IET Professional Applications of Computing Series 4

# UML for Systems Engineering: watching the wheels

## Second Edition

Jon Holt

IET PROFESSIONAL APPLICATIONS OF COMPUTING SERIES 4

Series Editors: Professor P. Thomas
Dr R. Macredie
J. Smith

# UML for Systems Engineering: watching the wheels

Second Edition

## Other volumes in this series:

Volume 1 **Knowledge discovery and data mining** M.A. Bramer (Editor)
Volume 3 **Troubled IT projects: prevention and turnaround** J.M. Smith
Volume 4 **UML for systems engineering: watching the wheels, 2nd edition** J. Holt
Volume 5 **Intelligent distributed video surveillance systems** S.A. Velastin and P. Remagnino (Editors)
Volume 6 **Trusted computing** C. Mitchell (Editor)

# UML for Systems Engineering: watching the wheels

## Second Edition

Jon Holt

The Institution of Engineering and Technology

Published by The Institution of Engineering and Technology, London, United Kingdom

First edition © 2004 The Institution of Electrical Engineers
Reprint with new cover © 2007 The Institution of Engineering and Technology

First published 2004
Reprinted 2007

This publication is copyright under the Berne Convention and the Universal Copyright Convention. All rights reserved. Apart from any fair dealing for the purposes of research or private study, or criticism or review, as permitted under the Copyright, Designs and Patents Act, 1988, this publication may be reproduced, stored or transmitted, in any form or by any means, only with the prior permission in writing of the publishers, or in the case of reprographic reproduction in accordance with the terms of licences issued by the Copyright Licensing Agency. Inquiries concerning reproduction outside those terms should be sent to the publishers at the undermentioned address:

The Institution of Engineering and Technology
Michael Faraday House
Six Hills Way, Stevenage
Herts, SG1 2AY, United Kingdom

www.theiet.org

While the author and the publishers believe that the information and guidance given in this work are correct, all parties must rely upon their own skill and judgement when making use of them. Neither the author nor the publishers assume any liability to anyone for any loss or damage caused by any error or omission in the work, whether such error or omission is the result of negligence or any other cause. Any and all such liability is disclaimed.

The moral rights of the author to be identified as author of this work have been asserted by him in accordance with the Copyright, Designs and Patents Act 1988.

**British Library Cataloguing in Publication Data**
Holt, Jon
UML for systems engineering, 2nd edn – (IEE professional
applications of computing series 4)
1. UML (Computer science) 2. Systems engineering
I. Title II. Institution of Electrical Engineers
003 . 3'5117

**ISBN (10 digit) 0 86341 354 4**
**ISBN (13 digit) 978-0-86341-354-4**

Typeset in India by Newgen Imaging Systems (P) Ltd
Printed in the UK by MPG Books Ltd, Bodmin, Cornwall
Reprinted in the UK by Lightning Source UK Ltd, Milton Keynes

*This book is* still *dedicated to my mother,*
*Christine Holt*

# Contents


**Acknowledgments** **xv**

**1 Systems engineering** **1**

1.1 Problems with systems 1
1.1.1 Systems 1
1.1.2 Defining systems engineering – first attempt 2
1.1.3 The need for systems engineering 2
1.1.4 Disasters and failures 3
1.1.5 Causes of problems 4
1.1.6 Defining a system 5
1.1.7 Defining systems engineering – second attempt 6
1.1.8 Summary 6
1.2 Complexity, communication and understanding 7
1.2.1 Introduction 7
1.2.2 Complexity 7
1.2.3 Minimising complexity 9
1.2.4 Communications 9
1.2.5 A good approach 12
1.3 Quality 14
1.3.1 Introduction 14
1.3.2 The need for quality 15
1.3.3 Level of responsibility 15
1.3.4 Requirements for systems professionals 16
1.3.5 Processes and quality 17
1.3.6 Standards 17
1.3.7 Problems with standards 18
1.3.8 Consequences of problems with standards 20
1.3.9 Compliance with standards 20
1.3.10 Processes 21
1.3.11 Summary 22
1.4 Conclusions 22

1.5 Further discussion 23
1.6 References 23

**2 Modelling 25**
2.1 Introduction to modelling 25
2.1.1 The importance of modelling 25
2.1.2 Why projects fail 29
2.1.3 Modelling 29
2.1.4 Defining a 'model' 30
2.1.5 Principles of modelling 31
2.2 The UML 33
2.2.1 Introduction 33
2.2.2 Aspects of the model 33
2.2.3 Views versus models 34
2.2.4 Extending the UML 35
2.2.5 Background of the UML 35
2.2.6 A nonproprietary language 39
2.3 Conclusions 39
2.4 Further discussion 40
2.5 References 40

**3 Structural modelling 43**
3.1 Introduction 43
3.2 Structural modelling using class diagrams 44
3.2.1 Modelling classes and relationships 44
3.2.2 Basic modelling 44
3.2.3 Adding more detail to classes 47
3.2.4 Adding more detail to relationships 48
3.3 Other structural diagrams 54
3.4 Conclusion 55
3.5 Further discussion 57
3.6 References 57

**4 Behavioural modelling 59**
4.1 Introduction 59
4.1.1 Behavioural modelling using state machine diagrams 60
4.1.2 Behavioural modelling – a simple example 63
4.1.3 Solving the inconsistency 67
4.2 Alternative state machine modelling 70
4.2.1 Actions and activities 70
4.2.2 Comparing the approaches 71
4.3 Other behavioural models 72
4.4 Conclusions 73

4.5 Further discussion 73
4.6 References 73

**5 The UML diagrams 75**
5.1 Introduction 75
5.1.1 Overview 75
5.1.2 Conceptual changes between UML 1.x and UML 2.0 76
5.1.3 Terminology 76
5.1.4 The structure of UML 2.0 diagrams 77
5.1.5 Comparison of the types of diagram 78
5.1.6 The UML meta-model 81
5.2 Class diagrams (structural) 82
5.2.1 Overview 82
5.2.2 Diagram elements 82
5.2.3 Example diagrams and modelling – class diagrams 84
5.2.4 Using class diagrams 88
5.3 Object diagrams 88
5.3.1 Overview 88
5.3.2 Diagram elements 89
5.3.3 Example diagrams and modelling – object diagrams 90
5.3.4 Using object diagrams 94
5.4 Composite structure diagrams (structural) 94
5.4.1 Overview 94
5.4.2 Diagram elements 95
5.4.3 Examples and modelling – composite structure diagrams 97
5.4.4 Using composite structure diagrams 100
5.5 Package diagrams (structural) 101
5.5.1 Examples of package diagrams 103
5.5.2 Using package diagrams 104
5.6 State machine diagrams (behavioural) 104
5.6.1 Overview 104
5.6.2 Diagram elements 105
5.6.3 Examples and modelling – state machine diagrams 108
5.6.4 Using state machine diagrams 112
5.7 Interaction diagrams 112
5.7.1 Introduction 112
5.7.2 Overview – communication diagrams 113
5.7.3 Diagram elements – communication diagrams 113
5.7.4 Examples and modelling – communication diagrams 115
5.7.5 Using communication diagrams 117
5.7.6 Overview – sequence diagrams 117

| | | |
|---|---|---|
| 5.7.7 | Diagram elements – sequence diagrams | 118 |
| 5.7.8 | Examples and modelling – sequence diagrams | 118 |
| 5.7.9 | Using sequence diagrams | 122 |
| 5.7.10 | Overview – interaction overview diagrams | 123 |
| 5.7.11 | Diagram elements – interaction overview diagrams | 123 |
| 5.7.12 | Examples and modelling – interaction overview diagrams | 124 |
| 5.7.13 | Using interaction overview diagrams | 126 |
| 5.7.14 | Overview – timing diagrams | 126 |
| 5.7.15 | Diagram elements – timing diagrams | 127 |
| 5.7.16 | Examples and modelling – timing diagrams | 128 |
| 5.7.17 | Using timing diagrams | 130 |
| 5.8 | Activity diagrams | 131 |
| 5.8.1 | Overview | 131 |
| 5.8.2 | Diagram elements | 131 |
| 5.8.3 | Examples and modelling | 134 |
| 5.8.4 | Using activity diagrams | 137 |
| 5.9 | Use case diagrams | 138 |
| 5.9.1 | Overview | 138 |
| 5.9.2 | Diagram elements | 139 |
| 5.9.3 | Examples and modelling | 141 |
| 5.9.4 | Using use case diagrams | 145 |
| 5.10 | Component diagrams | 145 |
| 5.10.1 | Overview | 145 |
| 5.10.2 | Diagram elements | 146 |
| 5.10.3 | Examples and modelling | 148 |
| 5.10.4 | Using component diagrams | 151 |
| 5.11 | Deployment diagrams | 152 |
| 5.11.1 | Overview | 152 |
| 5.11.2 | Diagram elements | 152 |
| 5.11.3 | Examples and modelling | 154 |
| 5.11.4 | Using deployment diagrams | 157 |
| 5.12 | Summary | 158 |
| 5.13 | Conclusions | 158 |
| 5.14 | Further discussion | 161 |
| 5.15 | References | 161 |
| **6** | **Modelling standards, processes and procedures** | **163** |
| 6.1 | Introduction | 163 |
| 6.1.1 | Introduction | 163 |
| 6.2 | Standards | 164 |
| 6.2.1 | Standards, standards, standards | 164 |
| 6.2.2 | Requirements for using standards | 166 |

6.3 Analysing standards 167
6.3.1 Aspects of modelling using the UML 167
6.3.2 Modelling example 168
6.3.3 EIA 632 Processes for engineering a system 168
6.3.4 ISO 15504 Software process improvement and capability determination (SPICE) 171
6.3.5 Comparing models 173
6.3.6 Other standards and processes 177
6.3.7 Summary 182
6.4 Defining new processes using the UML 182
6.4.1 A new procedure 182
6.4.2 Completing the model 189
6.4.3 Summary 190
6.5 Tailoring the process 191
6.6 Defining deliverables 194
6.6.1 Deliverables by type 194
6.6.2 Deliverables by process 195
6.6.3 Deliverables by structure 196
6.7 Implementing the systems engineering process using a tool 197
6.7.1 Uses for a tool 198
6.7.2 The information model 199
6.7.3 The profile definition 200
6.7.4 The implementation model 201
6.7.5 Summary 202
6.8 Conclusions 202
6.9 Further discussion 203
6.10 References 203

**7 Modelling requirements 205**
7.1 Introduction 205
7.2 Requirements engineering basics 206
7.2.1 Introduction 206
7.2.2 The requirements phase 206
7.2.3 Capturing requirements 209
7.2.4 Requirements 210
7.2.5 Stakeholders 215
7.2.6 Summary 218
7.3 Using use case diagrams (usefully) 219
7.4 Context modelling 219
7.4.1 Types of context 219
7.4.2 Practical context modelling 222
7.4.3 Summary 226
7.5 Requirements modelling 226
7.5.1 Introduction 226

7.5.2 Modelling requirements 227
7.5.3 Describing use cases 232
7.6 Modelling scenarios 236
7.6.1 Scenarios in the UML 236
7.6.2 Example scenarios 237
7.6.3 Communication versus sequence diagrams 240
7.6.4 Wrapping up scenarios 241
7.7 Documenting requirements 241
7.7.1 Overview 241
7.7.2 Populating the document 243
7.7.3 Finishing the document 244
7.8 Summary and conclusions 245
7.9 Further discussion 247
7.10 References 248

**8 Life cycle management 251**
8.1 Life cycles and life cycle models 251
8.2 Different life cycle models 253
8.2.1 Introduction 253
8.2.2 When to choose 253
8.2.3 Linear versus iterative models 253
8.2.4 The linear approach 254
8.2.5 The iterative approach 256
8.3 STUMPI – an example of life cycle management 263
8.3.1 Introduction 263
8.3.2 The STUMPI life cycle 264
8.3.3 The STUMPI life cycle model 265
8.3.4 The STUMPI process model 266
8.3.5 Phase iterations 270
8.3.6 Process behaviour 273
8.4 Project planning 273
8.4.1 Introduction 273
8.5 Summary 278
8.6 Exercise 278
8.7 References 278

**9 System architectures 279**
9.1 Introduction 279
9.1.1 Introduction to architectures 279
9.1.2 Architecture – definitions 279
9.1.3 Not an architecture 281
9.1.4 Architecture versus model 282
9.1.5 Architectural views 282

9.2 Example of architecture – Department of Defense Architectural Framework (DoDAF) 284
9.3 Example architecture – IEEE 1471 286
9.3.1 Conceptual model of architectural description 287
9.4 Common themes 291
9.5 Validation with architectures 291
9.5.1 Introduction 291
9.5.2 Generating the architecture 292
9.5.3 Using an architecture for project validation 293
9.5.4 Conclusion 295
9.6 Applying the UML 298
9.6.1 Introduction 298
9.6.2 The requirements model architecture 298
9.6.3 The analysis model architecture 299
9.6.4 The design model architecture 301
9.7 Conclusions 302
9.8 References 303

**10 Extending the UML 305**
10.1 Introduction 305
10.2 Creating a profile 306
10.2.1 Introduction 306
10.2.2 Background to process modelling example 306
10.2.3 Identifying and reading stereotypes 307
10.2.4 Declaring and defining stereotypes 308
10.2.5 Adding more knowledge 308
10.2.6 Declaring a profile 309
10.2.7 Stereotyping symbols 310
10.2.8 Summary 311
10.2.9 Introduction to constraints 312
10.2.10 Formal constraints – the object constraint language (OCL) 313
10.2.11 Introduction to tagged values 318
10.3 Conclusions 319
10.4 Further discussion 319
10.5 References 319

**11 Tools 321**
11.1 Introduction 321
11.2 CASE tools 322
11.3 Selecting a tool 322
11.3.1 What tools can do 322
11.3.2 What tools cannot do 324

11.4 A requirements model for CASE tool evaluation 325
11.4.1 Requirements model overview 325
11.4.2 Modelling capability provision 326
11.4.3 Provision of interoperability capability 329
11.4.4 Provision of other capabilities 331
11.4.5 Provision of application capability 336
11.4.6 Ensuring process compatibility 337
11.4.7 Provision of system functionality 338
11.4.8 Ensuring compatibility with the UML 339
11.4.9 Ensuring vendor's quality of service 341
11.5 Conclusions 343
11.6 Further discussion 343

**Bibliography** **345**

**Index** **351**

# Acknowledgments

'Will heaven be like Swansea?' 'Yes Tubbs, only bigger!'

*- Tubbs & Edward*

The new edition of the book finds me still in Swansea and desperately trying to think of all the people who have contributed to this edition of the book and, once more, it is a very long list. As ever, the staff at Brass Bullet Ltd must take the first nod. Secondly, I should acknowledge all of our clients – I shan't attempt to name each one as it would take too long and I would only miss people out.

A few people deserve individual mentions, such as: Ian Wilson, now a fellow-director; Simon 'there ain't no monkeys in Cap'n Chunky's' Perry, who is a great advocate of the PAPS modelling approach, Mike and Sue Rodd who set me on this road in first place, all individuals who sit on the IEE Professional Network for Systems Engineering, Ann – who made me write the bit on architectures, Paul and crew from the BSI, Richard and gang from the Rail industry and, I am sure, many, many others.

The IEE and INCOSE also deserve recognition for their commitment to systems engineering and for putting up with me for the last few years. Whilst on an IEE note, the publishing department, in particular Wendy Hiles and Sarah Kramer, who have had to put up with my rude comments and insane ravings about all things publishing.

Many people emailed me with corrections and typos in the original book – thanks for that and, hopefully, these have all been put right now.

A mention must also go to the three men who started all this: James Rumbaugh, Grady Booch & Ivar Jacobsen – I still owe you that drink.

Academically, I must thank all the good folk from the School of Engineering at the University Wales Swansea, the systems engineering department at UCL and a few choice people at Penn State University in the good ol' US of A.

Also, Paul Labbett for his original cover art which is now actually represented on the cover of the book.

Finally, writing books is clearly good for certain aspects of human biology as, once more, I find myself with another beautiful baby and the Holt clan continues to expand. Finally, as ever, my unconditional love and thanks go to my family, Rebecca, Jude, Eliza and the baby Roo.

# *Chapter 1*
# Systems engineering

if you're looking for trouble, you came to the right place

*Elvis Presley*

## 1.1 Problems with systems

### *1.1.1 Systems*

Systems form an integral part of every aspect of our lives. When most people hear the term 'systems engineering' they think of technical systems, such as an aeroplane, car or train but, in many cases, this is only the tip of the iceberg. Consider the sheer range of systems that can be classified into different types. The following list of types of systems is not intended to be exhaustive but is presented here, rather, to try to encourage the readers to expand the context of systems in their own minds.

- Large-scale, organisational-type systems, such as social and political systems.
- Living systems, such as the human body and the environment.
- Nontangible systems, such as process models and software.
- Behavioural systems, such as martial arts, crossing the road, doing the shopping and going about everyday life.
- Physical systems, such as vehicles, buildings and home appliances.

Of course, any complete system consists of a number of these systems to become a system of systems. The term 'systems engineering' then, means applying engineering to such systems, but what do we mean by engineering? A simple way to define engineering is by thinking of it as problem solving, therefore, when we talk about systems engineering, we are talking about problem solving with systems at any point in the life cycle. It would seem appropriate, therefore, at this point to try to define 'systems engineering' properly.

### 1.1.2 Defining systems engineering – first attempt

This book is concerned, primarily, with applying the Unified Modelling Language (UML) to systems engineering. It would seem appropriate, therefore, to give some definitions for systems engineering at this initial stage. However, this is not as simple as it may appear as there are many definitions from which to choose and it would be precipitous to state a definition at this stage without giving some thought to the issues involved. It is not only important to understand the needs for systems engineering, but also to understand what a system is. Therefore, deriving a definition for systems engineering will be left until later in this chapter when we have more of an idea about the scope and complexity of the field of systems engineering. Rather than a single, definitive statement of what exactly systems engineering is, this book will put forward two definitions – one technical, one practical – which should always be kept at the back of your mind while going about life, but more of this later.

### 1.1.3 The need for systems engineering

It was stated in the introduction to this chapter that an understanding of why we need systems is required, together with an understanding of what, exactly, a system is [1, 2 and 12]. This section aims to address the first of these two points, by asking the question, why do we need systems engineering? Is there a real need or is it just some academic exercise to appear knowledgeable and to sell books?

In order to understand the need for systems engineering, let us first look at some of the driving forces behind it. These include the following:

*Increased size of the global market and the global village*: As the world gets smaller and many boundaries, whether social, political or economical are removed, the need to compete in this freer world becomes more important. The vast increase in the efficiency of communications over the last few years has meant that it is now possible to trade on a worldwide scale at the touch of a button. The presence of the Internet and, more to the point, its increased use by all sections of society, means that if a product is cheaper in one country than another, customers do not have to leave the comfort of their armchairs to go there and buy it effectively. With virtual working it is now feasible to work 'closely' with people on the other side of the world, rather than being restricted to working with 'local' people. With such an increase in competition, it is important that the systems we produce are the best anyone anywhere can offer.

*Increased technological performance*: As the speed and efficiency of technology increase, the demand for such systems also increases. Customers will always want products that are smaller, more efficient and smarter than previous models and will accept no arguments as to why this is not so. It is crucial that these rapidly-changing, advanced systems are managed and understood if they are to provide the safety and reliability required of a modern-day system.

*Ubiquitous computers*: Computers are now an integral part of almost any complex system and will generally involve software of some description which is, arguably, the most complex of man's creations. Software must, therefore, be fully understood

and managed as part of a large, complex system otherwise we run the risk of another software crisis. The world of software engineering has recently undergone a watershed of understanding, which has led to the realisation that the Mickey Mouse sorcerer's apprentice approach to writing code was simply not good enough. Software is an unpredictable beast that needs to be contained by applying software engineering techniques that increase the chance of project success enormously. However, when this software is released into an integrated complex system, software engineering techniques alone are not sufficient for the increased problem space, hence the variety and range of techniques that can control the beast must be increased accordingly. Software engineering is a subset of systems engineering that limits itself to a single discipline. Clearly, there are many similarities between the two disciplines and it is important to learn lessons from any source available.

*Quality*: This must be an integral part of the whole life cycle of system development and will ensure the future of systems and their associated business [3]. One of the problems with quality is that it is often perceived as simply a 'certificate on a wall' or a 'badge with a profile drawn on it'. However, quality should be viewed as a frame of mind that should be adopted by all project personnel. Only when we think in terms of quality can we hope to produce systems that have quality that is in-built, rather than tacked on as an afterthought.

*Integration of systems*: This brings together all the points listed above. Systems engineering encompasses all areas of engineering and management and, as such, requires new levels of understanding. This introduces problems of complexity, communication between different disciplines and conditions ripe for potential misunderstandings between project personnel. All of this, naturally, should be avoided.

Finally, it must be remembered that massive losses are made each year due to project failure and disasters, many of which could have been avoided with the application of adequate systems engineering.

The points raised here are not complete but provide a brief introduction as to why people are sometimes caught unawares and can end up with a potential disaster on their hands once it is too late to do anything about it. Also, it should be borne in mind that it doesn't matter how many successful projects you run or how many perfect systems you develop as everybody will remember your only project failure over any number of successes. It only takes a single failure to occur for people to lose confidence in you, your organisation or your system and, once that happens, it can be almost impossible to put it right.

### *1.1.4 Disasters and failures*

Left unchecked in any system, problems may lead to either failure or disaster. In order to understand this, consider the following two broad groups that categorise most results of problems:

- *System disasters*: By which we mean that something going wrong results in the loss of human life, damage to the environment or damage to property. In many

cases, these are systems that have run successfully and error free for many years, sometimes decades, before disaster strikes.

- *Project failures*: By which we mean a project that is either never completed or one that was abandoned soon after its initial operation.

Although there are many reasons behind these two types of system problems, there are a number of common themes that emerge from almost all examples of them, and these will be explored in the next section.

For examples of failures and disasters, see References 4–7.

### *1.1.5 Causes of problems*

History has taught us that, whatever the nature of disaster or failure, there are a few common causes that crop up time and time again. This conclusion has been reached by many authors and may be summarised by considering three points – complexity, communications and a lack of understanding – the 'three evils of systems engineering'.

*Complexity*: This is certainly one of the main causes of system problems. This may be because the complexity of a system may have been underestimated or not taken into account at all [8]. It may also be that the complexity has been 'unmanaged', by which we mean that it may have been uncontrolled or unmeasured. In order to manage such complexity, however, it is important to have a good idea of what, exactly, complexity is – which will be discussed in some detail in a subsequent section. Mismanaging complexity will lead to a lack of understanding and bad communications.

*Communications*: A lack of communication or inefficient communication will contribute to project failure. The communication here could be between people, systems, organisations or any project stakeholder. It is important to identify the communication needs in a project and then to communicate effectively. Poor communications will lead to ambiguities, which may result in complexity and confusion that is inevitable if people cannot understand each other. Communications will be discussed in more detail later in this chapter.

*Lack of understanding*: This is a problem that may occur for any number of reasons, such as a lack of experience, not understanding the domain, not speaking the right language, underestimating complexity, and so forth. There are many aspects of a project that must be understood, but two aspects that are common failures are a lack of understanding of the problem and a lack of understanding of the solution. A lack of understanding of the problem could be due to a number of factors, including a lack of specific domain knowledge, differences in terminology or not understanding how an external system works. A lack of understanding of the solution will often be due to people choosing solutions that they have no experience of, or trying out immature technology or, in many cases, because they have caught the dreadful affliction of 'buzzword-fever' and want to use particular solutions as they are perceived as exciting, or sexy new techniques. Remember, a little knowledge can be a dangerous thing.

One point that should stand out after reading this list is that each of the problems is likely to lead to difficulties in the other two areas, or to put it another way, it is a circular problem. This means that we must try to address all of these issues in some way and keep the problems to a minimum. Half of the battle with these problems is in trying to highlight where they occur during a project. At what stage then should we concentrate on trying to spot these areas and are they more likely to occur early on during the development of a project or in the later stages?

These problems do not simply occur during the design and implementation of a project, but may appear at any phase in the project's life cycle, hence there is a need for effective control and management of a project's life cycle. The problems themselves are just as likely to happen at any point of project development, but the later they do, the more they become compounded. For example, by the design phase it is far more expensive to correct a fault than if it were found during the requirements phase. Therefore, the way in which we approach systems engineering must be applicable at any point in the project life cycle rather than simply focusing on a single phase or phase activity. This will turn out to be a fundamental requirement for systems engineering.

### *1.1.6 Defining a system*

The motivation behind systems engineering has been discussed, but what do we actually mean by a system? A common misconception is that a system is simply a product that may be delivered at the end of a project. However, this is not the case. A system may include any or all of the following points:

- A product or set of products that are delivered to the customer as part of the overall system. The product may be a physical system that may be picked up and touched or it may be a virtual system such as software. A system may not necessarily be a technical system, but may be an ecosystem, a social system, a political system or an economical system. Taking matters to an extreme point, a system may be a human being, which is, arguably, one of the most complex systems in the world.
- Operational procedures or knowledge concerning the final product that will enable the customer to successfully install and run the system, and the end users to successfully operate the system. It is important that this information is built into the system design, rather than being added after the system has been fully developed.
- Support and maintenance of the system, to take into account future refinements, error corrections, updates and servicing.
- Training to enable customers and end users to use and operate the system in a safe and reliable fashion. This will not only ensure the safety and comfort of end users, but may also prolong the life of the system itself.
- Integration with other systems, as it is crucial to the operation of the system to make sure that it fits seamlessly with other components in the operational environment.

A system, therefore, is more than a final product. It is a living, evolving entity that requires looking after and occasional help and therapy. In order to keep a system in

top condition, a sound understanding of the way it looks and behaves is required both under normal conditions and under atypical conditions.

### *1.1.7 Defining systems engineering – second attempt*

Based on what has been discussed so far in this chapter, two definitions for systems engineering are suggested. The first definition of systems engineering that is derived directly from these discussions, taking all the points that were raised here, is:

> Systems engineering is concerned with defining and implementing an approach to solving problems, while managing complexity and communicating effectively over the entire lifetime of a project.

Another definition of systems engineering that has been derived indirectly from all the points raised in this chapter is:

> Systems engineering is the implementation of common sense.

This may seem like a strange definition, but all of the points that have been raised in this section are sound, basic engineering principles. In addition, when problems are analysed and the resultant causes identified, the initial reactions from most people are shock, horror and amusement that people could be stupid enough to allow it to happen in the first place!

### *1.1.8 Summary*

In summary, therefore, the following points are raised so far in this chapter.

- There is a need to engineer systems in order to make them as safe and reliable as possible and in order to give people confidence in the systems that we produce.
- Many problems with systems have three common themes: complexity, communications and a lack of understanding. All three of the problem areas must be borne in mind when engineering a system.
- A system was defined not simply as a product, but as products with a set of associated services.
- The nature of a system need not be physical, as it could be an intangible system, such as software or a social system.
- Systems engineering was defined in the context of this book as an approach to solving problems, while managing complexity and communicating effectively over a whole life cycle.
- Systems engineering was also defined as the implementation of common sense.

This section introduces the main issues involved with systems engineering and the following sections look at some of these issues in more detail. These issues are complexity, communications and understanding in Section 1.2, and quality in Section 1.3.

## 1.2 Complexity, communication and understanding

### *1.2.1 Introduction*

This section looks at the points that were raised in the previous sections as major causes of problems and discusses them in more detail. The main proposal here is that if we can understand a little about the nature of these problems, hopefully we can go some way towards addressing them so that we do not allow history to repeat itself with yet more failures and disasters. The main aim is to try to draw up a set of requirements that, when met, will allow us to go some way towards solving these problems. The term 'some way' is used here as it is impossible to eliminate all problems, but we must still do as much as we can. This is the same theory as the one concerning testing a system, according to which it is impossible to test a system 100 per cent and it is only possible to test a system to give a degree of confidence that the system will work under all anticipated circumstances.

The first of the points to be addressed is that of complexity, which will be discussed in the following section.

### *1.2.2 Complexity*

Complexity is currently a hot topic in systems engineering and there are many excellent texts available that describe it in great detail. This section aims to define complexity at a high level so that it can be understood, to a limited degree, by everyone reading this book. For more in-depth discussions and philosophical musing, readers are directed towards the titles referenced at the end of this chapter.

To start the discussion, it is necessary to visit the world of software engineering and to consider the work of Fred Brooks. Brooks, often viewed as the father of software engineering and one of the great software philosophers (if such a thing exists), identifies two main types of complexity that exist in almost all systems (software or otherwise): *essential* complexity and *accidental* complexity [9].

It is important to be able to distinguish between these two types of complexity, so that they can be identified, or potential areas identified, within a system.

#### 1.2.2.1 Essential complexity

Essential complexity is so called, not because it is vital to the system, but because it is in the essence of the system. This means that it is an inherent part of a system and, as such, cannot be eliminated. Clearly, the presence of such complexity is a big problem, which means that it is even more important that these areas of essential complexity are identified and have attention drawn to them in order that their effect on the rest of the system be minimised.

As an example of essential complexity, consider any system using a PC or similar computer. Within a PC there is a level of complexity that will exist within the PC itself. There is nothing that can be done to minimise the complexity of the hardware of the PC, short of taking it apart and completely redesigning it. The next level of complexity that exists is that of the operating system about which, again,

it is not possible to do anything, short of reinstalling another operating system. It may even be the case that there is more than one operating system, although more recently developed operating systems now claim seamless integration with the hardware, rather than using a middle-level operating system. If someone wants to develop software on the PC, there is the complexity of an interactive development environment that enables the user to write and manage the code, together with the complexity of the language and the compiler itself. This gives a number of factors that cannot be changed in any way and, hence, such a system will exhibit essential complexity as it lies within the essence of the system. All this before even a single line of code has been written!

Another example may be a legacy system that may have to exist as part of a new system. In real life, systems tend to evolve rather than be started from scratch, and thus legacy systems are often a necessary evil. It is often impossible or unfeasible to change a legacy system in any way; therefore, essential complexity will always exist within the legacy system.

One argument that is often put forward is that if there is nothing that can be done about essential complexity, then what is the point of thinking about it in the first place? The answer to this concerns project risk. If all areas of complexity are identified, it is possible to perform a risk-assessment exercise to determine any parts of the system that are critical to the success of the project. If these areas of complexity are known, it is possible to design the system so as to minimise the risk of the essentially complex parts.

It is essential that, in the case of legacy systems, the interfaces between the systems are fully understood. This means that they should be modelled not just from a data-driven view, or the 'what' of an interface, but also in terms of its behaviour, or the 'how' of the interface.

#### 1.2.2.2 Accidental complexity

Accidental complexity differs from essential complexity as, unlike essential complexity, which is inherently part of a system, accidental complexity is complexity that creeps into the system by accident. As accidental complexity is caused by accident or, to put it another way, by error (or even omission), it is possible to do something about it. In an ideal world, the solution would be to eliminate accidental complexity altogether, but in reality the best that can be hoped for is to minimise accidental complexity as much as possible.

In order to be able to minimise accidental complexity, it is crucial to understand how it may creep into a system. A number of causes of accidental complexity are discussed in the list below:

- Poor front-end life cycle activities. This includes, for example, requirements engineering, which drives the rest of the project. If the requirements are not right or have not been engineered adequately, complexity may creep in right at the beginning of the system's life cycle. This may be due to the fact that a requirement has not been analysed sufficiently and the complexity has been underestimated.

- Too few levels of abstraction are understood. It is crucial to look at different levels of focus, or abstraction, of a system ranging from high-level conceptual views to very low-level detailed views.
- A lack of understanding of the system or the techniques being used. This is particularly prevalent in the problem domain when the analysis is being carried out, which will lead to accidental complexity. Many projects fail due to the fact that the problem is not understood well enough before the design is started, which means that the likelihood of creating a correct solution is very small indeed.
- A lack of effective communications. This may well lead to complexity and sets off a vicious circle as communications and complexity tend to chase each other's tails and each results in the other. Poor communications, particularly at earlier life cycle phases such as requirements and analysis, may lead to all sorts of problems later on in the project. Even very simple terms will have meaning to some people but not others, or may have totally different meanings altogether.

Accidental complexity, therefore, is something that we can go a long way towards addressing and, as such, it is important to be able to minimise such complexity.

### *1.2.3 Minimising complexity*

In an ideal world, complexity would be eliminated altogether. However, in reality this is almost always not the case and thus complexity must be first identified and then minimised.

Complexity may be minimised by effective communication and by applying that great maxim that impacts upon all areas of life, KISS, or 'keep it simple stupid!'.

In order to do this, a well-defined approach is essential. This approach must be simple, straightforward and correct. By 'correct', we mean that the approach must be consistent within itself and comply with the way in which people work. The approach must also be known by all staff and interpreted by each team member in the same way.

### *1.2.4 Communications*

#### 1.2.4.1 Introduction

The subject of communications has already reared its ugly head at several points during this chapter, but what exactly is meant by communications? It is probably safe to say that everyone realises that communication is important for any project or any system, but what channels of communication exist? They exist at different levels and it is important to have an idea of what these levels are. These include:

- A channel of communication that exists at an organisational level. Many projects will consist of workers from more than one organisation, which has the potential to lead to communication problems. The languages used by organisations may

differ, with both technical and spoken languages and the protocols involved in talking to other people or groups of people possibly differing radically between organisations.

- A channel of communication that exists between different teams within a single organisation. People may work in disparate areas of the company and will thus have a completely different vocabulary due to their different areas of technical expertise. In addition, communications between teams may be semi-deliberate for fear of competition from rival groups within a single organisation, for example. This may, in extreme cases, even lead to sabotage between groups as one group is desperate to keep up with another. This may take on a relatively harmless form of sabotage, such as withholding information, or a far more malicious form of sabotage, such as feeding deliberate misinformation to another team. A further aspect of communication between teams that may lead to problems is that people assume other teams know something or have done something that may be crucial to the project. Making assumptions can be dangerous when it is taken for granted that everybody knows what they are, and they should, therefore, be explicitly defined and communicated.
- At a lower level of communication, a channel exists between different resources, such as hardware, software, networks, protocols, and so on. This may manifest itself by a particular piece of hardware not working with other hardware or new versions of software not working with older operating systems or software. It may also be caused by legacy systems that have existed within a system for many years yet, for whatever reason, remain a crucial part of a system.
- Problems may also exist between different levels of management. Many reasons exist for this, such as fear of 'shooting the messenger'. People are often scared of reporting failure, even if it has nothing to do with them, as it may be perceived as being the direct fault of the messenger. This may also manifest itself because there are too many layers of management within an organisation, making effective communication impossible due to the complexity of getting a single message to a high level of management via several other managers. People will often gloss over problems, which may result in the snowball effect of a simple problem the size of a small snowball gently rolling down a hill for as long as it is ignored, before finally reaching the bottom of the hill where it can no longer be ignored, is the size of a house, and destroys everything in its path or, in this case, leads to the demise of the project.

These levels of communication have been seen to have their own dangers within a project if not properly identified. It is crucial, therefore, to identify communication interfaces between the roles of people involved within a project. For example, who reports to whom within the organisation? What are the interfaces to other organisations and who is allowed to communicate across these boundaries? How are problems reported and to whom? Is blame associated with the reporting of problems or is the cause of the problem the main concern?

These difficulties can all be amplified by a single aspect of communication – that of the language being used.

#### 1.2.4.2 Languages

One of the basic requirements for communicating between one or more entities is to have some sort of language that each can speak. Any problems with a language will compound the problems related in the previous section. Many problems with languages exist, including the following:

- People from different countries may speak different languages, hence making communication almost impossible without an interpreter. Of course, the actual interpreter may introduce a number of ambiguities into the translation, thus causing additional complications and confusions.
- People with different working backgrounds may understand different things from the same term. For example, the word ‘fencing’ may apply to making fences and barriers, the noble art of swordplay or, indeed, selling stolen goods. The meaning of each of these is completely different and such a misunderstanding could potentially lead to large problems within a project.
- People from different countries may ‘think’ that they speak the same language, but do not! Perhaps one of the most unpredictable problems occurs when people think that they speak the same language but, in reality, there are big differences in the meaning of words. An example of this is American English vis-à-vis English. These two languages are deceptively similar but each has many words that either have no meaning in the other language or a completely different meaning. There is an old adage that the United States and the United Kingdom are separated by a common language and this is absolutely true. Many terms are taken for granted as having the same meaning when they do not. There are many examples of embarrassing situations that have occurred through simple miscommunications but one that occurs often concerns the definition of the verb ‘to table’. In the United Kingdom, if you are in a meeting and something comes up that is not on the original agenda, often a request is made to ‘table’ it, which means to discuss it in detail at some point in the meting. In the United States, however, the term ‘to table’ means to put it under the table and not to talk about it!

The previous three points refer to spoken languages, but these are not the only types of language that may be used on a project, and problems may also exist as a result of differences in technical languages such as:

*Programming languages*: In the case of software, there may be different versions, or flavours, of languages. Anyone who has ever used the C programming language, for example, will know that there are numerous definitions of the language itself, which will depend on many things, including the manufacturer chosen. Even when a language is chosen from a particular manufacturer, there will be different versions of the language, the compiler and the interactive development environment (where appropriate).

*Modelling languages*: It was very much the case that before the advent of the UML, there were over 100 modelling languages, making intercompany communication extremely difficult. Even with a standard language, such as the UML, different

people have different slants on the same concepts. Luckily, there is a single source of reference for the UML, so any ambiguities can be quickly sorted.

Clearly, there is a need for an unambiguous language that all interested parties can understand, which points towards a requirement for a common language.

#### 1.2.4.3 Common languages

As the points raised in the previous section have made clear, there is a need for a common language. This relates to the old adage about 'everybody singing from the same sheet' where it is crucial that everyone understands the same thing by every term used. This also relates back to the famous 'Tower of Babel' story from the Bible (suggested as a good example by Brookes), which involved a group of people who wanted to build a tower up to heaven, so that they could talk to God (this is a somewhat abridged version of the Bible story!). According to the original story, the builders worked very well together and were actually achieving their original objective and reaching heaven with their tower. God, however, had other plans for the people of Babel and made each of them speak a different language, thus rendering communication impossible. Rather than try to solve this problem, the Babelonians decided to go their separate ways, with the result that the world is now made up of people who speak different languages and the tower, sadly, was never completed. This is one of the first ever recorded project failures.

There are many advantages to having a common language, including:

*Saving on training*: If an organisation uses three or four different languages to represent different areas of business, clearly it is necessary to train staff in all languages. By using a single, unified language, it is possible to save by training all staff once in the same language. This also means that the company knowledge of that language will increase and improve as every member of staff will be familiar with the language.

*Cross-boundary communication*: If people from different areas of business or departments in a company all speak the same language, clearly the communications between such people will improve and leave less room for ambiguity.

A common language also equates to having a common medium of communication and thus a common ground for everyone to use.

### *1.2.5 A good approach*

In order to address the three evils of systems engineering, it is important that any project is approached in a concise, well-understood and well-defined manner. If the approach that is adopted does not take these three evils into account, the three most common causes of project failure and disaster are left unchecked and the battle to deliver a project on time will be lost almost before it is begun. Such an approach, therefore, has the following requirements:

*Consistent*: The approach must be correct with regard to itself and must make sense to those adopting it. Many approaches that are defined in books, standards and other

forms of reference are actually not correct, in that they contain inconsistencies or do not have a common vocabulary. If this is the case, there is a strong possibility that people will not understand the approach properly and this will lead to communication problems and complexity.

*Concise*: Any approach chosen must contain enough information to convey all of its meaning, yet not so much that it becomes verbose. One of the major problems facing anyone who is trying to introduce a new approach, or to standardise an existing approach, is to get it into a form that is manageable and that will not deter potential users by its sheer volume and complexity.

*Repeatable*: In order to perform any type of worthwhile assessment on a project, it is important that the approach is repeatable. This means that results from more than one project may be compared and value drawn from these comparisons. If the approach is required to be improvable in some way, then making sure that it is repeatable is absolutely critical.

*Measurable*: Having a concise, consistent and repeatable approach is still not enough unless the outputs of the approach (the things that are delivered on the project) are measurable in some way. Again, if the approach is to be improvable, then being able to measure it is essential.

*Improvable*: This is important in order to ensure that the approach is as good and reliable as possible with sustained implementation over a long period of time. This is a point that has already been mentioned twice and is one that feeds into two of the other requirements – those of repeatability and measurability.

*Tailorable*: No single approach will be applicable to every type of project and therefore it is essential that any defined approach must be 'tailored' to meet the needs of an individual project.

From this list of requirements, it should be clear that there is far more work involved in creating an approach than meets the eye. This will be a recurring theme throughout the rest of this book and is the focus of Chapter 6.

#### 1.2.5.1 Summary

In summary, therefore, the following points are raised in this section:

- Complexity will always exist in systems as either essential complexity or accidental complexity. Whichever form exists, it is crucial to identify all areas of potential complexity and, once identified, to address them in some way.
- Communication or, more to the point, effective communication, is key to the success of any project. One crucial part of this is to have a common language that can be adopted by all people involved in the project.
- Above all, a well-defined approach is required – one that can be understood by all people involved in the project.

These points raise issues that relate directly to 'quality', which is discussed in detail in the next section.

## 1.3 Quality

### *1.3.1 Introduction*

There is a need for quality throughout today's society. This quality is required for everything that people may buy, be it a product or a service. As engineers, it is important to have a good idea of what quality is and what it means to us as professionals. Apart from a moral and ethical obligation to produce quality systems, there is also a legal obligation, in many cases, to be able to demonstrate the quality of a product. An example of this is if, as a practising engineer, you produce a piece of software that results in the death or injury of another person, it is the engineer (you) who are personally responsible, rather than the company employing the engineer. It does not matter if there are disclaimers on the software (which so many manufacturers seem keen to put on their products) because the moment a person is hurt, all disclaimers mean nothing! The only way that it is possible to prove innocence under such circumstances is to demonstrate, in a court of law, that current best practice had been followed to produce the software or, put another way, you must demonstrate the quality of the approach to produce the code [10, 11].

It is crucial, therefore, that engineers provide products and services in which people can have a high degree of trust and confidence. There are several ways that, as a customer, it is possible to judge or assess the quality of the service or product. Consider, for example, purchasing any product for use by your family. Imagine that you were about to buy a car seat for a baby or small child for use in a family car. Clearly, the product that is bought must inspire a high degree of confidence from the customer's point of view as, in the worst-case scenario, there is a child's life at risk if something goes wrong.

It is crucial, particularly in the case of children, that the product is safe and has met stringent quality guidelines, but the problem is, how does one know that the product has met these guidelines? In real life, the first thing that should be looked for would be an indication of the quality of the product. In the United Kingdom, for example, one would look for the Kite mark, which is an indication that the product has met a particular British Standard. Indeed, one would hope that if a product such as a child's car seat did not display a Kite mark then you would not purchase that product. Another example of such a symbol that indicates quality is the European quality mark, which is depicted by the 'CE' symbol.

Any product that displays such a quality symbol gives the customer confidence in the product and assurance that the product has passed stringent safety and quality tests. The same is true for services, not just products. When employing a person or awarding a contract it is usual to look for certain indications of quality of work, such as: examples of previous work, qualifications, experience and professional status (for example, membership of a professional organisation, chartered status, etc.).

Certain manufacturers or service providers have such a good reputation that simply seeing their logo or trademark will inspire confidence in a product or service. Think of car manufacturers and then think of the following criteria: safety, comfort, build quality, performance and budget. When looking at each of these criteria, it is likely that particular car manufacturers will spring immediately to mind.

However, can the same be said of the way that we approach systems engineering? What should people look for when purchasing or specifying systems?

This section addresses this point in particular and discusses how to demonstrate quality of both products and services and relate them to systems engineering. It also derives a number of definitions for terms such as quality that are crucial to our understanding of systems engineering.

### *1.3.2 The need for quality*

The term 'quality' is frequently used and quoted, yet is often misunderstood. Before we can understand quality, it is important that we understand the need for quality. There are many reasons why people require quality for their organisation:

*Customer confidence*: If the customer has confidence in a product or brand of product, they are likely to select these products over others. Certain organisations are always associated with a particular level of quality of product and, given a choice, customers will almost always select the quality brand. Consider the example of buying a brand-new car. Which manufacturer will you choose? Clearly, money is a major driving factor behind the purchase of a car, but what about the actual name of the manufacturer? How much impact will that have on the decision to buy or not to buy the particular brand of vehicle?

*More maintainable*: Systems that are more maintainable will save money over those that are not very maintainable. Many examples of this can be found in current literature. For example, experience has shown that the most expensive phase of the life cycle is the maintenance phase. Systems should be designed with maintainability in mind.

*Common sense*: At the end of the day, it is simply common sense to provide quality systems.

Unfortunately, it is a fact that many organisations will not act unless it either saves money or makes money. It is therefore very important to be able to relate quality back to financial terms as, above all else, this is (unfortunately) the only way that some individuals or organisations will react. It should be made quite clear how each of the points raised here may relate directly back to finance.

### *1.3.3 Level of responsibility*

The level of responsibility taken on by complex systems is constantly increasing. As these systems permeate our lives, often in ways that go unnoticed, they are being given more and more responsibility, especially where safety-critical systems are concerned. Some areas in which complex systems play a role that includes safety-critical elements are:

*Closed loop control systems*: Today, some computer-based systems are completely in control of a number of areas of process control, such as manufacturing processes and plant monitoring processes. It is essential, therefore, that any such control systems have been designed and built with safety and reliability in mind.

*Autopilots for aeroplanes and ships*: Autopilots are increasingly being used in modern transport. Many air passengers clap after the successful landing of an aeroplane, but in most cases they are applauding a piece of software, as humans are mainly required for emergency or atypical situations. Needless to say, the software has no appreciation of such applause although it does amuse pilots!

*Braking systems in cars*: As technology is used increasingly in cars, so we see an increase in the level of responsibility that these complex systems have. The modern car may have computer-controlled brakes, suspension and steering, in addition to non-safety-critical applications such as electric windows, electric wing mirrors and windscreen wipers. Many cars today have cruise control, systems to aid steering and even systems to aid more complex tasks such as parking.

This increased level of responsibility thus increases the need for quality in the systems and the way in which they are produced.

### *1.3.4 Requirements for systems professionals*

Over the last few years, the requirements for systems professionals have changed. It is no longer good enough to be a programming guru or an extraordinary designer. Instead, it is becoming increasingly important to show an awareness of wider issues, even if you are not an expert in them. These include:

*Quality policies, standards and procedures, both in-house and those in the public domain*: Any member of an organisation who is not aware of quality issues, even at a high level, runs the risk of causing problems on a project. Even if no problems arise directly from this, it is highly possible (and, indeed, probable) that an organisation will fail a quality audit if members of their staff are not aware of the company's quality policies.

*Safety policies*: These may be policies that relate to the system itself, such as safety-critical systems, or they may be basic safety policies for staff, such as fire regulations. Clearly, failure to meet either of these could result in injury or loss of life.

*Roles and responsibilities*: It is essential that every task or action carried out by an organisation has a role defining who is responsible for that action. This is particularly important when things go wrong, or if things 'don't get done', as it provides a good starting point for getting to the heart of the problem. For example, who takes the blame when a system developed by you causes an accident? This has already been mentioned and, if appropriate roles and responsibilities are not defined, it could mean blame for something arriving on your shoulders, which, by rights, should not be there.

These other areas of interest of which engineers must have knowledge are really concerned with the approach taken, or the process followed, in certain areas. Clearly, it is impossible for any single person to have a complete, expert knowledge in all areas, but is important to have at least an appreciation of the other processes that exist within an organisation.

### *1.3.5 Processes and quality*

The points raised in the previous section are all well and good, but how does this relate to quality? In order to answer this, it is time to come up with some definitions for quality. This is very difficult as there are many, many different definitions for quality. The two definitions chosen here are as defined by ISO (International Standards Organisation) and are:

- 'fitness for purpose'
- 'conformance to requirements'.

'Fitness for purpose' means that the system must do what it is supposed to do. This is actually not the same as saying that the system should work. Proving that a system works is 'verification', while proving that something does what it is supposed to do is 'validation'. Therefore, it is possible to have a fully working system that has been verified, but if it does not meet its original requirements it cannot be validated.

'Conformance to requirements' means that the system must be able to demonstrate that it meets its original requirements. The original requirements state what the system is supposed to do and thus conformance to requirements is really another way of saying 'validation'.

In order to demonstrate something, it is vital to have a source of reference against which the demonstration can be compared. In the case of demonstrating quality, the source of reference that the system is compared against is usually a standard, a procedure, or both.

When demonstrating quality, it is the approach taken that is compared to an established norm in order to demonstrate compliance. Formally, this approach is known as its 'process'. Processes have been encountered twice already in this chapter in the following way:

- Requirements for a good approach were drawn up, which means that they are requirements for a process.
- A sound knowledge of processes was required for systems professionals, which means that these processes describe an approach to engineering, management, support services, customer supplier relationships and organisational activities.

Therefore, it is the 'process' that is assessed or audited when demonstrating compliance with a standard, which will demonstrate the quality of an approach.

### *1.3.6 Standards*

An important aspect, therefore, of demonstrating quality is to have a standard according to which compliance can be demonstrated. The definition of the 'standard' as used here is 'a form of reference that has been agreed upon'.

Unfortunately, in the world of systems engineering, there are literally hundreds of standards, which is simply too many! Interpreting this according to our definition, there are a great many references that have been agreed upon!

In order to make sense of this incredible number of standards, they may be grouped into four broad categories:

- International standards, which are recognised in many countries, such as: ISO, IEEE (Institute of Electrical and Electronic Engineers), IEC (International Electrotechnical Commission), IEE (Institution of Electrical Engineers), BSI (British Standards Institution), EN (European Normative – except in French) and ANSI (American National Standards Institute). Although some of these appear to be country specific, such as BSI and ANSI, they are in fact recognised in many countries. Most companies will comply, or try to comply, with at least one international standard. These are usually the standards that give a basic quality indicator recognised in many countries.
- Publicly-available specification (PAS) is one step down from the full-blown international standard and is often generated as a fast-track route to get a standard into circulation quickly. Organisations such as the British Standards Institution and ISO offer the creation of PASs as a service and they will often lead to becoming an internationally-recognised standard.
- Industry standards, which are defined when a group of industrial companies get together and decide to do things in the same way. Industry standards include: UML, CORBA (Common Object Request Broker Architecture) and ODBC (Open Data Base Connectivity). At least some of these will be familiar to most readers (if not now, then by the end of this book). The rationale behind this is that international standards are typically very slow to produce and many never make it past the draft stage before they are abandoned.
- In-house standards, which are defined on a company-by-company basis and are found in all large organisations. Examples of these are available in the public domain and include the European Space Agency (ESA), which publishes its software standards in a book, and DERA (Defence Evaluation and Research Agency [whose name is apt to change], which is UK-based) standards, which are available on the Internet.

Perhaps it is due to the large number of standards that it is very difficult to understand many of them in any practical fashion. The next section looks at a number of the problems faced by anyone wishing to use any of the standards.

### *1.3.7 Problems with standards*

Many problems exist with standards, which is particularly annoying as much of this chapter has so far been devoted to stating how important they are for effective systems engineering. There are two main areas into which these problems fall, which are problems within individual standards and problems between different standards.

- Some standards are too long. For example, both ISO 15504 Software process improvement and capability determination (SPICE) and EIA 731 Standard for systems engineering capability are several hundred pages long. The problem here is that many potential users will be deterred even before they begin to try

to understand and use the standards. However, length is not always superfluous, nor is it an indication of complexity as in the case of the two standards mentioned here – the information in the standards is very clear and, quite unbelievably, very easy to understand. The length of the documents stem from the fact that they are simply very thorough and both attempt to provide a framework for assessing every process within an organisation with regard to process improvement.

- Some standards are too brief and, hence, are too high level to be of any practical use. As an example of this, ISO 9001 (arguably the most widely recognised standard in the world) is only 24 pages long. This is a standard that covers almost all applications of systems specification and design, yet it is incredibly brief. As a consequence, the standard itself is written at a very high level and is thus open to a great deal of ambiguity when being interpreted by different people. The balance between a standard being too long or too brief is a difficult concept to achieve.
- Some standards are very difficult to understand. This may be for any number of reasons, such as: they are too long or too brief, as mentioned previously, or they have been written by a committee. Of course, they may simply be badly written!

The points raised so far are relevant when a single standard is being used; however, in many cases, it may be desirable to meet more than one standard. Consider the case where a company may have an organisation-wide policy to meet an established international standard such as ISO 9001 Model for quality assurance in design development, production, installation and servicing, but may also have to meet an industry-specific standard such as EN 50128 Railway applications, software for railway control and protection systems, which applies to the rail industry. There may also be government standards that have to be met, such as, in the case of the rail industry in the United Kingdom, the guidelines laid out by HMRI (Her Majesty's Railway Inspectorate).

There are problems associated with trying to meet more than one standard that are in addition to all the points raised concerning single standards:

- Nomenclature. The terms used by different standards differ enormously, with some basic concepts having completely different meanings. This comes back to communication problems as each standard is written in a different language.
- Some standards assume that particular processes may already exist or have already been defined or complied with. A number of standards, for example, assume an ISO 9001 standard-compliant management system. This adds additional constraints to what may at first impression seem like a single standard and will include all the problems mentioned previously in this section.

Meeting a single standard is, in many cases, difficult to demonstrate. Meeting more than one standard is often orders of magnitude more difficult than meeting a single one.

### *1.3.8 Consequences of problems with standards*

All of the problems raised in the previous section will lead to adverse affects on the project. These will include:

*Complexity*: If not identified and addressed, this may lead to communication problems if complex information is conveyed to other people and, hence, to a lack of understanding.

*Communication problems*: These may lead to complexity if ideas are not communicated properly by a team and, hence, to a lack of understanding.

*Lack of understanding*: This may lead to communication problems if the information is not fully understood and, hence, complexity if the information modelled is not correct, concise or consistent.

These three points should be quite familiar by now and should only serve to reinforce how important it is that they can be addressed.

### *1.3.9 Compliance with standards*

Compliance with a standard is generally demonstrated by carrying out either an assessment or an audit. These two are similar yet subtly different and, in very simple terms, can be differentiated as follows:

*An audit*: During an audit, an organisation has its practices compared to that of a standard as defined and documented in the organisation's operating procedures. This often takes the form of taking these procedures as a starting point and then actively looking for discrepancies or noncompliances with the standard. Formal audits, which result in certification according to a standard, are carried out by independent, third-party organisations. The result of an audit is a simple 'yes/no' or 'pass/fail' with minimal information concerning the noncompliances, usually in the form of a simple reference to the relevant section in the standard.

*An assessment*: During an assessment, an organisation has its processes compared to those of an established process model, usually a standard. An assessment starts with a blank sheet of paper and whatever processes are executed (whether defined and documented, or even assumed) are assessed. Assessments may be carried out independently or in-house, providing that the assessor is qualified in process assessment. The result of an assessment is a 'capability profile', wherein each profile that has been assessed is rated according to a predefined scale, which provides an indication of how 'mature' each process is.

In order to demonstrate compliance with a standard, there are a few basic requirements that must be met:

- A defined process must exist. Without a defined process, it is impossible to meet any type of standard with any degree of satisfaction as this is usually the thing being assessed in an audit, for example. By 'defined', it means that the process must be documented in some way, such as a formal document or electronic copy that is available to staff (such as an Intranet or Internet-based definition).

- Use of the process must be demonstrated. It is not good enough simply to have a defined process if nobody follows it. It is important to be able to show examples of the processes in action or, to put it another way, projects. The things that are evaluated in order to demonstrate the use of the process are the process deliverables, or artefacts. These deliverables may be documents, code, hardware, reports, etc.
- The process must be improvable. A defined and implemented process is just a first step. It must be shown that the process works effectively and that mechanisms are in place to continuously improve the process as time goes on.

One approach to demonstrating continuous compliance with a standard is to represent both the process and the standard using a common language, which will overcome many of the communication problems. Once both the process and the standard are represented in a common language, it is possible to compare them directly. This approach is also applicable to the case where more than one standard is required for compliance.

### *1.3.10 Processes*

In very simple terms, a process describes an approach to doing something. It is something that transforms inputs into outputs and has responsibility associated with it.

Everybody actually follows a process whenever they do anything, even if they fail to realise it and it does not even have to be written down. When people do things in a ‘certain way’, they are actually following a process, albeit an informal one. Thus, if we want to demonstrate the quality of the way in which we work, it is important to be able to identify the different areas of activity that relate to work. It is necessary, therefore, that key processes in an organisation can be identified and defined in a way that everyone can understand. This relates directly back to the point made previously concerning people having a greater understanding of the way in which a company works. In real terms, this equates to staff having a high-level knowledge of the types of process (describing an approach to work) that exist within their organisation. Typical processes within an organisation include:

- Customer supplier processes, which include protocols and a defined way of dealing with customers and for customers to deal with suppliers.
- Support services, which represent services such as change control, configuration management, etc.
- Management, which includes processes such as project management and project planning, which apply on a project-by-project basis rather than across the whole organisation.
- Organisational processes, which will be applied across the whole enterprise, such as prescribing standards, health and safety recommendations, etc.
- Engineering, which covers the development of a product from requirements to retirement.

It is important that these processes are identified and categorised in order to make understanding them as simple as possible. The categories introduced here are based

on the ISO definitions of processes, but are by no means carved in stone. In fact, Chapter 6 shows several other categorisations of processes according to different standards.

### *1.3.11 Summary*

In summary, therefore, the following points are raised in this section:

- There is a need for quality in everything that we, as systems professionals, do and produce.
- Quality is most often demonstrated by compliance with a standard. However, many problems exist in understanding such standards, either individually or across more than one standard.
- In order to demonstrate compliance with a standard, a well-defined process is essential.

Processes are discussed in more detail in Chapter 6, where an approach to solving the problems raised here is suggested. Several real-life examples of modelling standards are given, together with an example process that is defined according to these standards.

## 1.4 Conclusions

This chapter raises several of the main issues that affect systems engineering in today's world. In summary, the key points are:

- Project failures and disasters are commonplace in today's technology-driven society. The result of such failures and disasters can range from loss of money, to loss of life and damage to the environment.
- Some of the common themes behind many such failures and disasters are the three evils of systems engineering: complexity, communications and a lack of understanding.
- In order to control and minimise these problems, it is essential that the way we work is of the highest quality and that this quality can be demonstrated to other people. As quality is defined as 'fitness for purpose' and 'conformance to requirements', it is these two definitions that must be demonstrated. In practical terms, this translates to having a well-defined process.

Finally, two definitions of systems engineering are suggested:

- 'Systems engineering is concerned with defining and implementing an approach to solving problems, while managing complexity and communicating effectively over the entire lifetime of a project.'
- 'Systems engineering is the implementation of common sense.'

The information in this chapter forms the basis for the remainder of the book.

## 1.5 Further discussion

1. Think of the following terms and see if you can associate a car manufacturer with the term: 'expensive', 'safety', 'reliability', 'expensive to maintain', etc.
2. Is ISO correct in its definition of the five main categories for processes within an organisation? Do these adequately represent the way that you or your organisation work?
3. Look out for quality symbols on products and services that you encounter. Find out which standard they comply with and check this standard to see what it actually means to you as a customer.

## 1.6 References


1 STEVENS, R., BROOK, P., JACKSON, K., and ARNOLD, S.: 'Systems engineering, coping with complexity' (Prentice Hall, Europe, 1998)
2 O'CONNOR, J., and McDERMOTT, I.: 'The art of systems thinking, essential skills for creativity and problem solving' (Thorsons, London, 1997)
3 SCHACH, S. R.: 'Software engineering with Java' (McGraw-Hill International Editions, New York, 1997)
4 BIGNELL, V., and FORTUNE, J.: 'Understanding systems failures' (Open University Press, London, 1984)
5 COLLINS, T., and BICKNELL, D.: 'Crash, ten easy ways to avoid a computer disaster' (Simon and Schuster, London, 1997)
6 FLOWERS, S.: 'Software failure: management failure, amazing stories and cautionary tales' (John Wiley and Sons Ltd, Chichester, 1997)
7 LEVESON, N. G.: 'Safeware, system safety and computers' (Addison-Wesley Publishing Inc., New York, 1995)
8 LEWIN, R.: 'The major new theory that unifies all sciences, complexity, life on the edge of chaos' (Phoenix Paperbacks, London, 1995)
9 BROOKS, F. P.: 'The mythical man-month' (Addison-Wesley Publishing Inc., New York, 1995)
10 AYRES, R.: 'The essence of professional issues in computing' (Prentice Hall, London, 1999)
11 BAINBRIDGE, D.: 'Computer law' (Pitman Publishing, London, 1996)
12 BERTALANFFY, L.: 'General systems theory – foundations, development applications' (George Brazillier, New York, 1968)

# *Chapter 2*
# Modelling

better to understand a little, than to misunderstand a lot

*Homer Simpson*

## 2.1 Introduction to modelling

This section introduces the concept of modelling and includes a discussion on why modelling is so important and why we need to model. It also contains information that forms the basis for the remainder of this part of the book and introduces a number of basic concepts that will be referred to constantly throughout the rest of the book.

### *2.1.1 The importance of modelling*

It is vital to understand the reasons why we must model things. In order to justify the need for models, it is probably easiest to look at a number of simple examples, based on real-world systems, to which people can relate.

Note that here we are justifying modelling in general terms and not simply with regard to software systems, which demonstrates the need for flexibility when modelling – indeed, the three examples used here are nonsoftware systems. Many examples are used in different books, but the examples used here are based on those defined by the master of modelling, Grady Booch [1]. There seems little point in contriving a new modelling example when there is a perfect set of examples that has already been defined and is widely accepted as the norm. Therefore, the examples chosen here are based on Booch and his doghouse, house and office block.

#### 2.1.1.1 The kennel (doghouse)

For the first example of modelling, consider the example of building a kennel. A kennel is a small house where pets, usually dogs, can spend some time outside without being exposed to the elements. The basic requirement for a kennel is to keep the dog happy. This will include building a structure that is waterproof and

large enough for the dog to fit inside. In order for it to get inside, there must be an entrance in the kennel that is, preferably, larger than the dog itself. The inside should also be large enough for the dog to be able to turn around in order to leave. Dogs are particularly bad at walking backwards, which makes this last point crucial. Finally, the dog should be comfortable enough to sleep in the kennel and thus some bedding or cushions may be in order.

If you were about to build this kennel, then consider for a moment the basic skills and resources that you would require. You would be wise to have:

- Basic materials such as timber, nails, etc. The quality is not really that important as it is only for a dog. This might involve looking for any old pieces of wood around the house or even making a trip to a hardware store.
- The money needed to pay for the kennel would be your own, but is unlikely to be a large outlay. In terms of your personal income, it would be a fraction of a week's salary – perhaps the cost of a social evening out.
- Some basic tools, such as a hammer, a saw, a tape measure etc. Again, the quality of the tools need not be wonderful, providing they get the job done.
- Some basic building skills. You need not be a skilled craftsman, but basic hand to eye co-ordination would be an advantage.

At the end of the day (or weekend), you will probably wind up with a kennel that is functional and in which the dog would be happy to shelter from the rain.

If the kennel was somewhat less than functional and the dog was not very happy with its new accommodation, you could always start again (after destroying the first attempt) and try a bit harder, learning from past mistakes. It would also be possible to destroy the doghouse and then to deny all knowledge of ever having built one in the first place, thus avoiding embarrassment later. Alternatively, you could get rid of the dog and buy a less demanding pet such as a tortoise, as there is no need to build a kennel for an animal that carries its own accommodation on its back. After all, the dog is in no position to argue or complain.

This is Booch's first example of modelling: the kennel or doghouse.

#### 2.1.1.2 The house

Consider now, maybe based on the resounding success of your kennel, that you were planning to build a house for your family. This time the requirements would be somewhat different. There would need to be adequate space for the whole family in which to sit and relax. In addition, there would have to be adequate sleeping arrangements in the number of bedrooms that are chosen. There would need to be a kitchen, maybe a dining room and one or more bathrooms and toilets. As there will be more than one room, some thought should be given to the layout of the rooms in terms of where they are in the house and where they are in relation to one another.

If you were to build a house for your family, you would (hopefully) approach the whole exercise differently from that of the kennel:

- You would have to start with some basic materials and tools, but the quality of these resources would no doubt be of a higher concern than those used for the

kennel. It would not be good enough to simply drive down to a local hardware store and pick up some materials as the quantity would need to be far greater and it would not be possible to guess, with any degree of accuracy, the amount of materials required.

- Your family would also be far more demanding and vocal than the dog. Rather than simply guessing your family's requirements, it would be more appropriate to ask them their opinions and perhaps get a professional architect in to listen to and discuss their needs.
- Unless you have built many houses before, it would be a good idea to draw up some plans. If you were hiring skilled craftsmen to do the job, you would certainly have to draw up plans in order to communicate your requirements to the builders. These plans may require some input from an architect in order that they achieve a standard that may be used effectively by the people who will be building the house.
- The house would also have to meet building regulations and require planning permission. This may involve contacting the local council or government representative and possibly applying for permission to build. This in turn would almost certainly involve submitting plans for approval before any work could be started.
- The money for the house would probably be yours, and thus you would have to monitor the work and ensure that people stick to the plans in order to get the job done in time, within budget and to meet your family's original requirements. The scale of the financial outlay is likely to be in the order of several years' salary and would probably be borrowed from a bank or building society and would thus have to be paid back, regardless of the outcome of the project.

If the house turns out not to suit the requirements, the consequences would be more serious than in the case of the kennel. The house cannot be knocked down and started again as the kennel could, because considerably more time and money would have gone into the house building. Similarly, you cannot simply get a less demanding family (in most cases) and living with the consequences of failure is not worth thinking about!

This is Booch's second example: the house.

#### 2.1.1.3 The office block

Taking the two building projects that have been discussed so far even further, imagine that your ambition knows no bounds and that you decide to build an entire office block.

Consider once more the resources that would be required for this, the third and final building project.

- It would be infinitely stupid to attempt to build an office block by yourself.
- The materials required for building an office block would be in significantly larger quantities that the house (and most definitely the kennel). The materials would be bought direct from source and may even need to be brought in from specialist suppliers, perhaps even in different counties or countries.

- You will probably be using other people's money and thus the requirements for the building will probably be their requirements. In addition, their requirements will no doubt change once you have started building the tower block.
- More permissions are required to build an office block than a house and many more regulations must be considered. Consider, for example, environmental conditions that the office building may have to meet – the building must not block anyone's light, it will be required to blend in with its surroundings, or it may be deemed too ugly for a particular area.
- You will have to carry out extensive planning and be part of a larger group who are responsible for the building. Many teams will be involved from different areas of work (builders, plumbers, electricians, architects, etc.), all of whom must intercommunicate.

If you get the right teams involved and enjoy a degree of luck, you will produce the desired building.

If the project does not meet the investor's requirements, you would face severe repercussions, including the potential of no further work and the loss of reputation.

This is Booch's third example: the office block.

#### 2.1.1.4 The point

These three examples from Booch may seem a little strange and somewhat trivial at first glance; however, there is a very serious and fundamental point behind all of this.

Nobody in their right mind would attempt to build an office block with basic DIY skills. In addition, there is the question of resources, and not only in terms of the materials needed. In order to build an office block, you would need the knowledge to access the necessary human resources (including people such as architects, builders, crane operators, etc.), plenty of time and plenty of money.

The strange thing is that many people will approach building a complex system with the skills and resources of a kennel-builder, without actually knowing if it is a kennel, house or office block. When contemplating any complex system, you should assume that it will be, or has the potential to turn into, an office block building. Do not approach any project with a 'kennel' mentality. If you approach a project as if it were an office block and it turns out to be a kennel, you will end up with a very well-made kennel that is the envy of all canines. If, however, you approach a project as if it were a kennel and it turns out to be an office block, the result will be pure disaster!

One of the reasons why it is so easy to misjudge the size and complexity of a project is that, in many cases, many elements of the system will not be tangible or comprehensible. Consider integrated circuits: who can say, simply by looking at an IC, how many transistors are inside it? Is it a processor or a simple logic gate? Consider software: simply by looking at a CD-ROM it is impossible to judge the size or complexity of the information contained on it. In terms of size, this could range anywhere from a single kilobyte to several gigabytes (if the information is compressed or 'zipped'). Even if you have an idea of the type of application that is on the CD-ROM, it is still difficult to judge the size of the software. Take for example, software

that may help people write letters: this may be a simple line editor program that may be a few kilobytes of information or it could be a full-blown office application that may not fit on to a single CD-ROM. The fact is that all projects that involve complex systems will have an intangible element about them, whether it is a control system, a process, or whatever.

The important term that is used here is 'complexity' rather than size, as size is not necessarily a reflection of the complexity of a system.

### *2.1.2 Why projects fail*

Projects fail for many different reasons. However, there are several underlying reasons, or themes, that tend to emerge when looking at project failures and disasters in general. There are three main themes, which have already been discussed in some detail in Chapter 1:

*Complexity*: Complexity is a huge problem. Some complexity is due to poor engineering or the second and third reasons on this list, which is known as 'accidental complexity'. Accidental complexity can be avoided or, at the very least, should be minimised. Unfortunately, there is a second type of complexity, 'essential' complexity, which is unavoidable as it is in the essence of the system.

