy bien, deténganse. Ahora haremos una lista grupal en el pizarrón. Yo escribiré; cada uno de ustedes dirá una entrada de su lista.

[Transcurren quince minutos; crean la lista].

Jamie (apunta hacia el pizarrón y ríe): Vinod, ese riesgo (apunta hacia una entrada en el pizarrón) es ridículo. Hay una mayor probabilidad de que a todos nos caiga un rayo. Debemos removerlo.

Doug: No, dejémoslo por ahora. Consideremos todos los riesgos, sin importar cuán locos parezcan. Más tarde filtraremos la lista.

Jamie: Pero ya tenemos más de 40 riesgos... ¿cómo vamos a manejarlos todos?

Doug: No podemos. Es por eso por lo que definiremos un corte después de ordenarlos. Yo haré ese corte y nos reuniremos de nuevo mañana. Por ahora, regresen a trabajar... y en su tiempo libre piensen en cualquier riesgo que hayan olvidado.

28.5 REFINAMIENTO DEL RIESGO

Durante las primeras etapas de la planificación del proyecto, un riesgo puede enunciarse de manera muy general. Conforme pasa el tiempo y se aprende más acerca del proyecto y de los riesgos, es posible refinar el riesgo en un conjunto de riesgos más detallados, cada uno un poco más sencillo de mitigar, monitorear y manejar.

¿Cuál es una buena forma de describir un riesgo?

Una forma de hacer esto es representar el riesgo en formato *condición-transición-consecuencia* (CTC) [Glu94]. Es decir, el riesgo se enuncia en la forma siguiente:

> Dado que <condición> entonces hay preocupación porque (posiblemente) <consecuencia>.

Al usar el formato CTC para el riesgo de reutilización anotado en la sección 28.4.2, podría escribir:

> Dado que todos los componentes de software reutilizables deben apegarse a estándares de diseño específicos y dado que algunos no se apegan, entonces existe preocupación de que (posiblemente) sólo 70 por ciento de los módulos reutilizables planeados puedan realmente integrarse en el sistema que se va a construir, lo que da como resultado la necesidad de ingeniería a la medida del restante 30 por ciento de los componentes.

Esta condición general puede refinarse en la forma siguiente:

> **Subcondición 1.** Ciertos componentes reutilizables los desarrolló una tercera persona sin conocimiento de los estándares de diseño internos.
>
> **Subcondición 2.** El estándar de diseño para interfaces de componente todavía no se consolida y puede no apegarse a ciertos componentes reutilizables existentes.
>
> **Subcondición 3.** Ciertos componentes reutilizables se implementaron en un lenguaje que no se soporta en el entorno blanco.

Las consecuencias asociadas con estas subcondiciones refinadas permanecen iguales (es decir, 30 por ciento de componentes de software deben someterse a ingeniería a la medida), pero el refinamiento ayuda a aislar los riesgos subyacentes y puede conducir a análisis y respuestas más sencillos.

28.6 MITIGACIÓN, MONITOREO Y MANEJO DE RIESGO

Cita:

"Si tomo muchas precauciones, es porque no dejo nada al azar."

Napoleón

Todas las actividades de análisis de riesgos presentadas hasta el momento tienen una sola meta: auxiliar al equipo del proyecto a desarrollar una estrategia para lidiar con el riesgo. Una estrategia efectiva debe considerar tres temas: 1) evitar el riesgo, 2) monitorear el riesgo y 3) manejar el riesgo y planificar la contingencia.

Si un equipo de software adopta un enfoque proactivo ante el riesgo, evitarlo siempre es la mejor estrategia. Esto se logra desarrollando un plan para *mitigación del riesgo*. Por ejemplo, suponga que una alta rotación de personal se observa como un riesgo de proyecto r_1. Con base en la historia y la intuición administrativa, la probabilidad l_1 de alta rotación se estima en 0.70 (70 por ciento, más bien alta) y el impacto x_1 se proyecta como crítico, es decir, la alta rotación tendrá un impacto crítico sobre el costo y el calendario del proyecto.

Para mitigar este riesgo se desarrollará una estrategia a fin de reducir la rotación. Entre los posibles pasos por tomar están:

¿Qué puede hacerse para mitigar el riesgo?

- Reunirse con el personal actual para determinar las causas de la rotación (por ejemplo, pobres condiciones laborales, salario bajo, mercado laboral competitivo).
- Mitigar aquellas causas que están bajo su control antes de comenzar el proyecto.

- Una vez iniciado el proyecto, suponer que la rotación ocurrirá y desarrollar técnicas para asegurar la continuidad cuando el personal se vaya.
- Organizar equipos de trabajo de modo que la información acerca de cada actividad de desarrollo se disperse ampliamente.
- Definir estándares de producto operativo y establecer mecanismos para asegurar que todos los modelos y documentos se desarrollen en forma oportuna.
- Realizar revisiones de pares de todo el trabajo (de modo que más de una persona "se ponga al día").
- Asignar un miembro de personal de respaldo para cada técnico crítico.

Conforme avanza el proyecto, comienzan las actividades de *monitoreo de riesgos*. El gerente de proyecto monitorea factores que pueden proporcionar un indicio de si el riesgo se vuelve más o menos probable. En el caso de alta rotación de personal se monitorean: la actitud general de los miembros del equipo con base en presiones del proyecto, el grado en el que el equipo cuaja, relaciones interpersonales entre miembros del equipo, potenciales problemas con la compensación y beneficios, y la disponibilidad de empleos dentro de la compañía y fuera de ella.

Además de monitorear dichos factores, un gerente de proyecto debe dar seguimiento a la efectividad de los pasos de mitigación del riesgo. Por ejemplo, un paso de mitigación del riesgo anotado aquí requiere la definición de estándares de producto operativo y mecanismos para asegurarse de que los productos operativos se desarrollan en forma oportuna. Éste es un mecanismo para asegurar continuidad en caso de que un individuo crucial deje el proyecto. El gerente de proyecto debe monitorear los productos operativos cuidadosamente para asegurarse de que cada uno puede sostenerse por cuenta propia y que imparte información que sería necesaria si un recién llegado fuese forzado a unirse al equipo de software en alguna parte en medio del proyecto.

El *manejo del riesgo* y la *planificación de contingencia* suponen que los esfuerzos de mitigación fracasaron y que el riesgo se convirtió en realidad. Continuando con el ejemplo, el proyecto ya está en marcha y algunas personas anuncian que renunciarán al mismo. Si se siguió la estrategia de mitigación, está disponible el respaldo, la información se documentó y el conocimiento se dispersó a través del equipo. Además, puede cambiar temporalmente el foco de los recursos (y reajustar el calendario del proyecto) hacia aquellas funciones que tengan personal completo, lo que permitirá "ponerse al día" a los recién llegados que deban agregarse al equipo. A los individuos que se retiran se les pide detener todo el trabajo y pasar sus últimas semanas en "modo de transferencia de conocimiento". Esto puede incluir captura de conocimiento en video, desarrollo de "documentos comentados o wikis" y/o reuniones con otros miembros del equipo que permanecerán en el proyecto.

Si la ER para un riesgo específico es menor que el costo de mitigación de riesgo, no intente mitigar el riesgo, sino continuar para monitorearlo.

Es importante anotar que los pasos de mitigación, monitoreo y manejo del riesgo (MMMR) incurren en costos adicionales para el proyecto. Por ejemplo, emplear el tiempo en respaldar a cada técnico crucial cuesta dinero. Por tanto, parte del manejo de riesgos es evaluar cuándo los beneficios acumulativos por los pasos MMMR sobrepasan los costos asociados con su implementación. En esencia, se realiza un análisis clásico costo-beneficio. Si los pasos para evitar el riesgo debido a la alta rotación aumentarán tanto el costo del proyecto como la duración del mismo por un estimado de 15 por ciento, pero el factor de costo predominante es "respaldo", la administración puede decidir no implementar este paso. Por otra parte, si los pasos para evitar el riesgo se proyectan para aumentar los costos en 5 por ciento y la duración sólo en 3 por ciento, la administración probablemente pondrá todo en su lugar.

Para un proyecto grande pueden identificarse 30 o 40 riesgos. Si para cada uno se identifican entre tres y siete pasos de manejo de riesgo, ¡el manejo del riesgo puede convertirse en un proyecto por sí mismo! Por esta razón, debe adaptar al riesgo de software la regla de Pareto de 80-20. La experiencia indica que 80 por ciento del riesgo de proyecto global (es decir, 80 por

ciento del potencial para falla del proyecto) puede explicarse por sólo 20 por ciento de los riesgos identificados. El trabajo realizado durante los primeros pasos del análisis de riesgos ayudará a determinar cuáles de ellos residen en ese 20 por ciento (por ejemplo, riesgos que conducen a la exposición más alta al riesgo). Por esta razón, algunos de los riesgos identificados, valorados y proyectados pueden no llegar al plan MMMR, no se ubican en el crucial 20 por ciento (los riesgos con prioridad de proyecto más alta).

El riesgo no está limitado al proyecto de software en sí. Pueden ocurrir después de que el software se desarrolló exitosamente y de que se entregó al cliente. Dichos riesgos por lo general se asocian con las consecuencias de falla del software en el campo.

La *seguridad del software* y el *análisis de riesgos* (por ejemplo, [Dun02], [Her00], [Lev95]) son las actividades de aseguramiento de la calidad del software (capítulo 16) que se enfocan en la identificación y valoración de los riesgos potenciales que pueden afectar al software negativamente y hacer que falle todo un sistema. Si los riesgos pueden identificarse tempranamente en el proceso de ingeniería del software, pueden especificarse características de diseño del software que eliminarán o controlarán los riesgos potenciales.

28.7 El plan MMMR

En el plan de proyecto del software puede incluirse una estrategia de administración del riesgo, o los pasos de administración del riesgo pueden organizarse en un *plan de mitigación, monitoreo y manejo de riesgo* (MMMR) por separado. El plan MMMR documenta todo el trabajo realizado como parte del análisis de riesgos y el gerente del proyecto lo usa como parte del plan de proyecto global.

Algunos equipos de software no desarrollan un documento MMMR formal. En vez de ello, cada riesgo se documenta individualmente usando una *hoja de información de riesgo* (HIR) [Wil97]. En la mayoría de los casos, la HIR se mantiene con un sistema de base de datos de modo

HERRAMIENTAS DE SOFTWARE

Manejo de riesgo

Objetivo: El objetivo de las herramientas de manejo de riesgo es auxiliar de un equipo de proyecto para definir riesgos, valorar su impacto y probabilidad, y monitorear los riesgos a lo largo de un proyecto de software.

Mecánica: En general, las herramientas de manejo de riesgo auxilian en la identificación de riesgos genéricos al proporcionar una lista de riesgos empresariales y de proyecto usuales, proporcionar listas de verificación u otras técnicas de "entrevista" que auxilien en la identificación de riesgos específicos del proyecto, asignar probabilidad e impacto a cada riesgo, apoyar las estrategias de mitigación de riesgo y generar muchos reportes diferentes relacionados con el riesgo.

Herramientas representativas:[4]

@risk, desarrollada por Palisade Corporation (**www.palisade.com**), es una herramienta de análisis de riesgo genérico que usa simulación Monte Carlo para impulsar su motor analítico.

Riskman, distribuida por ABS Consulting (**www.absconsulting.com/riskmansoftware/index.html**), es un sistema experto de evaluación de riesgos que identifica riesgos relacionados con proyectos.

Risk Radar, desarrollada por SPMN (**www.spmn.com**), ayuda a los gerentes de proyecto a identificar y manejar riesgos de proyecto.

Risk+, desarrollada por Deltek (**www.deltek.com**), se integra con Microsoft Project para cuantificar incertidumbres de costo y calendario.

X:PRIMER, desarrollada por GrafP Technologies (**www.grafp.com**), es una herramienta genérica web que predice qué puede salir mal en un proyecto e identifica las causas raíz para potenciales fallos y contramedidas efectivas.

4 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas que hay en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

FIGURA 28.4

Hoja de información de riesgo.
Fuente: [Wil97].

Hoja de información de riesgo			
Riesgo ID: P02-4-32	Fecha: 5/9/09	Prob: 80%	Impacto: alto
Descripción: De hecho, sólo 70 por ciento de los componentes de software calendarizados para reuso se integrarán en la aplicación. La funcionalidad restante tendrá que desarrollarse a la medida.			
Refinamiento/contexto: Subcondición 1: Ciertos componentes reutilizables se desarrollaron por una tercera persona sin conocimiento de los estándares de diseño internos. Subcondición 2: El estándar de diseño para interfaces de componente no se consolidó y puede ser que no se apegue a ciertos componentes reutilizables existentes. Subcondición 3: Ciertos componentes reutilizables se implementaron en un lenguaje que no es soportado en el entorno meta.			
Mitigación/monitoreo: 1. Contactar tercera persona para determinar conformidad con los estándares de diseño. 2. Presionar por terminación de estándares de interfaz; considerar estructura de componente cuando se decida acerca de protocolo de interfaz. 3. Comprobar para determinar el número de componentes en la categoría de subcondición 3; comprobar para determinar si se puede adquirir soporte de lenguaje.			
Manejo/plan de contingencia/disparador: *ER* calculada en US$20 200. Asignar esta cantidad dentro de los costos de contingencia del proyecto. Desarrollar revisión de calendario y suponer que 18 componentes adicionales tendrán que construirse a la medida; asignar personal en concordancia. Disparador: Pasos de mitigación improductivos al 7/1/09.			
Estado actual: 5/12/09: Pasos de mitigación iniciados.			
Originador: D. Gagne		Asignado: B. Laster	

que la entrada de creación e información, el orden de prioridad, las búsquedas y otros análisis pueden realizarse con facilidad. El formato de la HIR se ilustra en la figura 28.4.

Una vez documentada la MMMR y comenzado el proyecto, inician los pasos de mitigación y monitoreo del riesgo. Como ya se estudió, la mitigación del riesgo es una actividad que busca evitar el problema. El monitoreo del riesgo es una actividad de seguimiento del proyecto con tres objetivos principales: 1) valorar si los riesgos predichos en efecto ocurren, 2) asegurar que los pasos para evitar el riesgo definidos para un riesgo determinado se aplican de manera correcta y 3) recopilar información que pueda usarse para futuros análisis de riesgos. En muchos casos, el problema que ocurre durante un proyecto puede monitorearse en más de un riesgo. Otra actividad del monitoreo de riesgos es intentar asignar orígenes (cuál riesgo causó cuál problema a lo largo del proyecto).

28.8 RESUMEN

Siempre que un colectivo cabalga en un proyecto de software, el sentido común dicta análisis de riesgos. E incluso así, la mayoría de los gerentes de proyectos de software lo hacen de manera informal y superficial, si acaso lo hacen. El tiempo que se emplea en identificar, analizar y manejar el riesgo rinde sus frutos en muchas formas: menos agitación durante el proyecto, una mayor capacidad para monitorear y controlar un proyecto, y la confianza que conlleva la planificación de los problemas antes de que se presenten.

El análisis de riesgos puede absorber una cantidad significativa del esfuerzo de planificación del proyecto. Identificación, proyección, valoración, manejo y monitoreo, todos requieren tiempo. Pero el esfuerzo vale la pena. Para citar a Sun Tzu, el general chino que vivió hace 2 500 años: "si conoces al enemigo y te conoces a ti mismo, no necesitas temer al resultado de cien batallas". Para el gerente de proyecto de software, el enemigo es el riesgo.

Problemas y puntos por evaluar

28.1. Proporcione cinco ejemplos de otros campos que ilustren los problemas asociados con una estrategia de riesgo reactiva.

28.2. Describa la diferencia entre "riesgos conocidos" y "riesgos predecibles".

28.3. Agregue tres preguntas o temas adicionales a cada una de las listas de comprobación de ítem de riesgo que se presentan en el sitio web de esta obra.

28.4. Se le pide construir software para apoyar un sistema de edición de video de bajo costo. El sistema acepta video digital como entrada, almacena el video en disco y luego permite al usuario aplicar una amplia variedad de ediciones al video digitalizado. Después, el resultado puede exhibirse mediante DVD u otros medios. Haga una pequeña cantidad de investigación acerca de sistemas de este tipo y luego elabore una lista de riesgos tecnológicos que enfrentaría mientras comienza un proyecto de este tipo.

28.5. Usted es el gerente de proyecto de una gran compañía de software. Se le pide dirigir un equipo que desarrolle software de procesamiento de palabra de "próxima generación". Cree una tabla de riesgo para el proyecto.

28.6. Describa la diferencia entre componentes de riesgo y promotores de riesgo.

28.7. Desarrolle una estrategia de mitigación de riesgo y actividades específicas de mitigación de riesgo para tres de los riesgos anotados en la figura 28.2.

28.8. Desarrolle una estrategia de monitoreo de riesgo y actividades específicas de monitoreo de riesgo para tres de los riesgos anotados en la figura 28.2. Asegúrese de identificar los factores que monitoreará para determinar si el riesgo se vuelve más o menos probable.

28.9. Desarrolle una estrategia de manejo de riesgo y actividades específicas de manejo de riesgo para tres de los riesgos anotados en la figura 28.2.

28.10. Intente refinar tres de los riesgos anotados en la figura 28.2 y luego cree hojas de información de riesgo para cada uno.

28.11. Represente tres de los riesgos anotados en la figura 28.2 usando un formato CTC.

28.12. Vuelva a calcular la exposición al riesgo que estudió en la sección 28.4.2 cuando costo/LOC es US$16 y la probabilidad es 60 por ciento.

28.13. ¿Puede pensar en una situación en la que un riesgo con alta probabilidad y alto impacto no se considerará como parte de su plan MMMR?

28.14. Describa las cinco áreas de aplicación de software en las que la seguridad del software y el análisis de riesgos serían una preocupación principal.

Lecturas y fuentes de información adicionales

La literatura de gestión del riesgo de software se expandió significativamente en las décadas anteriores. Vun (*Modeling Risk*, Wiley, 2006) presenta un tratamiento matemático detallado del análisis de riesgos que puede aplicarse a proyectos de software. Crohy *et al*. (*The Essentials of Risk Management,* McGraw-Hill, 2006), Mulcahy (*Risk Management, Tricks of the Trade for Project Managers,* RMC Publications, Inc., 2003), Kendrick (*Identifying and Managing Project Risk,* American Management Association, 2003), y Marrison (*The Fundamentals of Risk Measurement,* McGraw-Hill, 2002) presentan métodos y herramientas útiles que puede usar todo gerente de proyecto.

DeMarco y Lister (*Dancing with Bears*, Dorste House, 2003) escribieron un entretenido e inteligente libro que guía a los gerentes y profesionales del software a través de la gestión de riesgos. Moynihan (*Coping with IT/IS Risk Management*, Springer-Verlag, 2002) presenta consejos pragmáticos de gerentes de proyecto que lidian con el riesgo continuamente. Royer (*Project Risk Management*, Management Concepts, 2002) y Smith y Merrit (*Proactive Risk Management*, Productivity Press, 2002) sugieren un proceso proactivo para gestión del riesgo. Karolak (*Software Engineering Risk Management*, Wiley, 2002) escribió un manual que introduce un modelo de análisis de riesgo fácil de usar, con valiosas listas de comprobación y cuestionarios apoyados por un paquete de software.

Capers Jones (*Assesment and Control of Software Risks*, Prentice Hall, 1994) presenta un detallado análisis de los riesgos de software, que incluye datos recopilados de cientos de proyectos de software. Jones define 60 factores de riesgo que pueden afectar el resultado de los proyectos de software. Boehm [Boe89] sugiere excelentes cuestionarios y formatos de lista de verificación que pueden resultar invaluables en la identificación del riesgo. Charette [Cha89] presenta un tratamiento detallado de la mecánica del análisis de riesgos, y se apoya en teoría de probabilidad y técnicas estadísticas para analizar los riesgos. En otro volumen, Charette (*Application Strategies for Risk Analysis*, McGraw-Hill, 1990) analiza el riesgo en el contexto de la ingeniería de sistemas y de software, y sugiere estrategias pragmáticas para la gestión del riesgo. Gilb (*Principles of Software Engineering Management*, Addison-Wesley, 1988) presenta un conjunto de "principios" (que con frecuencia son entretenidos y en ocasiones profundos) que pueden servir como una valiosa guía para la gestión del riesgo.

Ewusi-Mensah (*Software Development Failures: Anatomy of Abandoned Projects*, MIT Press, 2003) y Yourdon (*Death March*, Prentice Hall, 1997) estudian lo que ocurre cuando los riesgos abruman a un equipo de proyecto de software. Bernstein (*Against the Gods*, Wiley, 1998) presenta una entretenida historia del riesgo, que se remonta a tiempos antiguos.

El Software Engineering Institute publicó muchos reportes detallados y manuales acerca del análisis y la gestión del riesgo. El panfleto AFSCP 800-45 del Air Force Systems Command [AFC88] describe la identificación del riesgo y técnicas para su reducción. Cada tema del *ACM Software Engineering Notes* tiene una sección titulada "Riesgos para el público" (editor, P. G. Neumann). Si quiere las más recientes y mejores historias de horror del software, éste es el lugar al que debe ir.

En internet está disponible una gran variedad de fuentes de información acerca de la gestión del riesgo de software. Una lista actualizada de referencias en la World Wide Web que son relevantes para la gestión del riesgo puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

29

MANTENIMIENTO Y REINGENIERÍA

CONCEPTOS CLAVE

análisis de inventarios 662
ingeniería hacia adelante . . . 669
ingeniería inversa 664
  datos 665
  procesamiento 666
  interfaces de usuario 667
mantenibilidad 657
mantenimiento de software . 656
reestructuración 668
  código 668
  datos 668
reestructuración de documentos 662
reingeniería de procesos de empresa (RPE) 658
reingeniería de software . . . 661
soportabilidad. 657

Sin importar su dominio de aplicación, su tamaño o su complejidad, el software de computadora evolucionará con el tiempo. El cambio impulsa este proceso. Para el software de computadora, el cambio ocurre cuando se corrigen los errores, cuando el software se adapta a un nuevo entorno, cuando el cliente solicita nuevas características o funciones y cuando la aplicación se somete a reingeniería para ofrecer beneficio en un contexto moderno. Durante los pasados 30 años, Manny Lehman [por ejemplo, Leh97a] y sus colaboradores realizaron análisis detallados de software de grado industrial y de sistemas con la intención de desarrollar una *teoría unificada para evolución del software*. Los detalles de este trabajo están más allá del ámbito de este libro, pero vale la pena destacar las leyes subyacentes derivadas de ella [Leh97b]:

Ley de cambio continuo (1974): El software que se implementó en un contexto de cómputo del mundo real y que, por tanto, evolucionará con el tiempo (llamados *sistemas tipo E*) debe adaptarse continuamente o de otro modo se volverá progresivamente menos satisfactorio.

Ley de complejidad creciente (1974): Conforme un sistema tipo E evoluciona, su complejidad aumenta, a menos que se haga trabajo para mantenerlo o reducirlo.

Ley de autorregulación (1974): El proceso de evolución del sistema tipo E es autorregulable con medidas de distribución de producto y de proceso cercanas a lo normal.

UNA MIRADA RÁPIDA

¿Qué es? Considere cualquier producto tecnológico que le haya funcionado bien. Lo usa con regularidad, pero está envejeciendo. Se descompone con frecuencia, se tarda más tiempo en reparar del que quisiera y ya no representa a la más reciente tecnología. ¿Qué hacer? Durante algún tiempo intenta repararlo, parcharlo, incluso extiende su funcionalidad. A esto se le llama mantenimiento. Pero éste se vuelve cada vez más difícil conforme pasan los años. Llega un momento en el que necesitará reconstruirlo. Creará un producto con funcionalidad agregada, mejor desempeño y confiabilidad, así como mantenibilidad mejorada. A eso se le llama reingeniería.

¿Quién lo hace? En el nivel de la organización, el mantenimiento lo realiza el personal de apoyo que es parte de la organización de ingeniería de software. La reingeniería la realizan especialistas en negocios (con frecuencia compañías consultoras). En el nivel de software, la reingeniería la realizan ingenieros de software.

¿Por qué es importante? Al vivir en un mundo que cambia rápidamente, las demandas sobre las funciones empresariales y la tecnología de la información que las apoyan cambian a un paso que pone enorme presión competitiva sobre toda organización comercial. Por esto, el software debe mantenerse continuamente y, en el momento adecuado, someterse a reingeniería para sostener el paso.

¿Cuáles son los pasos? El mantenimiento corrige los defectos, adapta el software para satisfacer un entorno cambiante y mejorar la funcionalidad a fin de cubrir las necesidades evolutivas de los clientes. Estratégicamente, la reingeniería de procesos de empresa (RPE) define las metas empresariales, identifica y evalúa los procesos empresariales existentes y crea procesos empresariales revisados que satisfacen mejor las metas del momento. La reingeniería de software abarca análisis de inventarios, restructuración de documentos, ingeniería inversa, reestructuración de programas y datos e ingeniería hacia adelante. La intención de dichas actividades es crear versiones de los programas existentes que muestren mayor calidad y mejor mantenibilidad.

¿Cuál es el producto final? Se producen varios productos operativos de mantenimiento y reingeniería (por ejemplo, casos de uso, modelos de análisis y diseño, procedimientos de prueba). El resultado final es actualización de software.

¿Cómo me aseguro de que lo hice bien? Con el uso de las mismas prácticas SQA que se aplican en todo proceso de ingeniería de software: revisiones técnicas para valorar los modelos de análisis y diseño; revisiones especializadas para considerar aplicabilidad y compatibilidad empresarial; y aplicación de pruebas para descubrir errores en contenido, funcionalidad e interoperabilidad.

Ley de conservación de estabilidad organizativa (1980): La tasa de actividad global efectiva promedio en un sistema tipo E en evolución no varía durante el tiempo de vida del producto.

Ley de conservación de familiaridad (1980): Conforme un sistema tipo E evoluciona, todo lo asociado con él: desarrolladores, personal de ventas, usuarios, etc., deben mantener el dominio de su contenido y comportamiento para lograr evolución satisfactoria. El crecimiento excesivo disminuye dicho dominio. Por tanto, el crecimiento incremental promedio permanece sin variación conforme el sistema evoluciona.

¿Cómo evolucionan los sistemas heredados conforme pasa el tiempo?

Ley de crecimiento continuo (1980): El contenido funcional de los sistemas tipo E debe aumentar continuamente para mantener la satisfacción del usuario durante su tiempo de vida.

Ley de declive de la calidad (1996): La calidad de los sistemas tipo E declinará, a menos que se mantengan y adapten rigurosamente a los cambios del entorno operativo.

Ley de realimentación del sistema (1996): Los procesos evolutivos tipo E constituyen sistemas de realimentación multinivel, multibucle y multiagente, y deben tratarse como tales para lograr mejora significativa sobre cualquier base razonable.

Las leyes que definieron Lehman y sus colegas son parte inherente de una realidad de la ingeniería de software. En este capítulo se estudia el reto del mantenimiento del software y las actividades de reingeniería que se requieren para extender la vida efectiva de los sistemas heredados.

29.1 MANTENIMIENTO DE SOFTWARE

Éste comienza casi de inmediato. El software se libera a los usuarios finales y, en cuestión de días, los reportes de errores se filtran de vuelta hacia la organización de ingeniería de software. En semanas, una clase de usuarios indica que el software debe cambiarse de modo que pueda ajustarse a las necesidades especiales de su entorno. Y en meses, otro grupo corporativo, que no quería saber nada del software cuando se liberó, ahora reconoce que puede ofrecerle beneficios inesperados. Necesitará algunas mejoras para hacer que funcione en su mundo.

El reto del mantenimiento del software comienza. Uno se enfrenta con una creciente lista de corrección de errores, peticiones de adaptación y mejoras categóricas que deben planearse, calendarizarse y, a final de cuentas, lograrse. Mucho antes, la fila creció bastante y el trabajo que implica amenaza con abrumar los recursos disponibles. Conforme pasa el tiempo, la organización descubre que emplea más dinero y tiempo en mantener los programas existentes que en someter a ingeniería nuevas aplicaciones. De hecho, no es raro que una organización de software emplee entre 60 y 70 por ciento de todos sus recursos en mantenimiento del software.

Acaso el lector pregunte por qué se requiere tanto mantenimiento y por qué se emplea tanto esfuerzo. Osborne y Chikofsky [Osb90] proporcionan una respuesta parcial:

> Mucho del software del que dependemos en la actualidad tiene en promedio una antigüedad de 10 a 15 años. Aun cuando dichos programas se crearon usando las mejores técnicas de diseño y codificación conocidas en la época [y muchas no lo fueron], se produjeron cuando el tamaño del programa y el espacio de almacenamiento eran las preocupaciones principales. Luego migraron a nuevas plataformas, se ajustaron para cambios en máquina y tecnología de sistema operativo, y aumentaron para satisfacer las necesidades de los nuevos usuarios, todo sin suficiente preocupación por la arquitectura global. El resultado es estructuras pobremente diseñadas, pobre codificación, pobre lógica y pobre documentación de los sistemas de software que ahora debemos seguir usando...

Otra razón del problema del mantenimiento del software es la movilidad del personal. Es probable que el equipo (o la persona) de software que hizo el trabajo original ya no esté más por ahí. Peor aún, otras generaciones de personal de software modificaron el sistema y se mudaron.

Y puede ser que ya no quede alguien que tenga algún conocimiento directo del sistema heredado.

Como se anotó en el capítulo 22, la naturaleza ubicua del cambio subyace a todo el trabajo del software. El cambio es inevitable cuando se construyen sistemas basados en computadoras; por tanto, deben desarrollarse mecanismos para evaluar, controlar y realizar modificaciones.

Cita:

"Mantenibilidad y comprensión de un programa son conceptos paralelos: mientras más difícil sea entender un programa, más difícil será darle mantenimiento."

Gerald Berns

A lo largo de este libro se enfatiza la importancia de entender el problema (análisis) y de desarrollar una solución bien estructurada (diseño). De hecho, la parte 2 del libro se dedica a la mecánica de tales acciones de ingeniería de software y la 3 se enfoca en las técnicas requeridas para asegurarse de que se hicieron correctamente. Análisis y diseño conducen a una importante característica del software que se llamará mantenibilidad. En esencia, la *mantenibilidad* es un indicio cualitativo[1] de la facilidad con la que el software existente puede corregirse, adaptarse o aumentarse. Gran parte de lo que trata la ingeniería de software es acerca de la construcción de sistemas que muestren alta mantenibilidad.

¿Pero qué es mantenibilidad? El software mantenible muestra modularidad efectiva (capítulo 8). Usa patrones de diseño (capítulo 12) que permiten facilidad de comprensión. Se construyó con estándares y convenciones de codificación bien definidos, que conducen a código fuente autodocumentable y comprensible. Experimentó varias técnicas de aseguramiento de calidad (parte 3 de este libro) que descubrieron potenciales problemas de mantenimiento antes de que el software se liberara. Fue creado por ingenieros de software que reconocen que acaso ya no estén presentes cuando deban realizarse cambios. En consecuencia, el diseño y la implementación del software debe "auxiliar" a la persona que realice el cambio.

29.2 Soportabilidad del software

Con la finalidad de dar soporte efectivo al software de grado industrial, su organización (o su encargado) deben poder realizar las correcciones, adaptaciones y mejoras que son parte de la actividad de mantenimiento. Pero, además, la organización debe proporcionar otras importantes actividades de soporte que incluyen soporte operativo en marcha, soporte al usuario final y actividades de reingeniería durante el ciclo de vida completo del software. Una definición razonable de *soportabilidad del software* es

> … la capacidad de dar soporte a un sistema de software durante toda la vida del producto. Esto implica satisfacer cualquier necesidad o requisito, pero también provisión de equipo, infraestructura de soporte, software adicional, instalaciones, mano de obra o cualquier otro recurso requerido para mantener el software operativo y capaz de satisfacer su función [SSO08].

En esencia, la soportabilidad es uno de los muchos factores de calidad que deben considerarse durante las acciones de análisis y diseño que son parte del proceso de software. Deben abordarse como parte del modelo (o especificación) de requisitos y considerarse conforme el diseño evoluciona y comienza la construcción.

WebRef

En **www.software-supportability.org/Downloads.html**, puede encontrar una amplia variedad de documentos descargables acerca de soportabilidad del software.

Por ejemplo, la necesidad de software "antierrores" en el nivel de componente y código se estudió anteriormente en este libro. El software debe contener facilidades para auxiliar al personal de apoyo cuando se encuentre un defecto en el entorno operativo (y de no cometer equívocos, se *encontrarán* los defectos). Además, el personal de apoyo debe tener acceso a una base de datos que contenga registros de todos los defectos que ya se encontraron: sus características, causas y cura. Esto permitirá al personal de apoyo examinar defectos "similares" y poder brindar un medio para un diagnóstico y corrección más rápidos.

---

1 Existen algunas medidas cuantitativas que proporcionan un indicio indirecto de la mantenibilidad (ver, por ejemplo, [Sch99], [SEI02]).

Aunque los defectos encontrados en una aplicación son un tema de soporte crucial, la soportabilidad también demanda que se proporcionen recursos para dar soporte diario a los conflictos del usuario final. La labor del personal de soporte al usuario final es responder las consultas del usuario acerca de instalación, operación y uso de la aplicación.

29.3 Reingeniería

En un artículo fundamental escrito para *Harvard Business Review*, Michael Hammer [Ham90] sienta las bases para una revolución en el pensamiento administrativo acerca de los procesos empresariales y la computación:

> Es momento de dejar de pavimentar el camino de las vacas. En lugar de incrustar procesos caducos en silicio y software, debemos eliminarlos y comenzar de nuevo. Debemos "reingeniar" nuestras empresas: usar el poder de la moderna tecnología de la información para rediseñar radicalmente nuestros procesos empresariales con la finalidad de lograr mejoras dramáticas en su rendimiento.
>
> Toda compañía opera de acuerdo con un gran número de reglas desarticuladas [...] La reingeniería lucha por liberarse de las antiguas reglas acerca de cómo organizarnos y dirigir nuestros negocios.

Cita:

"Enfrentar el mañana con el pensamiento puesto en usar los métodos de ayer es vislumbrar la vida paralizado."

James Bell

Como toda revolución, el llamado a las armas de Hammer resultó en cambios tanto positivos como negativos. Durante los años de 1990, algunas compañías hicieron un esfuerzo legítimo por someterse a reingeniería y los resultados condujeron a mejora competitiva. Otros se apoyaron exclusivamente en reducción y subcontratación (en lugar de en reingeniería) para mejorar su línea de referencia. Con frecuencia resultaron organizaciones "medias" con poco potencial para crecimiento futuro [DeM95a].

Hacia finales de la primera década del siglo XXI, la publicidad relacionada con la reingeniería disminuyó, pero el proceso en sí continúa en las compañías grandes y pequeñas. El nexo entre reingeniería empresarial e ingeniería de software yace en una "visión de sistema".

Con frecuencia, el software es la realización de las reglas empresariales que Hammer analiza. En la actualidad, las principales compañías tienen decenas de miles de programas de cómputo que soportan las "antiguas reglas empresariales". Conforme los administradores trabajan para modificar las reglas a fin de lograr mayor efectividad y competitividad, el software debe seguirles el paso. En algunos casos, esto significa la creación de grandes y novedosos sistemas basados en cómputo.[2] Pero en muchos otros, significa la modificación o reconstrucción de las aplicaciones existentes.

En las secciones que siguen, se examina la reingeniería en una forma descendente, comenzando con un breve panorama de la reingeniería de los procesos de empresas y avanzando hacia un análisis más detallado de las actividades técnicas que ocurren cuando el software se somete a reingeniería.

29.4 Reingeniería de procesos de empresa

PUNTO CLAVE

La RPE con frecuencia da como resultado nueva funcionalidad de software, mientras que la reingeniería de software trabaja para sustituir la funcionalidad de software existente con software mejor y más mantenible.

La *reingeniería de procesos de empresa* (RPE) se extiende más allá del ámbito de las tecnologías de la información y de la ingeniería de software. Entre las muchas definiciones (la mayoría un tanto abstractas) que se han sugerido para la RPE, está una publicada en *Fortune Magazine* [Ste93]: "la búsqueda, e implementación, de cambios radicales en los procesos de las empresas para lograr resultados innovadores". ¿Pero cómo se realiza la búsqueda y cómo se logra la implementación? Más importante, ¿cómo puede asegurarse que el "cambio radical" sugerido conducirá realmente a "resultados innovadores" en lugar de a caos organizacional?

2 La explosión de las aplicaciones y sistemas basados en web es indicativo de esta tendencia.

29.4.1 Procesos empresariales

Un proceso empresarial es "un conjunto de tareas lógicamente relacionadas, que se realizan para lograr un resultado empresarial definido" [Dav90]. Dentro del proceso empresarial, personal, equipo, recursos materiales y procedimientos empresariales se combinan para producir un resultado específico. Los ejemplos de procesos empresariales incluyen diseñar un nuevo producto, comprar servicios y suministros, contratar un nuevo empleado y pagar a proveedores. Cada uno demanda un conjunto de tareas y usa diversos recursos dentro de la empresa.

Todo proceso empresarial tiene un cliente definido: una persona o grupo que recibe el resultado (por ejemplo, una idea, un reporte, un diseño, un servicio, un producto). Además, los procesos empresariales atraviesan las fronteras de la organización. Requieren que diferentes grupos organizativos participen en las "tareas lógicamente relacionadas" que definen el proceso.

CONSEJO

Como ingeniero de software, su trabajo se realiza en el fondo de esta jerarquía. Sin embargo, asegúrese de que alguien considera seriamente los niveles superiores. Si esto no se ha hecho, su trabajo está en riesgo.

Todo sistema es en realidad una jerarquía de subsistemas. Una empresa no es la excepción. Toda la empresa está segmentada en la forma siguiente:

Empresa → sistemas empresariales → procesos empresariales → subprocesos empresariales

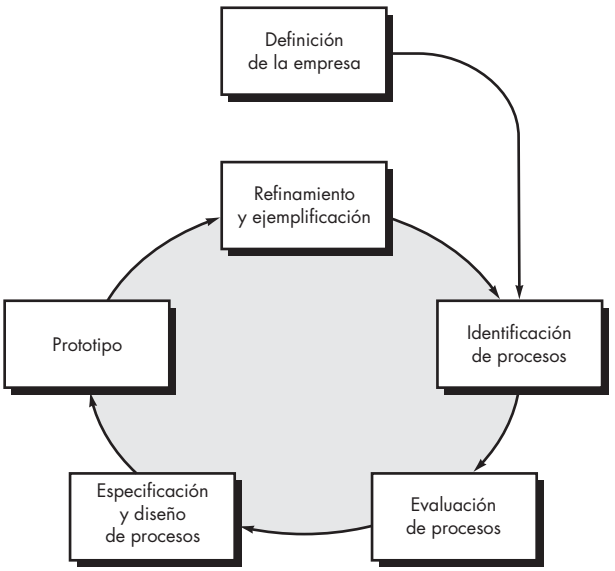
Cada sistema empresarial (también llamado *función empresarial*) está compuesto de uno o más procesos empresariales, y cada proceso empresarial se define mediante un conjunto de subprocesos.

La RPE puede aplicarse en cualquier nivel de la jerarquía, pero conforme se ensancha su ámbito (es decir, conforme se avanza hacia arriba en la jerarquía), los riesgos asociados con la RPE crecen de manera dramática. Por esta razón, la mayoría de los esfuerzos RPE se enfocan en procesos o subprocesos individuales.

29.4.2 Un modelo RPE

Como la mayoría de las actividades de ingeniería, la reingeniería de procesos de empresa es iterativa. Las metas de la empresa y los procesos que los logran deben adaptarse a un entorno empresarial cambiante. Por esta razón, no hay inicio ni fin de la RPE: es un proceso evolutivo. En la figura 29.1 se muestra un modelo para reingeniería de proceso de empresa. El modelo define seis actividades:

FIGURA 29.1
Un modelo RPE



Definición de la empresa. Las metas de la empresa se identifican dentro del contexto de cuatro motores clave: reducción de costo, reducción de tiempo, mejora de la calidad y desarrollo y fortalecimiento del personal. Las metas pueden definirse para toda la empresa o para un componente específico de ella.

Identificación de procesos. Se identifican los procesos que son cruciales para lograr las metas definidas en la definición de la empresa. Luego pueden clasificarse por importancia, por necesidad para cambiar o en cualquier otra forma que sea adecuada para la actividad de reingeniería.

Evaluación de procesos. Los procesos existentes se analizan y miden ampliamente. Se identifican las tareas de proceso, se anotan los costos y el tiempo consumido por ellas y se aíslan los problemas de calidad/desempeño.

Especificación y diseño de procesos. Con base en la información obtenida durante las primeras tres actividades RPE, se preparan casos de uso (capítulos 5 y 6) para cada proceso que deba someterse a reingeniería. Dentro del contexto de la RPE, los casos de uso identifican un escenario que entrega algún resultado a un cliente. Con el caso de uso como la especificación del proceso, se diseña un nuevo conjunto de tareas para el proceso.

Cita:

"Tan pronto como algo viejo se nos muestra en forma novedosa, nos pacificamos."

F. W. Nietzsche

Prototipo. Un proceso empresarial rediseñado debe convertirse en prototipo antes de que se integre plenamente en la empresa. Esta actividad "pone a prueba" el proceso, de modo que puedan hacerse refinamientos.

Refinamiento y ejemplificación. Con base en la realimentación del prototipo, el proceso empresarial se refina y luego se ejemplifica dentro de un sistema empresarial.

En ocasiones, dichas actividades RPE se usan en conjunto con herramientas de análisis de flujo de trabajo. La intención de dichas herramientas es construir un modelo del flujo de trabajo existente con la intención de analizar mejor los procesos actuales.

HERRAMIENTAS DE SOFTWARE

Reingeniería de procesos de empresa (RPE)

Objetivo: El objetivo de las herramientas RPE es dar soporte al análisis y valoración de los procesos empresariales existentes, y a la especificación y diseño de nuevos.

Mecánica: La mecánica de las herramientas varía. En general, las herramientas RPE permiten a un analista empresarial modelar los procesos empresariales existentes con la intención de valorar las ineficiencias del flujo de trabajo o problemas funcionales. Una vez identificados los problemas existentes, las herramientas permiten el análisis para crear prototipos y/o simular procesos empresariales revisados.

Herramientas representativas:[3]

Extend, desarrollada por ImagineThat, Inc. (**www.imaginethatinc.com**), es una herramienta de simulación para modelar procesos existentes y explorar nuevos. Extend proporciona abundante capacidad "y si…", que permite a un analista empresarial explorar diferentes escenarios de proceso.

e-Work, desarrollada por Metastorm (**www.metastorm.com**), proporciona soporte de gestión de procesos empresariales para procesos tanto manuales como automatizados.

IceTools, desarrollada por Blue Ice (**www.blueice.com**), es una colección de plantillas RPE para Microsoft Office y Microsoft Project.

SpeeDev, desarrollada por SpeeDev Inc. (**www.speedev.com**), es una de muchas herramientas que permiten a una organización modelar flujo de trabajo de procesos (en este caso, flujo de trabajo IT).

Workflow tool suite, desarrollada por MetaSoftware (**www.metasoftware.com**), incorpora una suite de herramientas para modelado, simulación y calendarización de flujo de trabajo.

Una útil lista de ligas a herramientas RPE puede encontrarse en **www.opfro.org/index.html?Components/Producers/Tools/BusinessProcessReengineeringTools.html~Contents**

3 Las herramientas que se mencionan aquí no representan un respaldo, sino que son una muestra de las herramientas que hay en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

29.5 Reingeniería de software

El escenario es demasiado común: una aplicación que atendió las necesidades empresariales de una compañía durante 10 o 15 años. Durante ese tiempo se corrigió, adaptó y mejoró muchas veces. Las personas realizaban esta tarea con las mejores intenciones, pero las buenas prácticas de ingeniería siempre se hicieron a un lado (debido a la presión de otros asuntos). Ahora la aplicación es inestable. Todavía funciona, pero cada vez que se intenta un cambio, ocurren inesperados y serios efectos colaterales. Aun así, la aplicación debe seguir evolucionando. ¿Qué hacer?

El software sin mantenimiento no es un problema nuevo. De hecho, el énfasis ampliado acerca de la reingeniería de software se produjo por los problemas de mantenimiento de software que se acumularon durante más de cuatro décadas.

29.5.1 Un modelo de proceso de reingeniería de software

La reingeniería toma tiempo, cuesta cantidades significativas de dinero y absorbe recursos que de otro modo pueden ocuparse en preocupaciones inmediatas. Por todas estas razones, la reingeniería no se logra en pocos meses o incluso en algunos años. La reingeniería de los sistemas de información es una actividad que absorberá recursos de tecnología de la información durante muchos años. Por esto, toda organización necesita una estrategia pragmática para la reingeniería de software.

WebRef

Una excelente fuente de información acerca de la reingeniería de software puede encontrarse en **reengineer.org**

Una estrategia factible se contempla en un modelo de proceso de reingeniería. Más tarde, en esta sección, se estudiará el modelo, pero primero se presentan algunos principios básicos.

La reingeniería es una actividad de reconstrucción. Para entenderla mejor, piense en una actividad análoga: la reconstrucción de una casa. Considere la siguiente situación. Usted compra una casa en otro estado. En realidad nunca ha visto la propiedad, pero la adquirió a un precio sorprendentemente bajo, con la advertencia de que es posible que deba reconstruirla por completo. ¿Cómo procedería?

- Antes de comenzar a reconstruir, parecería razonable inspeccionar la casa. Para determinar si necesita reconstruirse, usted (o un inspector profesional) crea una lista de criterios, de modo que su inspección sea sistemática.
- Antes de demoler y reconstruir toda la casa, asegúrese de que la estructura es débil. Si la casa es estructuralmente sólida, acaso sea posible "remodelar" sin reconstruir (a un costo mucho más bajo y en mucho menos tiempo).
- Antes de comenzar a reconstruir, asegúrese de entender cómo se construyó la original. Eche un vistazo detrás de las paredes. Entienda cómo están el alambrado, la plomería y la estructura interna. Incluso si tira todo a la basura, la comprensión que obtenga le servirá cuando comience la construcción.
- Si comienza a reconstruir, use solamente los materiales más modernos y más duraderos. Esto puede costar un poco más ahora, pero le ayudará a evitar costos y tardados mantenimientos posteriores.
- Si decide reconstruir, sea disciplinado en ello. Use prácticas que resultarán en alta calidad, hoy y en el futuro.

Aunque estos principios se enfocan en la reconstrucción de una casa se aplican igualmente bien a la reingeniería de los sistemas y aplicaciones basados en cómputo.

Para implementar estos principios puede usar un modelo de proceso de reingeniería de software que defina seis actividades, como se muestra en la figura 29.2. En algunos casos, dichas actividades ocurren en secuencia lineal, aunque no siempre. Por ejemplo, es posible que se tenga que recurrir a la ingeniería inversa (comprender el funcionamiento interno de un programa) antes de que pueda comenzar la reestructuración de documentos.

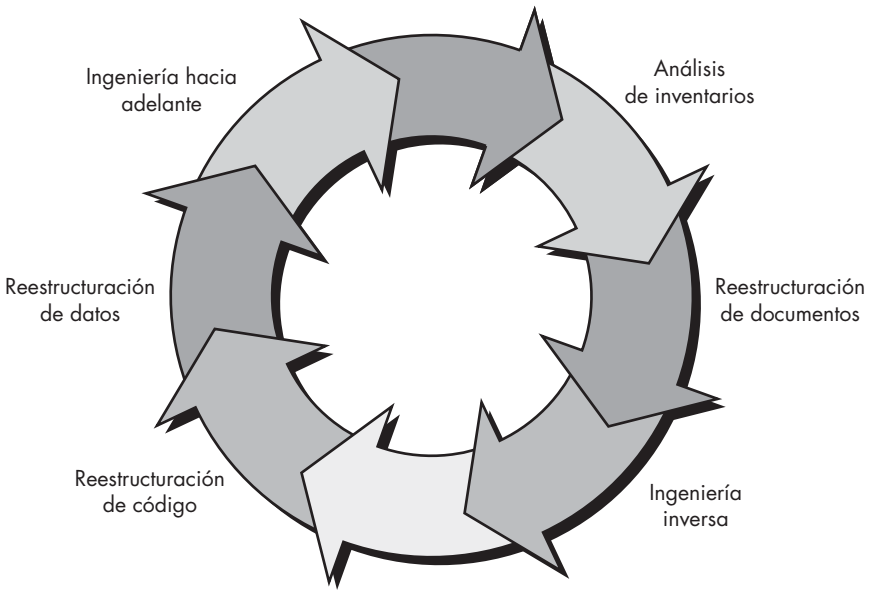


FIGURA 29.2
Modelo de proceso de reingeniería de software

CONSEJO

Si el tiempo y los recursos son escasos, puede considerar la aplicación del principio de Pareto al software que se someterá a reingeniería. Aplique el proceso de reingeniería al 20 por ciento del software que represente el 80 por ciento de los problemas.

29.5.2 Actividades de reingeniería de software

El paradigma de reingeniería que se muestra en la figura 29.2 es un modelo cíclico. Esto significa que cada una de las actividades presentadas como parte del paradigma puede revisarse. Para algún ciclo particular, el proceso puede terminar después de cualquiera de estas actividades.

Análisis de inventarios. Toda organización de software debe tener un inventario de todas las aplicaciones. El inventario puede ser nada más que un modelo de hojas de cálculo que contenga información que ofrezca una descripción detallada (por ejemplo, tamaño, edad, importancia empresarial) de cada aplicación activa. Al ordenar esta información de acuerdo con importancia empresarial, longevidad, mantenibilidad actual, soportabilidad y otros importantes criterios locales, aparecen los candidatos para reingeniería. Entonces pueden asignarse recursos a esas aplicaciones.

CONSEJO

Cree sólo tanta documentación como necesite para entender el software, ni una página más.

Es importante observar que el inventario debe revisarse con regularidad. El estado de las aplicaciones (por ejemplo, importancia empresarial) puede cambiar con el tiempo y, como resultado, cambiarán las prioridades para aplicar la reingeniería.

Reestructuración de documentos. La documentación débil es el distintivo de muchos sistemas heredados. Pero, ¿qué puede hacer con ella? ¿Cuáles son sus opciones?

1. *La creación de documentación consume demasiado tiempo*. Si el sistema funciona puede elegir vivir con lo que tiene. En algunos casos, éste es el enfoque correcto. No es posible volver a crear documentación para cientos de programas de cómputo. Si un programa es relativamente estático, se aproxima al final de su vida útil y es improbable que experimente cambio significativo, ¡déjelo así!
2. *La documentación debe actualizarse, pero su organización tiene recursos limitados.* Use un enfoque "documente cuando toque". Acaso no sea necesario volver a documentar por completo una aplicación. En vez de ello, aquellas porciones del sistema que en el momento experimenten cambio se documentan por completo. Con el tiempo, evolucionará una colección de documentación útil y relevante.
3. *El sistema tiene importancia empresarial y debe volver a documentarse por completo.* Incluso en este caso, un enfoque inteligente es recortar la documentación a un mínimo esencial.

Cada una de estas opciones es viable. Su organización de software debe elegir aquella que sea más adecuada para cada caso.

Ingeniería inversa. El término *ingeniería inversa* tiene su origen en el mundo del hardware. Una compañía desensambla un producto de hardware de otra empresa con la intención de entender los "secretos" de diseño y fabricación de su competidor. Dichos secretos podrían entenderse fácilmente si se obtuvieran las especificaciones de diseño y fabricación. Pero esos documentos son propiedad de la empresa competidora y no están disponibles para la compañía que hace la ingeniería inversa. En esencia, la ingeniería inversa exitosa deriva en una o más especificaciones de diseño y fabricación para un producto al examinar especímenes reales del mismo.

WebRef

En **www.comp.lancs.ac.uk/projects/RenaissanceWeb/**, puede encontrar varios recursos para la comunidad de reingeniería.

La ingeniería inversa para el software es muy similar. No obstante, en la mayoría de los casos, el programa que se va a someter a ingeniería inversa no es de un competidor: es el propio trabajo de la compañía (con frecuencia, elaborado muchos años atrás). Los "secretos" por entender son oscuros porque jamás se desarrollaron especificaciones. Por tanto, la ingeniería inversa para software es el proceso de analizar un programa con la intención de crear una representación del mismo en un nivel superior de abstracción que el código fuente. La ingeniería inversa es un proceso de *recuperación de diseño*. Las herramientas de ingeniería inversa extraen información de diseño de datos, arquitectónico y procedimental de un programa existente.

Reestructuración de código. El tipo más común de reingeniería (en realidad, en este caso es cuestionable el uso del término reingeniería) es la *reestructuración de código*.[4] Algunos sistemas heredados tienen una arquitectura de programa relativamente sólida, pero los módulos individuales fueron codificados en una forma que los hace difíciles de entender, poner a prueba y mantener. En tales casos, el código dentro de los módulos sospechosos puede reestructurarse.

Para realizar esta actividad se analiza el código fuente con una herramienta de reestructuración. Las violaciones a los constructos de programación estructurada se anotan y luego el código se reestructura (esto puede hacerse automáticamente) o incluso se reescribe en un lenguaje de programación más moderno. El código reestructurado resultante se revisa y pone a prueba para garantizar que no se introdujeron anomalías. La documentación de código interna se actualiza.

Reestructuración de datos. Un programa con arquitectura de datos débil será difícil de adaptar y mejorar. De hecho, para muchas aplicaciones, la arquitectura de información tiene más que ver con la viabilidad a largo plazo de un programa que con el código fuente en sí.

A diferencia de la reestructuración de código, que ocurre en un nivel de abstracción relativamente bajo, la reestructuración de datos es una actividad de reingeniería a gran escala. En la mayoría de los casos, la reestructuración de los datos comienza con una actividad de ingeniería inversa. La arquitectura de datos existente se diseca y se definen modelos de datos necesarios (capítulos 6 y 9). Se identifican los objetos y atributos de datos, y se revisa la calidad de las estructuras de datos existentes.

Cuando la estructura de datos es débil (por ejemplo, si se implementan archivos planos, cuando un enfoque relacional simplificaría enormemente el procesamiento), los datos se someten a reingeniería.

Puesto que la arquitectura de datos tiene una fuerte influencia sobre la arquitectura del programa y sobre los algoritmos que los pueblan, los cambios a los datos invariablemente resultarán en cambios arquitectónicos o en el nivel de código.

---

4 La reestructuración de código tiene algunos de los elementos de la "refactorización", un concepto de rediseño introducido en el capítulo 8 y estudiado en otras partes de este libro.

Ingeniería hacia adelante. En un mundo ideal, las aplicaciones se reconstruirían usando un "motor de reingeniería" automático. El programa antiguo se alimentaría en el motor, se analizaría, se reestructuraría y luego se regeneraría de manera que mostrara los mejores aspectos de la calidad del software. A corto plazo es improbable que tal "motor" aparezca, pero los proveedores introdujeron herramientas que proporcionan un subconjunto limitado de dichas capacidades y que abordan dominios de aplicación específicos (por ejemplo, aplicaciones que se implementan usando un sistema de base de datos específico). Más importante, dichas herramientas de reingeniería se vuelven cada vez más sofisticadas.

La ingeniería hacia adelante no sólo recupera información de diseño del software existente, sino que también usa esta información para alterar o reconstituir el sistema existente con la intención de mejorar su calidad global. En la mayoría de los casos, el software sometido a reingeniería vuelve a implementar la función del sistema existente y también añade nuevas funciones y/o mejora el rendimiento global.

29.6 Ingeniería inversa

La ingeniería inversa conjura una imagen de la "rendija mágica". Usted alimenta en la rendija un archivo fuente sin documentar, diseñado de manera fortuita, y del otro lado sale una descripción y documentación completas del diseño para el programa de cómputo. Por desgracia, la rendija mágica no existe. La ingeniería inversa puede extraer información de diseño a partir del código fuente, pero el nivel de abstracción, la completitud de la documentación, el grado en el que las herramientas y un analista humano trabajan en conjunto, y la direccionalidad del proceso son enormemente variables.

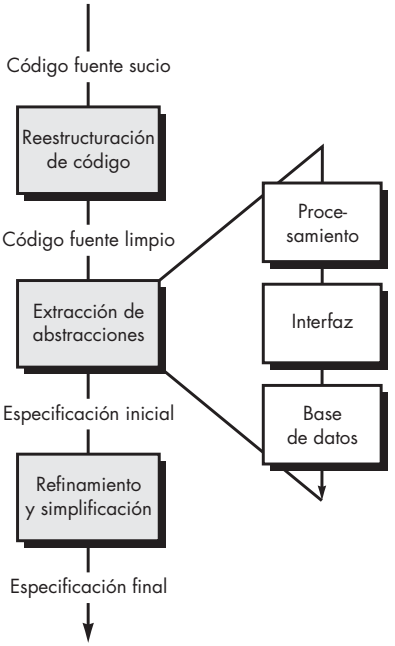
El *nivel de abstracción* de un proceso de ingeniería inversa y las herramientas usadas para efectuarla tienen que ver con la sofisticación de la información de diseño que puede extraerse del código fuente. De manera ideal, el nivel de abstracción debe ser tan alto como sea posible, es decir, el proceso de ingeniería inversa debe ser capaz de inferir representaciones de diseño procedimental (una abstracción de bajo nivel), información de estructura de programa y datos (un nivel de abstracción un poco más alto), modelos de objeto, modelos de datos y/o flujo de control (un nivel de abstracción relativamente alto) y modelos de relación de entidad (un nivel de abstracción alto). Conforme aumenta el nivel de abstracción se proporciona información que permitirá facilitar la comprensión del programa.

La *completitud* de un proceso de ingeniería inversa se refiere al nivel de detalle que se proporciona en un nivel de abstracción. En la mayoría de los casos, la completitud disminuye conforme aumenta el nivel de abstracción. Por ejemplo, dada una lista de código fuente, es relativamente sencillo desarrollar una representación de diseño procedimental completa. También pueden inferirse representaciones de diseño arquitectónico simples, pero es mucho más difícil desarrollar un conjunto completo de diagramas o modelos UML.

La completitud mejora en proporción directa a la cantidad de análisis realizado por la persona que efectúa la ingeniería inversa. La *interactividad* tiene que ver con el grado en el que el ser humano se "integra" con las herramientas automatizadas para crear un proceso de ingeniería inversa efectivo. En la mayoría de los casos, conforme aumenta el nivel de abstracción, la interactividad debe aumentar o decaerá la completitud.

Si la *direccionalidad* del proceso de ingeniería inversa es de una vía, toda la información extraída del código fuente se proporciona al ingeniero de software que luego puede usarla, durante cualquier actividad de mantenimiento. Si la direccionalidad es de dos vías, la información se alimenta a una herramienta de reingeniería que intenta reestructurar o regenerar el programa antiguo.

FIGURA 29.3
El proceso de ingeniería inversa



WebRef
Útiles recursos para "recuperación de diseño y comprensión de programa" pueden encontrarse en **wwwsel.iit.nrc.ca/projects/dr/dr.html**

El proceso de ingeniería inversa se representa en la figura 29.3. Antes de poder comenzar las actividades de ingeniería inversa, el código fuente no estructurado ("sucio") se reestructura (sección 29.5.1) de modo que sólo contenga los constructos de programación estructurados.[5] Esto hace que el código fuente sea más fácil de leer y que proporcione la base para todas las actividades de ingeniería inversa posteriores.

El núcleo de la ingeniería inversa radica en una actividad llamada *extracción de abstracciones*. Debe evaluar el programa antiguo y, a partir del código fuente (con frecuencia no documentado), desarrollar una especificación significativa del procesamiento que se realiza, de la interfaz de usuario que se aplica y de las estructuras de datos del programa o de la base de datos que se usa.

29.6.1 Ingeniería inversa para comprender datos

CONSEJO

En algunos casos, la primera actividad de reingeniería intenta construir un diagrama de clase UML.