*Lack of understanding*: The lack of understanding could concern almost any aspect of the development life cycle. For example, not understanding what the system should do (the requirements) can lead to disaster in later stages. Not understanding the technology that is being used to implement a solution can also lead to disaster, particularly when people get carried away with the latest techniques and buzzwords and use technology for technology's sake. Another common area of misunderstanding is the nature of the problem itself – often brought on by inadequate analysis.

*Communication*: Communication, or rather a lack of it, is a problem that occurs in all aspects of life. People may not be able to communicate effectively because they speak different languages, they have different areas of expertise, they have different definitions for the same term, or they cannot describe their thoughts effectively.

However, many projects do succeed. One reason is that they avoid, or minimise, the aforementioned problems by effective and appropriate modelling.

### *2.1.3 Modelling*

Are there any other reasons why we model?

The three points that were raised in the previous section have already been discussed and form some of the basic requirements for modelling. After all, if these are the three biggest problems with systems and modelling is one of the reasons why projects succeed, it is not unreasonable to relate the two together.

Other requirements for modelling emerge from the three themes mentioned previously. There is a need, for example, to visualise systems as they will appear as a final product. Reconsider, for a moment, the three examples already introduced in this chapter: the kennel, the house and the office block. Almost anybody

should be able to distinguish between the three types of building, even if they are shown different representations of the same type of building. This is because when we hear a term, such as house or office block, we can immediately visualise what the system will look like. However, when we hear other terms, such as automated control system or ecosystem, it is unlikely that any two people would come up with a similar image of what the final system would look like as they are not physical systems. It is therefore important that we can visualise the final system at the outset to give some idea of the usability and functionality of the system. This may include building a small physical model of a building, drawing a diagram of what something should look like, and so on. This helps potential users identify with what they will be receiving. It would, after all, be very foolish to pay for an office block when you had not seen any plans or models of what the final building would look like!

These plans and models may also be used for another purpose – that of analysing a final product before it is built. This may be something very simple, such as getting the colour or size right, or it may be something far more complex, such as stress analysis, wind-tunnel testing, performance loading, and so forth.

Models may also be used as a template for creating or constructing the final system. This is simple to understand within the context of building a house as the house is based directly on the plans that are drawn up by the architect. The same is true for circuit designs, which are simply models, albeit in an abstract form, of the circuit that is to be built.

In order to understand one final use for models, it is necessary to take a step back and consider what happens in a project when something does go wrong. Typically, what will happen is that someone will ask why a particular design had been adopted. By modelling throughout a project it is possible to model not only a single solution, but also a set of candidate solutions that may be assessed and a final solution can then be chosen that best meets the original requirements of the system. These models of the complete set of candidate solutions may then be used as a basis for documenting decisions and if the models are retained so is the decision-making process. This is helpful in justifying why a particular approach or solution was selected over any other choices, which is particularly useful when people are trying to 'pass the buck'.

After all these discussions concerning the rationale for modelling, it is now time to define exactly what a model is.

### *2.1.4 Defining a 'model'*

It is now possible, bearing in mind the previous discussion, to come up with a definition for a model, which will be taken once more from Booch.

> A model is defined as a simplification of reality that is created in order to better understand the system under development, as we cannot comprehend complex systems.

We return to this definition throughout this chapter and the rest of the book and thus it should be kept at the forefront of your mind while reading.

In summary, therefore, we use modelling in four explicit ways:

- To visualise a system. To quite literally get a mental picture of what the final system will look like from the user or operator's point of view.
- To specify a system. By specify, we mean to state specific needs or requirements concerning the system.
- As a template for creation, for example, of a plan from which to build the final system.
- In order to document decisions made throughout the project. Quite often, especially when things start to go wrong, people will look to lay blame or to try to establish why particular decisions were made at key points in the project. By effective modelling it is possible to capture thought processes throughout the life cycle of the project, which can be enormously powerful later on.

These four aims of modelling are used to help to reduce complexity, improve communications and enhance your understanding of a system.

### *2.1.5 Principles of modelling*

Booch identifies four principles of modelling that are deemed crucial for successful and consistent modelling: the choice of model, the level of abstraction, connection to reality and independent views of the same system.

#### 2.1.5.1 The choice of model

The choice of model will have a profound influence on how a problem is approached. Approaching a problem the right way can make a job much simpler and will be quicker than adapting the wrong approach.

Consider for example other subjects that are taught in a school or college. Many subjects are taught by advocating a particular approach to analysis or problem solving but an inherent part of the approach will be to choose the right way to solve the problem. As an example of this, consider a simple mathematical operation, such as finding the slope of a line. This may be done in many ways, depending on the type of line. If it is a straight line, a simple 'height over length' equation will solve the problem. However, if, on the other hand, the line is a curve, this simple formula will not work, as the slope of the line will change depending on the point in the line at which the slope is measured. In order to solve this problem, calculus and in particular, differentiation, should be applied. Applying this retrospectively, differentiation can be applied to a straight line with the same result as the simple formula, but not *vice versa*.

Everybody must have sat in an examination and started out answering a question, only to find that, halfway through, the wrong approach has been taken and the problem must be restarted. This could have a big influence on your life, particularly if you do not finish the exam or if you get the answers wrong. Also, remember, as some people are so fond of saying: you get more marks for your approach than for the actual answer! It is the same with all engineering, a well-defined approach is vital, which involves choosing the appropriate model.

This is a topic that will be raised many times once the UML has been introduced.

#### 2.1.5.2 Abstraction of the model

For the purposes of this book, 'abstraction' refers to the level of detail of a model. Imagine the office block problem. You may need to create a model to show the size of the whole building, or how a single storey is constructed or a single light switch operates. Each of these requires a different level of abstraction for a successful model.

The point to be made here is that any model will require different levels of abstraction to be represented, otherwise it will have little chance of being correct. Imagine the scenario in which a single model was drawn up to represent the office block. Although useful for the financial backers to see a high-level view of the building, it would be next to useless for an electrician, plumber or ventilation engineer.

#### 2.1.5.3 Connection to reality

One problem with modelling is that, according to our definition, models simplify reality. This means that some information must be lost somewhere along the line, which can cause problems. Therefore, it is vital to know both how the model relates to real life and how far it is divorced from real life.

From a practical point of view, initial models tend to have quite a loose connection to reality and, as these models evolve, they get closer and closer to reality. The final connection to reality will be at the point when the model actually becomes reality, which is when the project is implemented or constructed based on the model.

From a conceptual point of view, it is important that whoever is looking at a model can make a connection to reality, otherwise the model becomes meaningless. Consider again the example of differentiation. School children often find differentiation difficult to understand as they cannot 'see the point' of it. Only when practical examples can be shown for using differentiation, such as working out speed and acceleration, can people truly understand the concept.

#### 2.1.5.4 Independent views of the same system

It has already been stated that a good model requires views that represent different levels of abstraction, but it is also true that a good model requires views made from different vantage points.

Consider again the office block building where many different views will be required to completely model the system. The different people who work on the project will require different views of the same model. For example, electricians should really have wiring diagrams, while painters will require colour charts. There is no point giving the colour chart to the electrician, nor the wiring diagram to the painter, as they are simply not relevant.

One crucial point that must be made here, however, is that each of these independent views must be consistent with one another, or, to put it another way, they must integrate correctly.

## 2.2 The UML

### *2.2.1 Introduction*

The previous sections have identified many requirements for modelling and justified why modelling is so important. In order to model successfully, a common language is required and that common language is, quite unsurprisingly, the Unified Modelling Language (UML).

The UML is a general-purpose modelling language that is intended for software-intensive systems. This definition is far too restricting, however. The UML is a language and can thus be used to communicate any form of information and should not be limited to software. The main aim of this book, therefore, is to take the UML and apply it to systems engineering, rather than simply to software engineering. There are many excellent books that relate the UML to software, so this is kept to a minimum in this book.

### *2.2.2 Aspects of the model*

There are 13 diagrams in the UML that may be grouped into two broad categories, each of which represents a particular aspect of the model. These two aspects of the model are the 'structural' (often referred to as static) and 'behavioural' (often referred to as dynamic, or timing) aspects of the model. It is vital that both of these aspects exist for any system, otherwise the system is not fully defined.

Each of the 13 diagrams belongs to one of these categories. Each of them has strong relationships with other types of diagrams and both aspects of the model have strong relationships with each another. These strong relationships between the types of diagrams and aspects of the model are what allow consistency checks to be performed on UML models, and give confidence in the model itself.

The structural aspect of the model shows the 'things' or entities in a system and the relationships between them. It is crucial to remember that the structural aspect of the model shows 'what' the system looks like and 'what' it does, but not 'how'. A structural aspect of the model may be thought of as a snapshot in time of any system.

Figure 2.1 shows the UML diagrams that relate to structural modelling. The structural aspect of the model may be realised by six UML diagrams: class diagrams, object diagrams, package diagrams, composite structure diagrams, deployment diagrams and component diagrams. Structural modelling is the topic of Chapter 3, while each of the types of structural diagram is discussed in more detail in Chapter 5.

The structural aspect of the model shows the 'what' of the system, while the behavioural aspect of the model shows the 'how'. The behavioural aspect of the model demonstrates how a system behaves over time by showing the order in which things happen, the conditions under which they happen and the interactions between things.

The behavioural aspect of the model may be realised using seven UML diagrams: state machine diagrams, use cases, interaction diagrams (sequence, communication, interaction overview and timing diagrams) and activity diagrams (Figure 2.2).

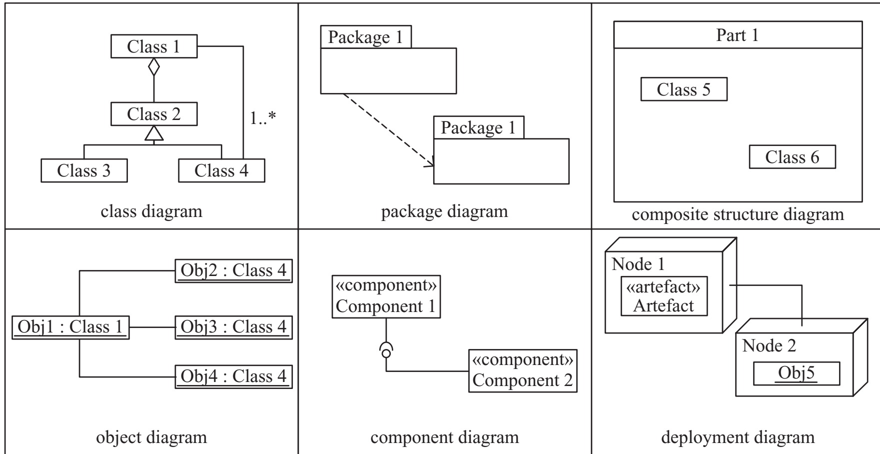


*Figure 2.1 Diagrams that realise a structural aspect of the model in the UML*

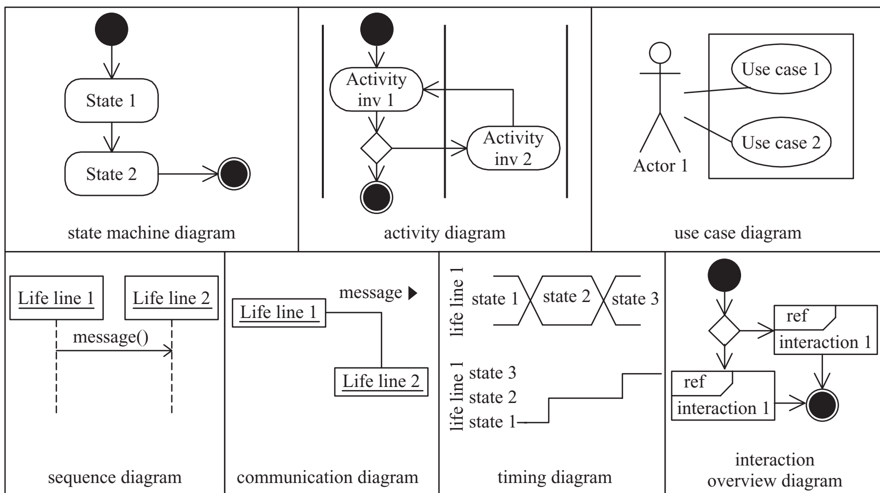


*Figure 2.2 Diagrams that realise the behavioural aspect of the model in the UML*

Behavioural modelling is the subject of Chapter 4, while each of the behavioural diagrams is discussed in more detail in Chapter 5.

### *2.2.3 Views versus models*

Many books refer to the different types of 'view' of a system that must exist throughout the life cycle of the project. This term is often used interchangeably

with the term types of 'model', but for the purposes of this book it is important to distinguish between them. A view refers to a collection of UML diagrams that are created for a specific purpose and that are often related to different phases of the project life cycle. The two aspects of the model that exist are still applicable to all types of view that may be defined. The reason why these views are not discussed in much detail here is that they often constrain development to a particular process or design approach, while this book introduces the UML purely as a language and nothing more. Where appropriate, as for particular applications, these views are described, for example: process views, requirement views, logical views, and so on.

The UML should not change the way in which you approach your work – instead, it should complement it and make it more efficient.

### *2.2.4 Extending the UML*

Despite what others may say, the UML is not some sort of panacea that will cure all your problems. After all, there is no silver bullet and the UML should not be considered as such.

The main shortfall of the UML is seen when it comes to formal, mathematical specification. This criticism is that the UML is not a truly formal approach. Again, this is true, but the UML is intended to be a general-purpose modelling language with a scope that is as wide as possible. In addition, there is nothing to stop the UML being used in conjunction with more formal techniques, such as VDM or Z (formal specification languages), in order to formally specify crucial parts of a system.

The UML was designed to be as flexible as possible and, as such, has a number of built-in extension mechanisms that may be used to extend the functionality and scope of the language itself. These basic mechanisms are stereotypes, constraints and tagged values, which are discussed in detail in Chapter 10.

### *2.2.5 Background of the UML*

This section gives a very brief overview of the background of the UML and discusses where it came from and why.

The UML is a general-purpose visual modelling language that is used to specify, visualise, construct and document the artefacts of any system. These terms have already been defined, but will be mentioned again here:

- To specify means to refer to or state specific needs, requirements or instructions concerned with a complex system.
- To visualise means to realise in a visual fashion or, in other words, to represent information using diagrams.
- To construct means to create a system based on specified components of that system.
- To document means to record the knowledge and experience of the software in order to show how the software was conceived, designed, developed, verified and validated.

The UML is very flexible and has a wide scope that reflects this. The UML may be used for:

*Any development environment*: The UML is not restricted to any particular development environment or language. Although some people claim that the UML is more suited to object-oriented environments, it should be remembered that object orientation is, in one sense, a frame of mind, and that the UML is simply a language. Therefore you should not constrain yourself by thinking that the UML may only be used for object-oriented software systems, as it can be used for any system with any implementation in mind.

*Any process model*: The UML is a language that, unlike its predecessors, does not have an inherent process. There is a defined process that is recommended by the authors of the UML, which is known as the Rational Unified Process (RUP). The RUP is an excellent, well-proven approach to software design, but there is no reason to suggest that this approach must be adopted in order to use the UML successfully. Indeed, Chapter 6 examines process modelling and makes the point that the UML can be applied to any process. Remember, the UML should not change the way that you work, but should make you work more efficiently.

*All life cycle phases*: The UML should not only be considered for analysis and design, but may also be used effectively, at all life cycle phases, from requirements through to operations (or inception to transition, if you are an RUP person). The more that you use the UML, the more benefits will be derived from using the language. It makes sense that if you are using a particular language for requirements, the same language should be used for analysis and design. Apart from anything else, it aids traceability enormously, which from a systems engineering quality point of view is absolutely crucial to the success of a project. The UML should also be utilised after the implementation-related phase, when it can be used to plan and document the installation and transfer of a system and can even be used for user manuals and other documentation.

*Any application domain*: The UML may be used for any application area, even those areas that do not include software development. Historically, many modelling techniques have emerged from a particular industry, which is fine if you happen to work in that industry, but if you move careers or if two separate industries have to talk to each other, problems can arise – the increasingly widespread use of the UML overcomes this.

The UML represents a unification of past experience and methodologies in a single, cohesive language that can be applied to any sort of modelling.

#### 2.2.5.1 History of the UML

The UML is relatively new, having only been fully defined since 1997. Modelling languages, however, have existed for many years in many different forms. There were, though, simply too many modelling languages, methods and methodologies. One of the main reasons for modelling – that of communicating effectively – was being destroyed as, in a bizarre Babel-like scenario, everyone ended up speaking

different languages when they were actually trying to solve this problem. There was thus a huge industrial need to consolidate all these techniques into a single, usable, flexible language. In addition, by making the UML a language rather than a methodology (with an inherent process), it became more generally acceptable and far more powerful than its predecessors.

The UML, therefore, was developed in order to simplify and consolidate the many existing methods that may be grouped into the two main categories of traditional techniques and object-oriented techniques.

#### 2.2.5.2 Traditional modelling techniques

Many traditional development methods exist. The term 'traditional' here refers to the non-object-oriented techniques sometimes referred to as 'classic' techniques.

Some traditional techniques include:

- Structure Analysis and Structured Design (SASD) created by Edward Yourdon and Larry Constantine in 1977. This was considered the 'father' of all other techniques as it was the original visual modelling technique [2].
- Structured Analysis for Real-time Systems, by P T Ward and S Mellor. This approach was based on function-decomposition but with added diagrams to take into account timing relationships in a system [3].

Many other techniques exist which are based mainly on the data flow diagram approach. These techniques were widely accepted from the 1980s onwards and remain in widespread use today. For a discussion of these and other techniques, see Reference 4.

#### 2.2.5.3 Object-oriented modelling techniques

The late 1980s saw the advent of object-oriented (OO) techniques. Although object-oriented languages have been around since the late 1960s (Simula-67), it took 20 years for their associated modelling techniques to be developed.

These techniques include:

- Object-oriented systems analysis, by Shlaer and Mellor in 1988. This was an early analysis approach that made use of a particular type of object diagram, state transition diagrams, tables and domain models [5].
- Object-oriented analysis, by Peter Coad and Edward Yourdon in 1991. This approach used structural diagrams that represented subjects and objects [6].
- The Booch method, by Grady Booch in 1991. One of the most widely adopted approaches and, of course, of the three main authors of the UML, the Booch approach used: class diagrams, object diagrams, module diagrams, process diagrams, state transition diagrams and timing diagrams [7].
- The object modelling technique (OMT) by James Rumbaugh, Michael Blaha, William Lorensen, *et al.* also in 1991. Again, one of the most widely adopted techniques that made use of three main types of diagrams: class diagrams, state machine diagrams and data flow diagrams. The class diagrams from OMT are very similar indeed to those in the UML using much of the same notation [8].

- Object-oriented software engineering (OOSE) by Ivar Jacobson in 1992. Jacobson's main contribution to the world of OO has been the use of use case diagrams that dictated much of the OOSE approach. The use cases that are part of the UML are derived directly from Jacobson's work [9].
- CRC-cards by Rebecca Wirfs-Brock. CRC-cards represent a very pragmatic approach to applying OO techniques and are still widely used today. Indeed, some people still like to use them in conjunction with the UML [10].

Many other techniques also exist and, if you are interested in them, see [11, 12] for a detailed description and comparison of the different techniques.

Anyone familiar with any of the techniques mentioned here or other OO techniques will no doubt recognise elements of them in the UML. This is because the UML is a unification of all these techniques.

#### 2.2.5.4 Unification of different modelling techniques

There were a number of early attempts to unify concepts among the various modelling techniques. Perhaps the most widely known (apart from the UML, that is) was Fusion, by Coleman *et al.* The Fusion method combined elements of Booch, OMT and CRC. However, as none of the original authors was involved, it is judged to be a new method rather than a unification effort.

The first real unification effort came from Rumbaugh and Booch from Rational software corporation in 1994, known as the Unified Method. The following year, they were joined by Ivar Jacobson. This work was then renamed the Unified Modelling Language – the UML.

In 1996, the Object Management Group (OMG) issued a request for proposals for a standard approach to object-oriented modelling. As it turned out, the UML was the sole submission to the OMG as it had been agreed upon by methodologists, industry and CASE (computer aided/assisted software engineering) tool vendors.

As stated previously, Rumbaugh, Booch and Jacobson worked with many other methodologists, including those responsible for the Fusion method, the result of which was a single submission to the OMG in the form of the UML. The UML was accepted as a standard by the OMG in 1997 and responsibility for the UML is now assumed by the OMG [1].

As the UML is now a standard, the information is nonproprietary and thus there is little risk of being tied to a single vendor or supplier when adopting the UML. With the announcement of the acceptance of the UML by the OMG there came a flood of textbooks and CASE tools on the market, of very mixed and variable standards.

The list of backers for the UML is as impressive as it is long. The core team included Hewlett-Packard, I-Logix, IBM, ICON Computing, Intellicorps and James Martin and Company, MCI Systemhouse, Object Time, Oracle, Platinum Technology, Rational Software, Taskon, Texas Instruments and Unisys.

The current version of the UML is UML version 2.0. Despite the fact that UML 2.0 is viewed by many as a radical change from version 1.x, it is not. UML 2.0 is simply a natural evolution of UML 1.x and, yes, there are many changes to the syntax, but the concepts remain the same.

There are many textbooks on the market and many CASE tools. All of these resources vary enormously in quality and cost and it is important to be able to make an informed decision as to where to go next with regard to the UML.

### *2.2.6 A nonproprietary language*

Perhaps one of the most crucial requirements for the UML was to make it nonproprietary so that it is not owned by a single organisation. Although the UML was developed by Rational software, it is the OMG who retain ownership and responsibility for the UML standard itself and, as such, it is in the public domain. This means that you can use any UML-compliant CASE tool to aid modelling, any company (with UML knowledge, obviously) for consultancy and training and choose any textbook from which to learn the UML. If there are any inconsistencies with the resources being used, or any queries with the UML, there is a single source of reference, which is the standard itself.

Another important aspect of the nonproprietary nature of the UML is that there is far less risk of being locked into a single vendor for tools and support. One massive risk to any project is that a single vendor is used to supply all tools and support (training, consultancy, etc.) and this company either goes bust and hence, all support is gone, or doubles its training and consultancy fees overnight, leaving the customer in a very precarious situation.

## 2.3 Conclusions

In summary, therefore, there are many reasons why the UML is truly unified. The UML provides unification:

*Across historical methods and notations*: The UML combines notation and concepts from many modelling techniques. The UML can represent most existing models as well as, if not better than, their existing techniques. Anyone familiar with an existing technique will no doubt recognise some elements of the UML.

*Across development life cycles*: The UML may be used at all phases of the software development life cycle and thus the transition between phases is seamless. As the UML is used more often, the advantages associated with its use will increase.

*Across application domains*: Many existing techniques were focused in a single industry and thus had limited scope. The UML, however, is designed to be applicable to any aspect of complex software-intensive systems.

*Across implementation languages*: The UML is language-independent and thus may be used for any form of implementation. This may relate to a particular software language or may not even apply to software at all.

*Across development processes*: The UML may be used for any development process and, as will be seen in due course, may even be used to define such a process. Although there is a recommended process (the RUP) that is promoted

by the designers of the UML, there is no obligation whatsoever to follow such an approach, as the UML is a language rather than a methodology.

It is established, therefore, that the UML is truly unified. Moreover, it also has the following advantages over all of its predecessors:

*Widely accepted*: The list of industrial supporters is very long (the list provided here being only a subset of full supporters) and is growing all the time. The UML is industry driven and satisfies a real industrial need.

*Nonproprietary*: This is crucial from a project risk point of view as the risk of being tied to a single supplier or vendor is minimised.

*Commercially supported*: There are many tools and textbooks on the market to support UML efforts.

*Extendable*: Although the UML is no 'silver bullet', it is designed to be as flexible as possible and to be extendable for specific applications. Chapter 10 discusses extending the UML in more detail.

All of these points contribute towards making the UML the most flexible and widely used modelling language today, which has superseded all previous techniques and notations.

The UML is the industry standard for software engineering and is being increasingly used for systems engineering. Knowledge of the UML is becoming more important, to the point of becoming essential for all systems engineers.

## 2.4 Further discussion

1. Think of some examples of the three types of model that were discussed: the kennel, the house and the office block.
2. Consider the office block and try to think of different types of model that would be required to construct the building. What different teams of people would be involved?
3. What levels of abstraction would be required for the house? Who would require these different levels of abstraction?
4. Consider the mathematical operations of differentiation and integration. What can they actually be used for in real life? How would you compare their connection to reality?
5. Consider visual modelling and formal mathematical modelling. Which would be the most appropriate for representing the architecture of a system and which for representing a numerical simulation of the same system?

## 2.5 References

1 BOOCH, G., RUMBAUGH, J., and JACOBSON, I.: 'The unified modelling language user guide' (Addison-Wesley, Massachusetts, 1999)

2 YOURDON, E., and CONSTANTINE, L. L.: ‘Structured design: fundamentals of a discipline of computer program and systems design’ (Prentice-Hall, Englewood Cliffs, NJ, 1979)
3 WARD, P. T., and MELLOR, S.: ‘Structured development for real-time systems’ (Yourdon Press, New York, 1985)
4 SCHACH, S. R.: ‘Classical and object-oriented software engineering’ (Irwin, Singapore, 1996)
5 SHLAER, S., and MELLOR, S. J.: ‘Object-oriented systems analysis – modelling the world in data’ (Yourdon Press, Englewood Cliffs, NJ, 1988)
6 COAD, P., and YOURDON, E.: ‘Object-oriented analysis’ (Yourdon Press, Englewood Cliffs, NJ, 1991)
7 BOOCH, G.: ‘Object-oriented analysis and design, with applications’ (The Benjamin/Cummings Publishing Company Inc., California, 1994)
8 RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., and LORENSON, W.: ‘Object-oriented modelling and design’ (Prentice-Hall, Englewood Cliffs, NJ, 1991)
9 JACOBSON, I.: ‘Object-oriented software engineering, a use case-driven approach’ (Addison-Wesley Publishing Company, Washington, 1992)
10 WIRFS-BROCK, R., WILKERSON, B., and WIENER, L.: ‘Designing object-oriented software’ (Prentice-Hall, New York, 1990)
11 GRAHAM, I.: ‘Object-oriented methods’ (Addison-Wesley Publishing Company, London, 1994)
12 SCHACH, S. R.: ‘Software engineering with Java’ (McGraw-Hill International Editions, Singapore, 1997)

# Chapter 3
# Structural modelling

(Camelot!) It's only a model

*Patsy*

## 3.1 Introduction

Previously we have discussed the point that each system must be modelled in two different aspects. This chapter examines one of these aspects: the structural aspect of the model.

Figure 3.1 shows the six types of diagrams that can be used to realise structural models: class diagrams, package diagrams, object diagrams, composite structure diagrams, component diagrams and deployment diagrams.

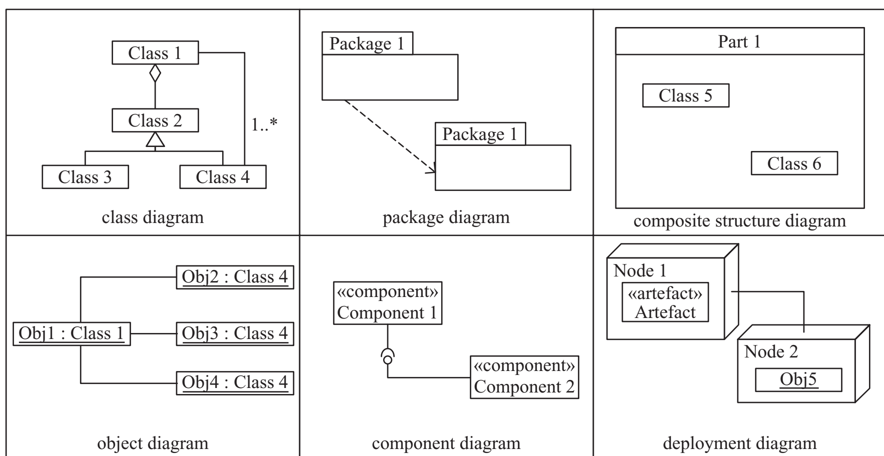


*Figure 3.1 Types of structural model*

In order to illustrate the concepts behind structural modelling, one of the six structural diagrams will be used to show some simple examples. The diagram chosen is the class diagram as this forms the backbone of the Unified Modelling Language (UML). In addition, the structural modelling principles shown in this section with reference to class diagrams can be applied when using any type of structural model. For more in-depth discussion concerning class diagrams and structural modelling, see References 1–3.

## 3.2 Structural modelling using class diagrams

### *3.2.1 Modelling classes and relationships*

The class diagram is, arguably, the most widely used diagram in the UML. Class diagrams have a long history and are present in some shape or form in all of the other methodologies that were mentioned previously in this part of the book. The class diagram is also the richest diagram in terms of the amount of syntax available to the modeller. As with all UML diagrams, it is not necessary to use every piece of syntax, as experience has shown that 80 per cent of any modelling task can be achieved by using approximately 20 per cent of class diagram syntax. The simplest (and wisest) approach is to learn the bare minimum syntax and then to learn more as and when circumstances dictate.

### *3.2.2 Basic modelling*

There are two basic elements that make up a class diagram, which are the 'class' and the 'relationship' and, at a very simple level, that is it! Clearly, there are many ways to expand on these basic elements, but providing that they are understood clearly and simply, the rest of the syntax follows on naturally and is very intuitive.

A 'class' represents a type of 'thing' that exists in the real world and, hence, should have a very close connection to reality. Classes are almost always given names that are nouns, as nouns are 'things' and so are classes. This may seem a trivial point, but it can form a very powerful heuristic when assessing and analysing systems as it can be an indicator of whether something may appear as a class on a model.

The second element in a class diagram is a relationship that relates together one or more classes. Relationships should have names that form sentences when read together with their associated classes. Remember that the UML is a language and should thus be able to be 'read' as one would read any language. If a diagram is difficult to read, it is a fairly safe bet that it is not a very clear diagram and should perhaps be 'rewritten' so that it can be read more clearly. Reading a good UML diagram should not involve effort or trying, in the same way that any sentence should not be difficult to read.

Now that the basics have been covered, it is time to look at example class diagrams. It should be pointed out that from this point, all diagrams that are shown will be UML diagrams. In addition, although some of the diagrams may seem trivial, they are all legitimate and correct diagrams and convey some meaning – which is the whole point of the UML!

*Figure 3.2 Representing classes*

Figure 3.2 shows two very simple classes. Classes are represented graphically by rectangles in the UML and each must have a name, which is written inside the rectangle. In order to understand the diagram, it is important to read the symbols. The diagram here shows that two classes exist: ‘Class 1’ and ‘Class 2’. This is the first UML diagram in the book and, if you can read this, then you are well on the way to understanding the UML.

*Figure 3.3 Representing a relationship*

Figure 3.3 shows how to represent a relationship between two classes. This particular relationship is known as an ‘association’ and is simply a general type of relationship that relates together one or more classes. The association is represented by a line that joins two classes, with the association name written somewhere on the line. This diagram reads: two classes exist: ‘Class 1’ and ‘Class 2’ and ‘Class 1’ associates ‘Class 2’.

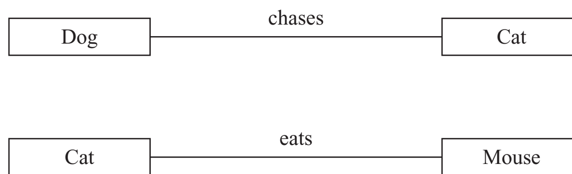


*Figure 3.4 Examples of classes and associations*

Figure 3.4 shows two more examples that are based on real life. The top part of the diagram reads: there are two classes: ‘Dog’ and ‘Cat’ where ‘Dog’ chases ‘Cat’. Likewise, the lower part of the diagram reads: there are two classes: ‘Cat’ and ‘Mouse’ where ‘Cat’ eats ‘Mouse’.

This illustrates a very important point concerning classes, as classes are abstract and do not actually exist in the real world. There is no such thing as ‘Cat’, but there do exist many examples of ‘Cat’. A class represents a grouping of things that look and behave in the same way, as, at one level, all examples of ‘Cat’ will have a common set of features and behaviours that may be represented by the class ‘Cat’. What this class is really representing is the blueprint of ‘Cat’, or the essence of ‘Cat’.

Another important point illustrated here is that every diagram, no matter how simple, has the potential to contain a degree of ambiguity. Figure 3.4 is actually

ambiguous as it could be read in one of two ways, depending on the direction in which the association is read. Take, for example, the top part of the diagram: who is to say that the diagram is to be read 'Dog' chases 'Cat' rather than 'Cat' chases 'Dog', as it is possible for both cases to be true. It should be pointed out that for this particular example the world that is being modelled is a stereotypical world where dogs always chase cats, and not the other way around. Therefore, there is some ambiguity as the diagram must only be read in one direction for it to be true; thus, a mechanism is required to indicate direction.

The simplest way to show direction is to place a direction marker on the association that will dictate which way the line should be read, as shown in the top part of Figure 3.5. The diagram now reads 'Dog' chases 'Cat' and definitely not 'Cat' chases 'Dog' and is thus less ambiguous than Figure 3.4.

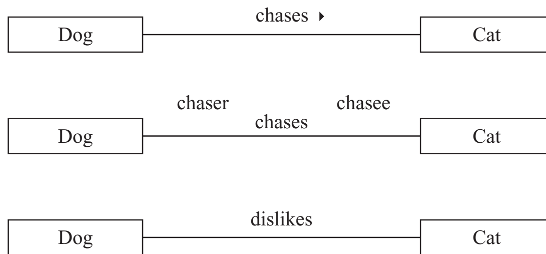


*Figure 3.5 Showing direction*

The second way to show direction is to define a 'role' on each end of the association, as shown in the middle part of Figure 3.5. In this case, the two roles that have been defined are 'chaser' and 'chasee', which again eliminates the ambiguity that existed in Figure 3.4.

The lower part of Figure 3.5 introduces a new association called 'dislikes'. This time, however, the lack of direction is intentional as both statements of 'Dog' dislikes 'Cat' and 'Cat' dislikes 'Dog' are equally true. Therefore, when no direction is indicated, it is assumed that the association can be read in both directions or, to put it another way, the association is said to be 'bi-directional'.

Another reason why the diagram may be misunderstood is because there is no concept of the number of cats and dogs involved in the chasing of the previous diagrams. Expressing this numbering is known as 'multiplicity', which is illustrated in Figure 3.6.

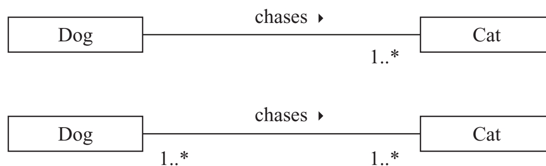


*Figure 3.6 Showing numbers using multiplicity*

The top part of Figure 3.6 shows that each ‘Dog’ chases one or more ‘Cat’. If no number is indicated, as in the case of the ‘Dog’ end of the association, it is assumed that the number is ‘one’. Although the number is one, it does not necessarily indicate that there is only one dog, but rather that the association applies for each dog. The multiplicity at the other end of the ‘chases’ association states ‘1..*’, which means ‘one or more’ or somewhere between one and many. Therefore, the association shows that each ‘Dog’ chases one or more ‘Cat’.

The lower part of the diagram shows a case where the multiplicity has been changed, which changes the entire meaning of the model. In this case, the diagram is read as: one or more ‘Dog’ chases one or more ‘Cat’. This could mean that a single dog chases a single cat, a single dog chases any number or a herd of cats, or that an entire pack of dogs is chasing a herd of cats.

Which of the two examples of multiplicity is correct? The answer will depend on the application that is being modelled and it is up to the modeller to decide which is the more accurate of the two.

### *3.2.3 Adding more detail to classes*

So far, classes and relationships have been introduced, but the amount of detailed information for each class is very low. After all, it was said that the class ‘Cat’ represented all cats that looked and behaved in the same way, but it is not defined anywhere how a cat looks or behaves. This section examines how to add this information to a class by using ‘attributes’ and ‘operations’.

Consider again the class of ‘Cat’, but now think about what general properties the cat possesses. It is very important to limit the number of general properties that are identified to only those that are relevant, as it is very easy to get carried away and over-define the amount of detail for a class.

For this example, suppose that we wish to represent the features ‘age’, ‘weight’, ‘colour’ and ‘favourite food’ on the class ‘Cat’. These features are represented on the class as ‘attributes’ – one for each feature (Figure 3.7).

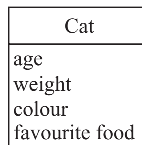


*Figure 3.7 Attributes of the class ‘Cat’*

Attributes are written in a box below the class name box. When modelling software, it is possible to add more detail at this point, such as the visibility of the attribute, type, default values, and so forth. As attributes represent features of a class, they are usually represented by nouns and they must also be able to take on different values. For example, ‘colour’ is a valid attribute, whereas ‘red’ would not be in most cases, as ‘red’ would represent an actual value of an attribute rather than an attribute itself. It is possible for ‘red’ to be an attribute, but this would mean that

the attribute would have a Boolean type (true or false) to describe a situation where we would only be interested in red cats and not any other type.

Attributes thus describe how to represent features of a class, or to show what it looks like, but they do not describe what the class does, which is represented using 'operations'. Operations show what a class does, rather than what it looks like, and are thus usually represented by verbs. Verbs, as any schoolchild will be able to tell you, represent 'doing' words, and operations describe what a class 'does'. In the case of the class 'Cat' we have identified three things that the cat does, which are 'eat', 'sleep' and 'run' (Figure 3.8).

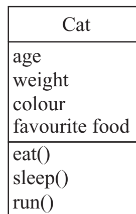


*Figure 3.8 Operations of the class 'Cat'*

Operations are represented in the UML by adding another rectangle below the attribute rectangle and writing the operation names within it. When modelling software, operations may be defined further by adding extra detail – for example, arguments, return values, visibility and so forth.

The class 'Cat' is now fully defined for our purposes and the same exercise may be carried out on any other classes in the diagram in order to fully populate the model. It should also be pointed out that the classes may be left at a high level with no attributes or operations. As with everything in the UML, only use as much detail as is necessary, rather than as much as is possible.

### *3.2.4 Adding more detail to relationships*

Section 3.1.2.1 showed how to add more detail to classes, while this section shows how to add more detail to relationships, by defining some special types that are commonly encountered in modelling. There are four types of relationships that will be discussed here: 'association', 'aggregation', 'specialisation' and 'instantiation'. Many types of relationships exist, but these four represent the majority of the most common uses of relationships.

Associations have already been introduced and shown to be a very general type of relationship that relate together one or more class. Therefore, they will not be discussed in any further detail, so the next three sections cover each of the other special types of relationships.

#### **3.2.4.1 Aggregation and composition**

The second type of relationship is a special type of association that allows assemblies and structures to be modelled and is known as 'aggregation'.

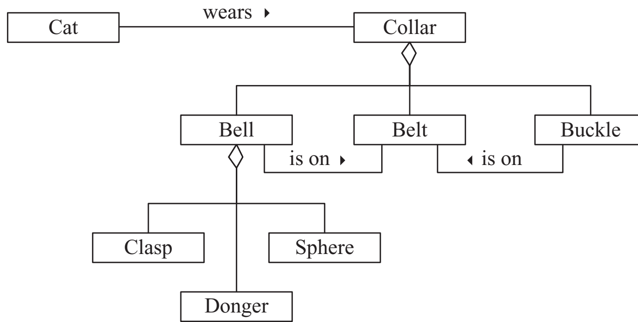


*Figure 3.9 Example of aggregation*

Figure 3.9 provides an example of aggregation. Aggregation is shown graphically in the UML by a diamond or rhombus shape and when reading the diagram, is read by saying 'is made up of'. Starting from the top of the diagram, the model is read as: 'Cat' wears 'Collar'. The direction is indicated with the small arrow and there is a one on one relationship between the two classes. The multiplicity here is implied to be one to one as there is no indication otherwise. This makes sense as it is very unlikely that one cat would wear one or more collar and almost impossible to imagine that more than one cat would wear a single collar!

The 'Collar' is made up of (the aggregation symbol) a 'Bell', a 'Belt' and a 'Buckle'. The 'Bell' is on the 'Belt' and the 'Buckle' is on the 'Belt'.

The 'Bell' is made up of (the aggregation symbol) a 'Clasp', a 'Donger' and a 'Sphere'.

This is the basic structure of the bell and allows levels of abstraction of detail to be shown on a model.

There is also a second special type of association that shows an aggregation-style style of relationship, known as 'composition'. The difference between composition

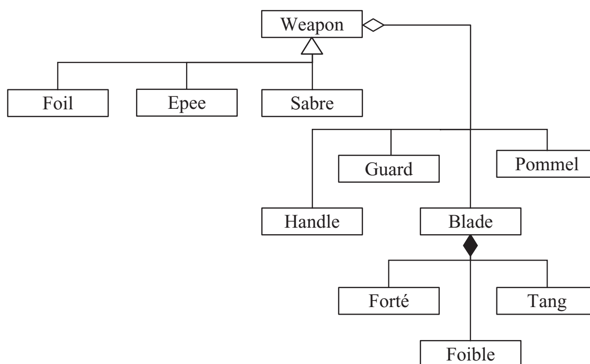


*Figure 3.10 Example of the difference between composition and aggregation*

and aggregation is subtle but very important and can convey much meaning. The simplest way to show this difference is to consider an example, as shown in Figure 3.10.

This model represents the structural aspect of the types of weapon that may be used in the sport of fencing (as opposed to putting up fences or selling stolen goods). From the model, there are three types of 'Weapon': 'Foil', 'Epée' and 'Sabre' (this is a 'type of', or 'specialisation' relationship that will be discussed in a subsequent section). Each weapon is made up of a 'Handle', a 'Pommel', a 'Blade' and a 'Guard'. The 'Blade' is made up of a 'Forté', a 'Foible' and a 'Tang'. There is a clear visual difference here as the aggregation symbol under 'Blade' is filled in, rather than empty as in the case of the aggregation under 'Weapon'. The semantic difference, however, is that an aggregation is made up of component parts that may exist in their own right. It is possible to buy or make any of the components under the class 'Weapon' as they are assembled into the completed class 'Weapon'. The class 'Blade', however, has three components that cannot exist independently of the class 'Blade'. This is because a fencing blade is a single piece of steel that is composed of three distinct sections. For example, there is no such thing as a 'Foible' as it is an inherent part of the 'Blade' rather than being an independent part in its own right.

### 3.2.4.2 Specialisation and generalisation

The third special type of relationship is known as 'specialisation and generalisation', depending on which way the relationship is read. 'Specialisation' refers to the case when a class is being made more special or is being refined in some way. Specialisation may be read as 'has types' whenever its symbol, a small triangle, is encountered on a model. If the relationship is read the other way around, then the triangle symbol is read as 'is a type of', which is a generalisation. Therefore, read one way the class becomes more special (specialisation) and read the other way, the class becomes more general (generalisation).

Specialisation is used to show 'child' classes, sometimes referred to as subclasses, of a 'parent' class. Child classes inherit their appearance and behaviour from their parent classes, but will be different in some way in order to make them special. In UML terms, this means that a child class will inherit any attributes and operations that its parent class has, but may have additional attributes or operations that make the child class special.

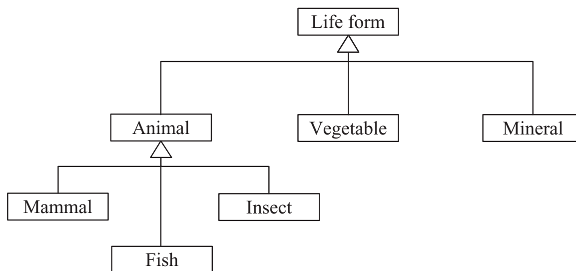


*Figure 3.11 Life form hierarchy*

As an example of this, consider Figure 3.11, which shows different types of life known to man. The top class is called 'Life form' and has three child classes: 'Animal', 'Vegetable' and 'Mineral', which makes 'Life form' the parent class. Going down one level, it can be seen that 'Animal' has three child classes: 'Mammal', 'Fish' and 'Insect'. Notice now how 'Animal' is the parent class to its three child classes, while still being a child class of 'Life form'.

The diagram may be read in two ways:

- From the bottom up: 'Mammal', 'Fish' and 'Insect' are types of 'Animal'. 'Animal', 'Vegetable' and 'Mineral' are all types of 'Life form'.
- From the top down: 'Life form' has three types: 'Animal', 'Vegetable' and 'Mineral'. The class 'Animal' has three types: 'Mammal', 'Fish' and 'Insect'.

This explains how to read the diagrams, but does not cover one of the most important concepts associated with generalisation and specialisation, which is the concept of inheritance, best illustrated by considering the diagram in Figure 3.12.

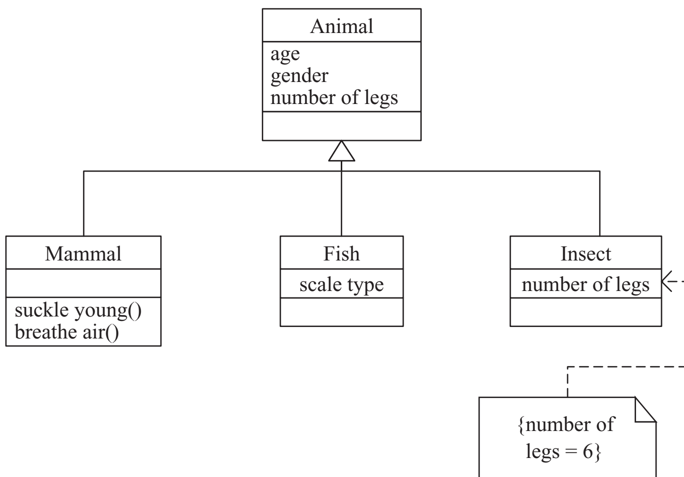


*Figure 3.12 Example of inheritance*

This figure shows an expanded version of Figure 3.11 by adding some attributes and operations to the classes. It can be seen that the class 'Animal' has three identifiable attributes: 'age', 'gender' and 'number of legs'. These attributes will apply to all types of animals and will therefore be inherited by their child classes. That is to say, any of the child classes will automatically have the same three attributes. This inheritance also applies to the behaviour of the parent class; therefore, if some operations had been defined for the class 'Animal', these would also be inherited by each of the child classes. What makes the child class different or special and therefore an independent class in its own right, is the addition of extra attributes and operations or constraints on existing attributes. Let us consider an example of each one of

these cases:

- The class 'Mammal' has inherited the three attributes from its parent class. Inherited properties (attributes and operations) are not usually shown on child classes. In addition, it has had an extra two operations identified, which the parent class does not possess: 'breathe air' and 'suckle young'. This is behaviour that all mammals possess, whereas not all animals do, or to put it another way, the reason why mammals are special compared to animals generally.
- The class 'Fish' has inherited the three attributes from its parent class (which are not shown), but has had an extra attribute added that its parent class will not possess: 'scale type'. This makes the child class more specialised than the parent class.
- The class 'Insect' has no extra attributes or operations but has a constraint on one of its attribute values. The attribute 'number of legs' is always equal to six, as this is in the nature of insects. The same could be applied to 'Mammal' to some extent as 'number of legs' would always be between zero (in the case of whales and dolphins) and four, while 'number of legs' for 'Fish' would always be zero. Such a limitation is known in the UML as a 'constraint', which is one of the standard extension mechanisms for the UML and which is covered in some detail in Chapter 10.

From a modelling point of view, it may be argued that the attribute 'number of legs' should not be present in the class 'Animal' as it is not applicable to fish. This is fine and there is nothing inherently wrong with either model, however, it is important to pick the most suitable model for the application at hand. Remember that there are many correct solutions to any problem, and thus people's interpretation of information may differ. By the same token, it would also be possible to define two child classes of 'Animal' called 'Male' and 'Female', which would do away with the need for the attribute 'gender' as shown in Figure 3.13.

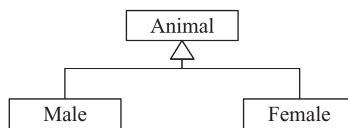


*Figure 3.13 Another way to model gender*

The model in Figure 3.13 shows another approach to modelling the gender of an animal. Which approach is the better of the two, the one shown in Figure 3.12 or the one shown in Figure 3.13? Again, it is necessary to pick the most appropriate visual representation of the information and one that you, as the modeller, are comfortable with.

As a final example, let us revisit our old friend the cat who is undeniably a life form, an animal and also a mammal. This gives three layers of inheritance, so that the class 'Cat' may now be represented as in Figure 3.14.

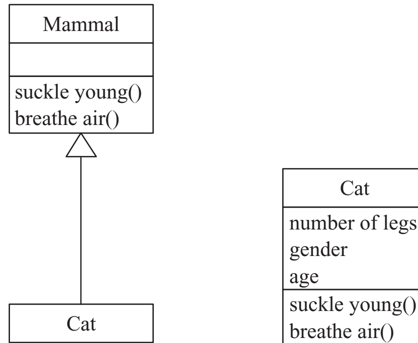


*Figure 3.14 A cat's inheritance*

The model in Figure 3.14 shows an example of the cat that is now applied to the life form hierarchy. Here, it can be seen that 'Cat' is a type of 'Mammal' and therefore inherits all its features and behaviour from its parent class. The class 'Cat' shown on the right-hand side of the diagram shows all the inherited attributes and operations from its parent classes. Note the inclusion of the three attributes that were inherited from 'Animal', which, although the class is not included on this particular diagram, still influences the child classes, no matter how far removed.

#### 3.2.4.3 Instantiation

The final type of relationship that will be defined in this book is known as 'instantiation', which is a way of showing real-life examples of a particular class. These examples are known as 'instances' and an instance of a class is also known as an 'object'.

Strictly speaking, the instantiation relationship is actually a special type of relationship that is known as a 'dependency'. A dependency, as the name implies, relates two classes together but indicates that if one class changes in any way, so will its dependent class. A dependency relationship is shown graphically by using a dashed directed line, which indicates the direction of the dependency. Several other types of dependency exist, but they are not covered in this book. Clearly, an instantiation is a dependency relationship since an object is entirely dependent on its class to define its features and behaviour.

Instances are represented in the UML by underlining the class name and showing the instance name next to the class name, separated by a colon, as shown in Figure 3.15. It is also possible to use shorthand to represent some instances.

From the model, it can be seen that the class 'Cat' has six instances: 'TC', 'Brains', 'Choo-Choo', 'Benny-the-Ball', 'Fancy' and an extra instance with no name. These instances are represented in three ways:

- The full representation of the instance: instance name, a colon and the class name all underlined. This is shown by "TC : Cat" and "Brains : Cat".

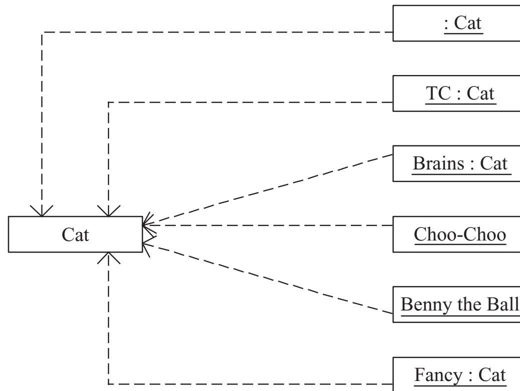


*Figure 3.15 Example of instantiation*

- An abbreviated instance, where only the instance name is shown, as in 'Choo-Choo' and 'Benny-the-Ball'. This leaves room for ambiguity, however, as it is assumed that whoever is reading the diagram has knowledge concerning the classes of these instances. This is useful if all the instances on a particular model are of the same class and serves as a type of shorthand.
- The final example of the class 'Cat' is an anonymous instance, which means that although there is definitely a real-life example of the class 'Cat', there is no specific name to differentiate it from other examples of the class 'Cat'.

One point that should be borne in mind here is that the instance name is a unique identifier, rather than the actual name of the cat. There should only be one instance for each different identifier, as these identifiers need to be unique. What would happen, however, if an instance were named after the actual cat's name (TC for example) and the other cat appeared with the same name? It is impossible to have two objects, or instances, that are both called TC, so how can this be resolved? The answer would be to add another attribute called 'name' and then fill it in for each object, making the word chosen for the unique identifier irrelevant.

Instantiation completes the list of types of relationship that are introduced in this book, but remember that there are many other types that are used for specific applications or under particular conditions. References to these can be found in any other UML book, and a number of them are provided at the end of this chapter.

## 3.3 Other structural diagrams

This chapter is intended to introduce structural modelling as a concept, but has so far only considered one of the six structural diagrams: the class diagram. However, the concepts introduced here, that of things and the relationships between them, apply to each of the structural diagrams. Therefore, if you can understand the concepts used here, then it is simply a matter of learning some syntax and some example

applications for the other types of diagrams. Indeed, this is achieved in Chapter 5 where each of the nine UML diagrams is described in terms of its syntax and its typical uses. The important thing to realise about all six types of structural diagram is that they all represent the 'what' of a system. These other five diagrams are the object diagram, package diagram, composite structure diagram, component diagram and the deployment diagram.

## 3.4 Conclusion

In order to conclude this section and to see if structural modelling really works, a summary of this section is presented in the class diagram (Figure 3.16).

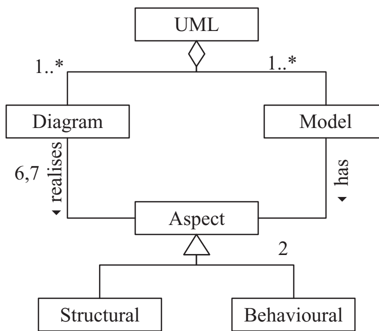


*Figure 3.16 Summary of UML models and their associated diagrams*

Figure 3.16 shows a summary of the UML models and how they may be realised using the various types of diagram. The UML is made up of one or more 'Diagram' and one or more 'Model'. Each 'Model' has two 'Aspect' whereas six or seven 'Diagram' realise each 'Aspect'. There are two types of 'Aspect' – 'Structural' and 'Behavioural'.

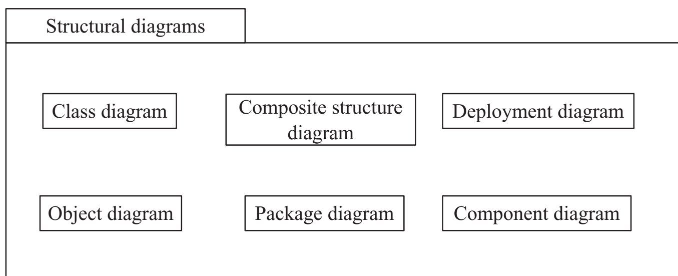


*Figure 3.17 Package of the types of structural model*

The actual types of diagrams are not shown here, but are shown in Figure 3.17, which introduces a package that groups together the various types of structural models.

This figure shows the various types of structural diagrams. It can be seen from the model that the package 'Structural' (a simple grouping) contains 'Class diagram', 'Object diagram', 'Package diagram', 'Composite structure diagram', 'Deployment diagram' and 'Component diagram'.

The actual elements that made up the class diagram can also be modelled using a class diagram. This is a concept known as the 'meta-model', which will be used extensively in Chapter 5.

Remember that the UML is a language and, therefore, should be capable of being read and understood just like any other language. Therefore, the model in Figure 3.18 may be read by reading class name, then any association that is joined to it, and then another class name.

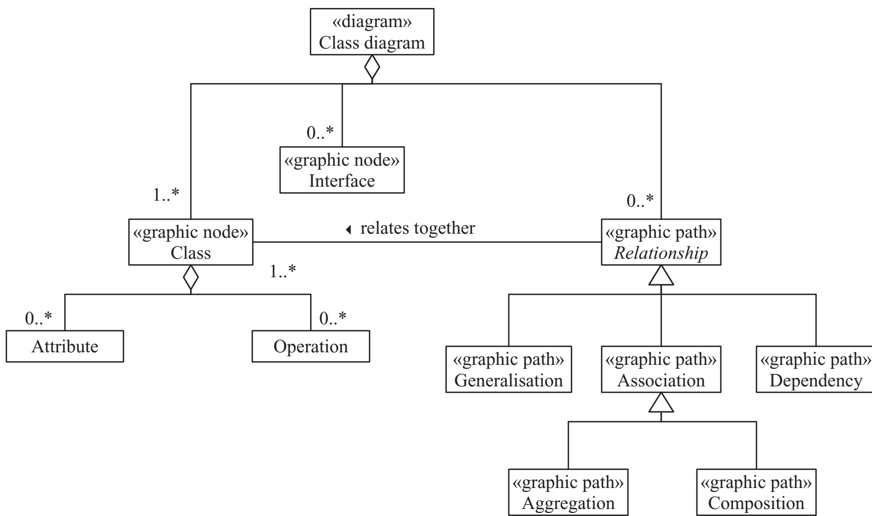


*Figure 3.18 Meta-model for a class diagram*

Therefore, reading the diagram: a 'Class diagram' is made up of one or more 'Class', zero or more 'Interface' and zero or more 'Relationship'. It is possible for a class diagram to be made up of just a single class with no relationships at all; however, it is not possible to have a class diagram that is made up of just a relationship with no classes. Therefore, the multiplicity on 'Class' is one or more, whereas the multiplicity on 'Relationship' is zero or more.

Each 'Relationship' relates together one or more 'Class'. Notice that the word 'each' is used here, which means that for every single 'Relationship' there are one or more 'Class'. It is also interesting to note that the multiplicity on the 'Class' side of the association is one or more, as it is possible for a 'Relationship' to relate together one 'Class' – that is to say that a 'Class' may be related to itself.

Each 'Class' is made up of zero or more 'Attribute' and zero or more 'Operation'. This shows how classes may be further described using attributes and operations, or that they may be left with neither.

There are three main types of 'Relationship':

- 'Association' that defines a simple association between one or more 'Class'. There are also two specialisations of 'Association': a special type of 'Association' known as 'Aggregation' and one known as 'Composition', which show 'is made up of' and 'is composed of' associations respectively.
- 'Generalisation' that shows a 'has types' relationship used to show parent and child classes.
- 'Dependency' that has only one type (of many) shown, called 'Instantiation', which describes an 'is an example of' relationship.

This meta-model concept may be extended to include all the other types of structural diagrams and, indeed, this is exactly the approach adopted in Chapter 5 to introduce all the other UML diagram types.

## 3.5 Further discussion

1. Create a new class based on an everyday object and model its components. Should the relationships be aggregation or composition?
2. Define a new type of 'Mammal' and represent it as a specialisation. Define some new attributes and operations that will make it special when compared to its parent class.
3. Define a completely new hierarchy of animals, but rather than defining types based on genetics, create new types based on the way that they move around. For example, land-based, air-based and water-based. How do the types of animal already defined fit in here? Indeed, do they still fit in?
4. Define some new instances of the new types of 'Mammal'. Show the attributes on each object and fill in the values of each attribute.

## 3.6 References

1 BOOCH, G., RUMBAUGH, J., and JACOBSON, I.: 'The unified modelling language user guide' (Addison Wesley, Massachusetts, 1998)
2 RUMBAUGH, J., JACOBSON, I., and BOOCH, G.: 'The unified modelling language reference manual' (Addison Wesley, Massachusetts, 1998)
3 STEVENS, P., and POOLEY, R.: 'Using UML' (Addison Wesley, Harrow, 1999)

*Chapter 4*

# Behavioural modelling

Oh, behave
*Austin Powers*

## 4.1 Introduction

Chapter 3 introduced structural modelling, one of the two basic aspects of modelling using the Unified Modelling Language (UML), by choosing one type of structural diagram and using it to explain basic principles that may then be applied to any sort of structural modelling. This chapter takes the same approach, but with behavioural modelling.

Behavioural models may be realised using seven types of UML diagram, which are: use case diagrams, state machine diagrams, activity diagrams and interaction diagrams, of which there are four types – communication, timing, interaction overview and sequence diagrams, as shown in Figure 4.1.

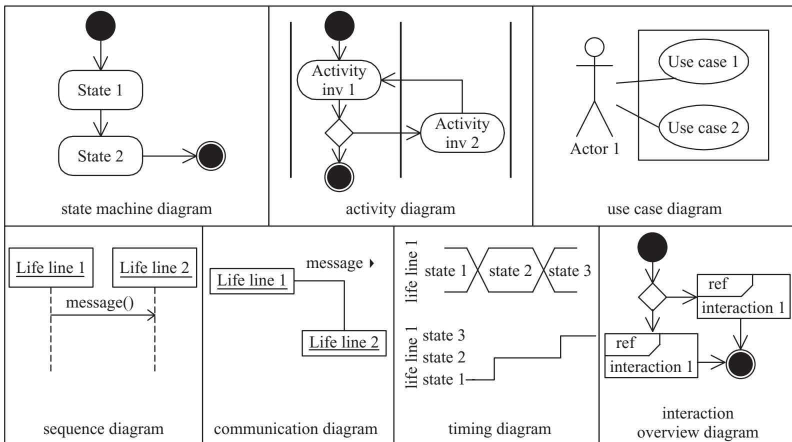


*Figure 4.1 The seven diagrams for realising the behavioural aspect of the model*

It is stated in Chapter 3 that structural modelling defines the 'what' of a system: what it looks like, what it does and what the relationships are. If structural modelling tells us 'what', then behavioural modelling tells us 'how'. This 'how' is described by modelling interactions within a system. These interactions may be modelled at many levels of abstraction; different types of behavioural diagrams allow the system to be modelled at different levels of abstraction. These levels of abstraction may be categorised as follows:

- Interactions may be modelled between objects or between subsystems. Such models are realised using the four types of interaction diagram: communication, timing, interaction overview or sequence diagram.
- Interactions may also be modelled at a slightly lower level of abstraction by considering the interactions within a class or its associated objects. Modelling the behaviour over the lifetime of a class, or an object, is achieved by using state machine type of diagrams. State machine diagrams are concerned with one, and only one, class and its objects, although it is possible to create more than one state machine diagram per class.
- The lowest level of abstraction for modelling interactions is concerned with the interactions within an operation or, in computer terms, at the algorithmic level. Interactions at the algorithmic level are modelled using activity diagrams. Activity diagrams may also be used to model interactions at a higher level of abstraction, which will be discussed in greater detail in Chapter 5.
- The remaining level of abstraction is at the highest possible level, which is the context level of the project, where high-level functionality of the system and its interaction with the outside world is modelled. These high-level interactions are realised using use case diagrams.

The diagram chosen to illustrate behavioural modelling is the state machine diagram. There are a number of reasons for choosing this behavioural diagram over the other four.

- State machine diagrams are one of the most widely used diagrams in previous modelling languages and, as such, people tend to have seen something similar to state machine diagrams, if not an actual state machine diagram. This makes the process of learning the syntax easier.
- State machine diagrams have a very strong relationship with class diagrams, which were discussed in Chapter 3.

It will be seen later in this chapter that behavioural modelling using state machine diagrams leads directly into other types of behavioural modelling diagrams. For more in-depth discussion concerning state machine diagrams and behavioural modelling, see References 1–3.

### *4.1.1 Behavioural modelling using state machine diagrams*

#### 4.1.1.1 Introduction

State machine diagrams are used to model behaviour over the lifetime of a class, or to put it another way, they describe the behaviour of objects. Both of these statements

are true and are used interchangeably in many books. Remember that objects are instances of classes and thus use the class as a template for the creation of its objects, which includes any behaviour.

The most obvious question to put forward at this point is 'does every class need to have an associated state machine diagram?' The simple answer is 'no', as only classes that exhibit some form of behaviour can possibly have their behaviour defined. Some classes will not exhibit any sort of behaviour, such as data structures or database structures. The simple way to spot whether a class exhibits any behaviour is to see whether it has any operations; if a class has operations, it does something, if it does not have any operations, it does nothing. If a class does nothing, it is impossible to model the behaviour of it. Therefore, a simple rule of thumb is that any class that has one or more operations must have its behaviour defined using, for example, a state machine diagram.

#### 4.1.1.2 Basic modelling

The basic modelling elements in a state machine diagram are states, transitions and events. States describe what is happening within a system at any given point in time, transitions show how to change between such states and events dictate which messages are passed on these transitions. Each of these elements will now be looked at in more detail, starting with the state, an example of which is shown in Figure 4.2.

*Figure 4.2 A UML representation of a state*

This figure shows a very simple state, which is shown in the UML by a box with rounded corners. This particular state has the name 'State 1' and this diagram should be read as: 'there is a single state, called "State 1" '. This shows what a state looks like, but what exactly is a state? The following four points discuss the basics of a state:

- A state may describe situations in which the system satisfies a particular condition, in terms of its attribute values or events that have occurred. This may, for example, be 'loaded' or 'saved', so that it gives an indication as to something that has already happened. States that satisfy a particular condition tend to be used when an action-based approach is taken to creating state machine diagrams. This will be discussed in more detail in due course.
- A state may describe a situation in which the system performs a particular activity or set of actions, or to put it another way, is actually doing something. States are assumed to take a finite amount of time, whereas transitions are assumed to take no time. There are two things that can be happening during such a state: one or more activities and/or one or more actions. An activity is nonatomic and, as such,

can be interrupted, hence, it takes up a certain amount of logical time. Actions, on the other hand, are atomic and, as such, cannot be interrupted and, hence, take up zero logical time. Therefore, activities can only appear 'inside' a state, whereas an action can exist either within a state or on a transition. Activities can be differentiated from actions inside states by the presence of the keyword 'do/', whereas action will have other keywords, including: 'entry/' and 'exit/'.

- A state may also describe a situation in which a system does nothing or is waiting for an event to occur. This is often the case with event-driven systems, such as windows-style software where, in fact, most of the time the system is sitting idle and waiting for an event to occur.
- There are different types of states in the UML; however, the states used in state machine diagrams are known as normal states. This implies that complexity may exist within a state, such as: a number of things that may happen, certain actions that may take place on the entry or exit of the state, and so forth.

In order for the object to move from one state to another, a transition must be crossed. In order to cross a transition, some sort of event must occur. Figure 4.3 shows a simple example of how states and transitions are represented using the UML.

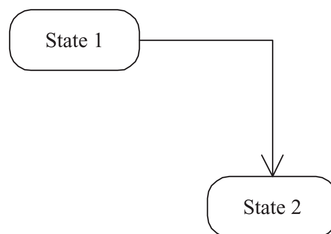


*Figure 4.3 States and transitions*

From the model in Figure 4.3 it can be seen that two states exist: 'State 1' and 'State 2', represented by rounded boxes. There is a single transition that goes from 'State 1' to 'State 2', which is represented by a directed line showing the direction of the transition. These transitions are unidirectional and, in the event of another transition being required going in the other direction, an entirely new transition is required – the original transition cannot be made bi-directional.

In order to cross a transition, which will make the object exit one state and enter another, an event must occur. This event may be something simple, such as the termination of an activity in a state (the state has finished what it is doing), at which point it leaves its present state, or may be more complex and involve receiving messages from another object in another part of the system. Event names are written on the transition lines.

State machine diagrams will now be taken one step further by considering a simple example.

### *4.1.2 Behavioural modelling – a simple example*

#### **4.1.2.1 Introduction**

In order to illustrate the use of state machine diagrams, a simple example will be used that will not only show all basic concepts associated with state machine diagrams, but will also provide a basis for further modelling and provide a consistent example that will be used throughout this section.

The simple example chosen is that of a game of chess as it is a game with which most people are at least vaguely familiar, and thus will have some understanding of what the game is all about. In addition, chess is, on one level, very simple, which means that it can be modelled very simply, yet, on the other hand, it possesses a great deal of hidden complexity that should emerge as the example is taken further.

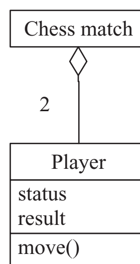


*Figure 4.4 A simple class diagram for a game of chess*

Figure 4.4 shows a class diagram that represents, at a very simple level, a game of chess. From the model, a 'Chess game' is made up of two 'Player'. Each 'Player' has attributes – 'result' and 'status' – and a single operation 'move'. The attribute 'result' reflects the result of the game and may have values: 'this player win', 'other player win', 'draw' or 'undecided'. The attribute 'status' reflects the current status of the game and may take the values 'checkmate', 'stalemate' and 'game in progress'.

#### **4.1.2.2 Simple behaviour**

It was stated previously that any class that exhibits behaviour must have an associated state machine diagram. Applying this rule reveals that, of the two classes present in the class diagram, only the class 'Player' needs to have a state machine diagram. A very simple state machine diagram for a game of chess is shown in Figure 4.5, by defining the behaviour of the class 'Player'.

From the model, the object may be in one of two states: 'waiting' or 'moving'. In order to cross from 'waiting' to 'moving', the event 'player 2 moved' must have occurred. In order to cross from 'moving' to 'waiting', the event 'player 1 moved' must have occurred.

The two pentagons, one convex and one concave represent 'send events' and 'receive events', respectively. A send event is an event that is broadcast outside the boundary of the state machine diagram or, to put it another way, is broadcast to other

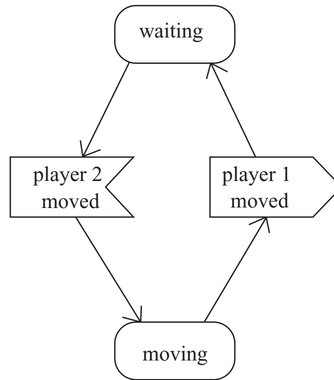


*Figure 4.5 A simple state machine diagram for a game of chess*

objects. A receive event is a complement of the send event in that it is accepted from outside the boundary of the state machine diagram.