La ingeniería inversa de datos ocurre en diferentes niveles de abstracción y con frecuencia es la primera tarea de reingeniería. En el nivel del programa, las estructuras de datos internas del programa con frecuencia deben someterse a ingeniería inversa como parte de un esfuerzo de reingeniería global. En el nivel del sistema, las estructuras de datos globales (por ejemplo, archivos, bases de datos) con frecuencia se someten a reingeniería para acomodar nuevos paradigmas de administración de base de datos (por ejemplo, moverse de un archivo plano a sistemas de bases de datos relacionales u orientadas a objetos). La ingeniería inversa de las estructuras de datos globales actuales monta el escenario para la introducción de una nueva base de datos en todo el sistema.

CONSEJO

El enfoque de la ingeniería inversa para datos para software convencional sigue una ruta análoga: 1) construir un modelo de datos, 2) identificar atributos de objetos de datos y 3) definir relaciones.

Estructuras de datos internas. Las técnicas de ingeniería inversa para datos internos del programa se enfocan en la definición de clases de objetos. Esto se logra al examinar el código del programa con la intención de agrupar variables del programa relacionadas. En muchos ca-

5 El código puede reestructurarse usando un *motor de reestructuración*, una herramienta que reestructura código fuente.

sos, la organización de datos dentro del código identifica tipos de datos abstractos. Por ejemplo, el registro de estructuras, archivos, listas y otras estructuras de datos con frecuencia proporciona un indicador inicial de clases.

Estructura de la base de datos. Sin importar su organización lógica y su estructura física, una base de datos permite la definición de objetos de datos y soporta algún método para establecer relaciones entre los objetos. Por tanto, la reingeniería de un esquema de base de datos en otro nuevo requiere comprender los objetos existentes y sus relaciones.

Puede usar los siguientes pasos [Pre94] para definir el modelo de extracción de datos como precursor de la reingeniería de un nuevo modelo de base de datos: 1) construir un modelo de objeto inicial, 2) determinar claves candidatas (los atributos se examinan para determinar si se usan para apuntar hacia otro registro o tabla; los que funcionan como punteros se convierten en clases candidatas), 3) refinar las clases tentativas, 4) definir generalizaciones y 5) descubrir asociaciones usando técnicas que sean análogas al enfoque CRC. Una vez que se conoce la información definida en los pasos anteriores, puede aplicarse una serie de transformaciones [Pre94] para mapear la antigua estructura de la base de datos en una nueva estructura.

29.6.2 Ingeniería inversa para entender el procesamiento

Cita:

"Existe una pasión por la comprensión, así como existe una pasión por la música. Dicha pasión es más bien común en los niños, pero se pierde en la mayoría de la gente tiempo después."

Albert Einstein

La ingeniería inversa para entender el procesamiento comienza con un intento por comprender y luego extraer abstracciones procedimentales representadas mediante el código fuente. Para comprender las abstracciones procedimentales se analiza el código en varios niveles de abstracción: sistema, programa, componente, patrón y enunciado.

La funcionalidad global de todo el sistema de aplicación debe entenderse antes de que ocurra trabajo de ingeniería inversa más detallado. Esto establece el contexto para un mayor análisis y proporciona comprensión acerca de los conflictos de interoperabilidad entre aplicaciones dentro del sistema. Cada uno de los programas que constituyen el sistema de aplicación representa una abstracción funcional en un nivel alto de detalle. Se crea un diagrama de bloques, que representa la interacción entre dichas abstracciones funcionales. Cada componente realiza alguna subfunción y representa una abstracción procedimental definida. Se desarrolla una narrativa de procesamiento para cada componente. En algunas situaciones, ya existen especificaciones de sistema, programa y componente. Cuando éste es el caso, las especificaciones se revisan para conformarse con el código existente.[6]

Las cosas se vuelven más complejas cuando se considera el código dentro de un componente. Debe buscar secciones de código que representen patrones procedimentales genéricos. En casi todo componente, una sección de código prepara los datos para procesamiento (dentro del módulo), una sección diferente del código realiza el procesamiento y otra prepara los resultados del procesamiento para exportarlos desde el componente. Dentro de cada una de esas secciones, pueden encontrarse patrones más pequeños; por ejemplo, validación de datos y comprobación de enlaces que con frecuencia ocurren dentro de la sección de código que prepara los datos para procesamiento.

Para sistemas grandes, la ingeniería inversa por lo general se logra usando un enfoque semiautomatizado. Es posible usar herramientas automatizadas para auxiliarse en la comprensión de la semántica del código existente. La salida de este proceso pasa entonces a reestructuración de herramientas de ingeniería hacia adelante a fin de completar el proceso de reingeniería.

6 Con frecuencia, las especificaciones escritas anteriormente, en la historia de vida del programa, nunca se actualizan. Conforme se realizan cambios, el código ya no es congruente con la especificación.

29.6.3 Ingeniería inversa de interfaces de usuario

Las GUI (interfaces de usuario gráficas) sofisticadas se han vuelto obligatorias para productos y sistemas basados en computadora de todo tipo. Por tanto, el redesarrollo de las interfaces de usuario se ha convertido en uno de los tipos más comunes de actividad de reingeniería. Pero antes de poder reconstruir una interfaz de usuario, debe realizarse ingeniería inversa.

Para comprender completamente una interfaz de usuario existente, deben especificarse la estructura y el comportamiento de la interfaz. Merlo *et al.* [Mer93] sugieren tres preguntas básicas que deben responderse conforme comienza la ingeniería inversa de la UI.

? ¿Cómo comprender el funcionamiento de una interfaz de usuario existente?

- ¿Cuáles son las acciones básicas (por ejemplo, golpes de tecla y clics de ratón) que debe procesar la interfaz?
- ¿Cuál es la descripción compacta de la respuesta de comportamiento del sistema a dichas acciones?
- ¿Qué se entiende por "reemplazo" o, más precisamente, qué concepto de equivalencia de interfaces es relevante aquí?

La notación de modelado de comportamiento (capítulo 7) puede proporcionar un medio para desarrollar respuestas a las primeras dos preguntas. Mucha de la información necesaria para crear un modelo de comportamiento puede obtenerse al observar la manifestación externa de la interfaz existente. Pero información adicional necesaria para crear el modelo de comportamiento debe extraerse del código.

Es importante observar que un reemplazo de GUI puede no reflejar con exactitud la antigua interfaz (de hecho, puede ser radicalmente diferente). Con frecuencia, vale la pena desarrollar una nueva metáfora de interacción. Por ejemplo, una antigua UI solicita que un usuario proporcione un factor de escala (que va de 1 a 10) para encoger o ampliar una imagen gráfica. Una GUI sometida a reingeniería puede usar una barra de desplazamiento y ratón para lograr la misma función.

HERRAMIENTAS DE SOFTWARE

Ingeniería inversa

Objetivo: Auxiliar a los ingenieros de software a comprender la estructura interna de diseño de programas complejos.

Mecánica: En la mayoría de los casos, las herramientas de ingeniería inversa aceptan código fuente como entrada y producen varias representaciones de diseño estructural, procedimental, de datos y de comportamiento.

Herramientas representativas:[7]

Imagix 4D, desarrollada por Imagix (**www.imagix.com**), "ayuda a los desarrolladores de software a comprender software C y C++ complejo o heredado" mediante ingeniería inversa y documentación de código fuente.

Understand, desarrollado por Scientific Toolworks, Inc. (**www.scitools.com**), analiza gramaticalmente Ada, Fortran, C, C++ y Java "para ingeniería inversa, documenta automáticamente, calcula métricas de código y le ayuda a comprender, navegar y mantener código fuente".

Una lista exhaustiva de herramientas de ingeniería puede encontrarse en **http://scgwiki.iam.unibe.ch:8080/SCG/370**

7 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas que hay en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

29.7 Reestructuración

La *reestructuración de software* modifica el código fuente y/o los datos con la intención de hacerlos sensibles a cambios futuros. En general, la reestructuración no modifica la arquitectura global del programa. Tiende a enfocarse sobre detalles de diseño de módulos individuales y sobre estructuras de datos locales definidas dentro de módulos. Si el esfuerzo de reestructuración se extiende más allá de las fronteras del módulo y abarca la arquitectura del software, la reestructuración se convierte en ingeniería hacia adelante (sección 29.7).

Aunque la reestructuración de código puede aliviar los problemas inmediatos asociados con la depuración o con pequeños cambios, no es reingeniería. El beneficio real se logra sólo cuando se reestructuran los datos y la arquitectura.

La reestructuración ocurre cuando la arquitectura básica de una aplicación es sólida, aun cuando el interior técnico necesite trabajarse. Se inicia cuando grandes partes del software son aprovechables y sólo un subconjunto de todos los módulos y datos necesitan modificación extensa.[8]

29.7.1 Reestructuración de código

La *reestructuración de código* se realiza para producir un diseño que produzca la misma función pero con mayor calidad que el programa original. En general, las técnicas de reestructuración de código (por ejemplo, las técnicas de simplificación lógica de Warnier [War74]) modelan la lógica del programa usando álgebra booleana y luego aplican una serie de reglas de transformación que producen lógica reestructurada. El objetivo es tomar una "ensalada" de código y derivar un diseño procedimental que se conforme con la filosofía de programación estructurada (capítulo 10).

También se han propuesto otras técnicas de reestructuración para su uso con herramientas de reingeniería. Un diagrama de intercambio de recursos mapea cada módulo de programa y los recursos (tipos de datos, procedimientos y variables) que se intercambiarán entre él y otros módulos. Al crear representaciones de flujo de recursos, la arquitectura del programa puede reestructurarse para lograr un mínimo acoplamiento entre módulos.

29.7.2 Reestructuración de datos

Antes de que pueda comenzar la reestructuración de datos debe realizarse una actividad de ingeniería inversa llamada *análisis de código fuente*. Se evalúan todos los enunciados de lenguaje de programación que contienen definiciones de datos, descripciones de archivo I/O y descripciones de interfaz. La intención es extraer ítems de datos y objetos, obtener información acerca del flujo de datos y entender las estructuras de datos existentes que se implementaron. Esta actividad en ocasiones se llama *análisis de datos*.

Una vez completado el análisis de datos, comienza el *rediseño de datos*. En su forma más simple, un paso de *estandarización de registro de datos* clarifica las definiciones de los datos para lograr consistencia entre nombres de ítem de datos o formatos de registro físico dentro de una estructura de datos existente o dentro de un formato de archivo. Otra forma de rediseño, llamada *racionalización de nombre de datos*, garantiza que todas las convenciones de nomenclatura de datos se establezcan de acuerdo con estándares locales y que los sobrenombres se eliminen conforme los datos fluyen a través del sistema.

Cuando la reestructuración avanza más allá de la estandarización y la racionalización, se realizan modificaciones físicas a las estructuras de datos existentes para hacer que el diseño de datos sea más efectivo. Esto puede significar una traducción de un formato de archivo a otro o, en algunos casos, traducción de un tipo de base de datos a otra.

---

8 En ocasiones es difícil hacer una distinción entre reestructuración extensa y redesarrollo. Ambos son reingeniería.

HERRAMIENTAS DE SOFTWARE

Reestructuración de software

Objetivo: El objetivo de las herramientas de reestructuración es transformar el software de computadora no estructurado más antiguo en lenguajes de programación y estructuras de diseño modernos.

Mecánica: En general, se ingresa el código fuente y se transforma en un programa mejor estructurado. En algunos casos, la transformación ocurre dentro del mismo lenguaje de programación. En otros, un lenguaje de programación más antiguo se transforma en un lenguaje más moderno.

Herramientas representativas:[9]

DMS Software Reengineering Toolkit, desarrollada por Semantic Design (**www.semdesigns.com**), proporciona varias capacidades de reestructuración para COBOL, C/C++, Java, Fortran 90 y VHDL.

Clone Doctor, desarrollada por Semantic Designs, Inc. (**www.semdesigns.com**), analiza y transforma programas escritos en C, C++, Java o COBOL o en cualquier otro lenguaje de computadora basado en texto.

plusFORT, desarrollada por Polyheddron (**www.polyhedron.com**), es una suite de herramientas FORTRAN que contiene capacidades para reestructurar programas FORTRAN pobremente diseñados en el estándar moderno FORTRAN o C.

Indicadores hacia varias herramientas de reingeniería e ingeniería inversa pueden encontrarse en **www.csse.monash.edu/~ipeake/reeng/free-swre-tools.html** y en **www.cs.ualberta.ca/~kenw/toolsdir/all.html**

29.8 INGENIERÍA HACIA ADELANTE

¿Qué opciones existen cuando uno se enfrenta con un programa diseñado e implementado pobremente?

Un programa con flujo de control que sea el equivalente gráfico de una olla de espagueti, con "módulos" que tienen 2 000 enunciados de largo, con pocas líneas de comentarios significativos en 290 000 enunciados fuente y sin otra documentación, debe modificarse para alojar los cambiantes requisitos de usuario. Se tienen las siguientes opciones:

1. Para implementar los cambios necesarios puede luchar a través de modificación tras modificación, combatir al diseño *ad hoc* y el código fuente enredado.
2. Puede intentar comprender los funcionamientos interiores más amplios del programa con la intención de hacer modificaciones de manera más efectiva.
3. Puede rediseñar, recodificar y poner a prueba aquellas porciones del software que requieran modificación y aplicar un enfoque de ingeniería de software a todos los segmentos revisados.
4. Puede rediseñar, recodificar y poner a prueba completamente el programa, y usar herramientas de reingeniería para ayudar a comprender el diseño actual.

La reingeniería es un poco como limpiar sus dientes. Puede pensar en miles de razones para demorarla, y la evitará con desidia durante un rato. Pero finalmente, sus tácticas de demora regresarán para causarle dolor.

No hay una sola opción "correcta". Las circunstancias pueden dictar la primera opción incluso si las otras son más deseables.

En lugar de esperar hasta recibir una solicitud de mantenimiento, la organización de desarrollo o soporte usa los resultados de un análisis de inventario para seleccionar un programa: 1) que permanecerá en uso durante un número preseleccionado de años, 2) que en el momento se use con éxito y 3) que tenga probabilidad de experimentar grandes modificaciones o aumentos en el futuro cercano. Entonces se aplican las opciones 2, 3 o 4.

A primera vista, la sugerencia de que redesarrolle un programa grande cuando ya existe una versión operativa puede parecer muy extravagante. Antes de juzgar, considere los siguientes puntos:

9 Las herramientas que se mencionan aquí no representan un respaldo, sino una muestra de las herramientas que hay en esta categoría. En la mayoría de los casos, los nombres de las herramientas son marcas registradas por sus respectivos desarrolladores.

1. El costo de mantener una línea de código fuente puede ser 20 a 40 veces el costo del desarrollo inicial de dicha línea.
2. El rediseño de la arquitectura del software (programa y/o estructura de datos), con el uso de modernos conceptos de diseño, puede facilitar enormemente el mantenimiento futuro.
3. Puesto que ya existe un prototipo del software, la productividad de desarrollo debe ser mucho más alta que el promedio.
4. Ahora el usuario experimenta con el software. Por tanto, pueden averiguarse con mayor facilidad los nuevos requisitos y la dirección del cambio.
5. Las herramientas automatizadas para reingeniería facilitarán algunas partes de la labor.
6. Existirá una configuración de software completa (documentos, programas y datos) al completar el mantenimiento preventivo.

Un gran desarrollador interno de software (por ejemplo, un grupo de desarrollo de sistemas de software empresarial para una gran compañía de productos al consumidor) puede tener una producción de 500 a 2 000 programas dentro de su dominio de responsabilidad. Dichos programas pueden clasificarse por importancia y luego revisarse como candidatos para ingeniería hacia adelante.

El proceso de ingeniería hacia adelante aplica los principios, conceptos y métodos de la ingeniería de software para volver a crear una aplicación existente. En la mayoría de los casos, la ingeniería hacia adelante no crea simplemente un equivalente moderno de un programa más antiguo. En vez de ello, en el esfuerzo de reingeniería se integran nuevos requisitos de usuario y tecnología. El programa redesarrollado extiende las capacidades de la aplicación más antigua.

29.8.1 Ingeniería hacia adelante para arquitecturas cliente-servidor

Durante las décadas pasadas, muchas aplicaciones de mainframe se sometieron a reingeniería para alojar arquitecturas cliente-servidor (incluidas las *webapps*). En esencia, los recursos de cómputo centralizados (incluido el software) se distribuyen entre muchas plataformas cliente. Aunque pueden diseñarse varios entornos distribuidos, la aplicación mainframe típica que se somete a reingeniería en una arquitectura cliente-servidor tiene las siguientes características:

- La funcionalidad de la aplicación migra a cada computadora cliente.
- Se implementan nuevas interfaces GUI en los sitios cliente.
- Las funciones de base de datos se ubican en el servidor.
- La funcionalidad especializada (por ejemplo, análisis con uso intenso de computadora) puede permanecer en el sitio servidor.
- Deben establecerse nuevos requisitos de comunicaciones, seguridad, archivado y control en los sitios cliente y servidor.

En algunos casos, la migración a una arquitectura cliente-servidor debe abordarse no como reingeniería, sino como un nuevo esfuerzo de desarrollo. La reingeniería entra al cuadro solamente cuando deba integrarse funcionalidad específica del sistema antiguo en la nueva arquitectura.

Es importante observar que la migración de mainframe a cómputo cliente-servidor requiere reingeniería tanto de la empresa como del software. Además, debe establecerse una "infraestructura de red empresarial" [Jay94].

La reingeniería para aplicaciones cliente-servidor comienza con un profundo análisis del entorno empresarial que abarca el mainframe existente. Pueden identificarse tres capas de abstracción. La *base de datos* que se asienta en los cimientos de una arquitectura cliente-servidor gestiona las transacciones y consultas de aplicaciones del servidor. Aunque dichas transacciones y consultas deben controlarse dentro del contexto de un conjunto de reglas empresariales (definidas por un proceso empresarial existente o de reingeniería), las aplicaciones cliente proporcionan funcionalidad dirigida a la comunidad usuaria.

Las funciones del sistema de gestión de base de datos existente y la arquitectura de datos de la base de datos existente deben someterse a ingeniería inversa como precursor del rediseño de la capa cimiento de la base de datos. En algunos casos, se crea un nuevo modelo de datos (capítulo 6). En todo caso, la base de datos cliente-servidor se somete a reingeniería para garantizar que las transacciones se ejecutan en forma consistente, que todas las actualizaciones se realizan sólo por usuarios autorizados, que las reglas empresariales núcleo se refuerzan (por ejemplo, antes de borrar el registro de un vendedor, el servidor se asegura de que no existan cuentas por pagar, contratos o comunicaciones para dicho proveedor), que las consultas puedan acomodarse de manera eficiente y que se establece la capacidad completa de archivado.

Las capas de reglas empresariales representan software residente tanto en el cliente como en el servidor. Este software realiza tareas de control y coordinación para garantizar que las transacciones y consultas entre la aplicación cliente y la base de datos se conforman con el proceso empresarial establecido.

Las capas de aplicaciones cliente implementan funciones empresariales que requieren grupos específicos de usuarios finales. En muchas instancias, una aplicación mainframe se segmenta en algunas aplicaciones de escritorio más pequeñas sometidas a reingeniería. La comunicación entre aplicaciones de escritorio (cuando sea necesario) se controla mediante la capa de reglas empresariales.

Un análisis profundo del diseño y reingeniería de software cliente-servidor se deja para libros dedicados a la materia. Si tiene más interés, consulte [Van02], [Cou00] u [Orf99].

29.8.2 Ingeniería hacia adelante para arquitecturas orientadas a objetos

La ingeniería de software orientada a objetos se ha convertido en el paradigma de desarrollo elegido por muchas organizaciones de software. Pero, ¿qué hay de las aplicaciones existentes que se desarrollaron usando métodos convencionales? En algunos casos, la respuesta es dejar tales aplicaciones "como están". En otras, las aplicaciones antiguas deben someterse a reingeniería, de modo que puedan integrarse con facilidad en sistemas grandes orientados a objetos.

La reingeniería de software convencional en una implementación orientada a objeto usa muchas de las mismas técnicas estudiadas en la parte 2 de este libro. Primero, el software existente se somete a ingeniería inversa para que puedan crearse modelos adecuados de datos, funciones y comportamientos. Si el sistema sometido a reingeniería extiende la funcionalidad o comportamiento de la aplicación original, se crean casos de uso (capítulos 5 y 6). Los modelos de datos creados durante la ingeniería inversa se usan entonces en conjunción con el modelado CRC (capítulo 6) para establecer la base para la definición de clases. Se definen entonces las jerarquías de clase, modelos objeto-relacional, modelos objeto- comportamiento y subsistemas, y se comienza el diseño orientado a objetos.

Conforme avanza la ingeniería hacia adelante orientada a objetos desde el análisis hacia el diseño, puede invocarse un modelo de proceso ISBC (capítulo 10). Si la aplicación existente reside dentro de un dominio que ya está poblado con muchas aplicaciones orientadas a objetos, es probable que exista una robusta librería de componentes y que pueda usarse durante la ingeniería hacia adelante.

Para aquellas clases que deban someterse a ingeniería desde cero, es posible reutilizar algoritmos y estructuras de datos de la aplicación convencional existente. No obstante, las mismas deben rediseñarse para estar de acuerdo con la arquitectura orientada a objetos.

29.9 Economía de la reingeniería

En un mundo perfecto, todo programa no mantenible se retiraría de inmediato para sustituirlo con aplicaciones sometidas a reingeniería de alta calidad desarrolladas mediante modernas

Cita:

"Puede pagar un poco ahora o puede pagar mucho tiempo después."

Anuncio en una empresa de venta de autos que sugiere una afinación.

prácticas de ingeniería de software. Pero se vive en un mundo de recursos limitados. La reingeniería gasta recursos que pueden usarse para otros propósitos empresariales. Por tanto, antes de que una organización intente someter a reingeniería una aplicación existente, debe realizar un análisis costo-beneficio.

Sneed [Sne95] propone un modelo de análisis costo-beneficio para reingeniería, definiendo nueve parámetros:

P_1 = costo de mantenimiento anual actual para una aplicación
P_2 = costo de operaciones anuales actuales para una aplicación
P_3 = valor empresarial anual actual de una aplicación
P_4 = costo de mantenimiento anual predicho después de reingeniería
P_5 = costo de operaciones anuales predichas después de reingeniería
P_6 = valor empresarial anual predicho después de reingeniería
P_7 = costo de reingeniería estimado
P_8 = tiempo calendario de reingeniería estimado
P_9 = factor de riesgo de reingeniería ($P_9 = 1.0$ es nominal)
L = vida esperada del sistema

El costo asociado con el mantenimiento continuo de una aplicación candidata (es decir, la reingeniería no se realiza) puede definirse como

$$C_{\text{mant}} = [P_3 - (P_1 + P_2)] \times \text{L} \tag{29.1}$$

El costo asociado con reingeniería se define usando la siguiente relación:

$$C_{\text{reing}} = P_6 - (P_4 + P_5) \times (L - P_8) - (P_7 \times P_9) \tag{29.2}$$

Con los costos presentados en las ecuaciones 29.1 y 29.2, el beneficio global de la reingeniería puede calcularse como

$$\text{Beneficio en costo} = C_{\text{reing}} - C_{\text{mant}} \tag{29.3}$$

El análisis costo-beneficio que se presenta en estas ecuaciones puede realizarse para todas las aplicaciones de alta prioridad identificadas durante el análisis de inventario (sección 29.4.2). Aquellas aplicaciones que muestren el mayor costo-beneficio pueden marcarse para reingeniería, mientras que el trabajo sobre otras puede posponerse hasta que haya recursos disponibles.

29.10 Resumen

El mantenimiento y soporte del software son actividades en marcha que ocurren a lo largo de todo el ciclo de vida de una aplicación. Durante dichas actividades se corrigen defectos, se adaptan aplicaciones a un entorno operativo o empresarial cambiante, se implementan mejoras a petición de los participantes y se da soporte a los usuarios conforme integran una aplicación en su flujo de trabajo personal o empresarial.

La reingeniería ocurre en dos niveles de abstracción diferentes. En el nivel empresarial, la reingeniería se enfoca en el proceso empresarial con la intención de realizar cambios para mejorar la competitividad en alguna área de la empresa. En el nivel de software, la reingeniería examina los sistemas y aplicaciones de información con la intención de reestructurar o reconstruirlos de modo que muestren mayor calidad.

La reingeniería de procesos empresarial define las metas de la empresa, identifica y evalúa los procesos empresariales existentes (en el contexto de metas definidas), especifica y diseña procesos revisados y crea prototipos, los refina y ejemplifica dentro de una empresa. La RPE representa un enfoque que se extiende más allá del software. El resultado de la RPE con frecuen-

cia es la definición de formas en las que las tecnologías de la información pueden apoyar mejor a la empresa.

La reingeniería de software abarca una serie de actividades que incluyen análisis de inventario, reestructuración de documentos, ingeniería inversa, reestructuración de programa y datos e ingeniería hacia adelante. La intención de dichas actividades es crear versiones de programas existentes que muestren mayor calidad y mejor mantenibilidad, mismos que serán viables bien entrado el siglo XXI.

El costo-beneficio de la reingeniería puede determinarse de manera cuantitativa. El costo del *status quo*, es decir, el costo asociado con el soporte y mantenimiento actuales de una aplicación existente, se compara con los costos de proyectos de la reingeniería y con la reducción resultante en costos de mantenimiento y soporte. En casi todos los casos en los que un programa tenga una vida larga y en el momento muestre pobre mantenibilidad o soportabilidad, la reingeniería representa una estrategia empresarial efectiva en costo.

PROBLEMAS Y PUNTOS POR EVALUAR

29.1. Considere cualquier trabajo que el lector haya tenido durante los cinco años anteriores. Describa el proceso empresarial en el que participó. Use el modelo RPE descrito en la sección 29.4.2 para recomendar cambios al proceso con la intención de hacerlo más eficiente.

29.2. Realice algo de investigación acerca de la eficacia de la reingeniería de procesos de empresa. Presente argumentos a favor y en contra para este enfoque.

29.3. Su instructor seleccionará uno de los programas que hayan desarrollado en la clase durante este curso. Intercambie su programa al azar con alguien más en la clase. No explique o repase el programa. Ahora, implemente una mejora (especificada por su instructor) en el programa que haya recibido.

a) Realice todas las tareas de ingeniería de software, incluida una breve explicación (mas no con el autor del programa).
b) Siga cuidadosamente la pista a todos los errores que encuentre durante las pruebas.
c) Analice en clase sus experiencias.

29.4. Explore la lista de verificación del análisis de inventario que se presenta en el sitio web del libro, e intente desarrollar un sistema de calificación de software cuantitativo que pueda aplicar a programas existentes con la intención de elegir programas candidato para reingeniería. Su sistema debe extenderse más allá del análisis económico que se presentó en la sección 29.9.

29.5. Sugiera alternativas a la documentación impresa o electrónica convencional que puedan servir como base para la reestructuración de documentos. (Sugerencia: piense en nuevas tecnologías descriptivas que puedan usarse para comunicar la intención del software.)

29.6. Algunas personas creen que la tecnología de inteligencia artificial aumentará el nivel de abstracción del proceso de ingeniería inverso. Realice investigación acerca de este tema (es decir, el uso de IA para ingeniería inversa) y escriba un breve ensayo que adopte una postura acerca de este punto.

29.7. ¿Por qué es difícil lograr la completitud conforme aumenta el nivel de abstracción?

29.8. ¿Por qué debe aumentar la interactividad si aumenta la completitud?

29.9. Con información obtenida en la web, presente a su clase las características de tres herramientas de ingeniería inversa.

29.10. Existe una sutil diferencia entre reestructuración e ingeniería hacia adelante. ¿Cuál es?

29.11. Investigue la literatura y/o fuentes en internet para encontrar uno o más artículos que analizan estudios de caso de reingeniería de mainframe a cliente-servidor. Presente un resumen.

29.12. ¿Cómo determinaría de P_4 a P_7 en el modelo costo-beneficio que se presentó en la sección 29.9?

Lecturas y fuentes de información adicionales

Es irónico que el mantenimiento y el soporte de software representen las actividades más costosas en la vida de una aplicación y, sin embargo, se hayan escrito menos libros acerca de mantenimiento y soporte que de cualquier otro tema importante de la ingeniería de software. Entre las adiciones recientes a la literatura están los libros de Jarzabek (*Effective Software Maintenance and Evolution,* Auerbach, 2007), Grubb y Takang *(Software Maintenance: Concepts and Practice,* World Scientific Publishing Co., 2a. ed., 2003), y Pigoski (*Practical Software Maintenance,* Wiley, 1996). Éstos cubren las prácticas básicas de mantenimiento y soporte, y presentan una guía administrativa útil. Las técnicas de mantenimiento que se enfocan en entornos cliente-servidor se estudian en Schneberger (*Client/Server Software Maintenance*, McGraw-Hill, 1997). La investigación actual en "evolución del software" se presenta en una antología editada por Mens y Demeyer (*Software Evolution*, Springer, 2008).

Como muchos temas calientes en la comunidad empresarial, el alboroto que rodea a la reingeniería de procesos de empresa dio lugar a una visión más pragmática de la materia. Hammer y Champy (*Reengineering the Corporation*, HarperBusiness, edición revisada, 2003) precipitaron un interés temprano con su libro tan solicitado. Otros libros de Smith y Fingar [*Business Process Management (BPM): The Third Wave,* Meghan-Kiffer Press, 2003], Jacka y Keller (*Business Process Mapping: Improving Customer Satisfaction,* Wiley, 2001), Sharp y McDermott (*Workflow Modeling,* Artech House, 2001), Andersen (*Business Process Improvement Toolbox,* American Society for Quality, 1999), y Harrington *et al*. (*Business Process Improvement Workbook,* McGraw-Hill, 1997) presentan estudios de caso y lineamientos detallados para RPE.

Fong (*Information Systems Reengineering and Integration*, Springer, 2006) describe técnicas de conversión de bases de datos, ingeniería inversa e ingeniería hacia adelante para grandes sistemas de información. Demeyer *et al.* (*Object Oriented Reengineering Patterns*, Morgan Kaufmann, 2002) ofrecen una visión basada en patrones sobre cómo refactorizar y/o someter a reingeniería sistemas OO. Secord *et al.* (*Modernizing Legacy Systems,* Addison-Wesley, 2003), Ulrich (*Legacy Systems: Transformation Strategies,* Prentice Hall, 2002), Valenti (*Successful Software Reengineering,* IRM Press, 2002), y Rada (*Reengineering Software: How to Reuse Programming to Build New, State-of-the-Art Software,* Fitzroy Dearborn Publishers, 1999) se enfocan en estrategias y prácticas para reingeniería en un nivel técnico. Miller (*Reengineering Legacy Software Systems*, Digital Press, 1998) "ofrece un marco conceptual para mantener los sistemas de aplicación en sincronía con las estrategias empresariales y los cambios tecnológicos".

Cameron (*Reengineering Business for Success in the Internet Age,* Computer Technology Research, 2000) y Umar [*Application (Re)Engineering: Building Web-Based Applications and Dealing with Legacies,* Prentice Hall, 1997] proporcionan valiosos lineamientos para organizaciones que quieren transformar sistemas heredados en un entorno basado en web. Cook (*Building Enterprise Information Architectures: Reengineering Information Systems*, Prentice Hall, 1996) analiza el puente entre RPE y tecnología de la información. Aiken (*Data Reverse Engineering*, McGraw-Hill, 1996) analiza cómo recuperar, reorganizar y reutilizar datos organizacionales. Arnold (*Software Reengineering*, IEEE Computer Society Press, 1993) reunió una excelente antología de los primeros ensayos que se enfocan en las tecnologías de reingeniería de software.

En internet, está disponible una gran variedad de fuentes de información acerca de la reingeniería de software. Una lista actualizada de referencias en la World Wide Web que son relevantes para el mantenimiento y la reingeniería de software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

PARTE

Cinco

TEMAS AVANZADOS

En esta parte de *Ingeniería del software. Un enfoque práctico*, se considerarán algunos temas avanzados que extenderán su comprensión de la ingeniería del software. En los siguientes capítulos se abordan las siguientes preguntas:

- ¿Qué es el mejoramiento del proceso de software y cómo puede usarse para optimizar el estado de la práctica de la ingeniería del software?
- ¿Qué tendencias emergentes es posible que tengan una influencia significativa sobre la práctica de la ingeniería del software en la siguiente década?
- ¿Cuál es el camino por venir para los ingenieros del software?

Una vez respondidas dichas preguntas, comprenderá los temas que pueden tener un profundo impacto sobre la ingeniería del software en los años por venir.

CAPÍTULO

30 MEJORAMIENTO DEL PROCESO DE SOFTWARE

CONCEPTOS CLAVE

CMM de personal 688
CMMI. 685
educación y capacitación . . . 682
evaluación 683
factores de éxito cruciales . . 685
gestión del riesgo 684
instalación/migración 683
justificación. 682
mejoramiento del proceso de software (MPS) 677
  aplicabilidad 679
  marcos conceptuales 677
  proceso 680
modelos de madurez 679
rendimiento sobre inversión. 691
selección. 682
valoración. 681

Mucho antes de que se usara ampliamente la frase "mejoramiento del proceso de software", el autor trabajó con grandes corporaciones con la intención de mejorar el estado de sus prácticas de ingeniería del software. Como consecuencia de sus experiencias, el autor escribió un libro titulado *Making Software Engineering Happen* [Pre88]. En el prefacio de dicho libro hizo el siguiente comentario:

> Durante los pasados diez años, he tenido la oportunidad de ayudar a algunas compañías grandes a implementar prácticas de ingeniería del software. La tarea es difícil y rara vez pasa tan suavemente como uno quisiera; pero cuando triunfa, los resultados son profundos: los proyectos de software tienen más probabilidad de completarse con el tiempo, mejora la comunicación entre todos los constituyentes involucrados en el desarrollo de software, el nivel de confusión y caos que con frecuencia prevalece para grandes proyectos de software se reduce de manera sustancial, el número de errores que encuentra el cliente disminuye sustancialmente, la credibilidad de la organización de software aumenta y la administración tiene un problema menos por el cual preocuparse.
>
> Pero no todo es dulzura y luz. Muchas compañías intentan implementar prácticas de ingeniería del software y caen en la frustración. Otras llegan a medio camino y nunca ven los beneficios anotados anteriormente. Otras más lo hacen en forma tan ruda que da como resultado rebelión abierta entre el personal técnico y los administradores, con la posterior pérdida de moral.

Aunque tales palabras se escribieron hace más de 20 años, siguen siendo igualmente ciertas el día de hoy.

Conforme se avanza hacia la segunda década del siglo XXI, la mayoría de las principales organizaciones de ingeniería del software intentaron "hacer posible la ingeniería del software". Algunas implementaron prácticas individuales que ayudaron a mejorar la calidad del producto que construían y la oportunidad de su entrega. Otras establecieron un proceso de software "maduro" que guía las actividades técnicas y administrativas del proyecto. Pero otras continúan luchando. Sus prácticas son acierto y error, y su proceso es *ad hoc*. Ocasionalmente, su trabajo

UNA MIRADA RÁPIDA

¿Qué es? El mejoramiento del proceso de software abarca un conjunto de actividades que conducirán a un mejor proceso de software y, en consecuencia, a software de mayor calidad y a su entrega en forma más oportuna.

¿Quién lo hace? El personal que impulsa el MPS (mejoramiento de proceso de software) proviene de tres grupos: gerentes técnicos, ingenieros de software e individuos que tienen responsabilidad en el aseguramiento de la calidad.

¿Por qué es importante? Algunas organizaciones de software tienen poco más que un proceso de software *ad hoc*. Conforme trabajan para mejorar sus prácticas de ingeniería del software, deben abordar las debilidades en sus procesos existentes e intentar mejorar su enfoque para el trabajo de software.

¿Cuáles son los pasos? El enfoque del MPS es iterativo y continuo, pero puede verse en cinco pasos: 1) valoración del proceso de software actual, 2) educación y capacitación de profesionales y gerentes, 3) selección y justificación de elementos de proceso, métodos de ingeniería del software y herramientas, 4) implementación del plan MPS y 5) evaluación y afinación con base en los resultados del plan.

¿Cuál es el producto final? Aunque existen muchos productos operativos MPS intermedios, el resultado final es un proceso de software mejorado que conduce a software de mayor calidad.

¿Cómo me aseguro de que lo hice bien? El software que produce su organización se entregará con menos defectos, se reducirá la repetición del trabajo en cada etapa del proceso de software y la entrega oportuna será mucho más probable.

es espectacular, pero, en promedio, cada proyecto es una aventura, y nadie sabe si terminará mal o bien.

Así que, ¿cuál de estas cohortes necesita mejoramiento del proceso de software? La respuesta (que puede sorprenderle) es *ambas*. Las que triunfaron en hacer posible la ingeniería del software no pueden volverse complacientes. Deben trabajar de manera continua para mejorar su enfoque de la ingeniería del software. Y las que luchan deben comenzar su viaje por el camino hacia el mejoramiento.

30.1 ¿Qué es MPS?

PUNTO CLAVE

MPS implica un proceso de software definido, un enfoque organizacional y una estrategia para el mejoramiento.

El término *mejoramiento del proceso de software* (MPS) implica muchas cosas. Primero, que los elementos de un proceso de software efectivo pueden definirse en forma efectiva; segundo, que un enfoque organizacional existente sobre el desarrollo del software puede valorarse en contraste con dichos elementos; y tercero, que es posible definir una estrategia de mejoramiento significativa. La estrategia MPS transforma el enfoque existente sobre el desarrollo del software en algo que es más enfocado, más repetible y más confiable (en términos de la calidad del producto producido y de la oportunidad de la entrega).

Puesto que el MPS no es gratuito, debe entregar un rendimiento sobre la inversión. El esfuerzo y el tiempo que se requieren para implementar una estrategia MPS deben pagar por sí mismos en alguna forma mensurable. Para hacer esto, los resultados del proceso y la práctica mejorados deben conducir a una reducción en los "problemas" del software que cuestan tiempo y dinero. Debe reducir el número de defectos que se entregan a los usuarios finales, la cantidad de repetición de proceso debida a problemas de calidad, los costos asociados con el mantenimiento y el soporte del software (capítulo 29) y los costos indirectos que ocurren cuando el software se entrega tarde.

30.1.1 Enfoques del MPS

Cita:

"Mucha de la crisis del software es autoinfligida, como cuando un presidente del área de informática en ejercicio dice: 'Prefiero que salga mal antes que entregarlo tarde. Siempre podremos repararlo después'."

Mark Paulk

Aunque una organización puede elegir un enfoque relativamente informal del MPS, la gran mayoría elige uno de los marcos conceptuales MPS. Un *marco conceptual MPS* define: 1) un conjunto de características que deben presentarse si quiere lograrse un proceso de software efectivo, 2) un método para valorar si dichas características están presentes, 3) un mecanismo para resumir los resultados de cualquier valoración y 4) una estrategia para auxiliar a una organización de software a implementar aquellas características del proceso que sean débiles o que hagan falta.

Un marco conceptual MPS valora la "madurez" del proceso de una organización y proporciona un indicio cualitativo de su nivel de madurez. De hecho, con frecuencia se aplica el término "modelo de madurez" (sección 30.1.2). En esencia, el marco conceptual MPS abarca un modelo de madurez que a su vez incorpora un conjunto de indicadores de calidad de proceso que ofrecen una medida global de la calidad del proceso que llevará a la calidad del producto.

La figura 30.1 proporciona un panorama de un marco conceptual MPS típico. Se muestran los elementos clave del marco conceptual y su relación mutua.

¿Qué grupos impulsan un esfuerzo MPS?

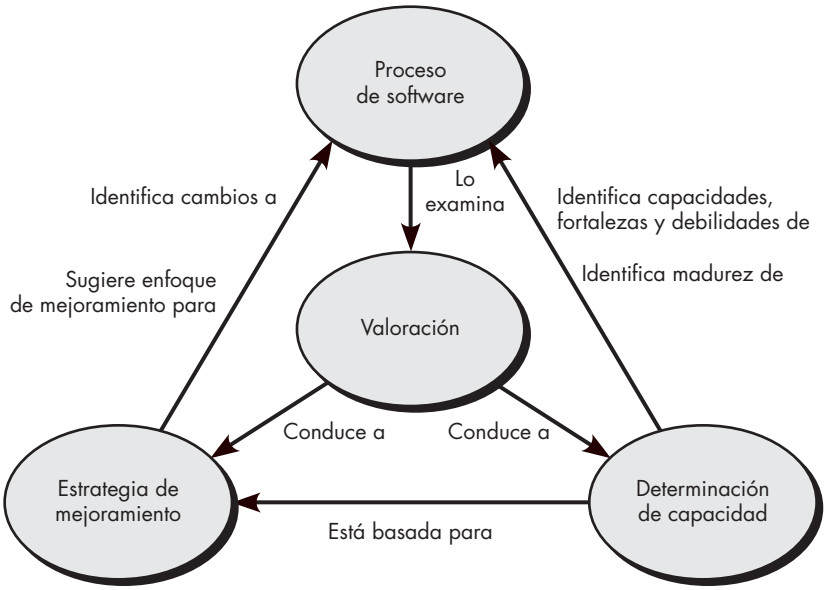
Debe observarse que no existe un marco conceptual MPS universal. De hecho, el marco conceptual MPS que elige una organización refleja las áreas que impulsan el esfuerzo MPS. Conradi [Con96] define seis diferentes grupos de apoyo al MPS:

Certificadores de calidad. Los esfuerzos de mejoramiento de proceso que impulsa este grupo se enfocan en la siguiente relación:

Calidad (*Proceso*) ⇒ **Calidad** (*Producto*)

Su enfoque consiste en enfatizar los métodos de valoración y examinar un conjunto bien definido de características que le permiten determinar si el proceso muestra calidad. Es

FIGURA 30.1
Elementos de un marco conceptual MPS
Fuente: Adaptado de [Rou02].



más probable que adopten un marco conceptual de proceso, como CMM, SPICE, TickIT o Bootstrap.[1]

Formalistas. Este grupo quiere entender (y cuando es posible, optimizar) el flujo de trabajo del proceso. Para lograrlo, usa lenguajes de modelado de proceso (PML) a fin de crear un modelo del proceso existente y luego diseñar extensiones o modificaciones que harán más efectivo el proceso.

¿Cuáles son los diferentes grupos que apoyan el MPS?

Defensores de las herramientas. Este grupo insiste en un enfoque del MPS asistido por herramientas que modelan el flujo de trabajo y otras características del proceso de manera que pueda analizarse para su mejoramiento.

Profesionales. Este grupo usa un enfoque pragmático, "que enfatiza la administración tradicional de proyecto, calidad y producto, y aplica planificación y métricas en el nivel de proyecto, pero con poco modelado de proceso formal o pronunciamiento de apoyo" [Con96].

Reformadores. La meta de este grupo es el cambio organizacional que pueda conducir a un mejor proceso de software. Tienden a enfocarse más en los temas humanos (sección 30.5) y enfatizan medidas de capacidad humana y estructura.

Ideólogos. Este grupo se enfoca en lo adecuado de un modelo de proceso particular para un dominio de aplicación específico o estructura organizativa. En lugar de los modelos de proceso de software típicos (por ejemplo, modelos iterativos), los ideólogos tendrían mayor interés en un proceso que, por ejemplo, apoyara el reuso o la reingeniería.

Conforme se aplica un marco conceptual MPS, el grupo impulsor (sin importar su enfoque global) debe establecer mecanismos para: 1) apoyar la transición tecnológica, 2) determinar el grado en el que una organización está lista para absorber los cambios de proceso que se propongan y 3) medir el grado en el que se adoptaron los cambios.

1 Cada uno de estos marcos conceptuales MPS se estudia más adelante, en este capítulo.

30.1.2 Modelos de madurez

Un *modelo de madurez* se aplica dentro del contexto de un marco conceptual MPS. La intención del modelo de madurez es proporcionar un indicio global de la "madurez del proceso" que muestra una organización de software, es decir, un indicio de la calidad del proceso de software, el grado en el que los profesionales entienden y aplican el proceso, y el estado general de la práctica de ingeniería del software. Esto se logra usando algún tipo de escala ordinal.

Por ejemplo, el *modelo de madurez de capacidad* (sección 30.4) del Software Engineering Institute sugiere cinco niveles de madurez [Sch96]:

PUNTO CLAVE

Un modelo de madurez define niveles de competencia e implementación de proceso de software.

Nivel 5, optimizado. La organización tiene sistemas de realimentación cuantitativa en su lugar para identificar las debilidades del proceso y fortalecer esos puntos de manera proactiva. Los equipos de proyecto analizan defectos para determinar sus causas; los procesos de software se evalúan y actualizan para evitar que recurran tipos conocidos de defectos.

Nivel 4, gestionado. Métricas de proceso de software y de calidad de producto detalladas establecen el cimiento de evaluación cuantitativa. Las variaciones significativas en el desempeño del proceso pueden distinguirse del ruido aleatorio, y pueden predecirse las tendencias en las cualidades del proceso y el producto.

Nivel 3, definido. Los procesos para administración e ingeniería se documentan, estandarizan e integran en un proceso de software estándar para la organización. Todos los proyectos usan una versión aprobada y a la medida del proceso de software estándar de la organización para desarrollo de software.

Nivel 2, repetible. Se establecen procesos de administración de proyecto básicos para rastrear costo, calendario y funcionalidad. La planificación y administración de nuevos productos se basa en la experiencia con proyectos similares.

Nivel 1, inicial. Pocos procesos definidos, y el éxito depende más del esfuerzo heroico individual que de seguir un proceso y usar un esfuerzo sinérgico de equipo.

La escala de madurez CMM va más allá, pero la experiencia indica que muchas organizaciones muestran niveles de "inmadurez de proceso" [Sch96] que minan cualquier intento racional por mejorar las prácticas de ingeniería de software. Schorsch [Sch06] sugiere cuatro niveles de inmadurez que se encuentran frecuentemente en el mundo real de las organizaciones de desarrollo del software:

? ¿Cómo se reconoce a una organización que resistirá los esfuerzos MPS?

Nivel 0, negligente. Fracaso para permitir que tenga éxito un proceso de desarrollo exitoso. Todos los problemas se perciben como problemas técnicos. Las actividades administrativas y de aseguramiento de la calidad están condenadas a situarse por encima y ser superfluas en relación con las tareas del proceso de desarrollo del software. Se confía en las balas de plata.

Nivel 1, obstructivo. Se imponen procesos contraproducentes. Los procesos se definen rígidamente y se adhieren a la forma que subrayan. Abundan las ceremonias rituales. La administración colectiva impide la asignación de responsabilidad. *Status quo über alles* (sobre todo).

Nivel 2, despreciador. No se preocupa por la buena ingeniería de software institucionalizada. Hay desunión completa entre actividades de desarrollo de software y actividades de mejoramiento del proceso de software y falta completa de programas de capacitación.

Nivel 3, socavación. Desprecio total por la propia organización, descrédito consciente de los esfuerzos de mejoramiento del proceso de software de los pares de la organización. Recompensa al fracaso y al pobre desempeño.

Los niveles de inmadurez de Schorsch son tóxicos para cualquier organización de software. Si usted encuentra alguno de ellos, los intentos por MPS están condenados al fracaso.

La pregunta decisiva es si las escalas de madurez, como las que se proponen como parte del CMM, proporcionan algún beneficio real. El autor cree que lo tienen. Una escala de madurez

proporciona una instantánea fácilmente comprensible de la calidad del proceso que pueden emplear los profesionales y administradores como hito desde el cual puedan planificar estrategias de mejoramiento.

30.1.3 ¿El MPS es para todos?

Durante muchos años, el MPS se vio como una actividad "corporativa", un eufemismo para algo que sólo realizan las grandes compañías. Pero hoy, un porcentaje significativo de todo el desarrollo de software lo realizan compañías que emplean menos de 100 personas. ¿Una compañía pequeña puede iniciar actividades MPS y realizarlas con éxito?

Si un modelo de proceso específico o enfoque MPS se percibe como excesivo en su organización, probablemente lo es.

Existen sustanciales diferencias culturales entre grandes y pequeñas organizaciones de desarrollo de software. No debe sorprender que las organizaciones pequeñas sean más informales, apliquen menos prácticas estándar y tiendan a la autoorganización. También tienden a enorgullecerse por la "creatividad" de los miembros individuales de la organización de software, e inicialmente ven un marco conceptual MPS como excesivamente burocrático y pesado. Sin embargo, el mejoramiento de los procesos es tan importante para una organización pequeña como para una grande.

Dentro de las organizaciones pequeñas, la implementación de un marco conceptual MPS requiere recursos que pueden tener un suministro reducido. Los administradores deben asignar personal y dinero para lograr que ocurra la ingeniería del software. Por tanto, sin importar el tamaño de la organización del software, es razonable considerar la motivación empresarial para el MPS.

Éste se aprobará e implementará sólo después de que sus proponentes demuestren apalancamiento financiero [Bir98], que se demuestra al examinar los beneficios técnicos (por ejemplo, menos defectos entregados al campo, reelaboración reducida, menores costos de mantenimiento o tiempo de llegada al mercado más rápido) y traducirlos en dinero. En esencia, deben demostrar un rendimiento realista sobre la inversión (sección 30.7) para los costos MPS.

30.2 El proceso MPS

La parte dura del MPS no es establecer las características que definen un proceso de software de alta calidad o la creación de un modelo de madurez de proceso. Ésas son relativamente sencillas. La parte dura es establecer un consenso para iniciar MPS y definir una estrategia continua a fin de implementarla a través de una organización de software.

El Software Engineering Institute desarrolló IDEAL, "un modelo de mejoramiento organizacional que funciona como mapa de caminos para iniciar, planificar e implementar acciones de mejoramiento" [SEI08]. IDEAL es representativo de muchos modelos de proceso para MPS y define cinco actividades distintas: inicio, diagnóstico, establecimiento, acción y aprendizaje, que guían a una organización a través de las actividades MPS.

En este libro se presenta un mapa de caminos un tanto diferente para MPS, con base en el modelo de proceso para MPS originalmente propuesto en [Pre88]. Aplica una filosofía de sentido común que requiere que una organización 1) se observe en el espejo, 2) se vuelva más astuta para tomar elecciones inteligentes, 3) seleccione el modelo de proceso (y los elementos tecnológicos relacionados) que satisfagan mejor sus necesidades, 4) ejemplifique el modelo en su entorno operativo y su cultura y 5) evalúe lo que se hizo. Estas cinco actividades (analizadas en las subsecciones[2] que siguen) se aplican en forma iterativa (cíclica) con la intención de fomentar el mejoramiento de proceso continuo.

2 Algunos de los contenidos de estas secciones han sido adaptados de [Pre88] con autorización.

30.2.1 Valoración y análisis de la desviación

Cualquier intento por mejorar un proceso de software existente sin primero valorar la eficacia de las actividades del marco conceptual y las prácticas de ingeniería del software asociadas sería como iniciar un largo viaje hacia una nueva localidad sin idea de dónde se comienza. Se parte con gran movimiento y se vaga por ahí intentando conseguir un rumbo, se emplea mucha energía y se padece de grandes dosis de frustración y, probablemente, se decide que en realidad no se quiere viajar de cualquier forma. Dicho de manera simple, antes de comenzar cualquier viaje, es buena idea saber precisamente dónde se está.

Asegúrese de entender sus fortalezas así como sus debilidades. Si es inteligente, construirá sobre la base de las fortalezas.

La primera actividad del mapa de caminos, llamada *valoración*, le permite adquirir rumbo. La intención de la valoración es descubrir las fortalezas y las debilidades en la forma en la que su organización aplica el proceso de software existente y las prácticas de ingeniería del software que pueblan el proceso.

La valoración examina un amplio rango de acciones y tareas que conducirán a un proceso de alta calidad. Por ejemplo, sin importar el modelo de proceso que elija, la organización de software debe establecer mecanismos genéricos como: enfoques definidos para comunicación con el cliente; establecimiento de métodos para representar requisitos de usuarios; definición de un marco conceptual de gestión del proyecto que incluya definición del ámbito, estimación, calendarización y rastreo del proyecto; métodos de análisis de riesgos; cambio de procedimientos administrativos; actividades de aseguramiento y control de la calidad que incluyan revisiones y muchas otras. Cada una se considera dentro del contexto del marco conceptual y las actividades sombrilla (capítulo 2) que se establecieron y valoraron para determinar si se responden las siguientes preguntas:

- ¿El objetivo de la acción está claramente definido?
- ¿Los productos operativos requeridos como entrada y producidos como salida se identifican y describen?
- ¿Las tareas de trabajo por realizar se describen claramente?
- ¿Las personas que deben realizar la acción se identifican por rol?
- ¿Se establecieron criterios de entrada y salida?
- ¿Se establecieron métricas para la acción?
- ¿Hay herramientas disponibles para apoyar la acción?
- ¿Existe algún programa de capacitación explícito que aborde la acción?
- ¿La acción se realiza de manera uniforme para todos los proyectos?

Aunque las preguntas anotadas implican una respuesta de *sí* o *no*, el papel de la valoración es mirar detrás de la respuesta para determinar si la acción en cuestión se realiza con las mejores prácticas.

Conforme se realiza el proceso de valoración, usted (o quienes se contraten para realizar la valoración) también deben enfocarse en los siguientes atributos:

¿Qué atributos genéricos se observan durante la valoración?

Consistencia. ¿Las actividades, acciones y tareas importantes se aplican de manera consistente a través de todos los proyectos de software y por todos los equipos de software?

Sofisticación. ¿Las acciones administrativas y técnicas se realizan con un nivel de sofisticación que implica una comprensión profunda de las mejores prácticas?

Aceptación. ¿El proceso de software y la práctica de ingeniería del software se aceptan ampliamente por parte del personal administrativo y técnico?

Compromiso. ¿La administración comprometió los recursos requeridos para lograr consistencia, sofisticación y aceptación?

La diferencia entre aplicación local y mejores prácticas representa una "brecha" que ofrece oportunidades para el mejoramiento. El grado en el cual se logren consistencia, sofisticación, aceptación y compromiso indica la cantidad de cambio cultural que se requerirá para lograr mejoría significativa.

30.2.2 Educación y capacitación

Aunque pocas personas de software cuestionan los beneficios de un proceso de software organizado y ágil o las prácticas sólidas de ingeniería del software, muchos profesionales y administradores no conocen lo suficiente acerca de alguno de los temas.[3] Como consecuencia, percepciones imprecisas de los procesos y de las prácticas conducen a decisiones inadecuadas cuando se introduce un marco conceptual MPS. Se concluye entonces que un elemento clave de cualquier estrategia MPS es la educación y capacitación de los profesionales, gerentes técnicos y gerentes ejecutivos que tengan contacto directo con la organización de software. Con ese fin, deben realizarse tres tipos de educación y capacitación:

Intente proporcionar capacitación "justo a tiempo" dirigida a las necesidades reales de un equipo de software.

Conceptos y métodos genéricos. Dirigida tanto a gerentes como a profesionales, esta categoría subraya tanto procesos como práctica. La intención es proporcionar a los profesionales las herramientas intelectuales necesarias para aplicar el proceso de software de manera efectiva y la toma de decisiones racionales acerca de las mejorías del proceso.

Tecnología y herramientas específicas. Dirigida principalmente a profesionales, esta categoría subraya las tecnologías y herramientas que se adoptaron para el uso local. Por ejemplo, si se eligió UML para modelado de análisis y diseño, se establece un programa de estudios de capacitación para ingeniería del software usando UML.

Comunicación empresarial y temas relacionados con la calidad. Dirigida a todos los participantes, esta categoría se enfoca en los temas "blandos" que ayudan a una mejor comunicación entre los participantes y fomentan un mayor foco de calidad.

En un contexto moderno, educación y capacitación pueden entregarse en varias formas distintas. Todo puede ofrecerse como parte de una estrategia MPS, desde podcast hasta capacitación basada en internet (por ejemplo, [QAI08]), pasando por DVD y cursos en aulas.

30.2.3 Selección y justificación

Conforme elija, asegúrese de considerar la cultura de su organización y el nivel de aceptación que cada elección probablemente provocará.

Una vez completada la actividad de valoración inicial[4] e iniciada la educación, una organización de software debe comenzar a elegir, lo que ocurre durante una *actividad de selección y justificación* en la que se eligen características del proceso y se especifican métodos de ingeniería del software determinados para poblar el proceso de software.

Primero, debe elegir el modelo de proceso (capítulos 2 y 3) que se ajuste mejor a su organización, a sus participantes y al software que construirán. Debe decidir cuáles de las actividades del conjunto del marco conceptual se aplicarán, los principales productos operativos que se producirán y los puntos de verificación de aseguramiento de la calidad que permitirán a su equipo valorar el progreso. Si la actividad de valoración MPS indica debilidades específicas (por ejemplo, no hay funciones SQA formales), debe enfocar su atención en las características del proceso que abordarán directamente dichas debilidades.

A continuación, desarrolle un trabajo innovador para cada actividad de marco conceptual (por ejemplo, modelado) y defina el conjunto de tareas que se aplicarán para un proyecto típico. También debe considerar los métodos de ingeniería del software que pueden aplicarse para

3 Si ha invertido tiempo en la lectura de este libro, ¡no será uno de ellos!

4 En realidad, la valoración es una actividad continua. Se realiza de manera periódica con la intención de determinar si la estrategia MPS logró sus metas inmediatas y si se establece el escenario para una futura mejoría.

lograr dichas tareas. Conforme se hagan estas elecciones, educación y capacitación deben coordinarse para garantizar el reforzamiento de la comprensión.

De manera ideal, todos trabajan en conjunto para seleccionar varios procesos y elementos tecnológicos y para moverse suavemente hacia la actividad de instalación o migración (sección 30.2.4). En realidad, la selección puede ser un camino empedrado. Con frecuencia es difícil lograr consenso entre diferentes grupos. Si un comité establece los criterios para la selección, las personas pueden discutir sin fin acerca de si los criterios son adecuados y si una elección realmente satisface los criterios que se establecieron.

Es cierto que una mala elección puede hacer más daño que bien, pero "parálisis por análisis" significa que ocurre poco progreso, tal vez, y que persisten los problemas del proceso. En tanto las características del proceso o los elementos tecnológicos tengan buena posibilidad de satisfacer las necesidades de una organización, a veces es mejor jalar el gatillo y hacer la selección, en lugar de esperar la solución óptima.

Una vez hecha la elección, deben emplearse tiempo y dinero para demostrarla dentro de una organización, y dichos gastos de recursos deben justificarse. En la sección 30.7 se presenta un análisis acerca de la justificación del costo y el rendimiento sobre la inversión para el MPS.

30.2.4 Instalación/migración

La *instalación* es el primer punto donde una organización de software siente los efectos de los cambios implementados como consecuencia del mapa de caminos MPS. En algunos casos, se recomienda un proceso completamente nuevo para una organización. Las actividades de marco conceptual, acciones de ingeniería del software y tareas de trabajo individuales deben definirse e instalarse como parte de una nueva cultura de ingeniería del software. Tales cambios representan una transición organizativa y tecnológica sustancial, y deben administrarse con mucho cuidado.

En otros casos, los cambios asociados con MPS son relativamente menores, lo que representa pequeñas modificaciones, pero significativas, a un modelo de proceso existente. A tales cambios con frecuencia se les conoce como *migración de proceso*. En la actualidad, muchas organizaciones de software tienen un "proceso" en su lugar. El problema es que no funciona de forma efectiva. Por tanto, una estrategia más efectiva es una *migración* incremental de un proceso (que no funciona tan bien como se desea) a otro proceso.

Instalación y migración en realidad son actividades de *rediseño de proceso de software* (RPS). Scacchi [Sca00] afirma que "el RPS se preocupa por la identificación, aplicación y refinamiento de nuevas formas de mejorar dramáticamente y de transformar los procesos de software". Cuando se inicia un proceso formal de RPS se consideran tres modelos de proceso diferentes: 1) el proceso existente ("como es"), 2) un proceso transicional ("de aquí a allá") y 3) el proceso meta ("por ser"). Si este último es significativamente diferente del proceso existente, el único enfoque racional de la instalación es una estrategia incremental en la que el proceso transicional se implemente en pasos. El proceso transicional ofrece una serie de puntos que permiten que la cultura de la organización de software se adapte a pequeños cambios a lo largo de un periodo.