At a very simple level, this is 'how' a game of chess is played, by modelling the behaviour of each player. However, the model is by no means complete as the chess game described here has no beginning or end and will thus go on forever. Despite the fact that chess games may seem to go on forever, there is an actual distinct start and end point for the game. This is modelled in the UML by introducing 'start states' and 'end states'.

The next step, therefore, is to add more detail to this state machine diagram. It is interesting to note that this is actually how a state machine diagram (or any other UML diagram, for that matter) is created. The diagram almost always starts off as a simple collection of states and then evolves over time and, as more detail is added, so the model starts to get closer and closer to the reality that it is intended to represent.

#### 4.1.2.3 Adding more detail

The next step, as stated in Section 4.1.2.2, is to add a beginning and an end for the state machine diagram, using start and end states. A start state describes what has happened before the object is created and is shown visually by a filled-in circle. An end state, by comparison, shows the state of the object once the object has been destroyed and is represented visually by a bull's-eye symbol.

Start and end states are treated just like other states in that they require transitions to take the object into another state, which will require appropriate events. The event would typically be creation- and destruction-type events that are responsible for the birth and subsequent demise of the object.

Figure 4.6 shows the expanded state machine diagram that has start and end states along with appropriate events.

It can be seen from the model that there are two possible start states, depending on which player goes first, and three different end states, depending on which player,

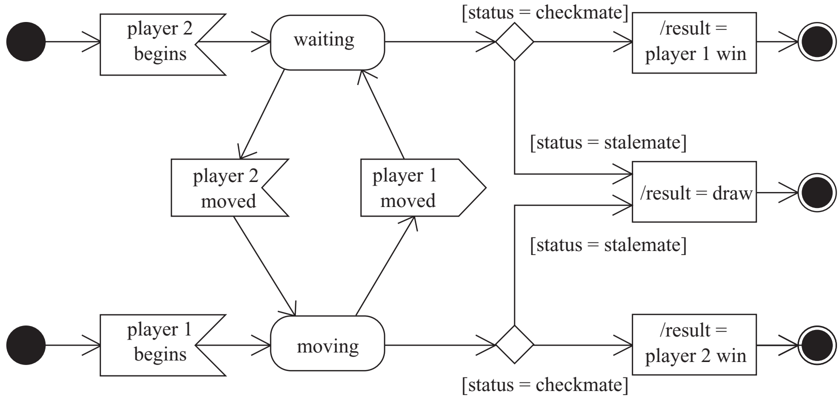


*Figure 4.6 Expanded state machine diagram showing start and end states*

if either, wins, or whether the game is a draw. The model is now becoming more realistic; its connection to reality is getting closer, but there is still room for ambiguity.

Notice also in this diagram that some conditions have been introduced, that are defined on the outputs of the decision symbol (the diamond). These conditions should relate directly back to attributes from its parent class hence providing some basic consistency between a class and its associated state machine diagram.

We know from the class diagram that the class 'Player' does one thing, 'move', but we do not know in which state of the state machine diagram this operation is executed. As it happens, it is fairly obvious which state the operation, occurs in: 'moving'. Operations on a state machine diagram may appear as either 'activities' or 'actions', which will be discussed in more detail in due course. This may now be shown by adding the activity to its appropriate state by writing 'do/' in the state box and then adding the activity (operation from the class diagram) name, which is shown in Figure 4.7.

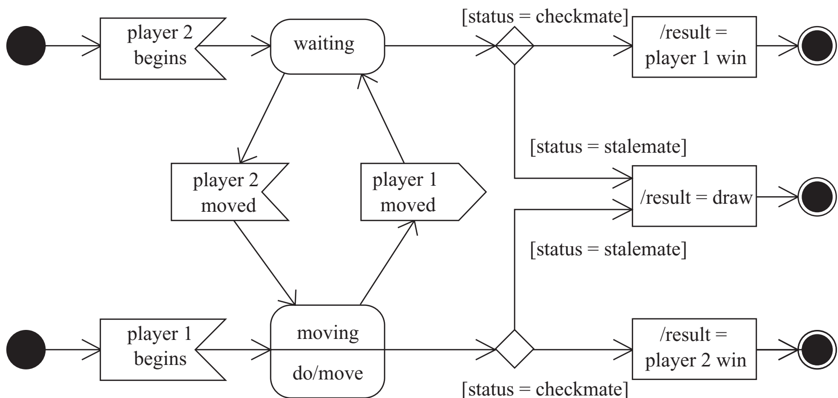


*Figure 4.7 Expanded state machine diagram showing activity*

The model is now getting even closer to reality; in fact, the model is evolving, which it always does! It is almost impossible to get models right the first time as they are living entities and will continue to evolve for as long as they exist. However, there is yet another problem with the state machine diagram, as, although it seems to work well for any situation in a chess game, it is impossible for the game to run! To illustrate this, consider what happens when we begin a game of chess.

#### 4.1.2.4 Ensuring consistency

The first thing that will happen is that two instances of the class 'Player' need to be created so that we have the correct number of players. The behaviour of each player is described by the state machine diagram for the class 'Player'. For arguments sake, we shall name the two players 'Player 1' and 'Player 2' and see if the state machine diagram will hold up to the full game of chess. Figure 4.8 shows two identical state machine diagrams, one for each object, that are positioned side by side to make comparisons easier.

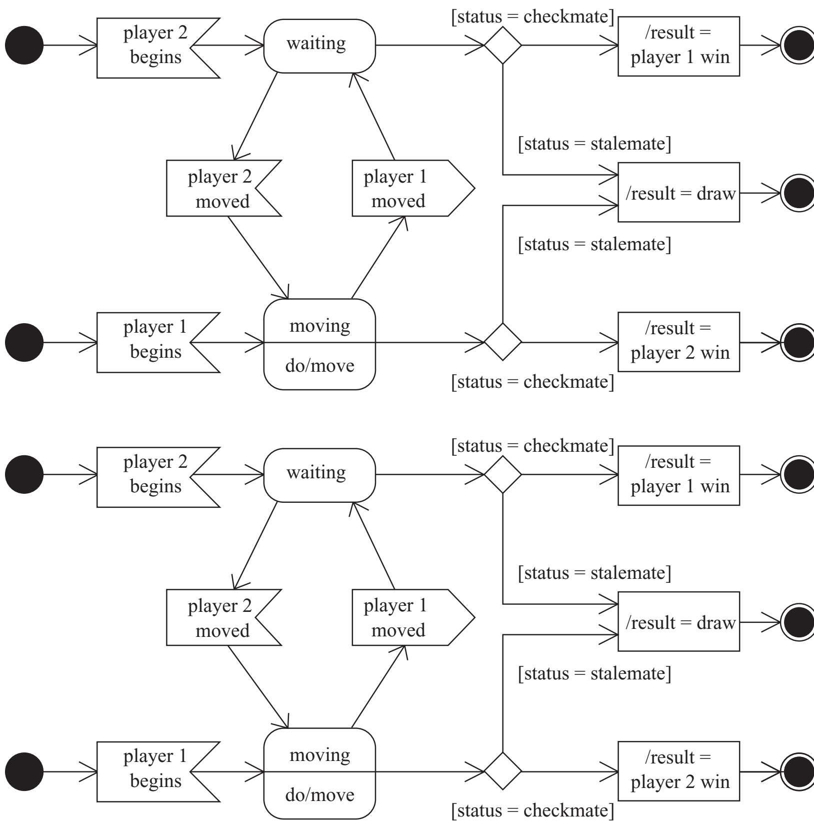


*Figure 4.8 Side by side comparison of two state machine diagrams*

In order to begin a game of chess, an instance of the class 'Chess match' would be created, which would in turn create two instances of the class 'Player'. In this example, the object names 'Player 1' and 'Player 2' have been chosen. Let us now imagine that a game of chess has been started and that 'Player 1' is to begin. The event that will occur is 'player 1 begins', which is present on both state machine diagrams. However, this will put both players straight into the 'moving' state, which will make the game of chess impossible to play, despite being slightly more entertaining. This is because the events were named with reference to one player, rather than being generic so that they are applicable to any player. In order to make the game work, it is necessary to rename the events so that they are player independent.

It is important to run through (by simulation or animation, if a suitable tool is available) a state machine diagram to check for consistency. In this case, the error was a simple, almost trivial, misnaming of an event. However, this trivial mistake will lead to the system actually failing!

This serves to illustrate a very important point that relates back to the section about modelling: the different levels of abstraction of the same model. The chess game was modelled only at the object level in terms of its behaviour, which, it is entirely possible, would have resulted in the class being implemented and even tested successfully if treated in isolation and under test conditions. It would only be at the final system test level that this error would have come to light. It would be useful, therefore, if it were possible to model behaviour at a higher level, where the interactions between objects could be shown, as in Figure 4.9.

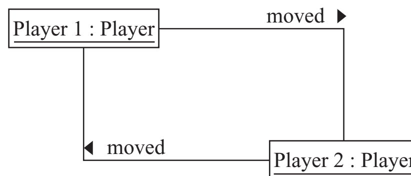


*Figure 4.9 Wouldn't it be nice model*

This figure shows the two objects from the original class diagram, but this time the events from the state machine diagrams have been shown. It is clear from this model that there would be a problem with the event names at a higher level of abstraction, which could lead to the problem being sorted out far earlier than in the previous case. It would have been nice if we had drawn this model as part of our behavioural modelling of the game of chess. Luckily, such a model does exist and is known as a communication diagram, which is covered in detail in Chapter 5.

### *4.1.3 Solving the inconsistency*

There are many ways to solve the inconsistency problems that were highlighted in the previous section, two of which are presented here. The first solution is to make

the generic state machine diagram correct, while the second is to change the class diagram to make the state machine diagrams correct.

#### 4.1.3.1 Changing the state machine diagram

The first solution to the inconsistency problem that is presented here is to make the state machine diagram more generic, so that the events now match up. The state machine diagram for this solution is shown in Figure 4.10.

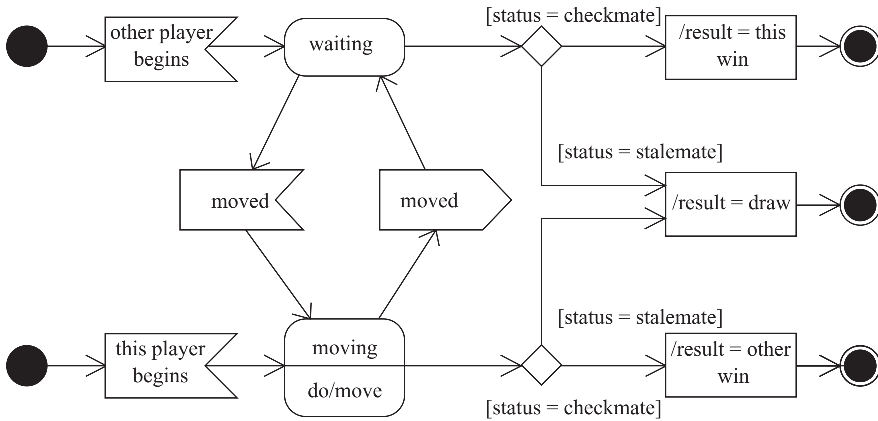


*Figure 4.10 New state machine diagram with correct event names*

This figure represents a correct solution to the chess model. Note that in this model the event names have been changed to make them more generic. The first events are the event on the transitions from the two start states. Note that this time the names have been changed so that they are no longer specific to either player one or player two. The names are now relative to each instance, rather than specific, so the event names have changed from ‘player 1 begins’ to ‘this player begins’ and from ‘player 2 begins’ to ‘other player begins’.

The other event names that have been changed are on the transitions between the two normal states. Previously, the names were object specific, being as they were, ‘player 1 moved’ and ‘player 2 moved’. These names have now changed to a more generic ‘moved’ event, which will apply equally to both objects.

This is by no means the only solution to the problem and another possible solution is presented in Section 4.1.4, where the class diagram is changed to make the state machine diagram correct, rather than changing the state machine diagram.

#### 4.1.3.2 Changing the class diagram

The second solution to the consistency problem is to change the class diagram rather than the state machine diagram, as shown in Figure 4.11.

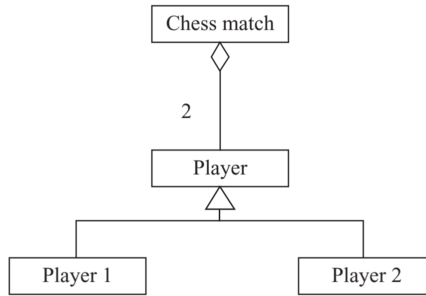


*Figure 4.11 A modified class diagram*

This figure shows a modified class diagram in which two new subclasses of ‘Player’ have been added. This would mean that, rather than the class ‘Player’ being instantiated, one instance of each class ‘Player 1’ and ‘Player 2’ would be created. This has implications on the state machine diagrams as the class diagram shown here would require a state machine diagram for both ‘Player 1’ and ‘Player 2’, rather than a single state machine diagram for ‘Player’. This would also mean that the initial state machine diagram shown in Figure 4.7 would now be correct for ‘Player 1’, but that a new state machine diagram would have to be created for the class ‘Player 2’.

Taking this idea a step further, it is also possible to make the two subclasses more specific as, in the game of chess, one player always controls white pieces and the other player only controls black pieces. This would have an impact on the class diagram again, as each subclass could now be named according to its colour. This is shown in Figure 4.12.

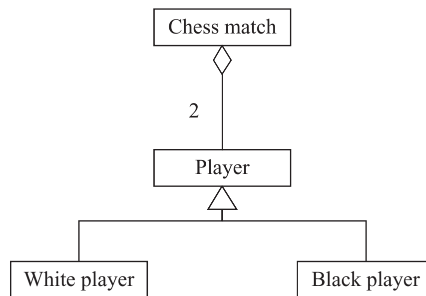


*Figure 4.12 Further modification of the chess class diagram*

This figure shows a class diagram where the classes have been named according to colour, rather than simply ‘Player 1’ and ‘Player 2’. This has even more effect on the state machine diagram as, in the standard rules of chess, the white player always moves first.

## 4.2 Alternative state machine modelling

### *4.2.1 Actions and activities*

The state machine diagrams presented so far have had an emphasis on the activities in the system where the main state in the state machine diagram had an explicit activity associated with it. This is slightly different from another approach that is often used in many texts, which is referred to here as an action-based approach. In order to appreciate the difference between the action-based and activity-based approaches, it is important to distinguish between actions and activities.

- An activity describes an ongoing, nonatomic execution within a system. In simple terms, this means that an activity takes time and can be interrupted. An activity is also directly related to the operations on a class.
- An action is an atomic execution within the system that results in either a change in state or the return of a value. In simple terms, this means that an action takes no time and cannot be interrupted.

Activities may be differentiated from actions as they will always appear inside a state (as what is known as an 'internal transition') and will use the keyword 'do/' as a prefix. Any other keyword used as a prefix (for example, 'entry/', 'exit' etc.) is assumed to be an action.

These two types of execution may change the way that a state machine diagram is created and the type of names that are chosen to define its state machine diagram.

#### 4.2.1.1 Activity-based state machine diagrams

Activity-based state machine diagrams are created with an emphasis on the activities within states. Activities must be directly related back to the class that the state machine diagram is describing and must provide the basis for a very simple, yet effective, consistency check between the class diagram and state machine diagram. As they take time to execute, activities may only be present on a state machine diagram within a state, as states are the only element on the state machine diagram that take time – transitions are always assumed to take zero time.

The state names, when using the activity-based approach, tend to be verbs that describe what is going on during the state and thus tend to end with 'ing', such as the two states seen so far in this chapter: 'moving' and 'waiting'.

#### 4.2.1.2 Action-based state machine diagrams

Action-based state machine diagrams are created with an emphasis on actions rather than activities. Actions are often not related directly back to classes, although they should be, in order to ensure that consistency checks are performed between the two types of diagram. Actions are assumed to take zero time to execute and thus may be present either inside states or on transitions – transitions also take zero time. With an action-based approach, the actions tend to be written on transitions rather than inside states, which leads to a number of empty states. These states have names that reflect the values of the system at that point in time.

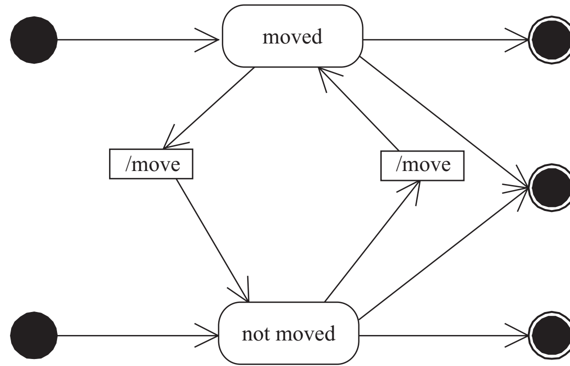


*Figure 4.13 Action-based state machine diagram*

The simplest way to demonstrate the difference is to create an action-based state machine diagram for the chess example and then to compare the two different styles of state machine diagram.

Figure 4.13 shows an action-based state machine diagram where the emphasis is on the action '/move' rather than on activities. Notice that the state names here reflect what has happened in the system, implying that the system is doing nothing while waiting for events.

### *4.2.2 Comparing the approaches*

Each of the two approaches shown here is valid and useful; however, there are a number of observations that can be made concerning the use of both approaches:

- The activity-based approach may lead to a more rigorous approach to modelling as the consistency checks with its associated class are far more obvious.
- The activity-based approach makes use of executions that take time and may be interrupted for whatever reason.
- The action-based approach is more applicable to event-based systems such as windows-based software, where for much of the time the system is idle while waiting for things to happen.
- In terms of real-time systems, the action-based approach is less suitable than the activity-based approach, as the basic assumption of executions taking zero time goes against much real-time thinking.
- Action-based diagrams tend to be less complex than activity-based, as they tend to have fewer states.

So which approach is the better of the two? The answer is whichever is most appropriate for the application at hand.

## 4.3 Other behavioural models

This chapter concentrates mainly on state machine diagrams, but the principles that apply to state machine diagrams apply to the other types of behavioural models. These can be seen in Figure 4.14.

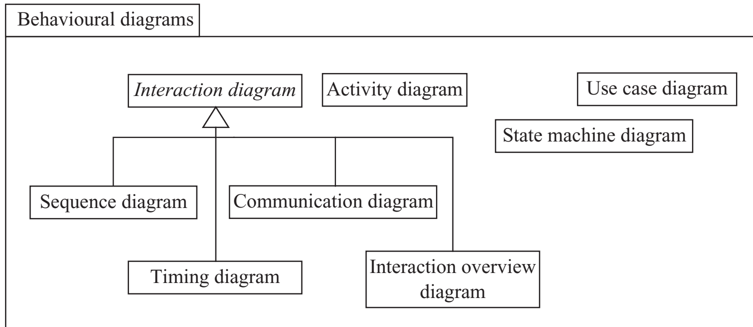


*Figure 4.14 Types of behavioural diagram*

If you can have a good grasp of the models shown here and, just as importantly, understand how they must be checked to ensure consistency between different diagrams (and models), the other diagrams are relatively straightforward.

- State machine diagrams are used to describe the behaviour of a class and, hence, its associated objects. They represent a medium level of abstraction.
- Activity diagrams emphasise the actions within a system and are generally used at a very low level, or algorithmic level, of abstraction. Activity diagrams may also be used at a medium level of abstraction, but with an emphasis on actions within a system, rather than states. The main difference between an activity diagram and a state machine diagram is that a state machine diagram contains normal states (that may be complex) whereas an activity diagram will only contain activation instances. This is elaborated upon in Chapter 5.
- All four types of interaction diagram – communication diagrams, timing diagrams, interaction overview diagrams and sequence diagrams – model interactions at a high level of abstraction and can be used, among other purposes, to ensure that events passed between state machine diagrams are consistent.
- Use case diagrams represent the highest level of abstraction and show the interactions between the system and its external actors. These are strongly related to both types of interaction diagram, which may be used to model scenarios as instances of use cases.

All of the points raised above are examined in Chapter 5, where every diagram is looked at in turn; however, one point that should emerge here is that there is a relationship between each type of behavioural diagram and they share a number of common features.

## 4.4 Conclusions

This chapter introduces behavioural modelling. Behavioural modelling shows the 'how' of a system, whereas structural modelling shows the 'what'. Behavioural modelling was illustrated by looking at a single diagram – the state machine diagram – where a simple example was taken and described in some detail.

It is shown that state machine diagrams describe the behaviour within a class, but that modelling at one level of abstraction only could lead to mistakes, despite the fact that the diagram itself may appear to be correct. The solution to this problem is to model the system at more than one level of abstraction by using other behavioural diagrams.

## 4.5 Further discussion

1. Take the chess example and add a new operation called 'think' to its class diagram, shown in Figure 4.4. The idea behind this is that the 'move' operation will represent the actual physical moving of the chess piece, whereas the 'think' operation represents the planning and strategy involved before the piece is actually moved. Modify the state machine diagram in Figure 4.5 to take into account the new operation.
2. Create two state machine diagrams for the classes 'White player' and 'Black player' shown in Figure 4.12, bearing in mind the rule stating that white always moves first. What effect does this have on the state machine diagrams, particularly with regard to the start states?
3. Apply question 1 to question 2, so that now each of the two diagrams must be modified to take into account the 'think' operation. Is this more convenient than having a single generic state machine diagram for 'Player', or is it more effort?
4. Taking the results of question 1, which, if any, of the two operations may be represented using actions rather than activities? Why is this?
5. Imagine that you are creating a chess computer game and create two new classes of 'Player' called 'Human player' and 'Computer player'. Can a single state machine diagram still be used by both types of player, or do they need individual state machine diagrams?
6. Would it have been beneficial to model the game at the higher level (the communication diagram from Figure 4.9) before modelling at the state machine diagram level? Revisit this point after you have read Chapter 5.

## 4.6 References


1 BOOCH, G., RUMBAUGH, J., and JACOBSON, I.: 'The unified modelling language user guide' (Addison Wesley, Massachusetts, 1998)

2 RUMBAUGH, J., JACOBSON, I., and BOOCH, G.: 'The unified modelling language reference manual' (Addison Wesley, Massachusetts, 1998)

3 STEVENS, P., and POOLEY, R.: 'Using UML' (Addison Wesley, Harrow, 1999)

*Chapter 5*

# The UML diagrams

try not. Do. Or do not. There is no try

*Yoda*

## 5.1 Introduction

### *5.1.1 Overview*

This chapter introduces the 13 types of diagrams that may be used in the Unified Modelling Language (UML). The information in this chapter is kept, very deliberately, at a very high level. There are several reasons for this:

- The main focus of this book is to provide practical guidelines and examples of how to use the UML efficiently and successfully. It is, therefore, a deliberate move to show only a small subset of the UML language in this chapter. In reality, it is possible to model 80 per cent of any problem with less than 20 per cent of the UML language.
- Experience has shown that the most efficient way to learn, and hence master, the UML is to learn just enough to get going and then to simply try out some models of your own. This chapter aims to give just enough information to allow you to get your hands dirty.
- There are a great many UML books published that are devoted entirely to the UML and its very rich syntax, see References 1–14 for examples of books concerning UML version 1.x and References 15–17 for examples of books concerning UML version 2.0. This book should be used in conjunction with such books – for example, with the UML reference manual [1].

This raises the immediate issue of the differences between the two major versions of the UML, version 1.x and version 2.0. The UML itself is a living entity and evolves over time to reflect its customers' needs as well as new technologies and modelling concepts. As such, the notation itself has undergone several minor revisions to improve its usability and applicability to different disciplines. The latest evolution

of the standard, UML version 2.0, represents the most significant change to date and has prompted much discussion, hence the existence of this section of the book.

### 5.1.2 Conceptual changes between UML 1.x and UML 2.0

There are no conceptual changes to the UML whatsoever. The concepts of structural and behavioural modelling are exactly the same. It is still essential to have both aspects of the model but now there are 13, rather than 9, diagrams available to realise these 2 aspects.

The concepts of choosing the right model, levels of abstraction, connection to reality and independent views of the same system remain the same as they are basic modelling concepts, rather than UML-specific concepts.

Therefore, anyone with a full and proper understanding of the concepts of the UML, should find the changes logical and easy to understand – the changes are all syntactical.

### 5.1.3 Terminology

Figure 5.1 summarises the basic terminology used for UML 2.0.

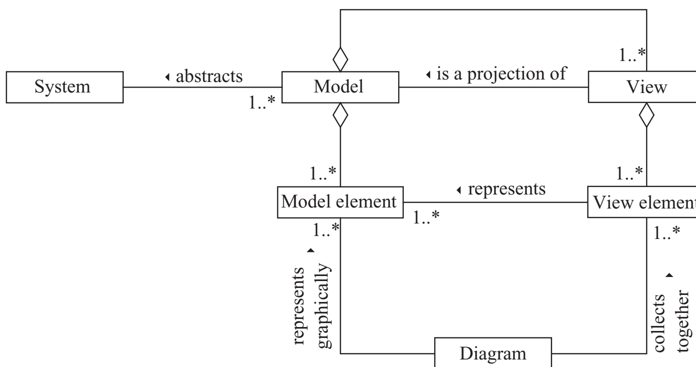


*Figure 5.1 UML 2.0 terminology*

It can be seen from the diagram in Figure 5.1 that one or more 'Model' abstracts a 'System'. Remember that a model is defined as a simplification of reality, therefore, there is a level of abstraction removed between the model and the system. The system itself can represent any sort of system, be it a technical, social, political, financial or any other sort of system.

Each 'Model' is made up of a number of 'View', each of which is a projection of a 'Model'. A view is simply the model itself looked at from a particular context or from a particular stakeholder's point of view. Examples of view include, but are not limited to: logical views, implementation views and contextual views. Therefore, it is possible (and usually very likely) that a model will have a number of views that will differ depending on the stage of the development life cycle or the reader's view point of the system.

Each 'Model' is made up of a number of 'Model element' and, in turn, each 'View' is made up a number of 'View element'. There is a direct comparison between these two types of elements as a 'View element' represents one or more 'Model element'. Model elements, in UML 2.0, represent the abstract concepts of each element, whereas the view elements represent whatever is drawn on the page and can be physically 'seen' by the reader.

The basic 'unit' of any model is the 'Diagram' that represents graphically a number of 'Model element' and that collects together a number of 'View element'. It is the diagram that is created and presented as the real-world manifestation of the model.

### *5.1.4 The structure of UML 2.0 diagrams*

Each diagram in the UML 2.0 has the same structure, which is intended to provide a similar appearance for each, as well as making cross referencing between diagrams simpler. The structure of each diagram is shown in Figure 5.2.

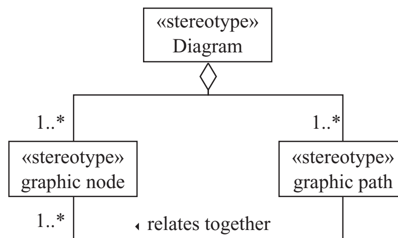


*Figure 5.2 Structure of each UML 2.0 diagram*

The diagram in Figure 5.2 shows that each diagram is made up of one or more 'graphic node' and one or more 'graphic path'. Each 'graphic path' relates together one or more 'graphic node'. Examples of graphic nodes include: classes on class diagrams, objects on object diagrams, etc. Examples of graphic paths include: relationships on class diagrams and links on object diagrams.

Any UML diagram may have an optional graphic node known as a 'Frame' that encapsulates the diagram and that has a unique identifier, in order to make identification of, and navigation between, diagrams simpler. An example of a frame is shown in Figure 5.3.

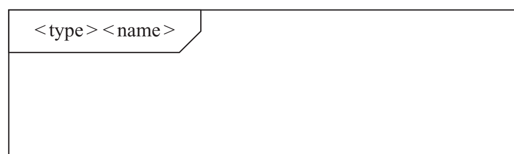


*Figure 5.3 Example of a frame in UML 2.0*

The diagram in Figure 5.3 shows the symbol for a frame, which is, quite simply, a large rectangle that will contain all the view elements in a particular diagram. The pentagon in the upper left-hand corner contains two references – one to the type of diagram (for example: 'cd' for class diagram, 'od' for object diagram, etc.) and then one to the name, or identifier for the diagram.

### *5.1.5 Comparison of the types of diagram*

The most immediately-apparent differences between UML 2.0 and UML 1.x are the number of diagrams – UML 1.x has 9 diagrams, whilst UML 2.0 has 13 diagrams – and the significant changes in terminology for diagram elements. A common initial reaction to this increase in the number of diagrams is one of horror, but, when considered in the cold light of day, the increase in the number of diagrams does not necessarily indicate an increase in the complexity of the language but, rather, an increase in the flexibility and usability of the language. Indeed, all the new mechanisms are intended to make modelling easier compared to UML 1.x where, in some instances, small tricks and rules of thumb were often applied to modelling in order to make diagrams easier to understand.

In order to understand the new diagrams and some of the term changes, UML 2.0 and UML 1.x will be compared at a diagrammatical level, first considering the structural diagram and then the behavioural diagrams. Figure 5.4 shows a comparison of the structural diagrams.

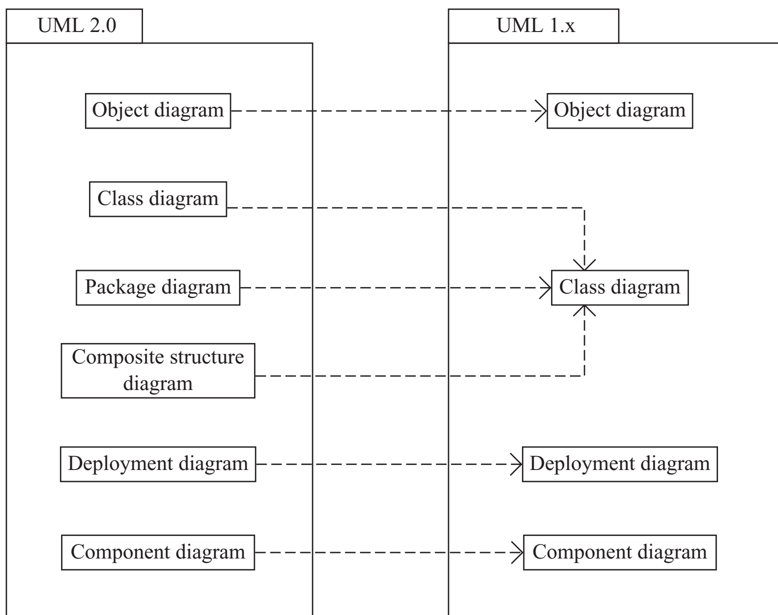


*Figure 5.4 Comparison of structural diagrams – UML 2.0 versus UML 1.x*

The diagram in Figure 5.4 shows how the structural diagrams relate between UML 2.0 and UML 1.x. The main differences are as follows:

- 'Object diagram' to 'Object diagram' – no significant changes.
- 'Package diagram' to 'Class diagram' – although the package diagram is a new type of diagram, it is simply the concept of a package from UML 1.x, but this time it is given its own diagram. These are useful for defining profiles, showing ownership of model elements and relating models together using their 'merges' and 'uses' relationships.
- 'Class diagram' to 'Class diagram' – no significant changes, although more emphasis is now put on identifying interfaces in class diagrams.
- 'Composite structure diagram' to 'Class diagram' – again, a new type of diagram, but this time there are two distinct uses: defining logical structure within a composite or aggregation and identifying collaborations. In UML 1.x it was possible for class diagrams to be created where the emphasis was on identifying the elements in a composite structure (aggregations and compositions) rather than on the logical relationships between these lower-level elements. The composite structure diagram allows an aggregation or composition to be replaced by a 'Part' that contains a number of structural elements, such as classes. This was only achievable in UML 1.x by 'cheating' by using a package to show the contents of a class. This usage is particularly good for modelling architectures of systems. The second usage is to identify static collaborations within a model that represent two diagram elements that will interact with each other in some way. There is then a clear, and obvious, connection to all interaction diagrams that then go on to explore the behaviour of such collaborations. In summary, therefore, it is a completely new diagram which, unlike the new package diagram, has many new symbols and notations.
- 'Deployment diagram' to 'Deployment diagram' – these diagrams are very similar, but now it is objects and artefacts that are deployed in UML 2.0, rather than objects and components in UML 1.x. Components are now abstract specifications whereas an artefact (note the American spelling of this word is 'artifact') represent the actual implementation of a component.
- 'Component diagram' to 'Component diagram' – significant improvements to interfaces that can now be output (required) or input (provided) and can have parts. The internal structure of a component can now be specified using parts and the symbol has been made far simpler to draw by abandoning the cumbersome 'single plus double rectangle' notation with a stereotyped class called «component».

Each of these diagrams is discussed in more detail later in this chapter.

Figure 5.5 shows a high-level comparison of the behavioural diagrams in UML 2.0, compared with UML 1.x.

The diagram in Figure 5.5 shows how the behavioural diagrams relate between UML 2.0 and UML 1.x. The main differences are as follows:

- 'Use case diagram' to 'Use case diagram' – no significant differences – unfortunately the dreaded stick person is still there, although there are recommendations to stereotype the symbol.

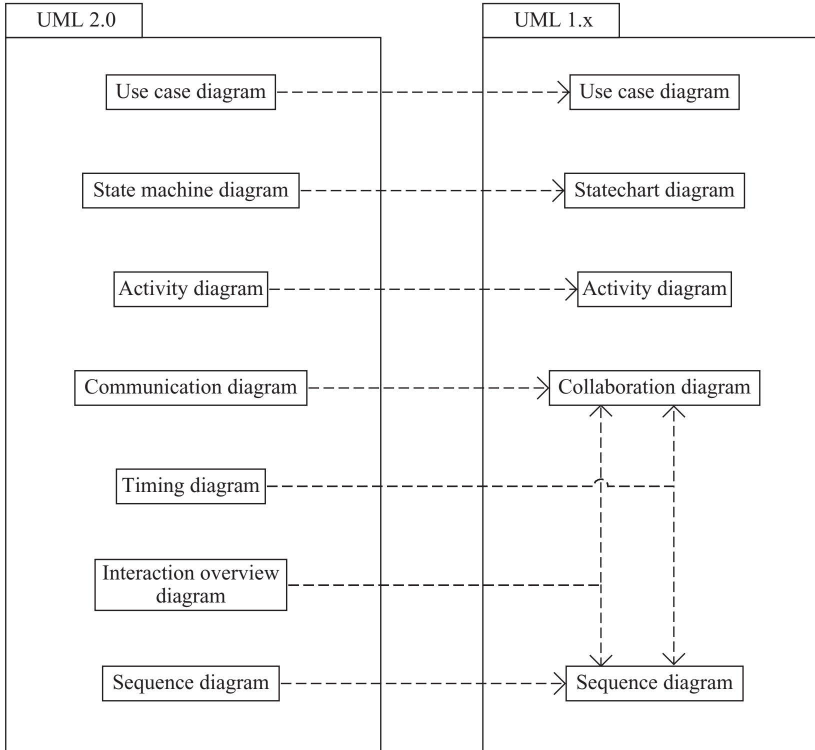


*Figure 5.5 Comparison of behavioural diagrams – UML 2.0 versus UML 1.x*

- 'State machine diagram' to 'Statechart diagram' – apart from the name change, optional graphical notation has been introduced for conditions, events and actions (the same as the optional notation in activity diagrams). Otherwise it is the same in UML 2.0 as in UML 1.x.
- 'Activity diagram' to 'Activity diagram' – enhanced regions (swimlanes and interruptible regions) and the concept of 'token flow' has now been introduced that has been taken from Petri nets. The main behavioural unit is now an 'activity activation' that replaces the 'action state' and 'activity state'. Graphically, they look very similar.
- 'Communication diagram' to 'Collaboration diagram' – the name has changed to make the distinction between a collaboration (structural) and interaction (behavioural) more apparent. Rather than objects, the main element in all interaction diagrams is now the 'Life line' (not the same as the UML 1.x life line, rather confusingly) that represents instances of classifiers (objects are instances of classes). This enables greater flexibility as it is now possible to model interactions between instances of any classifiers, such as components, classes, actors, and so on. Interactions are now bundled up into frames so that they can be reused.

- 'Timing diagram' to 'Collaboration diagram' and 'Sequence diagram' – a new diagram that looks at the timing involved with life lines and their states. Two graphical styles – one is a traditional time line, as seen on graphs (known as a 'state condition time line') whereas the other concentrates on the duration of each state (known as 'general value time line') and looks more like timing shown on communication protocols. The timing diagram has very strong relations with all other interaction diagrams and state machine diagrams.
- 'Interaction overview diagram' to 'Collaboration diagram' and 'Sequence diagram' – a new diagram that represents interactions at a high level. Effectively, individual scenarios can be connected together using activity diagram notation (with interactions replacing the nodes) to form complex behaviour, such as complete system behaviours.
- 'Sequence diagram' to 'Sequence diagram' – like the communication diagram, the new sequence diagrams now use life lines and enable scenarios to be bundled up using frames. This allows scenarios to be referenced from within scenarios. Also, the introduction of conditions allows several options to be explored on a single scenario.

Each of these diagrams is discussed in more detail later in this chapter.

### *5.1.6 The UML meta-model*

This chapter also introduces the concept of the UML meta-model. The UML meta-model is a UML model of the UML, or to put it another way, the UML may be completely modelled using the UML. The UML meta-model itself is concerned with the modelling elements within the UML, how they are constructed and how they relate to one another. The full UML meta-model is highly complex and, to someone without much UML experience, can be quite impenetrable. The meta-models presented in this book show highly-simplified versions of the actual meta-model in order to aid communication and group different aspects of the model according to each diagram – something that is not done in the actual meta-model.

So far, we have looked at two of the diagrams in some detail when class diagrams and state machine diagrams were used to illustrate structural and behavioural modelling; these diagrams are shown again in this chapter for the sake of completeness and also to introduce the meta-model using diagrams that are already well known.

The order in which the diagrams are introduced in this chapter has no real rhyme nor reason, except that they are individually categorised into structural and behavioural diagrams. They are simply presented in, what is from the author's point of view, a logical order. Therefore, the various parts of this chapter may be read in any order.

This chapter also uses a worked example, which is a game of chess. There are several reasons for choosing this as the subject of the example:

- Most people have a good understanding of the game of chess, yet very few people can actually agree on what this understanding is.

- Chess is a game that can be realised in a number of different ways, such as: in software, in hardware, as an off-the-shelf board game, or by using helicopters and a field (apparently). Thus, it is interesting to see how a high-level model may be common to all, but, as time goes on and the model approaches reality, the models start to differ.
- There are many aspects that can be modelled in the game of chess, from the board and pieces, to the rules of the game, and to strategies and skills in playing the game.

Now that the scene has been set, it is time to plunge straight in and look at the UML diagrams themselves.

## 5.2 Class diagrams (structural)

### *5.2.1 Overview*

This section introduces the first of the 13 UML diagrams: the class diagram. The class diagram was introduced in Chapter 3 in order to illustrate structural modelling, so this section should really only serve as a recap. It is useful to examine class diagrams again, however, as they are the diagrams that are used in order to realise the meta-model.

Class diagrams realise a structural aspect of the model of a system and show what conceptual 'things' exist in a system and what relationships exist between them. The 'things' in a system are represented by classes and their relationships are represented, unsurprisingly, by relationships.

Class diagrams are used to visualise conceptual, rather than actual, aspects of a system. Classes themselves represent abstractions of real-life entities. The actual real-life entities are represented by objects and, therefore, any object must have an associated class that defines what it looks like and what it does.

### *5.2.2 Diagram elements*

Class diagrams are made up of two basic elements: classes and relationships. Both classes and relationships may have various types and have more detailed syntax that may be used to add more information about them. However, at the highest level of abstraction, there are just the two very simple elements that must exist in the diagram.

Classes describe the types of 'things' that exist in a system, whereas relationships describe what the relationships are between various classes.

Figure 5.6 shows a high-level meta-model of class diagrams. Ironically, the meta-model for class diagrams is created using a class diagram, which makes it a class diagram of a class diagram. All future meta-models in this book will also be realised using class diagrams.

Remember that the UML is a language and, therefore, should be capable of being read and understood just like any other language. Therefore, the model in Figure 5.6 may be read by reading a class name, then any association that is joined to it, and then another class name.

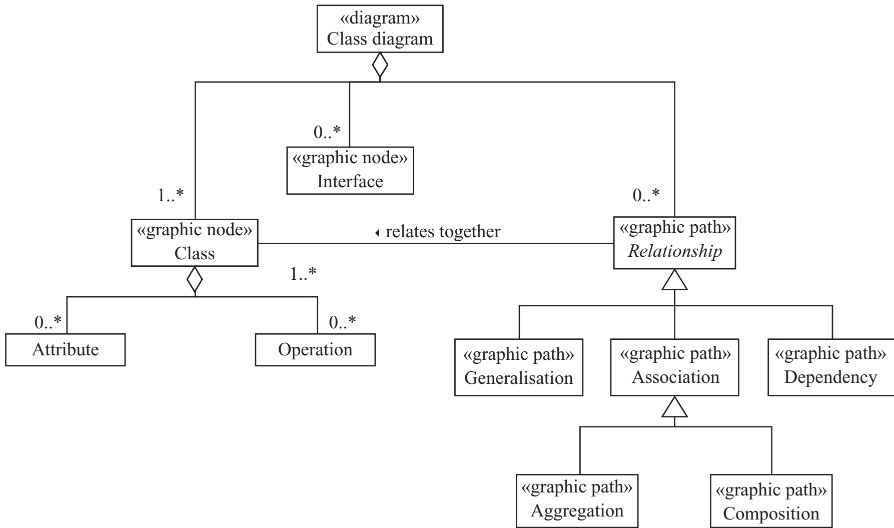


*Figure 5.6 Partial meta-model for class diagrams*

Therefore, reading the diagram: a 'Class diagram' is made up of one or more 'Class', zero or more 'Interface' and zero or more 'Relationship'. It is possible for a class diagram to be made up of just a single class with no relationships at all; however, it is not possible to have a class diagram that is made up of just a relationship with no classes. Therefore, the multiplicity on 'Class' is one or more, whereas the multiplicity on 'Relationship' is zero or more.

Each 'Relationship' relates together one or more 'Class'. Notice that the word 'each' is used here, which means that for every single 'Relationship' there are one or more 'Class'. It is also interesting to note that the multiplicity on the 'Class' side of the association is one or more, as it is possible for a 'Relationship' to relate together one 'Class' – that is to say that a 'Class' may be related to itself.

Each 'Class' is made up of zero or more 'Attribute' and zero or more 'Operation'. This shows how classes may be further described using attributes and operations, or that they may be left with neither.

There are three main types of 'Relationship':

- 'Association' that defines a simple relationship between one or more 'Class'. There are also two specialisations of 'Association': a special type of 'Association' known as 'Aggregation' and one known as 'Composition', which show 'is made up of' and 'is composed of' associations respectively.
- 'Generalisation' that shows a 'has types' relationship used to show parent and child classes.
- 'Dependency' that has only one type (of many) shown, called 'Instantiation', which describes an 'is an example of' relationship.

That, in a nutshell, is the meta-model for class diagrams. Notice how the meta-model, even at this very high level, describes all the concepts that were introduced in Chapter 3. Imagine how this model may be expanded upon to show that, for example, each attribute has a type, default value, etc.

Each of these diagram elements may be realised by either graphical nodes or graphical paths, as indicated by their stereotypes and are illustrated in Figure 5.7.

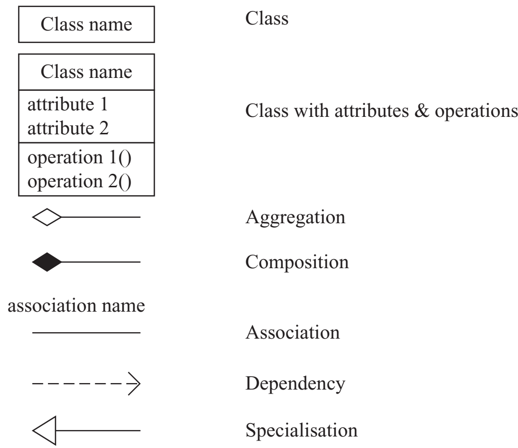


*Figure 5.7 Graphical symbols for elements in a class diagram*

The diagram in Figure 5.7 shows the graphical symbols used to represent elements in a class diagram. The basic symbol is the class that is represented by a rectangle. Rectangles are also used to show other types of element in the UML, so it is important to be able to differentiate between a class rectangle and any other sort of rectangle. A class rectangle will simply contain a single name, with no colons nor underlining. When attributes and operations are present, classes are easy to identify as they have two additional rectangles, drawn underneath the class name, with the names of the attributes and operations contained within. Relationships are shown by a simple line, with slight variations depending on the type of relationship. For example, a specialisation has a distinctive triangle symbol shown next to the parent class whereas a simple association has no such additional symbol.

### *5.2.3 Example diagrams and modelling – class diagrams*

Class diagrams may be used to model just about anything, including:

*Modelling physical systems*: Many examples of modelling physical systems are used throughout this book, ranging from modelling cats and swords to modelling board games!

*Process modelling*: Understanding processes, whether they be in standards procedures or someone's head, is very important in the world of systems engineering. The UML is amazingly powerful for modelling such processes and is the topic of Chapter 6.

*Teaching and training*: The UML is a very good tool for teaching and training. The fact that you are reading this book and (hopefully) understanding the concept associated with class diagrams means that the UML is a good and effective training tool. Remember, more than anything, that the UML is a means to communicate effectively, which is what training and teaching are all about.

*Software*: The UML is also very useful for modelling software, which is covered in enormous detail in just about every other UML book that exists and therefore will not be covered in much detail in this one!

Figure 5.8 shows that a 'Chess game' is made up of two 'Player' and that each player has attributes – 'result' and 'status' – and is able to perform an action called 'move'.

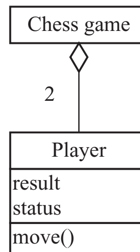


*Figure 5.8 High-level class diagram for the chess game example*

This was the example chosen to illustrate structural and behavioural modelling, which was covered in previous chapters, so let us now consider another aspect of chess that has been completely omitted from the model shown here, but without which the game cannot be played: the board. The next section, therefore, considers the board itself and how this may be modelled. It has been mentioned before that there are many correct solutions to a single problem, which will be illustrated over the next few diagrams.

Figure 5.9 shows a very simple representation of a chessboard. It can be seen that 'Board' is made up of 64 'Square'. Each 'Square' has a single attribute called 'colour', the value of which will dictate which colour the square will be on the board. This is a very simple model and is correct. It is may not be the best representation and does not contain much detail, but it is nonetheless correct.

It is interesting to see how more detail could be added, such as, is it possible to state that there are two, and only two, possible values for the attribute 'colour', being black or white. Also, is the aggregation symbol correct, or would a composition symbol be more accurate? This will depend on the nature of the board and it is necessary to ask the question of whether each square can actually exist by itself or whether it is,

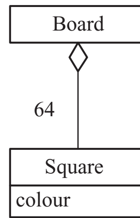


*Figure 5.9 Model of a chessboard*

inherently, part of the board. In a typical shop-bought chess set, the symbol is more appropriate if it is a composition symbol, rather than aggregation. However, what about if each square were actually a field and the pieces helicopters? This may seem slightly ridiculous, but it serves to illustrate the point that almost every model can be represented slightly differently and these are the types of questions that all modellers must begin to ask themselves.

The previous points were slight modifications of the existing model, but it is also possible to have a very different-looking model that still represents the same information, as shown in Figure 5.10.

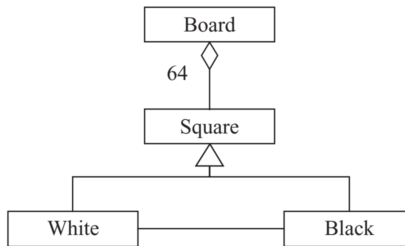


*Figure 5.10 Alternative chessboard model*

Figure 5.10 presents an alternative interpretation of the chessboard. This time, the ‘Board’ is still made up of 64 ‘Square’ (aggregation or composition?) but the class ‘Square’ has no attribute to represent colour. By changing the model, it is possible to show that the concept of colour still exists, but this time by defining two specialisations of ‘Square’: ‘Black’ and ‘White’. More information is added by showing that there is an association between ‘Black’ and ‘White’, but note the multiplicity on the association. As there is no multiplicity defined, one assumes that it is a one to one relationship. This implies that, not only are there 64 ‘Square’, but there is a one to one relationship between the two types of ‘Square’: ‘Black’ and ‘White’. Or to put it another way, there must be 32 ‘Black’ and 32 ‘White’ in order for the association to be true.

The model was changed by adding more classes, but more information was added by doing this. The model has become more complex as there are more classes and associations on it.

The final example of the chessboard is shown in Figure 5.11, illustrating a more radical change of model.

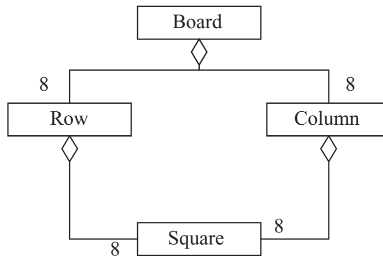


*Figure 5.11 Final example of the chessboard model*

Figure 5.11 shows the chessboard again, but this time the whole pattern has changed, as the basic structure of the aggregation has changed. This time, 'Board' is made up of eight 'Column' and eight 'Row'. Each 'Column' and each 'Row' is made up of eight 'Square'. There is no concept of colour here, but this could be addressed in both approaches shown previously: attributes or subclasses. In addition, there is the same issue of aggregation versus composition here.

All this information may be brought together on a single diagram, as shown in Figure 5.12.

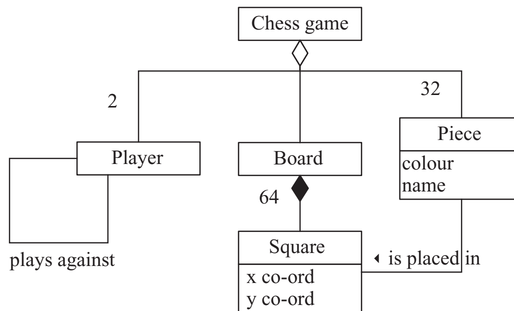


*Figure 5.12 Bringing it all together in a single class diagram*

The diagram in Figure 5.12 shows how some of the previous diagrams may be brought together into a single view. Note here that the composition relationship has been used, rather than the aggregation relationship, to show the breakdown of the chessboard. This is because, in the context of a game of chess, a 'Square' does not exist by itself, but is an inherent part of the whole 'Board'. Also notice how more

detail is being added to the diagram by adding new relationships, so now there is a relationship between 'Piece' and 'Board'.

The question that most people ask at this point is, 'which is the best model?' The question that should be asked, however, is 'which is the most appropriate model for the problem at hand?' Depending on the criteria for defining what 'best' means, the most appropriate model will change. Imagine if the problem called for the simplest (least complex) of models, the most appropriate would be Figure 5.9. If the criteria were more concerned with the layout of the squares on the board, then perhaps Figure 5.10 or Figure 5.11 would be more appropriate.

All the models shown previously in this section have described the same piece of information and this was very simple information. Imagine trying to represent a slightly more complex problem, such as the chess pieces, where there are still two colours but also six different types of pieces, some of which exist singularly, some in pairs and some in sets of eight. This is still a relatively simple problem. Now imagine scaling the problem right up to represent an automated chess machine that can also teach chess, tell the time and make coffee – imagine how many different possible models would exist there!

The main point that has to be made here comes back, once more, to communication. It does not matter how many or few models are generated, because if they cannot be communicated effectively to other people, then how is it possible to make an informed and intelligent decision as to which one is the most appropriate for the problem at hand? The UML offers this method of communication that can iron out many of the ambiguities in the form of a group decision, rather than a single person deciding.

### *5.2.4 Using class diagrams*

Class diagrams are used to model just about anything and form the backbone of any UML model. Class diagrams are perhaps the richest in terms of the amount of syntax available and, because of this, the meta-model for class diagrams is the most incomplete. Other features of class diagrams include, but are not limited to:

- Extra types of classes, such as association classes and interface classes.
- Extra types of relationships, such as other types of dependencies and extra detail that can be added to relationships, such as role names and qualifiers.

The main aim of the class diagram, as with all UML models, is clarity and simplicity. Class diagrams should be able to be read as any sentence and they should make sense. Diagrams that are difficult to read probably represent complex information and, as such, should be simplified or, if this is not possible, then lessons should be learned from them.

## 5.3 Object diagrams

### *5.3.1 Overview*

This section introduces the second of the 13 UML diagrams: the object diagram. Object diagrams realise a structural aspect of the model and are heavily related to

class diagrams. In fact, many people confuse the terms ‘object diagram’ and ‘class diagram’ and use them interchangeably, but they are in fact two separate diagrams. Indeed, many CASE tools that are available do not make any distinction between the two diagrams, often combining them into something known as a ‘structural diagram’ or ‘object model’ or something similar.

The definition of an object is that it is an instance of a class; therefore, it seems logical that object diagrams will represent instances of class diagrams, which is exactly what they do. Objects represent real-life examples of the abstract classes found in class diagrams and relate very closely to real life. Object diagrams also have strong relationships with component and deployment diagrams (both structural) as well as all types of interaction diagrams (communication, sequence, interaction overview and timing diagrams).

### *5.3.2 Diagram elements*

Object diagrams are made up of two main elements: objects and links. Objects are defined as an instance, or many instances, of a class. In the same way, links are instances of associations.

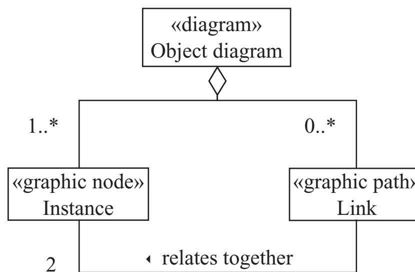


*Figure 5.13 Partial meta-model for object diagrams*

It can be seen from the diagram in Figure 5.13 that an ‘Object diagram’ is made up of one or more ‘Instance’ and zero or more ‘Link’. Also, each ‘Link’ relates together two ‘Instance’. When instances are identified, it is usual to assign a value to each attribute as each instance represents a real-life example of a class and, hence, must have values completed.

Each of these diagram elements may be realised by either graphical nodes or graphical paths, as indicated by their stereotypes and are illustrated in Figure 5.14.

The basic symbol for an instance is a rectangle that has the instance identifier (often referred to, incorrectly, as the ‘instance’ name) with the class that it is an instance of next to it, separated by a colon. Most importantly, all text is underlined in the top rectangle. This is crucial as, by omitting the underlining, it means that the symbol represents a part or even a class.

Object diagrams are very heavily dependent on class diagrams, as each object requires an associated class from which its appearance and behaviour can be taken.

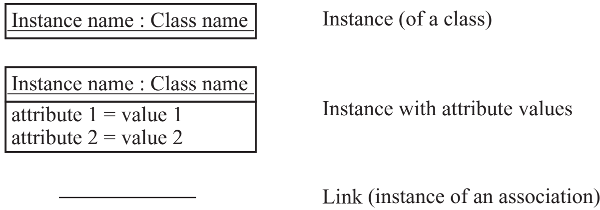


*Figure 5.14 Graphical symbols for elements in an object diagram*

This is illustrated in Figure 5.15, which shows a meta-model of how classes and objects are related.

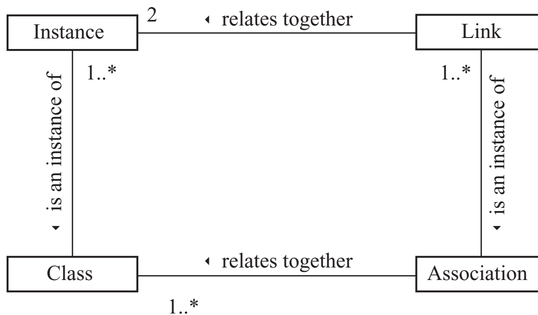


*Figure 5.15 Relationships between object diagram elements and the rest of the UML*

Reading the diagram, it can be seen that a ‘Link’ relates together two ‘Instance’. One or more ‘Instance’ is an instance of a ‘Class’ and one or more ‘Link’ is an instance of an ‘Association’.

Notice how the top and bottom half of Figure 5.15 have both been seen before in Figure 5.6 and Figure 5.13. This is an example of seeing patterns emerge from different models that can give a whole new view of a particular piece of information when shown together and with the addition of a few extra relationships [7].

### *5.3.3 Example diagrams and modelling – object diagrams*

Object diagrams are perhaps the least used of all the 13 UML diagrams and, indeed, many of the tools on the market do not support the creation of object diagrams. However, like all aspects of life, it is difficult to see how some things can be used effectively until a particularly awkward situation arises that cannot be modelled using any other diagram. The main aim of an object diagram is to enforce the connection to reality between the abstract class diagram and the real-life system that it is representing. Quite often, classes can be a little too abstract for people to understand, particularly when representing complex data structures or hierarchies. In a real-life

situation, if a concept is too abstract, it is natural for people to ask 'can you give me an example' or 'can you give me a for instance'. Thinking about these words, they are exactly the terminology used in the UML. Therefore, when someone asks 'can you give me an example', think immediately of instances, whether it is a structural aspect of the model using object diagrams, or a behavioural aspect of the model using interaction diagrams.

This section, therefore, aims to point out some real-life examples of how object diagrams can be very useful, if only to illustrate how a very abstract class diagram relates to reality. In other words, object diagrams are very good at making that all-important connection to reality that all good models should have. This will be illustrated by looking at two real-life examples of how useful an object diagram can be, before relating object diagrams back to the chess example.

Consider the example of a taught course and how different types of persons may exist who are involved with the course. Each of these people may have attributes that make them unique and that may be useful to administration staff who run the course. This may be modelled by the class diagram shown in Figure 5.16.

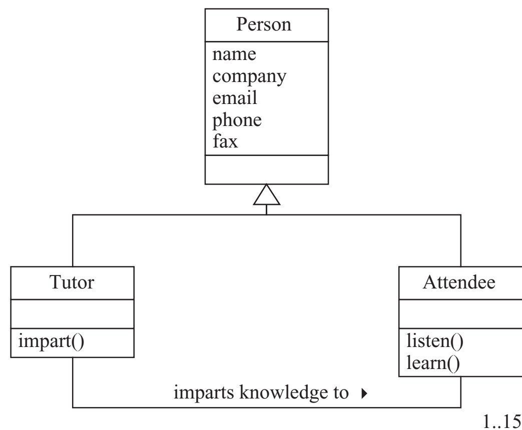


*Figure 5.16 Class diagram of a taught course*

It can be seen from Figure 5.16 that there is a class called 'Person' that has five attributes to describe that person: 'name', 'company', 'phone', 'email' and 'fax'. There are two types of 'Person' – 'Tutor' and 'Attendee' – each of whom inherits the attributes of their parent class. 'Tutor' is specialised by having the operation 'impart', while 'Attendee' has operations 'listen' and 'learn'.

It is now possible to show an example of some people who may have their records stored on a very simple database. Although the class diagram shown here is a far cry from a database design, it clearly illustrates the point that sometimes it is useful to see an example of how such structural information may be stored.

Figure 5.17 shows examples of three of the records that may exist: one for 'Tutor' and two for 'Attendee'. The visual clue that these three elements represent objects,

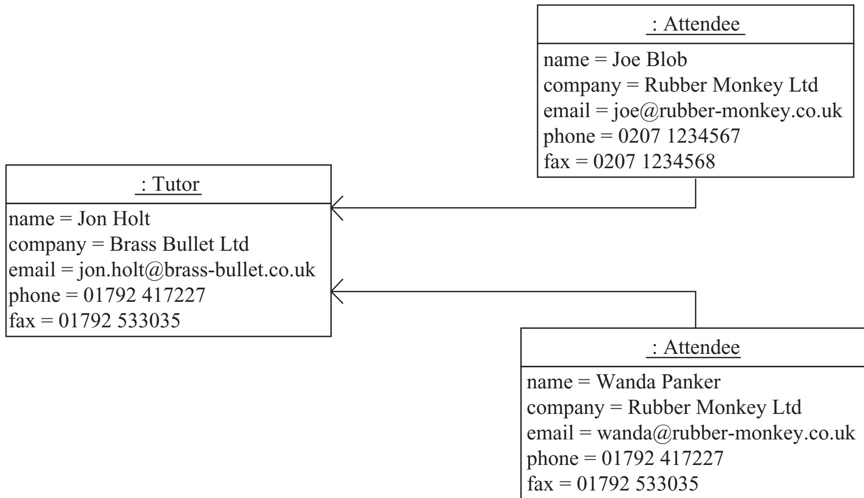


*Figure 5.17 Example of an object diagram*

rather than classes or parts, is the fact that the names are underlined – as soon as something is underlined, it is an object. Notice how the objects here are anonymous instances as they do not have unique object identifiers. It is assumed that each will actually have a unique identifier, but for whatever reason they are not shown here. The three objects are clearly instances of their associated classes, but the original association from Figure 5.16 has now been instantiated as a link that shows the relationship between the three objects. Notice how the multiplicity of the link is different from that of the association as the association is abstract, showing a 1 to 1 . . . 15 multiplicity. The link, however, shows a 1 to 1 multiplicity, which is still correct as the original multiplicity means between 1 and 15. If the diagram were completed, there would be between 1 and 15 instances of 'Attendee' each with a 1 to 1 relationship with the instance of 'Tutor'.

Object diagrams, it has been stated, represent real-life examples of their associated classes and thus have their attribute values filled in. These may never change once set, such as the malignly-monickered 'Joe Blob', or they may change over time, as in the chess example later in this section.

Another example of the use of object diagrams is when the class diagram is a little too abstract to get a clear handle on what is happening. This is often the case when massive amounts of data and information are abstracted into a very simple hierarchy, as often happens in databases. The following example is based on a system that must describe how different components stored in a database are configured to make a whole assembly.

Figure 5.18 shows a combined class and object diagram, which should usually be avoided. However, in this case, it shows the link relationship between a very abstract class diagram and a real-life object diagram quite nicely.

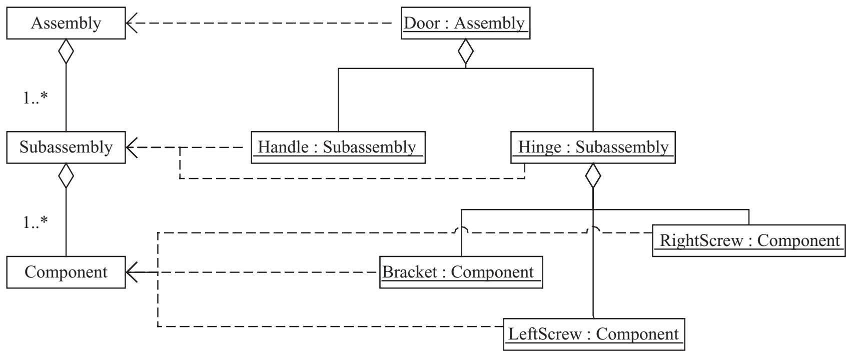


*Figure 5.18 Combined class and object diagram for a parts database*

The classes on the left of the model show a simple hierarchy that represents the way that parts in a database are classified before they are configured to make up a complete system. The highest level in the hierarchy is 'Assembly', which represents a completed unit that may be sent out of a warehouse to be used in the manufacture of vehicles, for example. It can be seen that each 'Assembly' is made up of one or more 'Subassembly', each of which is made up of one or more 'Component'. This class hierarchy is very easy to understand in terms of the class diagram syntax itself, but is very difficult to relate to a real-life situation. It does not matter how accurate the class hierarchy is if the person paying the bill cannot make head or tail of it; therefore, the obvious solution would be to show an example of the real-life situation in which it would appear. This is where the object diagram comes in.

Consider the right-hand side of the diagram now, which makes up the object diagram. It can be seen that an instance of 'Assembly' called 'Door' is made up of two instances of 'Subassembly': 'Handle' and 'Hinge'. 'Hinge' is further decomposed into three instances of 'Component': 'LeftScrew', 'RightScrew' and 'Bracket'.

Although very simple, it was necessary to show an example hierarchy to the customer otherwise there is no way that they could agree that the class diagram was correct as it was too abstract for the customer to understand. To explain this another way, the connection to reality was too far removed and thus the customer had problems understanding it. Obviously it is simply not feasible for the object diagram to show every instance of the class as it would be far too large to contemplate.

The model in Figure 5.19 continues the chess example that was introduced previously, but this time the object diagram is used.

*Figure 5.19 Object diagram for the chess game*

Figure 5.19 shows one possible object model for the chess game. Notice how each object holds true to the original classes that were introduced in Figure 5.8. This time, however, as we are considering actual objects, the attribute values are filled in. This may change throughout the lifetime of the object: as the object changes state so might the value of the attributes.

In addition, note that a link has been included that is an instance of the association from the class diagram. Also, note that the multiplicity here is one to one, which is the same as in the original class diagram.

There is a strong link between object diagrams and state machine diagrams, as state machine diagrams describe the behaviour over the lifetime of a class, which is the same as saying an object. Therefore, any state in an object's state machine should be representable by an object diagram that reflects the object's attribute values while it is in that state.

### *5.3.4 Using object diagrams*

There are a few hints to bear in mind when using object diagrams:

- Each object must be an instance of a class and there may be more than one object (instance) for every class. If an object needs to be represented on a diagram and a class does not already exist, it is necessary to revisit the class diagrams in the model and create an appropriate class.
- Objects exist in the real world as specific examples (instances) of things (classes). As a consequence of this, each object is unique and has a unique identifier. In many cases, this identifier need not be shown on the model but it should be remembered that it does exist.
- Each link is an instance of an association and one association may have more than one link as its instances. Again, this can provide a good consistency check with the associated class diagram.
- Attributes in an object have their values filled in, to reflect their 'real' nature. An object with its attributes' values filled in should correspond to at least one state in that object's statechart.
- Objects can also be seen on interaction diagrams (all four types) and deployment diagrams. An interaction diagram is basically a behavioural view of the object diagram, which will show messages on links and ordering information.

Object diagrams are one of the least used of all the UML diagrams and they are not supported by many CASE tools. Despite this, however, they do still have their uses, as described in this chapter.

## 5.4 Composite structure diagrams (structural)

### *5.4.1 Overview*

This section introduces a diagram that is completely new to UML version 2.0 and is known as the 'composite structure diagram'. The composite structure diagram

realises a structural aspect of the model and has two distinct purposes:

- Compositions and aggregations may be represented within the boundaries of a 'Part' without the need to show the aggregation or composition relationships explicitly. This means that an emphasis may be put on the logical relationships between elements of the composition, rather than the structural breakdown itself. This adds a great deal of value, as it forces the modeller to think about the logical relationship between elements, rather than simply which classes are part of which other classes.
- Collaborations may be identified in the composite structure diagram. Collaborations in UML version 2.0 are static structures that simply identify that there is some sort of communication between one or more elements within the model. This idea is slightly different from the concept of a collaboration in UML version 1.x. In the previous incarnations of the UML, there was a diagram known as the 'collaboration diagram' (now known as the 'communication diagram') that realised a behavioural aspect of the model, so this slight change in the usage of the term collaboration can lead to a degree of confusion. In summary, therefore, collaborations are structural elements that simply identify elements that will interact in some way – they identify 'what'. The definition of 'how' these collaborations behave is stated using 'interaction diagrams' so there is a clear link between the two concepts of collaboration (structural, 'what') and interaction (behavioural, 'how').

The composite structure diagram uses a number of new elements in UML version 2.0, which are introduced and discussed in the following sections.

### *5.4.2 Diagram elements*

The basic element within a composite structure diagram is the 'Part'. A part is a new UML element that describes a collection of UML elements, such as classes or instances of classes. The concept of the part is utilised in both usages of the composite structure diagram, as is described later in this section. A composite structure diagram identifies parts and their internal structures and also parts and their associated collaborations.

The diagram in Figure 5.20 shows the partial meta-model for the composite structure diagram. It can be seen that a 'Composite structure diagram' is made up of one or more 'Part', zero or more 'Port', one or more 'Collaboration', zero or more 'Collaboration occurrence', one or more 'Connector' and zero or more 'Role binding'. It can also be seen that a 'Collaboration occurrence' defines the occurrence of a 'Collaboration'.

Each of these diagram elements may be realised by either graphical nodes or graphical paths, as indicated by their stereotypes and illustrated in Figure 5.21.