30.2.5 Evaluación

Aunque se menciona como la última actividad en el mapa de caminos MPS, la *evaluación* ocurre a lo largo del MPS. La actividad de evaluación valora el grado en el cual los cambios se demostraron y adoptaron, el grado en el que tales cambios dan como resultado mejor calidad de software u otros beneficios tangibles de proceso y el estado global del proceso y de la cultura de la organización conforme avanzan las actividades MPS.

Durante la actividad de evaluación se consideran tanto factores cualitativos como métricas cuantitativas. Desde un punto de vista cualitativo, las actitudes anteriores de administradores y

profesionales acerca del proceso de software pueden compararse con las que tienen después de la instalación de los cambios del proceso. Las métricas cuantitativas (capítulo 25) se recopilan de proyectos que usaron el proceso transicional o del proceso "por ser" y se comparan con métricas similares que se recopilan para proyectos que se realizaron bajo el proceso "como es".

30.2.6 Gestión del riesgo para MPS

PUNTO CLAVE

El MPS con frecuencia fracasa porque los riesgos no se consideraron adecuadamente y no se tuvo un plan de contingencia.

El MPS es una empresa riesgosa. De hecho, más de la mitad de todos los esfuerzos MPS terminan en fracaso. Las razones del fracaso varían enormemente y son específicas de la organización. Entre los riesgos más comunes están: falta de apoyo administrativo, resistencia cultural por parte del personal técnico, una estrategia MPS pobremente planeada, un enfoque excesivamente formal del MPS, selección de un proceso inadecuado, falta de "adquisición" por parte de participantes clave, un presupuesto inadecuado, falta de capacitación de personal, inestabilidad de la organización, entre muchos otros factores. El papel de quienes tienen la responsabilidad del MPS es analizar los riesgos probables y desarrollar una estrategia interna para mitigarlos.

Una organización de software debe gestionar el riesgo en tres puntos clave en el proceso MPS [Sta97b]: antes del inicio del mapa de caminos MPS, durante la ejecución de las actividades MPS (valoración, educación, selección, instalación) y durante la actividad de evaluación que sigue a la ejemplificación de algunas características del proceso. En general, pueden identificarse las siguientes categorías [Sta97b] para factores de riesgo MPS: presupuesto y costo, contenido y entregables, cultura, mantenimiento de entregables MPS, misión y metas, administración de la organización, estabilidad de la organización, participantes en el proceso, calendario de desarrollo MPS, entorno de desarrollo MPS, proceso de desarrollo MPS, administración del proyecto MPS y personal MPS.

Dentro de cada categoría pueden identificarse algunos factores de riesgo genéricos. Por ejemplo, la cultura organizacional tiene un fuerte vínculo con el riesgo. Para la categoría cultura, pueden definirse los siguientes factores de riesgo genéricos[5] [Sta97b]:

- Actitud hacia el cambio, con base en esfuerzos previos por cambiar
- Experiencia con programas de calidad, nivel de éxito
- Orientación de la acción para resolver problemas frente a luchas políticas
- Uso de hechos para gestionar la organización y los negocios
- Paciencia con el cambio; habilidad para pasar tiempo socializando
- Orientación de las herramientas: esperanza de que las herramientas puedan resolver los problemas
- Nivel de "planificación": habilidad de la organización para planificar
- Habilidad de los miembros de la organización para participar abiertamente en las reuniones con varios niveles de la organización
- Habilidad de los miembros de la organización para administrar las reuniones de manera eficaz
- Nivel de experiencia en la organización con procesos definidos

Al usar los factores de riesgo y los atributos genéricos como guía, es posible calcular la exposición al riesgo en la forma siguiente:

$$\text{Exposición} = (\text{probabilidad de riesgo}) \times (\text{pérdida estimada})$$

Puede desarrollar una tabla de riesgo (capítulo 28) para aislar aquellos riesgos que garanticen mayor atención de la administración.

5 Factores de riesgo para cada una de las categorías de riesgo anotadas en esta sección pueden encontrarse en [Sta97b].

30.2.7 Factores de éxito cruciales

En la sección 30.2.6 se observó que el MPS es una empresa riesgosa y que la tasa de fracaso para las compañías que intentan mejorar su proceso es angustiosamente elevada. Los riesgos de la organización, del personal y de la administración de proyecto presentan retos para quienes dirigen cualquier esfuerzo MPS. Aunque la gestión del riesgo es importante, lo es igualmente reconocer aquellos factores cruciales que conducen al éxito.

Después de examinar 56 organizaciones de software y sus esfuerzos MPS, Stelzer y Mellis [Ste99] identifican un conjunto de factores de éxito cruciales (FEC) que deben presentarse si ha de triunfar el MPS. En esta sección se presentan los cinco principales FEC.

? ¿Qué factores de éxito cruciales son vitales para el éxito del MPS?

Compromiso y apoyo de la administración. Como la mayoría de las actividades que precipitan el cambio organizativo y cultural, el MPS triunfará sólo si la administración se involucra de manera activa. Los gerentes ejecutivos deben reconocer la importancia del software para sus compañías y ser patrocinadores activos del esfuerzo MPS. Los gerentes técnicos deben involucrarse enormemente en el desarrollo de la estrategia MPS local. Como anotan los autores del estudio: "el mejoramiento del proceso de software no es factible sin investigar tiempo, dinero y esfuerzo" [Ste99]. El compromiso y el apoyo administrativo son esenciales para sostener dicha inversión.

Involucramiento del personal. El MPS no puede imponerse de manera descendente ni desde el exterior. Si los esfuerzos MPS han de triunfar, el mejoramiento debe ser orgánico, patrocinado por gerentes técnicos y técnicos ejecutivos, y adoptado por profesionales locales.

Integración y comprensión del proceso. El proceso de software no existe en un vacío organizativo. Debe integrarse con otros procesos y requisitos empresariales. Para lograr esto, los responsables del esfuerzo MPS deben tener un conocimiento íntimo de los otros procesos empresariales. Además, deben entender el proceso de software "como es" y valorar cuánto cambio transicional es tolerable dentro de la cultura local.

Una estrategia MPS a la medida. No hay una receta para la estrategia MPS. Como se anotó anteriormente, en este capítulo, el mapa de caminos MPS debe adaptarse al entorno local: deben considerarse cultura de equipo, mezcla de producto, y fortalezas y debilidades locales.

Administración sólida del proyecto MPS. El MPS es un proyecto como cualquier otro. Involucra coordinación, calendarización, tareas paralelas, productos entregables, adaptación (cuando el riesgo se convierte en realidad), políticas, control presupuestal y mucho más. Sin administración activa y efectiva, un proyecto MPS está condenado al fracaso.

30.3 El CMMI

WebRef

Para obtener información más completa acerca del CMMI, dirigirse a **www.sei.cmu.edu/cmmi/**

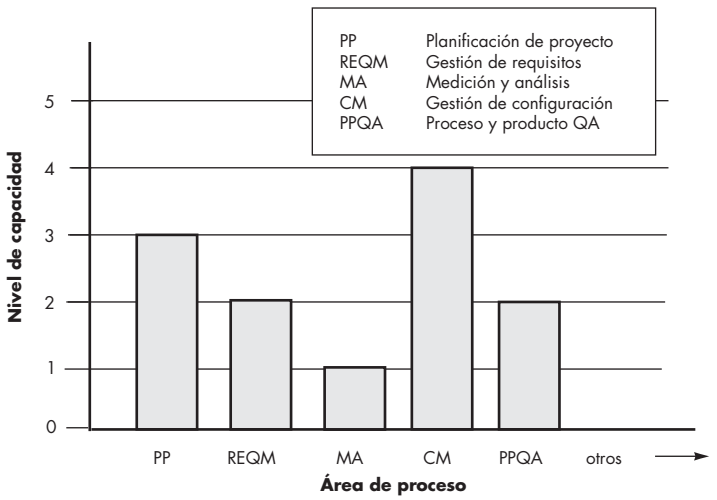
El CMM original se desarrolló y actualizó por parte del Software Engineering Institute a lo largo de los años de 1990 como un marco conceptual MPS completo. Más adelante, evolucionó en la *Integración del Modelo de Madurez de Capacidades* (CMMI) [CMM07], un metamodelo de proceso exhaustivo que se impulsa en un conjunto de sistemas y capacidades de ingeniería del software que deben presentarse conforme las organizaciones alcanzan diferentes niveles de capacidad y madurez del proceso.

El CMMI representa un metamodelo de proceso en dos formas diferentes: 1) como un modelo "continuo" y 2) como un modelo "en etapas". El metamodelo CMMI continuo describe un proceso en dos dimensiones, como se ilustra en la figura 30.2. Cada área de proceso (por ejemplo, planificación de proyecto o gestión de requisitos) se valora formalmente contra metas y prácticas específicas y se clasifica de acuerdo con los siguientes niveles de capacidad:

FIGURA 30.2

Perfil de capacidad de área de proceso CMMI

Fuente: [Phi02].



Nivel 0: *Incompleto*: el área del proceso (por ejemplo, gestión de requisitos) no se realiza o no logra todas las metas y objetivos definidos por la CMMI para la capacidad nivel 1 del área de proceso.

Nivel 1: *Realizado*: todas las metas específicas del área de proceso (como se define mediante la CMMI) están satisfechas. Se realizan las tareas de trabajo requeridas para producir productos operativos definidos.

Nivel 2: *Administrado*: se satisfacen todos los criterios del nivel 1 de capacidad. Además, todo el trabajo asociado con el área de proceso se encuentra acorde con una política definida de manera organizacional; todo el personal que realiza el trabajo tiene acceso a recursos adecuados para tener listo el trabajo; los participantes se involucran de manera activa en el área de proceso según se requiera; todas las tareas del trabajo y los productos operativos se "monitorean, controlan y revisan, y se evalúan para su adhesión a la descripción del proceso" [CMM07].

CONSEJO

Cada organización debe esforzarse por lograr el objetivo de la CMMI. Sin embargo, la aplicación de todos los aspectos del modelo puede ser exagerado.

Nivel 3: *Definido*: se logran todos los criterios del nivel 2 de capacidad. Además, el proceso se "hace a la medida, a partir del conjunto de procesos estándar y de acuerdo con los lineamientos de producción de la organización; contribuye con productos operativos, medidas y otra información para mejorar los procesos activos del proceso organizacional" [CMM07].

Nivel 4: *Administrado cuantitativamente*: se logran todos los criterios del nivel 3 de capacidad. Además, el área de proceso se controla y mejora, usando medición y valoración cuantitativa. "Los objetivos cuantitativos para el rendimiento cualitativo y de proceso se establecen y usan como criterios para gestionar el proceso" [CMM07].

Nivel 5: *Optimizado*: se logran todos los criterios del nivel 4 de capacidad. Además, el área de proceso se adapta y optimiza, usando medios cuantitativos (estadísticos) para satisfacer las necesidades cambiantes del cliente y para mejorar continuamente la eficacia del área de proceso bajo consideración.

La CMMI define cada área de proceso en términos de "metas específicas", y de "prácticas específicas" requeridas para lograr dichas metas. Las *metas específicas* establecen las características que deben existir si las actividades implicadas por un área de proceso han de ser efectivas. Las *prácticas específicas* desglosan una meta en un conjunto de actividades relacionadas con el proceso.

Por ejemplo, la **planificación del proyecto** es una de las ocho áreas de proceso definidas por la CMMI para la categoría "gestión de proyecto".[6] Las metas específicas (ME) y las prácticas específicas (PE) asociadas definidas para **planificación de proyecto** son [CMM07]:

WebRef

En **www.sei.cmu.edu/CMMI/** puede obtenerse información completa además de una versión descargable de la CMMI

ME 1 Establecimiento de estimaciones

PE 1.1-1 Estimación del ámbito del proyecto

PE 1.2-1 Establecimiento de estimaciones de producto operativo y atributos de tarea

PE 1.3-1 Definición de ciclo de vida del proyecto

PE 1.4-1 Determinación de estimaciones de esfuerzo y costo

ME 2 Desarrollo de un plan de proyecto

PE 2.1-1 Establecimiento del presupuesto y calendario

PE 2.2-1 Identificación de riesgos del proyecto

PE 2.3-1 Plan para gestión de datos

PE 2.4-1 Plan para recursos del proyecto

PE 2.5-1 Plan para conocimiento y habilidades necesarias

PE 2.6-1 Plan de involucramiento de participantes

PE 2.7-1 Establecimiento del plan del proyecto

ME 3 Obtención de compromiso del plan

PE 3.1-1 Revisión de planes que afectan el proyecto

PE 3.2-1 Reconciliación de niveles de trabajo y recursos

PE 3.3-1 Obtención de compromiso del plan

Además de las metas y prácticas específicas, la CMMI también define un conjunto de cinco metas genéricas y prácticas relacionadas para cada área de proceso. Cada una de las cinco metas genéricas corresponde a uno de los cinco niveles de capacidad. Por tanto, para lograr un nivel de capacidad particular deben lograrse la meta genérica para dicho nivel y las prácticas genéricas que correspondan a dicha meta. Para ilustrar, las metas genéricas (MG) y las prácticas genéricas (PG) para el área de proceso **planificación del proyecto** son [CMM07]:

MG 1 Logro de metas específicas

PG 1.1 Realización de prácticas base

MG 2 Institucionalización de un proceso administrado

PG 2.1 Establecimiento de una política organizacional

PG 2.2 Plan del proceso

PG 2.3 Provisión de recursos

PG 2.4 Asignación de responsabilidad

PG 2.5 Capacitación de personal

PG 2.6 Gestión de configuraciones

PG 2.7 Identificación e involucramiento de participantes relevantes

PG 2.8 Monitoreo y control del proceso

PG 2.9 Evaluación objetiva de la adhesión

PG 2.10 Revisión de estatus con administración de nivel superior

---

6 Otras áreas de proceso definidas para "gestión de proyecto" incluyen: monitoreo y control del proyecto, administración de acuerdo con proveedores, administración de proyecto integrado para IPPD, gestión del riesgo, formación de equipo integrado, gestión de proveedor integrado y gestión de proyecto cuantitativo.

MG 3 Institucionalización de un proceso definido

PG 3.1 Establecimiento de un proceso definido

PG 3.2 Recopilación de información de mejoría

MG 4 Institucionalización de un proceso administrado cuantitativamente

PG 4.1 Establecimiento de objetivos cuantitativos para el proceso

PG 4.2 Estabilización de desempeño de subprocesos

MG 5 Institucionalización de un proceso de optimización

PG 5.1 Aseguramiento del mejoramiento de proceso continuo

PG 5.2 Corrección de causas originales de problemas

El modelo CMMI por etapas define las mismas áreas de proceso, metas y prácticas que el modelo continuo. La diferencia principal es que el modelo por etapas define cinco niveles de madurez, en lugar de cinco niveles de capacidad. Para lograr un nivel de madurez deben lograrse las metas y prácticas específicas asociadas con un conjunto de áreas de proceso. La relación entre niveles de madurez y áreas de proceso se muestra en la figura 30.3.

INFORMACIÓN

La CMMI: ¿Se debe o no se debe?

La CMMI es un metamodelo de proceso. Define (en más de 700 páginas) las características de proceso que deben existir si una organización quiere establecer un proceso de software que sea completo. La pregunta que se ha debatido durante más de una década es: ¿es excesiva la CMMI? Como la mayoría de las cosas en la vida (y en el software), la respuesta no es un simple "sí" o "no".

El espíritu de la CMMI siempre debe adoptarse. Con riesgo de sobresimplifiación, se argumenta que el desarrollo del software debe tomarse con seriedad: planificarse a profundidad, controlarse de manera uniforme, rastrearse con precisión y llevarse a cabo con profesionalismo. Debe enfocarse en las necesidades de los participantes en el proyecto, en las habilidades de los ingenieros del software y en la calidad del producto final. Nadie estaría en desacuerdo con estas ideas.

Los requisitos detallados de la CMMI deben considerarse seriamente si una organización construye grandes sistemas complejos que involucran decenas o cientos de personas a lo largo de varios meses o años. Puede ser que la CMMI sea "correcta" en tales situaciones si la cultura de la organización es sensible a modelos de proceso estándar y si la administración se compromete para convertirlo en éxito. Sin embargo, en otras situaciones, la CMMI simplemente puede ser demasiado para que una organización lo asimile con éxito. ¿Esto significa que la CMMI es "mala" o "excesivamente burocrática" o "anticuada"? No..., no lo es. Simplemente significa que lo que es correcto para una cultura organizacional puede no serlo para otra.

La CMMI es un logro significativo en ingeniería del software. Proporciona un análisis amplio de las actividades y acciones que deben presentarse cuando una organización construye software de computadora. Incluso si una organización de software elige no adoptar sus detalles, todo equipo de software debe abrazar su espíritu y ganar comprensión de su análisis del proceso y de la práctica de la ingeniería del software.

30.4 El CMM de personal

El CMM de personal sugiere prácticas que mejoran la competencia y cultura de la fuerza laboral.

Un proceso de software, sin importar cuán bien se conciba, no triunfará sin personal de software talentoso y motivado. El *Modelo de Madurez de Capacidad de Personal* "es un mapa de caminos para implementar prácticas que mejoran de manera continua la capacidad de la fuerza de trabajo de una organización" [Cur02]. Desarrollado a mediado de los años de 1990 y refinado durante los años siguientes, la meta del CMM de personal es alentar el mejoramiento continuo del conocimiento de la fuerza laboral genérica (llamadas "competencias centrales"), de las habilidades específicas de los ingenieros del software y de la administración del proyecto (llamadas "competencias de la fuerza laboral") y de las habilidades relacionadas con el proceso.

Como el CMM, la CMMI y los marcos conceptuales MPS relacionados, el CMM de personal define un conjunto de cinco niveles de madurez organizativa que proporcionan un indicio de la sofisticación relativa de las prácticas y procesos de la fuerza laboral. Dichos niveles de madurez

FIGURA 30.3
Áreas de proceso requeridas para lograr un nivel de madurez
Fuente: [Phi02].

Nivel	Enfoque	Áreas de proceso
Optimización	***Mejora de proceso continua***	**Innovación y despliegue organizacional** **Análisis causal y resolución**
Administrado cuantitativamente	***Administración cuantitativa***	**Desempeño de proceso organizacional** **Administración de proyecto cuantitativa**
Definido	***Estandarización de proceso***	**Desarrollo de requisitos** **Solución técnica** **Integración de producto** **Verificación** **Validación** **Enfoque en proceso organizacional** **Definición de proceso organizacional** **Capacitación organizacional** **Administración de proyecto integrada** **Administración de proveedor integrada** **Gestión del riesgo** **Análisis de decisión y resolución** **Entorno organizacional para integración** **Formación de equipo integrado**
Administrado	***Administración básica de proyecto***	**Gestión de requisitos** **Planificación de proyecto** **Monitoreo y control del proyecto** **Administración de acuerdo con proveedor** **Medición y análisis** **Aseguramiento de calidad de proceso y producto** **Administración de configuración**
Realizado		

[CMM08] se ligan a la existencia (dentro de una organización) de un conjunto de áreas de proceso clave (APC). En la figura 30.4 se muestra un panorama de los niveles organizativos y las APC relacionadas.

El CMM de personal complementa cualquier marco conceptual MPS al alentar a una organización a nutrir y mejorar su activo más importante: su personal. Tan importante, que establece una atmósfera de fuerza de trabajo que permite a una organización de software "atraer, desarrollar y conservar talento sobresaliente" [CMM08].

30.5 OTROS MARCOS CONCEPTUALES MPS

Aunque los CMM y CMMI del SEI son los marcos conceptuales MPS de mayor aplicación, se han propuesto algunas alternativas[7] que están en uso. Entre las más ampliamente utilizadas se encuentran:

- **SPICE:** una iniciativa internacional para dar apoyo a la valoración de proceso ISO y a estándares de proceso de ciclo de vida [SPI99]
- **ISO/IEC 15504** para Valoración de Proceso (de Software) [ISO08]
- **Bootstrap:** un marco conceptual MPS para organizaciones pequeñas y medianas que se adecua a SPICE [Boo06]

7 Es razonable argumentar que algunos de estos marcos conceptuales no son tanto "alternativas" como enfoques complementarios del MPS. Una tabla exhaustiva de muchos más marcos conceptuales MPS puede encontrarse en **www.geocities.com/lbu_measure/spi/spi.htm#p2**

FIGURA 30.4
Áreas de proceso para el CMM de personal

Nivel	Enfoque	Áreas de proceso
Optimizado	***Mejoramiento continuo***	**Innovación continua de la fuerza laboral** **Alineación del desempeño organizativo** **Mejoramiento de capacidad continuo**
Predecible	***Cuantifica y gestiona conocimiento, capacidades y habilidades***	**Tutelaje** **Administración de capacidad organizativa** **Administración de desempeño cuantitativo** **Activos basados en competencia** **Grupos de trabajo fortalecidos** **Integración de competencia**
Definido	***Identifica y desarrolla conocimiento, capacidades y habilidades***	**Cultura de participación** **Desarrollo de grupos de trabajo** **Prácticas basadas en competencia** **Desarrollo profesional** **Desarrollo de competencias** **Planificación de fuerza de trabajo** **Análisis de competencia**
Administrado	***Prácticas de administración de personal básicas, repetibles***	**Compensación** **Capacitación y desarrollo** **Administración del desempeño** **Entorno laboral** **Comunicación y coordinación** **Dotación de personal**
Inicial	***Prácticas inconsistentes***	

- **PSP y TSP:** marcos conceptuales MPS individuales y específicos de equipo ([Hum97], [Hum00]) que se enfocan en procesos micro, un enfoque más riguroso del desarrollo de software acoplado con medición
- **TickIT:** método de auditoría [Tic05] que valora el cumplimiento de una organización al Estándar ISO 9001:2000

En los siguientes párrafos se presenta un breve panorama de cada uno de estos marcos conceptuales MPS. Si tiene más interés está disponible una gran variedad de recursos tanto impresos como en la web.

? Además del CMM, ¿existen otros marcos conceptuales MPS que puedan considerarse?

SPICE. El modelo SPICE (*Software Process Improvement and Capability dEtermination*: determinación de mejoramiento y capacidad del proceso de software) proporciona un marco conceptual de valoración MPS que cumple con ISO 15504:2003 e ISO 12207. La suite de documentos SPICE [SDS08] presenta un marco conceptual MPS completo, que incluye un modelo para gestión de proceso, lineamientos para realizar una valoración y clasificación del proceso bajo consideración, construcción, selección y uso de instrumentos y herramientas de valoración y capacitación para asesores.

Bootstrap. El marco conceptual MPS *Bootstrap* "se desarrolló para asegurar conformidad con el estándar ISO emergente para valoración y mejoramiento del proceso de software (SPICE) y para alinear la metodología con ISO 12207" [Boo06]. El objetivo de Bootstrap es evaluar un proceso de software, usando un conjunto de mejores prácticas de ingeniería del software como base para la valoración. Como el CMMI, Bootstrap proporciona un nivel de madurez de proceso, empleando los resultados de cuestionarios que recopilan información acerca del proceso de

software "como es" y proyectos de software. Los lineamientos MPS se basan en nivel de madurez y metas organizativas.

PSP y TSP. Aunque MPS generalmente se caracteriza como una actividad organizativa, no hay razón por la que el mejoramiento del proceso no pueda realizarse en un nivel individual o de equipo. Tanto PSP como TSP (capítulo 2) enfatizan la necesidad de recopilar datos continuamente acerca del trabajo que se realiza y de usar dichos datos para desarrollar estrategias para su mejoramiento. Watts Humphrey [Hum97], el desarrollador de ambos métodos, comenta:

> El PSP [y TSP] le mostrará cómo planificar y rastrear su trabajo y cómo producir software de alta calidad de manera consistente. Al usar PSP [y TSP], contará con los datos que muestran la efectividad de su trabajo y que identifican sus fortalezas y debilidades [...] Para tener una carrera exitosa y gratificante, necesita conocer sus capacidades y habilidades, luchar por mejorarlas y capitalizar sus talentos únicos en el trabajo que realice.

Cita:

"Las organizaciones de software muestran limitaciones significativas en su habilidad para capitalizar las experiencias obtenidas de los proyectos completados."

NASA

TickIT. El método de auditoría Ticket garantiza cumplimiento con ISO 9001:200 para software: un estándar genérico que se aplica a cualquier organización que quiera mejorar la calidad global de los productos, sistemas o servicios que ofrece. Por tanto, el estándar es directamente aplicable a organizaciones y compañías de software.

La estrategia subyacente sugerida por ISO 9001:2000 se describe de la forma siguiente [ISO01]:

> ISO 9001:2000 subraya la importancia para una organización de identificar, implementar, gestionar y mejorar continuamente la efectividad de los procesos que son necesarios para el sistema de gestión de la calidad, y de administrar las interacciones de dichos procesos con la finalidad de lograr los objetivos de la organización [...] La efectividad y eficiencia del proceso pueden valorarse a través de procesos de revisión interna o externa y evaluarse sobre una escala de madurez.

WebRef

Un excelente resumen de ISO 9001:2000 puede encontrarse en **http://praxiom.com/iso-9001.htm**

ISO 9001:2000 adoptó un ciclo de "planificar-hacer-verificar-actuar" que se aplica a los elementos de gestión de la calidad de un proyecto de software. Dentro de un contexto de software, "planificar" establece los objetivos, actividades y tareas del proceso, necesarios para lograr software de alta calidad y la resultante satisfacción del cliente. "Hacer" implementa el proceso de software (incluidas tanto actividades de marco conceptual como sombrilla). La "verificación" monitorea y mide el proceso para asegurar que se lograron todos los requisitos establecidos para gestión de la calidad. Al "actuar", se inician las actividades de mejoramiento del proceso de software que trabajan continuamente para mejorar el proceso. TickIt puede usarse a través del ciclo "planificar-hacer-verificar-actuar" a fin de garantizar que el proceso MPS avanza. Los auditores TickIT valoran la aplicación del ciclo como un precursor de la certificación ISO 9001:2000. Para un análisis detallado de ISO 9001:2000 y TickIT debe examinar [Ant06], [Tri05] o [Sch03].

30.6 Rendimiento sobre inversión de MPS

El MPS representa un trabajo duro y requiere inversión sustancial de dinero y de personal. Los administradores que aprueben el presupuesto y los recursos para MPS invariablemente plantearán la pregunta: ¿cómo sé que lograremos un rendimiento razonable por el dinero que gastemos?

Cualitativamente, quienes impulsan MPS arguyen que un proceso de software mejorado conducirá a calidad de software mejorada. Afirman que el proceso mejorado dará como resultado la implementación de mejores filtros de calidad (lo que arrojará menos defectos propagados), mejor control de cambio (que da como resultado menos caos de proyecto) y menos reelaboración técnica (lo que desemboca en menor costo y mejor tiempo de llegada al mercado).

¿Pero estos beneficios cualitativos pueden traducirse en resultados cuantitativos? La ecuación clásica de rendimiento sobre inversión (RSI) es:

$$\text{RSI} = \left[\frac{\Sigma(\textit{beneficios}) - \Sigma(\textit{costos})}{\Sigma(\textit{costos})}\right] \times 100\%$$

donde

Los *beneficios* incluyen los ahorros en costo asociados con productos de mayor calidad (menos defectos), menos reelaboración, esfuerzo reducido asociado con cambios e ingreso que se acumula por menor tiempo de llegada al mercado.

Los *costos* incluyen tanto costos MPS directos (por ejemplo, capacitación, medición) como costos indirectos asociados con un mayor énfasis en el control de la calidad y en las actividades de gestión del cambio, así como por la aplicación más rigurosa de los métodos de ingeniería del software (por ejemplo, la creación de un modelo de diseño).

En el mundo real, dichas cantidades de beneficios y costos en ocasiones son difíciles de medir con precisión y están abiertos a interpretación. Pero ello no significa que una organización de software debe realizar un programa MPS sin análisis cuidadoso de los costos y beneficios que acumula. Un tratamiento amplio de RSI para MPS puede encontrarse en un libro inigualable de David Rico [Ric04].

30.7 Tendencias MPS

Durante las dos décadas pasadas, muchas compañías intentaron mejorar sus prácticas de ingeniería del software al aplicar un marco conceptual MPS para efectuar cambio organizacional y transición tecnológica. Como se observó anteriormente, en este capítulo, más de la mitad fracasan en esta labor. Sin importar el éxito o el fracaso, todos gastan una cantidad significativa de dinero. David Rico [Ric04] reporta que una aplicación típica de un marco conceptual MPS como el CMM SEI, ¡puede costar entre $25 000 y $70 000 por persona y tardar años en completarse! No debe sorprenderle que el futuro de MPS deba enfatizar un enfoque menos costoso y consumidor de tiempo.

Para ser efectivo en el mundo de desarrollo del software del siglo XXI, los futuros marcos conceptuales MPS deben volverse significativamente más ágiles. En lugar de un enfoque centrado en la organización (que puede tardar años en completarse exitosamente), los esfuerzos MPS contemporáneos deben enfocarse en el nivel del proyecto y trabajar para mejorar un proceso de equipo en semanas, no meses ni años. Para lograr resultados significativos (incluso en el nivel del proyecto) en un tiempo corto, los modelos de marco conceptual complejos pueden dar lugar a modelos más simples. En lugar de decenas de prácticas clave y cientos de prácticas complementarias, un marco conceptual MPS ágil enfatiza solamente algunas prácticas esenciales (por ejemplo, análogas a las actividades de marco conceptual estudiadas a lo largo de este libro).

Cualquier intento de MPS demanda una fuerza laboral conocedora, pero los gastos de educación y capacitación pueden ser onerosos y deben minimizarse (y simplificarse). En lugar de cursos en aulas (costosos y consumidores de tiempo), los esfuerzos MPS futuros deben apoyarse en capacitación basada en web que se dirija a prácticas esenciales. En lugar de intentos a largo plazo por cambiar la cultura de la organización (con todos los peligros políticos que conllevan), el cambio cultural debe ocurrir como se hace en el mundo real: un pequeño grupo a la vez hasta alcanzar un punto de inflexión.

El trabajo MPS de las dos décadas anteriores tiene mérito significativo. Los marcos conceptuales y los modelos que se desarrollaron representan activos intelectuales sustanciales para la

comunidad de la ingeniería del software. Pero, como todas las cosas, dichos activos guían los intentos futuros de MPS no al convertirse en un dogma recurrente, sino al funcionar como la base para modelos MPS mejores, más simples y más ágiles.

30.8 Resumen

Un marco conceptual de mejoramiento del proceso de software define las características que debe presentar si debe lograrse un proceso de software efectivo, un método de valoración que ayuda a determinar si dichas características están presentes y una estrategia para auxiliar a una organización de software a implementar dichas características de proceso que se encuentren debilitadas o que falten. Sin importar los grupos que defienden el MPS, la meta es mejorar la calidad del proceso y, en consecuencia, la calidad y la puntualidad en la entrega del software.

Un modelo de madurez de proceso proporciona un indicio global de la "madurez del proceso" que muestra una organización del software. Asimismo, proporciona un sentimiento cualitativo sobre la efectividad relativa del proceso de software que se usa actualmente.

El mapa de caminos MPS comienza con la valoración, una serie de actividades de evaluación que descubren tanto fortalezas como debilidades en la forma en que la organización aplica el proceso de software existente y las prácticas de ingeniería del software que pueblan el proceso. Como consecuencia de la valoración, una organización de software puede desarrollar un plan MPS global.

Unos de los elementos clave de cualquier plan MPS son la educación y la capacitación, actividades que se enfocan en mejorar el nivel de conocimiento de administradores y profesionales. Una vez que el personal está versado en las tecnologías de software actuales, comienza la selección y la justificación. Dichas tareas conducen a elegir la arquitectura del proceso de software, los métodos que lo pueblan y las herramientas que soporta. Instalación y evaluación son actividades MPS que ejemplifican los cambios del proceso y que valoran su eficacia e impacto.

Para mejorar exitosamente su proceso de software, una organización debe mostrar las siguientes características: compromiso y apoyo de los administradores para el MPS, involucramiento del personal a lo largo del proceso MPS, integración del proceso en la cultura organizacional global, una estrategia MPS que se haya hecho a la medida de las necesidades locales y administración sólida del proyecto MPS.

En la actualidad se usan algunos marcos conceptuales MPS. Los CMM y CMMI de SEI se usan ampliamente. El CMM de personal se particulariza para valorar la calidad de la cultura de la organización y del personal que la puebla. SPICE, Bootstrap, PSP, TSP y TickIT son marcos conceptuales adicionales que pueden conducir a MPS efectivo.

El MPS representa un trabajo duro que requiere inversión sustancial de dinero y de personal. Para garantizar que se logre un rendimiento razonable sobre la inversión, una organización debe medir los costos asociados con el MPS y los beneficios que pueden atribuírsele de manera directa.

Problemas y puntos por evaluar

30.1. ¿Por qué las organizaciones de software con frecuencia luchan cuando se embarcan en un esfuerzo por mejorar el proceso de software local?

30.2. Con sus palabras, describa el concepto de "madurez de proceso".

30.3. Realice una investigación (verifique el sitio web SEI) y determine la distribución de madurez de proceso para organizaciones de software en Estados Unidos y el mundo.

30.4. Usted trabaja para una organización de software muy pequeña: sólo 11 personas se involucran en el desarrollo del software. ¿MPS es para usted? Explique su respuesta.

30.5. La valoración es análoga a un examen médico anual. Con un examen médico como metáfora, describa la actividad de valoración MPS.

30.6. ¿Cuál es la diferencia entre un proceso "como es", un proceso "de aquí a allá" y un proceso "por ser"?

30.7. ¿Cómo se aplica la gestión del riesgo dentro del contexto de MPS?

30.8. Seleccione uno de los factores cruciales de éxito anotados en la sección 30.2.7. Realice investigación y escriba un breve ensayo acerca de cómo puede lograrse.

30.9. Realice investigación y explique cómo difiere la CMMI de su predecesor, el CMM.

30.10. Seleccione uno de los marcos conceptuales MPS que estudió en la sección 30.5 y escriba un breve ensayo que lo describa con más detalle.

Lecturas y fuentes de información adicionales

Uno de los recursos de información más exhaustivo y de más fácil acceso acerca de MPS lo desarrolló el Software Engineering Institute y está disponible en **www.sei.cmu.edu**. El sitio web de SEI contiene cientos de artículos, estudios y descripciones detalladas acerca de marco conceptual MPS.

Durante los años recientes se han agregado algunos libros valiosos a una amplia literatura desarrollada durante las dos décadas anteriores. Land (*Jumpstart CMM/CMMI Software Process Improvements*, Wiley-IEEE Computer Society, 2007) fusiona los requisitos definidos como parte de los CMM e CMMI de SEI con los estándares de ingeniería de software del IEEE, con énfasis en la intersección de proceso y práctica. Mutafelija y Stromberg (*Systematic Process Improvement Using ISO 9001:2000 and CMMI*, Artech House Publishers, 2007) estudian los marcos conceptuales MPS de ISO 9001:2000 e CMMI, y su "sinergia". Conradi *et al.* (*Software Process Improvement: Results and Experiencia from the Field*, Springer, 2006) presentan los resultados de una serie de estudios de caso y experimentos relacionados con MPS. Van Loon (*Process Assessment and Improvement: A Practical Guide for Managers, Quality Professionals and Assessors*, Springer, 2006) estudia el MPS dentro del contexto de ISO/IEC 15504. Watts Humphrey (*PSP*, Addison-Wesley, 2005, y *TSP*, Addison-Wesley, 2005) aborda su marco conceptual MPS de proceso de personal de equipo y su marco conceptual MPS de proceso de equipo de software en dos libros separados. Fantina (*Practical Software Process Improvement*, Artech House Publishers, 2004) brinda lineamiento pragmático con énfasis en CMMI/CMM.

En internet, está disponible una gran variedad de fuentes de información acerca del mejoramiento del proceso de software. Una lista actualizada de referencias existentes en la World Wide Web que son relevantes para MPS puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

31

TENDENCIAS EMERGENTES EN INGENIERÍA DEL SOFTWARE

CONCEPTOS CLAVE

bloques constructores 703
ciclo de promoción excesiva. 698
ciclo de vida de innovación 697
complejidad. 700
desarrollo colaborativo 707
desarrollo impulsado por modelo 709
desarrollo impulsado por pruebas 710
direcciones de la tecnología. 704
diseño posmoderno 710
evolución tecnológica. 696
fuente abierta. 703
herramientas 711
ingeniería de requerimientos 708
requerimientos emergentes. 701
software de mundo abierto. 701
tendencias blandas 699

A lo largo de la relativamente breve historia de la ingeniería del software, los profesionales e investigadores desarrollaron una colección de modelos de proceso, métodos técnicos y herramientas automatizadas con la intención de fomentar el cambio fundamental en la forma de construir el software de computadoras. Aunque las experiencias anteriores indican otra cosa, existe un deseo tácito por encontrar la panacea: el proceso mágico o la tecnología trascendente que permitirá construir con facilidad grandes y complejos sistemas basados en software, sin confusión, sin errores, sin demora, sin los muchos problemas que todavía plagan el trabajo de software.

Pero la historia indica que la búsqueda de la panacea parece condenada al fracaso. Las nuevas tecnologías se introducen regularmente, publicitadas con exceso como una "solución" a muchos de los problemas que enfrentan los ingenieros del software y se incorporan en los proyectos, grandes y pequeños. Los expertos de la industria resaltan la importancia de estas "nuevas" tecnologías de software, los conocedores de la comunidad del software las adoptan con entusiasmo y, a final de cuentas, tienen un papel en el mundo de la ingeniería del software. Pero tienden a no cumplir su promesa y, como consecuencia, la búsqueda continúa.

Mili y Cowan [Mil00b] comentan acerca de los retos que se afrontan cuando se intenta aislar tendencias tecnológicas significativas:

> ***¿Qué factores determinan el éxito de una tendencia?*** ¿Qué caracteriza el éxito de las tendencias tecnológicas? ¿Su mérito técnico? ¿Su habilidad para abrir nuevos mercados? ¿Su pericia para alterar la economía de los mercados existentes?
>
> ***¿Qué ciclo de vida sigue una tendencia?*** Mientras que la visión tradicional es que las tendencias evolucionan a lo largo de un ciclo de vida predecible bien definido, que avanza de una idea de investigación a un producto terminado a través de un proceso de transferencia, se descubre que muchas tendencias actuales provocaron corto circuito en este ciclo o siguieron otro.
>
> ***¿Con cuánta anticipación puede identificarse una tendencia exitosa?*** Si se sabe cómo identificar los factores de éxito y/o se entiende el ciclo de vida de una tendencia, entonces se busca

UNA MIRADA RÁPIDA

¿Qué es? Nadie puede predecir el futuro con absoluta certeza. Pero es posible valorar las tendencias en el área de la ingeniería del software y, de dichas tendencias, sugerir posibles direcciones para la tecnología. Eso es lo que se intenta hacer en este capítulo.

¿Quién lo hace? Quienquiera que desee emplear el tiempo para situarse frente a los conflictos de la ingeniería del software puede intentar predecir la futura dirección de la tecnología.

¿Por qué es importante? ¿Por qué los antiguos reyes contrataban adivinos? ¿Por qué las grandes corporaciones multinacionales contratan firmas consultoras y a expertos para preparar pronósticos? ¿Por qué un sustancial porcentaje del público lee horóscopos? Todos ellos quieren conocer lo que está por venir para estar preparados.

¿Cuáles son los pasos? No hay una fórmula para predecir el futuro. Se intenta hacer esto al recopilar datos, organizarlos para proporcionar información útil, examinar asociaciones sutiles para extraer conocimiento y, a partir de este conocimiento, sugerir probables tendencias que predigan cómo serán las cosas en algún tiempo por venir.

¿Cuál es el producto final? Una visión del futuro cercano, que puede o no ser correcta.

¿Cómo me aseguro de que lo hice bien? Predecir el camino que está adelante es un arte, no una ciencia. De hecho, es muy raro cuando una predicción seria acerca del futuro es absolutamente correcta o inequívocamente errónea (con excepción, afortunadamente, de las predicciones del fin del mundo). Se observan las tendencias y se intenta extrapolarlas. Lo acertado de la extrapolación sólo puede valorarse conforme pasa el tiempo.

¿Cuáles son las "grandes preguntas" cuando se considera la evolución tecnológica?

identificar signos tempranos del éxito de una tendencia. De manera retórica se busca la habilidad de reconocer la siguiente tendencia antes que todos los demás.

¿Qué aspectos de la evolución son controlables? ¿Las corporaciones pueden usar su influencia en el mercado para imponer tendencias? ¿Qué papel juegan los estándares en la definición de tendencias? El análisis cuidadoso de Ada contra Java, por ejemplo, debe ser iluminador a este respecto.

No existen respuestas sencillas a estas preguntas, y puede no haber discusión acerca de que intentos anteriores por identificar tecnologías significativas fueron mediocres, cuando mucho.

En ediciones anteriores de este libro (durante los 30 años anteriores) se analizaron las tecnologías emergentes y su impacto proyectado sobre la ingeniería del software. Algunas se han adoptado ampliamente, pero otras nunca alcanzaron su potencial. La conclusión del autor es que las tecnologías vienen y van; las tendencias reales que se exploran son las más blandas. Por esto se entiende que el progreso en la ingeniería del software se guiará por las tendencias empresariales, organizativas, de mercado y culturales. Dichas tendencias conducen a innovación tecnológica.

En este capítulo se observarán algunas tendencias tecnológicas de la ingeniería del software, pero el énfasis principal se colocará en algunas tendencias empresariales, organizativas, de mercado y culturales que pueden tener una importante influencia sobre la tecnología de la ingeniería del software durante los próximos 10 o 20 años.

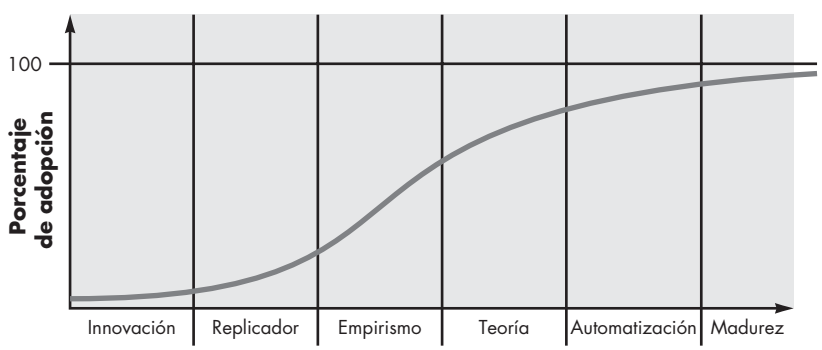
31.1 Evolución tecnológica

En un libro fascinante que ofrece una atractiva mirada sobre la evolución de las tecnologías de computación (y otras relacionadas), Ray Kurzweil [Kur05] argumenta que la evolución tecnológica es similar a la evolución biológica, pero que ocurre a un ritmo más rápido. La evolución (biológica o tecnológica) ocurre como resultado de realimentación positiva: "los métodos más capaces que resultan de una etapa del avance evolutivo se usan para crear la siguiente etapa" [Kur06].

Las grandes preguntas para el siglo XXI son: 1) ¿cuán rápidamente evoluciona la tecnología? 2) ¿cuán significativos son los efectos de la realimentación positiva? 3) ¿cuán profundos serán los cambios resultantes?

Cuando se introduce una nueva tecnología exitosa, el concepto inicial se mueve a través de un "ciclo de vida de innovación" [Gai95] razonablemente predecible, que se ilustra en la figura 31.1. En la fase de *innovación* se reconoce un problema y se realizan intentos repetidos para encontrar una solución viable. En algún punto, una solución se muestra como prometedora. El trabajo de innovación inicial se reproduce en la fase de *replicador* y obtiene un uso más amplio. El *empirismo* conduce a la creación de reglas empíricas que gobiernan el uso de la tecnología y su éxito repetido conduce a una *teoría* más amplia que da paso a la creación de herramientas

Figura 31.1
Ciclo de vida de una innovación tecnológica



Cita:

"Las predicciones son muy difíciles de hacer, en especial cuando tratan acerca del futuro."

Mark Twain

automatizadas durante la fase de *automatización*. Finalmente, la tecnología madura y se usa ampliamente.

El lector observará que muchas tendencias de investigación y tecnológicas nunca llegan a la madurez. De hecho, la gran mayoría de las tecnologías "prometedoras" en el dominio de la ingeniería del software reciben amplio interés durante algunos años y luego caen en un nicho de uso por parte de un grupo fiel de adherentes. Esto no quiere decir que dichas tecnologías carecen de mérito, sino más bien busca enfatizar que el viaje a través del ciclo de vida de la innovación es largo y duro.

PUNTO CLAVE

La tecnología de computación evoluciona a una tasa exponencial y su crecimiento pronto puede volverse explosivo.

Kurzweil [Kur05] está de acuerdo en que las tecnologías de computación evolucionan a través de una "curva S" que muestra crecimiento relativamente lento durante los años formativos de la tecnología, rápida aceleración durante su periodo de crecimiento y luego un periodo de nivelación conforme la tecnología llega a sus límites. Pero la computación y otras tecnologías relacionadas muestran crecimiento explosivo (exponencial) durante las etapas centrales que se muestran en la figura 31.1 y continuarán haciéndolo. Además, conforme una curva S termina, otra la sustituye con crecimiento incluso más explosivo durante su periodo de crecimiento.[1] En la actualidad, estamos en la rodilla de la curva S para las modernas tecnologías de computación, en la transición entre el crecimiento temprano y el crecimiento explosivo que sigue. La implicación es que, durante los próximos 20 o 40 años, se verán cambios dramáticos (incluso enloquecedores) en la capacidad de computación. Las décadas por venir darán como resultado cambios en la rapidez, tamaño, capacidad y consumo de energía de cómputo (por mencionar sólo algunas características).

Kurzweil [Kur05] sugiere que, dentro de 20 años, la evolución tecnológica acelerará a un ritmo cada vez más rápido, lo que a final de cuentas conducirá a una era de inteligencia no biológica que se fusionará con y extenderá la inteligencia humana en formas que son fascinantes de imaginar.

Y todo esto, sin importar cómo evolucione, requerirá software y sistemas que harán que, en comparación, los esfuerzos actuales parezcan infantiles. Hacia el año 2040, una combinación de computación extrema, nanotecnología, redes ubicuas con ancho de banda masivamente elevado, y robótica conducirán a un mundo diferente.[2] El software, posiblemente en formas que todavía no pueden comprenderse, continuará residiendo en el centro de este nuevo mundo. La ingeniería del software no se irá.

31.2 Observación de las tendencias en ingeniería del software

Cita:

"Creo que hay un mercado mundial para acaso cinco computadoras."

Thomas Watson, ejecutivo de IBM, 1943

La sección 31.1 consideró brevemente las fascinantes posibilidades que pueden acumularse a partir de tendencias a largo plazo en computación y tecnologías relacionadas. Pero ¿y en el corto plazo?

Barry Boehm [Boe08] sugiere que "los ingenieros del software enfrentarán los, con frecuencia, formidables desafíos de lidiar con rápidos cambios, incertidumbre y emergencia, dependencia, diversidad e interdependencia, pero que también tendrán oportunidades de realizar significativas aportaciones que harán la diferencia a fin de mejorar". Pero, ¿cuáles son las tendencias que le permitirán enfrentar dichos desafíos en los años por venir?

---

1 Por ejemplo, los límites de los circuitos integrados pueden alcanzarse dentro de la siguiente década, pero dicha tecnología puede sustituirse por tecnologías de cómputo molecular y vendrá otra curva S acelerada.

2 Kurzweil [Kur05] presenta un argumento técnico razonado que predice una fuerte inteligencia artificial (que pasará la prueba de Turing) hacia 2029 y sugiere que la evolución de los humanos y las máquinas comenzará a fusionarse hacia 2045. La gran mayoría de los lectores de este libro vivirán para ver si esto, en realidad, llega a suceder.

En la introducción a este capítulo se señaló que las "tendencias blandas" tienen un impacto significativo sobre la dirección global de la ingeniería del software. Pero otras tendencias ("más duras") orientadas a la investigación y la tecnología siguen siendo importantes. Las tendencias en investigación "están impulsadas por las percepciones generales del estado del arte y de la práctica, por percepciones del investigador acerca de las necesidades de los profesionales, por programas de financiamiento nacional que apresuran las metas estratégicas específicas y por mero interés técnico" [Mil00a]. Las tendencias tecnológicas ocurren cuando las tendencias en investigación se extrapolan para satisfacer necesidades industriales y cuando se les da forma de acuerdo con las demandas que impulsa el mercado.

PUNTO CLAVE

El "ciclo de promoción excesiva" presenta una visión realista de la integración tecnológica a corto plazo. Sin embargo, la tendencia a largo plazo es exponencial.

En la sección 31.1 se estudia el modelo de curva S para evolución tecnológica. La curva S es adecuada para considerar los efectos a largo plazo de las tecnologías centrales conforme evolucionan. ¿Pero qué hay acerca de las innovaciones, herramientas y métodos más modestos para el corto plazo? El Gartner Group [Gar08], un grupo consultor que estudia las tendencias tecnológicas a través de muchas industrias, desarrolló un *ciclo de promoción excesiva para tecnologías emergentes*, que se representa en la figura 31.2. El ciclo del Gartner Group muestra cinco fases:

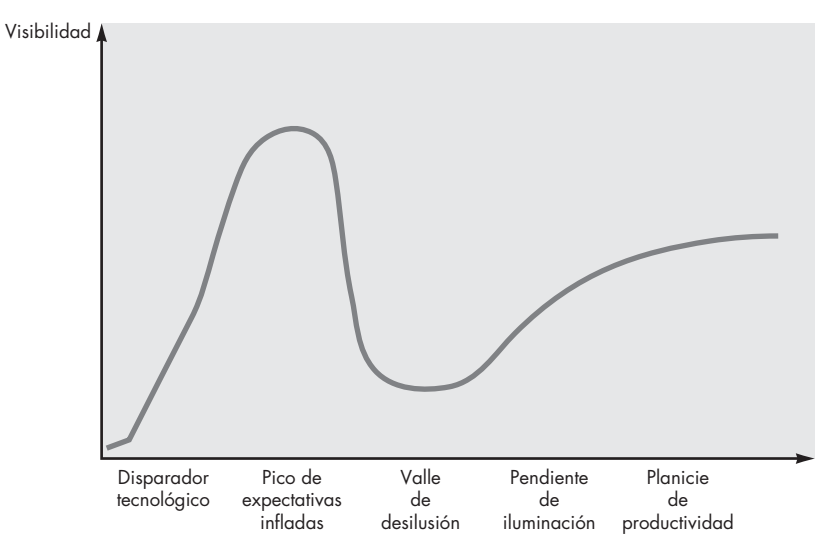
- *Disparador tecnológico:* un hallazgo de investigación o lanzamiento de un nuevo producto innovador que conduce a cobertura de los medios y a entusiasmo del público.
- *Pico de expectativas infladas:* entusiasmo exagerado y proyecciones demasiado optimistas del impacto con base en éxitos limitados, pero bien publicitados.
- *Desilusión:* las proyecciones de impacto demasiado optimistas no se satisfacen y los críticos comienzan a presionar; la tecnología pasa de moda entre los conocedores.
- *Pendiente de iluminación:* el uso creciente mediante una amplia variedad de compañías conduce a una mejor comprensión del verdadero potencial de la tecnología; surgen métodos y herramientas comerciales para apoyar la tecnología.
- *Planicie de productividad:* los beneficios en el mundo real ahora son obvios y el uso penetra en un significativo porcentaje del mercado potencial.

No toda la tecnología de ingeniería del software logra pasar a través del ciclo de promoción excesiva. En algunos casos, la desilusión se justifica y la tecnología se relega a la oscuridad.

FIGURA 31.2

Ciclo de promoción excesiva del Gartner Group para tecnologías emergentes

Fuente: [Gar08].



31.3 IDENTIFICACIÓN DE "TENDENCIAS BLANDAS"

Cita:

"640K deben ser suficientes para cualquiera."

Bill Gates, presidente de Microsoft, 1981

Cada nación con una sustancial industria de TI tiene un conjunto de características únicas que definen la forma en la que se dirigen los negocios, la dinámica organizacional que surge dentro de una compañía, los distintos conflictos de mercadeo que se aplica a los clientes locales y la decisiva cultura que dicta toda interacción humana. Sin embargo, algunas tendencias en cada una de dichas áreas son universales y tienen mucho que ver con sociología, antropología y psicología de grupos (que con frecuencia se conocen como "ciencias blandas"), como tienen que ver con la investigación académica o industrial.

Conectividad y colaboración (habilitadas por comunicación con alto ancho de banda) ya condujeron a equipos de software que no ocupan el mismo espacio físico (trabajo a distancia y empleo de tiempo parcial en un contexto local). Un equipo colabora con otros equipos que están separados por zonas horarias, lenguaje nativo y cultura. La ingeniería del software debe responder con un modelo de proceso que abarque a "equipos distribuidos" y que sea suficientemente ágil para satisfacer las demandas de inmediatez, pero suficientemente disciplinado para coordinar grupos dispares.

? ¿Qué tendencias blandas impactarán las tecnologías relacionadas con la ingeniería del software?

La *globalización* conduce a una fuerza de trabajo diversa (en idioma, cultura, resolución de problemas, filosofía administrativa, prioridades de comunicación e interacción persona a persona). Esto, a su vez, demanda una estructura organizativa flexible. Diferentes equipos (en distintos países) deben responder a los problemas de ingeniería de manera que se acomode mejor a sus necesidades únicas y, al mismo tiempo, fomentar un nivel de uniformidad que permita el avance de un proyecto global. Este tipo de organización sugiere menos niveles de administración y un mayor énfasis en la toma de decisiones por parte de cada equipo. Puede conducir a mayor agilidad, pero sólo si los mecanismos de comunicación se establecen de modo que todo equipo pueda entender el proyecto y su estado técnico (vía groupware en red) en cualquier momento. Los métodos y herramientas de ingeniería del software pueden ayudar a lograr cierto nivel de uniformidad (equipos que hablan el mismo "idioma", implementado a través de métodos y herramientas específicos). El proceso de software puede proporcionar el marco conceptual para la ejemplificación de estos métodos y herramientas.

En algunas regiones del planeta (Estados Unidos y Europa son ejemplos), la población está envejeciendo. Esta demografía innegable (y tendencia cultural) implica que muchos ingenieros y gerentes de software experimentados dejarán el campo de trabajo en la próxima década. La comunidad de la ingeniería del software debe responder con mecanismos viables que capturen el conocimiento de estos gerentes y técnicos que envejecen [por ejemplo, el uso de *patrones* (capítulo 12) es un paso en la dirección correcta], de modo que quede disponible para generaciones futuras de trabajadores del software. En otras regiones del mundo se multiplica el número de jóvenes disponibles para la industria del software. Esto proporciona una oportunidad para moldear una cultura de ingeniería del software sin la carga de 50 años de prejuicios "de la vieja escuela".

Se estima que más de mil millones de nuevos consumidores entrarán al mercado laboral mundial en la próxima década. Los consumidores que gastan en "economías emergentes superarán por mucho los US$9 billones" [PET06]. Hay poca duda de que un porcentaje no trivial de este gasto se aplicará a productos y servicios que tengan un componente digital, que se basen en software o se impulsen mediante él. La implicación: una creciente demanda para nuevo software. Entonces la pregunta es: ¿pueden desarrollarse nuevas tecnologías de ingeniería del software para satisfacer esta demanda mundial? Las modernas tendencias de mercado con frecuencia se impulsan mediante el lado suministro.[3] En otros casos, los requerimientos del lado

3 El lado suministro adopta un enfoque de mercado "construye y ellos vendrán". Se crean tecnologías únicas y los consumidores van en masa para adoptarlas... ¡a veces!

de la demanda impulsan el mercado. En cualquier caso, ¡un ciclo de innovación y demanda avanza de manera que en ocasiones hace difícil determinar cuál viene primero!

Finalmente, la cultura humana en sí impacta la dirección de la ingeniería del software. Toda generación establece su propia huella sobre la cultura y las próximas no serán diferentes. Faith Popcorn [Pop08], un consultor bien conocido que se especializa en tendencias culturales, las caracteriza en la forma siguiente: "nuestras tendencias no son fruslerías. Ellas perduran, evolucionan. Representan fuerzas subyacentes, causas primeras, necesidades humanas básicas, actitudes, aspiraciones. Nos ayudan a navegar por el mundo, a comprender lo que sucede y por qué, y nos preparan para lo que está por venir". Un análisis detallado de la manera en la que las modernas tendencias culturales tendrán un impacto sobre la ingeniería del software se deja para quienes se especializan en las "ciencias blandas".

31.3.1 Administración de la complejidad

Cuando se escribió la primera edición de este libro (1982), no existían los productos digitales al consumidor como se conocen en la actualidad, y los sistemas basados en mainframe que contenían un millón de líneas de código fuente (LOC) se consideraban muy grandes. Hoy no es raro que pequeños dispositivos digitales abarquen entre 60 000 y 200 000 líneas de software a la medida, acoplado con algunos millones de LOC para características de sistema operativo. Los modernos sistemas basados en computadora que contienen de 10 a 50 millones de líneas de código no son raros.[4] En el futuro relativamente cercano, comenzarán a surgir sistemas[5] que requieran más de mil millones de LOC.[6]

¡Piense en ello por un momento!

Considere las interfaces para un sistema de mil millones de LOC, tanto en el mundo exterior como en otros sistemas interoperables, para la internet (o su sucesor) y para los millones de componentes internos que deben trabajar en conjunto para hacer que opere exitosamente este monstruo de computación. ¿Existe alguna forma confiable de garantizar que todas estas conexiones permitirán que la información fluya de manera adecuada?

Considere el proyecto en sí. ¿Cómo se administra el flujo de trabajo y se rastrea el progreso? ¿Los enfoques convencionales escalarán hacia arriba en órdenes de magnitud?

Considere el número de personas (y sus ubicaciones) que harán el trabajo, la coordinación del personal y la tecnología, el imparable flujo de cambios, la probabilidad de una multiplataforma, en torno de un sistema multioperativo. ¿Existe alguna forma de administrar y coordinar al personal que trabaje en un proyecto monstruoso?

Considere el desafío de ingeniería. ¿Cómo pueden analizarse decenas de miles de requerimientos, limitaciones y restricciones de manera que se garantice que inconsistencia y ambigüedad, omisiones y errores categóricos se descubran y corrijan? ¿Cómo puede crearse una arquitectura de diseño que sea suficientemente robusta para manejar un sistema de este tamaño? ¿Cómo pueden los ingenieros del software establecer un sistema de gestión del cambio que tendrá que manipular cientos de miles de cambios?

Considere el reto de asegurar la calidad. ¿Cómo puede realizarse la verificación y la validación en forma significativa? ¿Cómo se pone a prueba un sistema de mil millones de LOC?

En los primeros días, los ingenieros de software intentaban administrar la complejidad en lo que sólo puede describirse como una forma *ad hoc*. En la actualidad, se usan procesos, métodos

Cita:

"No hay razón por la que alguien quiera una computadora en su casa."

Ken Olson, presidente y fundador de Digital Equipment Corp., 1977

4 Por ejemplo, los modernos sistemas operativos PC (como Linux, MacOS y Windows) tienen entre 30 y 60 millones de LOC. El software de sistema operativo para dispositivos móviles puede superar 2 millones de LOC.

5 En realidad, este "sistema" será un sistema de sistemas: cientos de aplicaciones interoperativas que trabajen en conjunto para lograr algún objetivo global.

6 No todos los sistemas complejos son grandes. Una aplicación relativamente pequeña (por ejemplo, menos de 100 000 LOC) todavía puede ser excesivamente compleja.

y herramientas para mantener bajo control la complejidad. ¿Pero mañana? ¿El enfoque actual depende de la tarea?

31.3.2 Software de mundo abierto

Conceptos tales como inteligencia ambiental,[7] aplicaciones conscientes del contexto y computación dominante/ubicua se enfocan todos en integrar sistemas basados en software en un entorno mucho más amplio que una PC, un dispositivo de computación móvil o cualquier otro dispositivo digital. Estas versiones separadas del futuro cercano de la computación sugieren de manera colectiva "software de mundo abierto", que se diseña para adaptarse a un entorno en cambio continuo "al autorganizar su estructura y autoadaptar su comportamiento" [Bar06].