The diagram in Figure 5.21 shows the symbols for each of the elements in a composite structure diagram. Several new symbols have been introduced here, compared to UML version 1.x. The symbol for a part looks similar to that of an instance, the difference being that the part has no underlining present, that is the graphical clue that

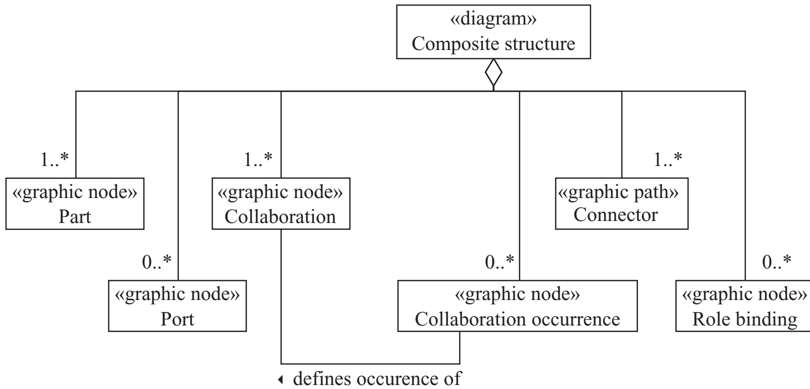


*Figure 5.20 Meta-model of the composite structure diagram*

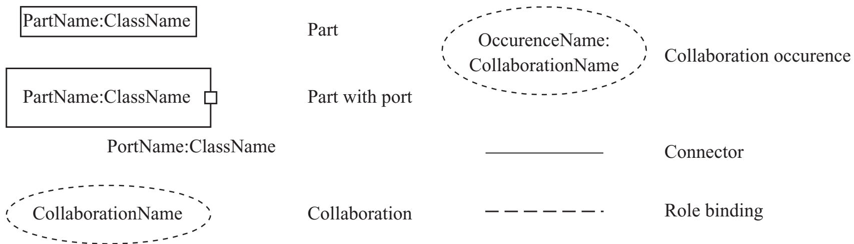


*Figure 5.21 Graphical symbols for the composite structure diagram elements*

identifies an instance. Note also that ports may be added onto a part by using a small rectangle and then naming it. The symbol for the collaboration and the collaboration occurrence is a dotted ellipse that contains the name.

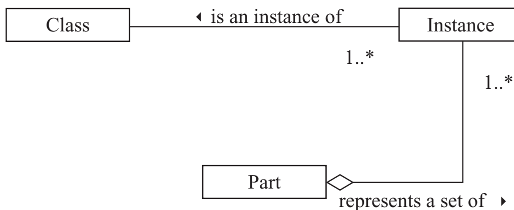


*Figure 5.22 Relationships between composite structure diagrams and other parts of the UML*

The diagram in Figure 5.22 shows the relationship between elements in a composite structure diagram and the rest of the UML. The main relationship here is that between the 'Part' and a set of 'Instance' which, in turn is related back to a 'Class'.

### *5.4.3 Examples and modelling – composite structure diagrams*

In order to illustrate the power of the composite structure diagram, it is first necessary to revisit an example class diagram. Consider the chess example that has been used so far in this section and the class diagram representation of it. An expanded view of the game of chess is shown in Figure 5.23, that brings together some of the concepts already introduced in this chapter.

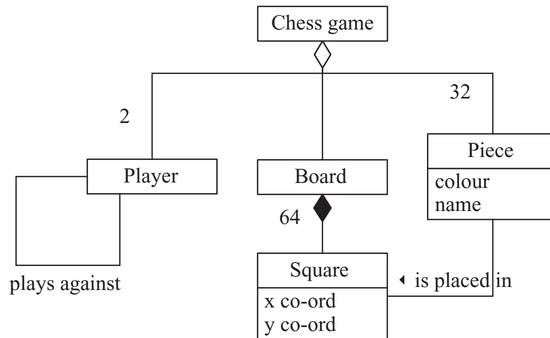


*Figure 5.23 Class diagram representation of the game of chess*

The diagram in Figure 5.23 shows the class diagram representation of the game of chess. This is identical to the class diagram described previously. Notice how there are three aggregations shown here below ‘Chess game’ and a single composition below ‘Board’. However, the nature of the interpretation of these relationships differs somewhat between ‘Chess game’, ‘Piece’ and ‘Board’ and between ‘Chess game’ and two ‘Player’. It is these two slightly different types of relationship that the composite structure diagram can help to clarify.

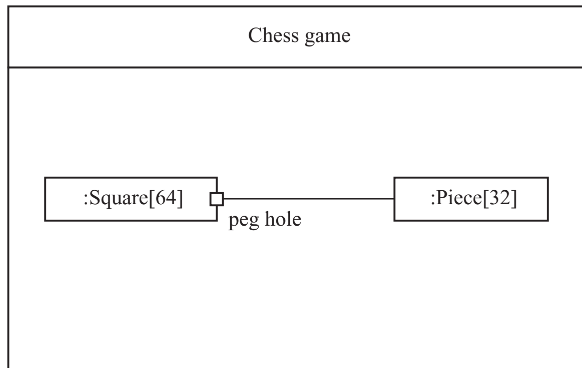


*Figure 5.24 The composite structure of a chess game at an abstract level*

The diagram in Figure 5.24 shows the composite structure of the 'Chess game' where, as can be seen, the main aggregations have been removed and the focus is now on the logical relationship between the two major elements in the diagram: 'Square' and 'Piece'. The class 'Chess game' is now represented by the large rectangle and the composition of this class is shown inside the larger, lower rectangle. One side effect of the aggregations no longer being present is that the multiplicity is no longer clear, but this is overcome by representing the set classes as a part, and then indicating the number of instances in square brackets after the part name. It is easy, at this stage to confuse a part with an instance. The visual clue that should be looked for is the underlining as only an instance has an underlining whereas the part does not. The part represents a set of instances, hence there is a similar notation to the instance by using the colon, but now the number of instances is indicated in square brackets after the part name.

This diagram is still quite abstract, but it is also possible to show the actual instance within the class 'Chess game' by instantiating all elements within the diagram, as shown in Figure 5.25.

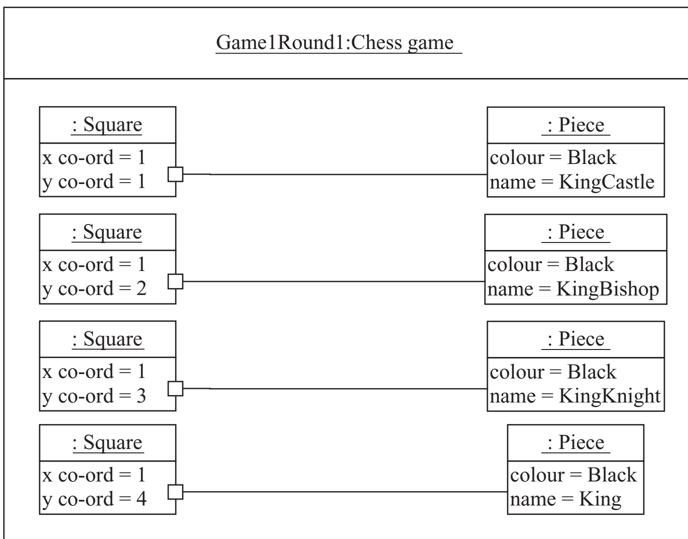


*Figure 5.25 The composite structure of a chess game at a specific, or instance level*

The diagram in Figure 5.25 shows the same information as the diagram in Figure 5.24 but, this time, the diagram is showing the actual instances (as the names are underlined), rather than a collection of instances as a part. Note how the attribute values are completed and there is a distinct usage of instances both at the main classes and within the structure of that instance.

The second usage of the composite structure diagram is to identify collaborations within a model. A collaboration, in UML 2.0, is simply a statement that a number of

elements in a diagram will interact in some way. The collaboration itself is structural but is strongly related to interactions, which describe the behaviour of a collaboration.

Therefore, considering the game of chess again, the main interaction is between the two players, so the collaboration will identify that there is some sort of communication between two instances of 'Player'. This is shown in Figure 5.26, which shows two slightly different notations for identifying collaborations. The upper representation shows a dotted ellipse that represents the collaboration, with its name inside a segment of the ellipse. The contents of the collaboration can be seen inside the ellipse, therefore this diagram reads: 'there is a collaboration that contains two parts: 'White:Player' and 'Black:Player' that move against one another'.

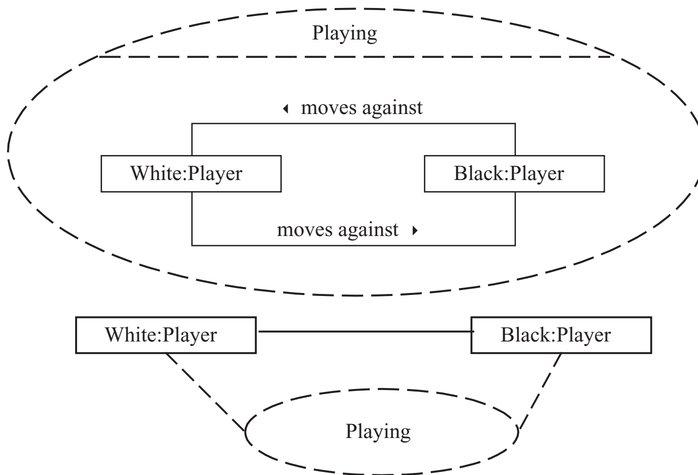


*Figure 5.26 Identifying collaborations (two notations) at an abstract level*

Exactly the same information is shown in the lower half of the diagram except, this time, the contents of the collaboration are shown outside of the dotted ellipse, with role bindings shown between the ellipse and its contents. A role binding, represented graphically by an undirected dotted line allows parts to be associated with collaborations without them being physically inside the collaboration symbol.

Either notation may be used, depending on the preference of the modeller. For example, simple collaborations may be easier to draw using the lower notation, whereas a more complex collaboration may be easier to show actually inside the ellipse. Also, depending on the tool being used to draw the diagram, it may be easier to show a uniformly-sized ellipse on all diagrams, rather than resizing it.

The two notations shown so far are concerned with abstract parts interacting in a collaboration, but it is also possible to show specific instances interacting in the collaboration. This is quite simple to achieve, as it merely involves instantiating each element within the diagram to show a specific example, in this case, a game of chess.

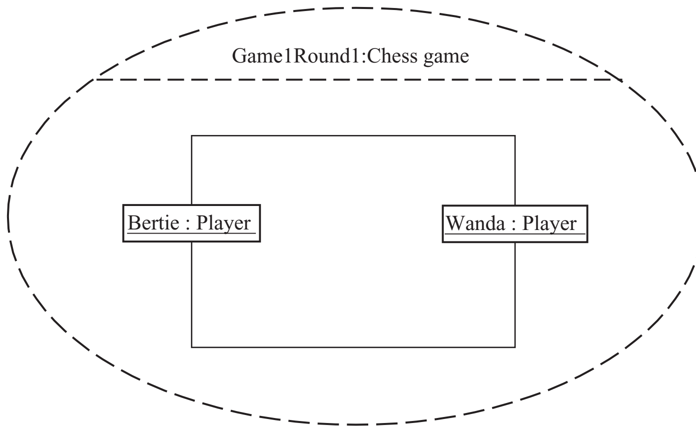


*Figure 5.27 Identifying a collaboration of the chess game at a specific, or instance level*

The diagram in Figure 5.27 shows an instance of the collaboration identified in Figure 5.26. It is clear that the diagram is dealing with instances as, this time, the names are underlined.

### *5.4.4 Using composite structure diagrams*

The composite structure diagram is a diagram that is completely new to the UML, but very strongly related to the class and object diagrams. The main new element introduced here is the 'Part' that represents a collection of instances of a class and is often thought of as a 'halfway house' between abstract classes and specific instances of classes.

There are two main uses for a composite structure diagram:

- To show the structure of a complex class by, effectively, representing the whole class as its own diagram. This allows the emphasis of the diagram to be placed more on the logical relationships between elements of the class, rather than identifying that they are actually elements of a particular chess game (such as aggregation and composition). This is particularly useful for showing elements within an architecture of a system.
- To identify collaborations between different elements of a system. It may be argued that the activity of identifying the structural collaborations is superfluous and that they are identified inherently within scenarios, which is true. However, by looking at the collaborations from the structural view, it is possible to ensure that appropriate coverage of the model is achieved before considering that coverage using scenarios. Again, remember that the use of any UML diagram is optional so it is very much up to the modeller's discretion where to use each type of diagram.

Composite structure diagrams have very strong relationships to class and object diagrams, which is only to be expected with a diagram that emphasises structure. Also, it has very strong relationships with all four types of interaction diagram, as the scenarios that are realised by interaction diagrams will be based on the collaborations identified in the composite structure diagrams.

## 5.5 Package diagrams (structural)

Package diagrams are new to UML 2.0, however, there is nothing new at all about the contents of a package diagram. Package diagrams, as the name implies, simply identify and relate together packages. The concept of the package is exactly the same as when used on other diagrams – each package shows a collection of diagram elements and implies some sort of ownership. The syntax for the package diagram is very simple and can be seen in Figure 5.28.

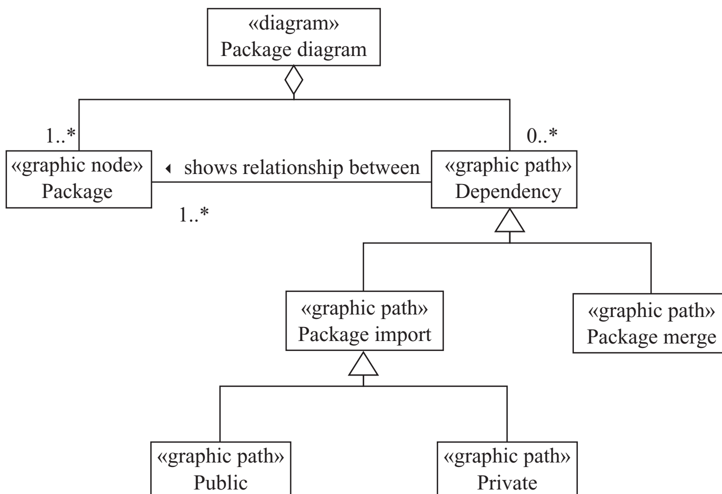


*Figure 5.28 Partial meta-model for package diagrams*

The diagram in Figure 5.28 shows the partial meta-model for the package diagram. It can be seen that there are two main elements in the diagram – the ‘Package’ that happens to be a graphical node and the ‘Dependency’ that happens to be a graphical path. There are two main types of ‘Dependency’ defined – ‘Package import’ and ‘Package merge’. The ‘Package import’ also has two types that are ‘Public’ and ‘Private’.

The graphical notation for the package diagram is shown here and, as is quite clear, this is identical to the graphical notation in UML 1.x.

The diagram in Figure 5.29 shows that there are really only two symbols on the diagram. The graphical node is the same as the original package symbol – the rectangle with a smaller tag rectangle on the top left-hand edge. This is similar to

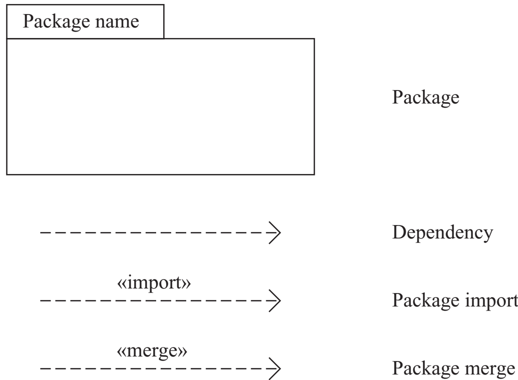


*Figure 5.29 Graphical symbols for elements in a package diagram*

the folder icon that can be seen in Windows systems and, indeed, has a very similar conceptual meaning. The name of the package can either be shown in the tag (as seen here) or, in the case of long names, will often be shown inside the main rectangle. The main graphical path in the package diagram is a dependency that has two basic stereotypes defined – «import» and «merge». The «merge» stereotype means that the package being pointed to (target) becomes part of the other package (source), with any elements with the same name being taken as specialisations of the source package elements. An «import», on the other hand, implies that the target package still remains its own package as part of the source package. Any element name clashes are resolved with the source package taking precedence over the target package. Other stereotypes will often be defined for the dependencies on a package diagram.

As the basic concept of the package is unchanged from UML 1.x, then so is the relationship between a package and the rest of the UML language.

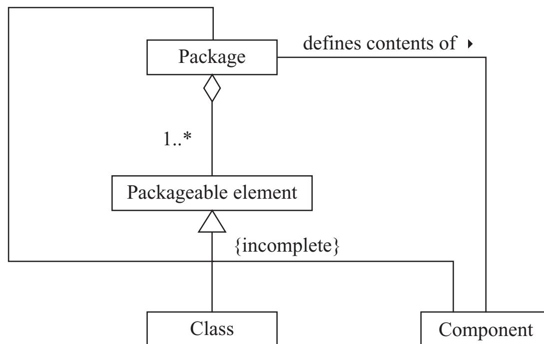


*Figure 5.30 Relationships between package diagram elements and the rest of the UML*

The diagram in Figure 5.30 shows that a 'Package' is made up of a number of 'Packageable element'. In the UML, almost any element can be enclosed within a package, so only a few examples are shown here. (Note that this is indicated by the {incomplete} constraint.) One of the main uses for a package is to define the contents of a component, which can be seen on this diagram by the pattern that shows each 'Package' defines the contents of zero or one 'Component'. The multiplicity here indicates that each component must have an associated package, but that the reverse is not true – it is possible to have a package without an associated component.

### *5.5.1 Examples of package diagrams*

Package diagrams are typically used to show relationships within a model at a very high level. For example, it is possible to define a particular pattern in a model and then to reuse this either in the same model or in other models.

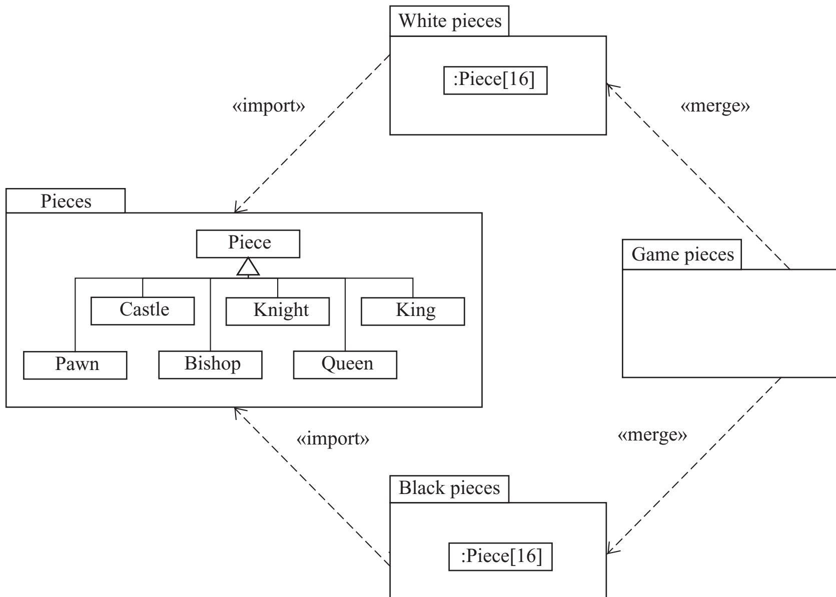


*Figure 5.31 Packages and their relationships for a chess game*

The diagram in Figure 5.31 shows a package named 'Pieces' that contains a pattern defining types of piece for the chess game model. This same pattern is used for both white and black pieces, which is shown by the «imports» relationship here. The packages 'White pieces' and 'Black pieces' both import the contents of the 'Pieces' package. Also, both the 'White pieces' and 'Black pieces' may be combined, shown by the «merges» relationship, to create the entire set of 'Game pieces'.

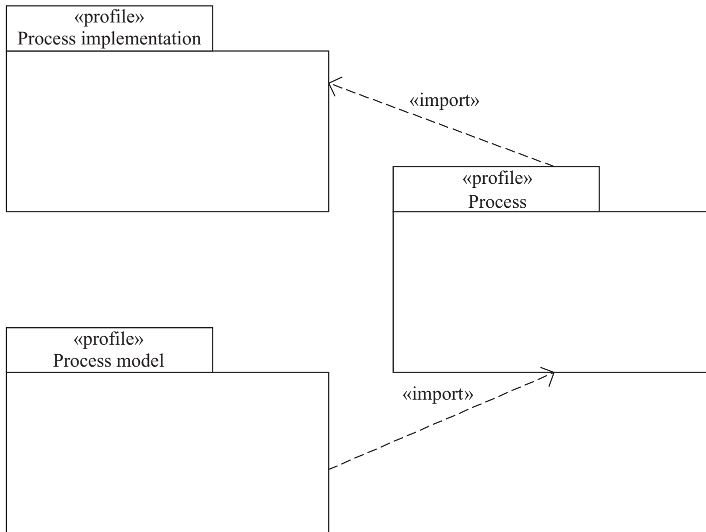


*Figure 5.32 Example of a package diagram used to represent various profiles*

The diagram in Figure 5.32 shows another package diagram being used to show which profiles are being used for a particular project (profiles are discussed in more detail in Chapter 10). It can be seen that there is a package named ‘Process model’ that «imports» a package called ‘Process’ that happens to be a profile. In turn, this package imports another package named ‘Process implementation’ that happens to be a profile.

### *5.5.2 Using package diagrams*

Package diagrams are new to UML 2.0 but there is nothing new, conceptually or syntactically, about the diagram elements. The only new aspect is that packages now have their own diagram, whereas in UML 1.x, they existed on other diagrams.

The main use for package diagrams is to show high-level relationships between groups of things in a model – particularly powerful for reuse of patterns and declaring profiles.

## 5.6 State machine diagrams (behavioural)

### *5.6.1 Overview*

This section introduces the first of the 13 diagrams that realise a behavioural aspect of the model: the state machine diagram. State machine diagrams are discussed in some detail in Chapter 4 and thus some of this section will serve as a recap. The focus here, however, is on the actual state machine diagram, whereas the emphasis previously has been on general behavioural modelling.

State machine diagrams realise a behavioural aspect of the model and are sometimes referred to as 'timing models', which may, depending on your background, be a misnomer. In UML 1.x, the term 'timing' referred purely to logical time, rather than real time. 'Logical time' here means the order in which things occur and the logical conditions under which they occur. There was no inherent mechanism for representing real time in state machine diagrams in UML 1.x.

This lack of real-time capability is often heralded as one of the shortfalls of the UML but, technically speaking, this was not actually true. The UML has never claimed to be a real-time modelling technique and thus these criticisms fall outside the scope of the UML. Up until now, that is! UML version 2.0 has been expanded to include mechanisms for representing timing information on certain diagrams – one of which is the state machine diagram, where timing attributes may be added to states. There are several extension mechanisms for the UML that claim full real-time capabilities, such as determinism and full temporal modelling. See Douglass' work for more details of real-time modelling using the UML [4, 5].

State machine diagrams model the behaviour of an object or, as it is sometimes written, they model the behaviour during the lifetime of a class, which is the same thing. It has already been stated that state machine diagrams model the order in which things occur and the conditions under which they occur. The 'things' in this context are, broadly speaking, activities and actions. Activities should be derived from class diagrams as they are equivalent to operations on classes. Indeed, one of the basic consistency checks that can be carried out is to ensure that any operations on a class are represented somewhere on its associated state machine as activities or actions.

### *5.6.2 Diagram elements*

State machine diagrams are made up of two basic elements: states and transitions. These states and transitions describe the behaviour of an instance of a class over logical time. States show what is happening at any particular instance in time when an object is active. States may show when an activity is being carried out or when the attributes of an object are equal to a particular set of values. They may even show that nothing is happening at all – that is to say that the object is waiting for something to happen.

Figure 5.33 shows the partial meta-model for state machine diagrams. State machine diagrams have a very rich syntax and thus the meta-model shown here omits some detail – for example, there are different types of action that are not shown.

From the model, it can be seen that a 'State machine diagram' is made up of one or more 'State' and zero or more 'Transition'. A Transition shows how to change between zero or two states. Remember that it is possible for a transition to exit a state and then enter the same state, which makes the multiplicity one or two rather than two, as would seem more logical.

There are four types of 'State': 'Final state', 'Simple state', 'Composite state' and 'Initial state'. Each 'Transition' may have zero or more 'Guard condition', which explains how a transition may be crossed. A condition is a Boolean condition that will usually relate to the value of an attribute.

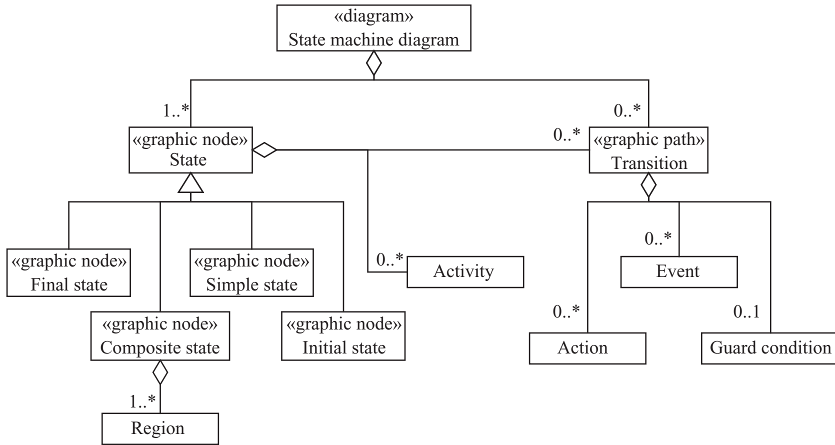


*Figure 5.33 Partial meta-model for state machine diagrams*

A 'Transition' may also have zero or more 'Action'. An action is defined as an activity that takes no time, and one that, depending on one's viewpoint, could be perceived as impossible. In software terms, this may be a very quick activity, such as opening a window.

Finally, a 'Transition' may have zero or more 'Event'. An event is, basically, the passing of a message, usually from one object to another, or to put it another way, from one state machine to another.

There are two main types of 'Event': 'send' and 'receive'. A 'send' event represents the origin of a message being sent from one state machine to another. It is generally assumed that a send event is broadcast to all objects in the system and thus each of the other objects has the potential to receive and react upon receiving the message. Obviously, for each send event there must be at least one corresponding receive event in another statechart. This is one of the basic consistency checks that may be applied to different state machine diagrams to ensure that they are consistent. This is analogous with component and integration testing, as it is possible to (and, in some cases, difficult not to) completely verify the functional operation of an object, yet when it comes to integrating the objects into a system, they do not interact correctly together. Indeed, this is the case in the chess example used in Chapter 4.

Each of these diagram elements may be realised by either graphical nodes or graphical paths, as indicated by their stereotypes and illustrated in Figure 5.34.

Perhaps the simplest and most common source of errors can be avoided by applying a simple consistency check between the state machine and its associated class. This relationship can be seen in Figure 5.35.

Figure 5.35 shows the relationship between a state machine and its associated class.

Each 'Activity' in a 'State' maps to zero or one 'Activity diagram' as any activity may be decomposed into its own diagram. Each 'Transition' in a 'State' maps to one

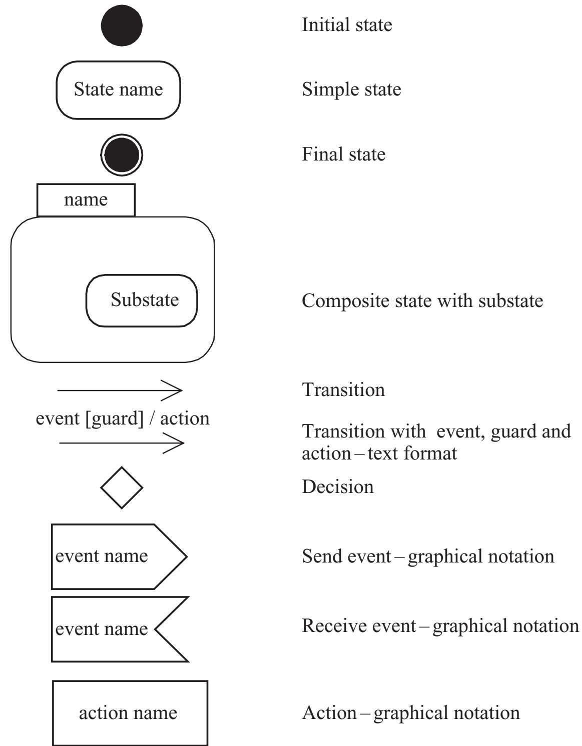


*Figure 5.34 Graphical representation of elements in a state machine diagram*

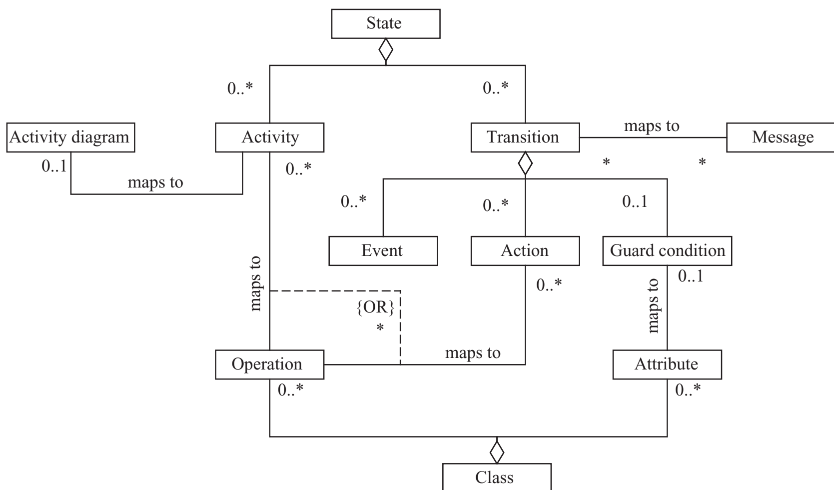


*Figure 5.35 Meta-model relationship between classes and state machine diagrams*

or more ‘Message’. Both an ‘Activity’ or an ‘Action’ may map to an ‘Operation’ from a class, and an ‘Attribute’ from a ‘Class’ maps to zero or more ‘Guard conditions’.

Therefore, the state machine diagram has consistency relationships and, hence, checks to: class diagrams, activity diagrams and all types of interaction diagrams.

### *5.6.3 Examples and modelling – state machine diagrams*

This section looks at some practical examples of modelling and revisits the chess game state machine that was used previously. This time, however, the emphasis is on the actual mechanics of using state machine diagrams, rather than on the behavioural concepts that were covered previously.

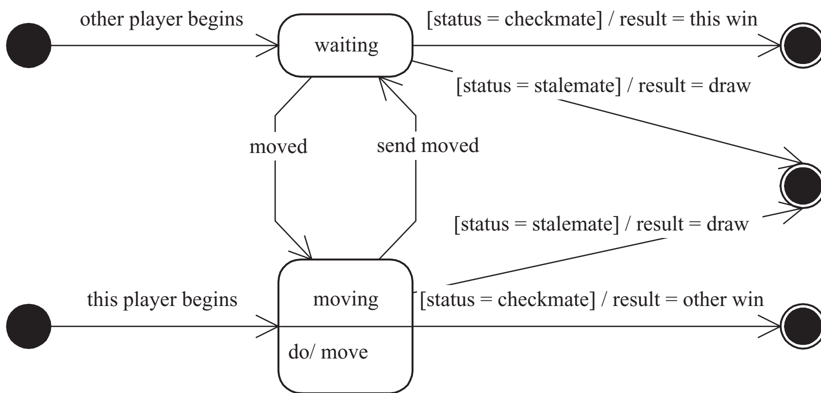


*Figure 5.36 State machine for the ‘Player’ class using basic graphical notation*

Figure 5.36 shows the state machine for the class ‘Player’ from the chess example using the basic graphical notation, as seen in UML 1.x.

Three types of state are shown here:

- A ‘Start’ state, realised by a filled-in circle that indicates the creation of an instance of the class.
- An ‘End’ state, realised by two concentric circles that indicates the destruction of the instance of the class.
- Normal states, indicated by rounded boxes, show what is happening in the object at any particular point in time. It is important to ensure that the rounded boxes actually have vertical straight sides so that it is just the corners that are rounded. The reason for this is that activity diagrams (which are actually a special type of statechart) have a different type of symbol and the only visual difference between this and a normal state is that states in an activity diagram have no straight verticals whereas normal states do. Mixing these types of state can lead to a great deal of confusion, even in state machine diagrams!

Normal states may or may not have activities or actions associated with them that should correspond directly to operations from the class diagram. Each operation from a class must exist as an activity on its associated state machine and, likewise, any activities that exist on the state machine must be present as operations on the governing class. Examples of both are shown here, where the 'moving' state has the 'move' activity, while the 'waiting' state has no activity.

The only way to go from one state to another is to cross a transition. Transitions are realised by directed lines that may have events and conditions on them. Transitions are uni-directional and if a return path of control is required, so too is another transition.

Reading the diagram, the object can begin its life in one of two ways: by the event 'this player begins' or 'other player begins'. If 'other player begins', the first state is 'waiting'. When the 'moved' event occurs, the object enters the moving state when 'move' is executed.

If 'this player begins', the first state is 'moving' where 'move' is executed. When the send event 'send moved' occurs, the object enters the 'waiting' state. The 'send' before the event 'moved' indicates, rather cunningly, that it is indeed a send event rather than a receive event.

There are three possible ends to the game, represented by the three end states:

- If 'status = checkmate' while the object is 'waiting', the attribute 'result' is set to 'this win'.
- If 'status = checkmate' while the object is 'moving', the attribute 'result' is set to 'other win'.
- If 'status = stalemate' from either 'moving' or 'waiting', 'result' is set to 'draw'.

Conditions are used to describe the difference between each transition as a state cannot have two identical conditions associated with it. The condition is a Boolean expression and, in this case, the left-hand side of the condition relates directly to an attribute. This provides rather a nice consistency check back to the state machine diagram's governing class.

Note the use of actions here to set the attribute value to one of its predetermined values. Actions actually have many types in the full UML meta-model, such as: assignments, call, clear, create, destroy, and so on. Actions may be related to operations in a class diagram or may even refer to a simple assignment, or very low-level aspect of a design or specification. Remember again the link back to object diagrams and imagine an object with its attribute values filled in, which would correspond to this state.

In the UML 2.0, a new graphical notation has been introduced to the state machine diagram to make the diagrams easier to read. One criticism of statecharts (as they were known in UML 1.x) was that they can get extremely confused very quickly and, hence, were difficult to read. In order to overcome this, a full graphical notation for events, conditions and actions has been introduced in UML 2.0. Figure 5.37 shows exactly the same information as in Figure 5.36 except, this time, the full graphical notation has been used.

The state machine diagrams used so far have had an emphasis on using states where something happens over a period of time that is represented as an activity.

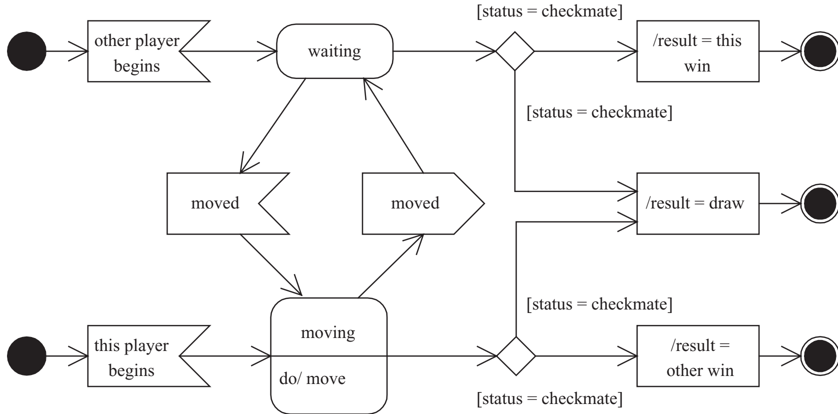


*Figure 5.37 State machine for chess example using extended graphical notation*

There is another approach to creating state machine diagrams that is more event driven and that uses far more actions than activities. Indeed, in many such state machine diagrams there are no activities at all, only actions. An example of an event-driven state machine can be seen in Figure 5.38.

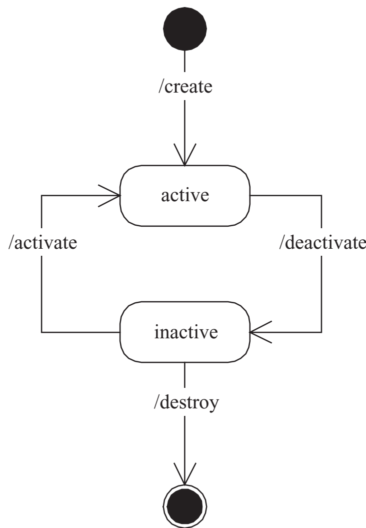


*Figure 5.38 Event-based state machine representing the lifetime of an object using basic graphical notation*

Figure 5.38 shows the state machine that represents the lifetime of an object. It can be seen here that the state name does not have the 'ing' ending as, this time,

it is representing the state when particular conditions are met, rather than when something is actually happening. The things that happen here are shown as actions on the transitions between states.

There is a fine line between whether something that happens should be an action or an activity, but the following guidelines should give some pointers as to which are which:

- Actions are assumed to be noninterruptible – that is, that they are atomic in nature and once they have been fired, they must complete their execution. Activities, on the other hand, may be interrupted once they are running.
- As a consequence of the first point, actions are assumed to take zero time! This, however, refers to logical time rather than real time, which reflects the atomic nature of the action. Activities, on the other hand, take time to execute.
- Activities may be described using activity diagrams, whereas an action, generally speaking, may not.

It is important to differentiate between these activities and actions as they can have a large impact on the way in which the system models will evolve.

Note how the state machine diagram uses the basic graphical notation and, clearly, this can also be shown using the enhanced graphical notation, as shown in Figure 5.39.

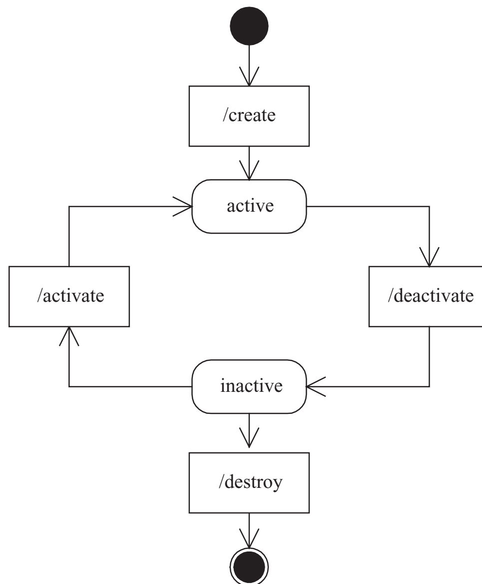


*Figure 5.39 Event-based state machine using enhanced graphical notation*

The diagram in Figure 5.39 shows the same state machine diagram as shown in Figure 5.38 except, this time, the enhanced graphical notation has been used.

### *5.6.4 Using state machine diagrams*

There are a few rules of thumb to apply when creating state machine diagrams:

- All classes that exhibit behaviour (have operations) must have an associated statechart, otherwise the system is not fully defined. This reflects the fact that there must always be the two views of the system and forms a fundamental consistency check for any model.
- All operations in a particular class must appear on its associated statechart. States may be empty and have no activities, which may represent, for example, an idle state where the system is waiting for an event to occur. Messages are sent to and received from other objects, or state machine diagrams.

One question that is often asked concerns which of the two approaches to state machine diagrams is the better: the activity approach or the event-driven approach? There is not much difference between them as it really is a matter of personal preference. However, having said this, the consistency checks for an activity-driven state machine are far stronger that the activity-driven statechart, as the activities are related directly back to the governing class. Many books only use examples where activity-driven state machine diagrams are considered, rather than both types. It should also be borne in mind that it is easier to transform an activity into an action, than it is to transform an action into an activity. Therefore, when creating state machine diagrams, it is suggested that activities are used on the first iteration of the model.

## 5.7 Interaction diagrams

### *5.7.1 Introduction*

This section takes a bit of a departure from the other sections in this chapter, as four diagrams are being discussed rather than one. This is because an interaction diagram can be illustrated using four different types of diagrams: communication diagrams, sequence diagrams, timing diagrams and interaction overview diagrams. All diagrams show the same information, but from slightly different viewpoints. This relationship can be seen in the simple meta-model shown in Figure 5.40.

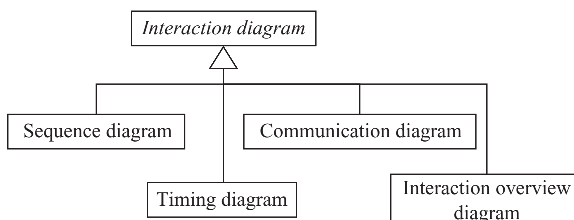


*Figure 5.40 Meta-model showing types of interaction diagram*

Each of the four types of diagrams shows the same information – how a collaboration behaves. The main aim of any interaction diagram is to show a particular example of operation of a system, in the same way as movie makers may draw up a storyboard. A storyboard shows the sequence of events in a film before it is made. Such storyboards in the UML are known as 'scenarios'. Scenarios highlight pertinent aspects of a particular situation and ignore all others. Each of these aspects is represented as an element known as a 'Life line'. This is rather confusing as UML 1.x had an element known as a 'Life line' with a slightly different use. Therefore, to make the difference explicit, each will be discussed below:

- A 'Life line' in UML 1.x represents time going down a page underneath an object, as seen in sequence diagrams.
- A 'Life line' in UML 2.0 represents an individual participant in an interaction, that will refer to an element from another aspect of the model, such as an instance of a class or a part. Where the life line refers to an element with a multiplicity of more than one, and the exact element needs to be explicitly identified, then it is possible to show which element is being referred to using a selector (shown in square brackets). Where the life line refers to an element with a multiplicity of more than one, and the exact element does not need to be explicitly identified, then no selector is shown.

Each of the four interaction diagrams is discussed in detail here.

### *5.7.2 Overview – communication diagrams*

This section introduces and discusses communication diagrams. Communication diagrams realise a behavioural aspect of the model of a system. A communication diagram models interactions between life lines in a system. The emphasis of a communication diagram is on the organisational layout of the diagram elements on the page. This can be very powerful as, quite often, people will like to assign conceptual meaning to where a particular diagram element lies on a page. For example, some people like to show the flow of control going from left to right, whereas others prefer top to bottom. Another possibility is to keep geographically located similar elements on the same part of the page, which can aid comprehension of a problem.

### *5.7.3 Diagram elements – communication diagrams*

Each of the four interaction diagrams will be shown with its own partial meta-model. The partial meta-model for the communication diagram is shown in Figure 5.41.

It can be seen that the 'Communication diagram' is made up of a single 'Frame', one or more 'Life line' and one or more 'Message'. Each 'Message' describes communication between one or two 'Life line'. Also, one or more 'Communication diagram' represents an 'Interaction'.

The 'Frame' element is particularly important when using interaction diagrams, as it allows individual scenarios to be put together to form complex behaviour within other diagrams.

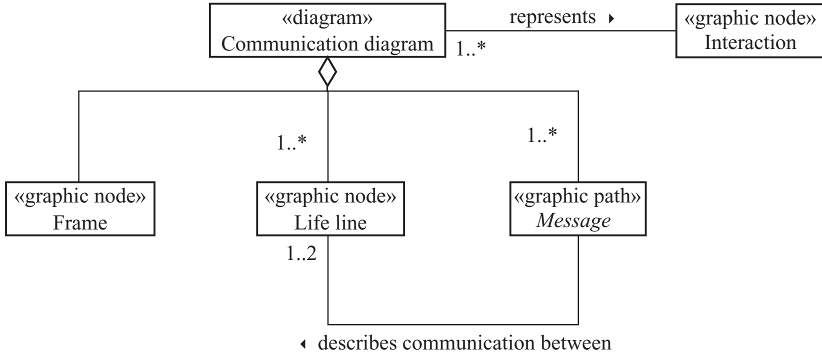


*Figure 5.41 Partial meta-model for communication diagrams*

Each of these diagram elements may be realised by either graphical nodes or graphical paths, as indicated by their stereotypes and illustrated in Figure 5.42.

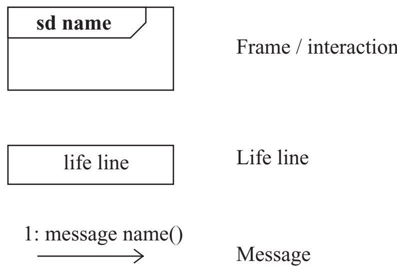


*Figure 5.42 Graphical representation of elements in a communication diagram*

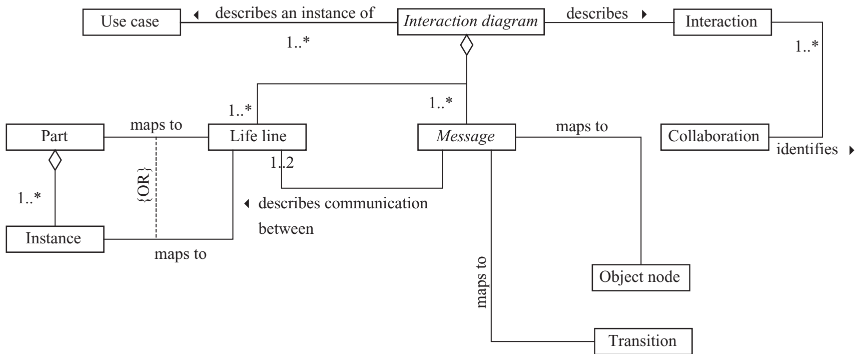


*Figure 5.43 Meta-model showing relationship between communication diagrams and other UML elements*

Figure 5.43 shows the relationships between communication diagrams and how their elements relate to other UML elements. The diagram in the figure shows the relationships between elements of the communication diagram and other aspects of the UML. It can be seen that one or more 'Communication diagram' describes an instance of a 'Use case', which is the same for all interaction diagrams. It can also be seen that each 'Communication diagram' describes an 'Iteration' which in turn is identified by a 'Collaboration'. Each 'Message' maps to an 'Object node' or 'Transition' and describes the behaviour between one or two 'Life line'. Also, each 'Life line' maps to each a 'Part' or an 'Instance'.

### *5.7.4 Examples and modelling – communication diagrams*

Communication diagrams are very useful for demonstrating the consistency of a set of models. In particular, they are crucial for ensuring that the behaviour of the system, at a high level, is correct. State machine diagrams are used to define the behaviour within objects and it is stated that it is very important to ensure that the events that are passed between state machine diagrams (objects) are consistent. One use for communication diagrams is to do exactly this.

Let us revisit the chess example and see how communication diagrams may be used to model that particular system. It may be argued that the chess example is so simple that there is little point in modelling the interaction between two such ordinary objects. However, as will be seen when the models are created, there has been an omission from all our previous chess game models, which only comes to light when the behaviour of a system is modelled at a higher level.

The class diagram for the game of chess indicated that each 'Chess game' was made up of two 'Player'. As the multiplicity is explicit, two instances of 'Player' are created and the behaviour of each instance is dictated by the state machine for 'Player'. The game begins with the creation of two objects – let us call them 'Player 1' and 'Player 2'. There is clearly going to be an interaction between the two objects that will be shown as a link. The message that is passed along this link will be the same

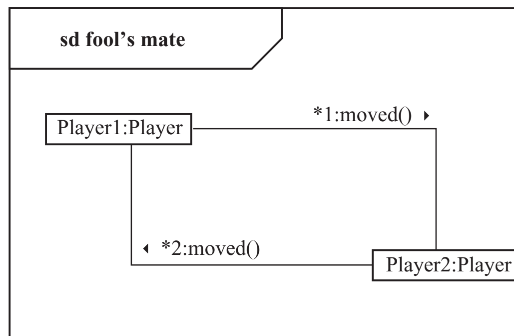


*Figure 5.44 Communication diagram for the chess game example*

as the events passed between the state machine diagrams. From this information, the model in Figure 5.44 can be drawn up.

Figure 5.44 shows the interactions between two life lines, each of which refer to a particular instance of 'Player', in this case shown as 'Player 1' and 'Player 2'. Let us now perform a few simple consistency checks on the model so far to ensure that everything is correct and consistent.

Each life line refers to an object and objects were defined as being instances of classes, therefore each object must have a class from which it takes its appearance and behaviour. A quick look at the class diagram in Figure 5.8 proves that this aspect of the model is consistent.

The message that is passed along the link has already been defined in the state machine for the two objects, so this may be verified as being consistent and correct.

Links were defined as being instances of association, therefore each link must have an association upon which it is based. Another quick look at Figure 5.8 shows that no such association exists. This means that the model is incorrect, as there is an association missing from the original class diagram. The class diagram, therefore, must be modified in order to bring it in line with the rest of the model. This modified class diagram is shown in Figure 5.45.

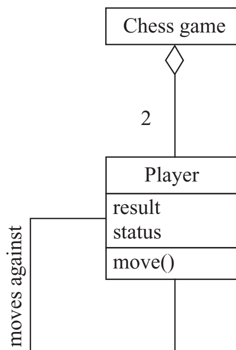


*Figure 5.45 Updated class diagram for the chess game*

Figure 5.45 shows the class diagram from Figure 5.8 but with an additional association. The association is called 'moves against' and provides the template required for all its instances, one of which is represented by the link that can be seen in the communication diagram. Note that the association is bi-directional as no direction is explicitly stated, which will translate into two uni-directional links when realised in the communication diagram. Note also, that as no multiplicity is indicated, it is assumed to be a 'one to one' relationship.

This has highlighted an essential feature of modelling that was first touched upon in the previous chapter, that of levels of abstraction of models. Both state machine diagrams and communication diagrams are types of behavioural models, but they represent different levels of abstraction of the model. It was only when the behavioural

model was looked at from both levels that an error was thrown up. Communication diagrams represent a high level of abstraction of any model, whereas a state machine shows a lower level of behaviour. It is vital to ensure that consistency checks are performed on, and between, models of the system.

This has also highlighted another crucial feature of the UML, which is that no matter how simple the model, it is almost impossible to get it right on the first attempt. Achieving a good, consistent model requires a great deal of iteration between various types of diagrams and the more modelling that takes place, the more consistent the model will become and the closer the connection to reality will be.

### *5.7.5 Using communication diagrams*

There are several rules of thumb that should be borne in mind when creating communication diagrams:

- Life lines in a communication diagram will always refer to another element in the model, such as instances of classes. This is true whether the other elements have been defined or not. If the elements have been defined, all is well and good. If, however, elements have not been defined, it is a good approach to identifying elements such as classes and parts based on example scenarios (perhaps created from particular use cases).
- In the same way, links always refer back to associations. This can prove to be a good consistency check back to the class diagram.
- Messages may be derived from state machine diagrams – if they already exist. If not, each message may show which send and receive events may be required for state machine diagrams.
- One final point that should be borne in mind is that communication diagrams model behaviour according to organisation or, to put it another way, the layout on the page becomes important.

Communication diagrams are typically used to model scenarios; however, there is another use that is rarely mentioned in the literature, which is that of a consistency check for lower-level behavioural diagrams, such as activity diagrams and state machine diagrams. This point was raised in Chapter 4 when the state machine for the class 'Player' was verified as working at one level, but it took a higher-level view to reveal that the messages defined on the state machine were incorrect. This also relates to Chapter 2 where it was stated that models should be looked at from more than one level of abstraction to ensure that the overall model is correct. Communication diagrams (and all the other interaction diagrams) are very powerful when used as a consistency check between various interacting life lines that already have their internal behaviour defined with state machine diagrams.

### *5.7.6 Overview – sequence diagrams*

This section introduces and discusses sequence diagrams, which realise a behavioural aspect of the model. Sequence diagrams are one of the types of interaction diagram

and, as such, behave very much like communication diagrams. Indeed, the information shown by both these types of interaction diagram is actually the same, but from a slightly different point of view. As such, sequence diagrams show life lines and the messages passed between them but with an emphasis on logical time or the sequence of messages.

The main difference between a communication diagram and a sequence diagram is that communication diagrams show the high-level behaviour of a system from an organisational point of view, whereas sequence diagrams show the high-level behaviour of a system from the logical timing point of view.

### *5.7.7 Diagram elements – sequence diagrams*

Sequence diagrams, are composed of the same basic elements as the communication diagram except, this time, there are more elements present which provide the sequence diagram with a very rich syntax.

Figure 5.46 shows that a 'Sequence diagram' is made up of a single 'Frame', zero or more 'Gate', one or more 'Message', one or more 'Life line' and zero or more 'Interaction occurrence'. Each life line in a sequence diagram has a dotted line underneath that indicates the order in which things happen (such as messages and execution occurrences) and shows when the life line is alive. The death, or termination of a life line is indicated by a 'Stop'. An interesting extra piece of syntax included here is the 'Interaction occurrence' that allows, effectively, another scenario to be inserted into a sequence diagram. This means that scenarios showing repeated behaviour can be bundled into a scenario on a separate interaction diagram and then inserted at the relevant point in time into the life line sequence.

Another new element here is the 'Gate' that allows a message to enter into a sequence diagram from somewhere outside the diagram. This is useful for showing the message that invokes, or kicks off a particular interaction without specifying exactly where it has come from (although this must be specified at some point in the model).

The final new element shown here is the 'Event occurrence' that shows when a message occurs on the life line.

Each of these diagram elements may be realised by either graphical nodes or graphical paths, as indicated by their stereotypes and are illustrated in Figure 5.47.

The diagram in Figure 5.47 shows the graphical notation used in sequence diagrams. The main symbol here is the life line with the dotted line underneath it that represents the life of the life line. It is from such life lines that all the behaviour of the interaction is shown.

### *5.7.8 Examples and modelling – sequence diagrams*

In order to show the differences between the various types of interaction diagram, the same example will be used for each – the chess example. Figure 5.48 shows the sequence diagram for the chess game, which gives the same information as the communication diagram in Figure 5.44. The life lines in the system are realised in the

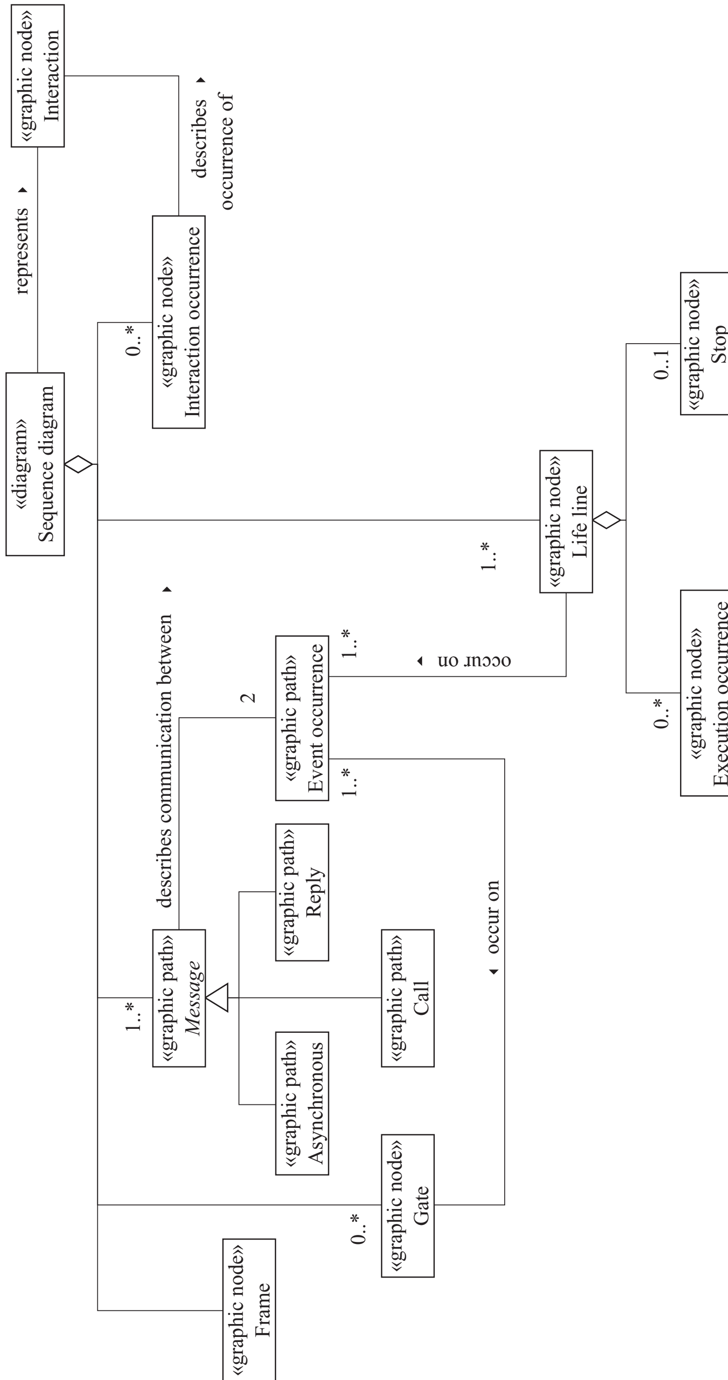


*Figure 5.46* *Partial meta-model for sequence diagrams*

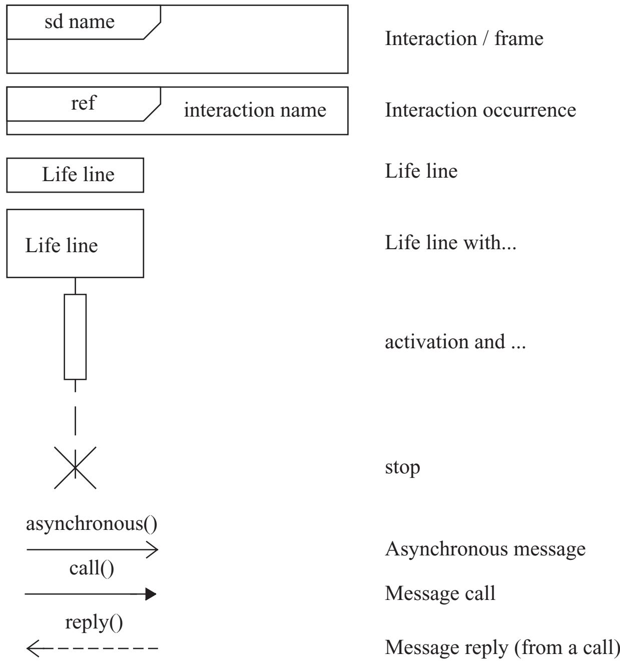


*Figure 5.47 Graphical representation of elements in a sequence diagram*

usual way: life line name, colon, class name, all underlined. Two objects are shown here: 'Player 1' and 'Player 2'. These will both be created at the same time – when the game of chess begins – and, as such, both life lines appear at the same height on the sequence diagram.

The diagram in Figure 5.48 shows the sequence diagram for the chess example and, before the actual behaviour of the diagram is discussed, it is worth looking at the graphical notation being used. First of all, note that the message that starts off the whole game comes from the edge of the frame, rather than from another life line – this an example of a gate. The boxes shown on each of the life lines indicate when the life line is executing something or, to put it another way, is busy. Each time a message is sent, there is an event occurrence at the point where the message joins the dotted life line. The life line is terminated using the stop symbol, represented by the large 'X' on the dotted line.

In terms of how the game is played, it can be seen that 'Player 1' begins the game and, hence, the first execution occurrence is with 'Player 1'. All the time that 'Player 1' has an active life line (shown visually by the focus of control), 'Player 2' has an inactive life line (no execution occurrence present). The execution occurrence may now be related back to the original state machine to see if the models are consistent – remember that both models show behaviour but at different levels of abstraction. When 'Player 1' has an active life line, this relates to the 'Player 1' state

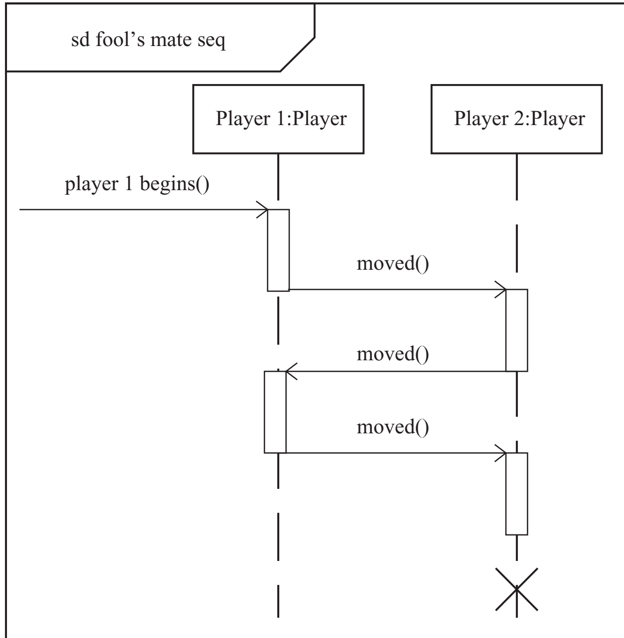


*Figure 5.48 Sequence diagram for the chess example*

machine being in the 'moving' state. Likewise, when 'Player 1' has an inactive life line, this relates to when the 'Player 1' state machine is in the 'waiting' state.

The execution occurrence for the game of chess passes from 'Player 1' to 'Player 2' once 'Player 1' has moved. This is indicated by the message 'moved' that is sent via the first link to 'Player 2'. 'Player 2' behaves in exactly the same way as 'Player 1' as they have the same statechart, hence the execution occurrence passes between 'Player 1' and 'Player 2' as the game progresses, until one player wins and the game is over. The end of the game and the end of both object's lives occur when one player wins or there is a draw. The end of the life line is indicated by a large 'X' life line, which terminates it. This is directly comparable with the end state from the original statechart.

The second example is that of the system life cycle model that was discussed in the communication diagram.

Figure 5.49 shows the sequence diagram for a life cycle model. This model shows the system life cycle without emphasis is on the sequence of the interactions.

From the model, the first thing that happens is that an instance of 'User requirements', called 'UR', is created, which has its own execution occurrence. When the execution occurrence shifts to the next object, the message will actually cause the creation of the second object, 'System spec', which is shown across from 'User requirements' but lower down the diagram, which indicates that it is created some time after the first object was created.

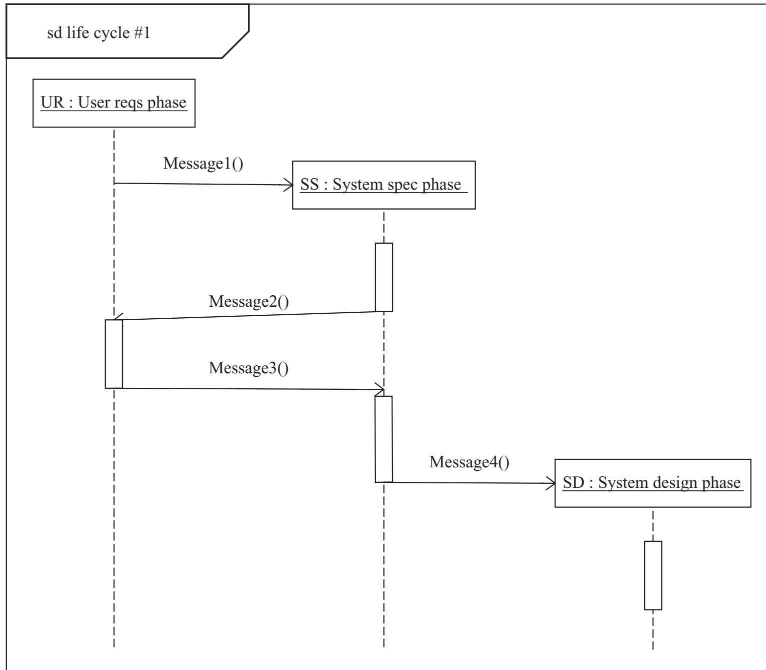


*Figure 5.49 Sequence diagram for a life cycle model*

The next thing that happens is that the execution occurrence is passed back to the first object, 'UR', indicating that, for some reason, the phase had to be revisited. In reality, this would relate to an iteration of the system life cycle. When the execution occurrence passes back to 'SS', the life line becomes active once more. The focus then shifts to the third object, 'SD', which is created for the first time, and so forth.

### 5.7.9 Using sequence diagrams

The rules of thumb for the sequence diagrams are very similar to those for the communication diagram.

Both types of interaction diagram are semantically equivalent, in that they are based on the same information in the system. Indeed, it is a relatively simple matter to convert between them and this task may be performed automatically by some tools.

The main difference is that the emphasis is on logical timing, or sequence, in the sequence diagram. Communication diagrams give a better indication of how two life lines are linked together but have weak logical ordering capabilities, as they are based on a simple numbering of events. Sequence diagrams, on the other hand, are weak at showing how life lines are linked but stronger in showing the order in which things happen, including return paths of control.

An analogy that is often quoted for the two types of diagrams is to think of Gantt charts and Pert charts in project management. A communication diagram is analogous

to a Pert chart and a sequence diagram is analogous to a Gantt chart. Indeed, one use for sequence diagrams is to use them to plan Gantt charts as they show behaviour over logical time, which is a good basis for a Gantt chart.

### *5.7.10 Overview – interaction overview diagrams*

The third type of interaction diagram to be discussed is a completely new diagram to the UML, which is the interaction overview diagram. Interaction overview diagrams are used to assemble complex behaviours from simple (or simpler) scenarios. For example, it may be that one scenario covers the initialisation of a system whereas there are three main modes of operation. Although this is easy to model in the UML by creating four different scenarios, it is often difficult to get an idea of the 'bigger picture' of the overall operation of the system. In fact, quite often people will use an activity diagram incorrectly to explain the high-level operation of a system as the paths in an activity diagram (decisions, control forks, joins etc.) lend themselves to showing an overall behaviour. The interaction overview provides the functionality of the paths of an activity diagram but at a high level, where the nodes represent entire interactions presenting a high-level, complex operation of the system. In fact, it is possible to include an iteration that is realised by any of the four interaction diagrams as an interaction – even an interaction overview diagram.

### *5.7.11 Diagram elements – interaction overview diagrams*

The interaction overview diagram is quite unusual as it looks, at first appearance, rather like an activity diagram. Indeed, it has similar syntax to an activity diagram with regard to the graphical paths and some graphical nodes but instead of states, the basic graphical node in the interaction overview diagram is the interaction occurrence.

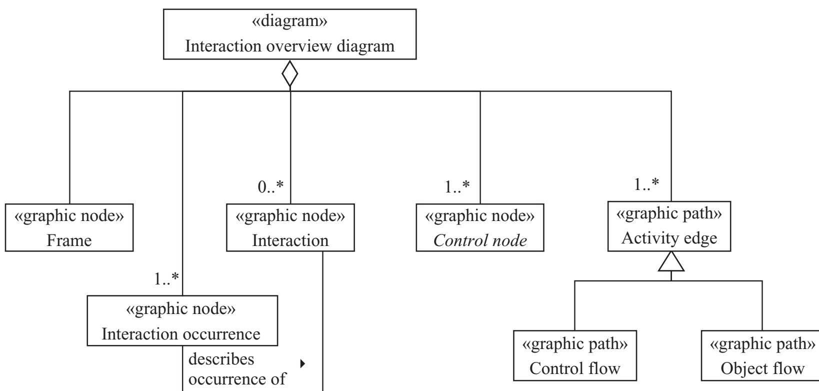


*Figure 5.50 Partial meta-model for the interaction overview diagram*

The diagram in Figure 5.50 shows the partial meta-model for the interaction overview diagram. Note the similarities to the other two types of interaction diagram

that have been discussed so far – the communication and sequence diagrams. Like the other two diagrams, the 'Interaction overview diagram' is made up of a single 'Frame' and number of 'Interaction' and a number of 'Interaction occurrence'. However, this time the elements that happen to be graphical paths have been borrowed from the activity diagram – one or more 'Control node' (of which there are many types not shown here) and one or more 'Activity edge' that has two types: 'Control flow' and 'Object flow'. The symbols for these elements are shown in Figure 5.51.

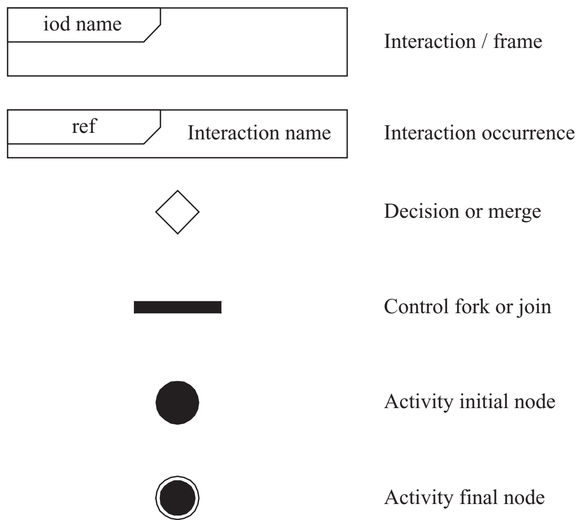


*Figure 5.51 Graphical representation of elements in an interaction overview diagram*

The diagram in Figure 5.51 shows the graphical representation of elements in an interaction diagram. Note how the symbols for the interaction and interaction occurrence are the same as the other types of interaction diagrams introduced so far, whereas the rest of the symbols look like the activity diagram symbols.

### *5.7.12 Examples and modelling – interaction overview diagrams*

Consider, once again, the chess example and imagine how many possible scenarios there are that could model the full potential functionality of the system. There are rather a lot! In fact, it is impossible to visualise every possible scenario in one view so the obvious thing to do is to break the behaviour down into a number of lower level, easier to understand scenarios. For the purposes of this example, let us consider four main scenarios:

- 'game set up', that describes a scenario that must happen at the beginning of every game, when the board is set up, the pieces are placed on the board and the opening player is decided upon.

- 'normal play', that describes the routine playing of the game – each player moving in turn, etc.
- 'player 1 wins', where player 1 wins the game.
- 'player 2 wins', where player 2 wins the game.

It should be clear by now that each of these scenarios could be modelled using either sequence or communication diagrams. In fact, the scenario 'normal play' could be represented by the diagrams shown in either Figure 5.44 or Figure 5.48. Each of these diagrams, according to their partial meta-models, should have a frame with a diagram identifier and a unique name. However, how do these four scenarios relate to one another? What is the overall behaviour of the system in this case? Consider the interaction overview diagram (Figure 5.52).

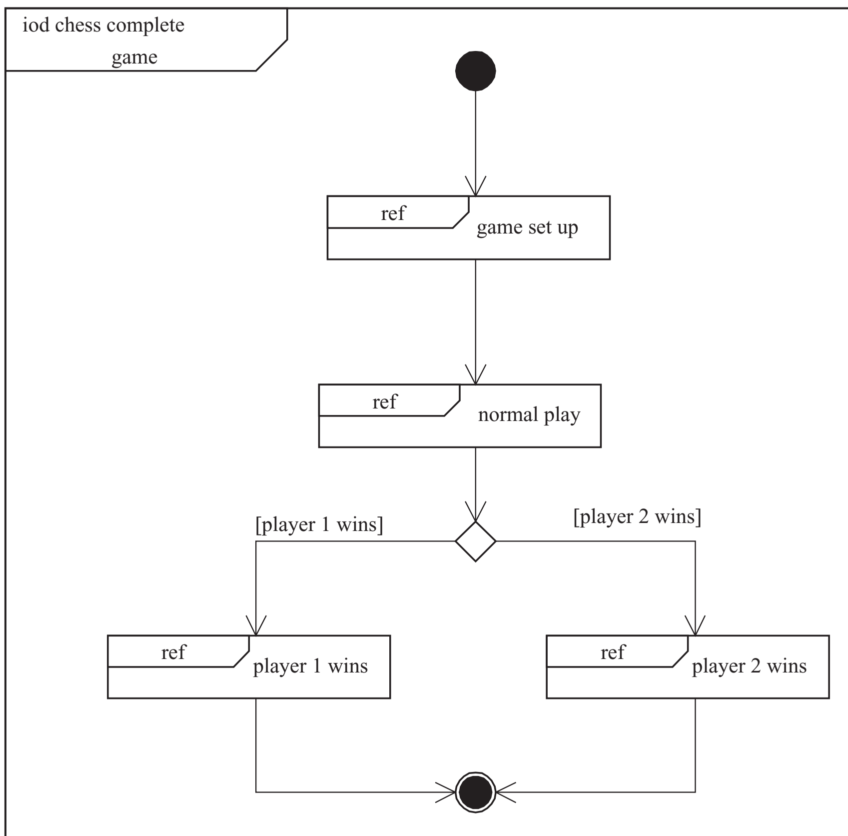


*Figure 5.52 Example of interaction overview diagram for a number of interactions*

The diagram in Figure 5.52 shows an interaction overview diagram for the chess game, using the four scenarios, or interactions that have just been defined. Notice how

much this diagram looks like an activity diagram but, this time, the main graphical nodes are interactions and, in fact, the whole diagram itself is a single interaction also!

The interaction shown in this diagram shows an occurrence of an interaction named 'game set up'. It is clear that the graphical node is representing a predefined occurrence of an interaction since the keyword 'ref' appears in the name box of the graphical node, whereas the body of the graphical node contains the unique name of the interaction diagram that specifies the interaction. Notice that it is the interaction name that is referenced, rather than a specific diagram name (there is no diagram identifier before the unique interaction name). This is because it may be that there are several different types of interaction diagram used to model a single interaction and it is this interaction that is important here, rather than any specific diagram.

Once the first scenario has been executed, the flow moves onto the 'normal play' scenario which continues until a decision point is reached regarding who the winner of the game is, represented by the decision branch. A single path is then chosen which will lead to either the 'player 1 wins' scenario or the 'player 2 wins' scenario and then onto the final state of the diagram.

### *5.7.13 Using interaction overview diagrams*

Interaction diagrams should be used to represent high-level system behaviour by combining a number of simple scenarios into complex behaviour. This does not mean, however, that the scenarios must preexist in order to use an interaction overview diagram, as they can be used at early points in the modelling cycle as a way to identify interactions that must exist within the system. This then has a clear effect on other aspects of the model, for example, interactions are related directly to collaborations, hence a strong link back to composite structure diagrams and, hence, class diagrams.

Interaction overview diagrams can also be a very good way to ensure consistency between different interactions from a system validation point of view. If one considers that scenarios are a very good way to validate individual requirements, then interaction overview diagrams are an excellent way to validate high-level requirements that have complex behaviours associated with their validation.

### *5.7.14 Overview – timing diagrams*

Timing diagrams are one of the new diagrams that have been introduced into UML 2.0 and they are intended to bridge the gap between UML and the several real-time approaches that exist in the modelling world. The world of real-time systems modelling is a large one and has been the subject of many debates for several years. In fact, one of the main criticisms aimed at UML 1.x was its lack of support for any type of real-time analysis. The original response from the authors was one of stating that UML was not intended to be a real-time modelling language. However, as time has gone on, more and more people have been tailoring the UML to allow real-time modelling capabilities so the UML 2.0 has introduced several mechanisms to allow this. Several diagrams now allow the inclusion of real-time information, such as the state machine diagram and the sequence diagram, but there is now a single diagram that is dedicated to realising timing aspects of interaction – the timing diagram.

The timing diagram allows timing information to be added to interactions and allows two main visual views of the timing behaviour – one based on general timing values and one based on states and condition on a time line. It should be pointed out at this juncture that there is no process nor methodology for real-time modelling in UML 2.0, just the mechanisms to allow timing to be associated with the model. This means that any real-time modelling methodology should be applicable to a UML 2.0 without having to tailor the diagram elements. For information regarding real-time modelling techniques, see Reference 4.

### *5.7.15 Diagram elements – timing diagrams*

The timing diagram allows timing information to be included as part of the UML model at the scenario or interaction level. The basic elements of the interaction diagram are still present but, this time, new elements have been introduced to allow real-time modelling. The graphical notation for timing diagrams is shown in the partial meta-model in Figure 5.53.

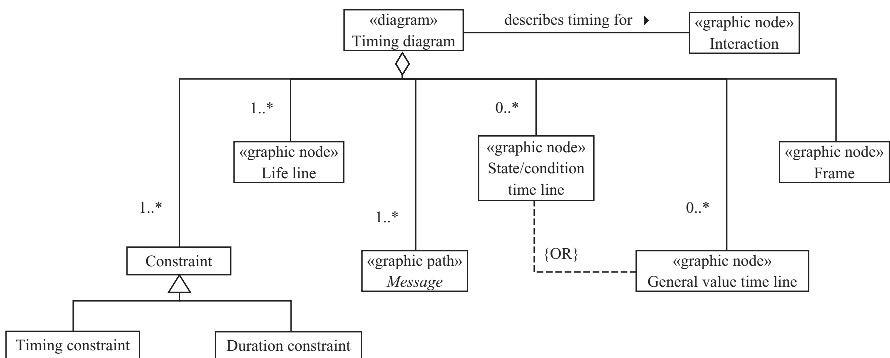


*Figure 5.53 Partial meta-model for timing diagrams*

The diagram in Figure 5.53 shows the partial meta-model for the timing diagram. It can be seen that the basic elements are still there – a 'Timing diagram' is made up of one or more 'Life line', one or more 'Message' and a single 'Frame'. Several new elements, however, have been introduced. Two types of graphical node have now been introduced that form the basis for the timing diagram and either one can be chosen as the main focus of the diagram. The 'State/condition time line' allows an emphasis to be put on the timing of the states of the system, whereas the 'General value time line' allows the emphasis to be put on the actual timing values. Time can be shown in two ways by the inclusion of a 'Constraint' that can be either a 'Timing constraint' or a 'Duration constraint'.

One of the key points of the timing diagram is how timing information is visualised, hence, the graphical notation associated with the timing diagram elements holds more visual information than most UML diagram elements.

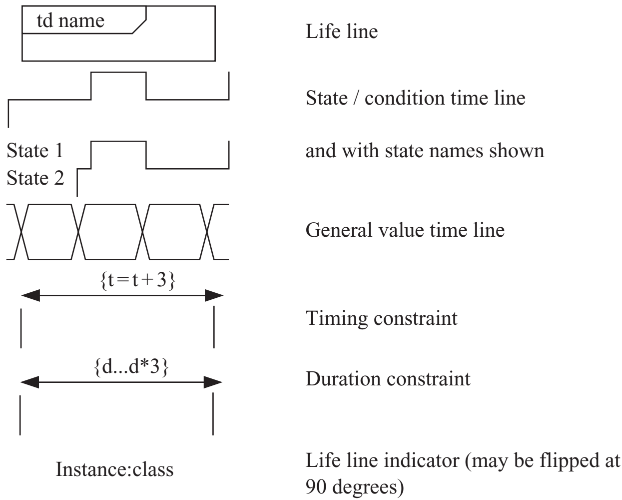


*Figure 5.54 Graphical representation of elements in a timing diagram*

The diagram in Figure 5.54 shows the graphical notation of elements in the timing diagram. The basic element of the standard 'Frame' is the same as other interaction diagrams. The 'State/condition time line' shows what most people associate with a generic timing diagram – a line that goes from left to right representing time and that changes level according to value. In this case, the values refer to states (check for consistency with state machine diagrams) which may or may not have state names shown.

Another way to show the same information visually is by using the 'General value time line' that shows, again, time going from left to right, however, this time there are no separate levels to indicate state, but blocks of time that may or may not have the state name written inside. This sort of visualisation may be more familiar to people who have worked in logic circuit simulations, where time is often represented in this format.

Time itself may be represented on the diagram by showing basic timing constraints associated with the length of the time line. This may either be in explicit time using the 'Timing constraint' or in terms of durations using the 'Duration constraint'.

On timing diagrams, several life lines may be shown on a single diagram, so individual life lines are simply referenced using text on the left-hand side of each time line and which may be flipped by 90 degrees to make the diagram more compact.

### *5.7.16 Examples and modelling – timing diagrams*

The timing diagram is a type of interaction diagram and, hence, should show the same sort of information as the other diagram but with an emphasis on visualising time. Figure 5.55 represents exactly the same information as the diagrams shown in Figure 5.44 and Figure 5.8 but visualises time.

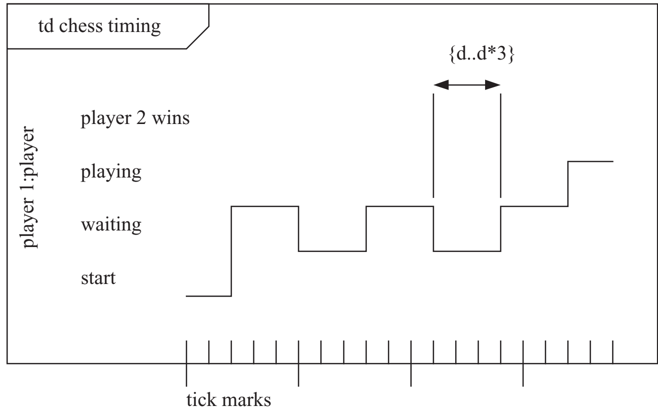


*Figure 5.55 Example of timing diagram with an emphasis on state/condition*

The diagram in Figure 5.55 shows the timing diagram for the chess example but highlights the timing of only one of the life lines. The life line is indicated on the left of the diagram with the words ‘Player 1:Player’ at 90 degrees to the page – the same text notation as shown previously to indicate a life line. Along the bottom of the

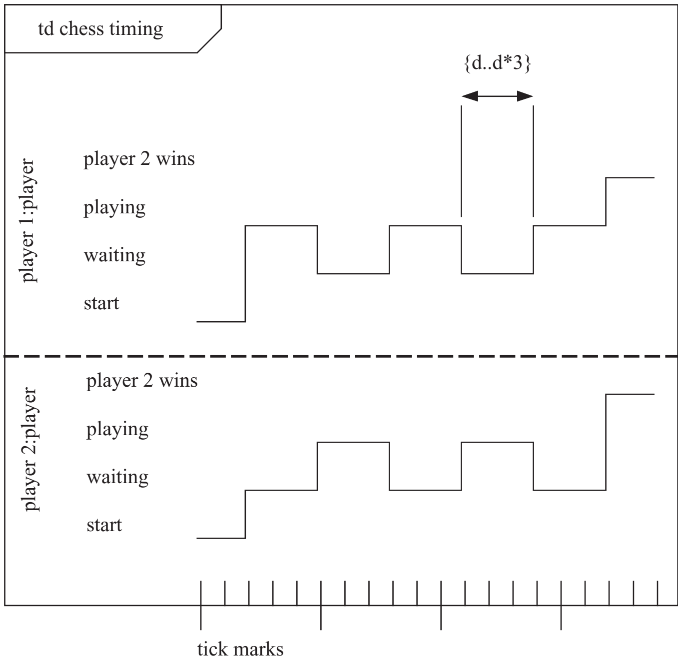


*Figure 5.56 Timing diagram showing more than one life line*

diagram there are several tick marks that represent the granularity of time. This can be any sort of time, for example explicit time, relative time, cyclic time, and so on. The states that the life line can be are shown on the left-hand side of the main time line and should be traceable directly back to their appropriate state machine diagrams. The main time line flows from left to right and changes level according to the state that the life line is in at any point in time.

It is possible to show more than one life line on a single timing diagram that can be very important when it comes to resolving synchronisation issues between different elements in the system. An example of a timing diagram with more than one life line can be seen in Figure 5.56.

The diagram in Figure 5.56 shows the same system but, this time, there are two life lines. The two life lines are separated by the dotted-line separator yet both share the same time scale at the bottom of the diagram.

The other main approach to visualising the timing aspect of a model is to view time as generic values, rather than by state and this can be seen in Figure 5.57.

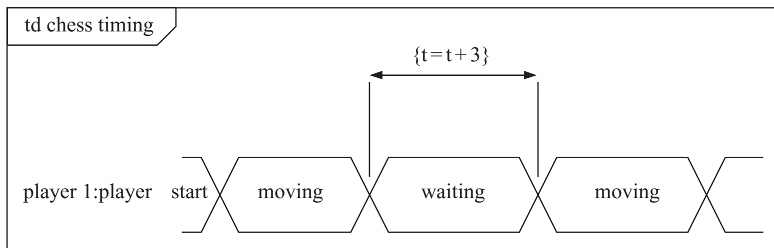


*Figure 5.57 Example timing diagram with an emphasis on general timing values*

The diagram in Figure 5.57 shows the same information as the previous two diagrams except that this time the main graphical element used is the 'General value time line' rather than the 'State/condition time line'. This view emphasises the timing values more than the state of the life line at any point in time, although the state names can still be shown on the diagram.

### *5.7.17 Using timing diagrams*

Timing diagrams should be used where real time is an issue. When modelling any sort of real-time system, particularly when wanting to carry out any sort of timing analysis, it is important to have some sort of process or methodology behind the timing. With the vanilla UML, there is no inherent process nor methodology but there are many approaches well documented in the public domain.

In fact, many commercial CASE tools have in-built real-time modelling capabilities that have provided one of the main inputs to the development of the timing diagrams in UML 2.0.

## 5.8 Activity diagrams

### *5.8.1 Overview*

This section looks at another behavioural model of a system: the activity diagram. Activity diagrams, generally, allow very low-level modelling to be performed compared to the behavioural models seen so far. Where interaction diagrams show the behaviour between objects, and state machine diagrams show the behaviour within objects, activity diagrams may be used to model the behaviour within an operation, which is about as low as it is possible to go.

The other main use for activity diagrams and certainly the most commonly seen usage of activity diagrams in other texts is to model 'workflows'. This will be discussed in some detail later in this chapter as the level of abstraction of a workflow can vary.

Activity diagrams are actually special types of state machine diagrams and, as will be seen, the constructs are very similar between the two types of diagram. Activity diagrams are also similar to traditional flow charts (from which they were derived), which is often a cause for comment or even criticism of the UML.

### *5.8.2 Diagram elements*

Activity diagrams have changed considerably in UML 2.0 and, indeed, some of the fundamental terms used in UML 1.x have now been replaced. Perhaps the largest difference is the basic type of graphical node – in UML 1.x there were states (action states and activity states) whereas in UML 2.0 these have been replaced by activity invocations. Figure 5.58 shows this and all the other main elements in an activity diagram.

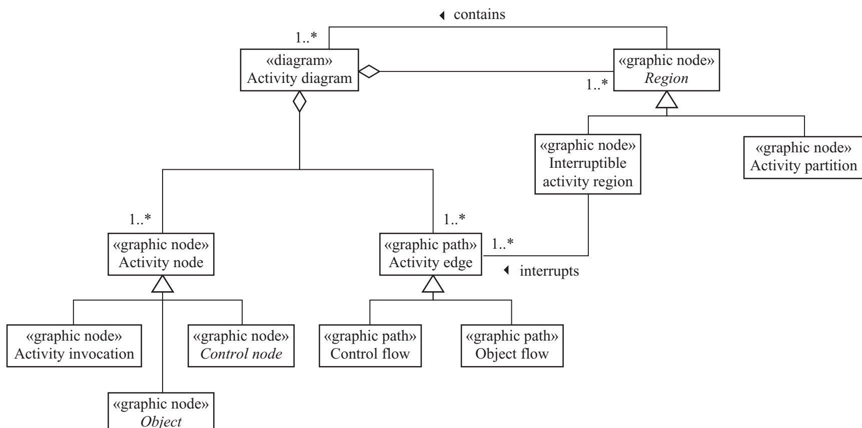


*Figure 5.58 Partial meta-model for activity diagrams*

Activity diagrams are made up of three basic elements: one or more ‘Activity node’, one or more ‘Activity edge’ and one or more ‘Region’. There are three main types of ‘Activity node’ which are the ‘Activity invocation’ and the ‘Object’ and ‘Control node’ both of which will be discussed in more detail later in this section. The ‘Activity invocation’ is where the main emphasis lies in these diagrams and it is through activity invocations that it is possible to establish traceability to the rest of the model via operations, activities and actions.

The ‘Activity edge’ element has two main types – ‘Control flow’ and ‘Object flow’ – both of which are the same as in UML 1.x and whose meaning is self explanatory.

The other major element in an activity diagram is the ‘Region’ that has two main types: ‘Interruptible activity region’ and ‘Activity partition’. An ‘Interruptible activity region’ allows a boundary to be put into a diagram that encloses any activity invocations that may be interrupted. This is particularly powerful for software applications where it may be necessary to model different areas of the model that can be interrupted, for example, by a direct user interaction or some sort of emergency event. The ‘Activity partition’ is the mechanism that is used to visualise swim lanes that allow different activity invocations to be grouped together for some reason – in the case of swim lanes for responsibility allocation.

It is worth, at this point, going into more detail with regard to control nodes and objects as there are several types of each. It should also be borne in mind that these types of control nodes are also applicable to the interaction overview diagram, so it is worth spending a little more time discussing the finer points of the syntax.

The diagram in Figure 5.59 shows an expanded view of the types of ‘Control node’ that exist in UML 2.0. Most of these go together in twos or threes and therefore will be discussed together.

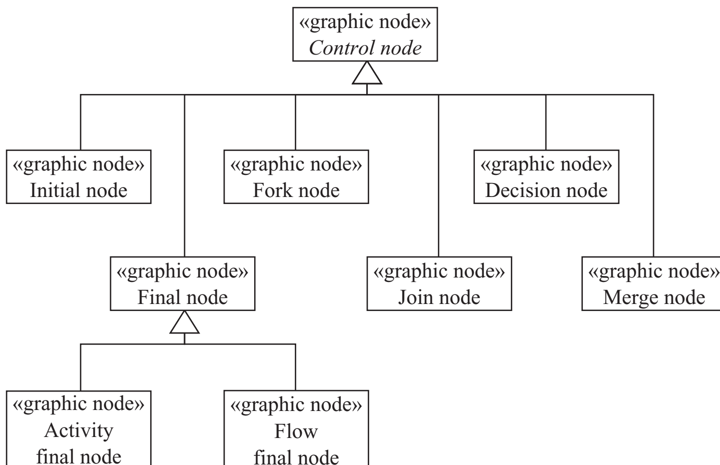


*Figure 5.59 Expanded partial meta-model, focusing on ‘Control node’*

- The 'Initial node' shows where the activity diagram starts, and kicks off the diagram. Conversely, the end of the activity diagram is indicated by the 'Activity final node'. These are all the same as in activity diagrams in UML 1.x, but there is also a new type of 'Final node' which is the 'Flow final node'. A 'Flow final node' allows a particular flow to be terminated without actually closing the diagram. For example, imagine a situation where there are two parallel control flows in a diagram and one needs to be halted whereas the other continues. In this case, a final flow node would be used as it terminates a single flow but allows the rest of the diagram to continue.
- The 'Join node' and 'Fork node' allow the flow in a diagram to be split into several parallel paths and then rejoined at a later point in the diagram. Forks and joins use the concept of 'token passing' that was used in Petri-net systems modelling, which basically means that whenever a flow is split into parallel flows by a fork, imagine that each flow has been given a token. These flows can only be joined together again when all tokens are present on the join flow. It is also possible to specify a Boolean condition on the join to create more complex rules for rejoining the flows.
- The 'Decision' and 'Merge' nodes also complement one another nicely. A 'Decision' node allows a flow to branch off down a particular route according to a condition, whereas a 'Merge' node allows several flows to be merged back into a single flow.

Each of these diagram elements may be realised by either graphical nodes or graphical paths, as indicated by their stereotypes and are illustrated in Figure 5.60.

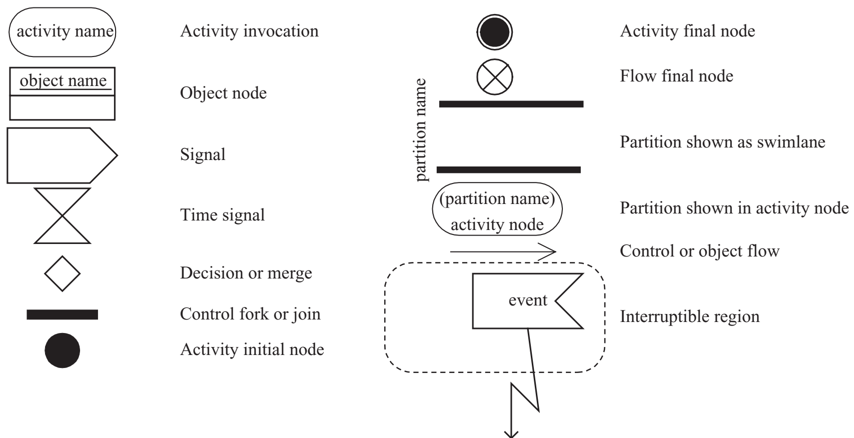


*Figure 5.60 Graphical representation of elements in an activity diagram*

The diagram in Figure 5.60 shows the graphical notation for all the elements mentioned so far. One point to note is that the activity invocation element symbol is not the same as the state symbol used in state machine diagrams – the state symbol is

a rounded box (it has four distinct sides) whereas the activity invocation is a 'sausage' shape (two straight edges and two rounded ends).

The object node symbol is the same as any other object symbol, but there are now two special types of objects that are defined – the 'Signal' and 'Time signal'. The 'Signal' symbol is used to show signals passed in and out of the activity diagram – the same as in a state machine diagram, whereas the 'Time signal' allows the visualisation of explicit timing events.

Partitions can be shown in one of two ways – the traditional swim lane way by drawing a box or two parallel lines enclosing the relevant region, or by simply writing the partition name in brackets above the activity name in the activity invocation symbol. Interruptible regions are shown by a dashed line with an arrow coming out of it to show where the flow goes in the event of an interruption.

### *5.8.3 Examples and modelling*

The two main uses for activity diagrams are to model workflows and operations. Conceptually, modelling operations is far simpler to understand than modelling workflows. The first example, therefore, shows how to model the behaviour of an operation. This would be a procedure when applied to software or a general algorithm when applied to general systems (perhaps a thought process or hardware design).

The first example shows how to model a simple software algorithm that has been taken directly out of a programming algorithms book [17].

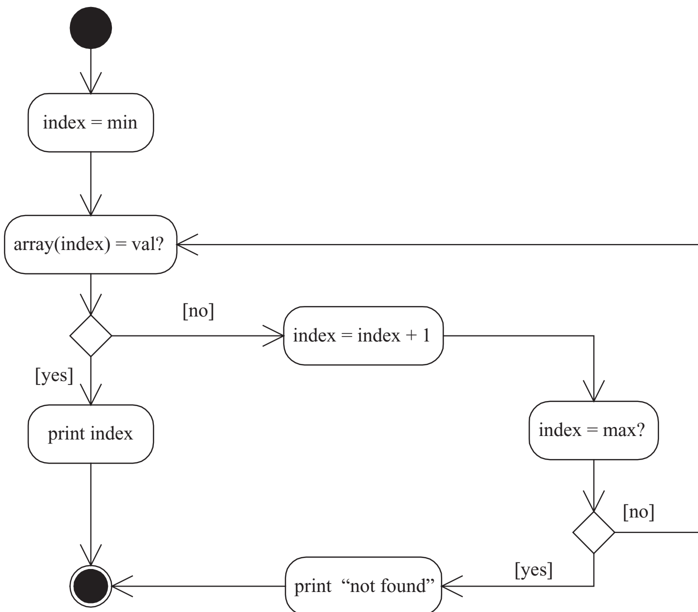


*Figure 5.61 Activity diagram describing the behaviour of an operation*

Figure 5.61 shows how a standard-defined algorithm for searching an array for a particular value may be modelled. The algorithm has been taken straight from a book on structured programming, which basically describes a library of useful algorithms that may be implemented in any programming language.

From the model, the first thing that happens is that 'index = min' where the attribute value (from an associated class diagram) is set to a predefined value of 'min'. Next, the array value indicated by 'index' is compared with 'val'. If the result of this comparison is 'yes' then 'print index' occurs, otherwise 'index = index + 1' occurs, where the 'index' attribute is incremented by one. Next 'index' is compared to a predefined value called 'max' to see if the end of the array has been reached. If not, the model reverts back to comparison state, otherwise the 'print 'not found'' state is entered.

Activity diagrams are very useful for modelling complex algorithms with much iteration, but may be 'overkill' for modelling very simple operations, particularly if many exist within the design. Indeed, some sort of simple functional descriptions, such as informal pseudocodes may be more appropriate for very simple operation definitions.

The activity diagram can also be used to model any low-level conceptual operation that need not be software, as shown in Figure 5.62.

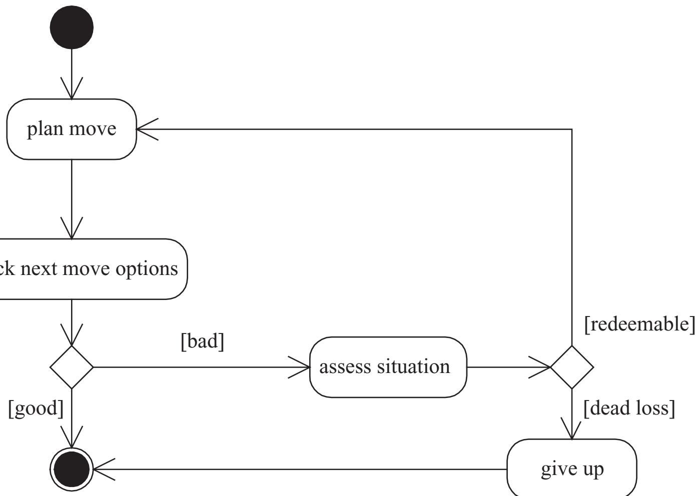


*Figure 5.62 Activity diagram showing behaviour of a thought process of how to play chess (badly)*

Figure 5.62 shows the 'move' operation from the original chess example. It can be seen that the first thing that happens is 'plan move'. The next is 'check next move options', which is then followed by a decision branch. There are two outcomes: 'bad', which leads to 'assess situation', and 'good', which leads to the end of the activity

diagram. Immediately following 'assess situation' is another decision branch where 'redeemable' leads back to 'plan move' and 'dead loss' leads to a 'give up' activity invocation.

The most widely used application (certainly from most other literature's point of view) for activity diagrams is concerned with modelling workflows. This can cause some confusion as the term 'workflow' is very-much associated with the Rational Unified Process (RUP) [3, 16], which has its own distinct terminology that can be different from other terminology such as the one adopted by ISO. Two examples of modelling workflows will be considered, one of an ISO-type interpretation of the term 'workflow' and one of the RUP interpretation of the term 'workflow'.

The first example will show a model of an ISO work practice, which is the lowest level of the ISO process structure and which is discussed in-depth in Chapter 6.

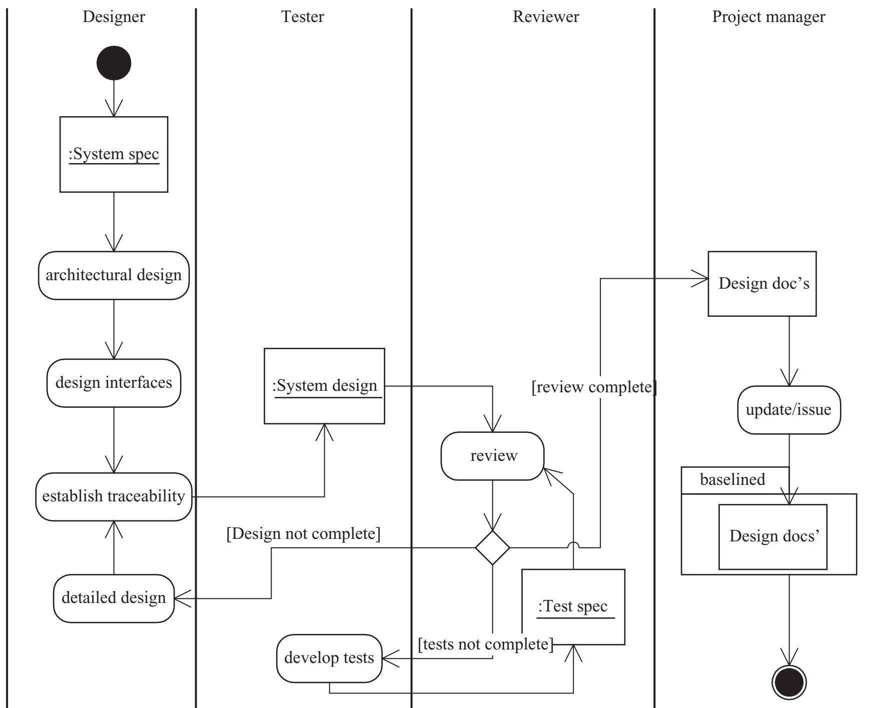


*Figure 5.63 Activity diagram for an ISO work practice*

Figure 5.63 shows how a design process behaves and is taken from Chapter 7. Note how this diagram uses partitions as swim lanes to show responsibility in the process. Also, note the cunning use of a package to indicate where one of the objects has been baselined.

The second example of modelling workflows is one that is taken from the definition of the RUP, which can be found in almost every UML textbook in existence

today. For a more in-depth discussion of this concept of workflows, see References 2 and 16. For now, however, it is enough to know that the activity diagram is recommended by the RUP to be the diagram that is used to model workflows.

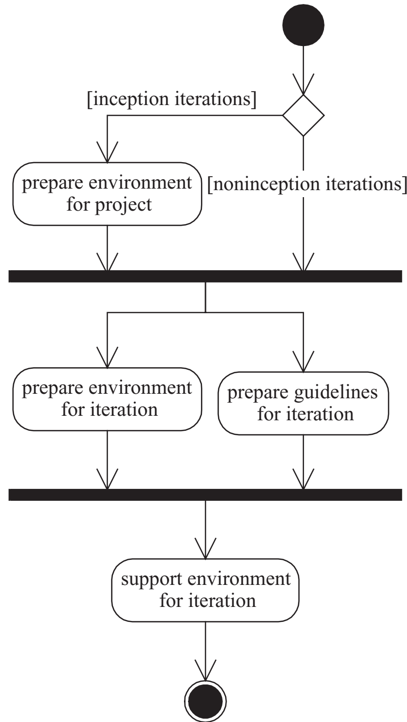


*Figure 5.64 Activity diagram showing workflow from the RUP*

Figure 5.64 shows an activity diagram that describes the behaviour of, or how to execute, a particular workflow. One point worth considering here is the use of the fork and join nodes. There are two activity invocations that are happening in parallel flows, but this does not necessarily indicate that the actual activities are being executed in parallel. What the diagram actually shows is that the flow is split into two flows and that both activities must complete before the flows can be joined. It could be the case that both activities do execute in parallel or it could be that one is executed before the other, or any other permutation. The main point to remember here is that both flows must complete before they can be joined.

### *5.8.4 Using activity diagrams*

Activity diagrams are one of the least used of the UML diagrams, which is reflected in the fact that many tools do not support activity diagrams! Perhaps one of the reasons why they are not used is because they are useful at the algorithm level. Unfortunately,

in many cases and especially for particularly simple algorithms, creating activity diagrams is often perceived as a waste of time. Another reason why activity diagrams are underused is that there is some confusion about the exact nature and structure of the RUP, mainly due to the fact that the RUP is an evolving (yet well-proven) process.

Activity diagrams are actually defined as a special type of state machine diagram, but it is sometimes difficult to see why two types of diagrams are required when, it may be argued, they are very similar indeed. The most fundamental conceptual difference between activity diagrams and state machine diagrams is that activity diagrams concentrate on activity flow and may thus show behaviour from more than one type of object, whereas a state machine only shows the behaviour from a single type of object. As a consequence, state machine diagrams may show wait, or idle, states, whereas an activity diagram may not. The states in a state machine are normal states, which means that they may be complex and contain more than one internal transition, represented by a number of actions or activities, whereas activity invocations may only contain one. Activity diagrams may also use advanced syntax such as 'swim lanes' and 'object flow', which will be discussed briefly in Chapter 6. Swim lanes allow particular states to be grouped together and associated with a particular class or object, which is useful for assigning responsibility. Object flows allow the creation and destruction of objects or classes to be associated with states, which may be used to show data flow.

## 5.9 Use case diagrams

### *5.9.1 Overview*

This section introduces use case diagrams, which realise a behavioural aspect of the model. The behavioural view has an emphasis on functionality, rather than the control and logical timing of the system. The use case diagram represents the highest level of abstraction of a view that is available in the UML and it is used, primarily, to model requirements and contexts of a system. Use cases are covered in greater depth in Chapter 7 and thus this section is kept deliberately short, emphasising, as it does, the structure of the diagrams.

Use case diagrams are arguably the easiest diagram to get wrong in the UML. There are a number of reasons for this:

- The diagrams themselves look very simple, so simple in fact that they are often viewed as being a waste of time.
- It is very easy to go into too much detail on a use case model and to accidentally start analysis or design, rather than very high-level requirements and context modelling.
- Use case diagrams are very easy to confuse with data flow diagrams as they are often perceived as being similar. This is because the symbols look the same since both use cases (in use case diagrams) and processes (in a data flow diagram) are represented by ellipses. In addition, both use cases and processes can be decomposed into lower-level elements.

- Information on the practical use of use cases is surprisingly sparse, bearing in mind that many approaches that are advocated using the UML are use case driven. In addition, much of the literature concerning use case diagrams is either incorrect or has major parts of the diagrams missing!

With these points in mind, the remainder of this section will describe the mechanics of use case diagrams. For an in-depth discussion concerning the actual use of use case diagrams, see Chapter 6.

### *5.9.2 Diagram elements*

Use case diagrams are composed of four basic elements: use cases, actors, associations and a system boundary. Each use case describes a requirement that exists in the system. These requirements may be related to other requirements or actors using associations. An association describes a very high-level relationship between two diagram elements that represents some sort of communication between them.

An actor represents the role of somebody or something that somehow interacts with the system.

The system boundary is used when describing the context of a system. Many texts and tools ignore the use of the system boundary or say that it is optional without really explaining why. Think of the system boundary as the context of the system and its use becomes quite clear and very useful. System boundaries and contexts are discussed in more detail in Chapter 6.

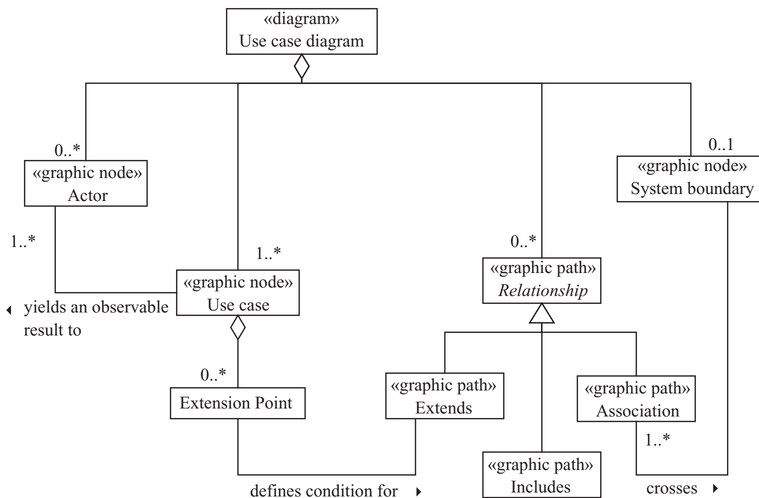


*Figure 5.65 Partial meta-model for use case diagrams*

Figure 5.65 shows that a 'Use case diagram' is made up of zero or more 'Actor', one or more 'Use case', one or more 'Relationship' and zero or one 'System boundary'. A 'Use case' yields an observable result to an 'Actor'.

There are three basic types of 'Relationship' which are 'Extends', 'Includes' and 'Association'. An 'Association' crosses a 'System boundary' and, wherever this happens, it means that an interface must exist. The 'Extends' relationship allows atypical characteristics of a use case to be defined via an 'Extension point' that will actually define these conditions explicitly.

Each of these diagram elements may be realised by either graphical nodes or graphical paths, as indicated by their stereotypes and are illustrated in Figure 5.66.

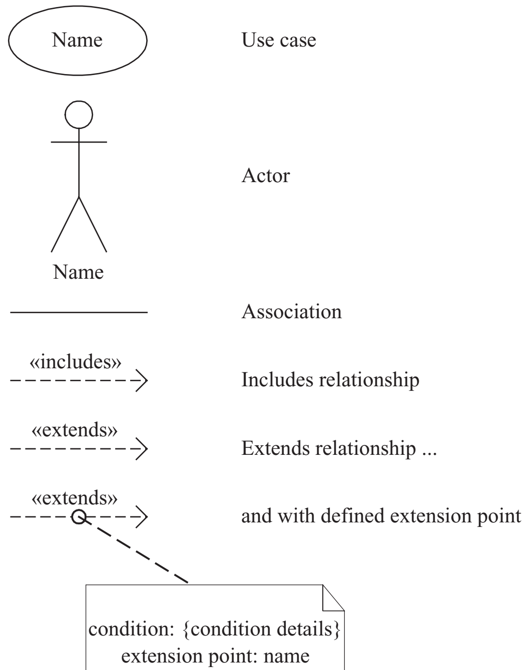


*Figure 5.66 Graphical representation of elements of a use case diagram*

One of the problems that many people have with using use case diagrams is how to tie them into the rest of the system model. It is not uncommon to be presented with a beautiful set of use case diagrams and then to be presented with a wonderful system model with no relationships whatsoever between the two! The diagram in Figure 5.67, therefore, seeks to define this more clearly.

The diagram in Figure 5.67 shows the relationship between use case diagram elements and the rest of the UML. Perhaps the most important relationship here is the one between 'Interaction diagram' and 'Use case' that states that one or more 'Interaction diagram' describes an instance of a 'Use case'. Each use case will have a number of interaction diagrams associated with it or, to put it another way, each requirement will have a number of scenarios associated with it.

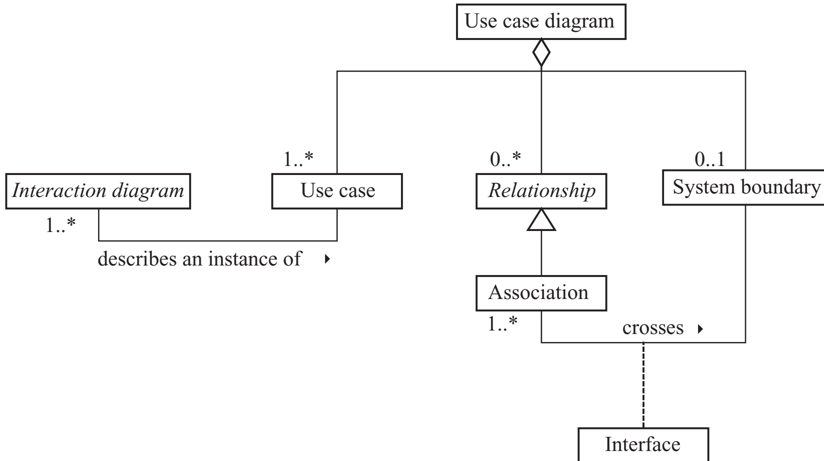


*Figure 5.67 Meta-model showing relationships between use cases and other UML elements*

On a related note, each time an 'Association' crosses the 'System boundary' then an interface has been defined. This becomes important as each interface will become the basis for message passing between different life lines in the interactions.

Once the interaction relationships have been established, it is then possible to navigate to any other diagram in the UML and, hence, any other part of the system model.

### *5.9.3 Examples and modelling*

Use case diagrams have two main uses: modelling contexts and modelling requirements. Although these two types of use should be related, they are distinctly different. For a full description of using use case diagrams, see Chapter 7. This section will concentrate purely on the mechanics and practical use of use case diagrams.

In addition, there are two types of context that can be modelled with regard to systems engineering: the system context and the business context. The first two example models show a business context and a system context for the ongoing chess game example.

Figure 5.68 shows the business context for the chess game. The business context shows the business requirements of the organisation and identifies actors (see the discussion on stakeholders in Chapter 7) that are associated with the business requirements. The business requirements that have been identified are as follows:

- 'make money', which is a business requirement that will be present on almost all business contexts in the world. This may seem obvious, but when asking for funding for a project the response from management is, invariably, 'make

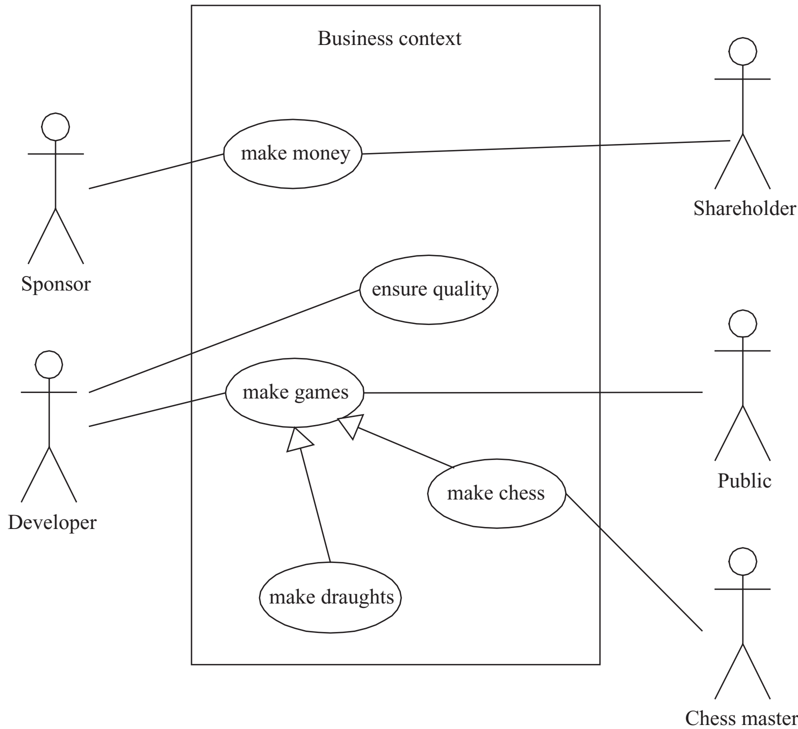


*Figure 5.68 Use case diagram showing the business context for the chess game*

a business case and then we'll look at it'. By identifying the business requirements of your organisation, it is possible to start justifying expenditure for future projects.

- 'ensure quality' is a nonfunctional requirement that will impact on everything that the organisation does. This is particularly true when organisations are trying to obtain, or have already obtained, some form of standard accreditation.
- 'make games' is basically what the organisation does, which has two subtypes: 'make chess' and 'make draughts'.
- 'make chess' is the main business requirement that the system used so far in this book has been concerned with. We can now see, however, how the requirement 'make chess' will help the organisation 'make money', which is very important.
- 'make draughts' is shown simply to indicate that the company may make more than one type of game.

The actors that have been identified for the system must each have some sort of association with at least one use case. If an actor exists that does not have an association with a use case, there is something seriously wrong with the model.

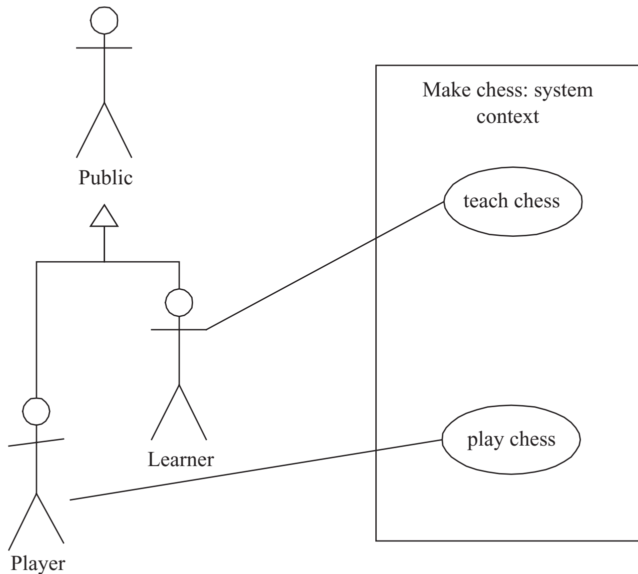


*Figure 5.69 Use case diagram showing system context for the chess game*

Figure 5.69 shows the system context, rather than the business context. It is important that these two contexts can be related in order to justify why a particular project should exist.

Figure 5.69 shows the system context for the chess game. The use cases that have been identified are, at this level, very simple and very high level. The use case 'play chess' represents the main use of the chess system that is being developed. The second use case, 'teach chess', shows the secondary function of the chess system, which is to teach a 'Learner' how to play chess. One argument that is often levelled at use case diagrams is that they are too simple and fail to add anything to a project. However, no matter how simple, a good use case diagram can contribute to a project in several ways:

- They can be traced back to a business context to help justify a project. Quite often it is the case that a simple one-sentence description is required to sum up a whole project and a good system context will show exactly that.
- They can give a very simple overview of exactly what the project is intended to achieve and for whom.
- They can form the basis of lower-level, decomposed use case diagrams that add more detail.

It is also worth remembering that just because a use case diagram looks simple, it does not mean that it took somebody two minutes to create and did not involve much effort. This is only true when it comes to realising the model using a CASE tool, but even the most simple of diagrams may take many hours, days or even months of work.

Remember that what is being visualised here is the result of requirements capture, rather than the actual work itself.

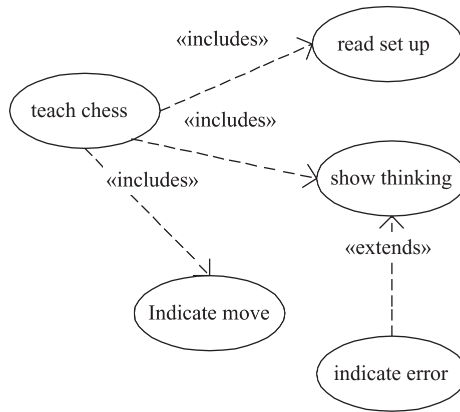


*Figure 5.70 Use case diagram showing modelling requirements*

Figure 5.70 shows how a single requirement may be taken and modelled in more detail, by decomposing the requirement into lower-level requirements. It shows the ‘teach chess’ requirement that has been decomposed into three lower-level requirements: ‘read set up’, ‘indicate move’ and ‘show thinking’. Each of these requirements is a component requirement of the higher-level requirement that is indicated by the special type of association ‘«includes»’, which shows an aggregation-style relationship.

The second special type of association is also shown here and is represented by the stereotype ‘«extends»’. The ‘«extends»’ relationship implies that the associated use case somehow changes the functionality of what goes on inside the higher-level use case. In the example here, the extending use case is ‘indicate error’, that extends the functionality of ‘indicate move’ in the event that something untoward happens and forces ‘indicate move’ to behave in a different way.

Figure 5.71 shows what a use case diagram should not look like, which is a mistake that is very easy to make.

The model in Figure 5.71 shows how a model should not be created as it defies the objectives of use case diagrams. There are several points to consider here:

- The definition of a use case is that it must yield some observable result to an actor. In the example shown here, the decomposed use cases do not yield any observable result to the actors – they are modelled at too low a level.
- Remember that the aim of modelling requirements is to state a user’s needs at a high level and not to constrain the user with proposed solutions. The example shown here has done exactly that and has started down the design road – which is inappropriate at this level.

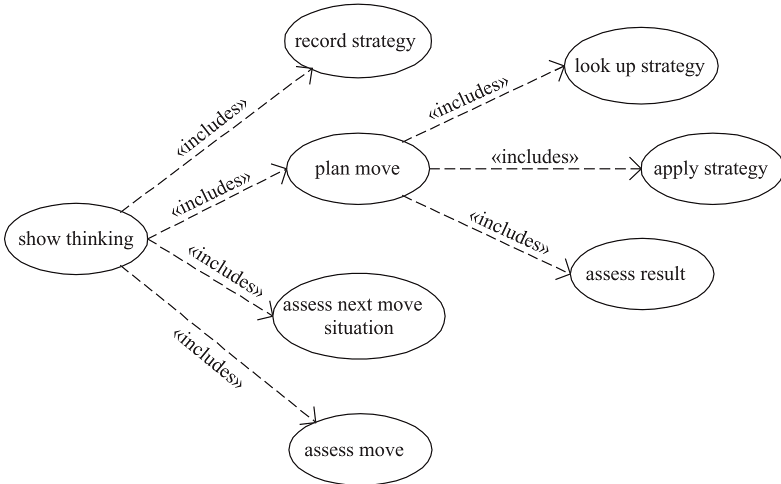


*Figure 5.71 Use case diagram showing a poor example of modelling*

- The diagram is turning into a dataflow diagram by starting at the highest level and decomposing until the problem is solved. Use cases are not the same as dataflow diagrams, despite appearances, and should not be used as such.

These points are discussed in more detail in Chapter 7.

### *5.9.4 Using use case diagrams*

Use case diagrams are, visually, one of the simplest diagrams in the UML, but they are also the most difficult to get right. Part of their downfall is this simplicity, which makes them easy to dismiss and easy to get wrong. Another problem with use case diagrams is that they look like dataflow diagrams and thus people tend to overdecompose them and start carrying out analysis and design rather than just simple requirements modelling.

One of the strengths of use case diagrams only comes out when they are compared with contexts. It is important to be able to trace any requirement back to the business context to justify a business case for the project. It is also important to be able to trace it to the system context as this provides the starting point for the project.

The meta-model shown here represents all of the basic UML elements of use case diagrams, which indicates the simplicity of the diagram.

## 5.10 Component diagrams

### *5.10.1 Overview*

This section introduces component diagrams that realise a structural view of a system. Component diagrams are often perceived as being used, in the main, at later stages of

system development when it comes to packaging up the final product and delivering to the customer. However, they are also very useful for modelling legacy systems which will usually happen towards the beginning of a project. Component diagrams show real-life aspects of a system and have strong relationships to both class and object diagrams.

The simplest way to summarise why a component diagram is used is to imagine buying any product that is packaged in a box. The component diagram, basically, shows 'what's in the box' using both components and artefacts.

### *5.10.2 Diagram elements*

Component diagrams in UML 2.0 are far more powerful (and useful) than those presented in UML 1.x. Several new elements are introduced and the definition of a component is tightened up to provide a far more precise definition of what a component actually is. The elements that make up a component diagram are shown in Figure 5.72.

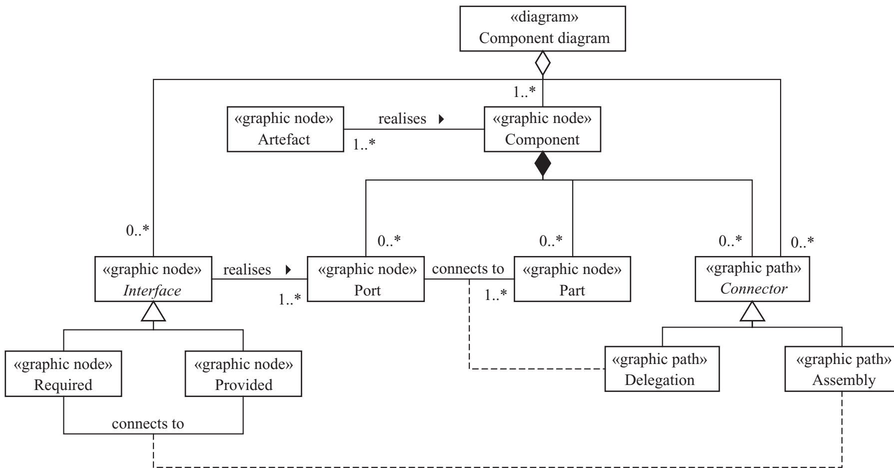


*Figure 5.72 Partial meta-model for component diagrams*

The diagram in Figure 5.72 shows the partial meta-model for component diagrams. There are three main elements that make up a component diagram that are: one or more 'Component', zero or more 'Interface' and zero or more 'Artefact'. Each of these is discussed in more detail since the meaning of components has changed compared to the one provided in UML 1.x:

- A component is the 'specification' of a replaceable module in a system that is defined in terms of its interfaces. A component, therefore, is not the actual physical entity that exists in real life, but the specification of that entity.
- An artefact is the actual physical entity existing in real life that meets the specification defined by the component.

- An interface describes the set of services provided by and required by a particular component.

From the diagram, it can be seen that a ‘Component’ is made up of zero or more ‘Port’, zero or more ‘Part’ and zero or more ‘Connector’. There are two types of ‘Connector’ in a component diagram, an ‘Assembly’ connecting required and provided interfaces, and a ‘Delegation’ connecting parts and ports.

All these elements may be realised by graphic paths or nodes using the symbols shown in Figure 5.73.

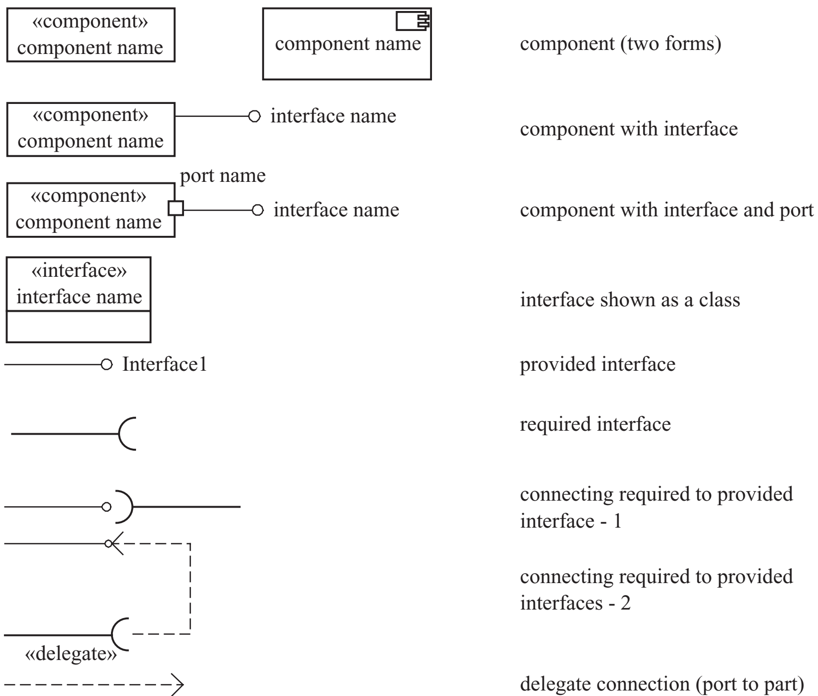


*Figure 5.73 Graphical representation of the elements in a component diagram*

The diagram in Figure 5.73 shows the graphical notation for the elements in a component diagram. Perhaps the most awkward symbol to draw in UML 1.x is the component symbol which is a rectangle with two smaller rectangles on the side of it. Not only is it awkward to draw, but the two smaller rectangles often take up space on the diagram that would be better served showing interfaces. In a very wise move, this symbol has been changed to the far more drawing-friendly rectangle with a simple stereotype of «component» shown at the top of the box. Alternatively, it is also possible to use the new graphical notation and then to draw a tiny version of the old symbol in one corner, which defeats the object of replacing the old symbol!

Artefacts are shown in the same way – a rectangle with a stereotype of «artefact» shown in the top of the box.

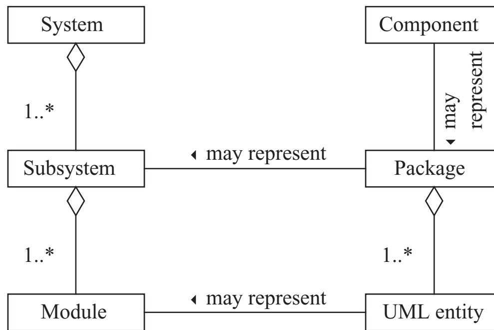


*Figure 5.74 Systems and components*

Figure 5.74 shows how components relate to a system. Any generic system may be thought of as being made up of one or more subsystems, each of which is made up of one or more modules. This may be related to UML constructs, such as a 'Package' being made up of one or more 'UML entity', as seen in Chapter 3. A 'UML entity' may represent a 'Module' and a 'Package' may represent a 'Subsystem'. Think of the 'Component' as a formal representation of a 'Package'. By tracing this relationship one step further, it tells us that a 'Component' is strongly related to a collection of UML entities, or a subsystem.

### *5.10.3 Examples and modelling*

One of the main uses for component diagrams is to show components and their interfaces. In UML 1.x, interfaces were drastically underused which was due, in the main, to the fact that they were too simple and lacked much functionality. With UML 2.0, all this has changed. Interfaces are far better defined (two types of interface), the concept of the port has been introduced and the internal working of a component can now also be shown. In order to illustrate these new concepts, consider the diagram in Figure 5.75.

The diagram in Figure 5.75 shows a component diagram for a computer games console. Notice that there are a lot of ports and interfaces shown here. It can be seen that there are two main components shown in the diagram: 'Game engine' that happens to be a component and 'game disc' that also happens to be a component. The small rectangles shown on the edges of the component represent the ports – notice that the naming of ports is optional, whereas it is usual to name all interfaces. Starting at the top of the component, it can be seen that there are two ports named 'Memory card 1' and 'Memory card 2' that share a common interface, named 'Memory card'. This is a good example of several ports sharing the same interface – there is no point

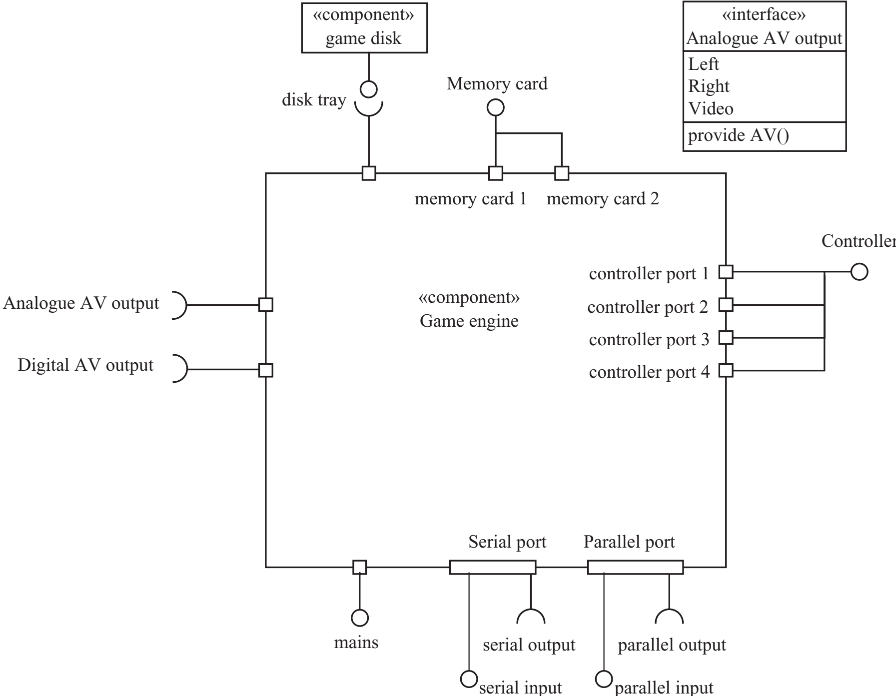


*Figure 5.75 Example of component and interfaces*

in defining an interface more than once, so a single definition of an interface is used twice. The same logic can be applied to the four controller points shown on the right-hand edge of the component. Note that these interfaces are provided interfaces into which must be plugged another component – in this case game controllers. It may be thought of as a provided interface allowing input to, whilst a required interface provides output from a component.

Moving around the component, the bottom edge of the component shows three ports, two of which have more than one interface. This is similar to the concept discussed previously concerning a single interface having several ports. In this case, however, there is a single port that has two distinctly different interfacing elements – one of input and one of output.

Therefore, the relationship between ports and interface can be one to one, many to one, one to many or many to many.

The diagram also shows more detail concerning an interface – in this case the ‘Analogue AV output’ – which is represented by a class that is stereotyped to show that it happens to be an interface. When defining an interface in this way, it is usual to use the attributes of the class to represent any data and the operations to represent any functionality. Although not shown here, an interface class should be defined for each of the interfaces identified on the diagram.

The component shown in this diagram represents the specification of the games console component, rather than the implementation of it. To understand this, consider the scenario where a console is produced by a particular manufacturer, but then is licensed out to third-party suppliers. For example, consider that the console manufacturer produces only the games console, whereas a third-party vendor may wish to integrate the games console with a DVD player and video system and sell it as a single package. The same will apply to any components that are plugged into the games console – every games console on the market has open specifications for its peripherals so that third-party suppliers can manufacture their own game controllers and memory cards and the only way to ensure that they will work is to have a common specification for the interfaces.

The actual implementation of a component is represented by an '«artefact»'.

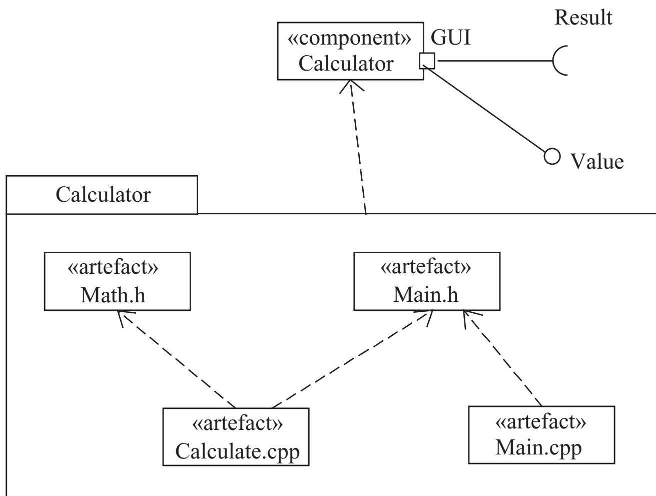


*Figure 5.76 Component diagram showing the use of artefacts*

The diagram in Figure 5.76 shows an example of several artefacts that are used to realise a component. It can be seen that there is a component called 'Calculator', that has a single port, named 'GUI' (graphical user interface) and two interfaces – a provided interface named 'Value' and a required interface named 'Result'. This component is realised (indicated by the dependency) by a package named 'Calculator' that contains four artefacts – represented by stereotyped classes again that will implement the component. It is simple to imagine, in the case of software, a situation where the same component may be realised by a number of different languages, hence each having its own artefact.

Another issue to consider is actually how these components are put together – this now gives some indication as to why the circle and semicircle notation was chosen for the interfaces as it is now possible to connect these interfaces to configure a system. An example of this is shown in Figure 5.77.

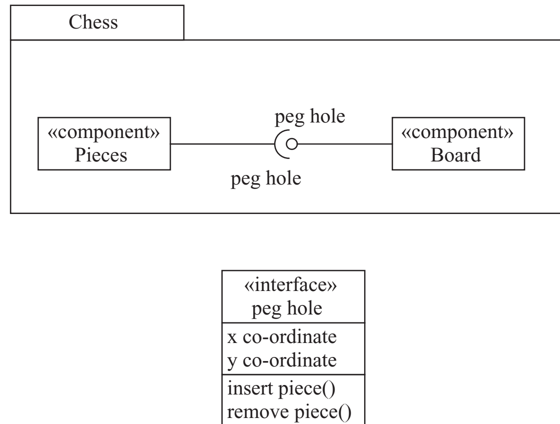


*Figure 5.77 Component diagram showing how to model a physical system*

The model in Figure 5.77 shows two components – ‘Board’ and ‘Pieces’, each of which has a single interface named ‘peg hole’. Note that only interfaces with the same name and, hence, same interface specification, can be connected and only then if they are complementary types (required and provided). The definition of the interface is also shown here as an interface class.

In some diagrams, it may not be possible to show the interfaces joined up like in this diagram, or it may just make the diagram unreadable. In this sort of situation, it is possible to show a dependency relationship between two interfaces that implies that they are connected together.

## *5.10.4 Using component diagrams*

Component diagrams are used to show ‘what’s in the box’ when dealing with a real system. The components represent the specifications of removable modules, whereas artefacts show the actual realisations of these interfaces.

Component diagrams are often hailed as being always used in later life cycle phases, towards the end of a project when delivering the final system. This is certainly true when talking about clean-sheet projects but is not true when dealing with legacy systems. When dealing with legacy systems, it is often the case that the component diagrams for legacy aspects of the system are modelled at the beginning of a project, maybe as part of a requirements or analysis exercise.

Component diagrams in UML 2.0 have changed significantly from UML 1.x and now have far more functionality, are easier to draw and are far more powerful.

## 5.11 Deployment diagrams

### *5.11.1 Overview*

This section introduces deployment diagrams, which realise a structural aspect of the model of the system. Like component diagrams, deployment diagrams are often hailed as being used towards the end of a project life cycle when the system has been packaged up, although this is often not the case. If a component diagram shows 'what's in the box', it is the deployment diagram that shows 'where to put the pieces'.

### *5.11.2 Diagram elements*

Deployment diagrams are made up of two main elements: nodes and relationships. Nodes represent real-world aspects of a system and show where artefacts from the component diagram are housed. Nodes are represented graphically by a 3-D box, on which instances or artefacts are shown.

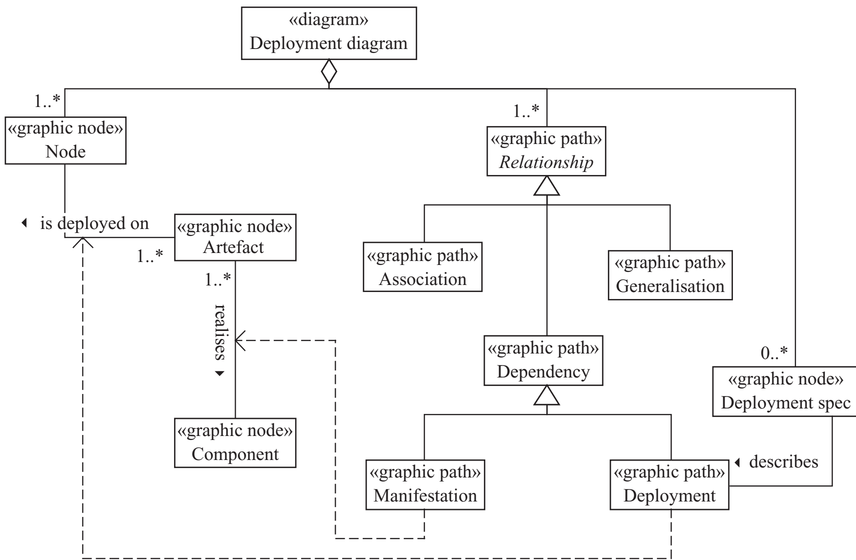


*Figure 5.78 Partial meta-model for deployment diagrams*

Figure 5.78 shows the partial meta-model for deployment diagrams in the UML. It can be seen from the diagram that a 'Deployment diagram' is made up of one or more 'Node', one or more 'Relationship' and zero or more 'Deployment spec'. There are three types of 'relationship': 'Association', 'Generalisation' and 'Dependency'. The 'Association' and 'Generalisation' relationships are the same as for all structural modelling but the 'Dependency' relationship has two specialisations defined – 'Manifestation' and 'Deployment'. The 'Manifestation' relationship describes the

relationship between a component and its associated artefacts so it may be thought of as a type of realisation. The 'Deployment' relationship describes an artefact being deployed onto a particular node. Such a deployment may have a 'Deployment specification' associated with it that describes features of the deployment.

Each of these diagram elements may be realised by either graphical nodes or graphical paths, as indicated by their stereotypes and illustrated in Figure 5.79.