El software de mundo abierto abarca inteligencia ambiental, aplicaciones conscientes del contexto y computación dominante.

Para ayudar a ilustrar los desafíos que enfrentarán los ingenieros de software en el futuro previsible, considere la noción de *inteligencia ambiental* (amI). Ducatel [Duc01] define la amI en la forma siguiente: "Las personas están rodeadas por interfaces intuitivas inteligentes que se incrustan en todo tipo de objetos. El entorno de la inteligencia ambiental es capaz de reconocer y responder a la presencia de diferentes individuos [mientras trabajan] sin obstrucciones y de manera continua".

Examine una visión del futuro cercano en el que la amI se ha vuelto ubicua. Usted acaba de comprar un comunicador personal (llamado P-com, un dispositivo móvil de bolsillo) y pasó las semanas anteriores creando[8] su "imagen": todo, desde su calendario diario, lista de actividades por hacer, libreta de direcciones, registros médicos, información relacionada con negocios, documentos de viaje, lista de deseos (cosas que busca, por ejemplo, un libro específico, una botella de vino difícil de conseguir, un curso local de soplado de vidrio) y su "Yo-digital" (D-me), que lo describe con un nivel de detalle que permite una presentación digital a otros (una especie de *MySpace* o *FaceBook* que se mueve con usted). El P-com contiene un identificador personal llamado "clave de claves", un identificador personal multifuncional que proporciona acceso y permite consultas desde un amplio rango de dispositivos amI y sistemas.

Debe ser obvio que los temas significativos de privacidad y seguridad entran en juego. Un "sistema de gestión de confianza" [Duc01] será parte integral de amI y gestionará los privilegios que permitan la comunicación con los sistemas de redes, salud, entretenimiento, finanzas, empleo y personal.

Los nuevos sistemas con capacidad amI se agregarán a la red constantemente, y cada uno ofrecerá capacidades útiles y demandará acceso a su P-com. Por tanto, el software P-com debe diseñarse de modo que pueda adaptarse a los requerimientos que surgen cuando algún nuevo sistema amI entra en línea. Existen muchas formas de lograr esto, pero la línea de referencia es la siguiente: el software P-com debe ser flexible y robusto en formas que el software convencional no puede relacionar.

31.3.3 Requerimientos emergentes

Al comienzo de un proyecto de software, existe una verdad obvia que se aplica por igual a todo participante involucrado: "no sabes lo que no sabes". Esto significa que los clientes rara vez definen requerimientos "estables". También significa que los ingenieros del software no siempre pueden prever dónde yacen las ambigüedades e inconsistencias. Los requerimientos cambian, pero eso no es algo nuevo.

---

7 Una valiosa y muy detallada introducción a la inteligencia ambiental puede encontrarse en **www.emerging-communication.com/volume6.html**. Puede obtener más información en **www.ambientintelligence.org/**

8 Toda interacción con el P-com ocurre mediante continuos comandos y enunciados de reconocimiento de voz, que evolucionaron para volverse 99 por ciento precisos.

Puesto que los requerimientos emergentes ya son una realidad, su organización debe considerar adoptar un modelo de proceso incremental.

Conforme los sistemas se vuelven más complejos, incluso un intento rudimentario por establecer requerimientos amplios está condenado al fracaso. Puede intentarse un enunciado de las metas globales, lograrse el delineado de los objetivos intermedios, pero los requerimientos estables, ¡ni por casualidad! Los requerimientos emergerán conforme todos los involucrados en la ingeniería y construcción de un sistema complejo aprendan más acerca de él, del entorno donde reside y de los usuarios con los que interactuará.

Esta realidad implica algunas tendencias en la ingeniería del software. Primero, deben diseñarse modelos de proceso para abarcar el cambio y adoptar los preceptos básicos de la filosofía ágil (capítulo 3). A continuación, deben usarse juiciosamente los métodos que producen modelos de ingeniería (por ejemplo, modelos de requerimientos y diseño) porque dichos modelos cambiarán repetidamente conforme se adquiera más conocimiento acerca del sistema. Finalmente, las herramientas que den apoyo tanto al proceso como a los métodos deben facilitar la adaptación y el cambio.

Pero existe otro aspecto de los requerimientos emergentes. La gran mayoría del software desarrollado a la fecha supone que la frontera entre el sistema basado en software y su entorno externo es estable. La frontera puede cambiar, pero lo hará en forma controlada, lo que permitirá al software adaptarse como parte de un ciclo de mantenimiento de software regular. Esta suposición comienza a cambiar. El software de mundo abierto (sección 31.2.2) demanda que los sistemas basados en computadora "se adapten y reaccionen a los cambios de manera dinámica, incluso si no se anticipan" [Bar06].

Por su naturaleza, los requerimientos emergentes conducen al cambio. ¿Cómo se controla la evolución, durante su ciclo de vida, de una aplicación o sistema que se usa ampliamente y qué efecto tiene esto sobre la forma en la que se diseña software?

Conforme crece el número de cambios, la probabilidad de efectos colaterales no intencionados también lo hace. Esto debe ser una causa de preocupación conforme los sistemas complejos con requerimientos emergentes se vuelven la norma. La comunidad de ingeniería del software debe desarrollar métodos que ayuden a los equipos de software a predecir el impacto de los cambios a través de todo un sistema, lo que, por tanto, mitiga los efectos colaterales no intencionales. En la actualidad, la capacidad para lograr esto está severamente limitada.

31.3.4 La mezcla de talento

A medida que los sistemas basados en software se vuelven más complejos, y conforme la comunicación y la colaboración entre equipos locales se vuelven un lugar común y si los requerimientos emergentes (con el flujo de cambios resultante) se vuelven la norma, la propia naturaleza de un equipo de ingeniería del software puede cambiar. Cada equipo de software debe devolver una variedad de talento creativo y habilidades técnicas a su parte de un sistema complejo, y el proceso global debe permitir que la salida de dichas islas de talento las fusione de manera efectiva.

Alexandra Weber Morales [Mor05] sugiere la mezcla de talento de un "equipo de ensueño de software". El *cerebro* es un arquitecto jefe que puede navegar entre las demandas de los participantes y mapearlas en un marco conceptual tecnológico que puede extenderse e implementarse. La *chica de datos* es una base de datos y gurú de estructuras de datos que "descompone filas y columnas con profunda comprensión de la lógica de predicados y teoría de conjuntos, como pertenecen al modelo relacional". El *bloqueador* es un líder técnico (gerente) que permite al equipo trabajar libre de interferencia de otros miembros del equipo mientras garantiza que ocurra la colaboración. El *hacker* es un programador consumado que está en casa con patrones y lenguajes y puede usarlos de manera efectiva. El *recopilador* "descubre hábilmente requerimientos de sistema con [...] comprensión antropológica" y los expresa con precisión y claridad.

31.3.5 Bloques constructores de software

Cita:

"La respuesta artística adecuada a la tecnología digital es abrazarla como una nueva ventana a todo lo que es eternamente humano, y usarla con pasión, sabiduría, valor y alegría."

Ralph Lombreglia

Quienes fomentan una filosofía de ingeniería del software enfatizan la necesidad de la reutilización de código fuente, clases orientadas a objeto, componentes, patrones y software COTS. Aunque la comunidad de ingeniería del software ha hecho progresos conforme intenta capturar el conocimiento pasado y reutilizar las soluciones probadas, un significativo porcentaje del software que se construye en la actualidad continúa construyéndose "desde cero". Parte de la razón de esto es un deseo continuo (de los participantes y profesionales de ingeniería del software) de "soluciones únicas".

En el mundo del hardware, los fabricantes de equipo original (FEO) de dispositivos digitales usan productos estándar específicos de aplicación (PEEA) producidos por proveedores de silicio de manera casi exclusiva. Estos "hardware mercantiles" proporcionan los bloques constructores necesarios para implementar todo, desde un teléfono celular hasta un reproductor HD-DVD. Cada vez más, los mismos FEO usan "software mercantil", bloques constructores de software diseñados específicamente para un dominio de aplicación único [por ejemplo, dispositivos VoIP]. Michael Ward [War07] comenta:

> Una ventaja del uso de componentes de software es que el FEO puede apalancar la funcionalidad proporcionada por el software sin tener que desarrollar experiencia doméstica en las funciones específicas o invertir tiempo de desarrollador en el esfuerzo por implementar y validar los componentes. Otras ventajas incluyen la habilidad para adquirir y desplegar sólo el conjunto de funcionalidades específicas que son necesarias para el sistema, así como la habilidad para integrar dichos componentes en una arquitectura ya existente.
>
> Sin embargo, el enfoque de componente de software tiene una desventaja porque existe un nivel dado de esfuerzo requerido para integrar los componentes individuales en el producto global. Este reto de integración puede complicarse aún más si los componentes se consiguen de varios proveedores, cada uno con su propia metodología de interfaz. Conforme se usen fuentes adicionales de componentes, el esfuerzo requerido para gestionar varios proveedores aumenta y hay un mayor riesgo de encontrar problemas relacionados con la interacción a través de los componentes de diferentes fuentes.

Además de las componentes empacadas como software mercantil, existe una creciente tendencia a adoptar *soluciones de plataforma de software* que "incorporan colecciones de funcionalidades relacionadas, por lo general proporcionados dentro de un marco conceptual de software integrado" [War07]. Una plataforma de software libera a un FEO del trabajo asociado con el desarrollo de la funcionalidad base y en su lugar permite que dedique el esfuerzo de software en aquellas características que diferencian su producto.

31.3.6 Cambio de percepciones de "valor"

Durante el último cuarto del siglo xx, la pregunta operativa que planteaban los empresarios cuando analizaban el software era: ¿por qué cuesta tanto? Esta pregunta rara vez se plantea en la actualidad y se sustituyó por: ¿por qué no podemos tenerlo (el software y/o el producto basado en software) más pronto?

Cuando se considera el software de computadora, la percepción moderna del valor cambia del valor empresarial (costo y rentabilidad) a valores de cliente que incluyen: rapidez de entrega, riqueza de funcionalidad y calidad global del producto.

31.3.7 Fuente abierta

¿Quién posee el software que usted o su organización usa? Cada vez más, la respuesta es "todo mundo". El movimiento "fuente abierta" se ha descrito de la forma siguiente [OSO08]: "Fuente abierta es un método de desarrollo para software que aprovecha el poder de la revisión de pares

distribuida y la transparencia de procesos. La promesa de la fuente abierta es mejor calidad, mayor confiabilidad, más flexibilidad, costo más bajo y terminar con el candado del proveedor depredador". El término *fuente abierta*, cuando se aplica a software de computadora, implica que los productos finales de ingeniería del software (modelos, código fuente, suites de pruebas) están abiertos al público y pueden revisarse y extenderse (con controles) por cualquiera que tenga interés y cuente con permiso.

Un "equipo" fuente abierta puede tener algunos miembros del "equipo de ensueño" de tiempo completo (sección 31.3.4), pero el número de personas que trabajan en el software se expande y contrae conforme el interés en la aplicación se fortalece o debilita. El poder del equipo de fuente abierta se deriva de la constante revisión de pares y de la refactorización de diseño/código que da como resultado un avance lento hacia una solución óptima.

Si el lector tiene más interés, Weber [Web05] proporciona una valiosa introducción y Feller *et al.* [Fel07] editaron una antología exhaustiva y objetiva que considera los beneficios y problemas asociados con la fuente abierta.

INFORMACIÓN

Tecnologías por llegar

Es probable que muchas tecnologías emergentes tengan un impacto significativo sobre los tipos de sistemas basados en computadora que evolucionan. Dichas tecnologías se agregan a los retos que enfrentan los ingenieros del software. Vale la pena anotar las siguientes tecnologías:

Computación de malla (*grid computing*): esta tecnología (disponible en la actualidad) crea una red que emplea los miles de millones de ciclos CPU que estén sin usar de cualquier máquina de la red y permite completar tareas de cómputo excesivamente complejas sin una supercomputadora dedicada a ello. Para un ejemplo de la vida real que abarca más de 4.5 millones de computadoras, visite http://setiathome.berkeley.edu/

Computación de mundo abierto: "Es ambiental, implícita, invisible y adaptativa. Consiste en que los dispositivos en red incrustados en el entorno proporcionan conectividad y servicios en todo momento sin obstrucciones" [McC05].

Microcomercio: una nueva rama del comercio electrónico que cobra cantidades muy pequeñas por acceder y/o comprar varios productos con propiedad intelectual. Apple iTunes es un ejemplo ampliamente usado.

Máquinas cognitivas: el "santo grial" en el campo de la robótica es el desarrollo de máquinas que estén conscientes de su entorno, que puedan "recoger pistas, responder a situaciones siempre cambiantes e interactuar con personas de manera natural" [PCM03]. Las máquinas cognitivas todavía están en las primeras etapas de desarrollo, pero su potencial es enorme.

Pantallas OLED: una OLED "utiliza una molécula de diseñador basada en carbono que emite luz cuando una corriente eléctrica pasa a través de ella. Coloque muchas moléculas juntas y obtendrá una pantalla superdelgada de sorprendente calidad; no se requiere iluminación trasera que extraiga energía" [PCM03]. El resultado: pantallas ultradelgadas que pueden enrollarse y doblarse, extenderse sobre superficies curvas o adaptarse de otro modo a un entorno específico.

RFID: la identificación por radiofrecuencia lleva a la computación de mundo abierto a una base industrial y a la industria de productos al consumidor. Todo, desde tubos de dentífrico hasta motores de automóviles, se identificará conforme se muevan a través de la cadena de suministro hasta su destino final.

Web 2.0: uno de varios servicios web que conducirá a una integración todavía mayor de la web tanto en comercio como en computación personal.

Para mayor análisis de las tecnologías por llegar, presentadas en una combinación única de video e impreso, visite el sitio web de Consumer Electronics Association en **www.ce.org/Press/CEA_Pubs/135.asp**

31.4 DIRECCIONES DE LA TECNOLOGÍA

La gente siempre parece creer que la ingeniería del software cambiará más rápidamente de lo que lo hace. Se introduce una nueva "promoción excesiva" de tecnología (un nuevo proceso, un método único o una herramienta excitante) y los expertos sugieren que "todo" cambiará. Pero la ingeniería del software tiene que ver menos con la tecnología y más con las personas y su habilidad para comunicar sus necesidades y para convertir dichas necesidades en realidad. Siempre que se involucran personas, ocurren cambios lentamente en irrupción repetida. Sólo

Cita:

"¿Pero para qué sirve?"

Comentario de un ingeniero de la División de Sistemas de Cómputo Avanzados de IBM, 1968, acerca del microchip

cuando se alcanza un "punto de no retorno" [Gla02], la tecnología cae en cascada a través de la comunidad de la ingeniería del software y ocurre verdaderamente un cambio muy amplio.

En esta sección se examinarán algunas tendencias sobre proceso, métodos y herramientas que es probable que tengan alguna influencia en la ingeniería del software durante la próxima década. ¿Ello conducirá a un punto de no retorno? Sólo es cuestión de esperar y ver.

31.4.1 Tendencias de proceso

Puede argumentarse que todas las tendencias empresariales, organizativas y culturales estudiadas en la sección 31.3 refuerzan la necesidad del proceso. ¿Pero los marcos conceptuales analizados en el capítulo 30 proporcionan un mapa de caminos hacia el futuro? ¿Los marcos conceptuales de proceso evolucionarán para encontrar un mejor equilibrio entre disciplina y creatividad? ¿El proceso de software se adaptará a las diferentes necesidades de los participantes que procuran el software, los que lo construyen y quienes lo usan? ¿Puede haber un medio para reducir el riesgo para los tres grupos al mismo tiempo?

Estas y muchas otras preguntas siguen abiertas. Sin embargo, algunas tendencias comienzan a surgir. Conradi y Fuggetta [Con02] sugieren seis "tesis acerca de cómo aumentar y aplicar mejor los marcos conceptuales MPS". Ellos comienzan su análisis con el siguiente enunciado:

> Una meta de quien procura el software es seleccionar al mejor contratista objetiva y racionalmente. El objetivo de una compañía de software es sobrevivir y crecer en un mercado competitivo. La de un usuario final es adquirir el producto de software que pueda resolver el problema correcto, en el momento correcto, a un precio aceptable. No podemos esperar el mismo enfoque MPS y el esfuerzo consecuente para alojar todos estos puntos de vista.

En los siguientes párrafos se adaptan las tesis propuestas por Conradi y Fuggetta [Con02] para sugerir posibles tendencias de proceso durante la próxima década.

¿Qué tendencias de proceso son probables durante la próxima década?

1. *Conforme evolucionan los marcos conceptuales MPS, enfatizarán "estrategias que se enfocan en la orientación de metas y en la innovación del producto"* [Con02]. En el mundo acelerado del desarrollo de software, las estrategias MPS a largo plazo rara vez sobreviven en un entorno empresarial dinámico. Demasiados cambios muy rápidamente. Esto significa que es posible que un mapa estable de caminos paso a paso deba sustituirse con un marco conceptual que enfatice las metas a corto plazo y que tenga una orientación de producto. Si los requerimientos para una nueva línea de productos basados en software surgirá durante una serie de liberaciones de producto incrementales (para entrega a usuarios finales mediante la web), la organización de software puede reconocer la necesidad de mejorar su capacidad para gestionar el cambio. El mejoramiento de proceso asociado con la gestión del cambio debe coordinarse con los ciclos de liberación del producto, de manera que mejorará la gestión del cambio mientras al mismo tiempo no lo perturbe.
2. *Puesto que los ingenieros del software tienen un buen sentido de dónde está débil el proceso, los cambios en el proceso por lo general deben impulsarse por sus necesidades y deben comenzar en forma ascendente.* Conradi y Fuggetta [Con02] sugieren que las futuras actividades MPS deben "usar una tarjeta de calificaciones simple y enfocada con la cual comenzar, no con una gran valoración". Al enfocar los esfuerzos MPS estrechamente y trabajar de manera ascendente, los profesionales comenzarán a ver cambios sustantivos más pronto, cambios que hacen una diferencia real en la forma como se realiza el trabajo en ingeniería del software.
3. *La tecnología de procesos de software automatizados (PSA) se alejará de la gestión de proceso global (amplio apoyo de todo el proceso de software) y se enfocará en aquellos aspectos del proceso de software que puedan beneficiarse mejor de la automatización.* Nadie está en

contra de las herramientas y de la automatización, pero, en muchas instancias, la PSA no cumple su promesa (vea la sección 31.2). Para ser más efectiva, debe enfocarse en actividades sombrilla (capítulo 2), los elementos más estables del proceso de software.

4. *Se colocará mayor énfasis en el rendimiento sobre la inversión de las actividades MPS.* En el capítulo 30 aprendió que el rendimiento sobre la inversión (RSI) puede definirse como:

$$\text{RSI} = \frac{\Sigma(\textit{beneficios}) - \Sigma(\textit{costos})}{\Sigma(\textit{costos})} \times 100\%$$

A la fecha, las organizaciones de software han luchado por delinear con claridad los "beneficios" en forma cuantitativa. Puede argumentarse [Con02] que "por tanto, necesitamos un modelo estandarizado con valor de mercado, como el que se utiliza en Cocomo II (vea el capítulo 26) para explicar las iniciativas de mejoramiento del software".

5. *Conforme pasa el tiempo, la comunidad del software puede llegar a entender que la experiencia en sociología y antropología puede tener tanto o más que ver con el éxito de MPS que otras disciplinas más técnicas.* Sobre todo, MPS cambia la cultura de la organización, y el cambio cultural involucra individuos y grupos de personas. Conradi y Fuggetta [Con02] anotan correctamente que "los desarrolladores de software son trabajadores del conocimiento. Tienden a responder negativamente a los dictados de esferas superiores acerca de cómo trabajar o cambiar los procesos". Es posible aprender mucho al examinar la sociología de los grupos para entender mejor las formas efectivas de introducir el cambio.

6. *Nuevos modos de aprendizaje pueden facilitar la transición hacia un proceso de software más efectivo.* En este contexto, "aprendizaje" implica aprendizaje de éxitos y errores. Una organización de software que recopila métricas (capítulos 23 y 25) se permite entender cómo los elementos de un proceso afectan la calidad del producto final.

31.4.2 El gran desafío

Existe una tendencia que es innegable: los sistemas basados en software sin duda se volverán más grandes y más complejos conforme pase el tiempo. Es la ingeniería de estos grandes sistemas complejos, sin importar la plataforma de entrega o el dominio de aplicación, la que impone el "gran desafío" [Bro06] a los ingenieros del software. Manfred Broy [Bro06] sugiere que los ingenieros del software pueden enfrentar el "intimidante reto de desarrollar sistemas de software complejos" al crear nuevos enfoques para entender los modelos de sistema y usar dichos modelos como base para la construcción de software de próxima generación de alta calidad.

Conforme la comunidad de ingeniería del software desarrolla nuevos enfoques impulsados por modelo (que se estudian más adelante en esta sección) para la representación de los requerimientos del sistema y su diseño, pueden abordarse las siguientes características [Bro06]:

? ¿Qué características del sistema deben considerar analistas y diseñadores para futuras aplicaciones?

- *Multifuncionalidad:* conforme los dispositivos digitales evolucionan hacia su segunda y tercera generaciones, comienzan a entregar un rico conjunto de, en ocasiones, funciones no relacionadas. El teléfono celular, alguna vez considerado un dispositivo de comunicación, ahora se utiliza para tomar fotografías, conservar un calendario, navegar por un diario y como reproductor de música. Si las interfaces de mundo abierto llegan a trascender, estos dispositivos móviles se usarán para mucho más durante los próximos años. Como anota Broy [Bro06], "los ingenieros deben describir el contexto detallado en el que se entregarán las funciones y, más importante, deben identificar las interacciones potencialmente dañinas entre las diferentes características del sistema".
- *Reactividad y oportunidad:* los dispositivos digitales interactúan cada vez más con el mundo real y deben reaccionar a estímulos externos en forma oportuna. Deben poner

interfaz con una amplia serie de sensores y responder en un marco temporal que sea adecuado a la tarea que se busca realizar. Deben desarrollarse nuevos métodos que 1) ayuden a los ingenieros de software a predecir la temporalidad de varias características reactivas y 2) implementen dichas características de manera que las haga menos dependientes de la máquina y más portátiles.

- *Nuevos modos de interacción con el usuario:* el teclado y el ratón funcionan bien en un entorno PC, pero las tendencias de mundo abierto para software señalan que deben modelarse e implementarse nuevos modos de interacción. Ya sea que dichos nuevos enfoques usen interfaces de toque múltiple, reconocimiento de voz o interfaces de mente directa,[9] las nuevas generaciones de software para dispositivos digitales deben modelar estas nuevas interfaces humano-computadora.
- *Arquitecturas complejas:* un automóvil lujoso tiene más de 2 000 funciones controladas mediante software que residen dentro de una compleja arquitectura de hardware que incluye múltiples CPU, una sofisticada estructura de bus, actuadores, sensores, una interfaz humana cada vez más sofisticada, y muchos componentes con clasificación de seguridad. Sistemas incluso más complejos están en el horizonte inmediato, lo que presenta retos significativos para los diseñadores de software.
- *Sistemas heterogéneos distribuidos:* los componentes en tiempo real para cualquier moderno sistema incrustado pueden conectarse mediante un bus interno, una red inalámbrica o a través de internet (o todo junto).
- *Crucialidad:* el software se ha convertido en el componente pivote en virtualmente todos los sistemas cruciales para los negocios y en la mayoría de los sistemas importantes para la seguridad. Sin embargo, la comunidad de ingeniería del software apenas comienza a aplicar incluso los principios más básicos de la seguridad de software.
- *Variabilidad de mantenimiento:* la vida del software dentro de un dispositivo digital rara vez dura más allá de 3 a 5 años, pero los complejos sistemas de aviónica dentro de una aeronave tienen una vida útil de al menos 20 años. El software de los automóviles falla en alguna parte intermedia. ¿Esto tendrá algún impacto sobre el diseño?

Broy [Bro06] argumenta que éstas y otras características del software pueden gestionarse solamente si la comunidad de ingeniería del software desarrolla una filosofía de ingeniería del software distribuida de manera más efectiva y colaborativa, mejores enfoques de ingeniería de requerimientos, un enfoque más robusto al desarrollo impulsado por modelo y mejores herramientas de software. En las secciones que siguen se explorará brevemente cada una de estas áreas.

31.4.3 Desarrollo colaborativo

PUNTO CLAVE
La colaboración involucra la diseminación oportuna de la información y un proceso efectivo para comunicarse y tomar decisiones.

Parece casi demasiado obvio de afirmar, pero se hará de cualquier forma: *la ingeniería del software es una tecnología de información.* Desde el inicio de cualquier proyecto de software, cada participante debe compartir información: acerca de las metas y objetivos empresariales básicos, de los requerimientos de sistema específicos, de conflictos de diseño arquitectónico, de casi todo aspecto del software que se va a construir.

En la actualidad, los ingenieros de software colaboran a través de zonas horarias y fronteras internacionales, y cada uno de ellos debe compartir información. Lo mismo es cierto para los proyectos de fuente abierta en los que trabajan cientos o miles de desarrolladores de software

9 Un breve estudio acerca de las interfaces de mente directa puede encontrarse en **http://en.wikipedia.org/wiki/Brain-computer_interface**, y un ejemplo comercial se describe en **http://au.gamespot.com/news/6166959.html**

para construir una aplicación de fuente abierta. De nuevo, la información debe diseminarse de modo que pueda ocurrir la colaboración abierta.

El reto durante la próxima década es desarrollar métodos y herramientas que faciliten dicha colaboración. Hoy día, continúa la lucha por facilitar la colaboración. Eugene Kim [Kim04] comenta:

> Considere una tarea de colaboración básica: compartir documentos. Algunas aplicaciones (tanto comerciales como de fuente abierta) afirman resolver el problema de compartir documentos y, sin embargo, el método predominante para compartir archivos es enviarlos por correo electrónico de ida y vuelta. Éste es el equivalente computacional de la *sneakernet*. Si las herramientas que tienen el propósito de resolver este problema son buenas, ¿por qué no se utilizan?
>
> En otras áreas básicas se ven problemas similares. Puedo ir a una reunión en cualquier parte del mundo con un trozo de papel en la mano, y puedo estar seguro de que la gente querrá leerlo, marcarlo, darle vueltas y archivarlo. No puedo decir lo mismo de los documentos electrónicos. No puedo anotar en una página web ni usar el mismo sistema de llenado para el correo electrónico y para los documentos de Word, al menos no en una forma que garantice la interoperabilidad con las aplicaciones de mi propia máquina o de otras. ¿Por qué no?
>
> ...Con la finalidad de tener un impacto real en el espacio colaborador, las herramientas no sólo deben ser buenas, deben ser interoperables.

Pero la falta de herramientas colaborativas amplias sólo es una parte del reto que enfrentan quienes deben desarrollar software de manera colaborativa.

En la actualidad, un porcentaje significativo[10] de los proyectos IT se subcontratan a nivel internacional, y el número crecerá sustancialmente durante la siguiente década. No es de sorprender que Bhat *et al.* [Bha06] aseveren que la ingeniería de requerimientos es la actividad crucial en un proyecto subcontratado. Los autores identifican algunos factores de éxito que conducen a esfuerzos de colaboración exitosos:

- *Metas compartidas:* las metas del proyecto deben enunciarse con claridad, y todos los participantes deben comprenderlas y estar de acuerdo con su intención.
- *Cultura compartida:* las diferencias culturales deben definirse con claridad; debe desarrollarse un enfoque educativo (que ayudará a mitigar dichas diferencias) y un enfoque de comunicación (que facilitará la transferencia de conocimiento).
- *Proceso compartido:* en algunas formas, el proceso funciona como el esqueleto de un proyecto colaborativo, lo que proporciona un medio uniforme para valorar el progreso y la dirección, y para introducir un "lenguaje" técnico común para todos los miembros del equipo.
- *Responsabilidad compartida:* todo miembro del equipo debe reconocer la importancia de la ingeniería de requerimientos y trabajar para ofrecer la mejor definición posible del sistema.

Cuando se combinan, dichos factores de éxito conducen a la "confianza": un equipo global que puede apoyarse en grupos dispares para lograr el trabajo que se les asignó.

31.4.4 Ingeniería de requerimientos

En los capítulos del 5 al 7 se presentaron las acciones básicas de la ingeniería de requerimientos: adquisición, elaboración, negociación, especificación y validación. El éxito o fracaso de dichas

10 Actualmente, alrededor de 20 por ciento de un presupuesto IT típico de las compañías grandes se dedica a subcontratación (outsourcing) y el porcentaje crece cada año. (Fuente: www.logicacmg.com/page/400002849.)

acciones tiene una influencia muy fuerte sobre el éxito o el fracaso de todo el proceso de ingeniería del software. Y, sin embargo, la ingeniería de requerimientos (IR) se compara con "intentar poner una abrazadera de sujeción alrededor de una gelatina" [Gon04]. Como se señala en muchos lugares a lo largo de este libro, los requerimientos de software tienen la tendencia a seguir cambiando, y con la llegada de los sistemas de mundo abierto, los requerimientos emergentes (y el cambio casi continuo) pueden volverse la norma.

En la actualidad, la mayoría de los enfoques de ingeniería de requerimientos "informales" comienzan con la creación de escenarios de usuario (por ejemplo, casos de uso). Los enfoques más formales crean uno o más modelos de requerimientos y el uso de los mismos como base para el diseño. Los métodos formales permiten que un ingeniero de software represente los requerimientos, usando una notación matemática verificable. Todo puede funcionar razonablemente bien cuando los requerimientos son estables, pero no se resuelve fácilmente el problema de los requerimientos dinámicos o emergentes.

Existen algunas direcciones distintas en la investigación en ingeniería de requerimientos, incluidos el procesamiento de lenguaje natural a partir de descripciones textuales traducidas en representaciones más estructuradas (por ejemplo, clases de análisis), mayor apoyo sobre bases de datos para estructurar y comprender los requerimientos de software, el uso de patrones IR para describir problemas y soluciones usuales cuando se realizan las tareas de ingeniería de requerimientos y la ingeniería de requerimientos orientada a metas. Sin embargo, industrialmente, las acciones de IR siguen siendo más o menos informales y sorprendentemente básicas. Para mejorar la forma en la que se definen los requerimientos, la comunidad de ingeniería del software probablemente implementará tres subprocesos distintos conforme se lleve a cabo la IR [Gli07]: 1) adquisición de conocimiento mejorado y compartición de conocimiento que permita comprensión más completa de las restricciones del dominio de aplicación y las necesidades de los participantes, 2) mayor énfasis en la iteración conforme se definen los requerimientos y 3) herramientas de comunicación y coordinación más efectivas que permitan a todos los participantes colaborar de manera efectiva.

PUNTO CLAVE

"Los nuevos subprocesos IR incluyen: 1) adquisición de conocimiento mejorado, 2) incluso más iteración y 3) herramientas de comunicación y coordinación más efectivas".

Los subprocesos IR señalados en el párrafo anterior solamente triunfarán si se integran de manera adecuada en un enfoque evolutivo a la ingeniería del software. Conforme la resolución de problemas basada en patrones y las soluciones basadas en componentes comiencen a dominar muchos dominios de aplicación, la IR debe acomodar el deseo de agilidad (rápida entrega incremental) y los requerimientos emergentes inherentes que resulten. La naturaleza concurrente de muchos modelos de proceso en ingeniería del software significa que la IR se integrará con actividades de diseño y construcción. En consecuencia, la noción de una "especificación de software" estática comienza a desaparecer, para ser sustituida por "requerimientos impulsados por valores" [Som05] derivada conforme los participantes responden a las características y funciones entregadas en los incrementos de software anteriores.

31.4.5 Desarrollo de software impulsado por modelo

Los ingenieros de software se aferran a la abstracción virtualmente en cada paso del proceso de ingeniería del software. Conforme comienza el diseño, las abstracciones arquitectónicas y en el nivel de componente se representan y valoran. Luego deben traducirse en una representación en lenguaje de programación que transforme el diseño (un nivel de abstracción relativamente alto) en un sistema operativo con un entorno de computación específico (un nivel de abstracción bajo). El *desarrollo de software impulsado por modelo*[11] acopla lenguajes de modelado específicos de dominio con motores y generadores de transformación, de manera que facilita la representación de la abstracción en niveles altos y luego los transforma en niveles más bajos [Sch06].

PUNTO CLAVE

El enfoque impulsado por modelo enfrenta un desafío permanente para todos los desarrolladores de software: cómo representar el software a un nivel mas alto de abstracción que en código.

11 También se usa el término *ingeniería impulsada por modelo* (IIM).

Los *lenguajes de modelado específicos de dominio* (LMED) representan "la estructura, comportamiento y requerimientos de la aplicación dentro de dominios de aplicación particulares" y se describen con metamodelos que "definen las relaciones entre conceptos en el domino y especifican con precisión la semántica y restricciones clave asociadas con dichos conceptos de dominio" [Sch06]. La diferencia principal entre un LMED y un lenguaje de modelado de propósito general como UML (apéndice 1) es que el primero se sintoniza con los conceptos de diseño inherentes en el dominio de aplicación y, por tanto, puede representar relaciones y restricciones entre elementos de diseño en forma eficiente.

31.4.6 Diseño posmoderno

En un interesante artículo acerca del diseño de software en la "era posmoderna", Philippe Kruchten [Kru05] hace la siguiente observación:

> La ciencia de la computación no ha logrado la gran narrativa que explique todo, el *gran cuadro*: no hemos encontrado las leyes fundamentales del software que jugarían el papel que las leyes fundamentales de la física juegan en otras disciplinas de la ingeniería. Todavía vivimos con el resabio amargo de la explosión de burbujas de internet y el día del juicio final Y2K. De modo que, en esta era posmoderna, donde parece que todo importa un poco, aunque realmente no importa mucho, ¿cuáles son las siguientes direcciones para el diseño de software?

Parte de cualquier intento por comprender las tendencias en el diseño del software es establecer fronteras al diseño. ¿Dónde se detiene la ingeniería de requerimientos y comienza el diseño? ¿Dónde se detiene el diseño y comienza la generación de código? Las respuestas a estas preguntas no son sencillas como podrían parecer al principio. Aun cuando el modelo de requerimientos deba enfocarse en "qué", no en "cómo", todo analista hace un poco de diseño y casi todos los diseñadores hacen un poco de análisis. De igual modo, conforme el diseño de componentes de software se acerca un poco más al detalle algorítmico, un diseñador comienza a representar el componente en un nivel de abstracción que está cerca del código.

El diseño posmoderno continuará enfatizando la importancia de la arquitectura del software (capítulo 9). Un diseñador debe enunciar un conflicto arquitectónico, tomar una decisión que aborde el conflicto y luego definir con claridad las suposiciones, restricciones e implicaciones que la decisión impone sobre el software como un todo. Pero, ¿existe un marco conceptual donde los conflictos pueden describirse y la arquitectura puede definirse? El desarrollo de software orientado a aspecto (capítulo 2) o el desarrollo de software impulsado por modelo (sección 31.4.4) pueden convertirse en importantes enfoques de diseño en los años por venir, pero todavía es muy pronto para decirlo. Puede ser que la innovación en el desarrollo basado en componentes (capítulo 10) pueda conducir a una filosofía de diseño que enfatice el ensamblado de los componentes existentes. Si el pasado es prólogo, es enormemente probable que surjan muchos "nuevos" métodos de diseño, pero pocos remontarán la curva de la promoción excesiva (figura 31.2) mucho más allá del "valle de la desilusión".

31.4.7 Desarrollo impulsado por pruebas

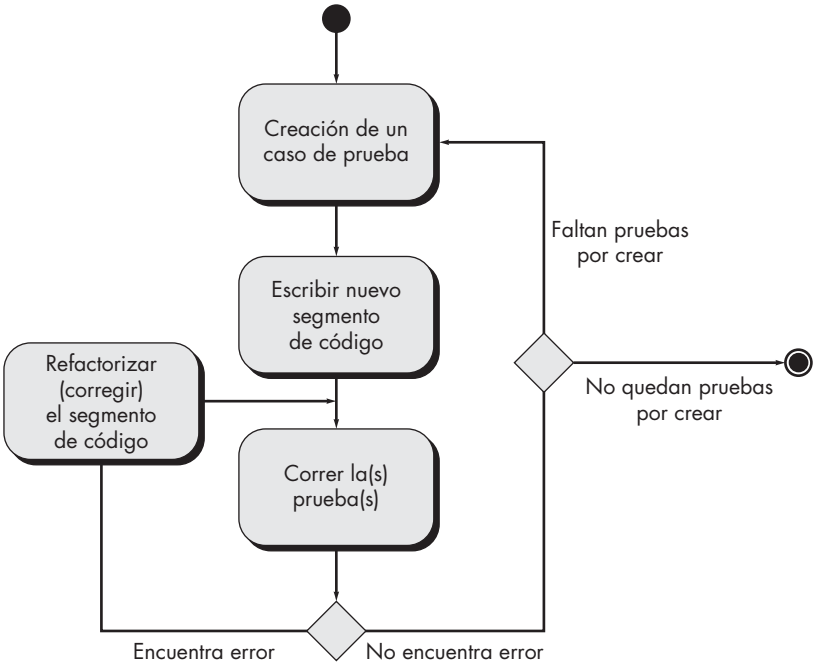
Los requerimientos impulsan el diseño y éste establece un cimiento para construcción. Esta simple realidad en ingeniería del software funciona razonablemente bien y es esencial conforme se crea una arquitectura de software. Sin embargo, un cambio sutil puede proporcionar beneficios significativos cuando se consideran el diseño en el nivel de componentes y la construcción.

PUNTO CLAVE

"DIP es una tendencia que enfatiza el diseño de casos de prueba antes de la creación de código fuente".

En el *desarrollo impulsado por pruebas* (DIP), los requerimientos para un componente de software funcionan como la base para la creación de una serie de casos de prueba que ejerciten la interfaz y que intenten encontrar errores en las estructuras de datos y en la funcionalidad que

FIGURA 31.3
Flujo de proceso de desarrollo impulsado por pruebas



entrega el componente. El DIP realmente no es una nueva tecnología, sino más bien una tendencia que enfatiza el diseño de casos de prueba *antes* de la creación de código fuente.[12]

El proceso DIP sigue el simple flujo procedural que se ilustra en la figura 31.3. Antes de crear el primer pequeño segmento de código, un ingeniero de software crea una prueba para ejercitar el código (intentar que fracase el código). Entonces el código se escribe para satisfacer la prueba. Si la pasa, se crea una nueva prueba para el siguiente segmento de código que se va a desarrollar. El proceso continúa hasta que el componente está completamente codificado y todas las pruebas se ejecutan sin error. Sin embargo, si alguna prueba triunfa al encontrar un error, el código existente se refactoriza (corrige) y todas las pruebas creadas para dicho punto se vuelven a ejecutar. Este flujo iterativo continúa hasta que no hay pruebas pendientes de crear, lo que implica que los componentes satisfacen todos los requerimientos definidos para él.

Durante el DIP, el código se desarrolla en incrementos muy pequeños (una subfunción a la vez) y no se escribe código hasta que exista una prueba que lo ejercite. Debe observar que cada iteración resulta en una o más pruebas nuevas que se agregan a una suite de pruebas de regresión que corren con cada cambio. Esto se hace para garantizar que el nuevo código no generó efectos colaterales que causen errores en el código anterior.

En DIP, las pruebas impulsan el diseño de componentes detallados y el código fuente resultante. Los resultados de dichas pruebas causan modificaciones inmediatas al diseño de componentes (vía el código) y, más importante, el componente resultante (cuando se completa) se verificó en forma independiente. Si tiene más interés en el DIP, consulte [Bec04b] o [Ast04].

31.5 TENDENCIAS RELACIONADAS CON HERRAMIENTAS

Cada año se introducen cientos de herramientas de ingeniería del software de grado industrial. La mayoría las aportan los proveedores de herramientas, quienes afirman que su herramienta

12 Recuerde que la programación extrema (capítulo 3) enfatiza este enfoque como parte de su modelo de proceso ágil.

mejorará la administración del proyecto, el análisis de requerimientos, el modelado de diseño, la generación de código, las pruebas, la gestión del cambio o cualquiera de las muchas actividades, acciones y tareas de la ingeniería del software que se estudian a lo largo de este libro. Otras herramientas se desarrollaron como ofrecimientos de fuente abierta. La mayoría de las herramientas de fuente abierta se enfocan en las actividades de "programación" con un énfasis específico en la actividad de construcción (particularmente la generación de código). Incluso otras herramientas se derivan de esfuerzos de investigación en universidades y laboratorios gubernamentales. Aunque tienen atractivo en aplicaciones muy limitadas, la mayoría no está lista para aplicación industrial amplia.

En el nivel industrial, los paquetes de herramientas más amplios forman *entornos de ingeniería del software* (EIS)[13] que integran una colección de herramientas individuales en torno de una base de datos central (repositorio). Cuando se considera como un todo, un EIS integra información a través del proceso de software y auxilia en la colaboración que se requiere para sistemas basados en computadora muy grandes y complejos. Pero los entornos actuales no son fácilmente extensibles (es difícil integrar una herramienta COTS que no sea parte del paquete) y tienden a ser de propósito general (es decir, no son específicas de dominio de aplicación). También existe una considerable demora temporal entre la introducción de las nuevas soluciones tecnológicas (por ejemplo, desarrollo de software impulsado por modelo) y la disponibilidad de EIS viables que den soporte a la nueva tecnología.

Las futuras tendencias en las herramientas de software seguirán dos rutas distintas: una *ruta enfocada a lo humano* que responda a algunas de las "tendencias blandas" estudiadas en la sección 31.3 y una ruta centrada en tecnología que aborde las nuevas tecnologías (sección 31.4) conforme se introduzcan y adopten. En las siguientes secciones se examinará brevemente cada ruta.

31.5.1 Herramientas que responden a tendencias blandas

Las tendencias blandas analizadas en la sección 31.3 (la necesidad de administrar la complejidad, acomodar requerimientos emergentes, establecer modelos de proceso que aborden el cambio, coordinar equipos globales con una mezcla de talento cambiante, entre otros) sugiere una nueva era en la que las herramientas que apoyen la colaboración de los participantes se volverán tan importantes como las herramientas que apoyan la tecnología. ¿Pero qué tipo de conjunto de herramientas soportan dichas tendencias blandas?

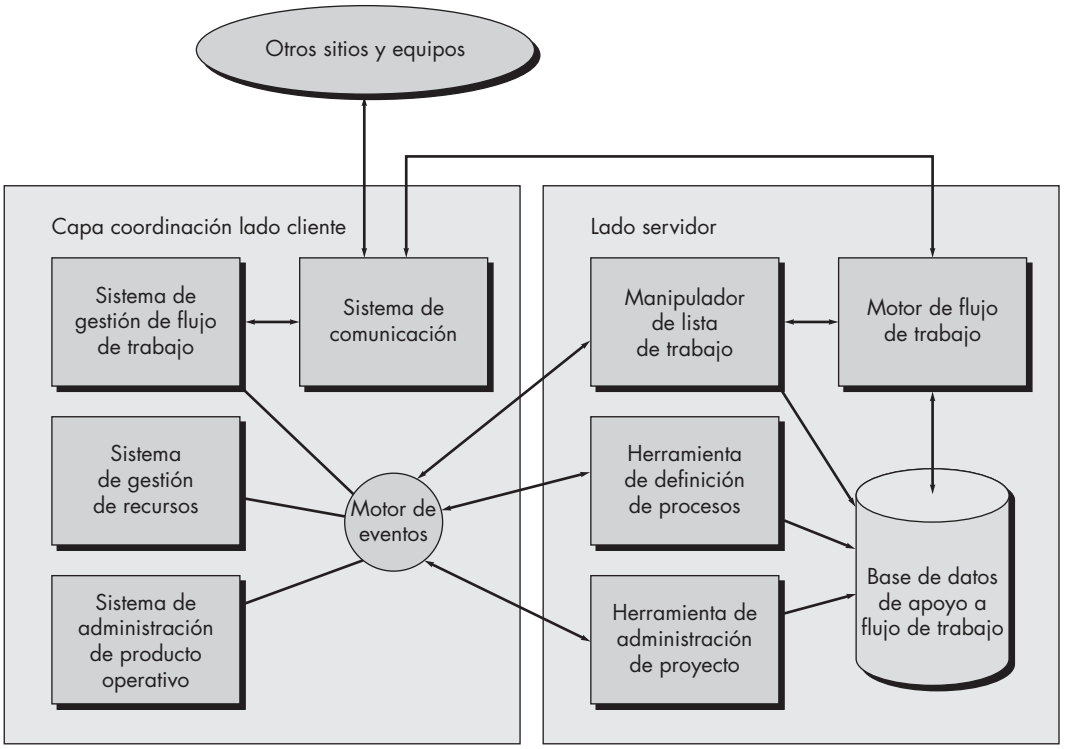
Un ejemplo de investigación en esta área es GENESIS, un entorno generalizado de fuente abierta diseñado para soportar trabajo colaborativo de ingeniería del software [Ave04]. El entorno GENESIS puede o no volverse de uso amplio, pero sus elementos básicos son representativos de la dirección de EIS colaboradores que evolucionarán para dar apoyo a las tendencia blandas anotadas en este capítulo.

Un EIS colaborativo "soporta la cooperación y la comunicación entre ingenieros de software que pertenecen a equipos de desarrollo distribuidos, involucrados en modelar, controlar y medir el desarrollo del software y los procesos de mantenimiento. Más aún, incluye una función de administración de artefacto que almacena y gestiona artefactos de software (productos operativos) producidos por diferentes equipos en el curso de su "trabajo" [Bol02].

La figura 31.4 ilustra una arquitectura para un EIS colaborativo. La arquitectura, basada en el entorno GENESIS [Ave04], se construye con subsistemas que se integran dentro de un cliente web común y que se complementa mediante componentes basados en servidor que proporcionan apoyo a todos los clientes. Cada organización de desarrollo tiene sus propios subsistemas en el lado cliente que se comunican con otros clientes. En la figura 31.4, un *subsistema de gestión*

13 También se usa el término *entorno de desarrollo integrado* (EDI).

FIGURA 31.4
Arquitectura EIS colaborativa
Fuente: Adaptado de [Ave04].



de recursos administra la asignación de recursos humanos a diferentes proyectos o subproyectos; un *sistema de administración de producto de trabajo* es "responsable de la creación, modificación, borrado", indexado, búsqueda y almacenamiento de todos los productos operativos de la ingeniería del software [Ave04]; un *subsistema de gestión de flujo de trabajo* coordina la definición, ejemplificación e implementación de las actividades, acciones y tareas del proceso de software; un *motor de evento* "recopila eventos" que ocurren durante el proceso de software (por ejemplo, una revisión exitosa de un producto operativo, la conclusión de pruebas de unidad de un componente) y lo notifica a otros; un *sistema de comunicación* soporta comunicación sincrónica y asíncrona entre los equipos distribuidos.

En el lado servidor, cuatro componentes comparten una base de datos de apoyo al flujo de trabajo. Los componentes implementan las siguientes funciones:

- *Definición de proceso:* un conjunto de herramientas que permiten a un equipo definir nuevas actividades, acciones o tareas de proceso y que define las reglas que gobiernan la manera en la que interactúan dichos elementos unos con otros y los productos operativos que producen.
- *Administración de proyecto:* un conjunto de herramientas que permiten al equipo construir un plan de proyecto y coordinar el plan con otros equipos o proyectos.
- *Motor de flujo de trabajo:* "interactúa con el motor de eventos para propagar eventos que son relevantes para la ejecución de procesos cooperativos que se ejecutan en otros sitios" [Ave04].
- *Manipulador de lista de trabajo:* interactúa con la base de datos del lado servidor para brindar a un ingeniero de software información acerca de la tarea que actualmente se desarrolla o cualquier tarea futura que se derive del trabajo que actualmente se realiza.

Aunque la arquitectura de un EIS colaborativo puede variar considerablemente del que se estudió en esta sección, los elementos funcionales básicos (sistemas y componentes de administración) parecerán lograr el nivel de coordinación que se requiere para un proyecto de ingeniería del software distribuida.

31.5.2 Herramientas que abordan tendencias tecnológicas

La agilidad en la ingeniería del software (capítulo 3) se logra cuando los participantes trabajan como un equipo. Por tanto, la tendencia hacia los EIS colaborativos (sección 31.5.1) brindará beneficios aun cuando el software se desarrolle de manera local. Pero, ¿qué hay acerca de las herramientas tecnológicas que complementan el sistema y los componentes que fortalecen una mejor colaboración?

Una de las tendencias dominantes en las herramientas tecnológicas es la creación de un conjunto de herramientas que da apoyo al desarrollo impulsado por modelo (sección 31.4.4) con énfasis en el diseño impulsado por arquitectura. Oren Novotny [Nov04] sugiere que el modelo, más que el código fuente, se convierte en el foco central de la ingeniería del software:

> En UML se crean modelos independientes de plataforma y luego se experimentan varios niveles de transformación para eventualmente devanarse como código fuente para una plataforma específica. Entonces, es lógico que el modelo, no el archivo, deba convertirse en la nueva unidad de salida. Un modelo tiene muchas visiones diferentes en diferentes niveles de abstracción. En el más alto, los componentes independientes de plataforma pueden especificarse en el análisis; en el más bajo, existe una implementación específica a plataforma que se reduce a un conjunto de clases en código.

Novotny argumenta que una nueva generación de herramientas funcionará en conjunción con un repositorio para: crear modelos en todos los niveles necesarios de abstracción, establecer relaciones entre varios modelos, traducir los modelos de un nivel de abstracción a otro (por ejemplo, traducir un modelo de diseño a código fuente), gestionar cambios y versiones, y coordinar las acciones de control y aseguramiento de la calidad en contraste con los modelos de software.

Además de completar los entornos de ingeniería del software, las herramientas de solución puntual que abordan todo, desde recopilación de requerimientos hasta refactorización de diseño/código y pruebas, continuarán evolucionando y se volverán más funcionalmente capaces. En algunas instancias, las herramientas de modelado y pruebas en un dominio de aplicación específico proporcionarán beneficios aumentados cuando se comparen con sus equivalentes genéricos.

31.6 Resumen

Las tendencias que tienen efecto sobre la tecnología de ingeniería del software con frecuencia provienen de las áreas de negocios, organizacional, de mercado y cultural. Dichas "tendencias blandas" pueden guiar la dirección de la investigación y la tecnología que se deriva como consecuencia de la investigación.

Conforme se introduce nueva tecnología, se avanza a través de un ciclo de vida que no siempre conduce a una adopción extensa, aun cuando las expectativas originales sean altas. El grado en el que cualquier tecnología de ingeniería del software gana adopción extensa está ligado a su habilidad para abordar los problemas impuestos por las tendencias blandas y duras.

Las tendencias blandas (la creciente necesidad de conectividad y colaboración, proyectos globales, transferencia de conocimiento, el impacto de las economías emergentes y la influencia de la cultura humana en sí) conducen a un conjunto de retos que abarcan la complejidad administrativa y los requerimientos emergentes, hasta hacer malabares con una mezcla de talentos siempre cambiante entre equipos de software dispersos geográficamente.

Las tendencias duras (el ritmo siempre acelerado del cambio tecnológico) fluyen desde las tendencias blandas y afectan la estructura del software y el ámbito del proceso y la forma en la que se caracteriza un marco conceptual de proceso. El desarrollo colaborativo, nuevas formas de ingeniería de requerimientos, desarrollo basado en modelo e impulsado por pruebas y el diseño posmoderno cambiarán el panorama de los métodos. Los entornos de herramientas responderán a una necesidad creciente de comunicación y colaboración y al mismo tiempo integrarán soluciones puntuales específicas de dominio que pueden cambiar la naturaleza de las actuales tareas de la ingeniería del software.

Problemas y puntos por evaluar

31.1. Obtenga una copia del libro *The Tipping Point*, de Malcolm Gladwell (disponible mediante Google Book Search) y analice cómo se aplican sus teorías a la adopción de nuevas tecnologías en ingeniería del software.

31.2. ¿Por qué el software de mundo abierto presenta un desafío a los enfoques convencionales de la ingeniería del software?

31.3. Revise el *ciclo de promoción excesiva para tecnologías emergentes* del Gartner Group. Seleccione un producto tecnológico muy conocido y presente una breve historia que ilustre cómo viajó a lo largo de la curva. Seleccione otro producto tecnológico muy conocido que no siguió la ruta sugerida por la curva.

31.4. ¿Qué es una "tendencia blanda"?

31.5. Usted se enfrenta con un problema extremadamente complejo que requerirá una solución extensa. ¿Cómo abordaría la complejidad y crearía una respuesta?

31.6. ¿Cuáles son los "requerimientos emergentes" y por qué presentan un reto para los ingenieros de software?

31.7. Seleccione un esfuerzo de desarrollo de fuente abierta (distinto a Linux) y presente una breve historia de su evolución y éxito relativo.

31.8. Describa cómo cree que cambiará el proceso de software durante la próxima década.

31.9. Usted está ubicado en Los Ángeles y trabaja en un equipo de ingeniería de software global. Usted y sus colegas en Londres, Mumbai, Hong Kong y Sidney deben editar una especificación de requerimientos de 245 páginas para un sistema grande. La primera edición debe completarse en tres días. Describa el conjunto ideal de herramientas en línea que le permitirían colaborar de manera efectiva.

31.10. Describa el desarrollo de software impulsado por modelo con sus propias palabras. Haga lo mismo para el desarrollo impulsado por pruebas.

Lecturas y fuentes de información adicionales

Los libros que estudian el camino por venir para el software y la computación abarcan una gran variedad de temas técnicos, científicos, económicos, políticos y sociales. Kurweil (*The Singularity Is Near*, Penguin Books, 2005) presenta una mirada persuasiva de un mundo que cambiará en formas realmente profundas hacia mediados de este siglo. Sterling (*Tomorrow Now*, Random House, 2002) recuerda que el progreso real rara vez es ordenado y eficiente. Teich (*Technology and the Future*, Wadworth, 2002) presenta ensayos concienzudos acerca del impacto social de la tecnología y cómo la cultura cambiante da forma a la tecnología. Naisbitt, Phillips y Naisbitt (*High Tech/High Touch*, Nicholas Brealey, 2001) observan que muchas personas se han "intoxicado" con la alta tecnología y que "la gran ironía de la era de la alta tecnología es que nos estamos volviendo esclavos de los dispositivos que se suponía que nos darían libertad". Zey (*The Future Factor*, McGraw-Hill, 2000) estudia cinco fuerzas que darán forma al destino humano durante este siglo. El de Negroponte (*Being Digital*, Alfred A. Knopf, 1995) fue un *best seller* a mediados de los años de 1990 y continúa ofreciendo una visión comprensiva de la computación y de su impacto global.

Conforme el software se vuelve parte del tejido de virtualmente cada faceta de la vida, la "cibernética" evolucionó como un importante tema de estudio. Los libros de Spinello (*Cyberethics: Morality and Law in Cyberspace*, Jones & Bartlett Publishers, 2002), Halbert y Ingulli (*Cyberethics,* South-Western College Publish-

ers, 2001), y Baird *et al*. (*Cyberethics: Social and Moral Issues in the Computer Age*, Prometheus Books, 2000) consideran el tema con detalle. El gobierno estadounidense publicó un voluminoso reporte en CD-ROM (*21st Century Guide to Cybercrime*, Progressive Management, 2003) que considera todos los aspectos del crimen computacional, los temas de la propiedad intelectual y el National Infraestrutura Protection Center (NIPC, Centro Nacional de Protección a la Infraestructura).

En internet, está disponible una gran variedad de fuentes de información acerca de las direcciones futuras en las tecnologías relacionadas con software y la ingeniería del software. Una lista actualizada de referencias en la World Wide Web que son relevantes para las futuras tendencias en ingeniería del software puede encontrarse en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

CAPÍTULO

32

COMENTARIOS FINALES

CONCEPTOS CLAVE

conocimiento 719
ética 721
futuro 720
nueva visita al software . . . 718
personal 718
tecnología de información . . 719

En los 31 capítulos que precedieron al actual, se exploró un proceso de ingeniería del software que abarca procedimientos administrativos y métodos técnicos, conceptos y principios básicos, técnicas especializadas, actividades orientadas al personal y tareas que son sensibles a la automatización, notación en papel y lápiz, y herramientas de software. Se argumentó que medición, disciplina y un enfoque centrado sobre todo en la agilidad y la calidad darán como resultado software que satisfaga las necesidades del cliente, que sea confiable, mantenible y mejor. Sin embargo, nunca se prometió que la ingeniería del software fuese una panacea.

Conforme se avanza en la segunda década de este nuevo siglo, las tecnologías de software y de sistemas siguen siendo un desafío para todo profesional de software y para toda compañía que construya sistemas basados en computadora. Aunque las siguientes palabras se escribieron con una perspectiva del siglo XX, Max Hopper [Hop90] describe con precisión el estado actual de las cosas:

> Puesto que los cambios en tecnología de la información se vuelven tan rápidos e implacables, y ya que las consecuencias de retrasarse son tan irreversibles, las compañías dominarán la tecnología o morirán... Piense en ello como en una caminadora de tecnología. Las compañías tendrán que correr cada vez más rápido sólo para mantener el paso.

Los cambios en la tecnología de ingeniería del software son de hecho "rápidos e implacables", pero, al mismo tiempo, el progreso con frecuencia es muy lento. Para cuando se toma una decisión a fin de adoptar un nuevo proceso, método o herramienta; realizar la capacitación necesaria para comprender su aplicación e introducir la tecnología en la cultura de desarrollo de software, llega algo más nuevo (e incluso mejor) y el proceso comienza de nuevo.

Una cosa que el autor ha aprendido a través de sus años en este campo es que los profesionales de la ingeniería del software están "conscientes de la moda". El camino por venir estará lleno de cascarones de excitantes nuevas tecnologías (la última moda) que realmente nunca lograron serlo (a pesar de la promoción excesiva). Ese camino se formará con tecnologías más modestas que de alguna manera modificarán la orientación y el ámbito de la vía pública. En el capítulo 31 se estudiaron algunas de ellas.

En este capítulo de conclusiones se tomará una visión más amplia y se considerará dónde se ha estado y a dónde se va desde una perspectiva más filosófica.

UNA MIRADA RÁPIDA

¿Qué es? Conforme se aproxima el final de un viaje relativamente largo a través de la ingeniería del software, es momento de poner las cosas en perspectiva y hacer algunos comentarios finales.

¿Quién lo hace? Autores como el actual. Cuando se llega al final de un largo y desafiante libro es bueno cerrar las cosas en forma significativa.

¿Por qué es importante? Siempre vale la pena recordar dónde se estuvo y considerar hacia dónde se va.

¿Cuáles son los pasos? Se considerará dónde se estuvo y se abordarán algunos temas centrales y orientaciones para el futuro.

¿Cuál es el producto final? Un análisis que le ayudará a entender el gran cuadro.

¿Cómo me aseguro de que lo hice bien? Es difícil de lograr en tiempo real. Sólo después de algunos años podrá decirse si los conceptos, principios, métodos y técnicas de ingeniería del software que se estudiaron en este libro le ayudaron a convertirse en un mejor ingeniero del software.

32.1 La importancia del software-revisión

La importancia del software de computadora puede establecerse de muchas maneras. En el capítulo 1 se caracterizó como un diferenciador. La función que entrega el software diferencia a productos, sistemas y servicios, y ofrece ventaja competitiva en el mercado. Pero el software es más que un diferenciador. Cuando se toma como un todo, los productos operativos de la ingeniería del software generan el artículo más importante que cualquier individuo, negocio o gobierno puede adquirir: información.