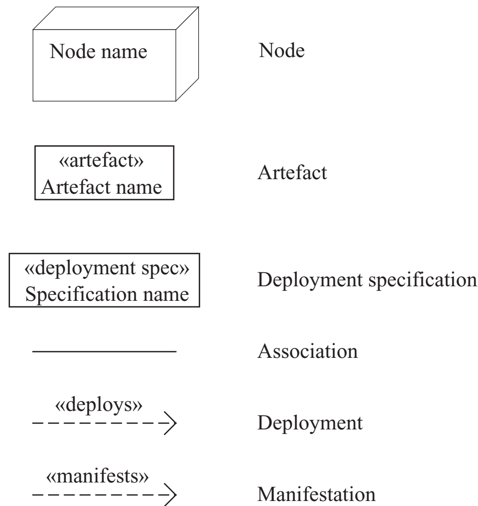


*Figure 5.79 Graphical representation of elements in a deployment diagram*

The graphical notation for a deployment diagram is straightforward, with the only element that has not been seen so far in other diagrams being the node, that is represented by a cube. The concept of a node is quite a strange one and is one of those situations where a concept in software engineering is far simpler than the same concept in systems engineering. When developing software, code is ultimately produced that may either be stored or converted in some way to executable code that may be run on some software or hardware platform. Therefore, there are only three types of node that may exist in the world of software engineering: some sort of storage medium (optical disc, memory, etc.), a hardware platform or some other piece of software. This is quite neat and makes deployment diagrams for software pretty much more straightforward. Consider now, however, the world of systems engineering. In systems engineering, we are talking about deploying actual systems and subsystems and, therefore, more often than not, a node in a systems project will represent some sort of location. This may be a geographical location, such as a country, or may be a location within another system, or whatever. The key point here is that a node in systems engineering has an infinite number of possible permutations for realisation, whereas in the world of software engineering, there are only three.

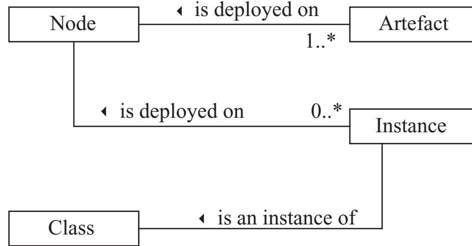


*Figure 5.80 Relationship between deployment diagram elements and the rest of the UML*

People often have a problem seeing how the deployment diagram ties in with the rest of the UML model, which can be seen in Figure 5.80.

The diagram in Figure 5.80 shows the relationship between the node from a deployment diagram and the rest of the UML model. It can be seen that there are two elements that may be deployed on a 'Node' – a number of 'Artefact' or a number of 'Instance'. This then leads to the rest of UML through relationships to other UML elements, such as the 'Instance' to 'Class' relationship shown here.

### *5.11.3 Examples and modelling*

There are several uses for deployment diagrams which include modelling embedded systems, modelling distributed systems, modelling client–server systems and modelling physical systems. An example diagram for each of these uses is discussed in this section.

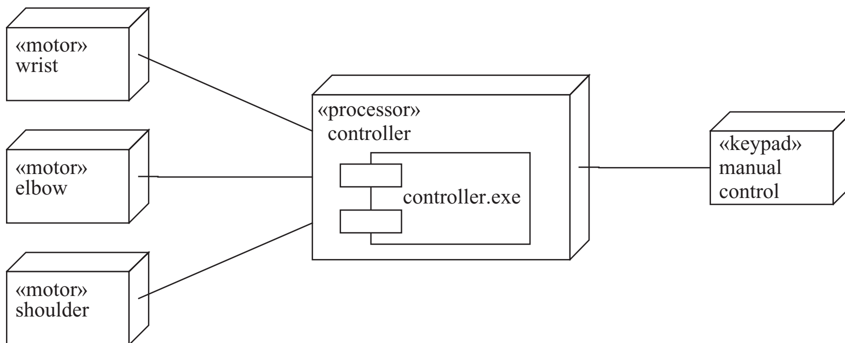


*Figure 5.81 Deployment diagram showing embedded systems*

Figure 5.81 shows an embedded system that represents a robot arm controller system. The actual part of the system that has been the subject of the project is the controller software, which is represented here as the 'controller.exe' component that

lives on the node 'controller' and that happens to be a '«processor»'. Other elements in the diagram include 'shoulder', 'elbow' and 'wrist', each of which happens to be a '«motor»'. The final element is the 'manual control', which happens to be a '«keypad»'.

Note the extensive use of stereotypes, which is typical for deployment diagrams. Stereotypes are discussed in more detail in Chapter 8 and caution must be exercised when using them.

Figure 5.81 shows how an embedded system may be modelled using a deployment diagram. The example chosen here is used as part of the user manual for a website in order to show the customer where the original website files must end up before the website will work and be accessible from the Internet.

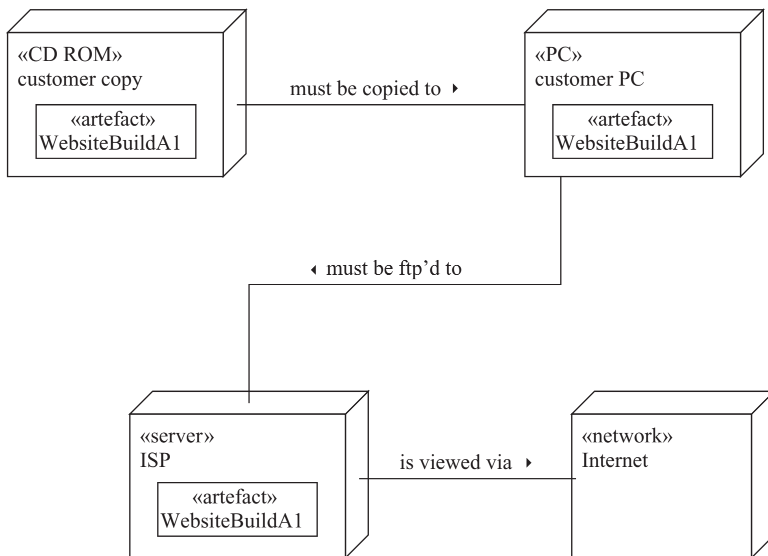


*Figure 5.82 Deployment diagram showing distributed systems*

Figure 5.82 shows a distributed system that has been modelled using a deployment diagram. The node on the left is called 'customer copy', which happens to be a '«CD-ROM»' representing the actual disc that is delivered to the customer. The next node is called 'customer PC', which happens to be a '«PC»'. This represents the customer's computer onto which the information on the CD-ROM must be copied. This is apparent from the diagram since the directory that must be copied across to it is represented by the artefact 'WebsiteBuildA1' and the association between the two nodes describes the nature and direction of the relationship.

The third node, called 'ISP' and which happens to be a '«server»', represents the customer's Internet service provider (ISP) to which the 'WebsiteBuildA1' files must be ftp'd.

Note again the extensive use of stereotypes and remember that this diagram would lose a great deal of meaning without some sort of legend to explain what the stereotypes mean. Such a legend is often called the 'assumption model' and is defined as part of a UML profile, which is discussed in detail in Chapter 8.

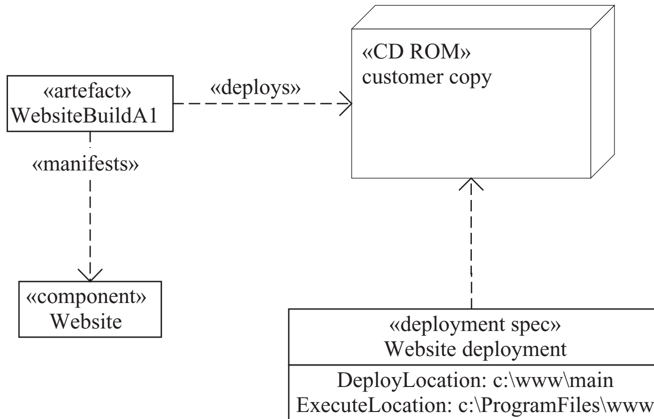


*Figure 5.83 Alternate representation of the chess example, using 'manifests' relationship*

The diagram in Figure 5.83 shows an alternate way of showing the same information. This time, the artefact 'WebsiteBuildA1' is shown outside the node, but the deployment is indicated by the «deploys» dependency. Also shown here, is the actual component that is being realised, that is indicated by the «manifests» dependency. Just to include the rest of the notation, a deployment specification is also shown here that shows the features of the deployment, in this case there are two attributes to represent the deployment location (indicated by the 'DeployLocation' attribute) and the execution location (indicated by the 'ExecuteLocation' location).

The next example of the use of a deployment diagram shows a client–server system that makes use of packages.

The final example of the use of deployment diagrams pays another visit to the game of chess. This time, as we know 'what's in the box' from the component diagram, we need to know where to put the things that are in the box. This may seem like stating the obvious, but it wraps up the chess example neatly and goes to show that it is possible to model just about anything with the UML, including where to sit during a game of chess!

The components that were defined in Figure 5.77 lead to two artefacts being identified: 'pieces' and 'board'. We also know from the object diagram in Figure 5.19 and the interaction diagrams in Figure 5.44 and Figure 5.48 that two instances exist: 'Player 1' and 'Player 2'. The deployment diagram relates all these elements together to show where they are deployed.

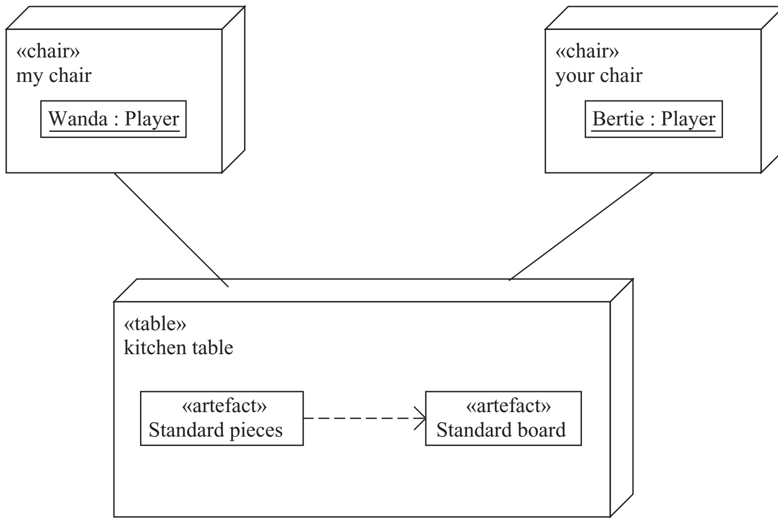


*Figure 5.84 Deployment diagram showing physical systems layout for the chess example*

Figure 5.84 shows how the components and objects defined in a chess game box set and the associated players are deployed in the real physical world.

There are two nodes that happen to be '«chair»', which are 'my chair' and 'your chair', where the instances 'Wanda' and 'Bertie' are deployed, or sit. The 'standard board' and 'standard pieces' are artefacts from a component diagram that are deployed on the 'kitchen table', which happens to be a '«table»'.

### *5.11.4 Using deployment diagrams*

Deployment diagrams have a very limited syntax and thus the meta-model shown here is almost complete. Advanced concepts that may be used for deployment diagrams include different types of nodes to represent processing nodes and nonprocessing nodes in the case of software engineering. This may be extended to the world of systems engineering by defining a set of locations specific to a particular project or system that may be defined as stereotypes and reused.

Deployment diagrams are also one of the more little-used UML diagrams, which is reinforced by the lack of CASE tool support for the diagram.

An interesting point about deployment diagrams is that, in most cases, they are hailed as being used towards the end of a project when the final product is being packaged up; however, in some cases, they are the first diagrams to be used. In the same way that component diagrams are very powerful when it comes to projects with a legacy element, so are deployment diagrams. Also, in examples where the system may have to be installed into a particular environment, deployment diagrams may be used to model that environment.

## 5.12 Summary

This chapter introduces each of the 13 UML diagrams in turn and provides examples of their use. Each diagram is discussed at a very high level, often missing out much of the diagram syntax, so that the modelling aspects of each diagram could be focused on, rather than being bogged down by all the syntax of each diagram.

The following models serve as a recap for everything said in this chapter, which should be readable by anyone who has read (and understood) this book so far. Even with the limited amount of syntax that has been introduced, it is still possible to model many aspects of a system.

Figure 5.85 shows the 13 types of UML diagram that have been introduced in this chapter. Note how the 'Sequence diagram', 'Timing diagram' , 'Interaction overview diagram' and 'Communication diagram' are grouped into a generalised class called 'Interaction diagram', which reflects that fact each of the two diagrams shows the same sort of information. Note also how the 'Activity diagram' is a specialised type of 'State machine diagram'.

These 13 diagrams are often grouped into categories to represent different views of the system. Rather than be prescriptive with these groupings, the diagrams in this book are grouped into two very broad types of model: behavioural and structural.

Figure 5.86 shows how the nine types of UML diagram are grouped into the two types of models. Many other texts on the market will show different 'views' of a system, such as logical view, physical view, use case view, and so on, and this is fine but it will not fit every application. These views are also consistent with the two types of models shown here; each type of view that is defined may consist of a structural and behavioural element. Treat the two models here as the absolute highest levels of types of model and use the views introduced in other books as necessary.

## 5.13 Conclusions

In order to conclude this chapter, there are a few pieces of practical advice that should be borne in mind when modelling using the UML. These rules are not carved in stone, but experience has shown that they will be applicable to 99 per cent of all modelling.

- *Use whatever diagrams are appropriate.* There is nothing to say that all 13 diagrams should be used in order to have a fully defined system – just use whatever diagrams are the most appropriate.
- *Use whatever syntax is appropriate.* The syntax introduced in this book represents only a fraction of the very rich UML language. It is possible to model most aspects of a system using the limited syntax introduced here. As you encounter situations that your known syntax cannot cope with, it is time to learn some more. There is a very good chance that there is a mechanism there somewhere that will.

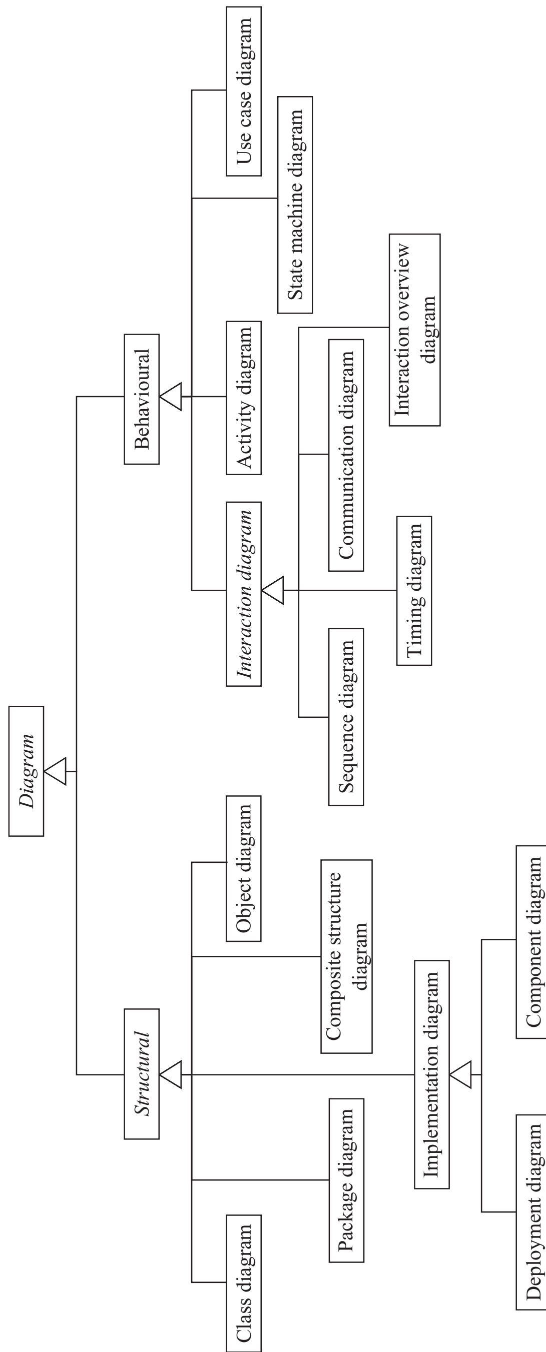


*Figure 5.85 The 13 types of UML diagram*

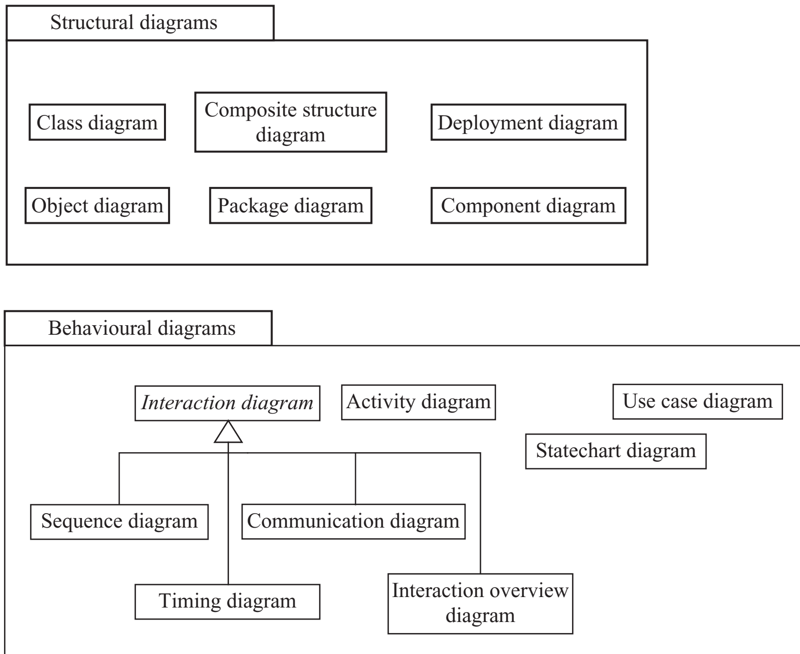


*Figure 5.86 The types of diagram grouped into types of models*

- *Ensure consistency between models.* One of the most powerful aspects of the UML is the ability to check the consistency between diagrams which is often glossed over. Certainly, in order to give a good level of confidence in your models, these consistency checks are essential.
- *Iterate.* Nobody ever gets a model right the first time, so iterate! A model is a living entity that will evolve over time and, as the model becomes more refined, the connection to reality will draw closer.
- *Keep all models.* Never throw away a model, even if it is deemed incorrect, since it will help you to document decisions made as the design evolved.
- *Ensure that the system is modelled in both views.* In order to meet most of these criteria, it is essential that the system is modelled in both views, otherwise the model is, quite simply, incomplete.
- *Ensure that the system is modelled at several levels of abstraction.* This is one of the fundamental aspects of modelling and will help to maintain consistency checks, as demonstrated in the chess example.

Finally, modelling using the UML should not change the way that you work, but should aid communication and help to avoid ambiguities. Model as many things as possible, as often as possible, because the more you use the UML, the more benefits you will discover.

## 5.14 Further discussion

1. Take one of the partial meta-models shown in this chapter and then select any other UML textbook – see the references at the end of this chapter for a list of alternatives. Based on the detailed information in the textbook, populate the meta-model further to gain a greater understanding of the type of diagram chosen for this example.
2. Model a set of chess pieces using a class diagram. How will the different types of pieces be represented – by colour or by movement?
3. Expand on the chess game example by adding a new operation called 'think' to the class diagram introduced in Figure 5.8. Which other diagrams will this affect?
4. Expand on the chess example in Figure 5.8 by defining two types of 'Player': 'Human' and 'Computer'. What impact will this have on the model and which models will need to be redefined or created from scratch?
5. Take any prerecorded CD-ROM and model the contents, based on what can be seen, for example, from a typical Windows-style browser.
6. Now model the steps that you took in order to put the CD into the computer and read the contents of the CD-ROM. Think about deployment and component diagrams for the structural view and interaction diagrams and state machine diagrams to model how this is achieved.

## 5.15 References


1 RUMBAUGH, J., JACOBSON, I., and BOOCH, G.: 'The unified modelling language reference manual' (Addison Wesley, Massachusetts, 1998)
2 BOOCH, G., RUMBAUGH, J., and JACOBSON, I.: 'The unified modelling language user guide' (Addison Wesley, Massachusetts, 1998)
3 JACOBSON, I., BOOCH, G., and RUMBAUGH, J.:, 'The unified software development process' (Addison Wesley, Massachusetts, 1999)
4 DOUGLASS, B. P.: 'Real-time UML' (Addison Wesley, Massachusetts, 1998)
5 FOWLER, M., and SCOTT, K.: 'UML distilled (applying the standard object modelling language)' (Addison Wesley, Boston, 2003, 3rd edn.)
6 LEE, R. C., and TEPFENHART, W. M.: 'UML and C++, a practical guide to object-oriented development' (Prentice Hall, New Jersey, 1997)
7 LARMAN, G.: 'Applying UML and patterns – an introduction to object-oriented analysis and design' (Prentice Hall Inc., New Jersey, 1998)
8 ROYCE, W.: 'Software project management – a unified framework' (Addison Wesley, Massachusetts, 1998)
9 CANTOR, M. R.: 'Object-oriented project management with UML' (John Wiley and Sons Inc., New York, 1998)
10 SI ALHIR, S.: 'UML in a nutshell – a desktop quick reference' (O'Reilly and Associates Inc., Cambridge, 1998)
11 ERIKSSON, H., and PENKER, M.: 'UML toolkit' (John Wiley and Sons Inc., New York, 2003)

12 MULLER, P.: ‘Instant UML’ (Wrox Press Ltd, Paris, 1997)
13 STEVENS, P., and POOLEY, R.: ‘Using UML’ (Addison Wesley, Harrow, 1999)
14 SCHMULLER, J.: ‘SAMS teach yourself UML in 24 hours’ (SAMS, Indiana, 1999)
15 HOYLE, D.: ‘ISO 9000, pocket guide’ (Butterworth Heinemann, Oxford, 1998)
16 KRUCHTEN, P.: ‘The rational unified process: An introduction’ (Addison-Wesley Publishing, Massachusetts, 1999)
17 RADER, J. R.: ‘Advanced software design techniques’ (Petrocelli Books, London, 1978)

*Chapter 6*

# Modelling standards, processes and procedures

process and procedure are the last hiding place of people without the wit and wisdom to do their job properly

*David Brent*

## 6.1 Introduction

### *6.1.1 Introduction*

In this chapter, standards, processes and procedures are modelled using the Unified Modelling Language (UML). The importance of having a defined process has been discussed previously in Chapter I. This chapter looks at how the UML may be used to effectively model processes in a correct and unambiguous fashion in order to minimise complexity of the process and to maximise the effective communication involved with implementing the process.

This section is split into six main sections:

- Introduction, where the rationale for modelling standards, processes and procedures is discussed.
- Analysing standards, where the UML is used to model and help to understand standards.
- Defining the procedure, where the analysis models from the previous sections are used as a reference for defining a new procedure.
- Life cycles, where life cycles and life cycle models will be modelled and then related to the new process defined in the previous section. In addition, the process will be tailored to allow for projects with unique requirements.
- Implementing the process, where the structures of the output from the newly defined process are modelled. Practical issues for implementing the procedure will be addressed, which will form the basis for using a systems engineering tool.
- Finally, conclusions are drawn about the whole issue of modelling standards.

For other examples of process modelling, see Reference 1.

## 6.2 Standards

The importance of having a well-defined process was discussed in Chapter 1. However, it is not good enough simply to have a defined process, as it is also necessary to demonstrate compliance with a norm, demonstrate usage of the process on projects and demonstrate that the process is effective. It is possible to use the UML in order to achieve these aims, and this chapter sets out to do exactly that.

The first of the points raised here, demonstrating that the process is compliant with an existing norm, is crucial to allow the process to be audited or assessed according to such a norm. In real terms, these norms are standards, procedures, best practice guides, textbooks [2], etc.

This may seem perfectly reasonable and might fit in with our definition of systems engineering being the implementation of common sense, but it is far easier said than done. In order to demonstrate compliance with a standard it is important to have a good understanding of that standard. In addition, it is necessary to be able to map between the relevant part of the chosen standard and the process under assessment.

### *6.2.1 Standards, standards, standards*

Unfortunately, there are many problems associated with standards that hinder their understanding, or lead to incorrect implementation of standards, and that were introduced briefly previously in this book.

One problem with standards is that there are so many of them – in fact, in some ways there are too many standards, which may lead to all sorts of problems. Here is a list of some of the more popular standards in use today in the world of systems engineering:

*EIA 632 Processes for engineering a system*: This is a popular standard that has seen a lot of implementation in the defence and aerospace industries.

*EIA 731 Systems engineering capability model*: This is a widely adopted standard that is concerned with capability determination and uses the processes defined in EIA 632 Processes for engineering a system.

*ISO 9001 Model for quality assurance in design, development, production, installation and servicing*: This is arguably the most widely used standard in the world as it provides the international benchmark for system quality. This includes all aspects of systems except for software-based systems! This has led to the development of a set of guidelines specifically for software systems, which is described in the next point.

*ISO 9000-3 Guidelines for the application of 9001 to the development supply and maintenance of software*: This is not a standard but a set of guidelines that describe how to apply ISO 9001 to software systems. This has also led to the definition of formal accreditation guidelines under the TickIt scheme.

*IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems*: This standard is aimed very much at programmable devices, such as PLCs, and is used heavily in the signalling world.

*ISO 15288 Life cycle management – system life cycle processes*: The latest attempt by ISO to make sense of the standards maze that has been generated in conjunction with major industry, which promises a large uptake in many industries [3].

*ISO 12207 Software life cycle processes*: This standard provides an input to many of the other standards mentioned here and is one of the most widely accepted standards. In fact, most standards will make explicit reference to ISO 12207 and cite it as a basis for life cycle definition.

*ISO 15504 Software process improvement and capability determination (SPICE)*: This very long and well-written standard defines a complete set of processes within an organisation that can be used as the basis for process improvement and/or capability determination.

*IEEE 1220 Application and management of the systems engineering process*: This US standard is a widely used international standard that has a strong track record for systems development.

These are just a small selection of standards that exist in the world of systems and do not include any industry-specific standards. Anybody familiar with standards should recognise at least one of those in the list above. Anyone who has ever tried to read and understand any of these standards (or any standard, for that matter) may have encountered some difficulty in understanding them, which may be due to a number of reasons:

- Standards are often very difficult to understand individually and, it may be argued, some are not terribly well written. As will be demonstrated later in this chapter, some standards are inconsistent with themselves and contain many errors and much ambiguity. This does not help when trying to read and understand standards (a lack of understanding), which puts us on the rocky road to complexity and communications problems. If a standard is not fully understood, other processes may be affected, such as training and assurance, and so on.
- If standards are difficult to understand individually, they are almost impossible to understand when they are read in groups. It is often the case that an organisation wants to comply with a generic standard, such as ISO 9001, but also wants to comply with a more industry-specific standard (e.g. EN 50128 Railway applications, software for railway control and protection systems) or even a more field-specific standard (e.g. ISO 15288 Life cycle management – system life cycle processes).
- Understanding the information may also be hindered by the fact that some standards are too long (some in excess of 300 pages), which can seem insurmountable when first encountered. Length is not necessarily an indication of complexity but in many cases will lead to such. Indeed, some of the longer standards are actually less complex than those with a far lower page count. For example, ISO 15504 is far less open to ambiguity that ISO 9001, despite having a page count that is over ten times higher.
- Some standards are too short and written at such a high level that they bear little or no resemblance to what actually occurs in real life. Take, for example, ISO 9001, which, it can be argued, is the most widely used standard in the world today. It is

perhaps because it is applicable to so many different areas that the standard is so generic. However, there is an inherent danger in this, as the more generic the standard, the more open to ambiguity it becomes. These ambiguities occur here as two different people reading the same paragraph of the standard may come up with two completely different interpretations.
- Standards are often written by committees and thus tend to end up somewhat less than comprehensible. They frequently take years to prepare and trying to keep all of the people happy all of the time can often lead to the production of a camel rather than a horse!

Assuming then, for one moment, that the standards can be understood, the next issue is that of demonstrating compliance. Compliance is demonstrated through audits or assessments. Audits are performed by an external, registered organisation and yield a straight pass or fail result. Assessments, on the other hand, may be carried out internally or externally, but usually yield more useful results, such as a capability profile. As a simple example of this, ISO 9001 requires a formal audit, which yields a 'yes or no' outcome. ISO 15504, on the other hand, requires an assessment (as opposed to an audit) that yields a potentially more useful capability profile that can be used to improve existing processes.

Compliance is actually achieved through having a defined process that is then the subject of an audit or assessment. In almost all cases, standards explicitly state that a defined process is essential to demonstrate compliance.

Most audits and assessments will call for some sort of mapping to be established between the process and the standard that is being used as a basis for the assessment or audits. In order to map between two of anything, it is important that they are both represented in a language or format that may be directly compared – otherwise mapping is impossible.

This then means that if any of these standards are to be met, they must be capable of being related to a defined process. In addition, if more than a single standard is read, they must be comparable and should be contrasted so that a common understanding may be achieved.

### *6.2.2 Requirements for using standards*

In summary, therefore, it is possible to draw up a set of requirements that are needed in order to use standards in a practical and effective manner:

- There is a need to be able to understand standards. This means that any problem areas such as ambiguities or inconsistencies can be sorted out and that everyone who reads the standard has a common understanding.
- It is vital to be able to compare standards with a defined process. It is no use at all if both the standard and the process can be understood individually, if they cannot be compared in an effective manner.
- It is also necessary to be able to compare different standards with one another. Again, standards often refer to other standards, or are required to be met in conjunction with some other standard.

It is essential, therefore, that all standards and processes can be represented in a common format or language. The UML represents such a common language and, as will be demonstrated, is the most effective at modelling processes and standards. This all boils down to being able to analyse effectively any standard or process by modelling them using the UML.

## 6.3 Analysing standards

Analysing may be thought of as understanding the problem at hand. In this case, the problem at hand is the actual standard, or standards, that need to be modelled.

Analysis may be performed practically by modelling information contained within a standard, or to put it another way, to model the actual content of a standard using the UML. Once models have been created in the UML, they may be compared and contrasted in order to gain a greater understanding. It is then possible to use these models as a basis for modelling a new standard that may be completely modelled and designed before it is actually written.

Another benefit of modelling is that potential areas of complexity may be highlighted and either eliminated or addressed in some other way. All these points will be demonstrated in this chapter.

### *6.3.1 Aspects of modelling using the UML*

It has been stated many times that the UML may be used to model standards, but which of the 13 diagrams that are available are suitable and what aspects of the standards may be modelled? The simple answer to this question is to use whatever diagrams seem appropriate, but this answer does not help us much in a practical way. Therefore, some UML diagrams and their usage are suggested here, but these are by no means carved in stone:

- Class diagrams, which form the backbone of the UML, may be used to model almost all structural aspects of a standard or process. This includes modelling the initial high-level views of a standard that define the way in which the whole process will be laid out. Class diagrams may also be used to show the types of process, task, activity, or whatever nomenclature is being adopted. This may also be taken a step further by modelling actual deliverables associated with particular processes. This will not only show the deliverable and which process it may be associated with, but will also show the actual structure of the deliverable, which may be used as the basis for a template. Implementing a process may also be made easier by defining an information model and implementation model that can then be the basis for implementation using an automated tool.
- The behavioural aspects of processes – that is to say in what order things must be done and under what conditions – may be shown by using statecharts or activity diagrams that are associated with classes that represent processes. This can be particularly useful from a project management point of view and also for training staff in how they should be carrying out a particular process.

- At a higher level, interaction diagrams may be used to model life cycle models that show both how the project is to be carried out (using communication diagrams) and how the project is actually carried out (using sequence diagrams).
- At a very low level, activity diagrams may be used to state explicitly how particular process steps are to be executed for a very rigidly defined process.

All of these techniques have been shown to be very effective for process modelling for both analysis and design of processes. The first step that is covered here, however, is how to analyse two standards.

### *6.3.2 Modelling example*

Imagine a situation where it was required to produce a new process for an organisation that needed to meet a particular systems engineering standard and also generic ISO processes. In the example here, the systems engineering standard that has been chosen is EIA 632 – Processes for engineering a system, and the ISO processes are to be based on those defined in ISO 15504 – Software process improvement and capability determination (SPICE).

This is not such an unusual combination as it may first appear, as ISO 15504 is a very well-specified standard and is fully compliant with ISO 9001. One of the advantages of ISO 15504 over ISO 9001, however, is that it is far more rigorously defined and is thus less open to ambiguity than ISO 9001. Showing any preworked example in a book can have its own danger. The danger with using such an example, however, is that it can appear to be like the work of a magician, where words and diagrams may be pulled, like rabbits, from a hat, yet nothing much will be said or learned – but it certainly looks good! To avoid such a display of sleight of hand, several other international standards and processes are presented towards the end of this section along with a discussion on how they could be related to the specific standard models shown here.

Having established the groundwork, it is time to dive straight in and start to model the standards.

### *6.3.3 EIA 632 Processes for engineering a system*

The first model that is drawn up is what will be referred to as the high-level structural model. There is no rule saying that this must always be the first model, but experience has shown that it is a useful one to get out of the way early on in the modelling.

The high-level structural model really sets the scene for the remainder of the modelling and contains the most fundamental concepts of the standard itself. It is crucial that this model can be clearly and unambiguously defined, otherwise the rest of the standard is rendered meaningless. The idea is to try to extract the actual structure, or hierarchy, of the processes that are defined in the standard. This is achieved by looking for UML-friendly words in the process descriptions in the standard, such as: ‘is a type of’, ‘is made up of’, and the like.

Figure 6.1 shows the high-level structure of EIA 632 and is read as follows.

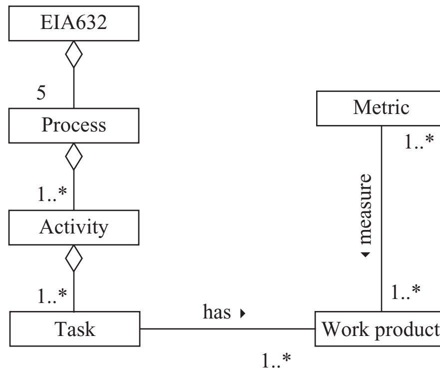


*Figure 6.1 High-level structure of EIA 632*

'EIA 632' is made up of five 'Process'. This is the highest-level piece of information that could be extracted and states the whole structure of the standard itself. This information is quite easy to extract as the five processes form the major sections in the standard.

The next item of information is the structure of 'Process', which is made up of one or more 'Activity', each of which is made up of one or more 'Task'. Each 'Task' has one or more 'Work product' associated with it. One or more 'Metric' measures one or more 'Work product'. This information was also quite easy to extract from the standard and is stated explicitly in the standard itself.

In a nutshell, this model describes the whole structure of the standard. All subsequent models go into greater detail, but will have to be consistent with this model.

The next step is to concentrate on one of the classes – in this case, 'Process'. It was stated in the previous model that there are five processes. Remember that the term 'is a type of' is a term used in the UML, which provides the basis of the model shown in Figure 6.2. Therefore, the diagram reads: there are five types of 'Process',

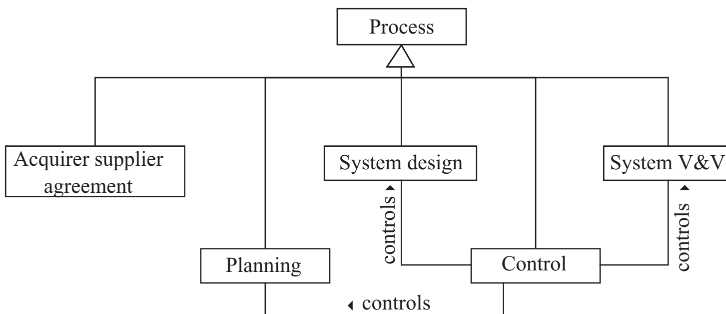


*Figure 6.2 Types of process in EIA 632*

which are 'Acquirer supplier agreement', 'Planning', 'System design', 'Control' and 'System V&V'. Several other relationships also exist in the model, which are: 'Control' controls, 'Planning', 'System design' and 'System V&V'.

As an interesting aside, note how 'Control' does not control 'Acquirer supplier agreement'. Is this correct, or is it an oversight by the authors? Already, even at this high level, it is possible to throw up potential inconsistencies or problems with the standard. This is an issue that may have to be addressed before the modelling can be taken any further and the full implications of the lack of 'Control' must be investigated.

The next step is to focus on one of the five processes, in this case 'Systems design', which is shown in Figure 6.3. Note how the original relationship from the high-level structural model in Figure 6.1 holds true here, in that 'System design' (a type of process) is made up of four (more than one) 'Activity'. Now there are four types of 'Activity': 'Requirements definition', 'Design Verification', 'Assessment and trade-off studies' and 'Design definition'. It can also be seen that 'Requirements definition' has types 'Stakeholder requirements', 'System technical requirements' and 'Detailed technical requirements'.

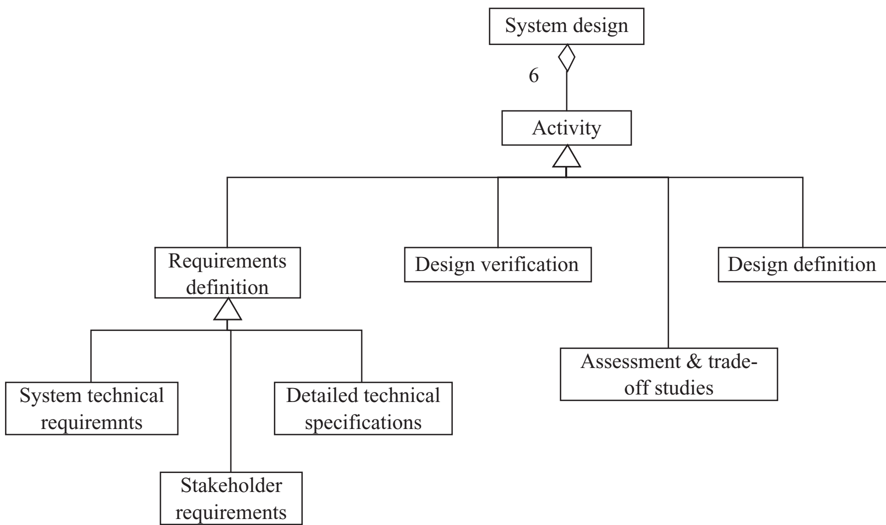


*Figure 6.3 Focus on system design*

This modelling can also be carried out on the other four types of 'Process' that were shown in Figure 6.2. This would result in a separate diagram each for 'Acquirer supplier agreement', 'Planning', 'Control' and 'System V&V', all of which must be consistent with Figure 6.1.

At this point, we leave EIA 632 and move on to the second standard to be modelled, ISO 15504, and carry out the same level of modelling on it.

### *6.3.4 ISO 15504 Software process improvement and capability determination (SPICE)*

This section looks at modelling ISO 15504, or SPICE as it is more commonly known, which will then be compared to EIA 632.

SPICE is the ISO standard that defines a set of ISO processes (that are completely compatible with ISO 9001) and relates them to a capability maturity model, similar to the CMM. The actual SPICE standard is monstrous in size, weighing in as it does at over 300 pages. This may seem fairly daunting to begin with, but SPICE itself is very well specified and is thus straightforward to model.

The first step in the model is to try to establish a high-level static view of the standard, as shown in Figure 6.4.

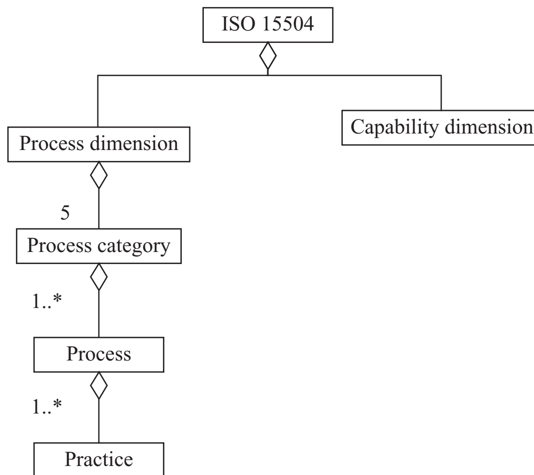


*Figure 6.4 High-level structure of ISO 15504*

It can be seen in Figure 6.4 that 'ISO 15504' is made up of a 'Process dimension' and a 'Capability dimension'. This is the highest possible level view of SPICE and the focus here will be on the 'Process dimension' rather than on the 'Capability dimension', as it is this part of SPICE that defines the process structure. The 'Capability dimension' describes the capability levels that are defined for SPICE and how they relate to practices. As this lies outside the scope of this book, the detail has been omitted from Figure 6.4.

Figure 6.4 shows that there exists a 'Process dimension' that is made up of five 'Process category'. Each 'Process category' is made up of one or more 'Process', each of which is made up of one or more 'Practice'.

The next question that needs to be asked is on which part of the model can we focus to show more detail? Note that there is an explicit multiplicity number on the association going from 'Process dimension' into 'Process category'. Therefore, it is likely that because there is an explicit number, it may be possible to look at what these process categories are. This is shown in Figure 6.5.

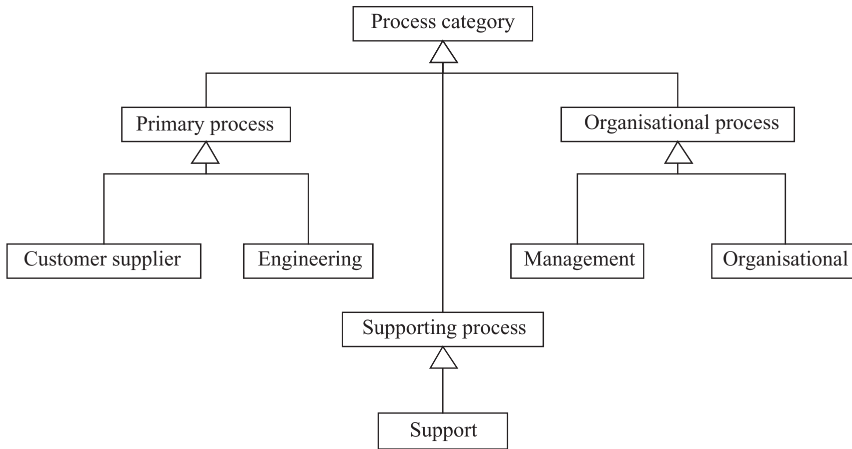


*Figure 6.5 Types of process category in ISO 15504*

Figure 6.5 shows types of 'Process category', bearing in mind that it was already stated in the previous model that there were five types of process category. From the model, there are three main types of 'Process category': 'Primary process', 'Organisational process' and 'Supporting process'. These are then further refined: 'Customer supplier' and 'Engineering' are types of 'Primary process'; 'Management' and 'Organisational process' are types of 'Organisational process'; and 'Support' is a type of 'Supporting process'. This brings the total number of actual process categories (or 'leaf' process categories) to five, as in the previous model. Note that these middle-level process categories ('Primary process', 'Supporting process' and 'Organisational process') are conceptual groupings of processes, rather than actual categories. In UML terms, these would be described as 'abstract classes', which is to say that they are not instantiated. In UML notation, this can be represented by showing the class name in italics to indicate that it is an abstract class.

The next step is to focus in on one of these categories and examine whether the model can be taken any further.

Figure 6.6 focuses on one of the process categories defined in the previous model – in this case, 'Engineering'.

'Engineering' is made up of eight 'Process', which is still consistent with the original high-level structural model that stated that each 'Process category' was made up of one or more 'Process'. There are two types of 'Process': 'Development' and 'Maintenance'. 'Development' has types 'System requirements', 'Software requirements', 'Software design', 'Software construction', 'Software integration', 'Software testing' and 'System integration and test'. The process named 'Development' may be thought of as being abstract and could be indicated as such on the model.

The model for ISO 15504 (SPICE) has now be modelled to approximately the same level as the EIA 632 and, at this point, the models will be compared.

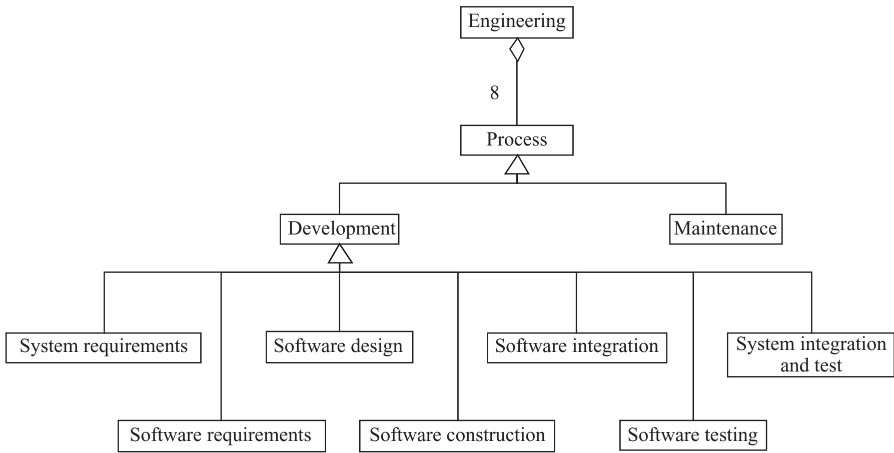


*Figure 6.6 Focus on engineering*

### 6.3.5 Comparing models

The next step is to compare the two sets of models that have been created. This is simply a matter of placing the diagrams adjacent to each other and looking for similarities in terms of names and patterns.

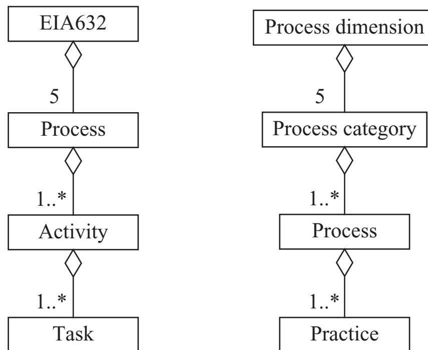


*Figure 6.7 High-level structure comparison*

Figure 6.7 gives both high-level models side by side, ready for comparison, and shows that the structure of the two standards is very similar. Both have similar-looking classes, aggregations and, indeed, even some of the terms are the same.

The most important thing to establish before the full comparison can be drawn is a common point between the two standards, from which all other relationships can be derived. The interesting thing here is that the common point occurs at the level of 'Task' (in EIA 632) and 'Practice' (in ISO 15504 (SPICE)), rather than at the common

term 'Process'. This was established by looking at the definitions of the terms and also by applying common sense with regard to how they are used in the standard. Even though both standards have been modelled, it is still necessary to apply a healthy dose of common sense to make sense of, and add value to, the models.

This now throws up something very interesting since the next level up in the aggregation hierarchy shows that 'Activity' (in EIA 632) is equivalent to 'Process' (ISO 15504 (SPICE)) and that the term 'Process' (in EIA 632) is equivalent to 'Process category' (in ISO 15504 (SPICE)). Note that the common term 'Process' actually has a completely different meaning in both standards! This means that perhaps the most fundamental term in each standard (and one that was discussed previously as being crucial to standards), means two different things in two different standards. This has the potential to lead to a massive amount of misunderstanding and ambiguity that could prove very harmful indeed to the project.

The next step is to compare the next level of diagrams and see if any more parallels or inconsistencies can be thrown up.

It is now possible to compare the types of processes with one another, as shown in Figure 6.8, with EIA 632 being on the left and ISO 15504 (SPICE) on the right.

There is a comparison between both types of process, which can be verified by looking at the standards. In this case, the comparison is as follows:

- 'Acquirer supplier agreement' in EIA 632 is equivalent to 'Customer supplier' in ISO 15504 (SPICE).
- 'Planning' in EIA 632 is equivalent to 'Management' in ISO 15504 (SPICE).
- 'Control' in EIA 632 is equivalent to both 'Organisational' and 'Support' in ISO 15504 (SPICE).
- Both 'System design' and 'System V&V' in EIA 632 are equivalent to 'Engineering' in ISO 15504 (SPICE).

In this way, the common elements of both standards may be highlighted in addition to those that exist in one and not the other.

Note that the mapping is not a simple one on one mapping as it might first appear, but is a slightly more complex relationship. 'Control' in EIA 632 maps to two process categories in ISO 15504 (SPICE), rather than one. In the same way, 'System design' in ISO 15504 (SPICE) relates to two processes in EIA 632.

Each of the two standards may be modelled in even more detail by adding lower-level process information. For example, it is possible to model each 'Practice' that is associated with each 'Process' in ISO 15504 (SPICE) and each 'Task' for each 'Activity' in EIA 632.

#### 6.3.5.1 Highlighting complexity

Another use for the models that are generated is to help to highlight areas of potential complexity within the standards. Although all of the models generated so far have been class diagrams and, hence, structural models, it is also possible to model behavioural aspects of standards using, for example, state machine diagrams or activity diagrams. The following example shows a state machine diagrams that was generated based on the description of the acquirer supplier processes. Note that this is a statechart because

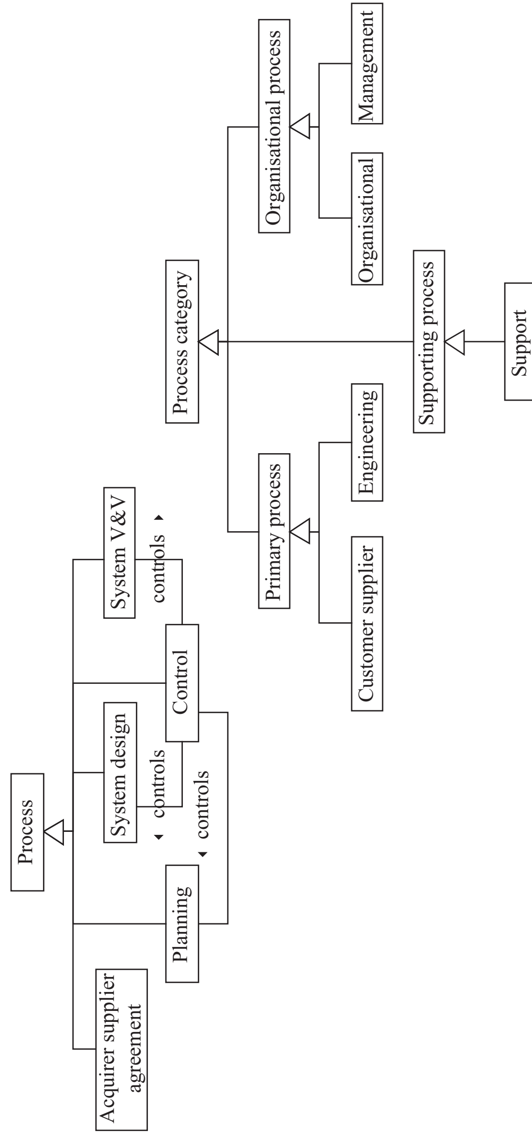


*Figure 6.8 Types of process comparison*

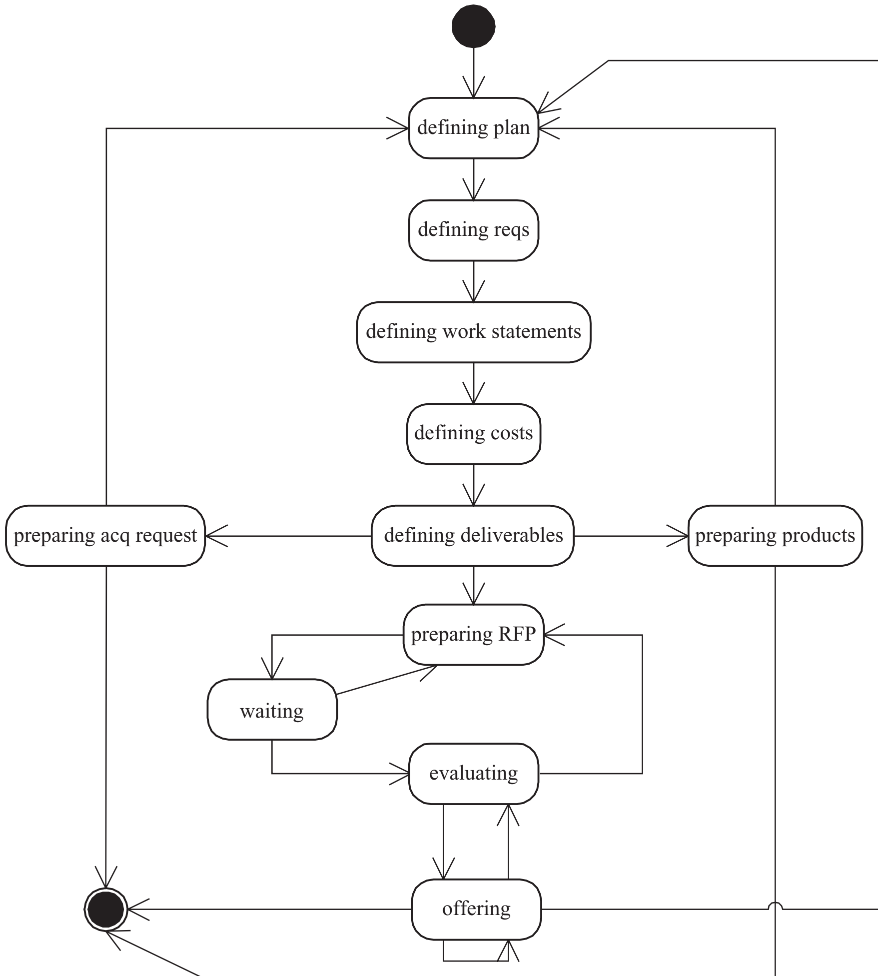


*Figure 6.9 Highlighting complexity*

each state is a normal state, as opposed to an activity or action state. This also implies that any or all of these states may contain more than one action or activity and, hence, be complex.

The first thing that springs to mind when looking at Figure 6.9 is that the model is messy. Look at the model and try to estimate the number of possible paths through this process. What this means in systems engineering terms is that the diagram is exhibiting complexity, which may be identified in a number of ways:

- If the information that is being modelled is difficult to model, it is quite likely that complexity exists in the original information. It may be that the information is ambiguous, underdefined or overdefined. Remember that the UML is, at the end

of the day, a language, and should therefore be easy to read. By the same token, if anything is difficult to model, the original source information is probably not very clear.

- If the model looks messy, complexity almost certainly exists in the model. This is a rather basic heuristic but one that tends to work. There are more formal ways to highlight complexity, such as applying metrics to the models that can actually give a quantitative result.

It may turn out that this complexity is inherent in the system (essential complexity) and, as such, nothing can be done about it. It is still important, however, to know about this complexity so that any interactions with this part of the system may be minimised. If, on the other hand, the complexity has been introduced by human error (accidental complexity), it should be possible to address this, such as redesigning or rewriting the original information.

#### 6.3.5.2 Continued analysis

The models shown so far have been concerned with the high-level views of the standards. How much detail is included in the model is dependent on what level of understanding is required by the reader. It is possible to model every single aspect of a standard but there is a danger that the model would become larger than the original standard.

The analysis that has been started here may be continued by creating more statecharts that describe the behaviour of processes, but, interestingly enough, this is not always possible. Many standards are only defined at the level where they state what must be done rather than how to do it, as they tend to avoid being overprescriptive. The fact that sometimes enough information is included in the standard to allow the behaviour or process to be modelled actually tells us something, as does the fact that sometimes a process cannot be modelled behaviourally. There is much information to be gained from modelling, even if modelling is not actually possible. When defining company procedures that may be compliant with these international standards, for example, it is often desirable to go down another level of detail and describe the behaviour of each process. Indeed, this topic is discussed later in this chapter when defining new procedures is discussed.

It is also possible to model life cycles and life cycle models, inputs and outputs for each process and the actual structure of deliverables, which is covered later in this chapter.

### *6.3.6 Other standards and processes*

#### 6.3.6.1 Introduction

This section looks at a number of high-level models of other standards and processes and discusses how each one may be compared to the previous models shown in this chapter. It should be quite clear from the models that there are similarities between each of the standards as patterns begin to emerge [4]. These patterns are exactly what they say and may be identified and reused for defining in-house procedures. These

are the same as design patterns that may be reused frequently and that will help to deliver a more robust design.

#### 6.3.6.2 ISO 15288 Life cycle management – system life cycle processes

This standard has been developed by taking into account the experiences of the use of many other standards. Inputs to this standard include: EIA 632, IEEE 1220, ISO 15504 (SPICE) and ISO 12207. As these standards have formed an input, it is not surprising that some common patterns emerge.

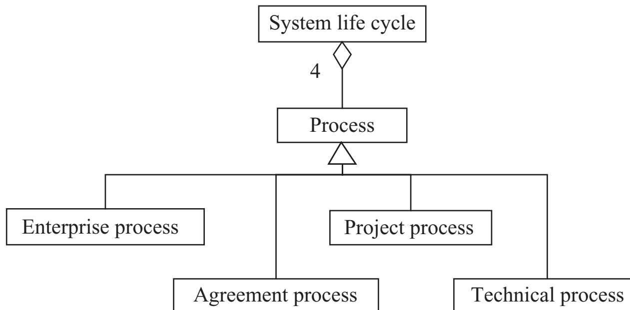


*Figure 6.10 High-level model of ISO 15288 Life cycle management – system life cycle processes*

Figure 6.10 shows that ISO 15288 Life cycle management – system life cycle processes is made up of four 'Process', with types 'Enterprise process', 'Agreement process', 'Project process' and 'Technical process'.

There is a clear mapping between this model and the two previous standards that were modelled, but note that this standard has only four main processes, whereas the previous two had five each. This means that there must be a degree of overlap between some of the types of process defined in this standard and in the others.

In addition, look at the use of the term 'Process' and try to decide exactly where this would map on to the previous two models. Would it map directly to SPICE or EIA 632, or does it have an entirely different meaning in this standard? This is exactly the type of information that we are trying to find out by modelling and it increases our understanding of the standard the more it is modelled.

#### 6.3.6.3 IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems

This standard has no direct relation to any of the other standards mentioned in this section, which becomes apparent when the model is considered.

From Figure 6.11 it can be seen that IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems is made up of 17 phases.

Although at first glance this looks messy, perhaps it would look cleaner if the phases were grouped into categories. This grouping into categories is a level of

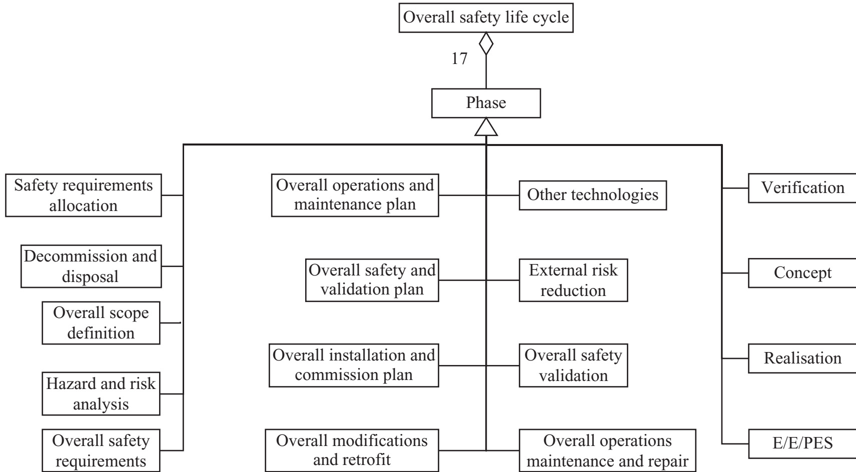


*Figure 6.11 High-level model of IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems*

organisation that exists in other standards but not in this one; this highlights that complexity exists at the highest level of the standard, whereas the lower-level models turn out to be quite concise and well defined.

It is interesting to see the term 'phase' used here, which has not been mentioned so far in this chapter, but which occurs later.

#### 6.3.6.4 IEEE 1220 Application and management of the systems engineering process

This standard has a unique pattern that stresses iteration and has been used successfully for many years, particularly in the United States. It has been superseded by the new ISO 15288 Life cycle management – system life cycle processes.

Figure 6.12 shows IEEE 1220 Application and management of the systems engineering process, which shows that a 'System life cycle' is made up of one or more 'Life cycle stage'. Each 'Life cycle stage' is carried out according to one or more 'Sys eng process' and one or more 'Sys eng process' is applied to a 'System life cycle'. One or more 'Sys eng req' plan and implement a 'Sys eng process'. In addition, there are eight types of 'Sys eng process'.

Again, although there are more processes than in some standards, it may be due to the grouping (or lack of it) or the processes.

An interesting point here that has been raised on numerous occasions is that there is no concept of design mentioned anywhere in the model. Does this mean that the standard is not at all concerned with design, or is it an oversight, or does design go under the auspices of some other term? The reality of this is that elements of design exist within 'Functional analysis' and 'Synthesis'.

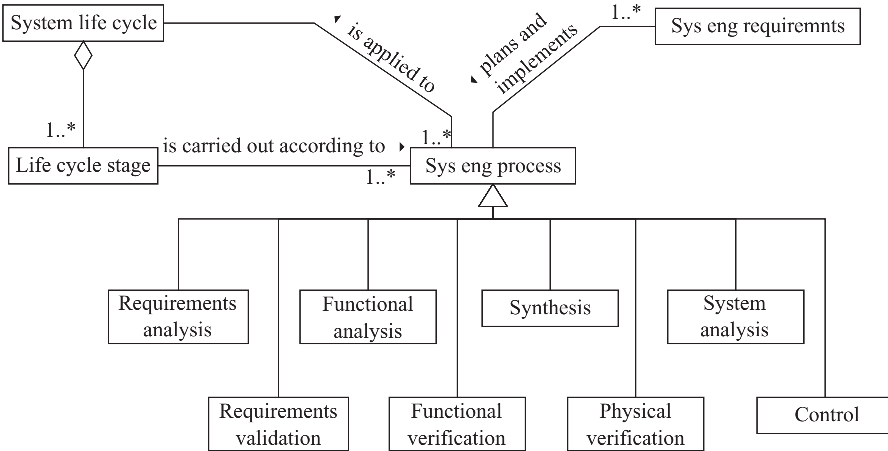


*Figure 6.12 High-level model of IEEE 1220 Application and management of the systems engineering process*

#### 6.3.6.5 The Rational Unified Process (RUP)

The Rational Unified Process (RUP) is a natural evolution of many existing processes that is the recommended process from the originators of the UML [5, 6]. The RUP has had a large uptake in the United States and is being adopted rapidly in the rest of the world, especially Europe. One of the initial problems that some people have with the RUP is that the terminology used is 'Americanised'. This can be quite confusing for people with an ISO background, but once the mapping between terms has been established, this should not be a problem.

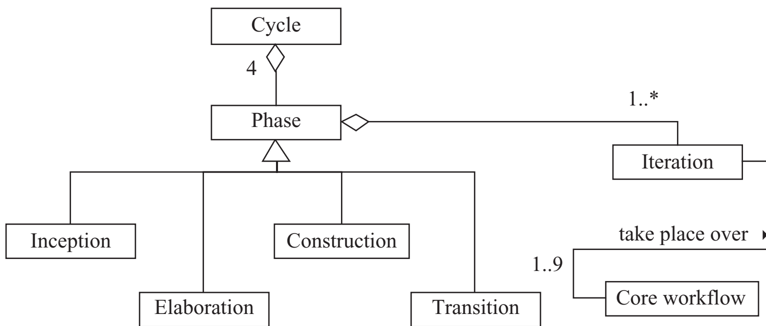


*Figure 6.13 High-level model of the Rational Unified Process (RUP)*

Figure 6.13 shows the RUP, which is recommended by the developers of the UML and which has been included here for completeness sake. It can be seen that a 'Cycle' is made up of four 'Phase', of which there are four types: 'Inception', 'Elaboration',

'Construction' and 'Transition'. Each 'Phase' is made up of one or more 'Iteration' and between one and nine 'Workflow' take place over each 'Iteration'.

If this is mapped on to the previous model, it can be seen that there is no actual term 'process', so where does the comparison start? The actual equivalent term on this model is 'Core workflow'.

#### 6.3.6.6 EIA 731 Systems engineering capability model

This standard is intended to be a standard for systems engineering capability in the same way that ISO 15504 (SPICE) is the standard for software engineering capability determination. It claims to complement both IEEE 1220 and EIA 632, as well as being consistent with ISO 9001.

Like ISO 15504 (SPICE), the standard itself is huge, weighing in at several hundred pages but, also like SPICE, it is very well written and surprisingly easy to model. The high-level structure model for EIA 731 can be seen in Figure 6.14.

This standard is widely used by the defence and aerospace industries, which have used it to great effect.

Figure 6.14 shows the high-level structure of EIA 731 Systems engineering capability model. It can be seen that 'Sys eng capability model' is made up of a 'Sys eng domain' and a 'Capability domain'. This pattern is very similar to the pattern in ISO 15504 and, like that standard, it is the process side of the model that we are interested in, rather than the capability side. The 'Sys eng domain' is made up of three 'Category', each of which is made up of one or more 'Focus area'. Each

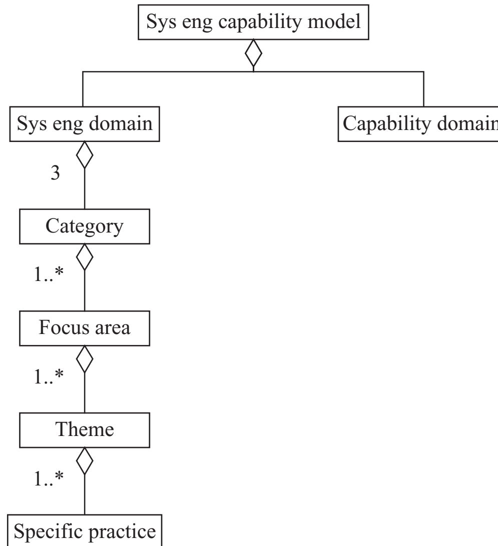


*Figure 6.14 High-level structure of EIA 731*

'Focus area' is made up of one or more 'Theme', each of which is made up of one or more 'Specific practice'.

### *6.3.7 Summary*

In summary, therefore, the following points were raised concerning the use of the UML for modelling standards and processes:

- The UML is suitable for modelling standards and their associated processes. Several standards were looked at and their processes modelled, and carried out at a high level.
- By modelling in this way, it is possible to highlight potential inconsistencies within a standard. Any internal inconsistencies, such as badly defined terms or inputs and outputs that do not match, would come to light when modelled visually.
- It is also possible to compare and contrast these standards once they have been modelled in order to throw up inconsistencies between standards. In the examples used here, it was quite clear from comparing the models that EIA 632 and ISO 15504 had a completely different meaning for the basic term 'process', which, if not spotted at this point, may have led to all sorts of trouble and misunderstanding later in the project.
- It is also possible to highlight areas of potential complexity within a standard. Indeed, one of the standards that was considered, EIA 632, had complexity in one of the processes, which was identified by creating a behavioural model of the process.

Now that some standards have been modelled, the next step is to use this information further to help us to define our own processes and procedures that are compatible with these standards.

## 6.4 Defining new processes using the UML

This section uses the information from the previous section to show how new processes can be defined that are compatible with, and fully traceable to, international standards. At the heart of this technique is, of course, modelling using the UML. The analysis models generated so far may be used in order to define a new process that may be used for in-house standards and procedures.

When more than one standard has been modelled, 'patterns' begin to emerge that are common throughout all the standards. These patterns are often the basis for defining new processes. In addition, particular patterns may be crucial for a process – such as real-time or safety-critical patterns.

### *6.4.1 A new procedure*

Imagine the scenario now where a new procedure must be defined that is compatible with a number of international standards. This new procedure is to be used by all

design teams in a systems engineering company and, therefore, it must be understood by everyone in the company and must be communicable effectively for training purposes.

The two standards that have been modelled so far (EIA 632 Processes for engineering a system and ISO 15504 Software process improvement and capability determination (SPICE)) will be used as a basis for international standards compliance.

The first few steps are the same as for analysing standards, in that the high-level models are created, but then the modelling is taken further. The first step in the analysis was the high-level structure model, so this is the first model that is created for process definition.

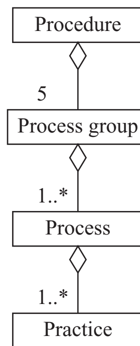


*Figure 6.15 High-level process structure model for the defined process*

Figure 6.15 shows the high-level process structure for a new procedure that is going to be defined. It can be seen that the 'Procedure' is defined as being made up of five 'Process group', each of which is made up of one or more 'Process', each of which is made up of one or more 'Practice'.

It can be seen that the terms used here are similar to the ones in the international standards that were modelled previously. This is intentional as it is intended to adopt an ISO-style terminology. Therefore, where inconsistencies do occur, the ISO term, rather than the EIA term, is opted for. The exception to this is the term 'Process group', which was decided upon as it was an established term used within the organisation. However, these areas of potential misunderstanding are now known and each term can be defined explicitly at the beginning of the procedure.

Also at this point, the information in the new procedure can be mapped on to the analysis models for the standards, ensuring traceability and compliance with them. It should be quite clear how the high-level structure model in Figure 6.15 can be mapped directly on to both EIA 632 (Figure 6.1) and ISO 15504 (SPICE) (Figure 6.4). The next step is to define the types of 'Process group' that can be defined, which are shown in Figure 6.16.

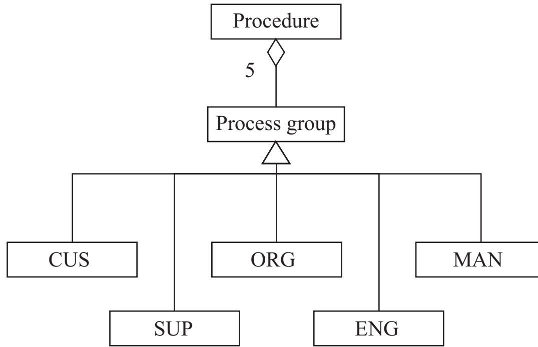


*Figure 6.16 Types of process group for the defined procedure*

Figure 6.16 shows the five process groups that were stated on the previous model and expands upon them.

There are five types of ‘Process group’ and these are ‘CUS’ (Customer/supplier relationship), ‘SUP’ (support services), ‘ORG’ (organisational), ‘ENG’ (engineering) and ‘MAN’ (management). Again, it is worth stressing that the language here is tending towards ISO rather than EIA, which is purely a matter of preference.

It is clear to see where these process groups have come from as both of the standards that were modelled have a similar structure, or pattern.

Note also, that new terms are being defined as the models evolve, but, once more, they are fully traceable back to the original terms in the original standards.

The next step is to focus in on one ‘Process group’ from the previous model. The one chosen for this example is ‘ENG’, or engineering. The process group ‘ENG’ is made up of one or more processes, which holds true from the previous models. There are seven processes: ‘User requirements’, ‘System specification’, ‘Design’, ‘Implementation’, ‘Integration’, ‘Installation’ and ‘Maintenance’ (Figure 6.17).

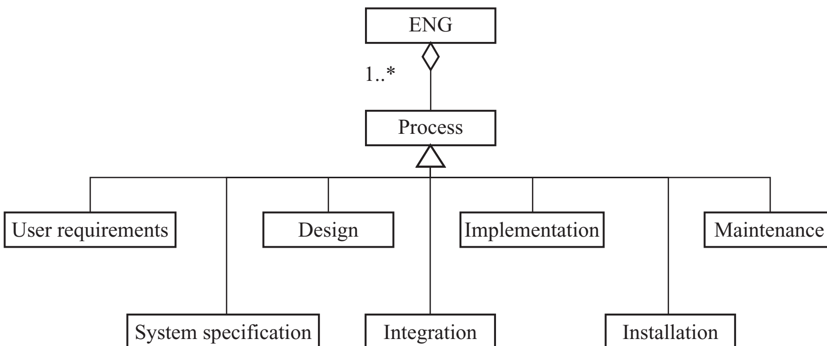


*Figure 6.17 Focus on the ‘Engineering’ process group*

Once again, it is a simple matter, at this level, to map these processes on to their corresponding elements from the two standards, simply by comparing the model in Figure 6.17 with their counterparts in Figures 6.3 and 6.6.

The modelling process definition is now at the same level as the two standards that were analysed previously. However, as this is a procedure, we want to go into more detail and explicitly define which 'Practice' makes up each 'Process'.

The next step is to define some practices for each process, as the previous models stated that each 'Process' was made up of one or more 'Practice'. In order to make the model more manageable, these practices will be represented by operations on the class – even though they were previously shown as component classes. All that has been done here was to abstract the information to a higher level to make the models simpler.

A similar abstraction was carried out to show what documents were needed as outputs, or deliverables, for each process. This time, however, the deliverables are shown as attributes.

Design

Integration test spec
System design
System test spec
Design review report

design interfaces()
architectural design()
detailed design()
develop tests()
review()
update / issue()
establish traceability()

*Figure 6.18 Focus on a single process – 'Design'*

Figure 6.18 shows an expansion of a single process, that of 'Design'. It can be seen that seven practices have been defined and are shown on the model as the operations 'design interfaces', 'architectural design', 'detailed design', 'develop tests', 'review', 'update/issue' and 'establish traceability'. The deliverables that are associated with the 'Design' process are shown as attributes and are 'Integration test spec', 'System design', 'System test spec' and 'Design review report'. This provides us with rather a neat representation of a single process that may be shown visually by a simple class. Note that the class exhibits behaviour, or to put it another way, it has operations, and therefore it is possible to describe this behaviour using a statechart.

The technique in Figure 6.18 used a single class to show structural aspect of a process, or what it should look like. Referring back to the UML, it was previously stated that in order to model a system fully, any class with behaviour (operations) must have an associated state machine diagram. The process is now described in terms of its behaviour, which shows how things are done, in what order and under what conditions, by creating a state machine diagram for it.

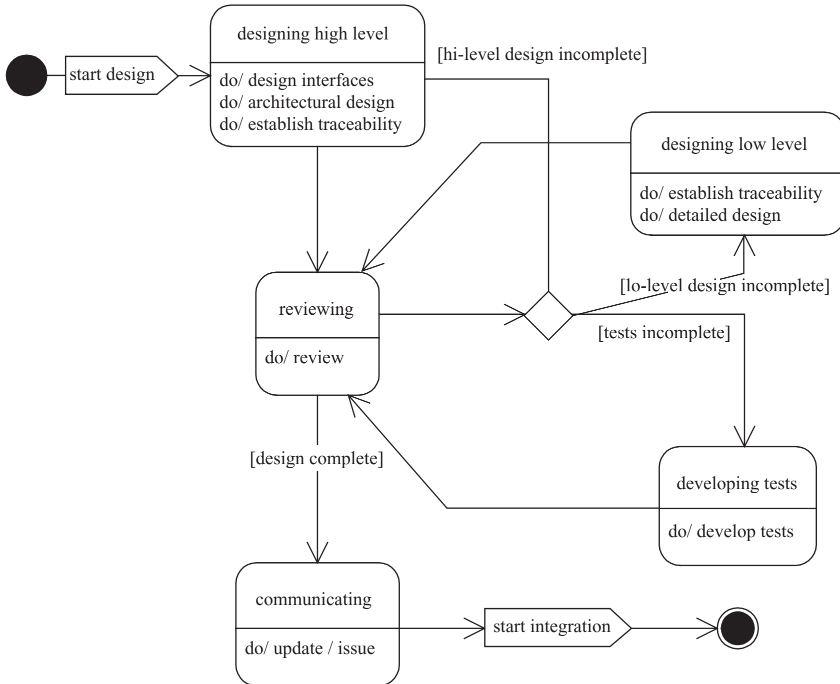


*Figure 6.19 Behavioural model for the 'Design' process*

Figure 6.19 shows that the 'Design' process begins by a 'start design' event, which passes the deliverable 'SS', which refers to the system specification. The first state is called 'designing high level' where three of the operations, or practices, are executed in sequence: 'architectural design', 'design interface' and 'establish traceability'. After this, a 'reviewing' state is entered where 'review' is executed and can continue in one of four ways:

- '[lo-level design incomplete]' that leads to 'designing low level' and executes 'detailed design' and 'establish traceability'. After this, control returns to the 'reviewing' state.
- '[tests incomplete]' that leads to 'developing tests' where 'develop tests' is executed. After this, control returns to the 'reviewing' state.
- '[hi-level design incomplete]' that leads back to the 'designing high-level' state.
- Finally, '[complete]', which leads to 'communicating' where 'update/issue' is executed, which then leads to the final state and a 'start integration' send event.

The model shown here has complex states by placing more than one activity in some of the states. This is perfectly acceptable from a UML point of view, but some people prefer to separate them out into a 'one-activity-per-state' diagram. Although there is nothing implicitly wrong with using a single activity per state, it may be worth asking if the use of activity diagrams may be more appropriate at this point. Activity diagrams

are often used to model processes or, as they are known in the RUP, workflows, so it seems appropriate to discuss them at this point. Suppose that it is possible to represent a single activity per state for the new procedure (which should be possible if it is being designed, rather than analysing what already exists), then it is possible to use an activity diagram, rather than a state machine diagram, but are there any advantages to this approach? The answer is 'yes', as there are two previously unmentioned activity diagram elements that may be brought into use here: 'swimlanes' and 'object flow'. Swimlanes may be used to group together states and then associate them with a particular class or object. This is useful when modelling processes and workflows as it allows responsibility to be modelled on the activity diagram by relating each swimlane to a class from the stakeholder model. This also fits in with the definition of a process where responsibility must be associated with all practices.

Object flows may then be used to show the flow of deliverables around the process, by showing the object name (that will relate directly back to a deliverable) and a dependency to and from its associated states. Figure 6.20 illustrates these two new activity diagram concepts.

Figure 6.20 shows the enhanced activity diagram that represents the same process as the model shown in Figure 6.19, with a number of notable differences. Responsibility may be shown in the activity diagram by using swimlanes, which are not available for use in standard state machine diagram. Information flow may be shown more explicitly using the activity diagram compared to the state machine diagrams, as deliverables may be shown as objects, rather than as arguments on the

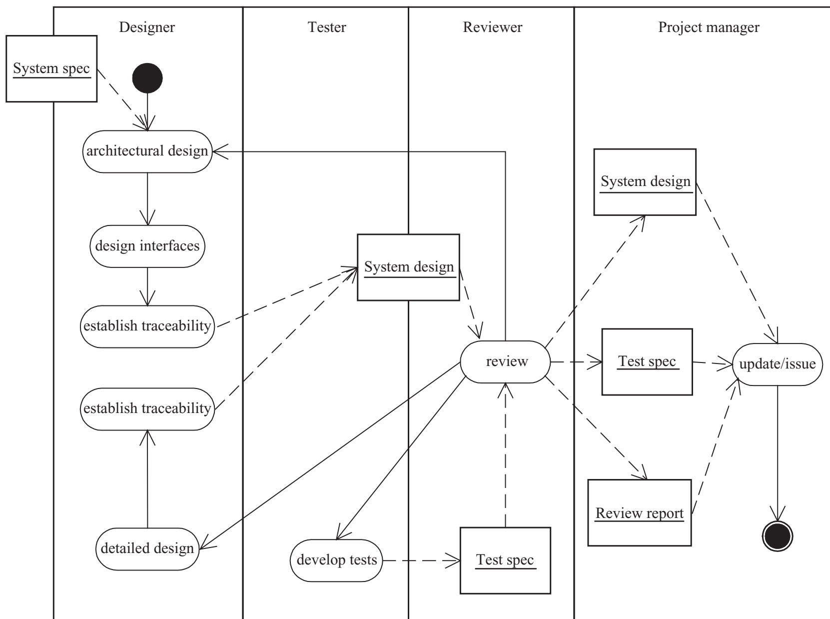


*Figure 6.20 Activity diagram with swimlanes and object flow*

state machine diagrams. There is a disadvantage, however, as it is assumed that all states are activity invocations, rather than normal states, as used in statecharts.

Returning to the new procedure, it should be noted that this level of behavioural modelling is usually one step further than the information contained in most standards (although there are exceptions to this, the 'Acquirer supplier agreement' of EIA 632 being one of them). Standards will normally dictate what must be done in order to comply with the standard, rather than how to actually do it. This is done in order to avoid being prescriptive. With a procedure, however, it is normally desirable to be more prescriptive and to describe 'how' things must be done. If you relate this to basic UML concepts, then it fits with the definitions of structural and behavioural modelling, where structural modelling describes the 'what' of a system and behavioural modelling describes the 'how' of a system.

This level of behavioural modelling is also very useful from a project management point of view, as at any point within a project it is possible to see exactly what is being done by relating current work activities to the statechart associated with a particular process.

The model may be taken one step further by modelling the structure of an individual operation that represents a basic practice within the procedure. These are modelled using activity diagrams and an example of such a model is shown in Figure 6.21.

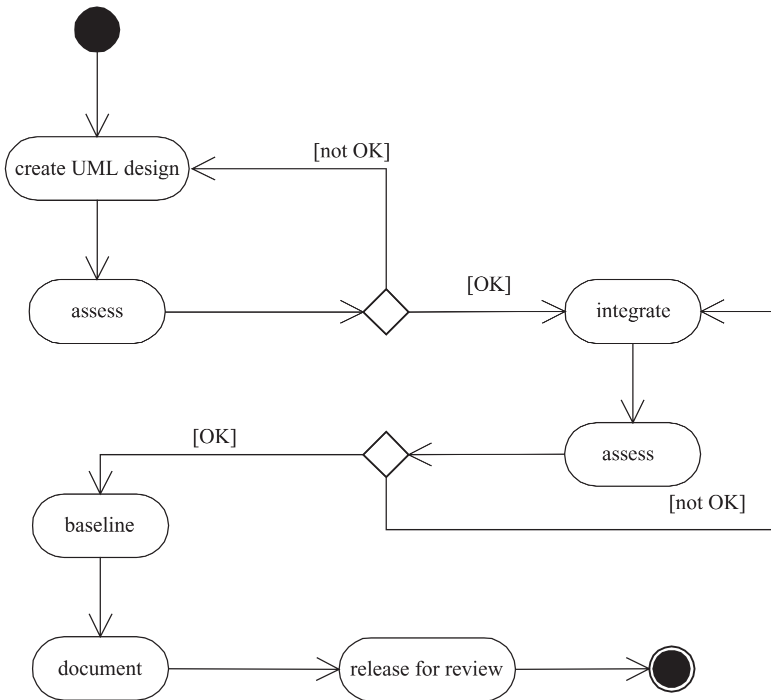


*Figure 6.21 Low-level description of the 'Architectural design' practice*

Figure 6.21 shows a low-level activity diagram that models the 'architectural design' practice from Figure 6.19.

From the model, therefore, it can be seen that the first step is to 'create UML design' and then 'assess' it. If the result of the assessment is 'OK', the next step is to 'integrate'. If the result of the assessment is 'not OK', the 'create UML design' state must be revisited.

After integration, there is another 'assess' state that leads to another decision block. If the assessment is 'not OK' then 'integrate' is revisited, whereas when the assessment is 'OK' the process enters the 'baseline' state, which represents baselining the actual designs themselves. After this baselining, the designs are written up formally in a 'document' state. The final state is 'release for review', which basically makes the document available for the next state from Figure 6.19, which is the 'reviewing state'.

This model may be enhanced by adding swimlanes and object flow, in the same way that they were shown on Figure 6.20, which may add more value to the diagram.

It may be argued that modelling individual practices is going a step too far and is too much effort for the amount of work that will go into defining the models at such a low level of detail. This is, of course, true, but like most aspects of UML modelling it is entirely up to the modeller how far they go. For many situations, it may indeed be the case that this level of modelling is too detailed, but there are a number of circumstances where it may be useful:

- Imagine the situation where the design was supposed to be applied to a safety-critical application. It may be essential that the design is carried out exactly according to a set of standards or guidelines. In this situation, this low level of modelling would be useful.
- Another situation where low-level modelling is useful is where new people may be coming into a project at a late stage. In such a case, it may be a good idea to have a set of guidelines that can be made available to the newcomers to bring them up to speed on the project. Although these guidelines may not be enforceable, they are useful as a learning aid.

This is the lowest level of modelling that will be performed for process definition.

### *6.4.2 Completing the model*

In order to complete the model, the steps that have been followed for the single process should be carried out on all the other processes. To recap, these steps are as follows:

- Define a high-level model that encapsulates the whole process structure in a single diagram. This is a good basis for comparison with other standards and processes and can form the basis of traceability for demonstrating compliance with such a standard. This model was shown in Figure 6.15 for the example of process definition.
- Categorise processes into groups that summarise the basic conceptual groupings of the processes. This can help to avoid complexity at a high level, as exists in

IEC 61508, and as shown in Figure 6.11. This model was shown in Figure 6.16 for the example of process definition.

- Define practices for each process as operations on the process class. This models the basic high-level allocation of practices to individual processes. In some circumstances, such as an organisation-wide procedure, it may be desirable to cease modelling at this point. An example of this was shown in Figure 6.18 for the example of process definition.
- Define deliverables for each process as attributes on the process class. These may be derived from a class diagram that identifies deliverables per process or may simply be stated and added to the class. An example of this was also shown in Figure 6.18 for the example of process definition.
- Define a state machine diagram or activity diagram for each process to describe the behaviour of the practices (the order in which they are carried out and the logical conditions under which the practices are executed). This will help people who need to be trained in using the procedure, or project managers who may need to know exactly what is happening (which state the system is in) at any point in time of the project. Examples of this were shown in Figures 6.19 and 6.20 for the example of process definition.
- Where called for, define activity diagrams for any complex practices. This may be desirable for safety-critical applications or by way of training new staff on a project. An example of this was shown in Figure 6.21 for the example of process definition.

These steps may be repeated (or ignored) as necessary until the procedure is fully modelled. It should be quite clear by now that there is a great deal of effort involved with process definition, which is absolutely true. However, defining processes and procedures is a very complex business and it is very easy to get it wrong. By modelling such processes visually, we can have more of an understanding of the process and have more confidence in the final deliverable process or procedure.

The process models that are produced should be reviewed before any part of the actual procedure is written. Think of this as a design review for the procedure.

### *6.4.3 Summary*

In conclusion, therefore, this section shows how a new procedure may be modelled that makes use of the models generated when the standards were analysed.

This modelling was carried out in order to visualise the concepts involved in standards or procedures and has several benefits over using an English text description of a procedure:

- Traceability to the original standards may be established at the modelling stage, which is enormously useful in the event of an audit or assessment of the process. This also has the advantage that if traceability needs to be established to a new standard (since they are emerging all the time), it can be established to the process model and the procedure itself need not change.

- Early review. When creating a new procedure, it is possible to review the process at the modelling stage in order to ensure correctness of models and consistency between processes before the process is written up as a process or procedure.
- The process knowledge is stored in the process, rather than in the documented procedure itself. Imagine the written procedure as a small window through which the model may be viewed. This is useful because the procedure may be split across several documents, each one perhaps having a different audience in mind. For example, one procedure may be applicable to engineers and one to managers, one procedure may be used by subcontractors and one by in-house staff, or a different procedure may be appropriate for different types of systems, such as bespoke, niche or COTS (commercial off the shelf) systems.
- The modelling does not stop here, as it is possible to model other information that may be useful in a procedure or standard such as life cycles and deliverables, both of which will be discussed in more detail later in this chapter.

This section has defined the processes and the behaviour within each, but how are theses processes executed during a real project? This is the subject of Section 6.5.

## 6.5 Tailoring the process

One common criticism that is levelled at any sort of process model or standard is its connection to reality. How is it possible to fully define a process yet make it flexible enough to be useable? This is often covered in standards by stating that a process must be 'tailorable', which is far easier said than done.

Depending on the type of project that a process is being used on, any or all of the processes being used may need to be specialised in some way to make them more suitable for the project at hand. This specialisation is also known as 'tailoring'. This is a useful definition of tailoring as there is a mechanism for showing special cases of classes in the UML, which is specialisation. An example of such a process is the case of a project that is concerned with safety-critical systems or real-time systems.

One practical problem that is associated with tailoring processes is that this tailoring knowledge is often lost after a project has run its course, partiularly if the tailored process was not defined well enough within the project. By using the UML it is not only possible to define how a process has been tailored, but also to retain this knowledge in the organisation by building up a process library.

In order to illustrate the tailoring process, an example is used that has already been seen: the 'Design' process from the 'ENG' process group, as shown in Figure 6.22.

The process had its deliverables defined (as attributes) and it practices defined (as operations). This is a generic engineering process that is intended to be used for generic applications. However, in some cases, this simple process may not be directly applicable to the application domain. For example, think about real-time systems or safety-critical systems where a more rigorous approach may be required. This is achieved by tailoring the process to the specific application domain.

In order to tailor a process, the class is simply specialised using the specialisation, or 'is a type of' relationship.

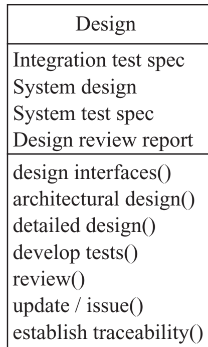


*Figure 6.22 A single process revisited*

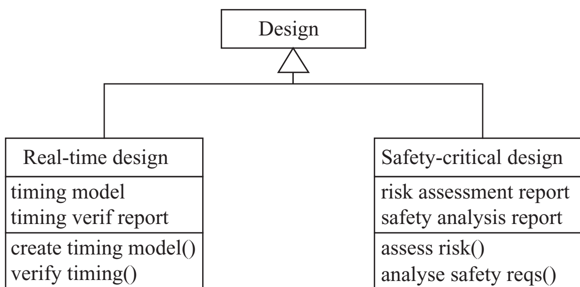


*Figure 6.23 Tailoring processes using a specialisation*

Figure 6.23 shows two specialised types or tailored processes: 'Real-time' and 'Safety-critical'.

Following the rules of UML, the 'Real-time' process will inherit all the attributes and operations (deliverables and practices) from its parent class. In addition to the deliverables and operations defined in the basic process, the 'Real-time' process also has two extra deliverables ('timing model' and 'timing verification report') and two new operations ('create timing model' and 'verify timing'). The same holds true for the 'Safety-critical' process, where two new deliverables have been introduced ('risk assessment report' and 'safety analysis report') and two new practices ('assess risk' and 'analyse safety reqs').

As a consequence of changing the behaviour of the process by adding new operations, a new state machine diagram or activity diagram will have to be created.

By changing the process model by adding specialisations for each tailored process, it is possible to build up a library of tailored processes. In addition, by changing the model and keeping the specialisation in the model, the new knowledge for each tailored process is retained and may be reused.

Figure 6.24 shows the whole class for 'Real-time design', including all its inherited attributes and operations. One point to note, however, is that the number of

| Real-time design |
|---|
| integration test spec<br>system design<br>system test spec<br>design review report<br>timing model<br>timing verification report |
| design interfaces()<br>architectural design()<br>detailed design()<br>develop tests()<br>review()<br>update/issue()<br>establish traceability()<br>create timing model()<br>verify timing() |

*Figure 6.24 The fully tailored process for 'Real-time design'*

attributes and operations is increasing. There is a danger that, as more tailored processes are defined, the processes will become unwieldy and overly complex. If this does occur, it is worth taking a step back and thinking about redefining the basic process to make it simpler. In some cases, it is even necessary to remove all attributes and operations from the basic process (which would make it non-instantiable) and to make all other processes capable of being tailored from this blank process.

#### 6.5.1.1 Creating a life cycle

Once the tailored processes have been defined, it is possible to use them when selecting a life cycle. The procedure for this is exactly the same as before, in that a life cycle is 'assembled' from a number of predefined processes. However, this time there may be a choice of processes where previously there was only one option. For example, a life cycle is chosen, but this time there are three choices of process that may be implemented during the design phase: 'Design' for normal projects, 'Real-time design' for real-time projects and 'Safety-critical design' for safety-critical applications.

Any special types of project will have their application-specific knowledge retained in the 'library' of processes. This ensures that in the future, these tailored processes may be reused and the knowledge concerning the tailoring retained in the organisation. It is often useful to attach UML notes to tailored processes to record any comments about the use of the process. This is important as practical details concerning the implementation of the process may be retained and learned from in the future. An important aspect of many standards is that lessons must be learned from each project that is carried out and they must be demonstrated. By attaching notes to processes that have been used, it is possible to demonstrate to a third party (such as an auditor) that lessons are indeed being learned from past projects.

In relation to lessons learned, consider the case where a tailored process was implemented and found to be an absolute disaster and it was decided that on no

account was that particular process to be used again. One initial reaction to this is to delete it from the process library, but this is potentially hazardous for future projects. If the process is left in the process library and a note is attached to it stating never to use it, then no one will use it again. If the process is deleted, however, there is a danger that someone may redefine it at a future date, add it back to the process library and reuse it with, once again, disastrous results. When the process is deleted, lessons are not learned from past mistakes and it is possible for history to repeat itself. By keeping the process in the library and recording negative experiences about the process, it is possible to learn from past mistakes and not make them all over again.

#### 6.5.1.2 Summary

In summary, therefore, the following points are illustrated:

- Tailoring a process means creating a special case of the process for a specific application or project.
- Processes are represented by classes, and thus tailoring may be represented by the specialisation UML relationship.
- Lessons learned from previous projects can be retained by saving these specialised classes in some sort of process library.

Tailoring of processes covers the penultimate aspect of process modelling before the project can take place. The next step is to define deliverables, which is covered in Section 6.6.

## 6.6 Defining deliverables

This section looks at implementing the process by, among other things, modelling deliverables that are outputs (or work products) from a process. These may be documents, diagrams, code, physical system components, and so on.

Deliverables are modelled in the same way as the rest of the standards. They may be modelled by type, process or structure.

When defining deliverables for a new process, it is possible to base them on existing analysis models from the standards, in exactly the same way as processes were treated previously in this chapter.

### *6.6.1 Deliverables by type*

The first thing to look at is the types of deliverables that may exist within a project. This may be represented using a class diagram.

Figure 6.25 shows that the highest level of class in the diagram is ‘Deliverable’. For this example, we are defining two types of ‘Deliverable’, which are ‘System’ and ‘Document’. This model is by no means complete and may include concepts such as models, modelling elements, code, subsystems, and the like. From the model, there are three types of ‘Document’ which are ‘Technical’ ‘Plan’ and ‘Report’.

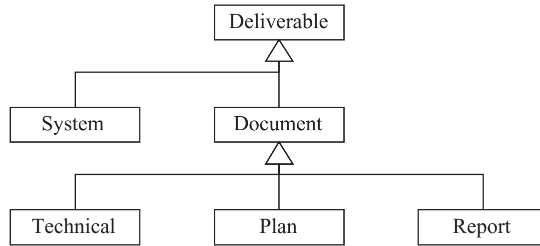


*Figure 6.25 Defining types of deliverable*

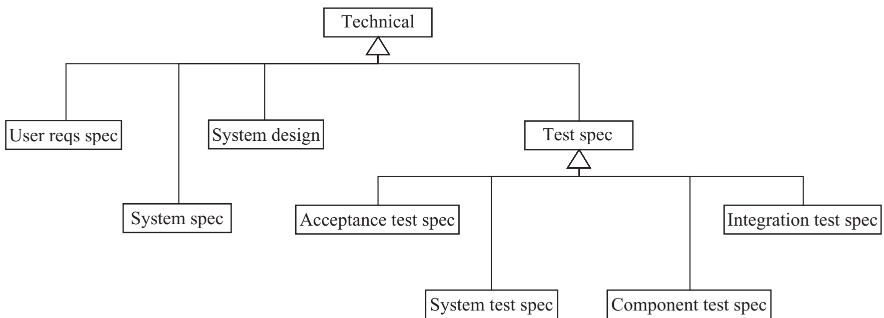


*Figure 6.26 Defining types of technical document*

The previous model may be further refined by defining types of 'Technical' document as shown in Figure 6.26. In this case, these types of technical documents are 'User reqs spec', 'System spec', 'System design' and 'Test spec'; 'Test spec' is further refined by the types 'Component test spec', 'System test spec', 'Integration test spec' and 'Acceptance test spec'.

This model may be expanded to include all types of document that will exist for a project. The models here have grouped the documents by type, but it is also useful to associate these with each process, which is, effectively, looking at them from a slightly different point of view.

### *6.6.2 Deliverables by process*

Deliverables may also be modelled by process. In fact, this has already been demonstrated once by the process descriptions, where deliverables were shown as attributes on the process class. This time, however, we shall show deliverables as classes and relate them directly to each process.

Figure 6.27 shows that 'Deliverable' has two types: 'Input' and 'Output'. There is one type of 'Input' defined as 'System spec' and four 'Output' defined as 'System design', 'System design review report', 'Integration test spec' and 'System test spec'.

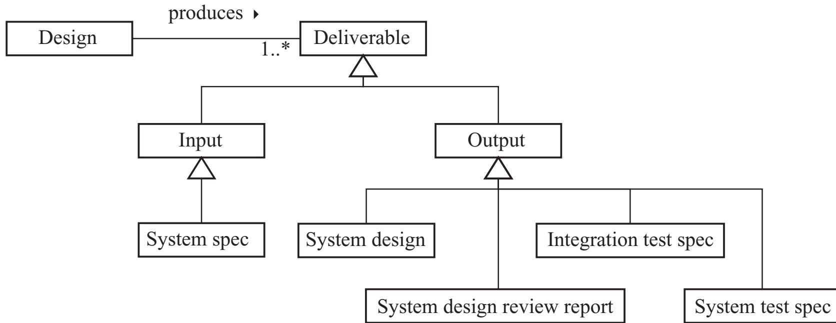


*Figure 6.27 Defining deliverables by process*

Note how these deliverables also appeared on the previous model but by creating this model we are adding information to the overall model. It is now known that not only is 'System design' an output deliverable, but also that it is a technical document.

It is possible to go into even more detail by modelling the structure of deliverables and this is discussed in Section 6.6.3.

### *6.6.3 Deliverables by structure*

So far, the types of deliverables have been modelled along with how these relate to each of the processes that have been defined. The actual structure, or contents, of each deliverable, however, has not been discussed. It is essential that a number of guidelines are given as to the layout of each deliverable so that they can be compared and traced in a constructive fashion.

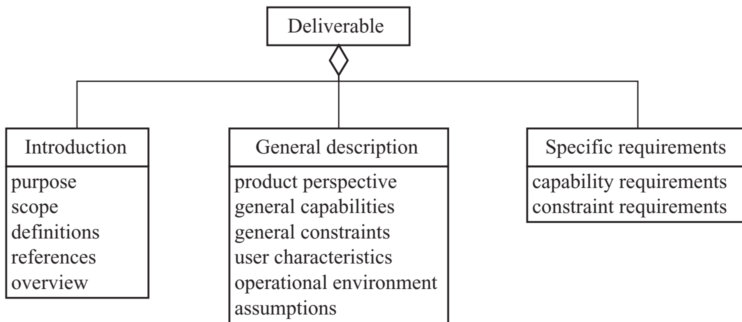


*Figure 6.28 Defining the structure of a deliverable*

Figure 6.28 shows the actual structure of a deliverable – in this case, the 'User requirement specification' – which is made up of 'Introduction', 'General description' and 'Specific requirements'. It is quite clear that these refer to sections of a document with the attributes representing subsections. This model becomes the template for the user requirements specification.

Modelling deliverables in this way may seem trivial, but patterns emerge when models are compared that allow such models to be used to define new deliverables. This particular pattern for the user requirements specification deliverable will be revisited in Chapter 7 when it is used in a practical way during the user requirements phase.

## 6.7 Implementing the systems engineering process using a tool

So far, the theoretical aspects of defining the process have been discussed, and the next step is to implement the processes in a real-life application. There is a great deal of information contained within a process model and implementation can be very tricky. Therefore, it should be possible to implement the process using a systems engineering tool since, frankly, we can all agree that tools exist in order to make life easier.

Before the requirements of tools are discussed, it is worth considering the nature of tools, if only for a moment. A tool is exactly that, a tool. A tool will not automatically make everything right and may not even make things more efficient. It is important to look at tools with your eyes wide open and to take a really serious look at what you want out of a tool before meeting with any vendors. Many tools will be hailed as a silver bullet, or panacea, to all your problems, but this is seldom, if ever, the case. However, as long as you know what tools can do and, sometimes more importantly, what they cannot do, they can be very powerful, save much time and effort, make life easier and make you work in a more efficient manner.

It is essential to choose a tool that will let you use your own defined processes and not just assume that the same process that the vendor uses or recommends is being used. One huge danger with many tools on the market is that they come with their own process that the tool follows. This is dangerous for a number of reasons:

- The tool is driving the process rather than the process driving the tool. Once the in-house process has been defined to reflect the way in which a particular organisation works, it is essential that this is followed, rather than some other, undefined and potentially unsuitable process that comes as part of a tool.
- By following a vendor's process, it means that the project or organisation is restricting itself to a single vendor, which is potentially expensive and, if the vendor goes bust, may pose a serious threat to the project.
- By tying in to a single vendor, it is not only the tool that will cost money, but also training and consultancy that will also be tied to a single vendor. Once you are committed to a single supplier, there is nothing to prevent the price of services from increasing dramatically.

Again, it is worth stressing that tools can be very good when used properly and for the right purpose. Anyone who has tried to hammer in a nail with anything other than a hammer (such as a pair of pliers, shoe, old house-brick or child's toy) will know that it often makes life far more difficult than walking to the tool box and getting the appropriate tool from the word go. In the same way, the tool for inserting nails is probably not appropriate for inserting windows. It is the same with systems

engineering tools. A systems engineering tool that claims to perform modelling, project management and life cycle definition may end up being a jack of all trades, yet a master of none. In addition, there is nothing wrong with using more than one tool from more than one vendor to carry out different tasks. Of course, it is helpful if they work together in some way, but with the convergence of computer styles and operating systems, this is becoming increasingly practical. For a more in-depth look at tool evaluation, see Chapter 11.

### *6.7.1 Uses for a tool*

There are many reasons for using a systems tool and the following list is a generic set of points that should apply to almost all projects:

- The tool will be used to implement the process that has been defined so far.
- The tool will be used to aid communication throughout a project by holding all project information in one place where it may be accessed by all project personnel.
- The tool will not only be used to store, but also to manage all project information in a controlled fashion.
- The tool will need traceability functions within it to ensure that all relationships between deliverables are maintained.
- The tool should be set up to hold templates for deliverables, according to the UML models, to make implementing the process as simple as possible.

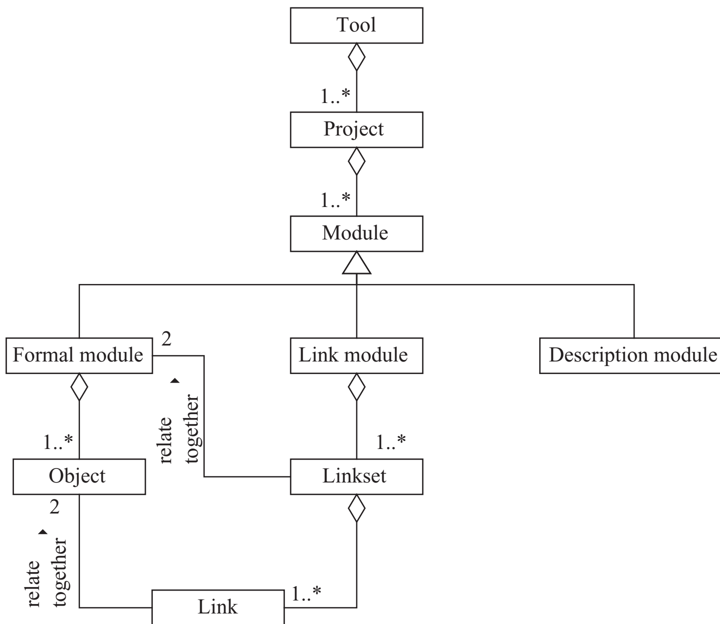


*Figure 6.29 Model of a popular systems engineering tool*

One way to assess a tool's performance at a high level and to ensure that the tool will work seamlessly with a defined process, is to model the actual tool using the UML. This may seem like a strange and useless thing to do, but it helps understand the tool and the model will provide the basis for the mapping between the theoretical process and the real-life implementation of the process on a project.

Figure 6.29 is a representation of a commercial tool that is in widespread use in today's industry. Although this is by no means a complete model of the functionality of the tool, this one diagram gives an excellent overview of the tool (useful for training) and provides vital information that is used later in this section to create the 'profile definition' and the 'implementation' model.

Figure 6.29 shows the UML diagram for a popular systems engineering tool. From the model, it can be seen that the 'Tool' is made up of one or more 'Project', each of which is made up of one or more 'Module'. There are three types of 'Module': 'Formal module', 'Link module' and 'Description module'.

Each 'Link module' is made up of one or more 'Linkset', each of which is made up of one or more 'Link'.

Each 'Formal module' is made up of one or more 'Object'. Two 'Object' are related together by one 'Link'.

This model does not show the complete functionality of the tool – in fact, it does not show any functionality of the tool! The model is, however, very useful for a number of reasons:

- It provides a very simple snapshot of the tool and shows the structure of the information in the tool.
- It is useful as a training aid to give a high-level view of the tool.
- It provides the basis for successful implementation of the process. If the process is represented using the UML and the tool is represented using the UML, it is possible to map between them and create something useful.

The key to successful implementation on the tool lies in defining the 'information model' and the 'implementation model'.

### *6.7.2 The information model*

The model that will actually be implemented on the tool is known as the 'implementation model'. This implementation model is based on a theoretical 'information model' that can be derived directly from the process models. The information model shows the deliverables from the process model and the relationships between them. These associations will, typically, represent traceability paths between the various deliverables, bearing in mind that traceability is a basic requirement of almost all processes and standards.

The relationships are very important here as they will determine how the different deliverables will be linked on the tool. For example, this will determine the traceability throughout the whole project, if the tool has traceability functions in it.

Figure 6.30 shows part of the information model for the process that has been defined so far. The information model shows that the 'User reqs spec' is traceable to

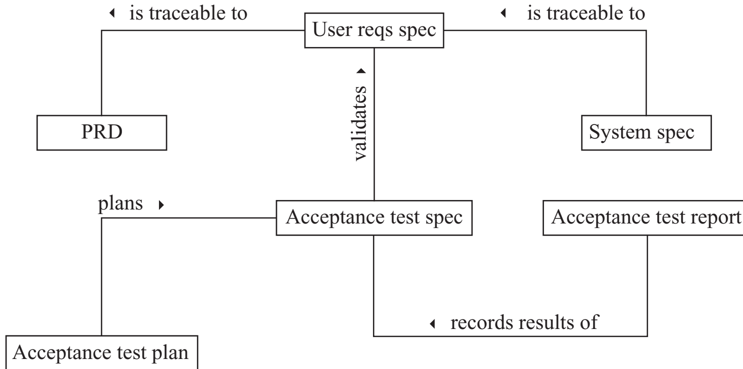


*Figure 6.30 Partial information model, based on the process that has been defined*

the 'PRD' (project requirements definition) and the 'System spec' is traceable to the 'User reqs spec'. The 'Acceptance test spec' validates the 'User reqs spec' and the 'Acceptance test plan' plans the 'Acceptance test spec', while the 'Acceptance test report' records the results of the 'Acceptance test spec'.

Note the use of association names that will relate to relationships within the tool such as 'is traceable to', 'plans', 'tests' and 'records results of'.

This model is taken from the deliverable models that were introduced in Figures 6.25 and 6.27. It provides a good consistency check between the information model and the defined process.

The model here is only a small section of the entire information model. The actual information model for the 'ENG' category alone consists of over 35 deliverables and 40 associations between them. The sheer size and complexity of the information model can potentially lead to many mistakes and errors, and thus it is important that it can be translated well and effectively to the tool in order to enable the tool to manage the data.

It is essential to understand that although this model is very useful, it is still very much theoretical and does not provide all the information required to navigate a real-life project. For example, there are no explicit references to modules in the tool that has been chosen, nor is there any way to differentiate between different versions of the same deliverable. This information will be contained in the 'implementation model'; however, the implementation model will be almost useless with no way to map to the model of the tool. This is achieved through the 'profile definition'.

### *6.7.3 The profile definition*

The 'profile definition' contains the key, or legend, as to how the information model will be implemented in real life using the implementation model. The profile definition provides a mapping between theory and real life: the information and implementation models.

In UML terms, the profile definition is a definition of stereotypes (or any other extensions – including constraints and tagged definitions) that will be used to make the

diagram more specific to a particular application – in this case, relating the implementation of the project to the process using the UML. Stereotypes and profile definitions are discussed in more detail in Chapter 8.

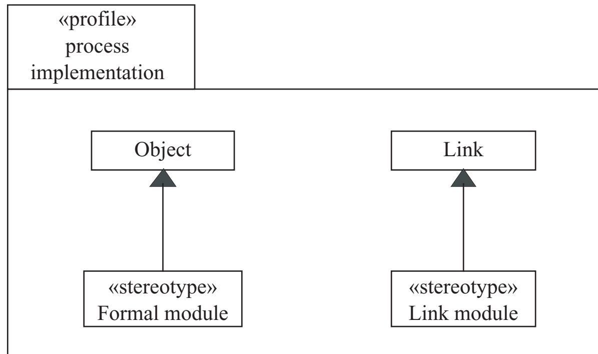


*Figure 6.31 Process implementation profile*

From Figure 6.31, it can be seen that two special types of class have been defined that will actually affect the UML meta-model, by defining special types (stereotypes) of basic UML entities. In this case, a stereotype of ‘Object’ has been defined that is called ‘Formal module’. In addition, a stereotype of ‘Link’ has been defined, which is called ‘Link module’. Note that the two terms chosen for the stereotypes are the same terms used in the tool model, which is to say that we are now using the tool’s vocabulary and introducing it into the UML using stereotypes.

This profile definition will now allow us to make sense of the ‘implementation’ model that will reflect real life on the project.

## *6.7.4 The implementation model*

The implementation model is the model that shows how the theory of the process, in the form of the information model, is represented in real life using the systems

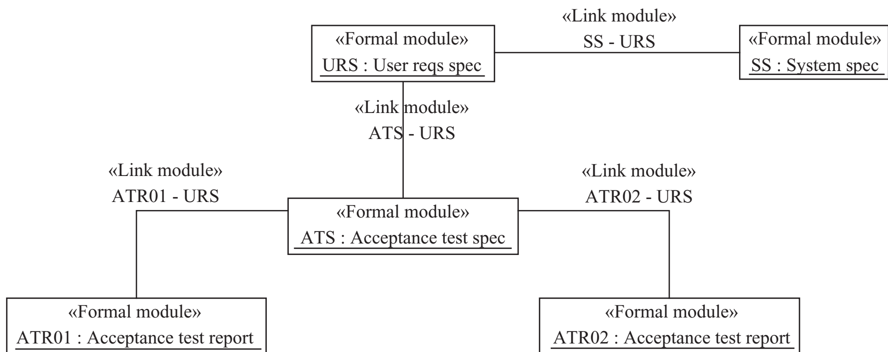


*Figure 6.32 Implementation model*

engineering tool. The implementation model uses an object diagram to show how real life actually exists, while still relating directly back to the defined process and using the tool's terminology.

Figure 6.32 shows the implementation model for the information model that was shown in Figure 6.31. Each object has the following features:

- *An object name*. This is exactly the same as the module names used on the tool. This gives a direct relation to the names of the files or modules used by the tool.
- *An associated class name for each object*. This is taken directly from the information model, to show the relationship to the process that has been defined using the UML.
- *A stereotype name*. This shows how the object is implemented on the tool – in this case, it shows whether the module is a formal module or a link module. This is taken directly from the profile definition.

The links in Figure 6.32 show the instances of the associations from the information model in Figure 6.31 and are described in the following way:

- *A link name*. This is the name of the instance of the association from the information model. The actual instance name is the same as the name of the module in the tool.
- *A stereotype name*. This shows how the link is implemented on the tool – in this case, the link happens to be a link module.

This implementation model may be used as a navigation aid to trace between the process definition and its actual implementation on the tool.

### *6.7.5 Summary*

This section has looked at how the process that has been defined so far in this chapter may be implemented using a basic off the shelf systems engineering tool.

We have now brought together the original theoretical model in the form of the information model (Figure 6.30), the model that bridges the theory and the tool in the form of the profile definition (Figure 6.31) and the actual model that is on the tool in the form of the implementation model (Figure 6.32).

## 6.8 Conclusions

In conclusion, therefore, this chapter covers the following points:

- Process analysis. The UML is used to analyse and understand existing processes in the form of international standards. By modelling in this way it is possible to see any inconsistencies both within a standard and between different standards.
- Modelling new processes. The same approach is applied to creating a new process in the form of a procedure that can be traced back to the international standards upon which it was based.

- A common approach. Note the diagrams that were used for both analysis and defining a new process. Structural models are realised by class diagrams and behavioural models realised by state machine diagrams, communication diagrams and activity diagrams. These are by no means the only diagrams that may be used because, as with all applications of the UML, any diagrams are accurate and appropriate.
- Deliverables. These are modelled by type, process and their structure to form templates.
- The process that is defined can be implemented using a basic systems engineering tool, which could also be modelled.

Finally, a point that cannot be stressed enough: the process must drive the tool, not the other way around. By following the approach suggested here, this is achievable.

## 6.9 Further discussion

1. Take the high-level process structure models from Figures 6.1 and 6.4 and map them on to Figure 6.15.
2. Map between the high-level process model from Figure 6.15 and those in Figures 6.10–6.14. Is it possible to map between all of them? If not, why not? What could be changed to make this mapping possible?
3. Take any process from the 'ENG' process and define a number of practices in the form of operations and deliverables in the form of attributes.
4. Define the statechart or activity diagram to describe the behaviour of this new process.
5. Define deliverables for this process by their type and structure.
6. Try to model a systems engineering tool with which you have experience. If you do not use one, try to model one based on any publicly available documentation.

## 6.10 References


1 ERIKSSON, H. E., and PENKER, M.: 'Business modelling with UML, business patterns at work' (John Wiley and Sons Inc, New York, 2000)

2 STEVENS, R., BROOK, P., JACKSON, K., and ARNOLD, S.: 'Systems engineering, coping with complexity' (Prentice Hall Europe, London, 1998)

3 EL EMAM, K., DROUIN, J. N., and MELO, W.: 'SPICE, the theory and practice of software process improvement and capability determination' (IEEE Computer Society, Washington, 1998)

4 LARMAN, G.: 'Applying UML and patterns – an introduction to object-oriented analysis and design' (Prentice Hall Inc., New Jersey, 1998)

5 KRUCHTEN, P.: 'The Rational Unified Process: an introduction' (Addison-Wesley Publishing, Massachusetts, 1999)

6 JACOBSON, I., BOOCH, G., and RUMBAUGH, J.: 'The unified software development process' (Addison Wesley, Massachusetts 1999)

# Chapter 7
# Modelling requirements

what you want and what you get are two different things

*Christine Holt*

## 7.1 Introduction

This chapter is concerned with modelling requirements. The diagram that is most frequently used for modelling requirements in the Unified Modelling Language (UML) is the use case diagram, although, technically speaking, any diagram may be used. In this section of the book, we will be concentrating on modelling requirements using use case diagrams. See References 1 to 3 for more discussion on use case models.

Use case diagrams, it may be argued, are perhaps one of the simplest of all diagrams, at least on the surface. However, they are perhaps the most misunderstood of all the UML diagrams. This is due in part to their inherent simplicity. Because the diagrams look so simple people often assume that very little effort is involved in generating the diagrams. This could not be further from the truth!

Requirements engineering is the discipline of engineering that is concerned with capturing, analysing and modelling requirements. In this book we are looking purely at modelling requirements with some degree of analysis. How these requirements are arrived at in the first place is entirely up to the engineer. Indeed, many different techniques for requirements capture will be mentioned, but will not be covered in any detail, as this is beyond the scope the book.

Before the UML can be related to requirements, it is important to understand some of the basic concepts behind requirements engineering. These concepts will be introduced at a high level before any mention of the UML is made. In this way, it is possible to set a common starting point from which to work with the UML. Newcomers to the field of requirements engineering should use this section of the book as an introduction and should consult some of the books referred to at the end of the chapter for more in-depth discussion about the whys and wherefores of requirements engineering. Experienced requirements engineers should treat this section as a brief

refresher, but it should be read at least once purely to determine the common starting point for the UML work.

## 7.2 Requirements engineering basics

### *7.2.1 Introduction*

This section introduces some fundamentals of requirements engineering. The emphasis in this chapter is on how to use the UML to visualise requirements rather than to preach about requirements engineering itself; this section is therefore kept as brief as possible, while covering the basics. The basic concepts that are covered here are all addressed practically in subsequent sections with respect to the UML.

Getting the user requirements right is crucial for any project for a number of reasons:

- The user requirements will drive the rest of the project, and all other models and information in the project should, therefore, be traceable back to its original requirement. If you look at a typical information model, such as the one discussed in Chapter 6, this can turn out to be a large amount of documentation, all of which will be driven by the requirements.
- The user requirements are the baseline against which all acceptance tests are defined and executed. Leading on from the previous point, the success of the project will be based on the project passing the acceptance tests, which rely entirely on the project meeting the original user requirements. It follows, therefore, that the success of the project is directly based on the user requirements.
- One of the definitions of quality that ISO uses is: 'conformance to requirements'. Therefore, requirements are absolutely crucial to achieving and then demonstrating quality.

One point that will emerge from this chapter is that much of the art of analysing requirements is about organising existing requirements so that they can be understood in a simple and efficient manner.

For a more in-depth discussion of requirements engineering in general, see References 4 to 6.

### *7.2.2 The requirements phase*

The requirements phase is, typically, the first phase of the systems engineering life cycle model, depending on the processes that are included in the life cycle. For example, it may be that some of the customer–supplier relationship processes are implemented in a phase before requirements, but, for argument's sake, let us assume that requirements is indeed the first phase of the life cycle. According to what is said in Chapter 6, the requirements phase should implement a defined process. The process that will be used in this section is taken directly from the worked examples used previously and the process is shown in Figure 7.1.

Figure 7.1 shows a class that is called 'Stakeholder requirements' and that represents a requirements process. This process is taken directly from the STUMPI

| Stakeholder requirements |
|---|
| Stakeholder model<br>Requirements model<br>Acceptance criteria definition<br>Requirements architecture<br>Review results |
| elicit requirements()<br>analyse requirements()<br>review()<br>define acceptance criteria()<br>identify stakeholders() |

*Figure 7.1 The stakeholder requirements process*

process model as discussed in Chapter 9 and uses the generic term ‘stakeholder’ to qualify the term ‘requirements’. Very often, the term ‘user requirements’ is used here, but the term ‘stakeholder’ is more accurate as a user is simply a special type of stakeholder.

Adopting the notation used in Chapter 6, it can be seen from the diagram that there are five artefacts associated with process (represented as attributes): ‘Stakeholder model’, ‘Requirements model’, ‘Acceptance criteria definition’, ‘Requirements architecture’ and ‘Review results’. Also, it can be seen that there are five main activities associated with the process (represented by operations), which are: ‘elicit requirements’, ‘analyse requirements’, ‘review’, ‘define acceptance criteria’ and ‘identity stakeholders’.

Now that the ‘what’ of the process has been defined, a logical next step is to define the ‘how’ of the process or, to put it another way, the behaviour of the process. The diagram chosen to represent the behaviour of the process is the activity diagram, which can be seen here.

The diagram in Figure 7.2 shows the behaviour of the stakeholder requirements process in the form of an activity diagram. Each of the activity executions represents an activity from the process and is the equivalent of an operation from the parent class being executed. The main flow of control can be seen here by looking at the activity executions (represented by the ‘sausage’ shape), their order and the conditions between them.

The flow of information around this process is shown by the object flow on the diagram, which provides an indication of which activity execution produces and/or consumes which objects. These objects show the process artefacts, which are represented on the parent class as attributes.

The swim lanes represent the roles, or stakeholders, which are responsible for various activities in the process. Any activity execution that falls within the boundary of a swim lane is the responsibility of the stakeholder named above the swim lane.

It should be stressed that the process shown here is simply an example and should be treated as such. The process itself is fully traceable back to ISO IEC 15288, which provides the model of best practice for this example. As an example of how this process may be tailored, many people will collect all the information shown here as attributes into a single ‘user requirements specification’ or something similar,

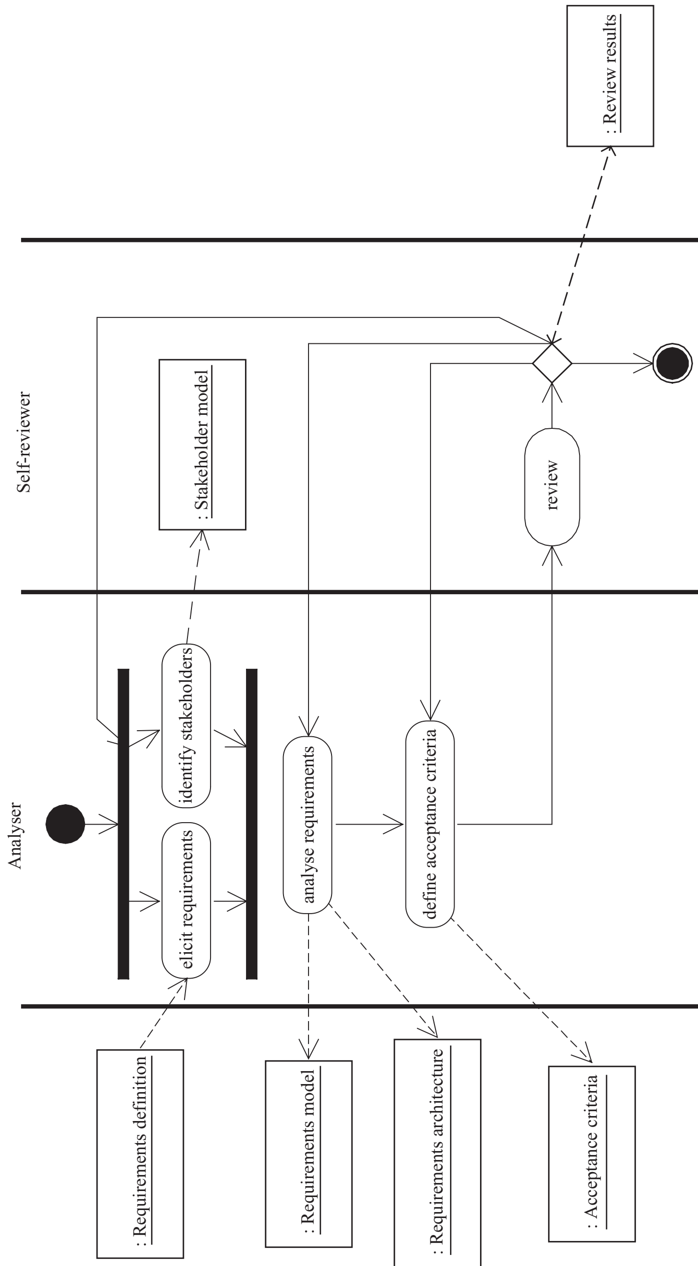


*Figure 7.2 Behaviour of stakeholder requirements process*

containing all the knowledge concerning the requirements of the system. Clearly, if this were the case, another attribute would be added.

### *7.2.3 Capturing requirements*

There are many ways to capture requirements, both formal and informal, and there is still much debate about which of the approaches yields the best requirements. Rather than enter into this debate, this following list will simply give an overview of the types of technique that may be adopted when capturing requirements.

*Interviews, both formal and informal*: Formal interviews follow a predefined format and have predefined questions that yield comparable results. Predefined forms may be used that define a set of questions and indeed even typical answers from which requirements may be drawn up. Informal interviews, on the other hand, start with a blank sheet of paper that is used to record information and comments during an informal discussion. These results are then analysed and from these, a set of requirements may be drawn up. Each of these has its own set of advantages and disadvantages. For example, formal interviews may be conducted in such a way that the questions given are quantifiable so that they may be directly compared or may be analysed statistically. On the other hand, although not so easily quantifiable, informal interviews may pick up requirements that may have been missed out by the formal questionnaire-style approach.

*Business requirements*: User requirements may be derived from business requirements. A business requirement is a very high-level requirement that drives the business rather than the product, yet may drive particular aspects of the project or set up some constraints. Business requirements are discussed in more detail in due course.

*Comparable systems*: Other systems with similar features may be looked at in order to capture their functionality, which may be transformed into user requirements. These other systems may be similar systems within the same organisation, legacy systems that may need to be updated or they may even be a competitor's product.

*Maintenance feedback*: The maintenance phase of a project should allow for customer feedback associated with a product. This is particularly important where the project is concerned with updating or replacing an existing system. The actual feedback may be in the form of error reports, customer wish-lists or periodic surveys of customers' opinions of a product.

*Working in the target environment*: Spending some time in the target environment can be invaluable when it comes to generating requirements. It may be that users of a system simply assume some functionality that may be overlooked when it comes to stating the system requirements. A fresh viewpoint from a nonspecialist may be exactly what is needed to draw out some obvious, yet crucial, system requirements.

*User groups*: Many companies with a large customer base set up special user groups and conferences where users can get together and exchange viewpoints and ideas. Although expensive, these meetings can prove invaluable for obtaining customers'

requirements. These meetings also demonstrate to the customer that the supplier actually cares about what customers think and has put some effort and investment into constantly improving the product that they already use.

*Formal studies*: Relevant studies such as journal and conference publications can often show up 'gaps' in products or provide a good comparison with other similar products. It is important, however, to assess how much detail the studies go into as some can be superficial. It is also important to find out the driving force behind studies, as any study carried out or financed by a product vendor may lead to distorted facts and, hence, any derived requirements may be suspect.

*Prototypes*: Prototypes give the customer some idea of what the final product will look like and hence can be useful to provide valuable early life cycle feedback. Indeed, this approach is so well promoted by some that it has led to the definition of a very popular life cycle model: the rapid prototype model. The use of prototypes is particularly tempting for systems with a large software content, or those that use a computer-based interface to control or operate a system. User interface packages are very simple (and cheap to come by) and can give potential users a perfect picture of what they will be getting at the end of a project. Non-software prototypes are also very useful but can be very expensive if physical models have to be constructed.

*User modifications*: These are not applicable to many systems, but some systems are designed so that they can be extended or modified by the customer. In order to relate this to a real-life example, think of a drawing or CAD package on a computer. Many of these packages allow the user to specify bespoke templates or drawing elements that may make the life of the user far simpler. Indeed, some UML tools currently on the market are basically drawing packages with predefined UML templates, and if getting the drawings looking good is a main requirement, this can often be the most economical solution to buying a CASE (computer aided/assisted software engineering) tool (see Chapter 11 for more information on selecting CASE tools). Whenever a modification is made, either by user or supplier, there is generally a good reason why it has been done. This reason will almost invariably lead to a new requirement for the system, which can be incorporated into subsequent releases of the system.

These are by no means a complete set of requirements capture techniques, but they give a good idea of the sheer diversity of techniques that have been adopted in the past. It may be that a completely different technique is more appropriate for your system – in which case use it!

Capturing requirements is all very well, but what in fact are requirements and what properties should they possess? The next section seeks to address some of these issues generally, before the UML is used to realise them.

### *7.2.4 Requirements*

#### 7.2.4.1 Overview

The previous section has introduced a number of techniques for capturing requirements, but there is more to requirements than meets the eye. For example, requirements are often oversimplified and one of the main reasons for this is because

there are actually three types of requirements. It has already been stated that much of the skill involved in modelling requirements is involved with organising requirements into logical and sensible categories. These three categories are dictated by the types of requirement that are shown in Figure 7.3.

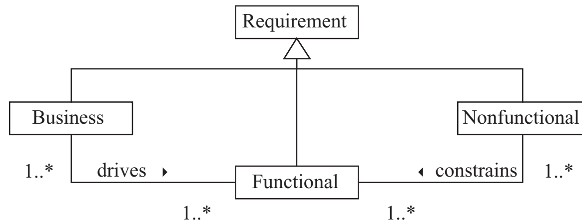


*Figure 7.3 Types of requirement*

Figure 7.3 shows that there are three types of ‘Requirement’: ‘Business’, ‘Functional’ and ‘Nonfunctional’. One or more ‘Business’ requirement drives one or more ‘Functional’ requirement, and one or more ‘Nonfunctional’ requirement constrains one or more ‘Functional’ requirement.

‘Functional’ requirements are what are thought of as traditional user requirements and ‘Nonfunctional’ requirements are often referred to as ‘constraints’ or ‘implementation requirements’. Each of these types of requirement is discussed in more detail in the following sections.

#### 7.2.4.2 Business requirements

The first type of requirement to be discussed is the business requirement. Business requirements, as the name implies, relate to the fundamental business of the organisation. Business requirements tend to be oriented more towards high-level business concerns, such as schedule and cost, rather than towards development itself and, as such, often fall outside the context of the system.

Business requirements and user requirements are often confused, but are not quite the same thing. Business requirements will relate to things that drive the business, such as ‘keep the customer happy’, ‘improve product quality’, etc. Because of this, business requirements are often implied, but rarely stated explicitly.

Business requirements will be aired from a point of view that ties it in to business processes. Chapter 6 introduces the subject of process modelling and shows that there were other concerns apart from engineering and, indeed, business requirements relate heavily to the ‘Organisational’ category, rather than ‘Engineering’.

Figure 7.4 shows a set of business requirements that have been generated for a systems engineering organisation, which may help to clear up some of these points.

Figure 7.4 is actually a use case diagram that is being used to describe the business context (discussed in more detail later) for a real-life systems engineering organisation. All the use cases on this model are actually representing business requirements, rather than functional requirements. Look at the nature of these requirements and it can be seen that they exist at a very high level conceptually. For example, ‘Promote excellence’ just happens to come directly from this organisation’s

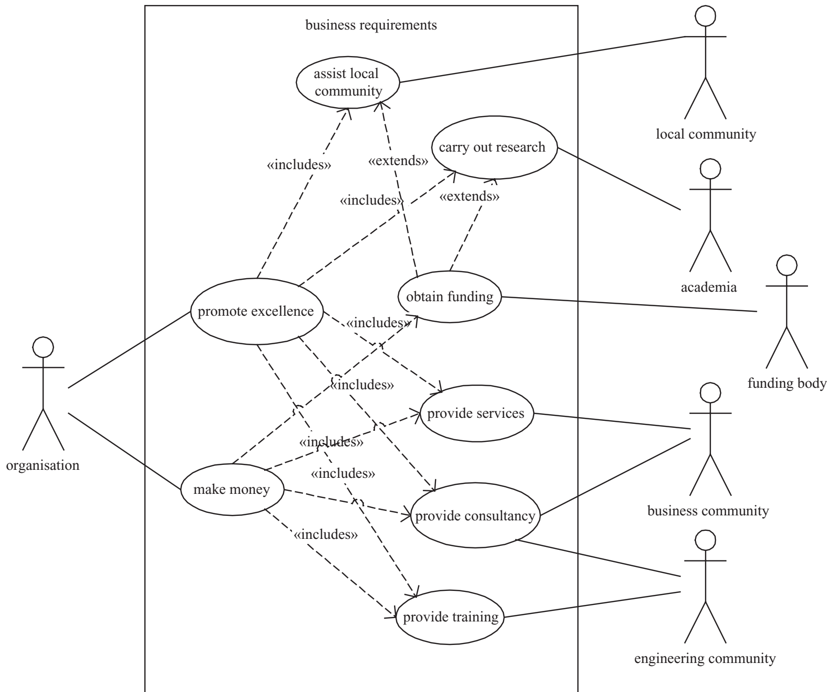


*Figure 7.4  Business requirements for a systems engineering organisation*

mission statement, which must be a business requirement for the organisation, otherwise they are not fulfilling their mission. The second high-level business requirement is to 'make money', which must be an underlying factor associated with almost every business in the world.

The 'make money' requirement has an association with four other requirements: 'obtain funding', 'provide services', 'provide consultancy' and 'provide training'. This relationship is an 'includes' relationship, which will be discussed in more detail later in this chapter, although its meaning is self-explanatory.

The 'promote excellence' requirement also has an association with several other requirements: 'assist local community', 'carry out research', 'provide services', 'provide consultancy' and 'provide training'.

The 'obtain funding' requirement 'extends' both 'carry out research' and 'assist local community', which is discussed in more detail later in this chapter.

The actors in this model, which are shown graphically by stick people, actually represent project stakeholders. Stakeholders represent any role, or set of roles, that has an interest in the project, and will be discussed in more detail later in this chapter.

The model that is used here is used throughout this chapter and will drive the functional requirements for projects within the organisation. Therefore, any projects that are used as examples from this organisation must be traceable back to the company's original business requirements.

#### 7.2.4.3 Functional requirements

Functional requirements are the typical user requirements for a system, in that they define the desired functionality of a system. User requirements should have some observable effect on a particular user of a system, otherwise they are possibly being defined at too low a level.

One of the relationships established in Figure 7.3 is that 'Business requirements' drives 'Functional requirements'. This means that all functional requirements should, in some way, be traceable back to the organisation's business requirements. If this is not the case, perhaps it should be questioned why this requirement is necessary.

#### 7.2.4.4 Nonfunctional requirements

The third and final type of requirement introduced in Figure 7.3 is the 'Nonfunctional' type of requirement, which is also sometimes referred to as 'Implementation requirement' or 'Constraint'. These 'Nonfunctional' requirements constrain functional requirements, which means that they may limit functional requirements in some way. Examples of nonfunctional requirements include:

*Quality issues*: A common nonfunctional requirement is 'comply with standard', where the word 'standard' may refer to some sort of established norm, such as an international standard or process. This is particularly relevant when read with Chapter 6, which discusses the importance of standards compliance. This may be a fundamental requirement of the system and may make all the difference between meeting acceptance tests and failing them.

*Implementation issues*: It may be desirable to use a particular technique or language to implement the final system. As an example of this, imagine a system whose target environment may contain other systems and which will require a particular platform or protocol to be used.

*Life cycle issues*: This may include the way in which a project is carried out. It may be that a particular project management technique should be used. An example of this is the PRINCE system, which is often cited in government projects as an essential part of the project. This may also include other constraints, such as design techniques and methodologies and even the modelling language (such as the UML!).

It is essential that these nonfunctional requirements are treated in the same way as both functional and business requirements, and that they exist on the requirements models.

#### 7.2.4.5 Properties of requirements

Once requirements have been identified, it is important that they are classified in some way so that their status may be assessed at any point in the project and they may be organised and manipulated properly. In order to classify requirements, they often have their features, or attributes, defined [7–9]. This may be represented graphically by adding, unsurprisingly, attributes to a class that represents a requirement. Figure 7.5 shows such a class with some example attributes added.

| Requirement |
| --- |
| source<br>priority<br>V&V criteria<br>ownership<br>absolute ref |

*Figure 7.5 Properties or a requirement shown as attributes*

Figure 7.5 shows a class that represents a requirement with several defined attributes describing the properties of a generic requirement. These requirements would be inherited by each of the three types of requirement and are particularly useful when it comes to implementing a project using a tool, which is discussed in more detail in Chapter 6.

The attributes chosen here represent typical properties that may be useful during a project; however, the list is by no means exhaustive. The attributes currently shown are:

- 'source', which represents where the requirement was originated. This is particularly useful when there are many source documents that the requirements have been drawn from, as it may reference the role of the person who asked for it.
- 'priority', which will give an indication of the order in which the requirements should be addressed. A typical selection of values for this attribute may be 'essential', 'desirable' and 'optional'. This is similar to the concept of 'enumerated types' in software, where the possible values for an attribute are predefined, rather than allowing a user to enter any text whatsoever. This is important, as in many cases it is simply not possible to meet all requirements in a project, and thus they are prioritised in terms of which are essential for the delivery of the system. This also takes into account the fact that many user requirements are user wishes that are no more than 'bells and whistles' and do not affect the core functionality of the system. There is often quite a large difference between what the customer wants and what the customer needs. Although both are valid requirements, it is important that the customer's needs are addressed before their wants so that a minimum working system can be delivered.
- 'verification/validation criteria', which gives a brief idea of how compliance with the requirement may be demonstrated. This may be split into two different attributes or may be represented as one, as shown here. It is crucial that there is some clear way to establish whether a requirement may be verified (it works properly) and validated (that it has been met) as the validation will form the basis for the customer accepting the system once it has been delivered. This is high-level information and not a test specification, for example.
- 'ownership', which will be related directly to the responsible stakeholder in the system. If the requirement is not owned by one of the defined stakeholders, there is something wrong with the stakeholder model. The stakeholder model is discussed later in this chapter.

- 'absolute reference', which represents a unique identifier by which the requirement may be identified and located at any point during the project. This also forms the basis for any traceability that may be established during the project to show, for example, how any part of any deliverable during any phase of the project may be traced back to its driving requirement from the user requirement specification.

Many other attributes may be defined depending on the type of project that is being undertaken. Examples of other requirement attributes include: urgency, performance and stability.

It is important that requirements are written in a concise and coherent way in order to avoid:

*Irrelevancies*: It is very easy to put in too much background information with each requirement description, so that the focus is lost amidst irrelevant issues.

*Duplication*: It is important that each requirement is only stated once. This is particularly important when requirements affect other requirements, such as constraints, which are ripe for duplication.

*Overdescription*: Verbosity should be avoided wherever possible.

One of the keys to successful requirements engineering is the way in which the requirements are organised. Much of this chapter will deal with organising the models of requirements into a coherent set of usable requirements.

### *7.2.5 Stakeholders*

#### 7.2.5.1 Overview

This section examines another fundamental aspect of requirements modelling, which is that of stakeholders. A stakeholder represents the role, or set of roles, anyone or anything that has a vested interest in the project, ranging from an end user to shareholders in an organisation, or the government of a particular nation.

It is very important to identify stakeholders early on in the project as they can be used to relate to any type of requirement and can also be used to define consistent responsibilities.

#### 7.2.5.2 Types of stakeholder

One of the practices that was identified in the requirements process was that of 'identify stakeholders'. This is often written as 'identify users' or 'identify roles' depending on which source is used for reference. The reason why these terms are often used interchangeably is that a 'user' is actually a type of stakeholder. A user will actually use the output of the project, the product, and may be thought of as having an interest in the project. Figure 7.6 shows this relationship from a UML point of view.

The relationship between stakeholders, users and the project are shown in Figure 7.6.