En el capítulo 31 se analizó brevemente la computación de mundo abierto (inteligencia ambiental, aplicaciones atentas al contexto y computación predominante/ubicua), una dirección que cambiará de modo fundamental la percepción de las computadoras, las cosas que uno hace con ellas (y las que ellas hacen para uno) y la percepción de la información como guía, producto y necesidad. También se señaló que el software requerido para dar soporte a la computación de mundo abierto presentará nuevos desafíos dramáticos para los ingenieros del software. Pero, mucho más importante, la penetración venidera del software de computadora presentará retos más dramáticos para la sociedad como un todo. Siempre que una tecnología tiene amplio impacto (un impacto que puede salvar vidas o ponerlas en peligro, construir empresas o destruirlas, informar o malinformar a los líderes del gobierno), debe "manejarse con cuidado".

32.2 Las personas y la forma en la que construyen sistemas

El software requerido para sistemas de alta tecnología se vuelve más complejo con cada año que transcurre y el tamaño de los programas resultantes aumenta de manera proporcional. El rápido crecimiento en tamaño del programa "promedio" presentaría algunos problemas si no fuese por un simple hecho: conforme aumenta el tamaño del programa, el número de personas que deben trabajar en el programa también debe aumentar.

Cita:

"El shock del futuro [es] la aplastante tensión y desorientación que inducimos en los individuos al sujetarlos a demasiado cambio en un periodo muy corto."

Alvin Toffler

La experiencia indica que, conforme aumenta el número de personas de un equipo de proyecto de software, la productividad global del grupo puede disminuir. Una forma de resolver este problema es crear algunos equipos de ingeniería del software, lo que, por tanto, divide al personal en grupos de trabajo menores. Sin embargo, conforme crece el número de equipos de ingeniería del software, la comunicación entre ellos se vuelve tan difícil y consumidora de tiempo como la comunicación entre individuos. Peor aún, la comunicación (entre individuos o equipos) tiende a ser ineficiente, es decir, se emplea mucho tiempo transfiriendo muy poco contenido de información y también, con mucha frecuencia, la información importante "cae en las grietas".

Si la comunidad de ingeniería del software ha de lidiar de manera efectiva con el dilema de la comunicación, el camino por delante para los ingenieros del software debe incluir cambios radicales en la forma en la que los individuos y los equipos se comunican entre ellos. En el capítulo 31 se estudiaron los entornos colaboradores que pueden proporcionar mejorías dramáticas en las formas en las que se comunican los equipos.

En última instancia, la comunicación es la transferencia de conocimiento, y la adquisición (y transferencia) de conocimiento representa cambiar profundamente. Conforme los motores de búsqueda se vuelven cada vez más sofisticados y las aplicaciones Web 2.0 ofrecen mejor sinergia, la biblioteca más grande del mundo de artículos y reportes, tutoriales, comentarios y referencias de investigación se vuelve más accesible y utilizable.

Si la historia pasada es un indicio, es justo decir que las personas no cambiarán. Sin embargo, las formas en las que se comunican, el entorno en el que trabajan, la forma en la que adquieren conocimiento, los métodos y herramientas que usan, la disciplina que aplican y, en consecuencia, la cultura global para el desarrollo del software cambiará significativa e, incluso, profundamente.

32.3 NUEVOS MODOS PARA REPRESENTAR LA INFORMACIÓN

Cita:

"La mejor preparación para un buen trabajo mañana es hacer un buen trabajo hoy."

Elbert Hubbard

A través de la historia de la computación, ha ocurrido una transición sutil en la terminología que se usa para describir el trabajo de desarrollo del software que realiza la comunidad empresarial. Hace 40 años, el término *procesamiento de datos* era la frase operativa que describía el uso de las computadoras en un contexto empresarial. En la actualidad, el procesamiento de datos originó otra frase, *tecnología de la información*, que implica lo mismo pero con un cambio sutil de enfoque. El énfasis ya no es simplemente procesar grandes cantidades de datos, sino, más bien, extraer información significativa de estos datos. Obviamente, ésta siempre fue la intención, pero el cambio en la terminología refleja una modificación mucho más importante en la filosofía administrativa.

Cuando hoy se estudian aplicaciones de software, las palabras *datos*, *información* y *contenido* ocurren repetidamente. La palabra *conocimiento* se encuentra en algunas aplicaciones de inteligencia artificial, pero su uso es relativamente raro. Virtualmente, nadie discute *sabiduría* en el contexto de aplicaciones de software.

Los datos son información bruta: colecciones de hechos que deben procesarse para ser significativas. La información se entrega al asociar hechos dentro de un contexto determinado. El conocimiento asocia información obtenida en un contexto con otra información obtenida en un contexto diferente. Finalmente, la sabiduría ocurre cuando se derivan principios generalizados a partir de conocimiento dispar. Cada una de estas cuatro visiones de "información" se representa de manera esquemática en la figura 32.1.

Cita:

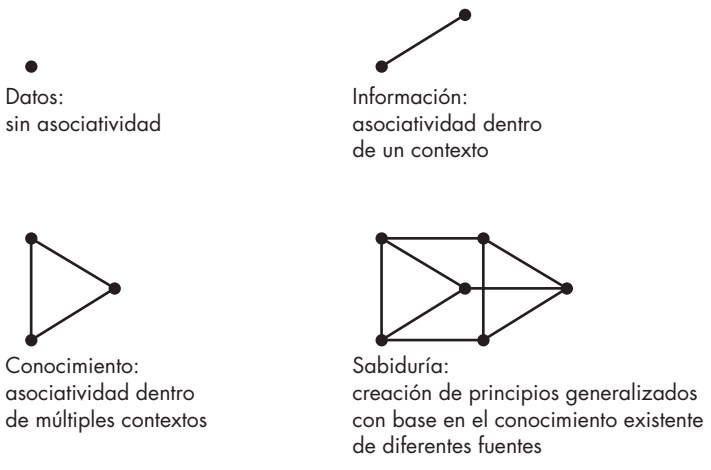
"La sabiduría es el poder que nos permite usar el conocimiento para beneficio de nosotros mismos y de los demás."

Thomas J. Watson

A la fecha, la gran mayoría del software se construyó para procesar datos o información. Los ingenieros de software ahora están igualmente preocupados con los sistemas que procesan conocimiento.[1] El conocimiento es bidimensional. La información recopilada acerca de varios temas relacionados y no relacionados se conecta para formar un cuerpo de hechos que se llama *conocimiento*. La clave es la habilidad para asociar información de una variedad de fuentes diferentes, que pueden no tener alguna conexión obvia, y para combinarla de manera que ofrezca algún beneficio diferente.[2]

Para ilustrar la progresión que conduce desde datos hasta conocimiento, considere los datos censales que indican que la tasa de natalidad en 1996 en Estados Unidos fue de 4.9 millones.

FIGURA 32.1
Un espectro de "información"



1 El rápido crecimiento de tecnologías de minado y de data warehouses refleja esta tendencia creciente.

2 La semántica web (Web 2.0) permite la creación de "purés" que pueden proporcionar un mecanismo sencillo para lograr esto.

Este número representa un valor de datos. Al relacionar este trozo de datos con las tasas de natalidad de los 40 años anteriores, puede inferir una pieza de información útil: los *baby boomers* de los años cincuenta y principios de los sesenta del siglo pasado, que hoy envejecen, hicieron un último esfuerzo para tener hijos antes del final de sus años de crianza infantil. Además, los de la generación X comenzaron sus años de crianza infantil. Los datos censales pueden conectarse entonces a otras piezas de información aparentemente no relacionadas. Por ejemplo, el número actual de profesores de enseñanza básica que se retirarán durante la próxima década, el número de estudiantes universitarios que se gradúan en educación primaria y secundaria, la presión sobre los políticos para mantener bajos los impuestos y, por tanto, limitar los aumentos de sueldo para los profesores. Todas estas piezas de información pueden combinarse para formular una representación de conocimiento: habrá una presión significativa sobre el sistema educativo de Estados Unidos a principios del siglo XXI y continuará durante algunas décadas. Al usar este conocimiento puede surgir una oportunidad de negocios. Puede haber una significativa oportunidad para desarrollar nuevos modos de aprendizaje que sean más efectivos y menos costosos que los enfoques actuales.

El camino futuro para el software conduce a sistemas que procesan el conocimiento. Durante más de 50 años se han procesado datos usando computadoras, y durante más de tres décadas se ha extraído información. Uno de los retos más significativos que enfrenta la comunidad de ingeniería del software es construir sistemas que den el siguiente paso en el espectro: sistemas que extraigan conocimiento a partir de datos e información, de manera práctica y benéfica.

32.4 La vista larga

En la sección 32.3 se sugirió que el camino por venir conduce a sistemas que "procesen conocimiento". Pero el futuro de la computación en general y de los sistemas basados en software en particular puede conducir a eventos que sean considerablemente más profundos.

En un libro fascinante que debe leer toda persona involucrada en las tecnologías de la computación, Ray Kurzweil [Kur05] sugiere que se ha llegado a un momento en el que "el ritmo del cambio tecnológico será tan rápido, y su impacto tan profundo, que la vida humana será transformada de manera irreversible". Kurzweil[3] plantea un argumento de peso: la humanidad actualmente está en la "rodilla" de una curva de crecimiento exponencial que conducirá a un enorme incremento en la capacidad de computación durante las siguientes dos décadas. Cuando se acople con avances equivalentes en nanotecnología, genética y robótica, será posible aproximarse a un momento, a mediados de este siglo, donde la distinción entre humanos (como se les conoce hoy día) y máquinas comience a empañarse: un momento en el que la evolución humana se acelere en formas que son tanto atemorizantes (para algunos) como espectaculares (para otros).

Kurzweil argumenta que, en algún momento en la década de 2030, la capacidad de computación y el software requerido serán suficientes para modelar cada aspecto del cerebro humano: todo, desde conexiones físicas, procesos analógicos y recubrimientos químicos. Cuando esto ocurra, los seres humanos habrán logrado "una elevada IA" (inteligencia artificial) y, como consecuencia, máquinas que realmente piensen (dentro del uso convencional actual de la palabra). Pero habrá una diferencia fundamental. Los procesos del cerebro humano son excesivamente complejos y sólo se conectan de manera holgada con las fuentes de información externas. También son computacionalmente lentos, incluso en comparación con la tecnología de compu-

3 Es importante notar que Kurzweil no es un escritor de ciencia ficción ordinario, o un futurólogo sin cartera. Es un tecnólogo serio que (según Wikipedia) "fue pionero en los campos de reconocimiento óptico de caracteres (OCR), síntesis de texto a habla, tecnología de reconocimiento de voz e instrumentos de teclado electrónico".

tación actual. Cuando ocurra la simulación completa del cerebro humano, el "pensamiento" ocurrirá con rapideces miles de veces mayores que su contraparte humana, con íntimas conexiones a un mar de información (piense en la web de la actualidad como un ejemplo primitivo). El resultado es... bueno... tan fantástico que es mejor dejar a Kurzweil describirlo.

Es importante notar que nadie cree que el futuro que Kurzweil describe sea algo bueno. En un ensayo ahora famoso, titulado "The Future Doesn't Need Us" (El futuro no nos necesita), Bill Joy [Joy00], uno de los fundadores de Sun Microsystems, argumenta que "robótica, ingeniería genética y nanotecnología amenazan con hacer a los humanos una especie en peligro de extinción". Sus argumentos, que predicen una distopia tecnológica, representan un contrapunto al futuro utópico predicho por Kurzweil. Ambos deben considerarse seriamente como ingenieros de software que juegan uno de los papeles principales en la definición de la visión larga de la especie humana.

32.5 La responsabilidad del ingeniero de software

La ingeniería del software ha evolucionado en una profesión mundial respetada. Como profesionales, los ingenieros de software deben acatar un código de ética que guíe el trabajo que realizan y los productos que elaboran. Una fuerza de trabajo conjunta ACM/IEEE-CS produjo un *Código de ética y práctica profesional de la ingeniería del software* (versión 5.1). El código [ACM98] afirma:

> Los ingenieros de software deben comprometerse con hacer del análisis, la especificación, el diseño, el desarrollo, la prueba y el mantenimiento del software una profesión benéfica y respetada. En concordancia con su compromiso con la salud, la seguridad y el bienestar del público, los ingenieros de software deben adherirse a los siguientes ocho principios:
>
> 1. PÚBLICO: Los ingenieros del software deben actuar consistentemente con el interés del público.
> 2. CLIENTE Y EMPLEADOR: Los ingenieros de software deben actuar en función del mejor interés de sus clientes y empleadores, coincidente con el interés del público.
> 3. PRODUCTO: Los ingenieros de software deben garantizar que sus productos y modificaciones relacionadas satisfagan los más altos estándares profesionales posibles.
> 4. JUICIO: Los ingenieros de software deben mantener integridad e independencia en su juicio profesional.
> 5. ADMINISTRACIÓN: Los administradores y líderes de ingeniería del software deben suscribirse y promover un enfoque ético acerca de la administración del desarrollo y del mantenimiento del software.
> 6. PROFESIÓN: Los ingenieros de software deben promover la integridad y reputación de la profesión, consistente con el interés del público.
> 7. COLEGAS: Los ingenieros de software deben ser justos con sus colegas y darles apoyo.
> 8. UNO MISMO: Los ingenieros de software deben aprender toda la práctica de su profesión y deben promover un enfoque ético acerca de ella.

WebRef

Un análisis completo del código de ética de ACM/IEEE puede encontrarse en **seeri.etsu.edu/Codes/default.shtm**

Aunque cada uno de estos ocho principios es igualmente importante, aparece un tema sensible: un ingeniero de software necesita trabajar en función del interés del público. En el nivel personal, un ingeniero de software debe conducirse de acuerdo con las siguientes reglas:

- Nunca robar datos para ganancia personal.
- Nunca distribuir o vender información con derechos de autor obtenida como parte de su trabajo en un proyecto de software.
- Nunca destruir o modificar maliciosamente los programas, archivos o datos de otra persona.

- Nunca violar la privacidad de un individuo, grupo u organización.
- Nunca *hackear* un sistema por deporte o beneficio.
- Nunca crear o propagar un virus o gusano.
- Nunca usar tecnología de computación para facilitar la discriminación o el hostigamiento.

Durante la década pasada, ciertos miembros de la industria del software acudieron a las autoridades en busca de legislación protectora que [SEE03]: 1) permita a las compañías liberar software sin revelar defectos conocidos, 2) exentar a los desarrolladores de responsabilidad por cualquier daño que resulte de dichos defectos conocidos, 3) restringir a otros en revelar defectos sin permiso del desarrollador original, 4) permitir la incorporación de software de "autoayuda" dentro de un producto que pueda deshabilitar (mediante comando remoto) la operación del producto y 5) exentar a los desarrolladores de software con "autoayuda" de daños en caso de que el software sea deshabilitado por una tercera persona.

Como toda legislación, el debate frecuentemente se centra en temas que son políticos, no tecnológicos. Sin embargo, muchas personas (incluido el autor) creen que la legislación protectora, si se emite de manera inadecuada, entra en conflicto con el código de ética de la ingeniería del software al exentar de manera indirecta a los ingenieros de software de su responsabilidad de producir software de alta calidad.

32.6 Un comentario final

Hace 30 años comenzó el trabajo en la primera edición de este libro. El autor todavía se recuerda sentado en su escritorio como joven profesor, escribiendo el manuscrito para un libro acerca de una materia por la que pocas personas se preocupaban e incluso aún menos entendían realmente. Recuerda las cartas de rechazo de los editores, quienes argumentaban (cortés, pero firmemente) que nunca habría un mercado para un libro acerca de "ingeniería del software". Por fortuna, McGraw-Hill decidió darle una oportunidad,[4] y el resto, como dicen, es historia.

Durante los pasados 30 años, este libro cambió dramáticamente: en visión, en tamaño, en estilo y en contenido. Como la ingeniería del software, creció y (con fortuna) maduró con los años.

Un enfoque de ingeniería centrado en el desarrollo del software de computadora ahora es sabiduría convencional. Aunque el debate continúa acerca del "paradigma correcto", la importancia de la agilidad, el grado de automatización y los métodos más efectivos, los principios subyacentes de la ingeniería del software, ahora se aceptan en toda la industria. ¿Por qué, entonces, se ha visto su adopción amplia sólo recientemente?

La respuesta, acaso, se encuentra en la dificultad de la transición tecnológica y el cambio cultural que la acompaña. Aun cuando la mayoría de las personas aprecian la necesidad de una disciplina de ingeniería para el software, se lucha contra la inercia de la práctica pasada y se enfrentan nuevos dominios de aplicación (y de los desarrolladores que los trabajan) que parecen listos a repetir los errores del pasado. Para facilitar la transición se necesitan muchas cosas: un proceso de software ágil, adaptable y sensible; métodos más efectivos; herramientas más poderosas; mejor aceptación por parte de los profesionales y apoyo de los administradores; y no pequeñas dosis de educación.

Acaso el lector no esté de acuerdo con todos los enfoques descritos en este libro. Algunas de las técnicas y opiniones son controvertidas; otras deben afinarse para trabajar bien en diferentes

4 En realidad, el crédito debe ir para Peter Freeman y Eric Munson, quienes convencieron a McGraw-Hill de que valía la pena probar. Más de un millón de copias después, es justo decir que tomaron una buena decisión.

entornos de desarrollo de software. Sin embargo, el autor desea sinceramente que *Ingeniería del software. Un enfoque práctico* haya delineado el problema que se enfrenta, demostrado la fuerza de los conceptos de la ingeniería del software y ofrecido un marco conceptual de métodos y herramientas.

Conforme se avanza aún más en el siglo XXI, el software sigue siendo el producto y la industria más importantes en la escena mundial. Su impacto e importancia han recorrido un largo camino. Y, sin embargo, una nueva generación de desarrolladores de software debe satisfacer muchos de los mismos desafíos que enfrentaron las generaciones anteriores. Con la esperanza de que las personas que enfrenten el reto, ingenieros del software, tendrán la sabiduría para desarrollar sistemas que mejoren la condición humana.

APÉNDICE 1

INTRODUCCIÓN A UML[1]

CONCEPTOS CLAVE

canales 736
diagramas de actividad 735
diagramas de clase 725
diagramas de comunicación 734
diagramas de estado 737
diagramas de implementación 729
diagramas de secuencia 732
diagramas de uso de caso . . 730
dependencia 728
estereotipo 726
generalización 727
lenguaje de restricción de objeto 740
marcos de interacción 733
multiplicidad 728

El *Lenguaje de Modelado Unificado* (UML) es "un lenguaje estándar para escribir diseños de software. El UML puede usarse para visualizar, especificar, construir y documentar los artefactos de un sistema de software intensivo" [Boo05]. En otras palabras, tal como los arquitectos de edificios crean planos para que los use una compañía constructora, los arquitectos de software crean diagramas de UML para ayudar a los desarrolladores de software a construir el software. Si usted entiende el vocabulario del UML (los elementos pictóricos de los diagramas y su significado) puede comprender y especificar con mucha más facilidad un sistema, y explicar su diseño a otros.

Grady Booch, Jim Rumbaugh e Ivar Jacobson desarrollaron el UML a mediados de los años noventa del siglo pasado con mucha realimentación de la comunidad de desarrollo de software. El UML fusionó algunas notaciones de modelado que competían entre sí y que se usaban en la industria del software en la época. En 1997, UML 1.0 se envió al Object Management Group, un consorcio sin fines de lucro involucrado en especificaciones de mantenimiento para su empleo en la industria de la computación. El UML 1.0 se revisó y dio como resultado la adopción del UML 1.1 ese mismo año. El estándar actual es UML 2.0 y ahora es un estándar ISO. Puesto que este estándar es tan nuevo, muchas antiguas referencias, como [Gam95], no usan notación de UML.

UML 2.0 proporciona 13 diferentes diagramas para su uso en modelado de software. En este apéndice se analizarán solamente diagramas de *clase, implementación, caso de uso, secuencia, comunicación, actividad* y *estado*, diagramas que se usan en esta edición de *Ingeniería del software. Un enfoque práctico*.

Debe observar que existen muchas características opcionales en diagramas de UML. El UML ofrece dichas opciones (en ocasiones complejas) de modo que pueda expresar todos los aspectos importantes de un sistema. Al mismo tiempo, tiene la flexibilidad para suprimir aquellas partes del diagrama que no son relevantes para el aspecto que se va a modelar, con la finalidad de evitar confundir el diagrama con detalles irrelevantes. Por tanto, la omisión de una característica particular no significa que ésta se encuentre ausente; puede significar que la característica se suprimió. En este apéndice, *no* se presenta la cobertura exhaustiva de todas las características de los diagramas de UML. El apéndice se enfocará en las opciones estándar, en especial en aquellas que se usaron en este libro.

DIAGRAMAS DE CLASE

Para modelar clases, incluidos sus atributos, operaciones, relaciones y asociaciones con otras clases,[2] el UML proporciona un *diagrama de clase*, que aporta una visión estática o de estructura de un sistema, sin mostrar la naturaleza dinámica de las comunicaciones entre los objetos de las clases.

---

1 Este apéndice fue una aportación de Dale Skrien y se adaptó de su libro *An Introduction to Object-Oriented Design and Design Patterns in Java* (McGraw-Hill, 2008). Todo el contenido se usa con permiso.

2 Si el lector no está familiarizado con los conceptos orientados a objeto, en el apéndice 2 se presenta una breve introducción.

Los elementos principales de un diagrama de clase son cajas, que son los íconos utilizados para representar clases e interfaces. Cada caja se divide en partes horizontales. La parte superior contiene el nombre de la clase. La sección media menciona sus atributos. Un *atributo* es algo que un objeto de dicha clase conoce o puede proporcionar todo el tiempo. Por lo general, los atributos se implementan como campos de la clase, pero no necesitan serlo. Podrían ser valores que la clase puede calcular a partir de sus variables o valores instancia y que puede obtener de otros objetos de los cuales está compuesto. Por ejemplo, un objeto puede conocer siempre la hora actual y regresarla siempre que se le solicite. Por tanto, sería adecuado mencionar la hora actual como un atributo de dicha clase de objetos. Sin embargo, el objeto muy probablemente no tendría dicha hora almacenada en una de sus variables instancia, porque necesitaría actualizar de manera continua ese campo. En vez de ello, el objeto probablemente calcularía la hora actual (por ejemplo, a través de consulta con objetos de otras clases) en el momento en el que se le solicite la hora. La tercera sección del diagrama de clase contiene las operaciones o comportamientos de la clase. Una *operación* es lo que pueden hacer los objetos de la clase. Por lo general, se implementa como un método de la clase.

La figura A1.1 presenta un ejemplo simple de una clase **Thoroughbred** (pura sangre) que modela caballos de pura sangre. Muestra tres atributos: **mother** (madre), **father** (padre) y **birthyear** (año de nacimiento). El diagrama también muestra tres operaciones: *getCurrentAge()* (obtener edad actual), *getFather()* (obtener padre) y *getMother()* (obtener madre); puede haber otros atributos y operaciones suprimidos que no se muestren en el diagrama.

Cada atributo puede tener un nombre, un tipo y un nivel de visibilidad. El tipo y la visibilidad son opcionales. El tipo sigue al nombre y se separa de él mediante dos puntos. La visibilidad se indica mediante un –, #, ~ o + precedente, que indica, respectivamente, visibilidad *privada*, *protegida*, *paquete* o *pública*. En la figura A1.1, todos los atributos tienen visibilidad privada, como se indica mediante el signo menos que los antecede (–). También es posible especificar que un atributo es estático o de clase, subrayándolo. Cada operación puede desplegarse con un nivel de visibilidad, parámetros con nombres y tipos, y un tipo de retorno.

Una clase abstracta o un método abstracto se indica con el uso de cursivas en el nombre del diagrama de clase. Vea, por ejemplo, la clase **Horse** (caballo) en la figura A1.2. Una interfaz se indica con la frase "<<interface>>" (llamada *estereotipo*) arriba del nombre. Vea la interfaz **OwnedObject** (objeto posesión) en la figura A1.2. Una interfaz también puede representarse gráficamente mediante un círculo hueco.

Vale la pena mencionar que el ícono que representa una clase puede tener otras partes opcionales. Por ejemplo, puede usarse una cuarta sección en el fondo de la caja de clase para mencionar las responsabilidades de la clase. Esta sección es particularmente útil cuando se realiza la transición de tarjetas CRC (capítulo 6) a diagramas de clase, donde las responsabilidades mencionadas en las tarjetas CRC pueden agregarse a esta cuarta sección en la caja de clase en el diagrama UML antes de crear los atributos y operaciones que llevan a cabo dichas responsabilidades. Esta cuarta sección no se muestra en ninguna de las figuras de este apéndice.

FIGURA A1.1
Diagrama de clase para una clase Thoroughbred

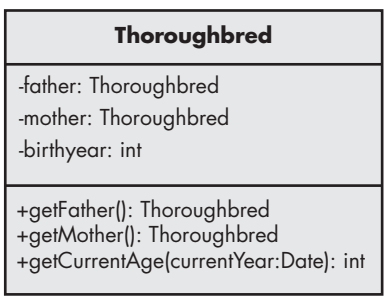
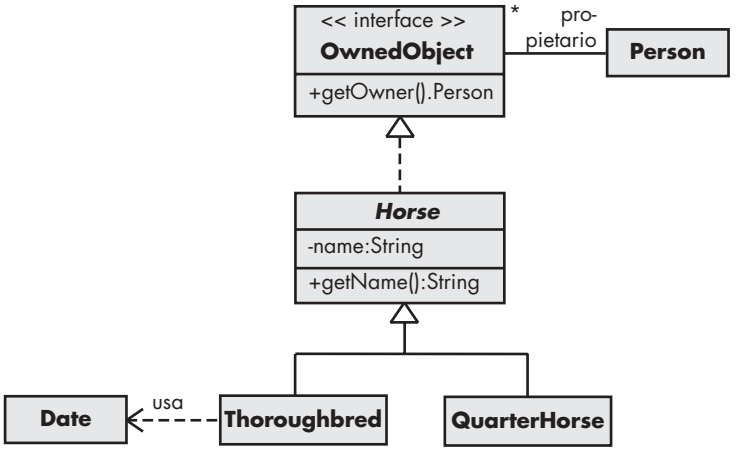


FIGURA A1.2
Diagrama de clase concerniente a caballos



Los diagramas de clase también pueden mostrar relaciones entre clases. Una clase que sea una subclase de otra clase se conecta con ella mediante una flecha con una línea sólida y con una punta triangular hueca. La flecha apunta de la subclase a la superclase. En UML, tal relación se llama *generalización*. Por ejemplo, en la figura A1.2, las clases **Thoroughbred** y **QuarterHorse** (caballo cuarto de milla) se muestran como subclases de la clase abstracta **Horse**. Una flecha con una línea punteada indica implementación de una interfaz. En UML, tal relación se llama *realización*. Por ejemplo, en la figura A1.2, la clase **Horse** implementa o realiza la interfaz **OwnedObject**.

Una *asociación* entre dos clases significa que existe una relación estructural entre ellas. Las asociaciones se representan mediante líneas sólidas. Una asociación tiene muchas partes opcionales. Puede etiquetarse, así como cada una de sus terminaciones, para indicar el papel de cada clase en la asociación. Por ejemplo, en la figura A1.2, existe una asociación entre **OwnedObject** y **Person**, en la que **Person** juega el papel de owner (propietario). Las flechas en cualquiera o en ambos lados de una línea de asociación indican navegabilidad. Además, cada extremo de la línea de asociación puede tener un valor de multiplicidad desplegado. Navegabilidad y multiplicidad se explican con más detalle más adelante, en esta sección. Una asociación también puede conectar una clase consigo misma, mediante un bucle. Tal asociación indica la conexión de un objeto de la clase con otros objetos de la misma clase.

Una asociación con una flecha en un extremo indica navegabilidad en un sentido. La flecha significa que, desde una clase, es posible acceder con facilidad a la segunda clase asociada hacia la que apunta la asociación; sin embargo, desde la segunda clase, no necesariamente puede accederse con facilidad a la primera clase. Otra forma de pensar en esto es que la primera clase está al tanto de la segunda, pero el segundo objeto de clase no necesariamente está directamente al tanto de la primera clase. Una asociación sin flechas por lo general indica una asociación de dos vías, que es lo que se pretende en la figura A1.2; pero también simplemente podría significar que la navegabilidad no es importante y, por tanto, que queda fuera.

Debe observarse que un atributo de una clase es muy parecido a una asociación de la clase con el tipo de clase del atributo. Es decir, para indicar que una clase tiene una propiedad llamada "name" (nombre) de tipo String, podría desplegarse dicha propiedad como atributo, como en la clase **Horse** en la figura A1.2. De manera alternativa, podría crearse una asociación de una vía desde la clase **Horse** hasta la clase **String** donde el papel de la clase String es "name". El enfoque de atributo es mejor para tipos de datos primitivos, mientras que el enfoque de asociación con frecuencia es mejor si la clase propietaria juega un papel principal en el diseño, en cuyo caso es valioso tener una caja de clase para dicho tipo.

Una relación de *dependencia* representa otra conexión entre clases y se indica mediante una línea punteada (con flechas opcionales en los extremos y con etiquetas opcionales). Una clase depende de otra si los cambios en la segunda clase pueden requerir cambios en la primera. Una asociación de una clase con otra automáticamente indica una dependencia. No se necesitan líneas punteadas entre clases si ya existe una asociación entre ellas. Sin embargo, para una relación transitoria (es decir, una clase que no mantiene alguna conexión de largo plazo con otra, sino que usa dicha clase de manera ocasional), debe dibujarse una línea punteada desde la primera clase hasta la segunda. Por ejemplo, en la figura A1.2, la clase **Thoroughbred** usa la clase **Date** (fecha) siempre que se invoca su método *getCurrentAge()*; por eso la dependencia se etiqueta "usa".

La *multiplicidad* de un extremo de una asociación significa el número de objetos de dicha clase que se asocia con la otra clase. Una multiplicidad se especifica mediante un entero no negativo o mediante un rango de enteros. Una multiplicidad especificada por "0..1" significa que existen 0 o 1 objetos en dicho extremo de la asociación. Por ejemplo, cada persona en el mundo tiene un número de seguridad social o no lo tiene (especialmente si no son ciudadanos estadounidenses); por tanto, una multiplicidad de 0..1 podría usarse en una asociación entre una clase **Person** y una clase **SocialSecurityNumber** (número de seguridad social) en un diagrama de clase. Una multiplicidad que se especifica como "1...*" significa uno o más, y una multiplicidad especificada como "0...*" o sólo "*" significa cero o más. Un * se usa como la multiplicidad en el extremo **OwnedObject** de la asociación con la clase **Person** en la figura A1.2, porque una **Person** puede poseer cero o más objetos.

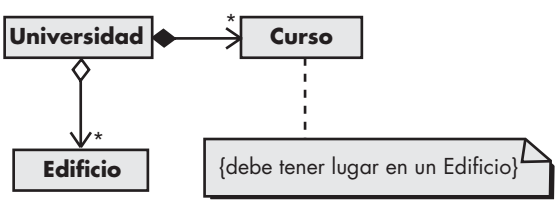
Si un extremo de una asociación tiene multiplicidad mayor que 1, entonces los objetos de la clase a la que se refiere en dicho extremo de la asociación probablemente se almacenan en una colección, como un conjunto o lista ordenada. También podría incluirse dicha clase colección en sí misma en el diagrama UML, pero tal clase por lo general queda fuera y se supone, de manera implícita, que está ahí debido a la multiplicidad de la asociación.

Una *agregación* es un tipo especial de asociación que se indica mediante un diamante hueco en un extremo del ícono. Ello indica una relación "entero/parte", en la que la clase a la que apunta la flecha se considera como una "parte" de la clase en el extremo diamante de la asociación. Una *composición* es una agregación que indica fuerte propiedad de las partes. En una composición, las partes viven y mueren con el propietario porque no tienen papel en el sistema de software independiente del propietario. Vea la figura A1.3 para ejemplos de agregación y composición.

Una **Universidad** tiene una agregación de objetos **Edificio**, que representan los edificios que constituyen el campus. La universidad también tiene una colección de cursos. Si la universidad quebrara, los edificios todavía existirían (si se supone que la universidad no se destruye físicamente) y podría usar otras cosas, pero un objeto **Curso** no tiene uso fuera de la universidad en la cual se ofrece. Si la universidad deja de existir como una entidad empresarial, el objeto **Curso** ya no sería útil y, por tanto, también dejaría de existir.

Otro elemento común de un diagrama de clase es una *nota*, que se representa mediante una caja con una esquina doblada y se conecta a otros íconos mediante una línea punteada. Puede

FIGURA A1.3
Relación entre universidades, cursos y edificios



tener contenido arbitrario (texto y gráficos) y es similar a comentarios en lenguajes de programación. Puede contener comentarios acerca del papel de una clase o restricciones que todos los objetos de dicha clase deban satisfacer. Si los contenidos son una restricción, se encierran entre llaves. Observe la restricción unida a la clase **Curso** en la figura A1.3.

Diagramas de implementación

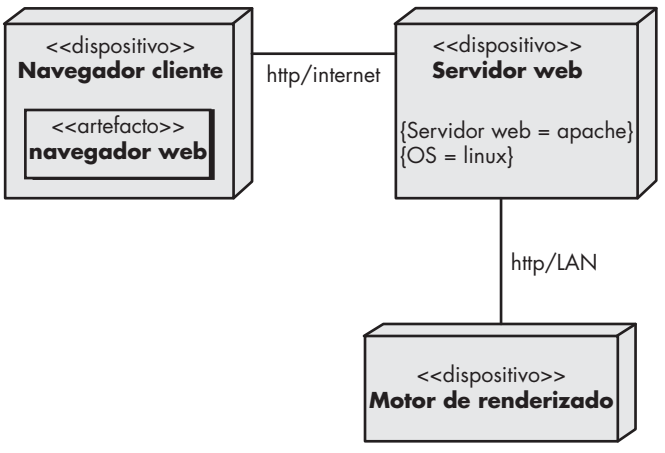
Un *diagrama de implementación* UML se enfoca en la estructura de un sistema de software y es útil para mostrar la distribución física de un sistema de software entre plataformas de hardware y entornos de ejecución. Suponga, por ejemplo, que desarrolla un paquete de renderizado de gráficos basado en web. Los usuarios de su paquete usarán su navegador web para ir a su sitio web e ingresar la información que se va a renderizar. Su sitio web renderizaría una imagen gráfica de acuerdo con la especificación del usuario y la enviaría de vuelta al usuario. Puesto que las gráficas renderizadas pueden ser computacionalmente costosas, usted decide mover el renderizado afuera del servidor web y hacia una plataforma separada. Por tanto, habrá tres dispositivos de hardware involucrados en su sistema: el cliente web (la computadora que corre un navegador del usuario), la computadora que alberga el servidor web y la computadora que alberga el motor de renderizado.

La figura A1.4 muestra el diagrama de implementación para tal paquete. En ese diagrama, los componentes de hardware se dibujan en cajas marcadas con "<<dispositivo>>". Las rutas de comunicación entre componentes de hardware se dibujan con líneas con etiquetas opcionales. En la figura A1.4, las rutas se etiquetan con el protocolo de comunicación y con el tipo de red utilizado para conectar los dispositivos.

Cada nodo que hay en un diagrama de implementación también puede anotarse con detalles del dispositivo. Por ejemplo, en la figura A1.4 se ilustra el navegador cliente para mostrar que contiene un artefacto que consiste en el software del navegador web. Un artefacto por lo general es un archivo que contiene software que corre en un dispositivo. También puede especificar valores etiquetados, como se muestra en la figura A1.4 en el nodo del servidor web. Dichos valores definen al proveedor del servidor web y al sistema operativo que usa el servidor.

Los diagramas de implementación también pueden mostrar nodos de entorno de ejecución, que se dibujan como cajas que contienen la etiqueta "<<entorno de ejecución>>". Dichos nodos representan sistemas, como sistemas operativos, que pueden albergar otro software.

Figura A1.4
Diagrama de implementación



Diagramas de uso de caso

Los casos de uso (capítulos 5 y 6) y el *diagrama de uso de caso* UML ayudan a determinar la funcionalidad y características del software desde la perspectiva del usuario. Para proporcionarle una aproximación a la manera en la que funcionan los casos de uso y los diagramas de uso de caso, se crearán algunos para una aplicación de software que gestiona archivos de música digital, similar al software iTunes de Apple. Algunas de las cosas que puede incluir el software son funciones para:

- Descargar un archivo de música MP3 y almacenarlo en la biblioteca de la aplicación.
- Capturar música de streaming (transmisión continua) y almacenarla en la biblioteca de la aplicación.
- Gestionar la biblioteca de la aplicación (por ejemplo, borrar canciones u organizarlas en listas de reproducción).
- Quemar en CD una lista de las canciones de la biblioteca.
- Cargar una lista de las canciones de la biblioteca en un iPod o reproductor MP3.
- Convertir una canción de formato MP3 a formato AAC y viceversa.

Ésta no es una lista exhaustiva, pero es suficiente para entender el papel de los casos de uso y los diagramas de uso de caso.

Un *caso de uso* describe la manera en la que un usuario interactúa con el sistema, definiendo los pasos requeridos para lograr una meta específica (por ejemplo, quemar una lista de canciones en un CD). Las variaciones en la secuencia de pasos describen varios escenarios (por ejemplo, ¿y si todas las canciones de la lista no caben en un CD?).

Un diagrama UML de uso de caso es un panorama de todos los casos de uso y sus relaciones. El mismo proporciona un gran cuadro de la funcionalidad del sistema. En la figura A1.5 se muestra un diagrama de uso de caso para la aplicación de música digital.

En este diagrama, la figura de palitos representa a un *actor* (capítulo 5) que se asocia con una categoría de usuario (u otro elemento de interacción). Por lo general, los sistemas complejos tienen más de un actor. Por ejemplo, una aplicación de máquina expendedora puede tener tres actores que representan clientes, personal de reparación y proveedores que rellenan la máquina.

En el diagrama de uso de caso, los casos de uso se muestran como óvalos. Los actores se conectan mediante líneas a los casos de uso que realizan. Observe que ninguno de los detalles de los casos de uso se incluye en el diagrama y, en vez de ello, necesita almacenarse por separado. Observe también que los casos de uso se colocan en un rectángulo, pero los actores no. Este rectángulo es un recordatorio visual de las fronteras del sistema y de que los actores están afuera del sistema.

Algunos casos de uso en un sistema pueden relacionarse mutuamente. Por ejemplo, existen pasos similares al de quemar una lista de canciones en un CD y cargar una lista de canciones en un iPod. En ambos casos, el usuario crea primero una lista vacía y luego agrega las canciones de la biblioteca a la lista. Para evitar duplicación en casos de uso, por lo general es mejor crear un nuevo caso de uso que represente la actividad duplicada y luego dejar que los otros casos de uso incluyan este nuevo caso de uso como uno de sus pasos. Tal inclusión se indica en los diagramas de uso de caso, como en la figura A1.6, mediante una flecha punteada etiquetada como “incluye”, que conecta un caso de uso con un caso de uso incluido.

Dado que despliega todos los casos de uso, un diagrama de uso de caso es un auxiliar útil para asegurar que cubrió toda la funcionalidad del sistema. En el organizador de música digital, seguramente querría más casos de uso, tal como uno para reproducir una canción de la biblioteca. Pero tenga en mente que la contribución más valiosa de casos de uso al proceso de desa-

FIGURA A1.5

Diagrama de caso de uso para el sistema de música

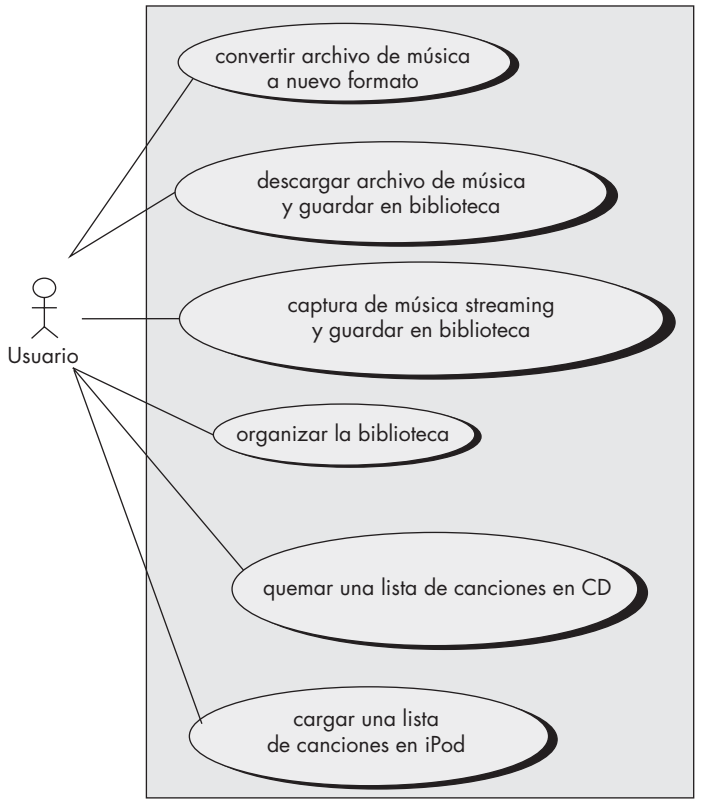
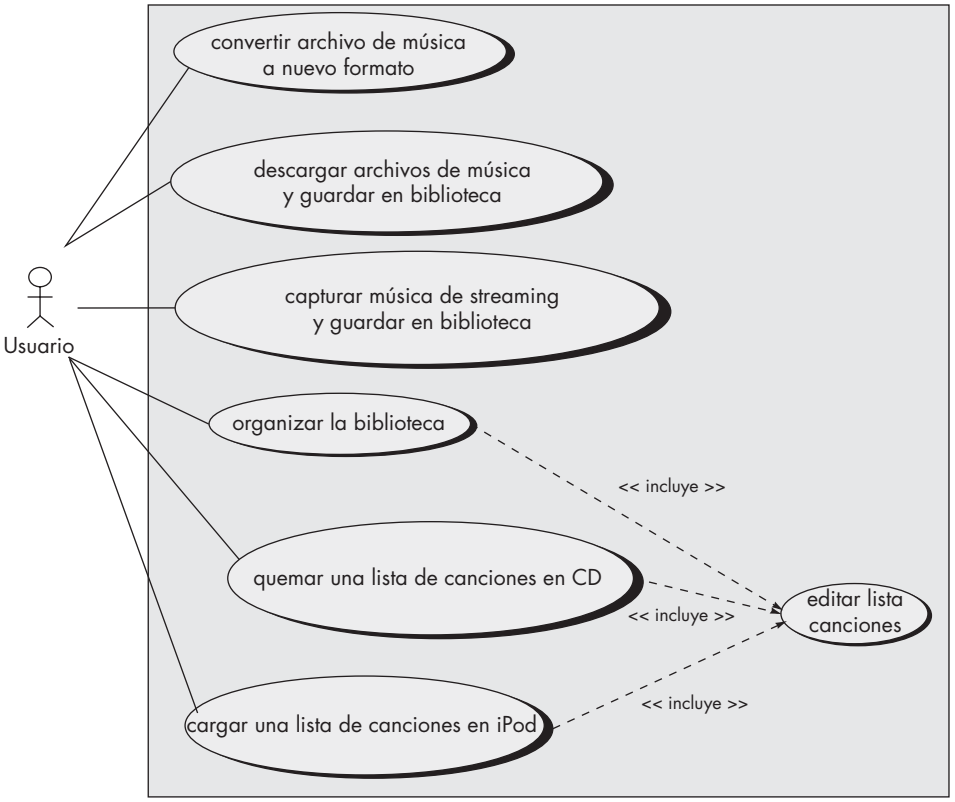


FIGURA A1.6

Diagrama de caso de uso con casos de uso incluidos



rrollo de software es la descripción textual de cada caso de uso, no el diagrama de uso de caso global [Fow04b]. Es a través de las descripciones que usted puede tener una comprensión clara de las metas del sistema que desarrolla.

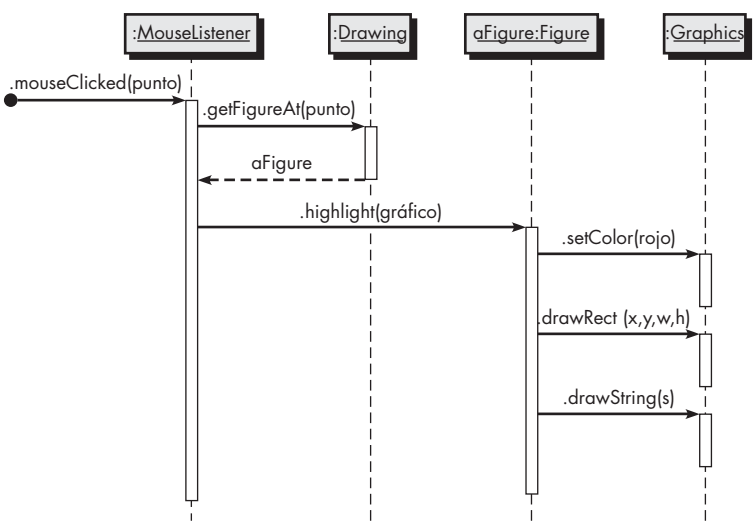
Diagramas de secuencia

En contraste con los diagramas de clase y con los diagramas de implementación, que muestran la estructura estática de un componente de software, un *diagrama de secuencia* se usa para mostrar las comunicaciones dinámicas entre objetos durante la ejecución de una tarea. Este tipo de diagrama muestra el orden temporal en el que los mensajes se envían entre los objetos para lograr dicha tarea. Puede usarse un diagrama de secuencia para mostrar las interacciones en un caso de uso o en un escenario de un sistema de software.

En la figura A1.7 se ve un diagrama de secuencia para un programa de dibujo. El diagrama muestra los pasos involucrados, resaltando una figura en un dibujo cuando se le da clic. Por lo general, cada caja de la fila que hay en la parte superior del diagrama corresponde a un objeto, aunque es posible hacer que las cajas modelen otras cosas, como clases. Si la caja representa un objeto (como es el caso en todos los ejemplos), entonces dentro de la caja puede establecerse de manera opcional el tipo del objeto, precedido por dos puntos. También se puede escribir un nombre del objeto antes de los dos puntos, como se muestra en la tercera caja de la figura A1.7. Abajo de cada caja hay una línea punteada llamada *línea de vida* del objeto. El eje vertical que hay en el diagrama de secuencia corresponde al tiempo, donde el tiempo aumenta conforme se avanza hacia abajo.

Un diagrama de secuencia muestra llamadas de método usando flechas horizontales desde el *llamador* hasta el *llamado*, etiquetado con el nombre del método y que opcionalmente incluye sus parámetros, sus tipos y el tipo de retorno. Por ejemplo, en la figura A1.7, **MouseListener** (escucha de ratón) llama al método *getFigureAt()* (obtener figura en) de **Drawing** (dibujo). Cuando un objeto ejecuta un método (es decir, cuando tiene un marco de activación en la pila), opcionalmente puede mostrar una barra blanca, llamada *barra de activación*, abajo de la línea de vida del objeto. En la figura A1.7, las barras de activación se dibujan para todas las llamadas de método. El diagrama también puede mostrar opcionalmente el retorno de una llamada de método con una flecha punteada y una etiqueta opcional. En la figura A1.7, el retorno de la llamada de método *getFigureAt()* se muestra con una etiqueta del nombre del objeto que regresa.

Figura A1.7
Ejemplo de diagrama de secuencia



Una práctica común, como se hizo en la figura A1.7, es dejar fuera la flecha de retorno cuando se llama un método nulo, pues desordena el diagrama a la vez que proporciona poca información de importancia. Un círculo negro con una flecha que sale de él indica un *mensaje encontrado* cuya fuente se desconoce o es irrelevante.

Ahora debe poder entenderse la tarea que implementa la figura A1.7. Una fuente desconocida llama al método *mouseClicked()* (clic del ratón) de un **MouseListener**, que pasa como argumento el punto donde ocurrió el clic. A su vez, el **MouseListener** llama al método *getFigureAt()* de un **Drawing**, que regresa una **Figure**. Luego el **MouseListener** llama el método resaltado de **Figure** y lo pasa como argumento al objeto **Graphics**. En respuesta, **Figure** llama tres métodos del objeto **Graphics** para dibujar la figura en rojo.

El diagrama en la figura A1.7 es muy directo y no contiene condicionales o bucles. Si se requieren estructuras de control lógico, probablemente sea mejor dibujar un diagrama de secuencia separado para cada caso, es decir, si el flujo del mensaje puede tomar dos rutas diferentes dependiendo de una condición, entonces dibuje dos diagramas de secuencia separados, uno para cada posibilidad.

Si se insiste en incluir bucles, condicionales y otras estructuras de control en un diagrama de secuencia, se pueden usar *marcos de interacción*, que son rectángulos que rodean partes del diagrama y que se etiquetan con el tipo de estructuras de control que representan. La figura A1.8 ilustra lo anterior, y muestra el proceso involucrado, resaltando todas las figuras dentro de un rectángulo determinado. El **MouseListener** envía el mensaje **rectDragged** (arrastre de recta). Entonces el **MouseListener** dice al dibujo que resalte todas las figuras en el rectángulo, llamando al método *highlightFigures()* (resaltar figuras) y pasando al rectángulo como argumento. El método hace bucles a través de todos los objetos **Figure** en el objeto **Drawing** y, si **Figure** intersecta el rectángulo, se pide a **Figure** que se resalte a sí mismo. Las frases entre corchetes se llaman *guardias*, que son condiciones booleanas que deben ser verdaderas si debe continuar la acción dentro del marco de interacción.

Existen muchas otras características especiales que pueden incluirse en un diagrama de secuencia. Por ejemplo:

1. Se puede distinguir entre mensajes sincrónicos y asíncronos. Los primeros se muestran con puntas de flecha sólidas mientras que los asíncronos lo hacen con puntas de flecha huecas.
2. Se puede mostrar un objeto que envía él mismo un mensaje con una flecha que parte del objeto, gira hacia abajo y luego apunta de vuelta hacia el mismo objeto.

FIGURA A1.8
Diagrama de secuencia con dos marcos de interacción

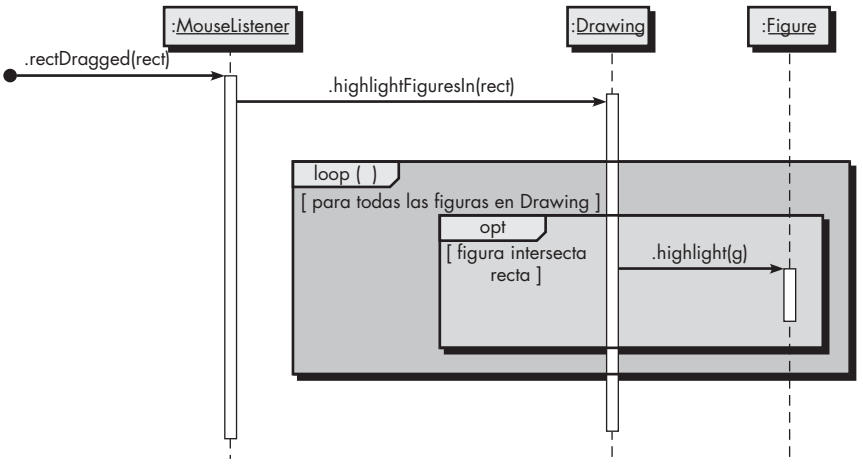
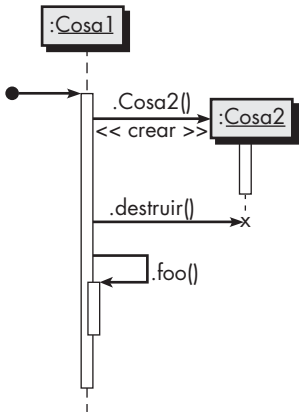


FIGURA A1.9
Creación, destrucción y bucles en diagramas de secuencia



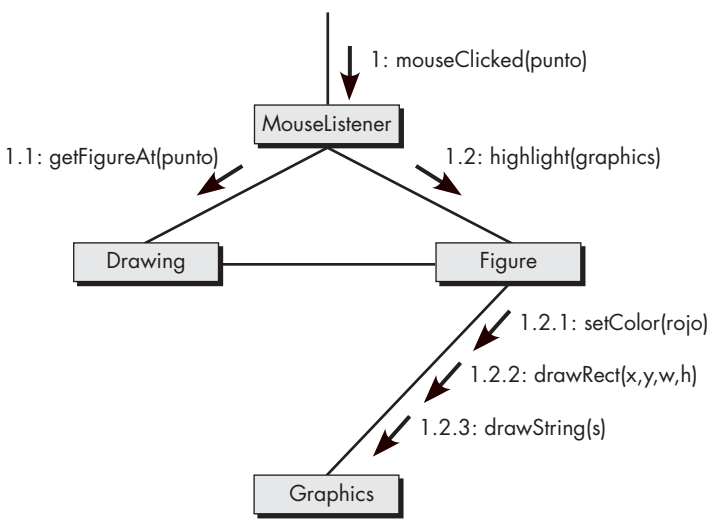
3. Se puede mostrar la creación de objeto dibujando una flecha etiquetada de manera adecuada (por ejemplo, con una etiqueta <<crear>>) hacia una caja de objeto. En este caso, la caja aparecerá en el diagrama más abajo que las cajas correspondientes a objetos que ya existen cuando comienza la acción.
4. Se puede mostrar destrucción de objeto mediante una gran X al final de la línea de vida del objeto. Otros objetos pueden destruir un objeto, en cuyo caso una flecha apunta desde el otro objeto hacia la X. Una X también es útil para indicar que un objeto ya no se usa y que, por tanto, está listo para la colección de basura.

Las últimas tres características se muestran en el diagrama de secuencia de la figura A1.9.

DIAGRAMAS DE COMUNICACIÓN

El *diagrama de comunicación* UML (llamado "diagrama de colaboración" en UML 1.X) proporciona otro indicio del orden temporal de las comunicaciones, pero enfatiza las relaciones entre los objetos y clases en lugar del orden temporal. El diagrama de comunicación que se ilustra en la figura A1.10 despliega las mismas acciones que se muestran en el diagrama de secuencia de la figura A1.7.

FIGURA A1.10
Diagrama de comunicación UML



En un diagrama de comunicación, los objetos interactuantes se representan mediante rectángulos. Las asociaciones entre objetos lo hacen mediante líneas que conectan los rectángulos. Por lo general, en el diagrama existe una flecha entrante hacia un objeto que comienza la secuencia de pase de mensaje. Esa flecha se etiqueta con un número y un nombre de mensaje. Si el mensaje entrante se etiqueta con el número 1 y si hace que el objeto receptor invoque otros mensajes en otros objetos, entonces los mencionados mensajes se representan mediante flechas desde el emisor hacia el receptor a lo largo de una línea de asociación y reciben números 1.1, 1.2, etc., en el orden en el que se llaman. Si tales mensajes a su vez invocan otros mensajes, se agrega otro punto decimal y otro número al número que etiqueta dichos mensajes para indicar un anidado posterior del pase de mensaje.

En la figura A1.10 se ve que el mensaje **mouseClicked** invoca los métodos *getFigureAt()* y luego *highlight()*. El mensaje *highligh t()* invoca otros tres mensajes: *setColor()* (establecer color), *drawRect()* (dibujar recta) y *drawstring()* (dibujar cadena). La numeración en cada etiqueta muestra el anidado y la naturaleza secuencial de cada mensaje.

Existen muchas características opcionales que pueden agregarse a las etiquetas de flecha. Por ejemplo, puede preceder el número con una letra. Una flecha entrante podría etiquetarse **A1: mouseClicked(point)**, lo que indica una hebra de ejecución, A. Si otros mensajes se ejecutan en otras hebras, su etiqueta estaría precedida por una letra diferente. Por ejemplo, si el método *mouseClicked()* se ejecuta en la hebra A, pero crea una nueva hebra B e invoca *highlight()* en dicha hebra, entonces la flecha desde **MouseListener** hacia **Figure** se etiquetaría **1.B2: highlight(graphics)**.

Si el lector está interesado en mostrar las relaciones entre los objetos, además de los mensajes que se envíen entre ellos, probablemente el diagrama de comunicación es una mejor opción que el diagrama de secuencia. Si está más interesado en el orden temporal del paso de mensajes, entonces un diagrama de secuencia probablemente es mejor.

Diagramas de actividad

Un *diagrama de actividad* UML muestra el comportamiento dinámico de un sistema o de parte de un sistema a través del flujo de control entre acciones que realiza el sistema. Es similar a un diagrama de flujo, excepto porque un diagrama de actividad puede mostrar flujos concurrentes.

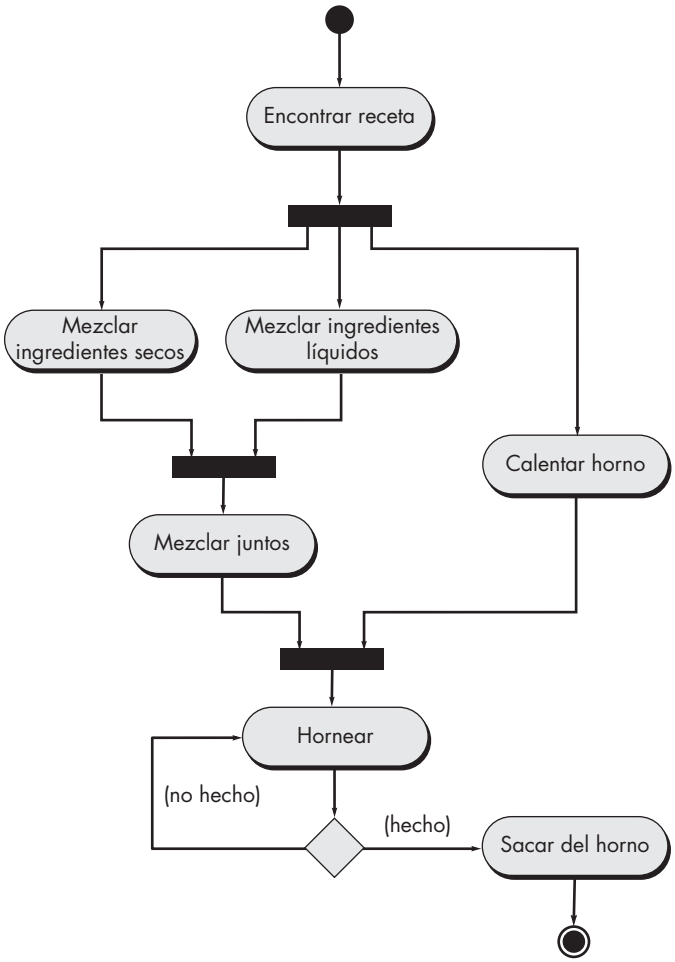
El componente principal de un diagrama de actividad es un nodo *acción*, representado mediante un rectángulo redondeado, que corresponde a una tarea realizada por el sistema de software. Las flechas desde un nodo acción hasta otro indican el flujo de control; es decir, una flecha entre dos nodos acción significa que, después de completar la primera acción, comienza la segunda acción. Un punto negro sólido forma el *nodo inicial* que indica el punto de inicio de la actividad. Un punto negro rodeado por un círculo negro es el *nodo final* que indica el fin de la actividad.

Un *tenedor* (*fork*) representa la separación de actividades en dos o más actividades concurrentes. Se dibuja como una barra negra horizontal con una flecha apuntando hacia ella y dos o más flechas apuntando en sentido opuesto. Cada flecha continua representa un flujo de control que puede ejecutarse de manera concurrente con los flujos correspondientes a las otras flechas continuas. Dichas actividades concurrentes pueden realizarse en una computadora, usando diferentes hebras o incluso diferentes computadoras.

La figura A1.11 muestra un ejemplo de diagrama de actividad que involucra hornear un pastel. El primer paso es encontrar la receta. Una vez encontrada pueden medirse los ingredientes secos y líquidos, mezclarse y precalentar el horno. La mezcla de los ingredientes secos puede hacerse en paralelo con la mezcla de los ingredientes líquidos y el precalentado del horno.

Una *unión* (*join*) es una forma de sincronizar flujos de control concurrentes. Se representa mediante una barra negra horizontal con dos o más flechas entrantes y una flecha saliente. El

FIGURA A1.11
Diagrama de actividad UML que muestra cómo hornear un pastel



flujo de control representado por la flecha saliente no puede comenzar la ejecución hasta que todos los flujos representados por las flechas entrantes se hayan completado. En la figura A1.11 se tiene una unión antes de la acción de mezclar en conjunto los ingredientes líquidos y secos. Esta unión indica que todos los ingredientes secos deben mezclarse y que debe hacerse lo mismo con todos los ingredientes líquidos antes de poder combinar las dos mezclas. La segunda unión en la figura indica que, antes de comenzar a hornear el pastel, todos los ingredientes deben mezclarse juntos y el horno debe estar a la temperatura correcta.

Un nodo de *decisión* corresponde a una rama en el flujo de control con base en una condición. Tal nodo se despliega como un triángulo blanco con una flecha entrante y dos o más flechas salientes. Cada flecha saliente se etiqueta con una guardia (una condición dentro de corchetes). El flujo de control sigue la flecha saliente cuya guardia es verdadera. Es recomendable asegurarse de que las condiciones cubran todas las posibilidades, de modo que exactamente una de ellas sea verdadera cada vez que se llegue a un nodo de decisión. La figura A1.11 muestra un nodo de decisión que sigue al horneado del pastel. Si el pastel está hecho, entonces se saca del horno. De otro modo, se hornea un poco más.

Una de las cosas que no dice el diagrama de actividad de la figura A1.11 es quién o qué hace cada una de las acciones. Con frecuencia, no importa la división exacta de la mano de obra. Pero si quiere indicar cómo se dividen las acciones entre los participantes, puede decorar el diagrama de actividad con "canales", como se muestra en la figura A1.12. Los *canales*, como el nombre

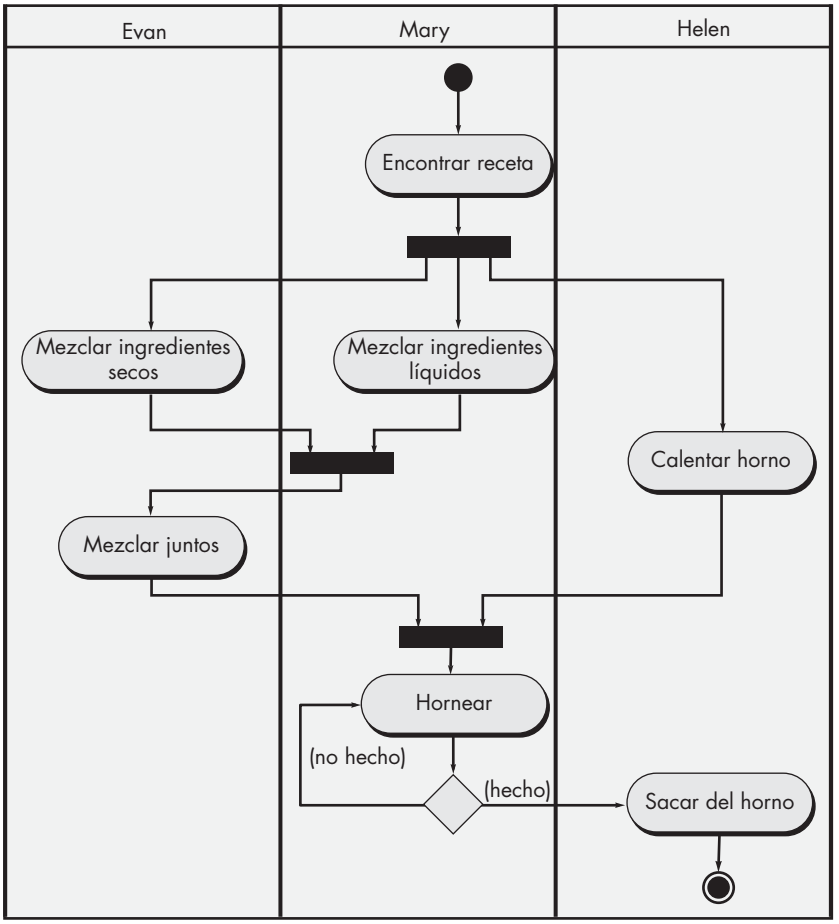


FIGURA A1.12
Diagrama de actividad de horneado de pastel con "carriles de natación" agregados

implica, se forman dividiendo el diagrama en tiras o "carriles" (como si fuera una alberca con carriles de natación), cada uno de los cuales corresponde a uno de los participantes. Todas las acciones en un carril las realiza el participante correspondiente. En la figura A1.12, Evan es responsable de la mezcla de los ingredientes secos y, luego, de mezclar juntos los ingredientes secos y los líquidos; Helen es responsable de calentar el horno y sacar el pastel; y Mary es responsable de todo lo demás.

DIAGRAMAS DE ESTADO

El comportamiento de un objeto en un punto particular en el tiempo con frecuencia depende del estado del objeto; es decir, de los valores de sus variables en dicho momento. Como ejemplo trivial, considere un objeto con una variable de instancia booleana. Cuando se pide realizar una operación, el objeto puede hacer una cosa si dicha variable es *verdadera* y hacer algo más si es *falsa*.

Un *diagrama de estado* UML modela los estados de un objeto, las acciones que se realizan dependiendo de dichos estados y las transiciones entre los estados del objeto.

Como ejemplo, considere el diagrama de estado para una parte de un compilador Java. La entrada al compilador es un archivo de texto, que puede considerarse como una larga cadena de caracteres. El compilador lee caracteres uno a uno y a partir de ellos determina la estructura del programa. Una pequeña parte de este proceso de lectura de caracteres involucra ignorar

caracteres de "espacio blanco" (por ejemplo, los caracteres *espacio*, *tabulador*, *línea nueva* y *retorno*) y caracteres dentro de un comentario.

Suponga que el compilador delega a un **EliminadorEspacioBlancoyComentario** la labor de avanzar sobre los caracteres de espacio blanco y sobre los caracteres dentro de un comentario, es decir, la labor de dicho objeto es leer los caracteres de entrada hasta que todos los caracteres de espacio blanco y comentario se leyeron, punto donde regresa el control al compilador para leer y procesar caracteres no en blanco y no de comentario. Piense en la manera en la que el objeto **EliminadorEspacioBlancoyComentario** lee los caracteres y determina si el siguiente carácter es de espacio blanco o parte de un comentario. El objeto puede verificar los espacios blancos, probando los siguientes caracteres contra " ", "\t", "\n" y "\r". ¿Pero cómo determina si el siguiente carácter es parte de un comentario? Por ejemplo, cuando ve "/" por primera vez, todavía no sabe si dicho carácter representa un operador división, parte del operador /= o el comienzo de una línea o bloque de comentario. Para hacer esta determinación, **EliminadorEspacioBlancoyComentario** necesita anotar el hecho de que vio una "/" y luego moverse hacia el siguiente carácter. Si el carácter siguiente a "/" es otra "/" o un "*", entonces **EliminadorEspacioBlancoyComentario** sabe que ahora lee un comentario y puede avanzar al final del comentario sin procesar o guardar algún carácter. Si el carácter siguiente al primer "/" es distinto a "/" o a "*", entonces **EliminadorEspacioBlancoyComentario** sabe que "/" representa el operador división o parte del operador /= y, por tanto, avanza a través de los caracteres.

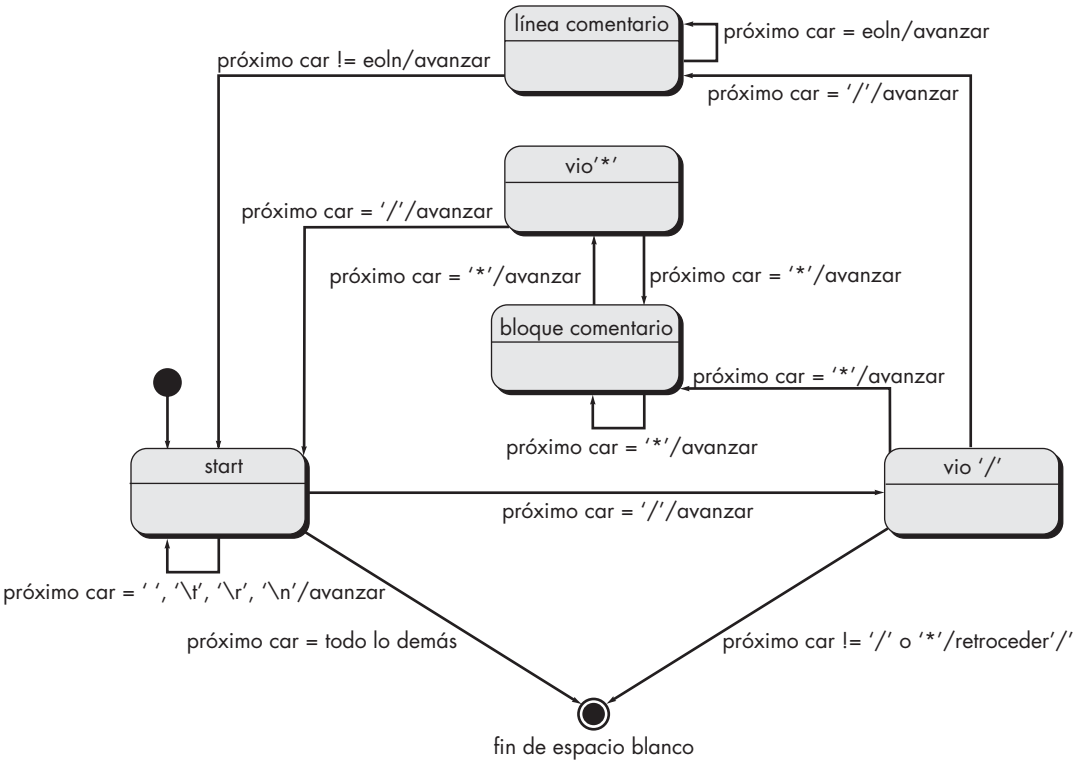
En resumen, conforme **EliminadorEspacioBlancoyComentario** lee los caracteres, necesita seguir la pista de muchas cosas, incluido si el carácter actual es espacio blanco, si el carácter previo que lee fue "/", si actualmente lee caracteres en un comentario, si llegó al final del comentario, etc. Todos éstos corresponden a diferentes estados del objeto **EliminadorEspacioBlancoyComentario**. En cada uno de estos estados, **EliminadorEspacioBlancoyComentario** se comporta de manera diferente con respecto al siguiente carácter que lee.

Para ayudarlo a visualizar todos los estados de este objeto y la manera en la que cambia de estado, puede usar un diagrama de estado UML, como se muestra en la figura A1.13. Un diagrama de estado muestra los estados mediante rectángulos redondeados, cada uno de los cuales tiene un nombre en su mitad superior. También existe un círculo llamado "pseudoestado inicial", que en realidad no es un estado y en vez de ello sólo apunta al estado inicial. En la figura A1.13, el estado **start** es el estado inicial. Las flechas de un estado a otro estado indican transiciones o cambios en el estado del objeto. Cada transición se etiqueta con un evento disparador, una diagonal (/) y una actividad. Todas las partes de las etiquetas de transición son opcionales en los diagramas de estado. Si el objeto está en un estado y el evento disparador ocurre para una de sus transiciones, entonces se realiza dicha actividad de transición y el objeto toma un nuevo estado, indicado por la transición. Por ejemplo, en la figura A1.13, si el objeto **EliminadorEspacioBlancoyComentario** está en el estado **start** y el siguiente carácter es "/", entonces **EliminadorEspacioBlancoyComentario** avanza desde dicho carácter y cambia al estado **vio '/'**. Si el carácter después de "/" es otra "/", entonces el objeto avanza al estado **línea comentario** y permanece ahí hasta que lee un carácter de fin de línea. Si en vez de ello el siguiente carácter después de "/" es "*", entonces el objeto avanza al estado **bloque comentario** y permanece ahí hasta que ve otro "*" seguido por un "/", que indica el final del bloque comentario. Estudie el diagrama para asegurarse de que lo entiende. Observe que, después de avanzar por el espacio en blanco o por un comentario, **EliminadorEspacioBlancoyComentario** regresa al estado **start** y comienza de nuevo. Dicho comportamiento es necesario, pues puede haber varios comentarios sucesivos o caracteres de espacio en blanco antes de cualquier otro carácter en el código fuente Java.

Un objeto puede transitar a un estado final, lo que se indica mediante un círculo negro con un círculo blanco alrededor de él, lo que indica que ya no hay más transiciones. En la figura

FIGURA A1.13

Diagrama de estado para avanzar sobre espacio blanco y comentarios en Java



A1.13, el objeto **EliminadorEspacioBlancoyComentario** termina cuando el siguiente carácter no es espacio en blanco o parte de un comentario. Observe que todas las transiciones, excepto las dos que conducen al estado final, tienen actividades que consisten en avanzar al siguiente carácter. Las dos transiciones hacia el estado final no avanzan sobre el siguiente carácter porque el siguiente carácter es parte de una palabra o símbolo de interés para el compilador. Observe que, si el objeto está en el estado **vio '/'**, pero el siguiente carácter no es "/" o "*", entonces "/" es un operador división o parte del operador /= y, por tanto, no se quiere avanzar. De hecho, se quiere regresar un carácter para hacer el "/" en el siguiente carácter, de modo que "/" pueda usarse por parte del compilador. En la figura A1.13, esta actividad se etiqueta como retroceder '/'.

Un diagrama de estado le ayudará a descubrir situaciones perdidas o inesperadas, es decir, con un diagrama de estado, es relativamente sencillo garantizar que todos los posibles eventos disparadores para todos los estados posibles se representaron. Por ejemplo, en la figura A1.13, puede verificar fácilmente que cada estado incluyó transiciones para todos los posibles caracteres.

Los diagramas de estado UML pueden contener muchas otras características no incluidas en la figura A1.13. Por ejemplo, cuando un objeto está en un estado, por lo general no hace más que sentarse y esperar que ocurra un evento disparador. Sin embargo, existe un tipo especial de estado, llamado *estado de actividad*, donde el objeto realiza alguna actividad, llamada *hacer actividad*, mientras está en dicho estado. Para indicar que un estado es un estado de actividad en el diagrama de estado, se incluye, en la mitad inferior del rectángulo redondeado del estado, la frase "do/" seguida por la actividad que debe realizar mientras está en dicho estado. El "hacer actividad" puede terminar antes de que ocurra cualquier transición de estado, después de lo cual el estado de actividad se comporta como un estado normal de espera. Si una transición del estado de actividad ocurre antes de terminar el "hacer actividad", entonces se interrumpe el "hacer actividad".

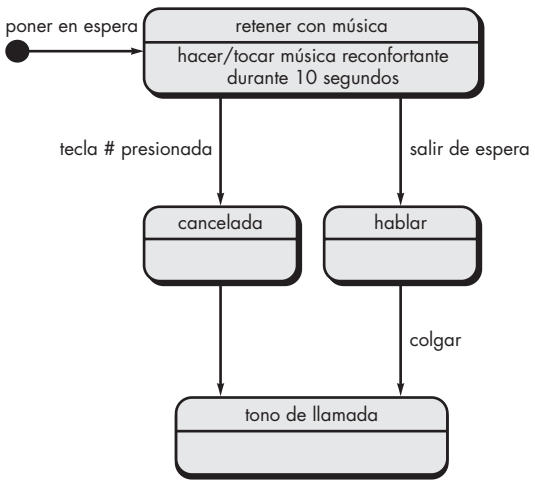


FIGURA A1.14
Diagrama de estado con un estado de actividad y una transición sin disparador

Puesto que un evento disparador es opcional cuando ocurre una transición, es posible que ningún evento disparador pueda mencionarse como parte de una etiqueta de transición. En tales casos, para estados de espera normales, el objeto inmediatamente transitará de dicho estado al nuevo estado. Para estados de actividad, tal transición se realiza tan pronto como termina el "hacer actividad".

La figura A1.14 ilustra esta situación, usando los estados para la operación de un teléfono. Cuando un llamador se coloca en espera, la llamada pasa al estado **retener con música** (la música reconfortante suena durante 10 segundos). Después de 10 segundos, el "hacer actividad" del estado se completa y el estado se comporta como un estado normal de no actividad. Si el llamador presiona la tecla # cuando la llamada está en el estado **retener con música**, la llamada transita hacia el estado **cancelado** e inmediatamente al estado **tono de llamada**. Si la tecla # se presiona antes de completar los 10 segundos de música reconfortante, el "hacer actividad" se interrumpe y la música cesa inmediatamente.

Lenguaje de restricción de objeto (panorama)

La gran variedad de diagramas disponibles como parte de UML le brindan un rico conjunto de formas representativas para el modelo de diseño. Sin embargo, con frecuencia son suficientes las representaciones gráficas. Acaso necesite un mecanismo para representar información de manera explícita y formal que restrinja algún elemento del modelo de diseño. Desde luego, es posible describir las restricciones en lenguaje natural como el inglés, pero este enfoque invariablemente conduce a inconsistencia y ambigüedad. Por esta razón, parece apropiado un lenguaje más formal, extraído de la teoría de conjuntos y de los lenguajes de especificación formales (vea el capítulo 21), pero que tenga la sintaxis un tanto menos matemática de un lenguaje de programación.

El *lenguaje de restricción de objeto* (LRO) complementa al UML, permitiéndole usar una gramática y sintaxis formales para construir enunciados sin ambigüedades acerca de varios elementos del modelo de diseño (por ejemplo, clases y objetos, eventos, mensajes, interfaces). Los enunciados LRO más simples se construyen en cuatro partes: 1) un *contexto* que define la situación limitada en la que es válido el enunciado, 2) una *propiedad* que representa algunas características del contexto (por ejemplo, una propiedad puede ser un atributo si el contexto es una clase), 3) una *operación* (por ejemplo, aritmética, orientada a conjunto) que manipule o califique

una propiedad y 4) palabras clave (por ejemplo, **if, then, else, and, or, not, implies**) que se usan para especificar expresiones condicionales.

Como ejemplo simple de una expresión LRO, considere el sistema de impresión que se estudió en el capítulo 10. La condición guardia se coloca en el evento **jobCostAccepted** que causa una transición entre los estados *computingJobCost* y *formingJob* dentro del diagrama de estado para el objeto **PrintJob** (figura 10.9). En el diagrama (figura 10.9), la condición guardia se expresa en lenguaje natural e implica que la autorización sólo puede ocurrir si el cliente está autorizado para aprobar el costo del trabajo. En LRO, la expresión puede tomar la forma:

```
customer
   self.authorizationAuthority = 'yes'
```

donde un atributo booleano, **authorizationAuthority**, de la clase (en realidad una instancia específica de la clase) llamada **Customer** debe establecerse en "sí" para que se satisfaga la condición guardia.

Conforme se crea el modelo de diseño, con frecuencia existen instancias en las que deben satisfacerse pre o poscondiciones antes de completar alguna acción especificada por el diseño. LRO proporciona una poderosa herramienta para especificar pre y poscondiciones de manera formal. Como ejemplo, considere una extensión al sistema local de impresión (que se estudió como ejemplo en el capítulo 10) donde el cliente proporcione un límite de costo superior (upper cost bound) para el trabajo de impresión y una fecha de entrega "fatal", al mismo tiempo que se especifican otras características del trabajo de impresión. Si las estimaciones de costo y entrega superan dichos límites, el trabajo no se presenta y debe notificársele al cliente. En LRO, un conjunto de pre y poscondiciones puede especificarse de la forma siguiente:

```
context PrintJob::validate(upperCostBound : Integer, custDeliveryReq :
   Integer)
 pre: upperCostBound > 0
   and custDeliveryReq > 0
   and self.jobAuthorization = 'no'
 post: if self.totalJobCost <= upperCostBound
      and self.deliveryDate <= custDeliveryReq
    then
     self.jobAuthorization = 'yes'
    endif
```

Este enunciado LRO define una invariante (**inv**): condiciones que deben existir antes (pre) y después (pos) de algún comportamiento. Inicialmente, una precondición establece que el límite de costo y la fecha de entrega deben especificarse por parte del cliente, y la autorización debe establecerse en "no". Después de determinar costos y entrega, se aplica la poscondición especificada. También debe observarse que la expresión:

```
self.jobAuthorization = 'yes'
```

no asigna el valor "sí", sino que declara que **jobAuthorization** debe establecerse en "sí" para cuando la operación termine. Una descripción completa del LRO está más allá del ámbito de este apéndice. La especificación LRO completa puede obtenerse en **www.omg.org/technology/documents/formal/ocl.htm**

Lecturas y fuentes de información adicionales

Decenas de libros estudian UML. Los que abordan la versión más reciente incluyen: Miles y Hamilton (*Learning UML 2.0,* O'Reilly Media, Inc., 2006); Booch, Rumbaugh, y Jacobson (*Unified Modeling Language User*

Guide, 2a. ed., Addison-Wesley, 2005), Ambler (*The Elements of UML 2.0 Style,* Cambridge University Press, 2005), y Pilone y Pitman (*UML 2.0 in a Nutshell,* O'Reilly Media, Inc., 2005).

En internet está disponible una gran variedad de fuentes de información acerca del UML en el modelado de ingeniería del software. Una lista actualizada de referencias en la World Wide Web puede encontrarse bajo "análisis" y "diseño" en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

APÉNDICE 2

CONCEPTOS ORIENTADOS A OBJETO

CONCEPTOS CLAVE

atributos 745
clases 744
características 748
frontera 745
controlador 746
definición 743
diseño 747
entidad 745
encapsulación 743
herencia 746
mensajes 746
métodos 745
operaciones 745
polimorfismo 747
servicios 745
subclase 744
superclase 744

¿Qué es un punto de vista orientado a objeto (OO)? ¿Por qué un método se considera orientado a objeto? ¿Qué es un objeto? Conforme los conceptos OO ganaron muchos adherentes durante las décadas de 1980 y 1990, había muchas opiniones diferentes acerca de las respuestas correctas a dichas preguntas, pero en la actualidad surgió una visión coherente de los conceptos OO. Este apéndice se diseñó para ofrecer un breve panorama de este importante tema e introducir los conceptos y terminología básicos.

Para entender el punto de vista orientado a objeto, considere un ejemplo de un objeto del mundo real: el objeto sobre el cual se sienta en este momento, una silla. **Silla** es una subclase de una clase mucho más grande que puede llamar **PiezaDeMobiliario**. Las sillas individuales son miembros (por lo general llamadas instancias) de la clase **Silla**. Un conjunto de atributos genéricos pueden asociarse con cada objeto en la clase **PiezaDeMobiliario**. Por ejemplo, todos los muebles tienen un costo, dimensiones, peso, ubicación y color, entre muchos posibles atributos. Lo mismo se aplica si se habla de una mesa, una silla, un sofá o un armario. Puesto que **Silla** es miembro de **PiezaDeMobiliario**, **Silla** hereda todos los atributos definidos para la clase.

Se busca una definición anecdótica de una clase al describir sus atributos, pero algo falta. Todo objeto de la clase **PiezaDeMobiliario** puede manipularse de varias formas. Puede comprarse y venderse, modificarse físicamente (por ejemplo, puede serruchar una pata o pintar el objeto de morado) o moverlo de un lugar a otro. Cada una de estas *operaciones* (otros términos son *servicios* o *métodos*) modificarán uno o más atributos del objeto. Por ejemplo, si el atributo ubicación es un ítem de datos compuesto definido como

ubicación = edificio + piso + habitación

entonces una operación denominada *mover()* modificaría uno o más de los ítems de datos (**edificio, piso** o **habitación**) que forman el atributo **ubicación**. Para hacer esto, *mover()* debe tener "conocimiento" de dichos ítems de datos. La operación *mover()* podría usarse para una silla o una mesa, en tanto ambos sean instancias de la clase **PiezaDeMobiliario**. Las operaciones válidas para la clase **PiezaDeMobiliario** [*comprar()*, *vender()*, *peso()*] son especificadas como parte de la definición de clase y son heredadas por todas las instancias de la clase.

La clase **Silla** (y todos los objetos en general) encapsulan datos (los valores de atributo que definen la silla), operaciones (las acciones que se aplican para cambiar los atributos de silla), otros objetos, constantes (valores de conjunto) y otra información relacionada. *Encapsulación* significa que toda esta información se empaca bajo un nombre y puede reutilizarse como un componente de especificación o programa.

Ahora que se introdujeron algunos conceptos básicos, una definición más formal de *orientado a objeto* resultará más significativa. Coad y Yourdon [Coa91] definen el término de esta forma:

Orientado a objeto = objetos + clasificación + herencia + comunicación

Tres de los conceptos ya se introdujeron. La comunicación se estudia más tarde, en este apéndice.

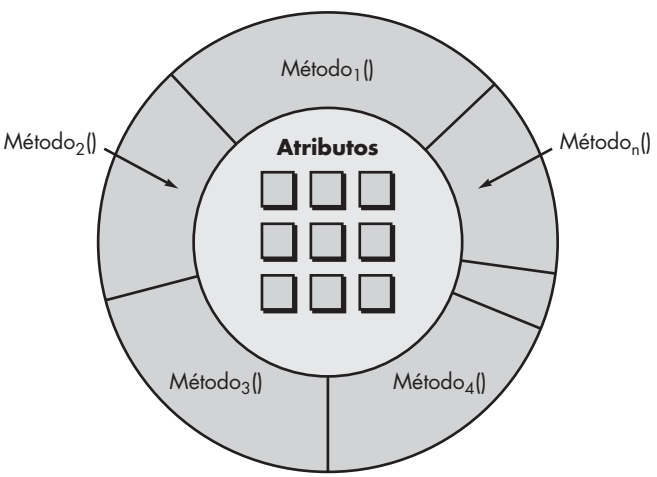
Clases y objetos

Una clase es un concepto OO que encapsula los datos y abstracciones procedurales requeridos para describir el contenido y el comportamiento de alguna entidad del mundo real. Las abstracciones de datos que describen la clase se encierran mediante una "pared" de abstracciones procedurales [Tay90] (representadas en la figura A2.1) que son capaces de manipular los datos de alguna forma. En una clase bien diseñada, la única forma de llegar a los atributos (y operar sobre ellas) es ir a través de uno de los métodos que forman la "pared" que se ilustra en la figura. Por tanto, la clase encapsula datos (dentro de la pared) y el procesamiento que los manipula (los métodos que constituyen la pared). Esto logra que la información se oculte (capítulo 8) y que se reduzca el impacto de los efectos colaterales asociados con el cambio. Dado que los métodos tienden a manipular un número limitado de atributos, su cohesión mejora, y puesto que la comunicación ocurre solamente a través de los métodos que constituyen la "pared", la clase tiende a estar menos fuertemente acoplada a otros elementos de un sistema.[1]

Dicho de otra forma, una clase es una descripción generalizada (por ejemplo, una plantilla o plano) de una colección de objetos similares. Por definición, los objetos son instancias de una clase específica y heredan sus atributos y las operaciones que están disponibles para manipular esos atributos. Una *superclase* (con frecuencia llamada *clase base*) es una generalización de un conjunto de clases que se relacionan con ella. Una *subclase* es una especialización de la superclase. Por ejemplo, la superclase **VehículoDeMotor** es una generalización de las clases **Camión**, **SUV**, **Automóvil** y **Van**. La subclase **Automóvil** hereda todos los atributos de **VehículoDeMotor**, pero además incorpora atributos adicionales que son específicos solamente de automóviles.

Estas definiciones implican la existencia de una jerarquía de clase en la que los atributos y operaciones de la superclase se heredan por parte de la subclase, que puede agregar atributos y métodos "privados" adicionales. Por ejemplo, las operaciones *sentarEn()* y *voltear()* pueden ser privativas de la subclase **Silla**.

Figura A2.1
Representación esquemática de una clase



1 Sin embargo, debe observar que el acoplamiento puede convertirse en un serio problema en los sistemas OO, cuando las clases de varias partes del sistema se usan como los tipos de atributos de datos y los argumentos como métodos. Aun cuando el acceso a los objetos sólo puede ser a través de llamados a procedimiento, esto no significa que el acoplamiento es necesariamente bajo, sólo más bajo que si se permitiera el acceso directo al interior de los objetos.

Atributos

Usted aprendió que los atributos se vinculan a las clases y que describen la clase en alguna forma. Un atributo puede tomar un valor definido por un *dominio* enumerado. En la mayoría de los casos, un dominio es simplemente un conjunto de valores específicos. Por ejemplo, suponga que una clase **Automóvil** tiene un atributo **color**. El dominio de valores para **color** es {**blanco, negro, plata, gris, azul, rojo, amarillo, verde**}. En situaciones más complejas, el dominio puede ser una clase. Continuando con el ejemplo, la clase **Automóvil** también tiene un atributo **trenPotencia** que en sí mismo es una clase. La clase **TrenPotencia** contendría atributos que describen el motor y la transmisión específicos del vehículo.

Las *características* (valores del dominio) pueden aumentar al asignar un valor por defecto (característica) a un atributo. Por ejemplo, el atributo **color** por defecto es **blanco**. También puede ser útil asociar una probabilidad con una característica particular al asignar pares {valor, probabilidad}. Considere el atributo **color** para automóvil. En algunas aplicaciones (por ejemplo, planificar la fabricación) puede ser necesario asignar una probabilidad a cada uno de los colores (por ejemplo, blanco y negro son enormemente probables como colores de automóvil).

Operaciones, métodos y servicios

Un objeto encapsula datos (representados como una colección de atributos) y los algoritmos que los procesan. Dichos algoritmos se llaman *operaciones*, *métodos* o *servicios*[2] y pueden verse como componentes de procesamiento.

Cada una de las operaciones que se encapsula mediante un objeto proporciona una representación de uno de los comportamientos del objeto. Por ejemplo, la operación *ObtenerColor()* para el objeto **Automóvil** extraerá el color almacenado en el atributo **color**. La implicación de la existencia de estas operaciones es que la clase **Automóvil** se diseñó para recibir un estímulo (a los estímulos se les llama *mensajes*) que solicitan el color de la instancia particular de una clase. Siempre que un objeto recibe un estímulo, inicia cierto comportamiento. Esto puede ser tan simple como recuperar el color del automóvil o tan complejo como el inicio de una cadena de estímulos que pasan entre varios objetos diferentes. En el último caso, considere un ejemplo donde el estímulo inicial recibido por el **Objeto 1** da como resultado la generación de otros dos estímulos que se envían al **Objeto 2** y al **Objeto 3**. Las operaciones encapsuladas por el segundo y tercer objetos actúan sobre los estímulos, y regresan información necesaria al primer objeto. Entonces el **Objeto 1** usa la información devuelta para satisfacer el comportamiento demandado por el estímulo inicial.

Conceptos de análisis y diseño orientado a objeto

El modelado de requerimientos (también llamado modelado de análisis) se enfoca principalmente en clases que se extraen directamente del enunciado del problema. Dichas *clases de entidad* por lo general representan cosas que deben almacenarse en una base de datos y que persisten a lo largo de la duración de la aplicación (a menos que se borren de manera específica).

El diseño refina y extiende el conjunto de clases de entidad. Las clases frontera y controlador se desarrollan y/o refinan durante el diseño. Las *clases frontera* crean la interfaz (por ejemplo, pantalla interactiva y reportes impresos) que el usuario ve y con la cual interactúa conforme se

2 En el contexto de esta discusión se usa el término *operaciones*, pero los términos *métodos* y *servicios* son igualmente populares.

usa el software. Las clases frontera se diseñan con la responsabilidad de gestionar la forma en la que los objetos entidad se presentan a los usuarios.

Las *clases controlador* se diseñan para gestionar: 1) la creación o actualización de objetos entidad, 2) la instanciación de objetos frontera conforme obtienen información de objetos entidad, 3) comunicación compleja entre conjuntos de objetos y 4) validación de datos comunicados entre objetos o entre el usuario y la aplicación.

Los conceptos analizados en los párrafos que siguen pueden ser útiles en trabajos de análisis y diseño.

Herencia. La herencia es uno de los diferenciadores clave entre sistemas convencionales y orientados a objeto. Una subclase **Y** hereda todos los atributos y operaciones asociadas con su superclase **X**. Esto significa que todas las estructuras de datos y algoritmos originalmente diseñados e implementados para **X** están disponibles de inmediato para **Y**; no se necesita hacer más trabajo. La reutilización se logra directamente.

Cualquier cambio a los atributos u operaciones contenidos dentro de una superclase se hereda inmediatamente para todas las subclases. Por tanto, la jerarquía de clase se convierte en un mecanismo mediante el cual los cambios (en niveles superiores) pueden propagarse inmediatamente a través de un sistema.

Es importante observar que en cada nivel de la jerarquía de clase pueden agregarse nuevos atributos y operaciones a las que se heredaron de niveles superiores en la jerarquía. De hecho, siempre que se crea una nueva clase, tendrá algunas opciones:

- La clase puede diseñarse y construirse desde cero, es decir, no se usa herencia.
- La jerarquía de clase puede revisarse para determinar si una clase superior en la jerarquía contiene más de los atributos y operaciones requeridos. La nueva clase hereda de la clase superior y entonces pueden agregarse adiciones, según se requiera.
- La jerarquía de clase puede reestructurarse de modo que los atributos y operaciones requeridos pueden heredarse en la nueva clase.
- Las características de una clase existente pueden ser excesivas, y diferentes versiones de atributos u operaciones se implementan para la nueva clase.

Como todo concepto de diseño fundamental, la herencia puede proporcionar beneficios significativos para el diseño, pero si se usa de manera inadecuada,[3] puede complicar un diseño de manera innecesaria y conducir a software proclive a errores, que es difícil de mantener.

Mensajes. Las clases deben interactuar unas con otras para lograr las metas del diseño. Un mensaje estimula la ocurrencia de algunos comportamientos en el objeto receptor. El comportamiento se logra cuando una operación se ejecuta.

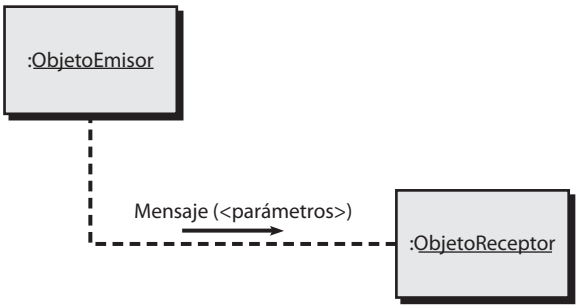
La interacción entre los objetos se ilustra de manera esquemática en la figura A2.2. Una operación dentro de **ObjetoEmisor** genera un mensaje de la forma *mensaje* (<parámetros>) donde los parámetros identifican **ObjetoReceptor** con el objeto que se va a estimular mediante el mensaje; la operación dentro de **ObjetoReceptor** consiste en recibir el mensaje y los ítems de datos que proporcionan información requerida para que la operación sea exitosa. La colaboración definida entre clases como parte del modelo de requerimientos proporciona lineamientos útiles en el diseño de mensajes.

Cox [Cox86] describe el intercambio entre clases en la forma siguiente:

> Un objeto [clase] se solicita para realizar una de sus operaciones al enviarle un mensaje que diga qué hacer al objeto. El receptor [objeto] responde al mensaje al elegir primero la operación que imple-

3 Por ejemplo, diseñar una subclase que herede atributos y operaciones de más de una superclase (en ocasiones llamada "herencia múltiple") es mal vista por la mayoría de los diseñadores.

FIGURA A2.2
Mensaje que pasa entre objetos



menta el nombre del mensaje, ejecutar esta operación y luego regresar el control al solicitante. Los mensajes ligan el sistema orientado a objeto. Los mensajes proporcionan comprensión acerca del comportamiento de objetos individuales y del sistema OO como un todo.

Polimorfismo. El *polimorfismo* es una característica que reduce enormemente el esfuerzo requerido para extender el diseño de un sistema orientado a objeto existente. Para entender el polimorfismo, considere una aplicación convencional que debe dibujar cuatro tipos diferentes de gráficas: gráficas de línea, gráficas de pastel, histogramas y diagramas Kiviat. De manera ideal, una vez recopilados los datos para un tipo particular de gráfica, ésta debe dibujarse a sí misma. Para lograrlo en una aplicación convencional (y mantener cohesión de módulo) sería necesario desarrollar módulos de dibujo para cada tipo de gráfica. Entonces, dentro del diseño, debería incrustarse lógica de control similar a la siguiente:

```
caso de gráficatipo:
   if gráficatipo = graficalinea then DibujarGráficaLinea (datos);
   if gráficatipo = graficapie then DibujarGráficaPie (datos);
   if gráficatipo = histograma then DibujarHisto (datos);
   if gráficatipo = kiviat then DibujarKiviat (datos);
termina caso;
```

Aunque este diseño es razonablemente directo, agregar nuevos tipos de gráfica podría ser truculento. Tendría que crearse un nuevo módulo de dibujo para cada tipo de gráfica y luego la lógica de control tendría que actualizarse para reflejar el nuevo tipo de gráfica.

Para resolver este problema en un sistema orientado a objeto, todas las gráficas se convierten en subclases de una clase general llamada **Gráfica**. Usando un concepto llamado *sobrecargar* [Tay90], cada subclase define una operación llamada *dibujar*. Un objeto puede enviar un mensaje *dibujar* a cualquiera de los objetos instanciados de cualquiera de las subclases. El objeto que recibe el mensaje invocará su propia operación *dibujo* para crear la gráfica adecuada. Por tanto, el diseño se reduce a

```
dibujar <gráficatipo>
```

Cuando un nuevo tipo de gráfica se agrega al sistema, se crea una subclase con su propia operación *dibujar*. Pero no se requieren cambios dentro de algún objeto que quiera una gráfica dibujada porque el mensaje **dibujar <gráficatipo>** sigue invariable. Para resumir, el polimorfismo permite que algunas operaciones diferentes tengan el mismo nombre. Esto a su vez desacopla objetos uno de otro, lo que los hace más independientes.

Clase de diseño. El modelo de requerimientos define un conjunto completo de clases de análisis. Cada una describe algún elemento del dominio de problema, y se enfoca en aspectos del

problema que son visibles al usuario o cliente. El nivel de abstracción de cualquier clase de análisis es relativamente alto.

Conforme evoluciona el modelo de diseño, el equipo de software debe definir un conjunto de *clase de diseño* que 1) refinen las clases de análisis, proporcionando detalle de diseño que permitirá la implementación de las clases y 2) crear un nuevo conjunto de clases de diseño que implementen una infraestructura de software que dé apoyo a la solución empresarial. Se sugieren cinco tipos diferentes de clases de diseño, cada una como representación de una capa diferente de la arquitectura de diseño [Amb01]:

- *Clases de interfaz de usuario:* definen todas las abstracciones que se necesitan para la interacción humano-computadora (IHC).
- *Clases de dominio empresarial:* con frecuencia son refinamientos de las clases de análisis definidas anteriormente. Las clases identifican los atributos y operaciones (métodos) que se requieren para implementar algún elemento del dominio empresarial.
- *Clases de proceso:* implementan abstracciones empresariales de nivel inferior requeridas para gestionar por completo las clases de dominio empresarial.
- *Clases persistentes:* representan almacenes de datos (por ejemplo, una base de datos) que persistirá más allá de la ejecución del software.
- *Clases de sistema:* implementan gestión de software y funciones de control que permiten al sistema operar y comunicarse dentro de su entorno de computación y con el mundo exterior.

Conforme evoluciona el diseño arquitectónico, el equipo de software debe desarrollar un conjunto completo de atributos y operaciones para cada clase de diseño. El nivel de abstracción se reduce conforme cada clase de análisis se transforma en una representación de diseño. Es decir, las clases de análisis representan objetos (y métodos asociados que se les aplican), usando la jerga del dominio empresarial. Las clases de diseño presentan detalle significativamente más técnico como una guía para la implementación.

Arlow y Neustadt [Arl02] sugieren que cada clase de diseño se revise para garantizar que está "bien formada". Definen cuatro características de una clase de diseño bien formada:

Completa y suficiente. Una clase de diseño debe ser el encapsulamiento completo de todos los atributos y métodos que razonablemente pueda esperarse que existan para la clase (con base en una interpretación enterada del nombre de la clase). Por ejemplo, la clase **Escena** definida para software de edición de video es completa sólo si contiene todos los atributos y métodos que razonablemente puedan asociarse con la creación de una escena de video. La suficiencia garantiza que la clase de diseño contiene solamente aquellos métodos que son suficientes para lograr la intención de la clase, ni más y ni menos.

Primitividad. Los métodos asociados con una clase de diseño deben enfocarse en lograr una función específica para la clase. Una vez implementada la función con un método, la clase no debe proporcionar otra vía para lograr lo mismo. Por ejemplo, la clase **VideoClip** del software de edición puede tener atributos **punto-inicial** y **punto-final** para indicar los puntos de inicio y finalización del clip (observe que el video en bruto cargado en el sistema puede ser más largo que el clip que se usa). Los métodos, *setStartPoint()* y *setEndPoint()* proporcionan el único medio para establecer puntos de inicio y finalización para el clip.

Alta cohesión. Una clase de diseño cohesiva tiene un solo propósito: tiene un pequeño conjunto enfocado en responsabilidades y aplica atributos y métodos decisivos para implementar dichas responsabilidades. Por ejemplo, la clase **VideoClip** del software de edición de video puede contener un conjunto de métodos para editar el clip de video. En tanto cada

método se enfoque exclusivamente en atributos asociados con el videoclip, la cohesión se mantiene.

Low coupling. Dentro del modelo de diseño es necesario que las clases de diseño colaboren unas con otras. No obstante, la colaboración debe mantenerse en un mínimo aceptable. Si un modelo de diseño está enormemente acoplado (todas las clases de diseño colaboran con todas las otras clases de diseño), el sistema es difícil de implementar, probar y mantener con el tiempo. En general, las clases de diseño dentro de un subsistema deben tener solamente conocimiento limitado de otras clases. Esta restricción, llamada *ley de Démeter* [Lie03], sugiere que un método sólo debe enviar mensajes a métodos en clases vecinas.[4]

Lecturas y fuentes de información adicionales

Durantes las tres décadas pasadas se escribieron cientos de libros acerca de programación, análisis y diseño orientado a objeto. Weisfeld (*The Object-Oriented Thought Process*, 2a. ed., Sams Publishing, 2003) presenta un valioso tratamiento de los conceptos y principios generales OO. McLaughlin *et al.* (*Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D*, O'Reilly Media, Inc., 2006) proporcionan un tratamiento accesible y disfrutable del análisis OO y de los enfoques al diseño. Un tratamiento más profundo del análisis y diseño OO se presenta en Booch *et al.* (*Objetct-Oriented Analysis and Design with Applications*, 3a. ed., Addison-Wesley, 2007). Wu (*An Introduction to Object-Oriented Programming with Java*, McGraw-Hill, 2005) escribió un libro exhaustivo acerca de programación OO que es clásico de decenas de escritos para muchos diferentes lenguajes de programación.

En internet está disponible una gran variedad de fuentes de información acerca de tecnologías orientadas a objeto. Una lista actualizada de referencias en la World Wide Web puede encontrarse bajo "análisis" y "diseño" en el sitio del libro: **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**

4 Una manera menos formal de plantear la ley de Démeter es, "Cada unidad sólo debe hablar con sus amigos; no hablar con extraños".

Referencias

[Abb83] Abbott, R., "Program Design by Informal English Descriptions", *CACM*, vol. 26, núm. 11, noviembre 1983, pp. 892-894.

[ACM98] ACM/IEEE-CS Joint Task Force, *Software Engineering Code of Ethics and Professional Practice*, 1998, disponible en www.acm.org/serving/se/code.htm.

[Ada93] Adams, D., *Mostly Harmless*, Macmillan, 1993.

[AFC88] *Software Risk Abatement*, AFCS/AFLC Pamphlet 800-845, U.S. Air Force, septiembre 30, 1988.

[Agi03] The Agile Alliance Home Page, www.agilealliance.org/home.

[Air99] Airlie Council, "Performance Based Management: The Program Manager's Guide Based on the 16-Point Plan and Related Metrics", Draft Report, marzo 8, 1999.

[Aka04] Akao, Y., *Quality Function Deployment*, Productivity Press, 2004.

[Ale77] Alexander, C., *A Pattern Language*, Oxford University Press, 1977.

[Ale79] Alexander, C., *The Timeless Way of Building*, Oxford University Press, 1979.

[Amb95] Ambler, S., "Using Use-Cases", *Software Development*, julio 1995, pp. 53-61.

[Amb98] Ambler, S., *Process Patterns: Building Large-Scale Systems Using Object Technology*, Cambridge University Press/SIGS Books, 1998.

[Amb01] Ambler, S., *The Object Primer*, 2a. ed., Cambridge University Press, 2001.

[Amb02a] Ambler, S., "What Is Agile Modeling (AM)?" 2002, www.agilemodeling.com/index.htm.

[Amb02b] Ambler, S. y R. Jeffries, *Agile Modeling*, Wiley, 2002.

[Amb02c] Ambler, S., "UML Component Diagramming Guidelines", disponible en www. modelingstyle.info/, 2002.

[Amb04] Ambler, S., "*Examining the Cost of Change Curve*", en *The Object Primer*, 3a. ed., Cambridge University Press, 2004.

[Amb06] Ambler, S., "The Agile Unified Process (AUP)", 2006, disponible en www.ambysoft.com/unifiedprocess/agileUP.html.

[And06] Andrews, M. y J. Whittaker, *How to Break Web Software: Functional and Security Testing of Web Applications and Web Services*, Addison-Wesley, 2006.

[ANS87] ANSI/ASQC A3-1987, *Quality Systems Terminology*, 1987.

[Ant06] Anton, D. y C. Anton, *ISO 9001 Survival Guide*, 3a. ed., AEM Consulting Group, 2006.

[AOS07] AOSD.net (Aspect-Oriented Software Development), glosario, disponible en http://aosd.net/wiki/index.php?title=Glossary.

[App00] Appleton, B., "Patterns and Software: Essential Concepts and Terminology", febrero 2000, disponible en www.cmcrossroads.com/bradapp/docs/patterns-intro.html.

[App08] Apple Computer, *Accessibility*, 2008, disponible en www.apple.com/disability/.

[Arl02] Arlow, J. y I. Neustadt, *UML and the Unified Process*, Addison-Wesley, 2002.

[Arn89] Arnold, R. S., "Software Restructuring", *Proc. IEEE*, vol. 77, núm. 4, abril 1989, pp. 607-617.

[Art97] Arthur, L. J., "Quantum Improvements in Software System Quality", *CACM*, vol. 40, núm. 6, junio 1997, pp. 47-52.

[Ast04] Astels, D., *Test Driven Development: A Practical Guide*, Prentice Hall, 2004.

[Ave04] Aversan, L., *et al.*, "Managing Coordination and Cooperation in Distributed Software Processes: The GENESIS Environment", *Software Process Improvement and Practice*, vol. 9, Wiley Interscience, 2004, pp. 239-263.

[Baa07] de Baar, B., "Project Risk Checklist", 2007, disponible en www.softwareprojects.org/ project_riskmanagement_starting62.htm.

[Bab86] Babich, W. A., *Software Configuration Management*, Addison-Wesley, 1986.

[Bac97] Bach, J., "Good Enough Quality: Beyond the Buzzword", *IEEE Computer*, vol. 30, núm. 8, agosto 1997, pp. 96-98.

[Bac98] Bach, J., "The Highs and Lows of Change Control", *Computer*, vol. 31, núm. 8, agosto 1998, pp. 113-115.

[Bae98] Baetjer, Jr., H., *Software as Capital*, IEEE Computer Society Press, 1998, p. 85.

[Bak72] Baker, F. T., "Chief Programmer Team Management of Production Programming", *IBM Systems Journal.*, vol. 11, núm. 1, 1972, pp. 56-73.

[Ban06] Baniassad, E., *et al.*, "Discovering Early Aspects", *IEEE Software*, vol. 23, núm. 1, enero-febrero, 2006, pp. 61-69.

[Bar06] Baresi, L., E. DiNitto y C. Ghezzi, "Toward Open-World Software: Issues and Challenges", *IEEE Computer*, vol. 39, núm. 10, octubre 2006, pp. 36-43.

[Bas84] Basili, V. R. y D. M. Weiss, "A Methodology for Collecting Valid Software Engineering Data", *IEEE Trans. Software Engineering*, vol. SE-10, 1984, pp. 728-738.

[Bas03] Bass, L., P. Clements y R. Kazman, *Software Architecture in Practice*, 2a. ed., Addison-Wesley, 2003.
[Bec00] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 1999.
[Bec01a] Beck, K., *et al.*, "Manifesto for Agile Software Development", www.agilemanifesto.org/.
[Bec04a] Beck, K., *Extreme Programming Explained: Embrace Change*, 2a. ed., Addison-Wesley, 2004.
[Bec04b] Beck, K., *Test-Driven Development: By Example*, 2a. ed., Addison-Wesley, 2002.
[Bee99] Beedle, M., *et al.*, "SCRUM: An Extension Pattern Language for Hyperproductive Software Development", incluido en: *Pattern Languages of Program Design 4*, Addison-Wesley Longman, Reading MA, 1999, descargable de http://jeffsutherland.com/scrum/ scrum_plop.pdf.
[Bei84] Beizer, B., *Software System Testing and Quality Assurance*, Van Nostrand-Reinhold, 1984.
[Bei90] Beizer, B., *Software Testing Techniques*, 2a. ed., Van Nostrand-Reinhold, 1990.
[Bei95] Beizer, B., *Black-Box Testing*, Wiley, 1995.
[Bel81] Belady, L., prólogo de *Software Design: Methods and Techniques* (L. J. Peters, autor), Yourdon Press, 1981.
[Bel95] Bellinzona R., M. G. Gugini y B. Pernici, "Reusing Specifications in OO Applications", *IEEE Software*, marzo 1995, pp. 65-75.
[Ben99] Bentley, J., *Programming Pearls*, 2a. ed., Addison-Wesley, 1999.
[Ben00] Bennatan, E. M., *Software Project Management: A Practitioner's Approach*, 3a. ed., McGraw-Hill, 2000.
[Ben02] Bennett, S., S. McRobb y R. Farmer, *Object-Oriented Analysis and Design*, 2a. ed., McGraw-Hill, 2002.
[Ber80] Bersoff, E. H., V. D. Henderson y S. G. Siegel, *Software Configuration Management*, Prentice Hall, 1980.
[Ber93] Berard, E., *Essays on Object-Oriented Software Engineering*, vol. 1, Addison-Wesley, 1993.
[Bes04] Bessin, J., "The Business Value of Quality", IBM developerWorks, junio 15, 2004, disponible en www-128.ibm.com/developerworks/rational/library/4995.html.
[Bha06] Bhat, J., M. Gupta y S. Murthy, "Lessons from Offshore Outsourcing", *IEEE Software*, vol. 23, núm. 5, septiembre-octubre 2006.
[Bie94] Bieman, J. M. y L. M. Ott, "Measuring Functional Cohesion", *IEEE Trans. Software Engineering*, vol. SE-20, núm. 8, agosto 1994, pp. 308-320.
[Bin93] Binder, R., "Design for Reuse Is for Real", *American Programmer*, vol. 6, núm. 8, agosto 1993, pp. 30-37.
[Bin94a] Binder, R., "Testing Object-Oriented Systems: A Status Report", *American Programmer*, vol. 7, núm. 4, abril 1994, pp. 23-28.
[Bin94b] Binder, R. V., "Object-Oriented Software Testing", *Communications of the ACM*, vol. 37, núm. 9, septiembre 1994, p. 29.
[Bin99] Binder, R., *Testing Object-Oriented Systems: Models, Patterns, and Tools*, Addison-Wesley, 1999.
[Bir98] Biró, M. y T. Remzsö, "Business Motivations for Software Process Improvement", ERCIM News, núm. 32, enero 1998, disponible en www.ercim.org/publication/Ercim_News/enw32/biro.html.
[Boe81] Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.
[Boe88] Boehm, B., "A Spiral Model for Software Development and Enhancement", *Computer*, vol. 21, núm. 5, mayo 1988, pp. 61-72.
[Boe89] Boehm, B. W., *Software Risk Management*, IEEE Computer Society Press, 1989.
[Boe96] Boehm, B., "Anchoring the Software Process", *IEEE Software*, vol. 13, núm. 4, julio 1996, pp. 73-82.
[Boe98] Boehm, B., "Using the WINWIN Spiral Model: A Case Study", *Computer*, vol. 31, núm. 7, julio 1998, pp. 33-44.
[Boe00] Boehm, B., *et al.*, *Software Cost Estimation in COCOMO II*, Prentice Hall, 2000.
[Boe01a] Boehm, B., "The Spiral Model as a Tool for Evolutionary Software Acquisition", *CrossTalk*, mayo 2001, disponible en www.stsc.hill.af.mil/crosstalk/2001/05/boehm.html.
[Boe01b] Boehm, B. y V. Basili, "Software Defect Reduction Top 10 List", *IEEE Computer*, vol. 34, núm. 1, enero 2001, pp. 135-137.
[Boe08] Boehm, B., "Making a Difference in the Software Century", *IEEE Computer*, vol. 41, núm. 3, marzo 2008, pp. 32-38.
[Boh66] Bohm, C. y G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *CACM*, vol. 9, núm. 5, mayo 1966, pp. 366-371.
[Boh00] Bohl, M. y M. Rynn, *Tools for Structured Design: An Introduction to Programming Logic*, 5a. ed., Prentice Hall, 2000.
[Boi04] Boiko, B., *Content Management Bible*, 2a. ed., Wiley, 2004.
[Bol02] Boldyreff, C., *et al.*, "Environments to Support Collaborative Software Engineering", 2002, descargable de www.cs.put.poznan.pl/dweiss/site/publications/download/csmre-paper.pdf.
[Boo94] Booch, G., *Object-Oriented Analysis and Design*, 2a. ed., Benjamin Cummings, 1994.
[Boo05] Booch, G., J. Rumbaugh y I. Jacobsen, *The Unified Modeling Language User Guide*. 2a. ed., Addison-Wesley, 2005.
[Boo06] Bootstrap-institute.com, 2006, www.cse.dcu.ie/espinode/directory/directory.html.
[Boo08] Booch, G., *Handbook of Software Architecture*, 2008, disponible en www.booch.com/architecture/systems.jsp.

[Bor01] Borchers, J., *A Pattern Approach to Interaction Design*, Wiley, 2001.
[Bos00] Bosch, J., *Design & Use of Software Architectures*, Addison-Wesley, 2000.
[Bra85] Bradley, J. H., "The Science and Art of Debugging", *Computerworld*, agosto 19, 1985, pp. 35-38.
[Bra94] Bradac, M., D. Perry y L. Votta, "Prototyping a Process Monitoring Experiment", *IEEE Trans. Software Engineering*, vol. 20, núm. 10, octubre 1994, pp. 774-784.
[Bre02] Breen, P., "Exposing the Fallacy of 'Good Enough' Software", informit.com, febrero 1, 2002, disponible en www.informit.com/articles/article.asp?p=25141&rl=1.
[Bro95] Brooks, F., *The Mythical Man-Month*, Silver Anniversary edition, Addison-Wesley, 1995.
[Bro96] Brown, A. W. y K. C. Wallnau, "Engineering of Component Based Systems", *Component-Based Software Engineering*, IEEE Computer Society Press, 1996, pp. 7-15.
[Bro01] Brown, B., *Oracle9i Web Development*, 2a. ed., McGraw-Hill, 2001.
[Bro03] Brooks, F, "Three Great Challenges for Half-Century-Old Computer Science", *JACM*, vol. 50, núm. 1, enero 2003, pp. 25-26.
[Bro06] Broy, M., "The 'Grand Challenge' in Informatics: Engineering Software Intensive Systems", *IEEE Computer*, vol. 39, núm. 10, octubre 2006, pp. 72-80.
[Buc99] Bucanac, C., "The V-Model", University of Karlskrona/Ronneby, enero 1999, descargable de www.bucanac.com/documents/The_V-Model.pdf.
[Bud96] Budd, T., *An Introduction to Object-Oriented Programming*, 2a. ed., Addison-Wesley, 1996.
[Bus96] Buschmann, F., *et al.*, *Pattern-Oriented Software Architecture*, Wiley, 1996.
[Bus07] Buschmann, F., *et al.*, *Pattern-Oriented Software Architecture, A System of Patterns*, Wiley, 2007.
[Cac02] Cachero, C., *et al.*, "Conceptual Navigation Analysis: A Device and Platform Independent Navigation Specification", *Proc. 2nd Intl. Workshop on Web-Oriented Technology*, junio 2002, descargable de www.dsic.upv.es/~west/iwwost02/papers/cachero.pdf.
[Cai03] Caine, Frarber y Gordon, Inc., *PDL/81*, 2003, disponible en www.cfg.com/pdl81/lpd.html.
[Car90] Card, D. N. y R. L. Glass, *Measuring Software Design Quality*, Prentice Hall, 1990.
[Cas89] Cashman, M., "Object Oriented Domain Analysis", *ACM Software Engineering Notes*, vol. 14, núm. 6, octubre 1989, p. 67.
[Cav78] Cavano, J. P. y J. A. McCall, "A Framework for the Measurement of Software Quality", *Proc. ACM Software Quality Assurance Workshop*, noviembre 1978, pp. 133-139.
[CCS02] CS3 Consulting Services, 2002, www.cs3inc.com/DSDM.htm.
[Cec06] Cechich, A., *et al.*, "Trends on COTS Component Identification", *Proc. Fifth Intl. Conf. on COTS-Based Software Systems*, IEEE, 2006.
[Cha89] Charette, R. N., *Software Engineering Risk Analysis and Management*, McGraw-Hill/ Intertext, 1989.
[Cha92] Charette, R. N., "Building Bridges over Intelligent Rivers", *American Programmer*, vol. 5, núm. 7, septiembre 1992, pp. 2-9.
[Cha93] de Champeaux, D., D. Lea y P. Faure, *Object-Oriented System Development*, Addison-Wesley, 1993.
[Cha03] Chakravarti, A., "Online Software Design Pattern Links", 2003, disponible en www.anupriyo.com/oopfm.shtml.
[Che77] Chen, P., *The Entity-Relationship Approach to Logical Database Design*, QED Information Systems, 1977.
[Chi94] Chidamber, S. R. y C. F. Kemerer, "A Metrics Suite for Object-Oriented Design", *IEEE Trans. Software Engineering*, vol. SE-20, núm. 6, junio 1994, pp. 476-493.
[Cho89] Choi, S. C. y W. Scacchi, "Assuring the Correctness of a Configured Software Description", *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, octubre 1989, pp. 66-75.
[Chu95] Churcher, N. I. y M. J. Shepperd, "Towards a Conceptual Framework for Object-Oriented Metrics", *ACM Software Engineering Notes*, vol. 20, núm. 2, abril 1995, pp. 69-76.
[Cig07] Cigital, Inc., "Case Study: Finding Defects Earlier Yields Enormous Savings", 2007, disponible en www.cigital.com/solutions/roi-cs2.php.
[Cla05] Clark, S. y E. Baniasaad, *Aspect-Oriented Analysis and Design*, Addison-Wesley, 2005.
[Cle95] Clements, P., "From Subroutines to Subsystems: Component Based Software Development", *American Programmer*, vol. 8, núm. 11, noviembre 1995.
[Cle03] Clements, P., R. Kazman y M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison-Wesley, 2003.
[Cle06] Clemmons, R., "Project Estimation with Use Case Points", *CrossTalk*, febrero 2006, p. 18-222, descargable de www.stsc.hill.af.mil/crosstalk/2006/02/0602Clemmons.pdf.
[CMM07] *Capability Maturity Model Integration (CMMI)*, Software Engineering Institute, 2007, disponible en www.sei.cmu.edu/cmmi/.
[CMM08] *People Capability Maturity Model Integration (People CMM)*, Software Engineering Institute, 2008, disponible en www.sei.cmu.edu/cmm-p/.
[Coa91] Coad, P. y E. Yourdon, *Object-Oriented Analysis*, 2a. ed., Prentice Hall, 1991.
[Coa99] Coad, P., E. Lefebvre y J. DeLuca, *Java Modeling in Color with UML*, Prentice Hall, 1999.
[Coc01a] Cockburn, A. y J. Highsmith, "Agile Software Development: The People Factor", *IEEE Computer*, vol. 34, núm. 11, noviembre 2001, pp. 131-133.
[Coc01b] Cockburn, A., *Writing Effective Use-Cases*, Addison-Wesley, 2001.
[Coc02] Cockburn, A., *Agile Software Development*, Addison-Wesley, 2002.

[Coc04] Cockburn, A., "What the Agile Toolbox Contains", *CrossTalk*, noviembre 2004, disponible en www.stsc.hill.af.mil/crosstalk/2004/11/0411Cockburn.html.
[Coc05] Cockburn, A., *Crystal Clear*, Addison-Wesley, 2005.
[Con96] Conradi, R., "Software Process Improvement: Why We Need SPIQ", NTNU, octubre 1996, descargable de www.idi.ntnu.no/grupper/su/publ/pdf/nik96-spiq.pdf.
[Con02] Conradi, R. y A. Fuggetta, "Improving Software Process Improvement", *IEEE Software*, julio-agosto 2002, pp. 2-9, descargable de http://citeseer.ist.psu.edu/conradi02improving.html.
[Con93] Constantine, L., "Work Organization: Paradigms for Project Management and Organization", *CACM*, vol. 36, núm. 10, octubre 1993, pp. 34-43.
[Con95] Constantine, L, "What DO Users Want? Engineering Usability in Software", *Windows Tech Journal*, diciembre 1995, disponible en www.forUse.com.
[Con03] Constantine, L. y L. Lockwood, *Software for Use*, Addison-Wesley, 1999; vea también www.foruse.com/.
[Cop05] Coplien, J., "Software Patterns", 2005, disponible en http://hillside.net/patterns/definition.html.
[Cor98] Corfman, R., "An Overview of Patterns", en *The Patterns Handbook*, SIGS Books, 1998.
[Cou00] Coulouris, G., J. Dollimore y T. Kindberg, *Distributed Systems: Concepts and Design*, 3a. ed., Addison-Wesley, 2000.
[Cox86] Cox, Brad, *Object-Oriented Programming*, Addison-Wesley, 1986.
[Cri92] Christel, M. G. y K. C. Kang, "Issues in Requirements Elicitation", Software Engineering Institute, CMU/SEI-92-TR-12 7, septiembre 1992.
[Cro79] Crosby, P., *Quality Is Free*, McGraw-Hill, 1979.
[Cro07] Cross, M. y M. Fisher, *Developer's Guide to Web Application Security*, Syngress Publishing, 2007.
[Cur86] Curritt, P. A., M. Dyer y H. D. Mills, "Certifying the Reliability of Software", *IEEE Trans, Software Engineering*, vol. SE-12, núm. 1, enero 1994.
[Cur88] Curtis, B., *et al.*, "A Field Study of the Software Design Process for Large Systems", *IEEE Trans. Software Engineering*, vol. SE-31, núm. 11, noviembre 1988, pp. 1268-1287.
[Cur01] Curtis, B., W. Hefley y S. Miller, *People Capability Maturity Model*, Addison-Wesley, 2001.
[CVS07] Concurrent Versions System, Ximbiot, http://ximbiot.com/cvs/wiki/index.php?title= Main_Page, 2007.
[DAC03} "An Overview of Model-Based Testing for Software", Data and Analysis Center for Software, CR/TA 12, junio 2003, descargable de www.goldpractices.com/dwnload/ practice/pdf/Model_Based_Testing.pdf.
[Dah72] Dahl, O., E. Dijkstra y C. Hoare, *Structured Programming*, Academic Press, 1972.
[Dar91] Dart, S., "Concepts in Configuration Management Systems", *Proc. Third International Workshop on Software Configuration Management*, ACM SIGSOFT, 1991, descargable de www.sei.cmu.edu/legacy/scm/abstracts/abscm_concepts.html.
[Dar99] Dart, S., "Change Management: Containing the Web Crisis", *Proc. Software Configuration Management Symposium*, Toulouse, Francia, 1999, disponible en www.perforce.com/perforce/ conf99/dart.html.
[Dar01] Dart, S., *Spectrum of Functionality in Configuration Management Systems*, Software Engineering Institute, 2001, disponible en www.sei.cmu.edu/legacy/scm/tech_rep/TR11_90/ TOC_TR11_90.html.
[Das05] Dasari, R., "Lean Software Development", a white paper, descargable de www .projectperfect.com.au/downloads/Info/info_lean_development.pdf, 2005.
[Dav90] Davenport, T. H. y J. E. Young, "The New Industrial Engineering: Information Technology and Business Process Redesign", *Sloan Management Review*, verano 1990, pp. 11-27.
[Dav93] Davis, A., *et al.*, "Identifying and Measuring Quality in a Software Requirements Specification", *Proc. First Intl. Software Metrics Symposium*, IEEE, Baltimore, MD, mayo 1993, pp. 141-152.
[Dav95a] Davis, M., "Process and Product: Dichotomy or Duality", *Software Engineering Notes*, ACM Press, vol. 20, núm. 2, abril, 1995, pp. 17-18.
[Dav95b] Davis, A., *201 Principles of Software Development*, McGraw-Hill, 1995.
[Day99] Dayani-Fard, H., *et al.*, "Legacy Software Systems: Issues, Progress, and Challenges", IBM Technical Report: TR-74.165-k, abril 1999, disponible en www.cas.ibm.com/toronto/ publications/TR-74.165/k/legacy.html.
[Dem86] Deming, W. E., *Out of the Crisis*, MIT Press, 1986.
[DeM79] DeMarco, T., *Structured Analysis and System Specification*, Prentice Hall, 1979.
[DeM95] DeMarco, T., *Why Does Software Cost So Much?* Dorset House, 1995.
[DeM95a] DeMarco, T., "Lean and Mean", *IEEE Software*, noviembre 1995, pp. 101-102.
[DeM98] DeMarco, T. y T. Lister, *Peopleware*, 2a. ed., Dorset House, 1998.
[DeM02] DeMarco, T. y B. Boehm, "The Agile Methods Fray", IEEE Computer, vol. 35, núm. 6, junio 2002, pp. 90-92.
[Den73] Dennis, J., "Modularity", en *Advanced Course on Software Engineering* (F. L. Bauer, ed.), Springer-Verlag, 1973, pp. 128-182.
[Dev01] Devedzik, V., "Software Patterns", en *Handbook of Software Engineering and Knowledge Engineering*, World Scientific Publishing Co., 2001.
[Dha95] Dhama, H., "Quantitative Metrics for Cohesion and Coupling in Software", *Journal of Systems and Software*, vol. 29, núm. 4, abril 1995.

[Dij65] Dijkstra, E., "Programming Considered as a Human Activity", en *Proc. 1965 IFIP Congress*, North-Holland Publishing Co., 1965.
[Dij72] Dijkstra, E., "The Humble Programmer", 1972 ACM Turing Award Lecture, *CACM*, vol. 15, núm. 10, octubre 1972, pp. 859-866.
[Dij76a] Dijkstra, E., "Structured Programming", en *Software Engineering, Concepts and Techniques*, (J. Buxton *et al.*, eds.), Van Nostrand-Reinhold, 1976.
[Dij76b] Dijkstra, E., *A Discipline of Programming*, Prentice Hall, 1976.
[Dij82] Dijksta, E., "On the Role of Scientific Thought", *Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, 1982.
[Dix99] Dix, A., "Design of User Interfaces for the Web", *Proc. User Interfaces to Data Systems Conference*, septiembre 1999, descargable de www.comp.lancs.ac.uk/computing/ users/dixa/topics/webarch/.
[Dob04] Dobb, F., *ISO 9001:2000 Quality Registration Step-by-Step*, 3a. ed., Butterworth- Heinemann, 2004.
[Don99] Donahue, G., S. Weinschenck y J. Nowicki, "Usability Is Good Business", Compuware Corp., julio 1999, disponible en www.compuware.com.
[Dre99] Dreilinger, S., "CVS Version Control for Web Site Projects", 1999, disponible en www.durak.org/cvswebsites/howto-cvs/howto-cvs.html.
[Dru75] Drucker, P., *Management*, W. H. Heinemann, 1975.
[Duc01] Ducatel, K., *et al.*, *Scenarios for Ambient Intelligence in 2010*, ISTAG-European Commission, 2001, descargable de ftp://ftp.cordis.europa.eu/pub/ist/docs/istagscenarios2010.pdf.
[Dun82] Dunn, R. y R. Ullman, *Quality Assurance for Computer Software*, McGraw-Hill, 1982.
[Dun01] Dunaway, D. y S. Masters, *CMM-Based Appraisal for Internal Process Improvement (CBA IPI Version 1,2 Method Description)*, Software Engineering Institute, 2001, descargable de www.sei.cmu.edu/publications/documents/01.reports/01tr033.html.
[Dun02] Dunn, W., *Practical Design of Safety-Critical Computer Systems*, William Dunn, 2002.
[Duy02] VanDuyne, D., J. Landay y J. Hong, *The Design of Sites*, Addison-Wesley, 2002.
[Dye92] Dyer, M., *The Cleanroom Approach to Quality Software Development*, Wiley, 1992.
[Edg95] Edgemon, J., "Right Stuff: How to Recognize It When Selecting a Project Manager", *Application Development Trends*, vol. 2, núm. 5, mayo 1995, pp. 37-42.
[Eji91] Ejiogu, L., *Software Engineering with Formal Metrics*, QED Publishing, 1991.
[Elr01] Elrad, T., R. Filman y A. Bader (eds.), "Aspect Oriented Programming", *Comm. ACM*, vol. 44, núm. 10, octubre 2001, número especial.
[Eri05] Ericson, C., *Hazard Analysis Techniques for System Safety*, Wiley-Interscience, 2005.
[Eri08] Erickson, T., *The Interaction Design Patterns Page*, mayo 2008, disponible en www.visi .com/~snowfall/InteractionPatterns.html.
[Eva04] Evans, E., *Domain Driven Design*, Addison-Wesley, 2003.
[Fag86] Fagan, M., "Advances in Software Inspections", *IEEE Trans. Software Engineering*, vol. 12, núm. 6, julio 1986.
[Fel89] Felican, L. y G. Zalateu, "Validating Halstead's Theory for Pascal Programs", *IEEE Trans. Software Engineering*, vol. SE-15, núm. 2, diciembre 1989, pp. 1630-1632.
[Fel07] Feller, J., *et al.* (eds.), *Perspectives on Free and Open Source Software*, The MIT Press, 2007.
[Fen91] Fenton, N., *Software Metrics*, Chapman y Hall, 1991.
[Fen94] Fenton, N., "Software Measurement: A Necessary Scientific Basis", *IEEE Trans. Software Engineering*, vol. SE-20, núm. 3, marzo 1994, pp. 199-206.
[Fer97] Ferguson, P., *et al.*, "Results of Applying the Personal Software Process", *IEEE Computer*, vol. 30, núm. 5, mayo 1997, pp. 24-31.
[Fer98] Ferdinandi, P. L., "Facilitating Communication", *IEEE Software*, septiembre 1998, pp. 92-96.
[Fer00] Fernandez, E. B. y X. Yuan, "Semantic Analysis Patterns", *Proceedings of the 19th Int. Conf. on Conceptual Modeling, ER2000*, Lecture Notes in Computer Science 1920, Springer, 2000, pp. 183-195. También disponible en www.cse.fau.edu/~ed/SAPpaper2.pdf.
[Fir93] Firesmith, D. G., *Object-Oriented Requirements Analysis and Logical Design*, Wiley, 1993.
[Fis06] Fisher, R. y D. Shapiro, *Beyond Reason: Using Emotions as You Negotiate*, Penguin, 2006.
[Fit54] Fitts, P., "The Information Capacity of the Human Motor System in Controlling the Amplitude of Movement", *Journal of Experimental Psychology*, vol. 47, 1954, pp. 381-391.
[Fle98] Fleming, Q. W. y J. M. Koppelman, "Earned Value Project Management", *CrossTalk*, vol. 11, núm. 7, julio 1998, p. 19.
[Fos06] Foster, E., "Quality Culprits", InfoWorld Grip Line Weblog, mayo 2, 2006, disponible en http://weblog.infoworld.com/gripeline/2006/05/02_a395.html.
[Fow97] Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.
[Fow00] Fowler, M., *et al.*, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2000.
[Fow01] Fowler, M. y J. Highsmith, "The Agile Manifesto", *Software Development Magazine*, agosto 2001, www.sdmagazine.com/documents/s=844/sdm0108a/0108a.htm.
[Fow02] Fowler. M., "The New Methodology", junio 2002, www.martinfowler.com/articles/newMethodology.html#N8B.
[Fow03] Fowler, M., *et al.*, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[Fow04] Fowler, M., *UML Distilled*, 3a. ed., Addison-Wesley, 2004.
[Fra93] Frankl, P. G. y S. Weiss, "An Experimental Comparison of the Effectiveness of Branch Testing and Data Flow", *IEEE Trans. Software Engineering*, vol. SE-19, núm. 8, agosto 1993, pp. 770-787.
[Fra03] Francois, A., "Software Architecture for Immersipresence", IMSC Technical Report IMSC-03-001, University of Southern California, diciembre 2003, disponible en http://iris.usc.edu/ ~afrancoi/pdf/sai-tr.pdf.
[Fre80] Freeman, P., "The Context of Design", en *Software Design Techniques*, 3a. ed. (P. Freeman y A. Wasserman, eds.), IEEE Computer Society Press, 1980, pp. 2-4.
[Fre90] Freedman, D. P. y G. M. Weinberg, *Handbook of Walkthroughs, Inspections and Technical Reviews*, 3a. ed., Dorset House, 1990.
[Gag04] Gage, D. y J. McCormick, "We Did Nothing Wrong", *Baseline Magazine*, marzo 4, 2004, disponible en www.baselinemag.com/article2/0,1397,1544403,00.asp.
[Gai95] Gaines, B., "Modeling and Forecasting the Information Sciences", Technical Report, University of Calgary, Calgary, Alberta, septiembre 1995.
[Gam95] Gamma, E., *et al.*, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[Gar84] Garvin, D., "What Does 'Product Quality' Really Mean?" *Sloan Management Review*, otoño 1984, pp. 25-45.
[Gar87] Garvin D., *"Competing on the Eight Dimensions of Quality", Harvard Business Review*, noviembre 1987, pp. 101-109. Resumen disponible en www.acm.org/crossroads/ xrds6-4/software.html.
[Gar95] Garlan, D. y M. Shaw, "An Introduction to Software Architecture", *Advances in Software Engineering and Knowledge Engineering*, vol. I (V. Ambriola and G. Tortora, eds.), World Scientific Publishing Company, 1995.
[Gar08] GartnerGroup, "Understanding Hype Cycles", 2008, disponible en www.gartner.com/ pages/story.php.id.8795.s.8.jsp.
[Gau89] Gause, D. C. y G. M. Weinberg, *Exploring Requirements: Quality Before Design*, Dorset House, 1989.
[Gey01] Geyer-Schulz, A. y M. Hahsler, "Software Engineering with Analysis Patterns", Technical Report 01/2001, Institut für Informationsverarbeitung und -wirtschaft, Wirschaftsuniversität Wien, noviembre 2001, descargable de wwwai.wu-wien.ac.at/~hahsler/ research/virlib_working2001/virlib/.
[Gil88] Gilb, T., *Principles of Software Project Management*, Addison-Wesley, 1988.
[Gil95] Gilb, T., "What We Fail to Do in Our Current Testing Culture", *Testing Techniques Newsletter* (online edition, ttn@soft.com), Software Research, enero 1995.
[Gil06] Gillis, D., "Pattern-Based Design", tehan + lax blog, septiembre 14, 2006, disponible en www.teehan-lax.com/blog/?p=96.
[Gla98] Glass, R., "Defining Quality Intuitively", *IEEE Software*, mayo 1998, pp. 103-104, 107.
[Gla00] Gladwell, M., *The Tipping Point*, Back Bay Books, 2002.
[Gli07] Glinz, M. y R. Wieringa, "Stakeholders in Requirements Engineering", *IEEE Software*, vol. 24, núm. 2, marzo-abril 2007, pp. 18-20.
[Glu94] Gluch, D., "A Construct for Describing Software Development Risks", CMU/SEI-94-TR-14, Software Engineering Institute, 1994.
[Gna99] Gnaho, C. y F. Larcher, "A User-Centered Methodology for Complex and Customizable Web Engineering", *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Angeles, mayo 1999.
[Gon04] Gonzales, R., "Requirements Engineering", Sandia National Laboratories, presentación de diapositivas, disponible en www.incose.org/enchantment/docs/04AprRequirementsEngineering.pdf.
[Gor02] Gordon, B. y M. Gordon, *The Complete Guide to Digital Graphic Design*, Watson-Guptill, 2002.
[Gor06] Gorton, I., *Essential Software Architecture*, Springer, 2006.
[Gra87] Grady, R. B. y D. L. Caswell, *Software Metrics: Establishing a Company-Wide Program*, Prentice Hall, 1987.
[Gra92] Grady, R. B., *Practical Software Metrics for Project Management and Process Improvement*, Prentice Hall, 1992.
[Gra99] Grable, R., *et al.*, "Metrics for Small Projects: Experiences at SED," *IEEE Software*, marzo 1999, pp. 21-29.
[Gra03] Gradecki, J. y N. Lesiecki, *Mastering AspectJ: Aspect-Oriented Programming in Java*, Wiley, 2003.
[Gru02] Grundy, J., "Aspect-Oriented Component Engineering", 2002, www.cs.auckland.ac.nz/~john-g/aspects.html.
[Gus89] Gustavsson, A., "Maintaining the Evolution of Software Objects in an Integrated Environment", *Proc. 2nd Intl. Workshop on Software Configuration Management*, ACM, Princeton, NJ, octubre 1989, pp. 114-117.
[Gut93] Guttag, J. V. y J. J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.
[Hac98] Hackos, J. y J. Redish, *User and Task Analysis for Interface Design*, Wiley, 1998.
[Hai02] Hailpern, B. y P. Santhanam, "Software Debugging, Testing and Verification", *IBM Systems Journal*, vol. 41, núm. 1, 2002, disponible en www.research.ibm.com/journal/sj/ 411/hailpern.html.
[Hal77] Halstead, M., *Elements of Software Science*, North-Holland, 1977.

[Hal90] Hall, A., "Seven Myths of Formal Methods", *IEEE Software*, septiembre 1990, pp. 11-20.
[Hal98] Hall, E. M., *Managing Risk: Methods for Software Systems Development*, Addison-Wesley, 1998.
[Ham90] Hammer, M., "Reengineer Work: Don't Automate, Obliterate", *Harvard Business Review*, julio-agosto 1990, pp. 104-112.
[Han95] Hanna, M., "Farewell to Waterfalls", *Software Magazine*, mayo 1995, pp. 38-46.
[Har98a] Harmon, P., "Navigating the Distributed Components Landscape", *Cutter IT Journal.*, vol. 11, núm. 2, diciembre 1998, pp. 4-11.
[Har98b] Harrison, R., S. J. Counsell y R. V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics", *IEEE Trans. Software Engineering*, vol. SE-24, núm. 6, junio 1998, pp. 491-496.
[Her00] Herrmann, D., *Software Safety and Reliability*, Wiley-IEEE Computer Society Press, 2000.
[Het84] Hetzel, W., *The Complete Guide to Software Testing*, QED Information Sciences, 1984.
[Het93] Hetzel, W., *Making Software Measurement Work*, QED Publishing, 1993.
[Hev93] Hevner, A. R. y H. D. Mills, "Box Structure Methods for System Development with Objects", *IBM Systems Journal*, vol. 31, núm. 2, febrero 1993, pp. 232-251.
[Hig95] Higuera, R. P., "Team Risk Management", *CrossTalk*, U.S. Dept. of Defense, enero 1995, pp. 2-4.
[Hig00] Highsmith, J., *Adaptive Software Development: An Evolutionary Approach to Managing Complex Systems*, Dorset House Publishing, 2000.
[Hig01] Highsmith, J. (ed.), "The Great Methodologies Debate: Part 1", *Cutter IT Journal.*, vol. 14, núm. 12, diciembre 2001.
[Hig02a] Highsmith, J. (ed.), "The Great Methodologies Debate: Part 2", *Cutter IT Journal.*, vol. 15, núm. 1, enero 2002.
[Hig02b] Highsmith, J., *Agile Software Development Ecosystems*, Addison-Wesley, 2002.
[Hil05] Hildreth, S., "Buggy Software: Up from a Low Quality Quagmire", *Computerworld*, julio 25, 2005, disponible en www.computerworld.com/developmenttopics/development/story/ 0,10801,103378,00.html.
[Hil08] Hillside.net, *Patterns Catalog*, 2008, disponible en http://hillside.net/patterns/ onlinepatterncatalog.htm.
[Hob06] Hoberman, S., *Data Modeling Made Simple*, Technics Publications, 2006.
[Hof00] Hofmeister, C., R. Nord y D. Soni, *Applied Software Architecture*, Addison-Wesley, 2000.
[Hof01] Hofmann, C., *et al.*, "Approaches to Software Architecture", 2001, descargable de http://citeseer.nj.nec.com/84015.html.
[Hol06] Holzner, S., *Design Patterns for Dummies*, For Dummies Publishers, 2006.
[Hoo96] Hooker, D., "Seven Principles of Software Development", septiembre 1996, disponible en http://c2.com/cgi/wikiSevenPrinciplesOfSoftwareDevelopment.
[Hop90] Hopper, M. D., "Rattling SABRE, New Ways to Compete on Information", *Harvard Business Review*, mayo-junio 1990.
[Hor03] Horch, J., *Practical Guide to Software Quality Management*, 2a. ed., Artech House, 2003.
[HPR02] Hypermedia Design Patterns Repository, 2002, disponible en www.designpattern .lu.unisi.ch/index.htm.
[Hum95] Humphrey, W., *A Discipline for Software Engineering*, Addison-Wesley, 1995.
[Hum96] Humphrey, W., "Using a Defined and Measured Personal Software Process", *IEEE Software*, vol. 13, núm. 3, mayo-junio 1996, pp. 77-88.
[Hum97] Humphrey, W., *Introduction to the Personal Software Process*, Addison-Wesley, 1997.
[Hum98] Humphrey, W., "The Three Dimensions of Process Improvement, Part III: The Team Process", *CrossTalk*, abril 1998, disponible en www.stsc.hill.af.mil/crosstalk/1998/apr/dimensions.asp.
[Hum00] Humphrey, W., *Introduction to the Team Software Process*, Addison-Wesley, 2000.
[Hun99] Hunt, A., D. Thomas y W. Cunningham, *The Pragmatic Programmer*, Addison-Wesley, 1999.
[Hur83] Hurley, R. B., *Decision Tables in Software Engineering*, Van Nostrand-Reinhold, 1983.
[Hya96] Hyatt, L. y L. Rosenberg, "A Software Quality Model and Metrics for Identifying Project Risks and Assessing Software Quality", NASA SATC, 1996, disponible en http://satc.gsfc.nasa.gov/support/STC_APR96/quality/stc_qual.html.
[IBM81] "Implementing Software Inspections", notas del curso, IBM Systems Sciences Institute, IBM Corporation, 1981.
[IBM03] IBM, *Web Services Globalization Model*, 2003, disponible en www.ibm.com/ developerworks/webservices/library/ws-global/.
[IEE93a] *IEEE Standards Collection: Software Engineering*, IEEE Standard 610.12-1990, IEEE, 1993.
[IEE93b] *IEEE Standard Glossary of Software Engineering Terminology*, IEEE, 1993.
[IEE00] IEEE Standard Association, IEEE-Std-1471-2000, *Recommended Practice for Architectural Description of Software-Intensive Systems*, 2000, disponible en http://standards.ieee.org/ reading/ieee/std_public/description/se/1471-2000_desc.html.
[IFP01] *Function Point Counting Practices Manual*, Release 4.1.1, International Function Point Users Group, 2001, disponible en www.ifpug.org/publications/manual.htm.
[IFP05] Function Point Bibliography/Reference Library, International Function Point Users Group, 2005, disponible en www.ifpug.org/about/bibliography.htm.
[ISI08] iSixSigma, LLC, "New to Six Sigma: A Guide for Both Novice and Experiences Quality Practitioners", 2008, disponible en www.isixsigma.com/library/content/six-sigma-newbie.asp.

[ISO00] ISO 9001: 2000 Document Set, International Organization for Standards, 2000, www.iso.ch/iso/en/iso9000-14000/iso9000/iso9000index.html.
[ISO02] *Z Formal Specification Notation—Syntax, Type System and Semantics*, ISO/IEC 13568:2002, Intl. Standards Organization, 2002.
[ISO08] ISO SPICE, 2008, www.isospice.com/categories/SPICE-Project/.
[Ivo01] Ivory, M., R. Sinha y M. Hearst, "Empirically Validated Web Page Design Metrics", ACM *SIGCHI'01*, marzo 31-abril 4, 2001, disponible en http://webtango.berkeley.edu/papers/chi2001/.
[Jac75] Jackson, M. A., *Principles of Program Design*, Academic Press, 1975.
[Jac92] Jacobson, I., *Object-Oriented Software Engineering*, Addison-Wesley, 1992.
[Jac98] Jackman, M., "Homeopathic Remedies for Team Toxicity", *IEEE Software*, julio 1998, pp. 43-45.
[Jac99] Jacobson, I., G. Booch y J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.
[Jac02a] Jacobson, I., "A Resounding 'Yes' to Agile Processes—But Also More", *Cutter IT Journal*, vol. 15, núm. 1, enero 2002, pp. 18-24.
[Jac02b] Jacyntho, D., D. Schwabe y G. Rossi, "An Architecture for Structuring Complex Web Applications", 2002, disponible en www2002.org/CDROM/alternate/478/.
[Jac04] Jacobson, I. y P. Ng, *Aspect-Oriented Software Development*, Addison-Wesley, 2004.
[Jal04] Jalote, P., *et al.*, "Timeboxing: A Process Model for Iterative Software Development", *Journal of Systems and Software,*vol. 70, núm. 2, 2004, pp. 117-127. Disponible en www.cse .iitk.ac.in/users/jalote/papers/Timeboxing.pdf.
[Jay94] Jaychandra, Y., *Re-engineering the Networked Enterprise*, McGraw-Hill, 1994.
[Jec06] Jech, T., *Set Theory*, 3a. ed., Springer, 2006.
[Jon86] Jones, C., *Programming Productivity*, McGraw-Hill, 1986.
[Jon91] Jones, C., *Systematic Software Development Using VDM*, 2a. ed., Prentice Hall, 1991.
[Jon96] Jones, C., "How Software Estimation Tools Work", *American Programmer*, vol. 9, núm. 7, julio 1996, pp. 19-27.
[Jon98] Jones, C., *Estimating Software Costs*, McGraw-Hill, 1998.
[Jon04] Jones, C., "Software Project Management Practices: Failure Versus Success", *CrossTalk*, octubre 2004. Disponible en www.stsc.hill.af.mil/crossTalk/2004/10/0410Jones.html.
[Joy00] Joy, B., "The Future Doesn't Need Us", *Wired*, vol. 8, núm. 4, abril 2000.
[Kai02] Kaiser, J., "Elements of Effective Web Design", About, Inc., 2002, disponible en http://webdesign.about.com/library/weekly/aa091998.htm.
[Kal03] Kalman, S., *Web Security Field Guide*, Cisco Press, 2003.
[Kan93] Kaner, C., J. Falk y H. Q. Nguyen, *Testing Computer Software*, 2a. ed., Van Nostrand-Reinhold, 1993.
[Kan95] Kaner, C., "Lawyers, Lawsuits, and Quality Related Costs", 1995, disponible en www.badsoftware.com/plaintif.htm.
[Kan01] Kaner, C., "Pattern: Scenario Testing" (draft), 2001, disponible en www.testing.com/test-patterns/patterns/pattern-scenario-testing-kaner.html.
[Kar94] Karten, N., *Managing Expectations*, Dorset House, 1994.
[Kau95] Kauffman, S., *At Home in the Universe*, Oxford, 1995.
[Kaz98] Kazman, R., *et al.*, *The Architectural Tradeoff Analysis Method*, Software Engineering Institute, CMU/SEI-98-TR-008, julio 1998.
[Kaz03] Kazman, R. y A. Eden, "Defining the Terms Architecture, Design, and Implementation", *news@sei interactive*, Software Engineering Institute, vol. 6, núm. 1, 2003, disponible en www.sei.cmu.edu/news-at-sei/columns/the_architect/2003/1q03/architect-1q03.htm.
[Kei98] Keil, M., *et al.*, "A Framework for Identifying Software Project Risks", *CACM*, vol. 41, núm. 11, noviembre 1998, pp. 76-83.
[Kel00] Kelly, D. y R. Oshana, "Improving Software Quality Using Statistical Techniques, Information and Software Technology", Elsevier, vol. 42, agosto 2000, pp. 801-807, disponible en www.eng.auburn.edu/~kchang/comp6710/readings/Improving_Quality_with_Statistical_ Testing_InfoSoftTech_agosto2000.pdf.
[Ker78] Kernighan, B. y P. Plauger, *The Elements of Programming Style*, 2a. ed., McGraw-Hill, 1978.
[Ker05] Kerievsky, J., *Industrial XP: Making XP Work in Large Organizations*, Cutter Consortium, Executive Report, vol. 6., núm. 2, 2005, disponible en www.cutter.com/content-and-analysis/ resource-centers/agile-project-management/sample-our-research/apmr0502.html.
[Kim04] Kim, E., "A Manifesto for Collaborative Tools", *Dr. Dobb's Journal*, mayo 2004, disponible en www.blueoxen.com/papers/0000D/.
[Kir94] Kirani, S. y W. T. Tsai, "Specification and Verification of Object-Oriented Programs", Technical Report TR 94-64, Computer Science Department, University of Minnesota, diciembre 1994.
[Kiz05] Kizza, J., *Computer Network Security*, Springer, 2005.
[Knu98] Knuth, D., *The Art of Computer Programming*, tres volúmenes, Addison-Wesley, 1998.
[Kon02] Konrad, S. y B. Cheng, "Requirements Patterns for Embedded Systems", *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering*, IEEE, septiembre 2002, pp. 127-136, descargable de http://citeseer.ist.psu.edu/669258.html.

[Kra88] Krasner, G. y S. Pope, "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", *Journal of Object-Oriented Programming*, vol. 1, núm. 3, agosto-septiembre 1988, pp. 26-49.
[Kra95] Kraul, R. y L. Streeter, "Coordination in Software Development", *CACM*, vol. 38, núm. 3, marzo 1995, pp. 69-81.
[Kru05] Krutchen, P., "Software Design in a Postmodern Era", *IEEE Software*, vol. 22, núm. 2, marzo-abril 2005, pp. 16-18.
[Kru06] Kruchten, P., H. Obbink y J. Stafford (eds.), "Software Architectural" (número especial), *IEEE Software*, vol. 23, núm. 2, marzo-abril, 2006.
[Kur05] Kurzweil, R., *The Singularity Is Near*, Penguin Books, 2005.
[Kyb84] Kyburg, H. E., *Theory and Measurement*, Cambridge University Press, 1984.
[Laa00] Laakso, S., *et al.*, "Improved Scroll Bars", *CHI 2000 Conf. Proc.*, ACM, 2000, pp. 97-98, disponible en www.cs.helsinki.fi/u/salaakso/patterns/.
[Lai02] Laitenberger, A., "A Survey of Software Inspection Technologies", en *Handbook on Software Engineering and Knowledge Engineering*, World Scientific Publishing Company, 2002.
[Lam01] Lam, W., "Testing E-Commerce Systems: A Practical Guide", *IEEE IT Pro*, marzo-abril 2001, pp. 19-28.
[Lan01] Lange, M., "It's Testing Time! Patterns for Testing Software", junio 2001, descargable de www.testing.com/test-patterns/patterns/index.html.
[Lan02] Land, R., "A Brief Survey of Software Architecture", Technical Report, Dept. of Computer Engineering, Mälardalen University, Suecia, febrero 2002.
[Leh97a] Lehman, M. y L. Belady, *Program Evolution: Processes of Software Change*, Academic Press, 1997.
[Leh97b] Lehman, M., *et al.*, "Metrics and Laws of Software Evolution—The Nineties View", *Proceedings of the 4a. International Software Metrics Symposium (METRICS '97)*, IEEE, 1997, descargable de www.ece.utexas.edu/~perry/work/papers/feast1.pdf.
[Let01] Lethbridge, T. y R. Laganiere, *Object-Oriented Software Engineering: Practical Software Development Using UML and Java*, McGraw-Hill, 2001.
[Let03a] Lethbridge, T., Personal communication on domain analysis, mayo 2003.
[Let03b] Lethbridge, T., Personal communication on software metrics, junio 2003.
[Lev95] Leveson, N. G., *Safeware: System Safety and Computers*, Addison-Wesley, 1995.
[Lev01] Levinson, M., "Let's Stop Wasting $78 billion a Year", *CIO Magazine*, octubre 15, 2001, disponible en www.cio.com/archive/101501/wasting.html.
[Lew06] Lewicki, R., B. Barry y D. Saunders, *Essentials of Negotiation*, McGraw-Hill, 2006.
[Lie03] Lieberherr, K., "Demeter: Aspect-Oriented Programming", mayo 2003, disponible en www.ccs.neu.edu/home/lieber/LoD.html.
[Lin79] Linger, R., H. Mills y B. Witt, *Structured Programming*, Addison-Wesley, 1979.
[Lin88] Linger, R. M. y H. D. Mills, "A Case Study in Cleanroom Software Engineering: The IBM COBOL Structuring Facility", *Proc. COMPSAC '88*, Chicago, octubre 1988.
[Lin94] Linger, R., "Cleanroom Process Model", *IEEE Software*, vol. 11, núm. 2, marzo 1994, pp. 50-58.
[Lis88] Liskov, B., "Data Abstraction and Hierarchy", *SIGPLAN Notices*, vol. 23, núm. 5, mayo 1988.
[Liu98] Liu, K., *et al.*, "Report on the First SEBPC Workshop on Legacy Systems", Durham University, febrero 1998, disponible en www.dur.ac.uk/CSM/SABA/legacy-wksp1/report.html.
[Lon02] Longstreet, D., "Fundamental of Function Point Analysis", Longstreet Consulting, Inc., 2002, disponible en www.ifpug.com/fpafund.htm.
[Lor94] Lorenz, M. y J. Kidd, *Object-Oriented Software Metrics*, Prentice Hall, 1994.
[Maa07] Maassen, O. y S. Stelting, "Creational Patterns: Creating Objects in an OO System", 2007, disponible en www.informit.com/articles/article.asp?p=26452&rl=1.
[Man81] Mantai, M., "The Effect of Programming Team Structures on Programming Tasks", *CACM*, vol. 24, núm. 3, marzo 1981, pp. 106-113.
[Man97] Mandel, T., *The Elements of User Interface Design*, Wiley, 1997.
[Mar94] Marick, B., *The Craft of Software Testing*, Prentice Hall, 1994.
[Mar00] Martin, R., "Design Principles and Design Patterns", descargable de www .objectmentor.com, 2000.
[Mar01] Marciniak, J. J. (ed.), *Encyclopedia of Software Engineering*, 2a. ed., Wiley, 2001.
[Mar02] Marick, B., "Software Testing Patterns", 2002, www.testing.com/test-patterns/ index.html.
[McC76] McCabe, T., "A Software Complexity Measure", *IEEE Trans. Software Engineering*, vol. SE-2, diciembre 1976, pp. 308-320.
[McC77] McCall, J., P. Richards y G. Walters, "Factors in Software Quality", tres volúmenes, NTIS AD-A049-014, 015, 055, noviembre 1977.
[McC94] McCabe, T. J. y A. H. Watson, "Software Complexity", *CrossTalk*, vol. 7, núm. 12, diciembre 1994, pp. 5-9.
[McC96] McConnell, S., "Best Practices: Daily Build and Smoke Test", *IEEE Software*, vol. 13, núm. 4, julio 1996, pp. 143-144.
[McC98] McConnell, S., *Software Project Survival Guide*, Microsoft Press, 1998.
[McC99] McConnell, S., "Software Engineering Principles", *IEEE Software*, vol. 16, núm. 2, marzo-abril 1999, disponible en www.stevemcconnell.com/ieeesoftware/eic04.htm.
[McC04] McConnell, S., *Code Complete*, Microsoft Press, 2004.

[McC05] McCrory, A., "Ten Technologies to Watch in 2006", SeachCIO.com, octubre 27, 2005, disponible en http://searchcio.techtarget.com/originalContent/0,289142,sid19_gci1137889,00.html.
[McDE93] McDermid, J. y P. Rook, "Software Development Process Models", en *Software Engineer's Reference Book*, CRC Press, 1993, pp. 15/26-15/28.
[McG91] McGlaughlin, R., "Some Notes on Program Design", *Software Engineering Notes*, vol. 16, núm. 4, octubre 1991, pp. 53-54.
[McG94] McGregor, J. D. y T. D. Korson, "Integrated Object-Oriented Testing and Development Processes", *Communications of the ACM*, vol. 37, núm. 9, septiembre, 1994, pp. 59-77.
[Men01] Mendes, E., N. Mosley y S. Counsell, "Estimating Design and Authoring Effort", *IEEE Multimedia*, vol. 8, núm. 1, enero-marzo 2001, pp. 50-57.
[Mer93] Merlo, E., *et al.*, "Reengineering User Interfaces", *IEEE Software*, enero 1993, pp. 64-73.
[Mic08] *Microsoft Accessibility Technology for Everyone*, 2008, disponible en www.microsoft.com/enable/.
[Mic04] Microsoft, "Prescriptive Architecture: Integration and Patterns", *MSDN*, mayo 2004, disponible en http://msdn2.microsoft.com/en-us/library/ms978700.aspx.
[Mic07] Microsoft, "Patterns and Practices", *MSDN*, 2007, disponible en http://msdn2.microsoft .com/en-us/library/ms998478.aspx.
[Mil72] Mills, H. D., "Mathematical Foundations for Structured Programming", Technical Report FSC 71-6012, IBM Corp., Federal Systems Division, Gaithersburg, MD, 1972.
[Mil77] Miller, E., "The Philosophy of Testing", en *Program Testing Techniques*, IEEE Computer Society Press, 1977, pp. 1-3.
[Mil87] Mills, H. D., M. Dyer y R. Linger, "Cleanroom Software Engineering", *IEEE Software*, septiembre 1987, pp. 19-25.
[Mil88] Mills, H. D., "Stepwise Refinement and Verification in Box Structured Systems", *Computer*, vol. 21, núm. 6, junio 1988, pp. 23-35.
[Mil00a] Miller, E., "WebSite Testing", 2000, disponible en www.soft.com/eValid/Technology/ White.Papers/website.testing.html.
[Mil00b] Mili, A. y R. Cowan, "Software Engineering Technology Watch", abril 6, 2000, disponible en www.serc.net/projects/TechWatch/NSF%20TechWatch%20Proposal.htm.
[Min95] Minoli, D., *Analyzing Outsourcing*, McGraw-Hill, 1995.
[Mon84] Monk, A. (ed.), *Fundamentals of Human-Computer Interaction*, Academic Press, 1984.
[Mor81] Moran, T. P., "The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems", *Intl. Journal of Man-Machine Studies*, vol. 15, pp. 3-50.
[Mor05] Morales, A., "The Dream Team", *Dr. Dobbs Portal*, marzo 3, 2005, disponible en www.ddj .com/dept/global/184415303.
[Mus87] Musa, J. D., A. Iannino y K. Okumoto, *Engineering and Managing Software with Reliability Measures*, McGraw-Hill, 1987.
[Mus93] Musa, J., "Operational Profiles in Software Reliability Engineering", *IEEE Software*, marzo 1993, pp. 14-32.
[Mut03] Mutafelija, B. y H. Stromberg, *Systematic Process Improvement Using ISO 9001:2000 and CMMI*, Artech, 2003.
[Mye78] Myers, G., *Composite Structured Design*, Van Nostrand, 1978.
[Mye79] Myers, G., *The Art of Software Testing*, Wiley, 1979.
[NAS07] NASA, *Software Risk Checklist*, Form LeR-F0510.051, marzo 2007, descargable de http://osat-ext.grc.nasa.gov/rmo/spa/SoftwareRiskChecklist.doc.
[Nau69] Naur, P. y B. Randall (eds.), *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*, NATO, 1969.
[Ngu00] Nguyen, H., "Testing Web-Based Applications", *Software Testing and Quality Engineering*, mayo-junio 2000, disponible en www.stqemagazine.com.
[Ngu01] Nguyen, H., *Testing Applications on the Web*, Wiley, 2001.
[Ngu06] Nguyen, T., "Model-Based Version and Configuration Management for a Web Engineering Lifecycle", *Proc. 15th Intl. World Wide Web Conf.*, Edinburgo, Escocia, 2006, descargable de www2006.org/programme/item.php?id=4552.
[Nie92] Nierstrasz, O., S. Gibbs y D. Tsichritzis, "Component-Oriented Software Development," *CACM*, vol. 35, núm. 9, septiembre 1992, pp. 160-165.
[Nie94] Nielsen, J. y J. Levy, "Measuring Usability: Preference vs. Performance", *CACM*, vol. 37, núm. 4, abril 1994, pp. 65-75.
[Nie96] Nielsen, J. y A. Wagner, "User Interface Design for the WWW", *Proc. CHI '96 Conf. on Human Factors in Computing Systems*, ACM Press, 1996, pp. 330-331.
[Nie00] Nielsen, J., *Designing Web Usability*, New Riders Publishing, 2000.
[Nog00} Nogueira, J., C. Jones y Luqi, "Surfing the Edge of Chaos: Applications to Software Engineering", Command and Control Research and Technology Symposium, Naval Post Graduate School, Monterey, CA, junio 2000, descargable de www.dodccrp.org/ 2000CCRTS/cd/html/pdf_papers/Track_4/075.pdf.
[Nor70] Norden, P., "Useful Tools for Project Management" en *Management of Production*, M. K. Starr (ed.), Penguin Books, 1970.

[Nor86] Norman, D. A., "Cognitive Engineering", en *User Centered Systems Design*, Lawrence Earlbaum Associates, 1986.
[Nor88] Norman, D., *The Design of Everyday Things*, Doubleday, 1988.
[Nov04] Novotny, O., "Next Generation Tools for Object-Oriented Development", *The Architecture Journal*, enero 2005, disponible en http://msdn2.microsoft.com/en-us/library/aa480062.aspx.
[Noy02] Noyes, B., "Rugby, Anyone?" *Managing Development* (an online publication of Fawcette Technical Publications), junio 2002, www.fawcette.com/resources/managingdev/methodologies/scrum/.
[Off02] Offutt, J., "Quality Attributes of Web Software Applications", *IEEE Software*, marzo-abril 2002, pp. 25-32.
[Ols99] Olsina, L., *et al.*, "Specifying Quality Characteristics and Attributes for Web Sites", *Proc. 1st ICSE Workshop on Web Engineering*, ACM, Los Angeles, mayo 1999.
[Ols06] Olsen, G., "From COM to Common", *Component Technologies*, ACM, vol. 4, núm. 5, junio 2006, disponible en http://acmqueue.com/modules.php?name=Content&pa=showpage& pid=394.
[OMG03a] Object Management Group, *OMG Unified Modeling Language Specification*, version 1.5, marzo 2003, disponible en www.rational.com/uml/resources/documentation/.
[OMG03b] "Object Constraint Language Specification", en *Unified Modeling Language*, v2.0, Object Management Group, septiembre 2003, descargable de www.omg.org.
[Orf99] Orfali, R., D. Harkey y J. Edwards, *Client/Server Survival Guide*, 3a. ed., Wiley, 1999.
[Osb90] Osborne, W. M. y E. J. Chikofsky, "Fitting Pieces to the Maintenance Puzzle", *IEEE Software*, enero 1990, pp. 10-11.
[OSO08] OpenSource.org, 2008, disponible en www.opensource.org/.
[Pag85] Page-Jones, M., *Practical Project Management*, Dorset House, 1985, p. vii.
[Pal02] Palmer, S. y J. Felsing, *A Practical Guide to Feature Driven Development*, Prentice Hall, 2002.
[Par72] Parnas, D. L., "On Criteria to Be Used in Decomposing Systems into Modules", *CACM*, vol. 14, núm. 1, abril 1972, pp. 221-227.
[Par96a] Pardee, W., *To Satisfy and Delight Your Customer*, Dorset House, 1996.
[Par96b] Park, R. E., W. B. Goethert y W. A. Florac, *Goal Driven Software Measurement—A Guidebook*, CMU/SEI-96-BH-002, Software Engineering Institute, Carnegie Mellon University, agosto 1996.
[Pat07] Patton, J., "Understanding User Centricity", *IEEE Software*, vol. 24, núm. 6, noviembre-diciembre, 2007, pp. 9-11.
[Pau94] Paulish, D. y A. Carleton, "Case Studies of Software Process Improvement Measurement", *Computer*, vol. 27, núm. 9, septiembre 1994, pp. 50-57.
[PCM03] "Technologies to Watch", *PC Magazine*, julio 2003, disponible en www.pcmag.com/article2/0,4149, 1130591,00.asp.
[Per74] Persig, R., *Zen and the Art of Motorcycle Maintenance*, Bantam Books, 1974.
[Pet06] Pethokoukis, J., "Small Biz Watch: Future Business Trends", *U.S. News & World Report*, enero 20, 2006, disponible en www.usnews.com/usnews/biztech/articles/060120/ 20sbw.htm.
[Pha89] Phadke, M. S., *Quality Engineering Using Robust Design*, Prentice Hall, 1989.
[Pha97] Phadke, M. S., "Planning Efficient Software Tests", *CrossTalk*, vol. 10, núm. 10, octubre 1997, pp. 11-15.
[Phi98] Phillips, D., *The Software Project Manager's Handbook*, IEEE Computer Society Press, 1998.
[Phi02] Phillips, M., "CMMI V1.1 Tutorial.", abril 2002, disponible en www.sei.cmu.edu/cmmi/.
[Pol45] Polya, G., *How to Solve It*, Princeton University Press, 1945.
[Poo88] Poore, J. H. y H. D. Mills, "Bringing Software Under Statistical Quality Control", *Quality Progress*, noviembre 1988, pp. 52-55.
[Poo93] Poore, J. H., H. D. Mills y D. Mutchler, "Planning and Certifying Software System Reliability", *IEEE Software*, vol. 10, núm. 1, enero 1993, pp. 88-99.
[Pop03] Poppendieck, M. y T. Poppendieck, *Lean Software Development*, Addison-Wesley, 2003.
[Pop06a] Poppendeick, LLC, *Lean Software Development*, disponible en www.poppendieck.com/.
[Pop06b] Poppendieck, M. y T. Poppendieck, *Implementing Lean Software Development*, Addison-Wesley, 2006.
[Pop08] Popcorn, F., *Faith Popcorn's Brain Reserve*, 2008, disponible en www.faithpopcorn.com/.
[Pot04] Potter, M., *Set Theory and Its Philosophy: A Critical Introduction*, Oxford University Press, 2004.
[Pow98] Powell, T., *Web Site Engineering*, Prentice Hall, 1998.
[Pow02] Powell, T., *Web Design*, 2a. ed., McGraw-Hill/Osborne, 2002.
[Pre94] Premerlani, W. y M. Blaha, "An Approach for Reverse Engineering of Relational Databases", *CACM*, vol. 37, núm. 5, mayo 1994, pp. 42-49.
[Pre88] Pressman, R., *Making Software Engineering Happen*, Prentice Hall, 1988.
[Pre05] Pressman, R., *Adaptable Process Model*, revision 2.0, R. S. Pressman & Associates, 2005, disponible en www.rspa.com/apm/index.html.
[Pre08] Pressman, R. y D. Lowe, *Web Engineering: A Practitioner's Approach*, McGraw-Hill, 2008.
[Put78] Putnam, L., "A General Empirical Solution to the Macro Software Sizing and Estimation Problem", *IEEE Trans. Software Engineering*, vol. SE-4, núm. 4, julio 1978, pp. 345-361.
[Put92] Putnam, L. y W. Myers, *Measures for Excellence*, Yourdon Press, 1992.

[Put97a] Putnam, L. y W. Myers, "How Solved Is the Cost Estimation Problem?" *IEEE Software*, noviembre 1997, pp. 105-107.
[Put97b] Putnam, L. y W. Myers, *Industrial Strength Software: Effective Management Using Measurement*, IEEE Computer Society Press, 1997.
[Pyz03] Pyzdek, T., *The Six Sigma Handbook*, McGraw-Hill, 2003.
[QAI08] *A Software Engineering Curriculum*, QAI, 2008, información obtenida en www .qaieschool.com/innerpages/offer.asp.
[QSM02] "QSM Function Point Language Gearing Factors", Version 2.0, Quantitative Software Management, 2002, www.qsm.com/FPGearing.html.
[Rad02] Radice, R., *High-Quality Low Cost Software Inspections*, Paradoxicon Publishing, 2002.
[Rai06] Raiffa, H., *The Art and Science of Negotiation*, Belknap Press, 2005.
[Ree99] Reel, J. S., "Critical Success Factors in Software Projects", *IEEE Software*, mayo 1999, pp. 18-23.
[Ric01] Ricadel, A., "The State of Software Quality", *InformationWeek*, mayo 21, 2001, disponible en www.informationweek.com/838/quality.htm.
[Ric04] Rico, D., *ROI of Software Process Improvement*, J. Ross Publishing, 2004. Se puede encontrar un artículo resumido en http://davidfrico.com/rico03a.pdf.
[Roc94] Roche, J. M., "Software Metrics and Measurement Principles", *Software Engineering Notes*, ACM, vol. 19, núm. 1, enero 1994, pp. 76-85.
[Roc06] *Graphic Design That Works*, Rockport Publishers, 2006.
[Roe00] Roetzheim, W., "Estimating Internet Development", *Software Development*, agosto 2000, disponible en www.sdmagazine.com/documents/s=741/sdm0008d/0008d.htm.
[Roo96] Roos, J., "The Poised Organization: Navigating Effectively on Knowledge Landscapes", 1996, disponible en www.imd.ch/fac/roos/paper_po.html.
[Ros75] Ross, D., J. Goodenough y C. Irvine, "Software Engineering: Process, Principles and Goals", *IEEE Computer*, vol. 8, núm. 5, mayo 1975.
[Ros04] Rosenhainer, L., "Identifying Crosscutting Concerns in Requirements Specifications", 2004, disponible en http://trese.cs.utwente.nl/workshops/oopsla-early-aspects-2004/ Papers/Rosenhainer.pdf.
[Rou02] Rout, T (project manager), *SPICE: Software Process Assessment—Part 1: Concepts and Introductory Guide*, 2002, descargable de www.sqi.gu.edu.au/spice/suite/download .html.
[Roy70] Royce, W. W., "Managing the Development of Large Software Systems: Concepts and Techniques", *Proc. WESCON*, agosto 1970.
[Roz05] Rozanski, N. y E. Woods, *Software Systems Architecture*, Addison-Wesley, 2005.
[Rub88] Rubin, T., *User Interface Design for Computer Systems*, Halstead Press (Wiley), 1988.
[Rum91] Rumbaugh, J., *et al.*, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.
[Sar06] Sarwate, A., "Hot or Not: Web Application Vulnerabilities", *SC Magazine*, diciembre 27, 2006, disponible en http://scmagazine.com/us/news/article/623765/hot-not-web-application-vulnerabilities.
[Sca00] Scacchi, W., "Understanding Software Process Redesign Using Modeling, Analysis, and Simulation", *Software Process Improvement and Practice*, Wiley, 2000, pp. 185-195, descargable de www.ics.uci.edu/~wscacchi/Papers/Software_Process_Redesign/ SPIP-ProSim99.pdf.
[Sce02] Sceppa, D., *Microsoft ADO.NET*, Microsoft Press, 2002.
[Sch95] Schwabe, D. y G. Rossi, "The Object-Oriented Hypermedia Design Model", *CACM*, vol. 38, núm. 8, agosto 1995, pp. 45-46.
[Sch96] Schorsch, T., "The Capability Im-Maturity Model", *CrossTalk*, noviembre 1996, disponible en www.stsc.hill.af.mil/crosstalk/1996/11/xt96d11h.asp.
[Sch98a] Schneider, G. y J. Winters, *Applying Use Cases*, Addison-Wesley, 1998.
[Sch98b] Schwabe, D. y G. Rossi, "Developing Hypermedia Applications Using OOHDM", *Proc. Workshop on Hypermedia Development Process, Methods and Models, Hypertext '98*, 1998, descargable de http://citeseer.nj.nec.com/schwabe98developing.html.
[Sch98c] Schulmeyer, G. C. y J. I. McManus (eds.), *Handbook of Software Quality Assurance*, 3a. ed., Prentice Hall, 1998.
[Sch99] Schneidewind, N., "Measuring and Evaluating Maintenance Process Using Reliability, Risk, and Test Metrics", *IEEE Trans. SE*, vol. 25, núm. 6, noviembre-diciembre 1999, pp. 768-781, descargable de www.dacs.dtic.mil/topics/reliability/IEEETrans.pdf.
[Sch01a] Schwabe, D., G. Rossi y Barbosa, S., "Systematic Hypermedia Application Design Using OOHDM", 2001, disponible en www-di.inf.puc-rio.br/~schwabe/HT96WWW/section1.html.
[Sch01b] Schwaber, K. y M. Beedle, *Agile Software Development with SCRUM*, Prentice Hall, 2001.
[Sch02] Schwaber, K., "Agile Processes and Self-Organization", Agile Alliance, 2002, www.aanpo.org/articles/index.
[Sch03] Schlickman, J., *ISO 9001: 2000 Quality Management System Design*, Artech House Publishers, 2003.
[Sch06] Schmidt, D., "Model-Driven Engineering", *IEEE Computer*, vol. 39, núm. 2, febrero 2006, pp. 25-31.
[SDS08] Spice Document Suite, "The SPICE and ISO Document Suite", ISO-Spice, 2008, disponible en www.isospice.com/articles/9/1/SPICE-Project/Page1.html.
[Sea93] Sears, A., "Layout Appropriateness: A Metric for Evaluating User Interface Widget Layout, *IEEE Trans. Software Engineering*, vol. SE-19, núm. 7, julio 1993, pp. 707-719.

[SEE03] The Software Engineering Ethics Research Institute, "UCITA Updates", 2003, disponible en http://seeri.etsu.edu/default.htm.
[SEI00] *SCAMPI, V1.0 Standard CMMI ®Assessment Method for Process Improvement: Method Description*, Software Engineering Institute, Technical Report CMU/SEI-2000-TR-009, descargable de www.sei.cmu.edu/publications/documents/00.reports/00tr009.html.
[SEI02] "Maintainability Index Technique for Measuring Program Maintainability", SEI, 2002, disponible en www.sei.cmu.edu/str/descriptions/mitmpm_body.html.
[SEI08] "The Ideal Model", Software Engineering Institute, 2008, disponible en www.sei.cmu .edu/ideal/.
[Sha95a] Shaw, M. y D. Garlan, "Formulations and Formalisms in Software Architecture", *Volume 1000—Lecture Notes in Computer Science*, Springer-Verlag, 1995.
[Sha95b] Shaw, M., *et al.*, "Abstractions for Software Architecture and Tools to Support Them", *IEEE Trans. Software Engineering*, vol. SE-21, núm. 4, abril 1995, pp. 314-335.
[Sha96] Shaw, M. y D. Garlan, *Software Architecture*, Prentice Hall, 1996.
[Sha05] Shalloway, A. y J. Trott, *Design Patterns Explained*, 2a. ed., Addison-Wesley, 2005.
[Shn80] Shneiderman, B., *Software Psychology*, Winthrop Publishers, 1980, p. 28.
[Shn04] Shneiderman, B. y C. Plaisant, *Designing the User Interface*, 4a. ed., Addison-Wesley, 2004.
[Sho83] Shooman, M. L., *Software Engineering*, McGraw-Hill, 1983.
[Sim05] Simsion, G. y G. Witt, *Data Modeling Essentials*, 3a. ed., Morgan Kaufman, 2005.
[Sin99] Singpurwalla, N. y S. Wilson, *Statistical Methods in Software Engineering: Reliability and Risk*, Springer-Verlag, 1999.
[Smi99] Smith, J., "The Estimation of Effort Based on Use Cases", Rational Software Corp., 1999, descargable de www.rational.com/media/whitepapers/finalTP171.PDF.
[Smi05] Smith, D, *Reliability, Maintainability and Risk*, 7a. ed., Butterworth-Heinemann, 2005.
[Sne95] Sneed, H., "Planning the Reengineering of Legacy Systems", *IEEE Software*, enero 1995, pp. 24-25.
[Sne03] Snee, R. y R. Hoerl, *Leading Six Sigma*, Prentice Hall, 2003.
[Sol99] van Solingen, R. y E. Berghout, *The Goal/Question/Metric Method*, McGraw-Hill, 1999.
[Som97] Somerville, I. y P. Sawyer, *Requirements Engineering*, Wiley, 1997.
[Som05] Somerville, I., "Integrating Requirements Engineering: A Tutorial", *IEEE Software*, vol. 22, núm. 1, enero-febrero 2005, pp. 16-23.
[SPI99] "SPICE: Software Process Assessment, Part 1: Concepts and Introduction", Version 1.0, ISO/IEC JTC1, 1999.
[Spl01] Splaine, S. y S. Jaskiel, *The Web Testing Handbook*, STQE Publishing, 2001.
[Spo02] Spolsky, J, "The Law of Leaky Abstractions", noviembre 2002, disponible en www .joelonsoftware.com/articles/LeakyAbstractions.html.
[Sri01] Sridhar, M. y N. Mandyam, "Effective Use of Data Models in Building Web Applications," 2001, disponible en www2002.org/CDROM/alternate/698/.
[SSO08] Software-Supportability.org, www.software-supportability.org/ 2008.
[Sta97] Stapleton, J., *DSDM—Dynamic System Development Method: The Method in Practice*, Addison-Wesley, 1997.
[Sta97b] Statz, J., D. Oxley y P. O'Toole, "Identifying and Managing Risks for Software Process Improvement", *CrossTalk*, abril 1997, disponible en www.stsc.hill.af.mil/crosstalk/1997/ 04/identifying.asp.
[Ste74] Stevens, W., G. Myers y L. Constantine, "Structured Design", *IBM Systems Journal*, vol. 13, núm. 2, 1974, pp. 115-139.
[Ste93] Stewart, T. A., "Reengineering: The Hot New Managing Tool", *Fortune*, agosto 23, 1993, pp. 41-48.
[Ste99] Stelzer, D. y W. Mellis, "Success Factors of Organizational Change in Software Process Improvement", *Software Process Improvement and Practice*, vol. 4, núm. 4, Wiley, 1999, descargable de www.systementwicklung.uni-koeln.de/forschung/artikel/dokumente/successfactors.pdf.
[Ste03] Stephens, M. y D. Rosenberg, *Extreme Programming Refactored*, Apress, 2003.
[Sto05] Stone, D., *et al.*, *User Interface Design and Evaluation*, Morgan Kaufman, 2005.
[Tai89] Tai, K. C., "What to Do Beyond Branch Testing," *ACM Software Engineering Notes*, vol. 14, núm. 2, abril 1989, pp. 58-61.
[Tay90] Taylor, D., *Object-Oriented Technology: A Manager's Guide*, Addison-Wesley, 1990.
[Tha97] Thayer, R. H. y M. Dorfman, *Software Requirements Engineering*, 2a. ed., IEEE Computer Society Press, 1997.
[The01] Thelin, T., H. Petersson y C. Wohlin, "Sample Driven Inspections", *Proc. of Workshop on Inspection in Software Engineering (WISE'01)*, París, Francia, julio 2001, pp. 81-91, descargable de http://www.cas.mcmaster.ca/wise/wise01/ThelinPeterssonWohlin.pdf.
[Tho92] Thomsett, R., "The Indiana Jones School of Risk Management", *American Programmer*, vol. 5, núm. 7, septiembre 1992, pp. 10-18.
[Tic05] *TickIT*, 2005, www.tickit.org/.
[Tid02] Tidwell, J., "IU Patterns and Techniques", mayo 2002, disponible en http://time-tripper.com/uipatterns/index.html.
[Til93] Tillmann, G., *A Practical Guide to Logical Data Modeling*, McGraw-Hill, 1993.
[Til00] Tillman, H., "Evaluating Quality on the Net", Babson College, mayo 30, 2000, disponible en www.hopetillman.com/findqual.html#2.

[Tog01] Tognozzi, B., "First Principles", *askTOG*, 2001, disponible en www.asktog.com/basics/firstPrinciples.html.
[Tra95] Tracz, W., "Third International Conference on Software Reuse—Summary", *ACM Software Engineering Notes*, vol. 20, núm. 2, abril 1995, pp. 21-22.
[Tre03] Trivedi, R, *Professional Web Services Security*, Wrox Press, 2003.
[Tri05] Tricker, R. y B. Sherring-Lucas, *ISO 9001: 2000 In Brief*, 2a. ed., Butterworth-Heinemann, 2005.
[Tyr05] Tyree, J. y A. Akerman, "Architectural Decisions: Demystifying Architecture", *IEEE Software*, vol. 22, núm. 2, marzo-abril, 2005.
[Uem99] Uemura, T., S. Kusumoto y K. Inoue: "A Function Point Measurement Tool for UML Design Specifications", *Proc. of Sixth International Symposium on Software Metrics*, IEEE, noviembre 1999, pp. 62-69
[Ull97] Ullman, E., *Close to the Machine: Technophilia and Its Discontents*, City Lights Books, 2002.
[UML03] The UML Café, "Customers Don't Print Themselves", mayo 2003, disponible en www.theumlcafe.com/a0079.htm.
[Uni03] Unicode, Inc., *The Unicode Home Page*, 2003, disponible en www.unicode.org/.
[USA87] *Management Quality Insight*, AFCSP 800-14 (U.S. Air Force), enero 20, 1987.
[Vac06] Vacca, J., *Practical Internet Security*, Springer, 2006.
[Van89] Van Vleck, T., "Three Questions About Each Bug You Find", *ACM Software Engineering Notes*, vol. 14, núm. 5, julio 1989, pp. 62-63.
[Van02] Van Steen, M. y A. Tanenbaum, *Distributed Systems: Principles and Paradigms*, Prentice Hall, 2002.
[Ven03] Venners, B., "Design by Contract: A Conversation with Bertrand Meyer", *Artima Developer*, diciembre 8, 2003, disponible en www.artima.com/intv/contracts.html.
[Wal03] Wallace, D., I. Raggett y J. Aufgang, *Extreme Programming for Web Projects*, Addison-Wesley, 2003.
[War74] Warnier, J. D., *Logical Construction of Programs*, Van Nostrand-Reinhold, 1974.
[War07] Ward, M., "Using VoIP Software Building zBlocks—A Look at the Choices", TMNNet, 2007, disponible en www.tmcnet.com/voip/0605/featurearticle-using-voip-software-building-blocks.htm.
[Web05] Weber, S., *The Success of Open Source*, Harvard University Press, 2005.
[Wei86] Weinberg, G., *On Becoming a Technical Leader*, Dorset House, 1986.
[Wel99] Wells, D., "XP—Unit Tests", 1999, disponible en www.extremeprogramming.org/rules/unittests.html.
[Wel01] vanWelie, M., "Interaction Design Patterns", 2001, disponible en www.welie.com/ patterns/.
[Whi95] Whittle, B., "Models and Languages for Component Description and Reuse", *ACM Software Engineering Notes*, vol. 20, núm. 2, abril 1995, pp. 76-89.
[Whi97] Whitmire, S., *Object-Oriented Design Measurement*, Wiley, 1997.
[Wie02] Wiegers, K., *Peer Reviews in Software*, Addison-Wesley, 2002.
[Wie03] Wiegers, K., *Software Requirements*, 2a. ed., Microsoft Press, 2003.
[Wil93] Wilde, N. y R. Huitt, "Maintaining Object-Oriented Software", *IEEE Software*, enero 1993, pp. 75-80.
[Wil97] Williams, R. C, J. A. Walker y A. J. Dorofee, "Putting Risk Management into Practice", *IEEE Software*, mayo 1997, pp. 75-81.
[Wil99] Wilkens, T. T., "Earned Value, Clear and Simple", Primavera Systems, abril 1, 1999, p. 2.
[Wil00] Williams, L. y R. Kessler, "All I Really Need to Know about Pair Programming I Learned in Kindergarten", *CACM*, vol. 43, núm. 5, mayo 2000, disponible en http://collaboration.csc.ncsu.edu/laurie/Papers/Kindergarten.PDF.
[Wil05] Willoughby, M., "Q&A: Quality Software Means More Secure Software", *Computerworld*, marzo 21, 2005, disponible en www.computerworld.com/securitytopics/security/story/ 0,10801,91316,00.html.
[Win90] Wing, J. M., "A Specifier's Introduction to Formal Methods", *IEEE Computer*, vol. 23, núm. 9, septiembre 1990, pp. 8-24.
[Wir71] Wirth, N., "Program Development by Stepwise Refinement", *CACM*, vol. 14, núm. 4, 1971, pp. 221-227.
[Wir90] Wirfs-Brock, R., B. Wilkerson y L. Weiner, *Designing Object-Oriented Software*, Prentice Hall, 1990.
[WMT02] Web Mapping Testbed Tutorial., 2002, disponible en www.webmapping.org/ vcgdocuments/vcg-Tutorial/.
[Woh94] Wohlin, C. y P. Runeson, "Certification of Software Components", *IEEE Trans. Software Engineering*, vol. SE-20, núm. 6, junio 1994, pp. 494-499.
[Wor04] World Bank, *Digital Technology Risk Checklist*, 2004, descargable de www.moonv6.org/lists/att-0223/WWBANK_Technology_Risk_Checklist_Ver_6point1.pdf.
[W3C03] World Wide Web Consortium, *Web Content Accessibility Guidelines*, 2003, disponible en www.w3.org/TR/2003/WD-WCAG20-20030624/.
[Yac03] Yacoub, S., *et al.*, *Pattern-Oriented Analysis and Design*, Addison-Wesley, 2003.
[You75] Yourdon, E., *Techniques of Program Structure and Design*, Prentice Hall, 1975.
[You79] Yourdon, E. y L. Constantine, *Structured Design*, Prentice Hall, 1979.
[You95] Yourdon, E., "When Good Enough Is Best", *IEEE Software*, vol. 12, núm. 3, mayo 1995, pp. 79-81.
[You01] Young, R., *Effective Requirements Practices*, Addison-Wesley, 2001.
[Zah90] Zahniser, R. A., "Building Software in Groups", *American Programmer*, vol. 3, núms. 7-8, julio-agosto 1990.
[Zah94] Zahniser, R., "Timeboxing for Top Team Performance," *Software Development*, marzo 1994, pp. 35-38.

[Zha98] Zhao, J, "On Assessing the Complexity of Software Architectures", *Proc. Intl. Software Architecture Workshop*, ACM, Orlando, FL, 1998, pp. 163-167.
[Zha02] Zhao, H., "Fitt's Law: Modeling Movement Time in HCI", *Theories in Computer Human Interaction*, University of Maryland, octubre 2002, disponible en www.cs.umd.edu/class/fall2002/cmsc838s/tichi/fitts.html.
[Zul92] Zultner, R., "Quality Function Deployment for Software: Satisfying Customers", *American Programmer*, febrero 1992, pp. 28-41.
[Zus90] Zuse, H., *Software Complexity: Measures and Methods*, DeGruyter, 1990.
[Zus97] Zuse, H., *A Framework of Software Measurement*, DeGruyter, 1997.

Índice analítico

A

Abstracción, 85
  de datos, 190
  dimensión de la, 197
Accesibilidad, 283, 306
Acción, 12, 167
Acoplamiento, 244-246
Actividad(es), 12
  estructural(es), 12-13, 28
  sombrilla, 13-14
Actor(es), 113, 132, 217
Administración
  de contenido, 517-519
  de la complejidad, 700-701
  de la configuración del software, 14, 508
  de la reutilización, 14
  de los requerimientos, 105, 508
  del cambio, 520-521
  del riesgo, 13
  efecto de las acciones de la, 349-350
  orientada a pruebas, 65
Administración de la configuración del software, 501
  auditoría de configuración, 514
  control de cambio, 511-513
  control de versión, 510-511
  elementos de la, 503-504
  escenario operativo, 502-503
  identificación de objetos, 509-510
  ítems (ICS), 502, 505
  línea de referencia, 504-505
  para webapps, 515-523
  reporte del estado de la configuración, 515
  repositorio, 506-508
Administración de la calidad; *véase* Aseguramiento de la calidad del software
Administración de proyectos, 350
  conceptos clave, 554-556
Agilidad, 56-57, 84
  costo del cambio, 57-58
  espíritu ágil, 59
  factores humanos, 60-61
  principios de, 58-59
  proceso, 58
Alcance del proyecto, 89
Ambiente de trabajo, 278
Ámbito del software, 595-596
Amenaza, 583
Análisis, 92, 528
  clases de, 143-145
  de la interfaz, 269, 271, 272-278
  de la tarea, 271, 273-274
  de patrones, 120-121
  de valor de frontera (BVA), 425-426, 466
  de valor ganado (AVG), 635-637
  del dominio, 129-130, 257
  del flujo del trabajo, 276-277
  del mercado, 273
  del usuario, 272-273
  estructurado, 130, 158
  gramatical, 143-144, 160
  orientado a objetos, 131, 159
  paquetes de, 154-155
Análisis de los requerimientos, 127
  objetivos, 128
  reglas prácticas, 128-129
  tipos de modelo, 127
Aplicaciones interactivas de inmersión, 211
Aprendizaje, 69
  continuo, 66
  en interfaz de webapp, 287
  patrones, 306-308
Árbol de decisión, 615-616
Aristas, 415
Arquetipo, 218
Arquitectura(s), 207
  de intercambio de objetos comunes solicitados (GAO/ATOCS), 259
  de una webapp, 328-329
  del contenido, 326-328
  funcional, 252
  sencillas de redes, 7
Arquitectura del software, 93, 190-191
  centrada en los datos, 213
  complejidad de la, 224
  de flujo de datos, 213
  de inmerpresencia (ASI), 211
  de llamada de procedimiento remoto, 214
  de llamar y regresar, 214, 225
  de programa principal/subprograma, 214
  decisiones arquitectónicas, 209, 210
  dependencias entre los componentes, 224
  descripción arquitectónica (DA), 208-209
  diagrama de contexto arquitectónico (DCA), 217
  diseño arquitectónico, 208, 217-221
  en capas, 214-215
  estilo arquitectónico, 211-212
  evaluación de la, 221-225
  género arquitectónico, 209-211
  importancia de la, 208
  instancias de la, 221-222
  lenguaje de descripción arquitectónica (LDA), 224-225
  mapeo, 225-232
  método de negociación para analizar la, 222-223
  orientadas a objetos, 214
  propiedades de la, 190
  refinamiento de la, 219-220, 231-232
Arreglo ortogonal, 426-428
Aseguramiento de la calidad del software, 14, 351
  acciones de, 371-372
  atributos, 373
  elementos de, 370-371
  enfoques formales, 373-374
  metas del, 372
  métodos estadísticos, 374-376
  Plan de, 379-380
  pocas vitales, 374
  Seis Sigma, 375-376
ASI; *véase* Arquitectura de Software de Inmerpresencia
Asignación de tiempo, 623
Asociaciones, 153-154
ATAM *véase* Método de la negociación para la arquitectura
Atractivo visual de webapps, 321
Atributos, 745
Auditoría, 522
  de configuración, 514
Autenticación, 471
Autonomía controlada, 286
Autorización, 471
AVG; *véase* Análisis de valor ganado

B

Barra de navegación horizontal, 331
Base de datos, prueba de la, 458-459
Belleza, 183
Biblioteca de reutilización, 261-262
Bucles, 421-422
BVA; *véase* Análisis de valor de frontera

C

Calendarización, 629
  calendario del proyecto, 631-632
  de proyectos webapp, 633-635
  del proyecto de software, 622-626
  evaluación del programa y la técnica de revisión (PERT), 629
  fechas límite agresivas, 621

método de ruta crítica (CPM), 628, 629
para un proyecto OO, 632-633
principios básicos, 621-622
Calidad, 84, 89, 185
árbol de requerimientos de la, 319
aseguramiento de la, 14
de la conformidad, 339
del diseño, 339
despliegue de la función de, 111
evaluación de la, 319-320
puntos de vista, 339
Calidad del software, 186, 340
administración de proyectos, 350
aseguramiento de la, 351
atributos de la, 187-188
control de, 351
costo de la, 346-348
decisiones administrativas, 349-350
dilema de la, 345
dimensiones de Garvin de la, 341
estándar ISO 9126, 343
factores de McCall, 342
lineamientos de la, 186-187
logro de la, 350-351
métodos, 350
naturaleza subjetiva de la, 344-345
negligencia, 348-349
prueba de aplicación web, 465-466
responsabilidad, 348
riesgos, 348
y seguridad, 349
Calificación del proyecto, 65
Cambio, 84, 89
dimensionamiento del, 600
"Campeón del proyecto", 107-108
Caos, 33, 71
Capacidad para resolver problemas difusos, 60
Capas de interacción, 459-460
Características, calidad de las, 341
Cardinalidad, 142
Carga impredecible, 9
CasaSegura
acoplamiento, 245-246
actores, 114
análisis de riesgos, 648
análisis del dominio, 130
aplicación de métricas CK, 541
aplicación de patrones, 308-309
árbol de datos, 176-177
arquetipos, 219
aspectos de la calidad, 366
caso de uso, 116, 132-133
clase de diseño, 197
clases potenciales, 145
cohesión, 244
complejidad ciclomática, 417-418
conceptos de diseño, 195
conflictos ACS, 513-514
debate acerca de las métricas, 531
desarrollo ágil de software, 67
diagrama de actividades, 179
diagrama de contexto arquitectónico (DCA), 217
diagrama de estado, 163
diagrama de flujo de datos, 160-161, 226-227
diagrama de secuencia UML, 171
diseño de la interfaz de usuario, 274-275
diseño de pruebas únicas, 413
diseño *vs.* codificación, 186
distribución preliminar de la pantalla, 281
enfoque de métricas, 575
errores de comunicación, 88
escenario preliminar de uso, 112
estilo de arquitectura, 216
estimación, 603
evaluación de la arquitectura, 223
formato de caso de uso, 136
inicio del proyecto, 20-21
modelado del comportamiento, 120
modelado del flujo de datos, 164
modelos CRC, 153
modelos de clase, 147-148
negociación, 122
outsourcing, 616-617
patrón de requerimientos, 170-174
preparación para la prueba, 387
Principio Abierto-Cerrado, 240
proceso, 406
prueba de clase, 447-448
PSPEC, 164-165
recabación de los requerimientos, 111
rediseño, 228-230
refinación de la arquitectura, 231
regla dorada, 268
revisión del diseño de la interfaz, 288
seguimiento de calendario, 635
Cascada, modelo de la, 34
Casos de prueba, 418-419
en OO, 443
Caso(s) de uso, 61, 112, 273-274
creación de un, 132-134
desarrollo de, 113-117
diagrama de actividad UML, 137-138
diagrama de canal de UML, 138
disparador (o trigger), 136
escenario, 136
eventos y, 166
excepciones, 135
formales, 135-137
formato, 115-116
informales, 132-134
mejora de un, 134-135
modelos UML, 137-139
objetivo en contexto, 135-136
precondición, 136
Castellano estructurado; *véase* Lenguaje de diseño del programa
CBA IPI; *véase* CMM
Centro de transformación, 225
Certificación, 487
Ciclo de vida
clásico; *véase* Modelo de la cascada
MDSD, 71
Clase(s), 118, 148-149, 744
categorías de, 149
de análisis, 138, 143-145, 155
de controlador, 746
de diseño, 747-749
de entidad, 149, 745
de equivalencia, 425
de frontera, 149
de proceso, 196
de sistemas, 196
de sustantivos, 144
de usuario de la interfaz, 196
del dominio de negocios, 196
estado de una, 166-167
inteligentes, 150
modelo o de negocio, 149
orientadas a objetos, 141
persistentes, 196
tontas, 150
Clases de análisis
atributos de las, 145-146
características de selección, 144-145
definición de las operaciones, 146-148
diagrama de estado para, 167-168
identificación de las, 143-145
tipos de, 149
Clientes, 87
expectativas de los, 96
CMM; *véase* Modelo de madurez de la capacidad
CMMI; *véase* Integración del Modelo de Madurez de Capacidades
Codificación XP, 64
principios de, 94-95
Coherencia, 85
Cohesión, 162, 243-244
Colaboración, 60, 69, 87, 107, 149
en la recabación, 109-110
Colaboraciones, 151-152
Colaboradores, 149
Columna de navegación vertical, 331
Comandos escritos, 283
Compartimentalización, 623
Compatibilidad de webapps, 321, 454
Competencia, 60
Complejidad
arquitectónica, 224
ciclomática, 417
Componente(s), 93, 235, 242
adaptación, 258
biblioteca de, 257
calificación de, 258
clasificación de, 260-262
combinación de, 259
comerciales, 597
de experiencia completa, 597
de experiencia parcial, 598
dimensionamiento de, 600
diseño del contenido en el nivel de, 251-252
envoltura de, 258-259
estándares del software de, 259
estándares y marcos basados en, 239
modelo 3C, 260-262
nuevos, 598
patrones de, 297
Comportamiento, 118
Comprensibilidad, 412

Comprobabilidad, 412
Computación en un mundo abierto, 7, 701
Comunicación, 13, 61, 84
  cara a cara, 87
  en interfaz de webapp, 285
  principios de, 86-88
Comunicaciones, 210
Comunidad del proyecto, 65
Concentración en interfaz de webapp, 285
Concepción, 46-47, 102-103, 132
Concepto, modelo 3C, 261
Concurrencia, 9, 306-307
Condición compuesta, 421
Confiabilidad, 188, 341, 342, 343
  del software, 376
  mediciones de la, 377
Confianza, 60
Conformidad, 341
Conjunto de tareas, 29
  identificación de un, 30
Consistencia
  de la interfaz, 268-269
  de webapps, 320
  en interfaz de webapp, 285-286
Construcción(es), 13
  basada en componentes, 5-6
  de la interfaz, 272
  estructuradas, 253
  principios de, 94-96
Contenido
  acoplamiento de, 244
  de autor, 210
  de la pantalla, 277-278
  diseño del, en el nivel de componentes, 251-252
  modelo 3C, 261
  pruebas de, 457-458
  sensible, 10
Contexto, modelo 3C, 261
Control
  acoplamiento del, 245
  de acceso, 306, 513
  de calidad, 351
  de cambio, 511-513
  de la sincronización, 513
  de versión, 510-511
  especificación de, 162-163
  para el usuario, 266-267
Controlabilidad, 412
Controlador, 329
Controlador de la vista del modelo (CVM), 328-329
Convergencia, 183-184
Cookies, 462
Coordinación, 84
Corrección, 342
Cosas, 143
Costo
  de la calidad, 346-348
  de las revisiones, 359
CPM; *véase* Método de ruta crítica
CRC; *véase* Modelo clase-responsabilidad-colaborador
Crítica de las arquitecturas candidatas, 223
Cronograma, 629-631
CSPEC; *véase* Especificación de control
CVM; *véase* Controlador de la vista del modelo

D

DA; *véase* Descripción arquitectónica
DAS; *véase* Desarrollo adaptativo de software
Datos
  abstracción de, 190
  acoplamiento de, 245
  atributos de los, 140
  diseño de los, 93
  estándar, 260
  modelado de, 142
  modelo de, 139
  modelo de flujo de, 159-162
  objeto de, 139-140, 141
  patrones de, 297
  relaciones de, 141
  vista de, abstractos (VDA), 334
DCA; *véase* Diagrama de contexto arquitectónico
Decisión(es), 87
  arquitectónicas, 209, 210
  hacer/comprar, 614-616
  tabla de, 254-255
Defecto, 355
  amplificación del, 356
Densidad del error, 358
Dependencias, 154, 224, 243
Depuración, 404-408
DER; *véase* Diagrama entidad-relación
Desarrollo, 49
  adaptativo de software (DAS), 68-69
  colaborativo, 707-708
  de casos de uso, 113-117
  de software orientado a aspectos, 44-45
  impulsado por las características (DIC), 72-73
Desarrollo esbelto de software (DES), 73-74
Descomponibilidad, 412
Descomposición, enfoque de, 599-607
Descripción
  arquitectónica (DA), 208-209
  de los estilos o patrones de arquitectura, 222
Desempeño, calidad del, 341
Despliegue, 13
  de la función de calidad (DFC), 111
  principios de, 96-97
DFC; *véase* Despliegue de la función de calidad
DFD; *véase* Diagrama de flujo de datos
Diagrama(s)
  de actividad UML, 137-138, 253, 735-737
  de canal de UML, 138
  de clase, 725-729
  de colaboración, 247
  de contexto, 159
  de contexto arquitectónico (DCA), 217
  de despliegue, 179, 251
  de estado, 118-119, 737-740
  de flujo, 253
  de flujo de datos (DFD), 159
  de implementación, 729
  de secuencia, 168, 732-734
  de uso de caso, 730-732
  entidad-relación (DER), 139, 142
  para clases de análisis, 167-168
Dibujo, 87
DIC; *véase* Desarrollo impulsado por las características
Diseño, 208
  abstracto de la interfaz, 333-334
  arquitectónico, 217-221
  calidad del, 339
  características del buen, 186
  de alto nivel, 49
  de la arquitectura, 184-185, 208
  de la interfaz, 185
  de la interfaz del usuario, 278-284
  de la navegación para el MDHOO, 333
  de los datos, 93, 184
  de navegación, 329-331
  en el nivel de componente, 185, 201-202, 234-262
  estructurado, 225
  granularidad, 314
  lineamientos para el, 187
  modelado del, 92-94
  notación gráfica del, 253-254
  notación tabular del, 254-255
  para la reutilización (DPR), 260
  principios de, 183
  XP, 63-64
Diseño basado en patrones
  contexto, 301-302
  errores comunes en el, 305-306
  forma de pensar, 302-303
  patrones arquitectónicos, 306-308
  tabla organizadora de patrones, 305
  tareas del, 303-305
Diseño de componentes, 234
  aplicado a un sistema orientado a objetos, 246-251
  componente, 235
  construcciones lógicas, 252-253
  estándares basados en componentes, 239
  interfaz UML, 247
  lineamientos, 242-243
  naturaleza del, 246
  orientación a objetos, 235-236
  para webapps, 251-252
  principios básicos, 239-242
  rediseño, 251
  visión del proceso, 239
  visión tradicional, 236-238, 252-256
  y clases, 239
Diseño de software, 183, 206
  abstracción, 189-190
  arquitectura del software, 190-191
  aspectos, 194

basado en patrones, 301-302
calidad, 185
características de las clases de, 196-197
clases de, 196-197
división de problemas, 191
evolución del, 188
flujo de la información, 184-185
independencia funcional, 193
manifiesto del, 183
modularidad, 191-192
ocultamiento de información, 192-193
orientado a objeto, 196
patrón de diseño, 191
rediseño, 195
refinamiento stepwise, 194
tareas generales, 189
ubicación en la ingeniería de software, 184
Diseño de webapps
arquitectura del contenido, 326-328
atractivo visual, 321
calidad del, 318-320
compatibilidad, 321
consistencia, 320
controlador de la vista del modelo (CVM), 328-329
controlador, 329
diseño gráfico, 324
disponibilidad, 318
distribución de la pantalla, 323
en el nivel de componentes, 331-332
escalabilidad, 319
estética, 323-324
estructuras, 326-328
evaluación de la calidad, 319-320
íconos gráficos, 322
identidad, 320-321
imágenes, 322
interfaz, 321-323
lista de revisión, 319
MDHOO, 332-334
menús de navegación, 322
metas para el, 320-321
modelo, 328
navegabilidad, 321
objetos de contenido, 324-325
oportunidad de mercado, 319
pirámide del, 321
robustez, 321
seguridad, 318
semántica de la navegación, 329-330
simplicidad, 320
sintaxis de navegación, 330-331
tutoriales, 326
vista, 328-329
Disparador (o *trigger*), 136, 166
Disponibilidad, 9
de las webapps, 318
del software, 377
Dispositivos, 210
Distribución, 307
de la pantalla de webapps, 323
Diversificación, 183-184
Documentación, 431-432
Dogma, 84, 91
Dominio(s)
análisis del, 129-130
de aplicación del software, 6-8
de aplicación específica, 129
de información, 91
mantenimiento del, funcional, 253
DPR; *véase* Diseño para la reutilización
DSOA; *véase* Desarrollo de software orientado a aspectos
Durabilidad, 341

E

Ecuación de software, 610-611
Eficiencia, 342, 343, 344
de remoción de defecto (ERD), 583, 584
en interfaz de webapp, 286
EIS; *véase* Entorno de ingeniería de software
Elaboración, 47, 103, 194
Encapsulamiento, 150
Encriptado, 471
Enfoque
común, 60
de descomposición, 599-607
Enlaces, 415, 423-424
Entidades externas, 143, 160
Entorno de ingeniería de software (EIS), 598, 712
Entrevistas, 272
Equipo, 84
ágil, 561
cuajado o tóxico, 560
estructura de, 558-559
líderes, 557-558
ERD; *véase* Eficiencia de remoción de defecto
Error, 355
corrección del, 408
densidad del, 358
ERS; *véase* Especificación de requerimientos de software
Escalabilidad, 319
Escenario(s)
de investigación, 222
de uso, 112
elementos basados en el, 118
primarios, 134
pruebas basadas en, 445-446
secundarios, 134-135
Escuchar, 86
Esfuerzo
distribución de, 625-626
prueba de, 402-403
pruebas para webapps de, 473
y personal, 624-625
Especificación
de control (CSPEC), 162-163
de requerimientos de software (ERS), 104
del proceso (PSPEC), 163-165
Especulación, 69
Espíritu ágil, 59
Estabilidad, 412
Estándar
CMMI, 32
Unicode, 284
Estereotipo, 154
Estética, 10, 341
de webapps, 323-324
Estilos arquitectónicos, 211-217
control, 216
datos, 216-217
evaluación del, 216-217
patrones, 215-216
taxonomía de, 213-215
Estimación(es), 89, 594-595
basada en problema, 600-601
basada en proceso, 604-605
con casos de uso, 605-607
decisiones de, 349
del proyecto de software, 598-599
empírica, 608-611
LOC, 601-602
modelo de, 608-609
modelo COCOMO II, 609-610
para proyectos ágiles, 612-613
para software OO, 611-612
para webapps, 613
PF, 602-604
reconciliación de, 607
Estrategia(s)
de depuración, 406-408
de desarrollo incremental, 58
de prueba de software, 386-387
de pruebas OO, 441-442
Estructura(s), 143, 299
arquitectónicas canónicas, 212
compuestas, 328
de caja, 479
de concurrencia, 212
de control, 420-422
de desarrollo, 212
de implementación, 212
de las webapps, 454
de malla, 327
de red, 328
del proceso, 12
física, 212
funcional, 212
jerárquicas, 327
lineales, 326
profunda, 446-447
superficial, 446
Evaluación
costos de, 346
de la calidad, 319-320
de la factibilidad, 65
de la interfaz de usuario, 290-292
de los atributos de calidad, 222
del programa y la técnica de revisión (PERT), 629
del riesgo, 84
Evento(s), 143
común, 166
Evolución
continua, 10
del software, 655-656

Exactitud, 583
Excepción, 135

F

Facilitador, 87
Factores de ajuste de valor (FAV), 532
Falla, 355, 376
  costos de, 346-348
Fallas en el tiempo (FET), 377
FdN; *véase* Formas de navegar
FET; *véase* Fallas en el tiempo
Firewall, 471
Flexibilidad, 342
  en interfaz de webapp, 286
Flujo
  de trabajo, 33, 48, 276, 289-290
  de transformación, 225
  del procesamiento, 248
  elementos orientados al, 120
  trayectoria de, 225
Formalidad de las revisiones técnicas, 359-361
Formas de navegar (FdN), 330
Formato
  de caso de uso, 115-116
  de patrón de diseño, 300
  del modelo del diseño, 203
Formulación, 528
Fuente abierta, 7
Funcionalidad, 183, 188, 343
  de las webapps, 454

G

GAO/ATOCS; *véase* Arquitectura de intercambio de objetos comunes solicitados
Género arquitectónico, 209-211
Gobierno, 211
GPI; *véase* Grupo de prueba independiente
Gráfico(s), 423
  de flujo o de programa, 415-416
  de Gantt, 629
Granularidad, 89
Grupo de prueba independiente (GPI), 385-386
Guardia, 167
GUI; *véase* Interfaz gráfica del usuario
Guiones CGI, 462

H

Hardware
  sustitución del, 5
  tasa de falla del, 4
Herramientas de software
  administración de contenido, 519-520
  administración de la calidad del software, 380
  administración de proyecto, 568
  administración del proceso, 45
  análisis estructurado, 165
  calendarización del proyecto, 630
  depuración, 407
  desarrollo de casos de uso, 117
  diseño arquitectónico, 221
  diseño de casos de prueba, 428
  enfoque de métricas, 584-585
  estimación de esfuerzo y costo, 614
  gestión del cambio, 521
  ingeniería de los requerimientos, 106
  ingeniería de software, 12
  ingeniería inversa, 667
  interfaz del usuario, 284
  ISBC, 261
  lenguajes de descripción arquitectónica, 224
  manejo de riesgo, 651
  métodos formales, 498
  métricas de producto, 550
  métricas del proyecto y del proceso, 582
  métricas técnicas para webapps, 547
  modelación de análisis con UML, 169
  modelado de datos, 142
  modelado del proceso, 51
  para el proceso ágil, 76-77
  para tendencias blandas, 712-714
  planeación y administración de pruebas, 403
  prueba de aplicaciones web, 474
  reestructuración de software, 668-669
  RPE, 660
  Sistema de Versiones Concurrentes (SVC), 511
  tecnología del proceso, 50-51
  tendencias, 711-714
Historias del usuario, 61
Hitos definidos, 624
HTML dinámico, 462

I

ICOA; *véase* Ingeniería de componentes orientada a aspectos
ICS; *véase* Ítems de configuración del software
Identidad de webapps, 320-321
Identificación de la sensibilidad, 223
Idiomas, 298
Imágenes, 322
Implementación del MDHOO, 334
Importancia del software, 718
IMS; *véase* Índice de madurez de software
Incrementos de software, 58
Indagación, 103, 132
Indicador, 527
Índice de madurez de software (IMS), 550
Información
  continuidad del flujo de, 160
  dominio de, 91
  tecnología de la, 719
  transferencia de, 85
Ingeniería
  concurrente, 40
  de componentes orientada a aspectos, 44-45
  hacia adelante, 669-671
  inversa, 664-667
Ingeniería de requerimientos, 102-106, 127
  colaboración, 107-108
  compuesta, 139-140
  bases, 106
  emergentes, 708-709
  indagación o recabación, 109-110
  miniespecificaciones, 110
  múltiples puntos de vista, 107
  participantes, 106-107
  preguntas, 108
  puntos de prioridad, 107
Ingeniería de software, 10
  asistida por computadora, 12
  basada en componentes (ISBC), 257
  de quirófano, 44
  de salas limpias, 388
  definición de, 11
  del dominio, 257
  diseño de software, 184-185
  entornos de, 712
  etapa de construcción, 184
  ética en la, 721-722
  fundamento de la, 11-12
  herramientas de la, 12
  impulsado por modelo, 709-710
  impulsado por pruebas, 710-711
  métodos de la, 12
  práctica de la, 15-18
  principios de la, 16-18
  principios fundamentales, 83-86
  proceso de, 11-12
  realidades de la, 10-11
  tendencias, 697-698
Ingeniería del software de cuarto limpio
  certificación, 487
  diseño, 483-485
  especificaciones, 480-483
  estrategia, 479-480
  pruebas, 485-487
Inmediatez, 10
Inmerpresencia, 211
Integración del Modelo de Madurez de Capacidades (CMMI), 685-688
Integridad, 342, 583
Inteligencia
  ambiental (amI), 701
  artificial, 210
Interacción flexible, 266
Interdependencia, 623
Interfaz, 93, 242-243
  diseño abstracto de la, 333-334
  para una webapp, 284-290
  protocolos de, 260
  prueba de mecanismo de, 461-462
Interfaz del usuario
  accesibilidad, 283
  ambiente de trabajo, 278
  análisis de la, 269, 271, 272-278
  análisis de la tarea, 271, 273-274

análisis del mercado, 273
casos de uso, 273-274
comandos escritos, 283
consistencia de la, 268-269
construcción de la, 272
contenido de la pantalla, 277-278
control al usuario, 266-267
definición de objetos de la, 279-280
diseño de la, 278-284
elaboración de la tarea, 275
elaboración del objeto, 275-276
entrevistas, 272
estándar Unicode, 284
evaluación de la, 290-292
gráfica (GUI), 266
herramientas de ayuda, 282
información de apoyo, 273
información de ventas, 272
internacionalización, 283-284
leyendas de menú, 283
memorización, 267-268
mensajes de error, 282-283
meta del diseño de la, 272
modelos del diseño de la, 269-271
patrones de diseño de la, 310-313
principios de diseño de la, 266-269
proceso de análisis y diseño de la, 271-272
prueba de, 460-465
tiempo de respuesta, 281-282
validación de la, 272
Interfaz para una webapp
aprendizaje, 287
autonomía controlada, 286
características fundamentales, 285
comunicación, 285
concentración, 286
consistencia, 285-286
eficiencia, 286
flexibilidad, 286
flujo de trabajo, 289-290
integridad de los productos, 287
legibilidad, 287
Ley de Fitt, 286
lineamientos prácticos, 288-289
metáforas, 287
navegación visible, 287
objetos de la interfaz, 286
previsión, 285
reducción de la latencia, 287
seguimiento del estado, 287
Interoperabilidad, 342
de las webapps, 454
Interpretación, 528
ISBC; *véase* Ingeniería de software basada en componentes
ISO9001:2000 para software, 32, 379
Ítems de configuración del software (ICS), 502, 505
IXP; *véase* XP industrial

J

Juegos, 210

K

Kit de Desarrollo Bean (KDB), 259

L

LDA; *véase* Lenguajes de descripción arquitectónica
LDP; *véase* Lenguaje de diseño del programa
Legibilidad en interfaz de webapp, 287
Lenguaje(s)
de descripción arquitectónica (LDA), 191, 224-225
de diseño del programa (LDP), 255-256
de especificación Z, 495-497
de restricción de objeto, 492-495, 740-741
Ley de Fitt, 286
Leyendas de menú, 283
Líneas de código (LOC), 575
Listas de verificación para RT, 362
LOC; *véase* Líneas de código
Lógica difusa, 600
Lugares, 143

M

MA; *véase* Modelado ágil
Maestro Scrum, 70
Mantenibilidad, 188, 342, 343
Mantenimiento, 583
del dominio funcional, 253
del software, 656-657
Mapas del sitio, 331, 468
Mapeo
arquitectónico, 225
de transformación, 225-231
Marcas de página, 467
Marcos y framesets, 467
Matriz de grafo, 420
MCO; *véase* Modelo de componentes de objetos
MDHOO; *véase* Método de Diseño de Hipermedios Orientado a Objetos
MDSD; *véase* Método de desarrollo de sistemas dinámicos
Mecanismos de interacción, 265-266
Medición, 14, 526, 527
del software, 575-582
Médicos, 211
Medida(s), 527
directas, 575
Mejoramiento del proceso de software (MPS), 677-678
actividad de selección y justificación, 682-683
CMM de personal, 688-689
CMMI, 685-688
educación y capacitación, 682
evaluación, 683-684
factores de éxito cruciales, 685
factores de riesgo, 684
futuros marcos conceptuales, 692-693
instalación, 683
marcos conceptuales, 689-691
migración de proceso, 683
modelo de madurez, 679-680
proceso, 680
rendimiento sobre inversión, 691-692
valoración, 681-682
Memorización, 267-268
Mensajes de error, 282-283
Metáforas, 287
Metas para el diseño de webapps, 320-321
Método(s)
de desarrollo de sistemas dinámicos (MDSD), 71
de Diseño de Hipermedios Orientado a Objetos (MDHOO), 332-334
de ingeniería web, 515
de la ingeniería de software, 12, 350
de la negociación para la arquitectura (ATAM), 222-223
de prueba OO, 442-447
de ruta crítica (CPM), 628, 629
formales, 487-497
o servicios, 142
Métrica(s), 527
CK, 539-540
de acoplamiento, 543
de cohesión, 543
de complejidad, 544
de contenido, 546-547
de diseño de interfaz de usuario, 545
de diseño en el nivel de componente, 542-544
de Halstead aplicadas, 549
de interfaz, 545-546
de las revisiones técnicas, 357-359
de navegación, 547
del diseño arquitectónico, 535-537
estéticas, 546
MOOD, 541-542
OO, 542
orientadas a operación, 544
para código fuente, 547-548
para diseño orientado a objetos, 537-539
para el mantenimiento, 550
para pruebas, 548-550
para pruebas orientadas a objetos, 549-550
para webapps, 545-547
Métricas de proceso, 572-574
Métricas de producto
basada en funciones, 531-534
conjunto de atributos, 530-531
de punto de función (PF), 531-532
marco conceptual, 527-531
medida, medición y métrica, 527
para valorar la calidad, 534-535
paradigma Meta/Pregunta/Métrica (MPM), 529-534
principios de medición, 528-529
Métricas de proyecto, 572, 574

Métricas de software, 575
de proyecto webapp, 580-582
en una organización pequeña, 587-588
medición de la calidad, 583-584
orientadas a caso de uso, 580
orientadas a función, 577
orientadas a tamaño, 576-577
para proyectos OO, 579-580
relación entre LOC y PF, 577-579
Militares, 211
Mitos del software, 18-20
MMMR; *véase* Plan de mitigación, monitoreo y manejo de riesgo
Modalidad, 142
Modelado, 13, 75
ágil (MA), 74-75
concurrente, 41-42
de datos, 139-142
de estado finito, 424-425
de flujo de datos, 425
de flujo de transacción, 424
de la navegación para webapps, 180
de los requerimientos, 92
de temporización, 425
del análisis, 127
del diseño, 92-94
del flujo de control, 162
Modelado basado en clases
asociaciones, 153-154
atributos, 145-146
clase-responsabilidad-colaborador (CRC), 148-152
clases, 149
clases de análisis, 143
colaboraciones, 151-152
colaboradores, 149
dependencias, 154
estereotipo, 154
multiplicidad, 153-154
operaciones, 146-148
responsabilidades, 148, 149-151
Modelado de los requerimientos, 92
análisis estructurado, 130, 158
basado en clases, 142-155
basado en escenarios, 131-137
desarrollo de casos de uso, 113-117
elemento del, 131
enfoques, 130-131
entrada del, 174-175
grado de profundidad, 174
modelo de comportamiento, 165-169
orientado al flujo, 159-165
para webapps, 174-180
patrones para el, 169-174
salida del, 175-176
Modelo(s), 93, 328
3c, 260-262
AOO y DOO, 439
clase-responsabilidad-colaborador (CRC), 148-152, 439-441
COCOMO II, 609-610
de amplificación del defecto, 356-357
de certificación, 487
de componente, 487
de componentes de objetos (MCO), 259, 429
de comportamiento, 165-169
de configuración para webapps, 179
de contenido para webapps, 176-177
de estimación, 608
de flujo de control, 162
de flujo de datos, 159-162
de implementación, 270
de interacción para webapps, 177-178
de madurez, 679-680
de Madurez de Capacidad de Personal, 688-689
de madurez de la capacidad (CMM), 32, 50, 555
de marco, 190
de muestreo, 487
de referencia para revisiones técnicas, 360
de requerimientos, 90, 93, 118-121, 126
de usuario, 269-270
del análisis, 90, 117
del diseño, 270
del proceso, 191
dinámicos, 191
estructurales, 190
funcional para webapps, 178-179
funcionales, 191
mental, 270
objeto de, 328
orientados a objetos, 439-441
prueba basada en (PBM), 429
Modelo del diseño
arquitectura de datos, 199
dimensión de la abstracción, 197
dimensión del proceso, 197
diseño de datos, 199
diseño de la arquitectura, 199
diseño de la interfaz, 199-201
diseño de la usabilidad, 200
diseño del despliegue, 202-203
diseño en el nivel de los componentes, 201-202
formato descriptor, 203
interfaz, 200
realización, 201
Modelos del proceso ágil, 14
desarrollo adaptativo de software (DAS), 68-69
desarrollo esbelto de software (DES), 73-74
desarrollo impulsado por las características (DIC), 72-73
familia Cristal, 72
método de desarrollo de sistemas dinámicos (MDSD), 71
modelado ágil (MA), 74-75
proceso unificado ágil (PUA), 75-76
programación extrema (XP), 61-67
Scrum, 69-71
Modelos del proceso prescriptivo, 14, 33
concurrente, 40-41
desarrollo basado en componentes, 43
especializado, 43-45
evolutivos, 36-40
flujo de trabajo, 33
incremental, 35-36
modelo de la cascada, 34-35
modelo de métodos formales, 44
modelo en V, 34
modelo espiral, 39
proceso del equipo de software, 49-50
proceso personal del software, 48-49
proceso unificado, 45-48
Modularidad, 85-86
Motores de búsqueda, 468
MPS; *véase* Mejoramiento del proceso de software
Multiplicidad, 153-154

N

Navegabilidad de webapps, 321
Navegación
diseño de, 329-331
diseño de la, para el MDHOO, 333
nodos de (NN), 330
semántica de la, 329-330
sintaxis de, 330-331
unidades semánticas de (USN, 330)
visible, 287
Negligencia, 348-349
Negociación, 87, 103-104, 121-122
NN; *véase* Nodos de navegación
Nodo(s)
de gráfico de flujo, 415
de navegación (NN), 330
ponderados, 423
predicado, 416
Notación matemática, 490-492

O

Objetivo en contexto, 135-136
Objeto(s), 744
agregado, 509-510
algoritmos, 745
análisis orientado a, 131
básico, 509
blanco, 279
clase orientada a, 141
de aplicación, 279
de contenido de webapps, 324-325
de datos, 141
de interfaz de webapp, 286
fuente, 279
orientación a, 235
Observabilidad, 412
Obtención de los requerimientos y restricciones, 222
Ocurrencias, 143
Operación *tick ()*, 307
Operatividad, 412
Oportunidad de mercado, 319
Organización propia, 60-61
Orientación a objetos (OO), 235
atributos, 745

calendarización para un proyecto, 632-633
casos de prueba, 444-445
clase, 745
clases de diseño, 747-749
clases independientes, 398
diseño de casos de prueba, 443
estimación para software, 611-612
estrategias de prueba, 441-442
herencia, 746
jerarquía de clase, 444-445
mensajes, 746
métodos de prueba, 442-447
métricas, 542
métricas del diseño, 549-550
métricas para proyectos, 579-580
modelos de análisis y de diseño, 438-441
objetos, 744
polimorfismo, 747
prueba basada en hebra, 398
prueba basada en uso, 398
prueba de grupo, 398
pruebas de integración de los sistemas, 398, 442
pruebas de unidad en, 397-398, 441
pruebas de validación en, 442
pruebas para el software, 397-398
Outsourcing, 616

P

PAC; *véase* Principio Abierto-Cerrado
Paradigma Meta/Pregunta/Métrica (MPM), 529
Partición de equivalencia, 466
Participantes, 106-107, 557
y la planeación, 89
Patrón(es), 86, 295
arquitectónicos, 306-308
de análisis, 120-121, 169-170
de diseño, 191, 296-301
de la interfaz de usuario, 280-281
de. prueba, 433-434
del proceso, 29-31
diseño basado en, 301-302
para el modelado de requerimientos, 169-174
Patrones de diseño, 191, 296
arquitectónicos, 297
conductuales, 299
creacionales, 298
de componentes, 297
de datos, 297
de interfaz de usuario, 310-313
de webapp, 297-298, 313-314
en el nivel de componentes, 308-310
estructurales, 299
formato simplificado de, 300
generativos, 297
granularidad, 314
idiomas, 298
lenguaje de, 300-301
nueva forma de pensar, 302
repositorio de, 301, 307-308, 314
sistema de fuerzas, 296
tabla organizadora de, 305
PBM; *véase* Prueba basada en modelo
PCC; *véase* Principio de cierre común
PCU; *véase* Puntos de caso de uso
People-CMM; *véase* Modelo de madurez de capacidades del personal
PER; *véase* Principio de equivalencia de la liberación de la reutilización
Percepción, 341
Perfil operativo, 431
Persistencia, 307
Personal, 555, 556-562
y esfuerzo, 624-625
PES; *véase* Proceso del equipo de software
Pestañas, 331
PF; *véase* Punto de función
PID; *véase* Principio de Inversión de la Dependencia
Pirámide del diseño de webapps, 321
Plan
de ACS, 379-380
de mitigación, monitoreo y manejo de riesgo (MMMR), 651-652
Planeación, 13, 49
iterativa, 89
principios de la, 88-90
XP, 62-63
Planificación de proyecto, 593-594
conjunto de tareas, 595
estimación, 594-595
recursos, 596-598
Plantillas de programa, 260
Plataformas, 211
POA; *véase* Programación orientada a aspectos
Pocas vitales, 374
Política del desarrollo ágil, 59
Portabilidad, 342, 343
Post mortem, 49
PPS; *véase* Proceso personal del software
Práctica, 82-83
eficaces respecto de los requerimientos, 101
esencia de la, 15-16
principios fundamentales de la, 85-86
PRC; *véase* Proceso de la reutilización común
Precondición, 136
Preguntas libres del contexto, 108
Preparación y producción del producto del trabajo, 14
Prevención, costos de, 346
Previsión, 285
Principio(s), 16
Abierto-Cerrado (PAC), 239-2401
adicionales de agrupamiento, 241-242
de agilidad, 58-59
de cierre común (PCC), 242
de codificación, 94-95
de comunicación, 86-88
de construcción, 94-96
de despliegue, 96-97
de diseño de la interfaz, 266-269
de equivalencia de la liberación de la reutilización (PER), 241
de Inversión de la Dependencia (PID), 241
de la ingeniería de software, 16-18, 83-86
de la planeación, 88-90
de la práctica, 85-86
de la prueba, 95-96
de la reutilización común (PRC), 242
de modelado, 90-94
de preparación, 94
de programación, 94-95
de segregación de la interfaz (PSI), 241
de sustitución de Liskov (PSL), 241
de validación, 95
del modelado del diseño, 92-94
del proceso del software, 84
W5HH, 567
Problemas, división de, 191
Proceso, 27, 556
ágil, 14
de análisis y diseño de la interfaz, 271-272
de ingeniería de software, 11-12
del equipo de software, 49-50
del software, 12-14
descomposición del, 564, 565
dimensión del, 197
dualidad del, 51-52
estructura del, 12
flujo del, 28-29
fusión de producto y, 564
modelo general del, 27-31
patrón de, 29-30
personal del software, 48-49
prescriptivo, 14
tendencias, 705-706
unificado, 45-48
unificado ágil (PUA), 75-76
XP, 62-65
Proceso del software, 12-14, 564
enfoques para la evaluación y mejora del, 32
métricas, 585-587
principios fundamentales del, 84
Proceso unificado
concepción, 46-47
construcción, 47-48
elaboración, 47
historia, 46
producción, 48
transición, 48
Productor, 363
Producto(s), 555-556, 562-563
del trabajo, 84, 112-113
dualidad del, 51-52
fundamental, 35
útil, 340
Programación
decisiones de, 349-350
orientada a aspectos, 44
por pares, 361-362
principios de, 94-95

Programación estructurada, 188, 253
Programación extrema, (XP), 61
  codificación, 64
  desventajas, 66-67
  diseño, 63-64
  industrial, 61, 65-66
  planeación, 62-63
  pruebas, 64-65
  valores, 61-62
Programación por parejas, 64
Propiedades de la arquitectura del software, 190
Protocolos de interfaz, 260
Prototipos, 37-38
Proyecto, 556, 566-567
  de software, 626-627
Prueba(s)
  alfa, 400
  análisis de valor de frontera, 425-426
  atributos, 412-413
  basada en escenario, 445-446
  basada en hebra, 398
  basada en modelo (PBM), 429
  basada en uso, 398
  beta, 400
  casos de, 418-419
  con módulos atómicos, 393-394
  construcciones o builds, 394
  criterios de terminación, 388
  de aceptación del cliente, 400
  de aceptación XP, 65
  de arquitecturas cliente-servidor, 430-431
  de arreglo ortogonal, 426-428
  de base de datos, 430-431
  de caja blanca, 414
  de caja de vidrio, 414
  de caja negra, 414, 423-428
  de comportamiento, 423, 433
  de comunicación de red, 431
  de condición, 421
  de configuración, 403
  de despliegue, 403-404
  de esfuerzo, 402-403
  de flujo de datos, 421
  de función de aplicación, 430
  de grupo, 398
  de GUI, 430
  de humo, 395-396
  de integración, 391-397
  de la estructura de control, 420-422
  de recuperación, 402
  de regresión, 394-395
  de rendimiento, 403
  de ruta o trayectoria básica, 414-420
  de seguridad, 402
  de sensibilidad, 402
  de servidor, 430
  de sistema, 433
  de tareas, 432
  de transacción, 431
  de unidad, 389-391
  de validación, 399-401
  definición ampliada de, 438
  del sistema, 401
  depuración
  documentación, 431-432
  eliminación de la causa, 407
  enfoque estratégico, 384-389
  especificación de pruebas, 396
  estrategia de, 386-387
  exhaustivas, 414
  factor humano, 408
  fundamentos, 412-413
  grupo de prueba independiente (GPI), 385-386
  integración primero en profundidad, 392-393
  intertarea, 433
  matriz de grafo, 420
  métodos basados en gráficos, 423-425
  módulo crítico, 396
  organización de, 385-386
  para el software OO, 397-398
  para sistemas en tiempo real, 432-433
  para software convencional, 389-397
  para webapps, 398-399
  partición de equivalencia, 425
  patrones de, 433-434
  principios de la, 95-96
  proceso de depuración, 404-405
  prueba sándwich, 396
  seguimiento hacia atrás, 407
  stubs, 393
  TMR, 402
  unitaria, 61
Pruebas de aplicaciones OO, 397-398
  aleatoria, 447
  basada en fallo, 444
  casos, 443
  de bucle, 421-422
  de clase múltiple, 449-450
  de integración, 398, 441-442
  de partición, 448
  de unidad, 397-398, 441
  de validación, 442
  del sistema, 441
  modelos AOO y DOO, 439
  modelos de comportamiento, 450-451
Pruebas para webapps, 398-399
  capas de interacción, 459-460
  de carga, 472-473
  de compatibilidad, 465
  de configuración, 469-470
  de contenido, 457-458
  de error forzado, 466-467
  de esfuerzo, 473
  de interfaz de usuario, 460-465
  de la base de datos, 458-459
  de la semántica de la interfaz, 463
  de la semántica de navegación, 468-469
  de mecanismo de interfaz, 461-462
  de rendimiento, 471-473
  de rutas, 466
  de seguridad, 470-471
  de sintaxis de navegación, 467-468
  de usabilidad, 463-464
  dimensiones de calidad, 454
  en el nivel de componente, 466-467
  errores, 455
  estrategia para, 455-456
  planificación de, 456
  proceso de, 456-457
PSI; *véase* Principio de segregación de la interfaz
PSL; *véase* Principio de sustitución de Liskov
PSPEC; *véase* Especificación del proceso
PUA; *véase* Proceso unificado ágil
Punto de función (PF), 577
  dimensionamiento del, 600
Puntos
  de caso de uso (PCU), 580
  de prioridad, 107
  de referencia puntuales, 39-40
  de vista múltiples, 107

R

Recolección, 528
Recursos
  ambientales, 598
  de software reutilizables, 597-598
  humanos, 596-597
Red de tareas, 628
Redes
  arquitecturas sencillas de, 7
  uso intensivo de, 9
Redirecciones, 467
Rediseño, 64
Reducción de la latencia, 287
Reestructuración de software, 668-669
Regiones, 416
Reglas doradas, 265-269
Reingeniería, 658
  análisis costo-beneficio para, 671-672
  de procesos de empresa (RPE), 658-660
  de software, 661-664
Relación(es), 141
  de dependencia, 154
Rendimiento, 9, 188
  de las webapps, 454
Reporte, 522-523
Repositorio, 260
  ACS, 506-508
  de hipermedios, 314
  de patrones de diseño, 301, 307-308, 314
Requerimientos
  análisis de los, 127-130
  de los principios de modelado, 91-92
  elementos del modelo de, 118-120
  emergentes, 701-702
  indagación o recabación de los, 109-110
  ingeniería de, 102-106
  modelado de los, 92, 130, 158-180
  negociación de los, 121-122
  tabla de, 112
  tipos de, 111
  validación de los, 122-123

Resistencia, 183
Respeto mutuo, 60
Responsabilidad(es), 148, 149-151, 348
  definidas, 623
Resultados definidos, 624
Retraso, 70
Retroalimentación, 61, 91, 528
Retrospectiva, 65
Reunión casual, 361
Reuniones Scrum, 70
Reusabilidad, 342
Reutilización
  administración de la, 14
  análisis para la, 259
  biblioteca de, 261-262
  diseño para la (DPR), 260
Revisión
  del diseño de alto nivel, 49
  líder de la, 363
  reunión de, 363
Revisiones técnicas (RT), 14, 105, 187
  eficacia del costo de las, 358-359
  formales (RTF), 362-366
  informales, 361-362
  lineamientos para las, 364-365
  lista de pendientes de las, 363
  listas de verificación para, 362
  métricas de las, 357-359
  modelo de referencia para, 360
  nivel de formalidad apropiado, 359-361
  objetivo principal de las, 355
  orientadas al muestreo, 365
  productor, 363
  programación por pares, 361-362
  reporte técnico formal de las, 363
  reunión casual, 361
  revisores, 363
  verificación de escritorio, 361
Riesgo(s), 89, 348
  administración del, 13, 640, 642
  componentes de, 644
  de software, 641-642
  decisiones orientadas al, 350
  evaluación del, 84
  identificación de, 642-643
  impacto de, 647-648
  manejo del, 650-651
  mitigación del, 649-650
  monitoreo de, 650
  plan de mitigación, monitoreo y manejo de (MMMR), 651-652
  proyección de, 644-645
  refinamiento del, 649
  tabla de, 645-647
  valoración del, 643-644
Riqueza, 344
Robustez, 344
  de webapps, 321
Roles, 143
RPE; *véase* Reingeniería de procesos de empresa
RT; *véase* Revisiones técnicas
RTF; *véase* Revisiones técnicas formales
Ruta independiente, 416-417

S

SCAMPI; *véase* Estándar CMM
Scripts de instalación, 97
Scrum, 69-71
SdE; *véase* Separación de entidades
Seguimiento
  del estado, 287
  y control del proyecto de software, 13
Seguridad, 10, 583
  de las webapps, 318, 454
  del software, 378
Seis Sigma, 375-376
Semántica de la navegación, 329-330
Sencillez, 90
Separación de entidades (SdE), 85
Servicio, 341
Seudocódigo; *véase* Lenguaje de diseño del programa
Simplicidad, 61, 412
  de webapps, 320
Sintaxis de navegación, 330-331
Sistema(s)
  de aseguramiento de la calidad, 378-379
  de fuerzas, 296
  en tiempo real, 432-433
  entre iguales, 217
  objetivo, 217
  operativos, 211
  pruebas del, 401-404
  subordinados, 217
  superiores, 217
Software
  ámbito del, 595-596
  aplicaciones web o webapps, 7, 9-10
  arquitectura del, 93, 190-191, 207-029
  aseguramiento de la calidad del, 14
  auditoría de configuración, 514
  bloques constructores de, 703
  calidad del, 186, 340
  características del, 4-6
  categorías de, 6-8
  comportamiento del, 92
  de aplicación, 6
  de ingeniería y ciencias, 6
  de inteligencia artificial, 7
  de línea de productos, 7
  de mundo abierto, 701
  de sistemas, 6
  definición de, 3-4
  dimensionamiento del, 599-600
  ecuación de, 610-611
  evolución del, 655-656
  fuente abierta, 703-704
  funciones del, 92
  heredado, 8-9
  importancia del, 718
  incrustado, 6-7
  mantenimiento del, 656-657
  mitos del, 18-20
  modularidad, 85-86
  muerte del, 2
  nuevos desafíos de, 7
  papel dual del, 2-3
  plan del proyecto de, 13
  preguntas acerca de, 3
  proceso del, 12-14, 26
  proceso eficaz de, 340
  punto de vista de un economista sobre el, 26
  seguimiento y control del proyecto de, 13
  soportabilidad de, 657-658
  tasa de fallas del, 4-5
Soportabilidad de software, 657-658
SPICE (ISO/IEC 15504), 32
Sprint, 69-70
Streaming, 462
Susceptibilidad de probarse, 342

T

Tabla
  de activación del proceso (TAP), 163
  de decisión, 254-255
  de la voz del cliente, 112
  organizadora de patrones, 305
TAP; *véase* Tabla de activación del proceso
Tarea(s)
  análisis de la, 271, 273-274
  casos de uso, 273-274
  elaboración de la, 275
  conjunto de, 29
  identificación de una, 30
  interfaz para webapp, 289-290
Tecnología(s)
  administración de la complejidad, 700-701
  ciclo de promoción excesiva, 698-699
  ciclo de vida de innovación, 697-697
  emergentes, 704
  evolución de las, 696-697
  gran desafío, 706-707
  tendencias, 695-696
  tendencias blandas, 699-700
  vista larga, 720-721
Tiempo
  de respuesta, 281-282
  medio al cambio (TMC), 583
  medio de reparación (TMR), 402
TMC; *véase* Tiempo medio al cambio
TMR; *véase* Tiempo medio de reparación
Toma de decisiones, 60
Transferencia de información, 85
Transporte, 211
Trayectoria de flujo, 225
Trigger; *véase* Disparador

U

UML
  diagrama de actividad, 137-138, 735-737
  diagrama de canal de, 138
  diagrama de clase, 725-729
  diagrama de comunicación, 734-735

diagrama de despliegue, 179
diagrama de estado, 737-740
diagrama de implementación, 729
diagrama de secuencia, 168, 732-734
diagrama de uso de caso, 730-732
historia, 725
interfaz, 247
modelos 137-139
Unidades organizacionales, 143
Unidades semánticas de navegación (USN), 330, 468-469
Usabilidad, 188, 269, 342, 343, 583
de las webapps, 454
USN; *véase* Unidades semánticas de navegación
Uso
de hipermedios, 9
de la abstracción, 85
Usuarios
finales, 87
frecuentes conocedores, 270
principiantes, 270
Utilidades, 211

V

Validación, 105, 384
de la interfaz, 272
de los requerimientos, 122-123
principios de, 95
verificación y (V & V), 384-385
Valor agregado, 340
Valores XP, 61-62
Variabilidad, 282
VDA; *véase* Vista de datos abstractos
Ventanas *pop-up*, 462
Ventas, información de, 272
Verificación, 384
de escritorio, 361
y validación (V & V), 384-385
Vínculo(s), 461
de navegación, 331, 467
Vista, 328-329
de datos abstractos (VDA), 334

W

Walkthroughs, 362-363
Webapp(s), 9-10
ACS para, 515-523
arquitectura de una, 328-329
calendarización de proyectos, 632-633
calidad del diseño de, 318-320
dimensiones de calidad, 454
diseño de componentes para, 251-252
diseño de la interfaz de, 321-323
diseño del contenido, 324-326
disponibilidad de las, 318
escalabilidad de las, 319
estética de la, 323
estimación para, 613
evaluación de la calidad, 319-320
funciones de la, 252
interfaz de, 284-290
metas para el diseño de, 320-321
métricas para, 545-547 580-582
modelado de la navegación para, 180
modelado de requerimientos para, 174-180
modelo de configuración para, 179
modelo de contenido para, 176-177
modelo de interacción para, 177-178
modelo funcional para, 178-179
oportunidad de mercado, 319
patrones de diseño de, 297-298, 313-314
seguridad de las, 318

X

XP; *véase* Programación extrema
XP industrial, 61, 65-66