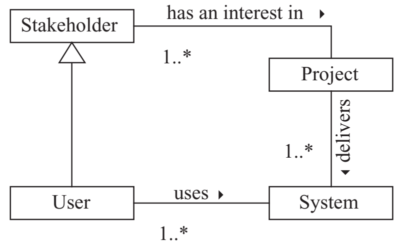


*Figure 7.6 Relationship between users and stakeholders*

From the model, 'User' is a type of 'Stakeholder'. One or more 'User' uses the 'System', while one or more 'Stakeholder' have an interest in a 'Project'. In addition, the 'Project' delivers one or more 'System'.

Although it is impossible to list all possible stakeholders in an organisation, it is possible to draw up a generic model that is very useful as a starting point for a stakeholder model. Figure 7.7 shows such a generic model that will not fit all projects or organisations, but is very useful as a starting point for identifying any potential stakeholders.

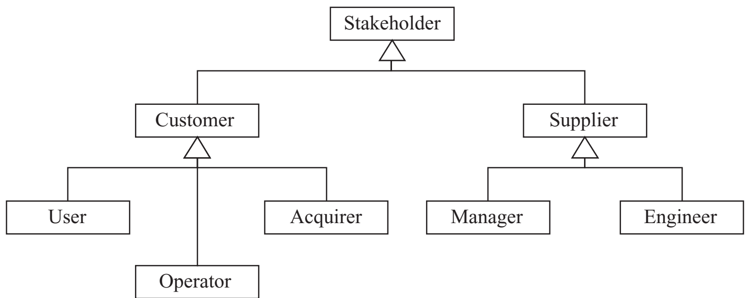


*Figure 7.7 Types of stakeholder*

It can be seen that there are two main types of 'Stakeholder': 'Customer' and 'Supplier'. These are chosen as the highest-level stakeholders as almost every project will have both a supplier and a customer. It should be pointed out that the name of the class that represents each stakeholder represents a role that is being performed, rather than the name of an individual or particular organisation. It may be that the 'Customer' and 'Supplier' stakeholders are actually within the same organisation, or may even be the same person in the case of one person creating a system for personal use.

The 'Customer' stakeholder has three main types: 'User', 'Operator' and 'Acquirer'. The 'User' will be the role that represents the end user of the system – for example, in the case of a transport system such as rail or air, the 'User' role would be fulfilled by the actual passengers. The role of the 'Operator' in the same example would be taken by the staff who actually work at the stations or ports where the transport is taking place and the people who drive or fly the vehicles. The 'Acquirer'

role represents whoever is responsible for paying for the project, which may be a large organisation such as an airline or rail company in the example.

The second main type of ‘Stakeholder’ is the ‘Supplier’, which has two main types: ‘Manager’ and ‘Engineer’. Again, these are generic roles that should be used as a starting point for the real stakeholder model.

In order to take the concept of stakeholders further, let us consider an example that fits in with the business requirements that were shown in Figure 7.4, with regard to the ‘provide training’ business requirement. The idea here is to identify the stakeholders for this requirement by using the model shown in Figure 7.4 as a starting point for the reasoning.

Imagine a situation where a client organisation has asked the organisation whose business requirements have been defined to provide a training course at the client’s premises. Therefore, a project is started that, for the sake of this example, will be referred to as the ‘provide training course’ project. The stakeholder model has been identified as shown in Figure 7.8.

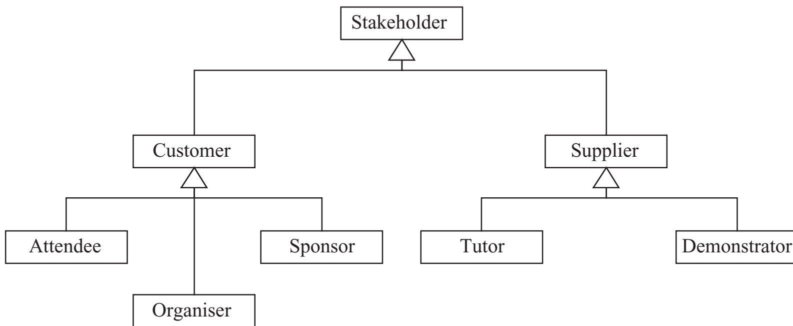


*Figure 7.8 Example of stakeholder model for ‘provide training requirement’*

The model in Figure 7.8 shows some stakeholders who have been identified for the ‘provide training’ business requirement. The two main types of ‘Stakeholder’ remain the same as ‘Customer’ and ‘Supplier’, because in order to deliver a training course there must be someone who wants the course, the ‘Customer’, and someone who can supply it, the ‘Supplier’.

The ‘Customer’ stakeholder has been split into three types in this example:

- The ‘Organiser’, whose role it is to set up the logistics of the course and ensure that there are enough attendees to justify the expense of the course.
- The ‘Attendee’ represents the people who will be attending the course.
- The ‘Sponsor’ represents the role that actually pays for the course.

In the same way, the ‘Supplier’ stakeholder has two types that differ from the original generic model. These two types are:

- The ‘Tutor’, who is the person or team of people who teach the course.
- The ‘Demonstrator’, who is the person or team of people that provide practical demonstrations that complement the tutor’s teaching efforts.

It is useful to remember that the stakeholders represent the roles, rather than individuals, involved in the project. There are a number of main reasons for this:

- There is nothing to say that each role has to be a separate individual or entity, as it is possible for one person to take on more than one role. It is important that the roles are considered rather than the individual, as it may be that the system is split up with regard to the functionality attached to each role, rather than to a person.
- If an individual leaves a project and has a number of roles, it may be that more than one person can replace the initial individual. It may also occur that somebody moves position within a project, maybe up the management hierarchy, so it is absolutely crucial that the roles associated with that person are not confused with any new roles. If all associations are with an individual, rather than a role, this will become confusing, to the possible detriment of the project.
- One person may take on two roles. For example, the role of the 'Tutor' and the 'Demonstrator' may be taken on by one person for one instance of the course and this may change for another instance of the course. It is important, therefore, to know the skills and specialist knowledge required by the role so that if a person who takes on both roles needs to be replaced, it can be ensured that the replacement can fulfil both roles.

This model was based on the information in Figure 7.4, but it was tailored for this particular project.

### 7.2.6 Summary

The information that has been introduced and discussed so far in this chapter may be summarised by the UML model shown in Figure 7.9.

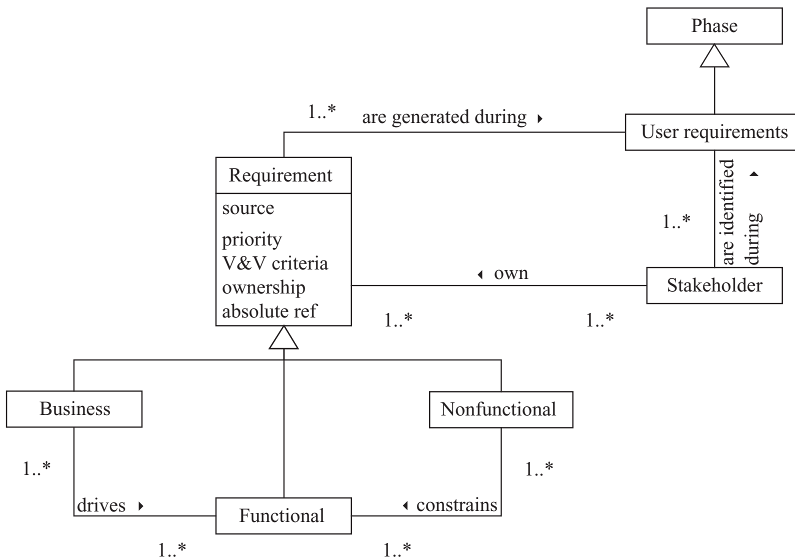


*Figure 7.9 Summary so far*

In summary, therefore:

One or more 'Requirement' are generated during 'User requirements', which is a type of 'Phase'. One or more 'Stakeholder' are identified during 'User requirements' and one or more 'Stakeholder' own one or more 'Requirement'. Each 'Requirement' has the attributes 'source', 'priority', 'V&V criteria', 'ownership' and 'absolute ref'.

There are three types of 'Requirement': 'Business', 'Functional' and 'Nonfunctional'. One or more 'Nonfunctional' requirements constrain one or more 'Functional' requirements. One or more 'Business' requirements drive one or more 'Functional' requirements.

This covers the basic concepts associated with requirements engineering. These concepts will now be related to UML issues in section 7.3.

## 7.3 Using use case diagrams (usefully)

Now that the basics of requirements engineering have been introduced, albeit briefly, it is useful to now take another look at the UML diagram that will be used for most of the requirements engineering models: the use case diagram. By way of a recap, the meta-model for use case diagrams is shown in Figure 7.10.

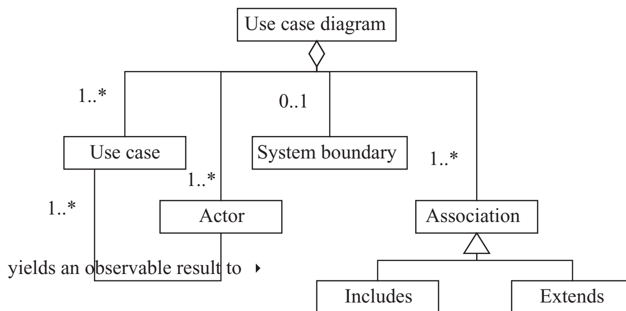


*Figure 7.10 Partial meta-model for use case diagrams*

Figure 7.10 shows the partial meta-model for use case diagrams, which is first introduced in Chapter 5. It can be seen that a 'Use case diagram' is made up of one or more 'Use case', one or more 'Actor', one or more 'Association' and zero or one 'System boundary'. There are two special types of 'Association': 'Extends' and 'Includes'. A 'Use case' yields an observable result to one or more 'Actor'.

## 7.4 Context modelling

### *7.4.1 Types of context*

Several types of requirements have been introduced and the relationship between them defined. However, in reality, it is useful to separate these types of requirements

and relate them to different types of stakeholder. This organisation of requirements results in defining the 'context' of a system.

The context of a system represents what the system is and its boundaries from a particular viewpoint. This is absolutely critical, as it is possible to have many contexts that relate to a single system, but that exist from different points of view. For example, the system context for a customer may be different from the system context of an operator. Something that is perceived as lying outside the context of one system may be included when looked at from a slightly different angle. Examples of this will be given in due course.

The context of the system shows the system boundary and any people or peripherals that communicate with the system. This may include 'who' interacts with the system and also 'what' interacts with the system. It should be noted that users are not necessarily people and may be peripherals, such as computers, hardware, databases, and so on, or, indeed, another system. In terms of what has been shown so far, these entities that interact with the system are equivalent to stakeholders. This means that if all stakeholders have already been identified correctly, they can be used to help model the context of the project.

Two types of context exist: the 'business context' and the 'system context'. Each context is defined by the system boundary, which is one of the basic elements of the use case diagram. The system boundary is actually a very interesting piece of syntax for the UML as it is either massively misused or underused. Many UML texts, tools and references ignore the system boundary altogether or simply state that it is optional. If the system boundary was optional, with no guidelines issued for when it should and should not be used, what would be the point of it in the first place? An excellent use for the system boundary element in the UML is to define a context in a system. In a nutshell, if a use case diagram has a system boundary, it is a context diagram. If it does not have a system boundary, it is simply showing the breakdown of requirements.

Each of the two types of context will now be looked at in some detail and the example used so far will be taken further and, in some cases, revisited with a new reflection on matters.

#### 7.4.1.1 Business context

The business context shows the business requirements of an organisation or a particular project. The business requirements that are shown in Figure 7.4 are really a very high-level business context that sets the context for the whole organisation. It may be that new, additional business contexts are generated for each project, but it is absolutely crucial that these are consistent with the high-level business requirements, or context, otherwise the company will be straying from its original mission.

As a general rule of thumb, business requirements will exist within a business context, rather than nonfunctional or functional requirements. In the same vein, the roles that exist outside the context of the system are generally high-level stakeholders, rather than actual users of the system. The use case diagram shows a generic business context that highlights these points.

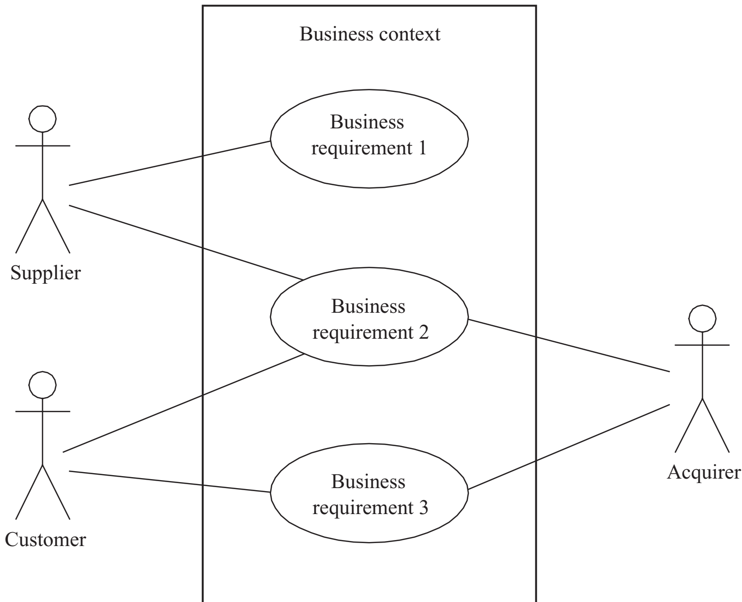


*Figure 7.11 Generic business context*

It can be seen from the model in Figure 7.11 that the various stakeholders – ‘Customer’, ‘Manager’ and ‘Acquirer’ – are associated with various business requirements: ‘Business requirement 1’, ‘Business requirement 2’ and ‘Business requirement 3’.

If this is related back to the original business requirements for the systems engineering organisation in Figure 7.11, it can be seen that these rules of thumb hold true. All the actors in the diagram are stakeholders rather than users and all the use cases are business requirements.

Note the explicit use of the UML entity, the system boundary, which shows that this model is a context model rather than a straight requirements model. The system boundary is represented very simply by a rectangular box that goes around the requirements (shown as use cases) in the context and keeps out the stakeholders (shown as actors).

The second type of context is the ‘system context’, which is discussed in section 7.4.1.2, before a full example is worked through, showing how to differentiate between the two types of contexts and offering advice on how to practically create both.

#### 7.4.1.2 System context

The system context relates directly to the project at hand and is concerned with the user requirements and constraints on them, rather than with the high-level business requirements. The stakeholders that exist as actors on the use case diagram are

typically the users and lower-level stakeholders, such as people who will be involved directly with the day to day running of the project – in other words, managers and engineers.

As a general rule of thumb, therefore, the actors on the system context will be users and low-level stakeholders and the use cases will be functional and nonfunctional requirements. Figure 7.12 shows a generic system context that represents these rules of thumb.

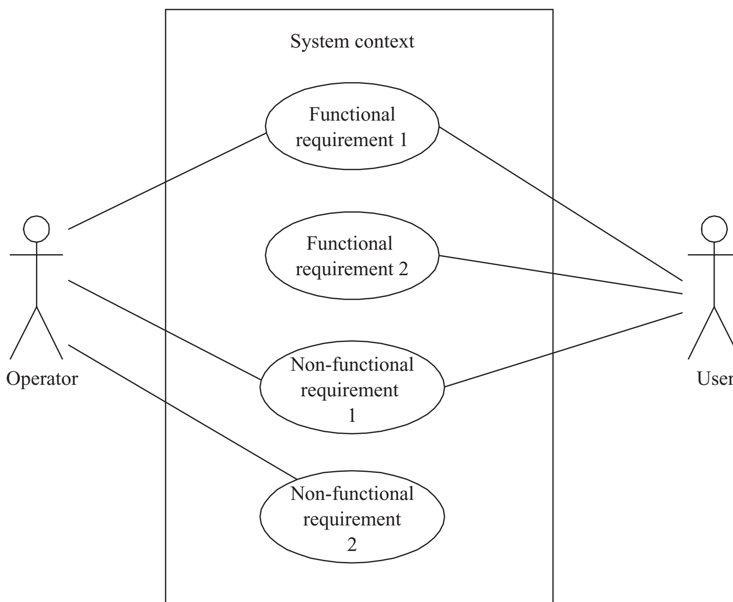


*Figure 7.12 Generic system context*

Figure 7.12 shows that the system context diagram consists of various types of functional requirement – ‘Functional requirement 1’ and ‘Functional requirement 2’ – and various types of nonfunctional requirement – ‘Nonfunctional requirement 1’ and ‘Nonfunctional requirement 2’.

Now that the basic rules of thumb have been established, it is possible to apply these to the example that is used earlier in this chapter.

### *7.4.2 Practical context modelling*

Consider now an example where the systems engineering organisation modelled previously in Figure 7.4 wants to start up a project to run training courses. This was introduced previously and, indeed, a set of stakeholders was identified. This example will now be taken one step further and the business and system contexts will be created for the ‘provide training course’ project. The first question that should be asked is whether or not the project is consistent with the organisation’s mission. In this case there is an obvious connection, as one of the business requirements for the organisation is to ‘provide training’. If there is no clear connection to the business requirements

of the organisation then one of two things must be done:

- The business requirements of the organisation must be changed to reflect the new project. If this is not done, the project does not fit in within the defined scope of the company's activities.
- If it is the case where the project is outside the scope of the company's current activities and the company is not willing to add it to its list of business requirements, the answer is quite clear: the project should not be taken on in the first place!

The first step is to ensure that the project can be proven to be consistent with the organisation's business requirements and then the next step is to identify the requirements for the project and the stakeholders. This will, in reality, be an iterative process and will take several iterations to get right.

As a first pass, let us try to identify some general requirements, regardless of type. These requirements were generated by simply asking a group of stakeholders what they perceived the requirements to be. At this stage, the requirements were not categorised, but were simply written down as a list. The list below shows a typical set of requirements that is generated when this example is used to illustrate context modelling:

- 'ensure quality', which refers to the work that is carried out by the supplier organisation.
- 'promote supplier organisation', in terms of advertising what other services the organisation can offer and hope for some repeat business or future business.
- 'improve course', which means that any lessons learned from delivering the course should be fed back into the organisation in order to constantly improve courses as time goes on.
- 'teach new skills', from the point of view of the supplier organisation.
- 'provide value', so that the clients will keep coming back for more business.
- 'organise course', which involves setting up, advertising etc.
- 'deliver course', which consists of the delivery of the course, in terms of teaching and practical demonstrations.

The next step, once the initial set of requirements has been identified, is to categorise each requirement by deciding whether it is a business requirement, a functional requirement or a nonfunctional requirement. The list of business requirements is:

- 'promote supplier organisation', which is clearly aimed at the business level and will relate directly back to the original business requirements of the organisation.
- 'provide value', which relates directly back to the supplier's business requirements.

The list of functional requirements is:

- 'teach new skills', which is what is expected to come out of the course.
- 'deliver course', which involves somebody turning up and teaching the course and providing course materials.

- ‘organise course’, which involves publicising the course, setting up the course and providing support for the course.

The list of nonfunctional requirements is:

- ‘ensure quality’, which means that the quality of the course must be maintained. In reality, this may equate to giving out course assessment forms and feeding them back into the system.
- ‘improve quality’, which relates to how the course may be improved constantly in order to meet one of the original business requirements of ‘promote excellence’.

It may be argued that these two nonfunctional requirements are one and the same and, hence, would be condensed into a single requirement. It may also be argued that they are separate, distinct requirements, and that they should remain as two requirements. The problem is, which of the two is correct? As there is no simple answer to this, there is a particular rule of thumb that should be applied under such circumstances. If the answer to whether to split up or condense requirements is unclear, keep them as two. This is because experience has shown that it is far easier to condense two requirements at a later time than it is to split a single requirement in two.

Now that a number of requirements have been identified, what about the stakeholders? The list of stakeholders has already been identified in a previous section and can thus be reused here. First of all, consider the business context and apply the rules of thumb discussed previously. The resulting business context is shown in Figure 7.13.

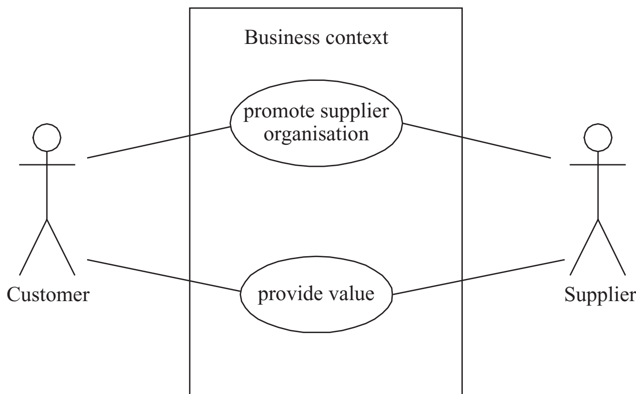


*Figure 7.13 Business context for the training courses example*

Figure 7.13 shows the business context for organising training courses. Note that the actors here are high-level stakeholders and that the use cases are business requirements. This model should be traceable directly back to the organisation’s business requirements model. It could be argued that both use cases relate directly to both the ‘provide value’ and ‘promote excellence’ business requirements. Note also that the actors in this model are high-level stakeholders from the stakeholder

model that was defined in Figure 7.8, which maintains the rule concerning high-level stakeholders and business contexts.

The next step is to create the system context, which should, hopefully, make use of the remaining requirements (both functional and nonfunctional) and the remaining stakeholders (user-level stakeholders). The resulting system context is shown in Figure 7.14.

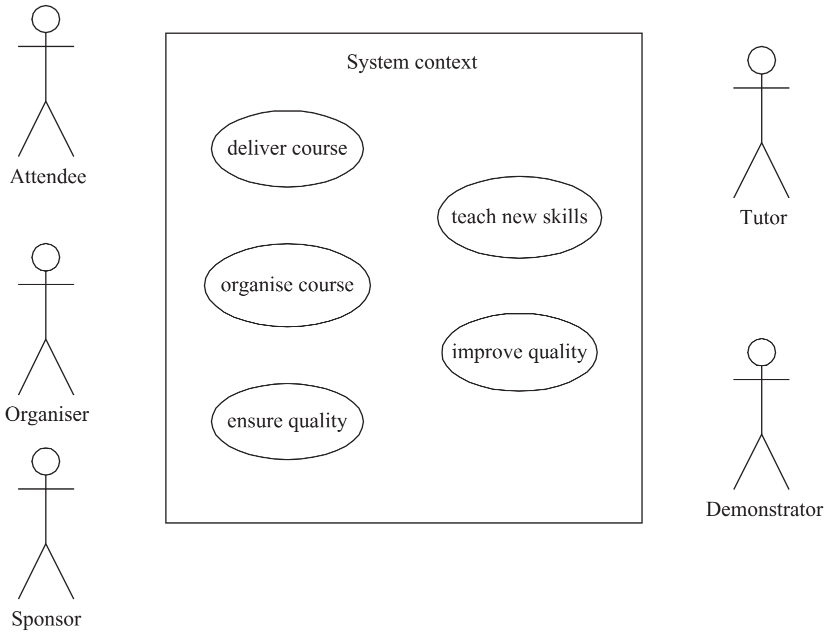


*Figure 7.14 System context for the training courses example*

Figure 7.14 shows the system context for the training courses project. This should be concerned with the nuts and bolts activities of actually running the course, rather than the high-level business requirements that appear on the business context.

The actors included on this model are actually the same as the stakeholders that were identified back in Figure 7.1. By using the use case diagrams in conjunction with the original stakeholder model (realised by the class diagram) it is possible to build up some consistency between the models, even at this early point in analysing the project. One of the problems with using use case diagrams very early in the project is that there is very little to relate them to. By relating the use cases to an associated class diagram, all diagrams become more consistent and the connection to reality gets stronger. Relationships between actors and use cases have been omitted from this model as they are discussed in more detail in due course.

So far, it has been possible to generate both the business context and the system context for a new project. The system context for the project is also traceable to the business context for the project. Note that both of these contexts are related back to

the business requirements of the organisation, which is very important when it comes to establishing a business case for the new project.

### *7.4.3 Summary*

The work covered so far in this chapter may be summarised in Figure 7.15 UML diagram.

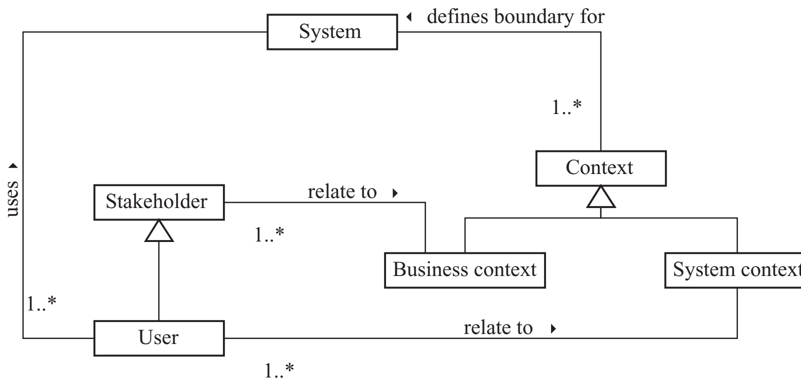


*Figure 7.15 Relationship between context and system*

In summary, Figure 7.15 can be shown to state that one or more ‘Stakeholder’ relate to a ‘Business context’ and one or more ‘User’ relate to a ‘System context’. The ‘Context’ (of which there are two types) defines the boundaries of the ‘System’ and one or more ‘Stakeholder’ are identified for the ‘System’.

## 7.5 Requirements modelling

### *7.5.1 Introduction*

Section 7.4 was concerned with modelling the business and system contexts of the system. These contexts actually represent requirements but at a very high level. For any project, the next step would be to look at these requirements in more detail and see if they can be broken down, or decomposed, into more detailed requirements. It should be pointed out that there is nothing to say that use case diagrams have to be used for requirements modelling. Indeed, in the days before UML when people used other techniques, such as Rumbaugh’s object modelling technique (OMT), class diagrams were used to model requirements. As with all aspects of the UML, you should use whatever diagrams you feel most comfortable with. It just so happens that use case diagrams were created specifically to model requirements and, therefore, tend to be the preferred technique. In addition, this is the approach that was advocated by the old Objectory process that has since evolved into the Rational Unified Process (RUP).

### *7.5.2 Modelling requirements*

The starting point for modelling the requirements of a system is to take the system context and use the high-level requirements contained therein. This gives an indication of what the overall functionality of the system should be. The most natural thing to do is to take each requirement and to decompose it, or break it down into smaller or lower-level requirements.

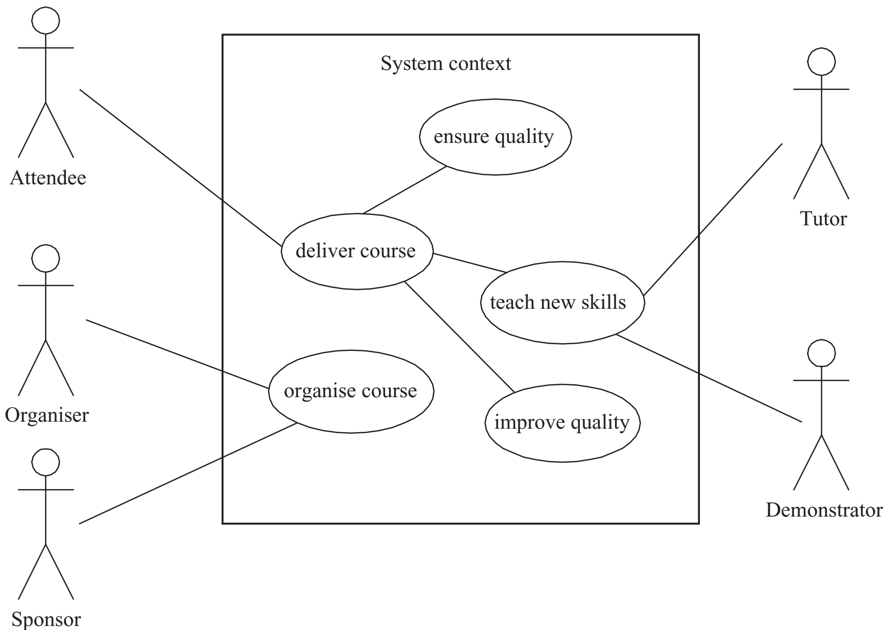


*Figure 7.16 The system context with initial relationships*

Figure 7.16 shows the systems context as generated in the previous section, except that the relationships have now been added in. Types of relationship are discussed in some detail later in this chapter and this model is revisited as part of the further discussions.

The first point discussed, however, is that of decomposing high-level requirements into lower-level requirements.

The use case that will be taken and used for this example is that of 'organise course'. When thinking about how to decompose this requirement, it is very tempting to overdecompose. This can lead to all sorts of problems as it is quite easy to accidentally go too far and to end up taking a dataflow diagram approach where the system is decomposed to such a level as to solve the problem, rather than state the requirements. This is compounded even further as use cases actually look a little like processes from dataflow diagrams. Indeed, dataflow diagrams are also used to define the context of a system, as are use case diagrams!

It is important, therefore, to know when to stop decomposing use cases. In order to understand when to stop, it is worth revisiting the definition of a use case, which states that a use case must have an observable effect on an actor. Therefore, consider each use case and then ask whether it actually has an effect on an associated actor. If it does not, it is probably not appropriate as a use case.

Another good way to assess whether the use case is at an appropriate level of functionality is to assess whether or not scenarios may apply directly to the use case. Remember again that a scenario is an instance of a use case and thus look to see if any scenarios are directly applicable to that use case. If no clear scenarios are apparent, it may be that the use case is too high level and needs to be decomposed somewhat. If there is only one scenario per use case, perhaps each one is too specific and the use case needs to be more generic. Scenarios will be discussed in more detail in due course.

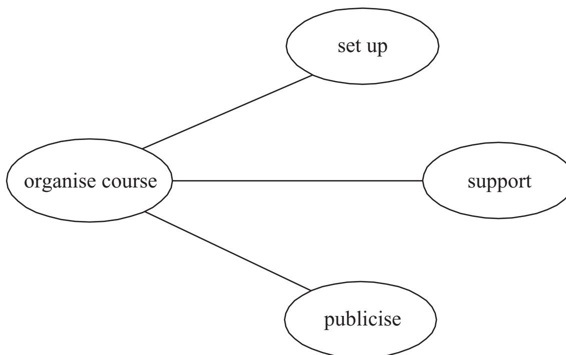


*Figure 7.17 Simple decomposition of an 'organise course' requirement*

Taking the example one step further, Figure 7.17 shows how the 'organise course' use case can be decomposed into lower-level use cases.

The model here is derived from the model in Figure 7.16, by selecting a single use case and decomposing its functionality. It can be seen now that the 'organise course' use case has been decomposed into three use cases: 'set up', 'support' and 'publicise'.

When this model is considered, imagine that one of the team also comes up with a few new use cases that are also applicable to the decomposed model, but that have slightly different relationships compared with the 'decomposed into' style relationship that is shown in this model. The three new use cases are 'cancel course', 'organise in-house course' and 'organise external course', as shown in Figure 7.18, with their relationships to other use cases indicated by simple associations.

Figure 7.18 shows the updated use case model that has been populated with some new requirements. The new use cases are:

- 'organise in-house course' and 'organise external course'. These represent two different ways in which the course may be organised as sometimes the clients will

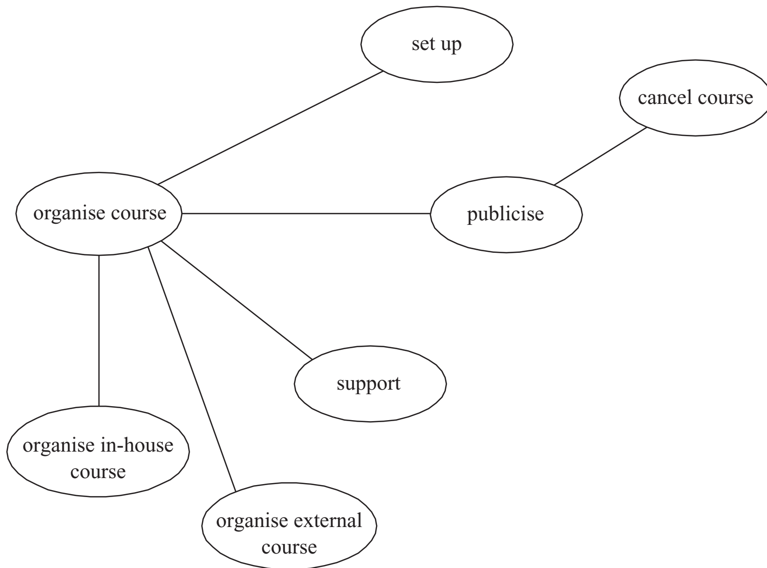


*Figure 7.18 A more populated use case model*

come to the organisation, while on other occasions the organisation will go to the client. Although the courses will still be set up in similar ways, it is important to distinguish between the two types of course. These two requirements are directly associated with the original 'organise course' requirement.

- 'cancel course'. It is highly possible that the publicity will not be successful for the course and that too few people register, thus making the course economically unviable for the organisation. In such an event, the course must be cancelled. This new requirement is related directly to the 'publicise course' use case.

Therefore, when added to the basic decomposition-style relationship, there are two more ways in which a relationship may be defined. Clearly, these are different types of relationships and there is no way to differentiate between them, as seen in this model. This is all clearly leading somewhere and there just happen to be three types of relationship that are defined as part of the regular UML (although two of these are in-built stereotypes). These types of relationship are shown in Figure 7.19.

Figure 7.19 illustrates the three basic types of relationship that may be used to add more detail to use case diagrams, which are part of the standard UML.

The '«includes»' association is used when a piece of functionality may be split from the main use case, for example, to be used by another use case. A simple way to think about this is to think of it as an aggregation-style association. This is used to try to spot common functionality within a use case. It is highly possible that one or more of the decomposed use cases may be used by another part of the system. The direction of the arrow should make sense when the model is read aloud. This part of the model is read as 'organise course' and includes 'set up'.

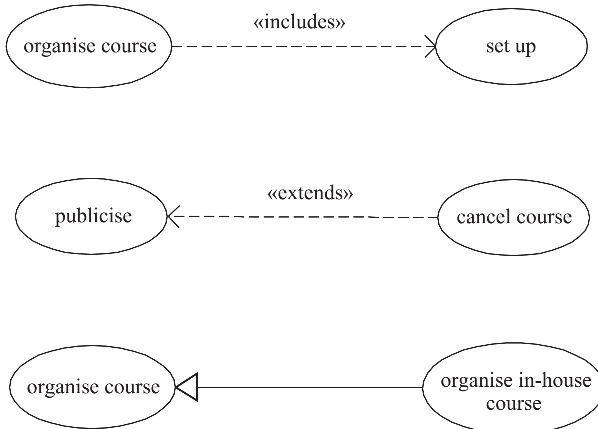


*Figure 7.19 Types of relationship*

The '«extends»' association is used when the functionality of the base use case is being extended in some way. This means that sometimes the functionality of a use case may change, depending on what happens when the system is running. An example of this is when the 'cancel course' use case is implemented. Most of the time, or at least so the organisation hopes, this use case is not required. However, there may be an eventuality when it is required. This is represented by the '«extends»' stereotype, but it is important to get the direction of the relationship correct, as it is different from the '«includes»' direction. Always remember to read the diagram and the direction of the relationship makes perfect sense. In addition, note that both '«includes»' and '«extends»' are types of dependency rather than a standard association.

The final type of relationship is the generalisation relationship that is exactly the same as when used for class diagrams. In the example shown here, there are two types of 'organise course', which have been defined as 'organise in-house course' and 'organise external course'.

These new constructs may now be applied to the requirements diagram to add more value to the model, as shown in Figure 7.20.

Figure 7.20 shows that the main use case 'organise course' has two types: 'organise in-house course' and 'organise external course'. The 'organise course' use case includes three lower-level use cases: 'set up', 'publicise' and 'support'. The 'publicise' use case may be extended by the 'cancel course' use case.

There is nothing to stop new types of relationship being defined – indeed, there are mechanisms within the UML that allow for this and these are discussed in greater detail in Chapter 10. However, let us consider a simple relationship that may be useful for requirements modelling and see how it may be applied to the example model.

It was stated previously that it is very important to distinguish between the different types of requirement that exist, whether they be 'Functional', 'Nonfunctional' or 'Business' requirements. However, it is unclear which of the requirements are which

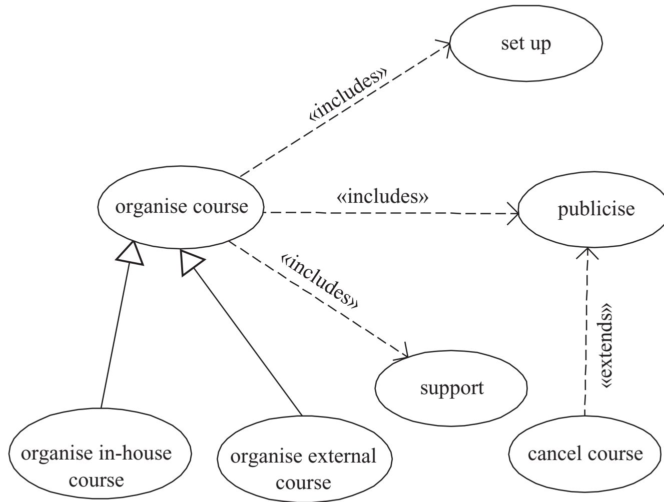


*Figure 7.20 Complete use case model for 'organise course' use case*

in some respects. We already know that anything on a business context will exclude the type 'Functional' and that any requirement in a system context will exclude 'Business' requirements, but what about 'Nonfunctional' requirements that may exist in both contexts? In order to help to understand this, the original definition of what exactly a 'Nonfunctional' requirement is must be revisited. The original definition, according to Figure 7.3, was that 'Nonfunctional' requirements constrain 'Functional' requirements. This will form the basis for the new type of relationship between use cases. In order to define a new UML element, the concept of a stereotype is introduced. The use of stereotypes is explained in detail in Chapter 8, but for the purposes of this example it is enough to assume that it is possible to define a new type of UML element and then, whenever it is encountered, an assumption is made about its semantic meaning. This assumed meaning is conveyed using an 'assumption model' as shown in Figure 7.21.

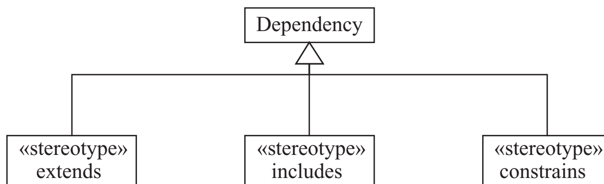


*Figure 7.21 Defining new stereotypes for use case relationships*

Figure 7.21 shows not only the existing stereotypes of relationships, but a new one called 'constrains'. Whenever the term '«constrains»' is encountered in a model, it is assumed that we are using a new type of UML element whose definition must be

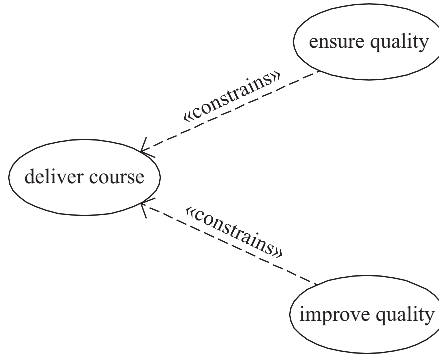


*Figure 7.22 Making 'Nonfunctional' requirements explicit using the 'constrains' stereotype*

explicitly recorded somewhere, ideally with the assumption model. This new type of relationship stereotypes the 'dependency' element from the UML meta-model and is shown in Figure 7.22.

Figure 7.22 shows how the new UML element has been used to show that 'improve quality' and 'ensure quality' constrain the requirement 'deliver course'. Based on the previous definitions in this chapter, it is now quite clear that both 'improve quality' and 'ensure quality' are 'Nonfunctional' requirements.

### *7.5.3 Describing use cases*

#### **7.5.3.1 Overview**

One possible next step is to describe what happens within each use case. This may be done in one of two ways: either by writing a text description or by modelling the use case visually.

A text description is exactly what it says it is. The text description should take the form of structured English text, much like pseudocode descriptions from the software world. These descriptions should describe the typical sequence of events for the use case, including any extension points that may lead to unusual behaviour (relating to the '«extends»' relationships). The text should be kept as simple as possible with each describing a single aspect of functionality. Many textbooks advocate the use of text descriptions, which is fine, but it may be argued that this is going slightly against the UML ethos, which is all about visualisation. For people who prefer visualisation, there are, of course, UML diagrams that allow the internal behaviour of a use case to be described.

The two diagrams that are used to describe what occurs inside a use case are the statechart diagram and the activity diagram, each of which shows a slightly different view of the internal workings of a use case. This, however, can lead to a certain amount of confusion when it comes to scenarios (described in more detail later in

this chapter), as scenarios show an instance of a use case and they are modelled using either, or both, of the interaction diagrams (collaboration or sequence).

#### 7.5.3.2 Confusion with visual use case descriptions

In order to illustrate the confusion that can arise from describing use cases visually, consider the following meta-model description of the relationships between types of diagram.

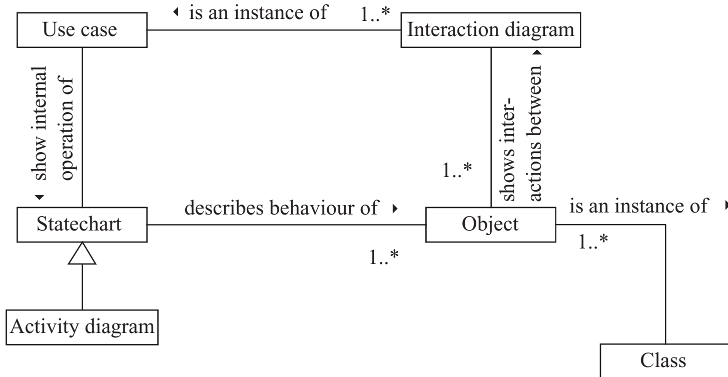


*Figure 7.23 Meta-model showing relationships between use cases and types of diagram*

It has already been stated that a scenario is an instance of a use case, and that an interaction diagram describes a scenario. Therefore, the relationship shown at the top of Figure 7.23 holds true, where one or more 'interaction diagram' describes an instance of a use case. In addition, remember that interaction diagrams are said to be used to show how societies of objects interact at a high level of abstraction, which is shown on the model as an 'Interaction diagram' showing interactions between one or more 'Object'.

Now, consider the relationship between statecharts and use cases. From the model, a 'Statechart' shows the internal operation of a 'Use case'. However, a 'Statechart' also describes the behaviour of one or more 'Object'. Bear in mind the previous point about interaction diagrams describing the interactions between objects, that describe an instance of a use case, which may be described by a statechart, and try not to get a headache!

The solution to this circular definition is to conceptualise the slightly different views that the different diagrams represent.

- The interaction diagram describes a scenario that focuses on the interactions between objects by defining the messages that are passed between them. The key to scenarios is to show interactions between the system and anything that interacts with it. In UML terms, this will equate to the system as defined by the system boundary and its external actors.
- The statechart describes what goes on inside a use case and, hence, within the system itself.

- Activity diagrams, in one sense, can be used to describe the order of execution of activities, regardless of which object they live in.

What this leads to is the fact that a single use case can be described by a statechart to show its internal operation or by an activity diagram to show how its activities are executed alongside activities from other objects. An example of a use case (an instance) is known as a scenario, which is modelled using an interaction diagram that describes the messages passed between the system and its actors.

Maybe this is why most sources of reference for the UML advocate the use of text descriptions for use cases!

#### 7.5.3.3 Text descriptions of use cases

The first approach to describe the internal operation of a use case is to write a simple text description of the sequence of steps that must be carried out in order for the use case to function. The example in Figure 7.24 shows a simple text description for the use case ‘publicise’.

*5.3.3.1 Publicise:*

*Receive suggested dates for course*
*Send out fliers for the course*
*Wait for a predetermined length of time*
*When time is up, assess the response*
*If the response is sufficient, send details to attendees*
*If response is insufficient, cancel the course*

*Figure 7.24 Use case description using text description*

It should be quite obvious to anyone looking at this description how the use case is intended to function. The advantage to using text descriptions is that they are very simple and, when written with care, are easy to understand. On the down side, however, a slightly more complex use case may be very difficult to represent in text terms. In addition, the formal tie in to the rest of the UML model is almost nonexistent, whereas a visual approach would have stronger ties.

The next way to describe a use case is to model it using a statechart.

#### 7.5.3.4 Statechart description of a use case

The second approach to describe the internal operation of a use case is to model visually using a statechart or activity diagram. The model in Figure 7.25 shows a statechart representation of the ‘publicise’ use case that is equivalent to the text description defined in Figure 7.24.

Figure 7.25 shows a visual description for the ‘publicise’ use case. One thing that is much clearer using the visual approach is that loops and iterations are far easier to spot. These can often be obscured, lost or misconstrued using an informal text description, but are immediately apparent when they can actually be seen. From a UML point of view, note that ‘receive date suggestions’ is an action whereas ‘send fliers’ is an activity. The keyword ‘entry’ applies to an action that takes zero time and

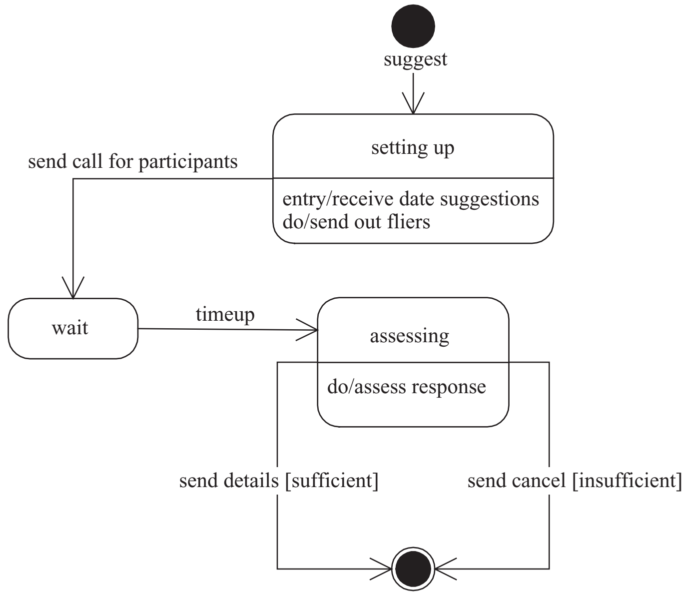


*Figure 7.25 Use case description using a state machine diagram*

is noninterruptible and so is correct for receiving a date. The keyword 'do', however, applies to an activity that indicates that the 'send fliers' will take time and may be interrupted.

#### 7.5.3.5 The better approach

The immediate question that most people ask at this point is 'which is the better of the two approaches?' This can be answered with the standard UML response of 'whichever is the more suitable for the application at hand'.

Text descriptions are often preferred as they are easier to create and, it may be argued, can be very simple for anyone to understand. They also have the advantage of being in a text form which will fit directly into a user requirements document, for example. However, anything written in plain English is more prone to error and misinterpretation, simply because any spoken language is so expressive and almost all words have more than one meaning.

A visual description of a use case may be preferred for someone wanting a more formal approach to describing use cases. In addition, it is easier to reference one model to another model than it is to relate a text description to another part of the model.

Some people may not even want to describe the use cases in any way, as perhaps they do not have enough information about the requirements. One way to derive more information about requirements is to consider a scenario that interacts with the outside world, from a single aspect of functionality. Defining scenarios will be covered in more detail in Section 7.6.

## 7.6 Modelling scenarios

Once a set of user requirements has been generated and they have been organised and analysed using use case diagrams, it is possible to go one step further and look at 'scenarios'. A scenario shows a particular aspect of a system's operational functionality with a specific goal in mind. The definition of a scenario from the UML point of view is that it is an instance of a use case. This then gives good, strong traceability back to the original requirements since scenarios are often generated as part of the analysis during the specification phase. This is not always the case as it will depend upon the process that is being followed – indeed, it is not unheard of to have a set of scenarios defined during the requirements phase. However, as we are following the process defined in Chapter 6, it is assumed that we have now satisfied all the criteria to exit the requirements phase and have moved into the system specification phase.

Scenarios have many uses, can be a crucial part of the analysis of the user requirements and will help to identify the intended functionality of the system. This has a number of practical uses:

- It is important to identify how the user will use, and interact with, the system. Bear in mind that the actual design will take its starting point as the system specification, and thus it is important to understand how the system will be used.
- Scenarios are a good source for system and acceptance tests for a system. A scenario represents a particular aspect of a system's functionality and models the interaction between the system and the outside world, thus they are an excellent source for defining acceptance tests.

When scenarios have been created as part of the analysis or specification phase, they will form the basis for the actual system design. Therefore, as they are traceable back to individual use cases and forward to the design, they enable a traceability path to be set up from the design right back to the requirements.

### *7.6.1 Scenarios in the UML*

A scenario models how the system interacts with the outside world. The outside world, in the case of our system, is actually a subset of the stakeholders that have already been defined during the requirements phase and exist both on the stakeholder model as classes and also on the business and system contexts as actors. Potential paths for communication between a stakeholder and a user requirement are clearly indicated on the UML diagrams as associations that cross the system boundary.

The diagram that is used in the UML to realise scenarios is the interaction diagram, of which there are two types: collaboration and sequence. Collaboration diagrams show the interaction between objects in a system from an organisational point of view, while sequence diagrams show the interactions between objects from a logical timing point of view.

When considering scenarios, we think about actual objects in the system, rather than classes, as we are looking at a real example of operation. This gives a strong link

to class diagrams as all objects must have an associated class. This is one approach to identifying classes for the analysis and design of the system.

### *7.6.2 Example scenarios*

The first step when defining scenarios is to choose a use case from the original requirements model, which, for the purposes of this example, is shown in Figure 7.20. The use case that will be chosen for the initial example is ‘publicise’, which describes a requirement for the system to help the organiser publicise a training course. This is the same use case that was described previously using both a statechart and a text description.

With the use case described, it is now time to look at some possible scenarios from that use case. Two spring immediately to mind: one where publicity is successful and the course goes ahead and one where the publicity is unsuccessful and the course does not go ahead. Each is modelled using two types of interaction diagrams, which also helps to highlight the differences between the two types of diagram.

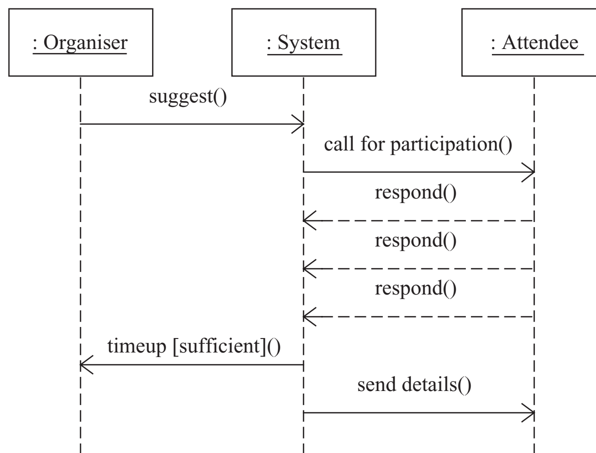


*Figure 7.26 Scenario #1 – successful publicity, sequence diagram*

Figure 7.26 shows a sequence diagram that represents a scenario where the publicity is successful.

The first thing to notice about the model is that the main components are instances that are real-life instances of classes. Sometimes, these instances may be shown using the actor symbol if they represent stakeholders from the system context. It is usual to show the system as a single instance that will allow us to identify the interactions that must occur with the outside world in order for the system to meet its original requirements.

Remember that the dotted line, shows logical time going down the page. Therefore, as time goes on, we would expect certain interactions to occur. These have

already been defined to a certain extent since the use case has already been described using a statechart.

The first thing to happen is that the 'Organiser' suggests some details to the 'System'. These details may include the suggested time, date and location. The 'System' then sends out a call for participation to a number of 'Attendee'. In this scenario, a number of attendees respond to the call for participation by sending messages back to the 'System'. Eventually, the time allowed for people to respond expires and the system has to make a decision about whether to progress with the course or not. In this scenario, there is sufficient demand for the course and thus the system sends out details of the course to the attendees who have applied to attend the course. That is the end of that particular scenario.

The diagram used here was the sequence diagram (a type of interaction diagram) but there is no reason why the same information could not be shown using a collaboration diagram. To this end, Figure 7.27 shows exactly the same scenario, but modelled using a communication diagram rather than the sequence diagram.

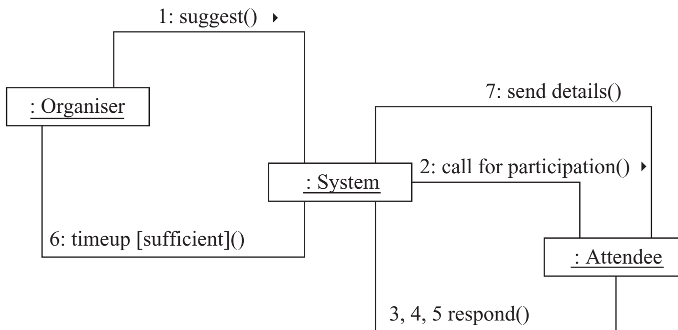


*Figure 7.27 Scenario #1 – successful publicity, communication diagram*

The same scenario is now modelled using the other type of interaction diagram: the communication diagram.

The sequence of events and the messages that are passed between two instances can still be seen, but on the communication diagram the emphasis is on the organisation of the instances, rather than the sequence. The actual order of events is indicated by sequence numbers that are represented by simple integers. It is possible to nest numbers by showing integers separated by points – for example, 1.1, 2.3.2 etc. It is quite clear to see that these numbers will become complicated and difficult to follow before too long.

Now consider a different scenario, one where the publicity is unsuccessful due to an insufficient number of responses. Although not frequent, this is certainly a scenario that must be taken into account when thinking about the system.

Figure 7.28 shows the second scenario. Note how the sequence of messages begins in exactly the same way as before, right up to the point when the call for participation is

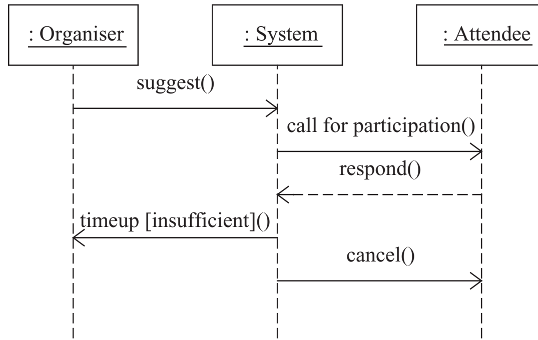


*Figure 7.28 Scenario #2 – unsuccessful publicity, sequence diagram*

broadcast. The differences occur because there are not many responses this time – in fact, there is only a single response in this scenario. The same 'timeup' event occurs, but in this scenario the condition has changed. Rather than there being a 'sufficient' number of responses, there is an 'insufficient' number. The consequence of this is that the course must be cancelled, hence the 'cancel' message is sent out to any 'Attendee' who may have responded to the original call for participation.

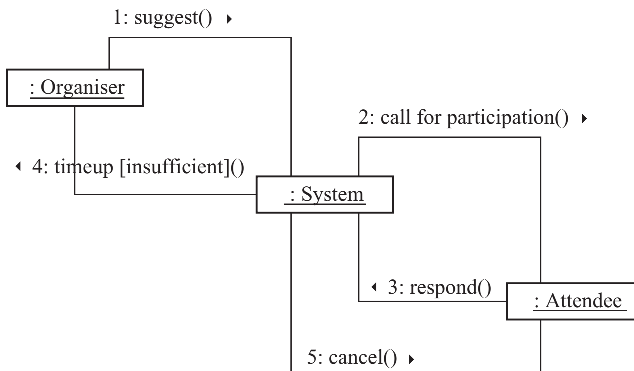


*Figure 7.29 Scenario #2 – unsuccessful publicity, communication diagram*

The second scenario is shown again in Figure 7.29, but with a communication diagram rather than a sequence diagram. As with the sequence diagrams for both scenarios, there are similar patterns between the two communication diagrams. Spotting these patterns can be quite beneficial to the design process as they help to identify common areas of functionality of the system.

The scenarios considered so far have been deliberately simple. It is often a good idea to start off with simple scenarios that are well defined before complex scenarios are considered.

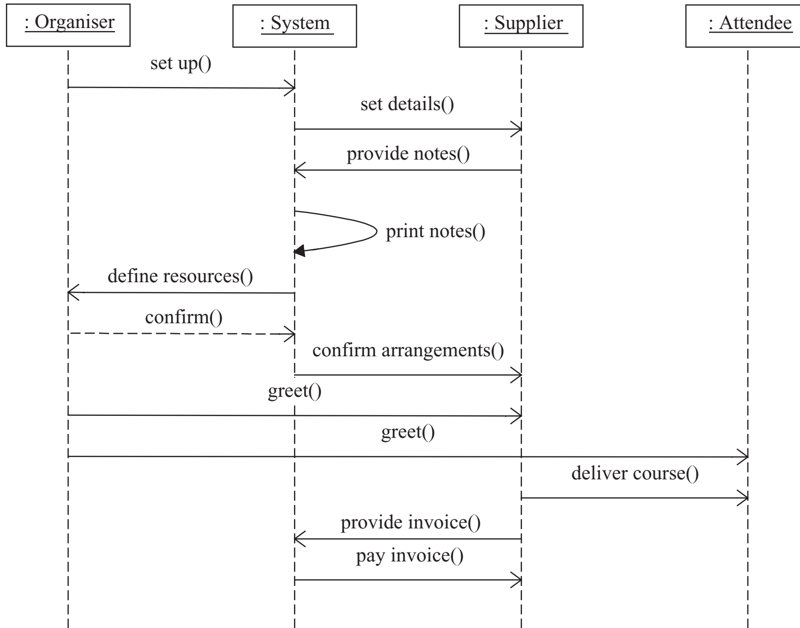


*Figure 7.30 A more complex scenario*

Figure 7.30 shows a new scenario that exhibits far more complexity than the previous two scenarios. This scenario relates to the 'set up' use case and describes the normal sequence of operations. Note that this sequence diagram contains a message that is passed from, and then back to, a single instance. Although these are not strictly necessary, they sometimes help the person reading the model to understand more about what is going on. The message in question here is the 'print notes' message. Although this message does not describe an interaction between objects, it was deemed as being useful to include it to make explicit the fact that there is part of the scenario where printing takes place.

### *7.6.3 Communication versus sequence diagrams*

The two types of interaction diagram have been discussed and example scenarios shown. The question that is asked most frequently at this point is 'what is the point of having two diagrams that show the same information?'. This is a legitimate question and one that requires some discussion in order to fully appreciate the subtle difference between the two types of diagrams.

Scenarios that are realised by communication diagrams are more generalised than sequence diagrams, particularly when there are many iterative interactions. These may be shown as a single interaction on a communication diagram, but may be many interactions on an equivalent sequence diagram. Iteration may be indicated on

a collaboration by using an '*' symbol before the sequence number, but the paths are not shown explicitly.

The structure of the communication diagram stresses the organisation of the instances, rather than their relative timing relationships.

Scenarios that are realised by sequence diagrams are more specific and less generalised than in communication diagrams. Iterations are shown very clearly on sequence diagrams and may show return paths and more types of message. The emphasis in sequence diagrams is on the logical timing rather than the organisational structure of the system. By logical timing, we mean the order in which things happen, rather than in real time.

### *7.6.4 Wrapping up scenarios*

In summary, therefore:

- Scenarios are defined as instances of use cases and, as use cases represent user requirements, the scenarios relate directly back to the original requirements for the system.
- Scenarios are realised by interaction diagrams, of which there are two types used here: communication and sequence. Communication diagrams stress the organisation of the instances in a system and their interactions, while sequence diagrams stress the logical timing relationships between instances.
- Several scenarios may be modelled for each use case. In fact, enough scenarios would be modelled in order to show a good representation of the overall system operation.

One question that is often asked about scenarios is 'how many should be produced?'. Like most of the questions that people tend to ask most often, there is a simple answer, but it is one that most people do not seem too happy with. The simple answer is 'just enough', which is very difficult to quantify. It is important to consider enough scenarios so that you have covered all typical operations of the system and several atypical operations. A good analogy here is to think about testing a system. It is impossible to fully test any complex system, but this does not mean that there is no point in testing it. In fact, it is important to test the system to establish a level of confidence on the final system. Indeed, this analogy is appropriate in another way, as scenarios form a very good basis for acceptance and system testing for just this reason.

## 7.7 Documenting requirements

### *7.7.1 Overview*

This section looks at the practical issue of documenting requirements. The way in which requirements are documented will depend upon the process that is being adopted by the organisation.

Some processes do not rely on documentation as such – for example, the RUP. In the RUP, each of the models that are created and in many cases some of the components that make up the models, are stored as artefacts. These artefacts are roughly equivalent to the deliverables in a more traditional approach and form the core of the project repository. In such an approach, each use case is classed as an artefact and the models themselves are also treated as artefacts. Therefore, when it comes to documenting such use cases or models, all information is stored in the model, which can then be automatically drawn out into a report as and when necessary. This approach relies heavily on tools that are compatible with the approach and that are intended to make life simple, and such documentation is a matter of pressing a single key, rather than assembling documents by hand. There are advantages and disadvantages with such an approach, but these are beyond the scope of this book.

For the sake of consistency, the example used here will follow on from the ISO-type process defined previously.

The actual structure of the requirements documentation used for this example is based on a model of the user requirements specification (URS) derived from existing standards and best practices using the approach introduced in Chapter 6. Let us suppose that, in order to define the contents of a URS, a source of information was looked for. In this case, the content of the URS is based on the one defined by the European Space Agency [10]. It is modelled according to the guidelines laid out in Chapter 6 and results in the model shown in Figure 7.31.

Figure 7.31 shows the structure for the URS, as derived from the process modelling exercise discussed previously. The model shows the various sections of the document as classes, and the various subheadings are shown as attributes on each class. Of course, depending on how many levels of nested headings exist in the document, there would be an associated hierarchy of classes in the document model.

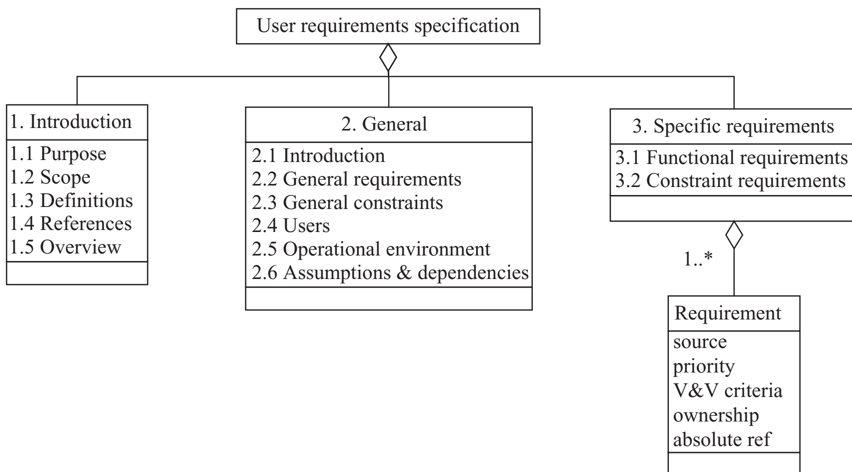


*Figure 7.31 Typical contents of a user requirements document*

It can be seen that the 'User requirements specification' is made up of '1. Introduction', '2. General' and '3. Specific requirements'. '1. Introduction' has the subsections '1.1 Purpose', '1.2 Scope', '1.3 Definitions', '1.4 References' and '1.5 Overview'. '2. General' has six subsections and '3. Specific requirements' has two subsections. In addition, '3. Specific requirements' is made up of one or more 'Requirement'. These requirements are described according to the model defined in Figure 7.5, which is actually included in this model.

This document may then be populated by making use of all the models that have been created so far, such as the contexts, stakeholder models, requirements models and scenarios.

### *7.7.2 Populating the document*

Each part of the document that has been identified in Figure 7.31 will now be discussed and appropriate diagrams are suggested for each section.

#### 7.7.2.1 Section 1 – introduction

The introductory section is mainly concerned with setting the scene for the project and perhaps justifying why it is being carried out:

- The business context and parts of the system context may be used to illustrate the first two subsections of '1.1 Purpose' and '1.2 Scope'.
- Definitions in '1.3 Definition' may be defined according to the terms used as UML elements.
- The information in '1.4 References' must be generated manually.
- Finally, '1.5 Overview' may again make use of the parts of the business context and system context information.

As can be seen, most of the information for the first part of the document can be derived directly from the UML models created so far.

#### 7.7.2.2 Section 2 – general

The second section of the document may be populated as follows:

- Section '2.2 General requirements' and section '2.3 General constraints' may be obtained from high-level use cases. This will include the use case models for the high-level requirements and the business context use case model. This is because, as stated previously, many user requirements and constraints will be derived from business requirements.
- Section '2.4 Users' may be defined directly from the user models and stakeholder models, which were represented previously using class diagrams.
- Section '2.5 Operational environment' may be described by the information in the system context model, which was realised by using a use case diagram.

Again, the UML diagrams can be used to populate most of this section of the document.

#### 7.7.2.3 Section 3 – requirements

The third section of the document may be populated as follows:

- Section '3.1 Functional requirements' and section '3.2 Constraint requirements' will form the bulk of the document. This information may be extracted from low-level requirement use case diagrams, requirements descriptions that were realised using text descriptions and statecharts, and, finally, scenarios. Scenarios were realised by interaction diagrams: sequence diagrams and collaboration diagrams.

The third section can be populated almost entirely using the UML models. This is not surprising, as the idea behind this chapter is to model requirements and this section is concerned purely with requirements.

### *7.7.3 Finishing the document*

In order to finish off the document, the following points should be borne in mind:

- Include the actual models, where appropriate, in the document itself. This is not always possible but can prove to be very useful. As an interesting aside, non-UML experts will be able to understand most simple UML models with a minimum of explanation, providing that the readers do not realise that they are UML diagrams. This may sound strange, but many people will mentally 'switch off' if they think that the diagrams are in a language that they do not understand and will make no attempt to understand them.
- Add text to help the reader understand the diagrams. By basing the text directly on the diagrams, there is less room for ambiguity through misinterpretation and it also ensures that the terms used in the document and the models are consistent. It is important to remember that the knowledge concerning the project exists in the models, rather than the documents. Think of the documents as a window into the model, not the other way around. Always remember to change the model first, rather than the document, to retain consistent project knowledge.
- Structure the actual requirement descriptions according to the structure of the requirements shown in the original requirement model, showing attributes associated with each requirement. Again, this leaves less room for ambiguity and, if the modelling has been performed well, should be a good way to structure things anyway.
- The document will then drive the rest of the project and be a formal view of the models. Any changes that are to be made to the system should be made in the model and then reflected into the user requirements document.

Above all, remember to make as much use as possible of the UML models. Every model will have been created for a reason, so bear them all in mind when writing the document. Modelling should be an essential, useful and time-saving part of the project, not just an academic exercise to produce nice pictures.

## 7.8 Summary and conclusions

In summary, therefore, this chapter has introduced and discussed the points covered by the following model.

Figure 7.32 shows the following points:

- Each 'Project' is made up of a number of one or more 'Phase'. One type of 'Phase' is 'User requirements'. No other phases were shown at this point as it was beyond the scope of the chapter.
- One or more 'Requirement' is generated during the 'User requirements' phase and each 'Requirement' must be one of three types: 'Business', 'Functional' or 'Nonfunctional'.
- One or more 'Business' requirement drives one or more 'Functional' requirement and one or more 'Nonfunctional' requirement constrains one or more 'Functional' requirement.
- One or more 'Stakeholder' is identified during the 'User requirements' phase and each owns one or more 'Requirement'. There were two main types of 'Stakeholder' identified: the 'Customer' and the 'Supplier'.
- There are two types of 'Context': the 'Business context' and the 'System context'. These types of 'Context' define the boundaries for the 'System', which is delivered by the 'Project'.

This basically summarises the concepts of requirements modelling, but this chapter relates all these concepts to UML modelling, which may be summarised by the following model.

The relationships between requirements engineering concepts and UML elements are summarised in Figure 7.33. From the model, it can be seen that:

- Class diagrams are used to create a stakeholder model, which is useful when trying to identify stakeholders, which is an essential part of the user requirements phase.
- Each stakeholder, once identified, could be represented as a UML actor on a use case diagram. This becomes particularly important when it comes to modelling the business and system contexts for a particular project.
- The context for the project, whether it is the system context or business context, is indicated by using a UML system boundary element. This element is underused by most modellers and there is a general lack of support for system boundaries on current CASE tools.
- Use cases are used to represent the three different types of requirement. Each use case is used to represent either a single requirement or a grouping of like requirements, which could then be decomposed into further requirements.
- Interaction diagrams are used to model specific scenarios, in order to understand the operation of the system. Each scenario is an instance of a use case that gives a good, strong traceability path back to the original requirements model.
- Use cases and thus requirements could be further described using either a text description or a statechart or activity diagram. It is important that these are

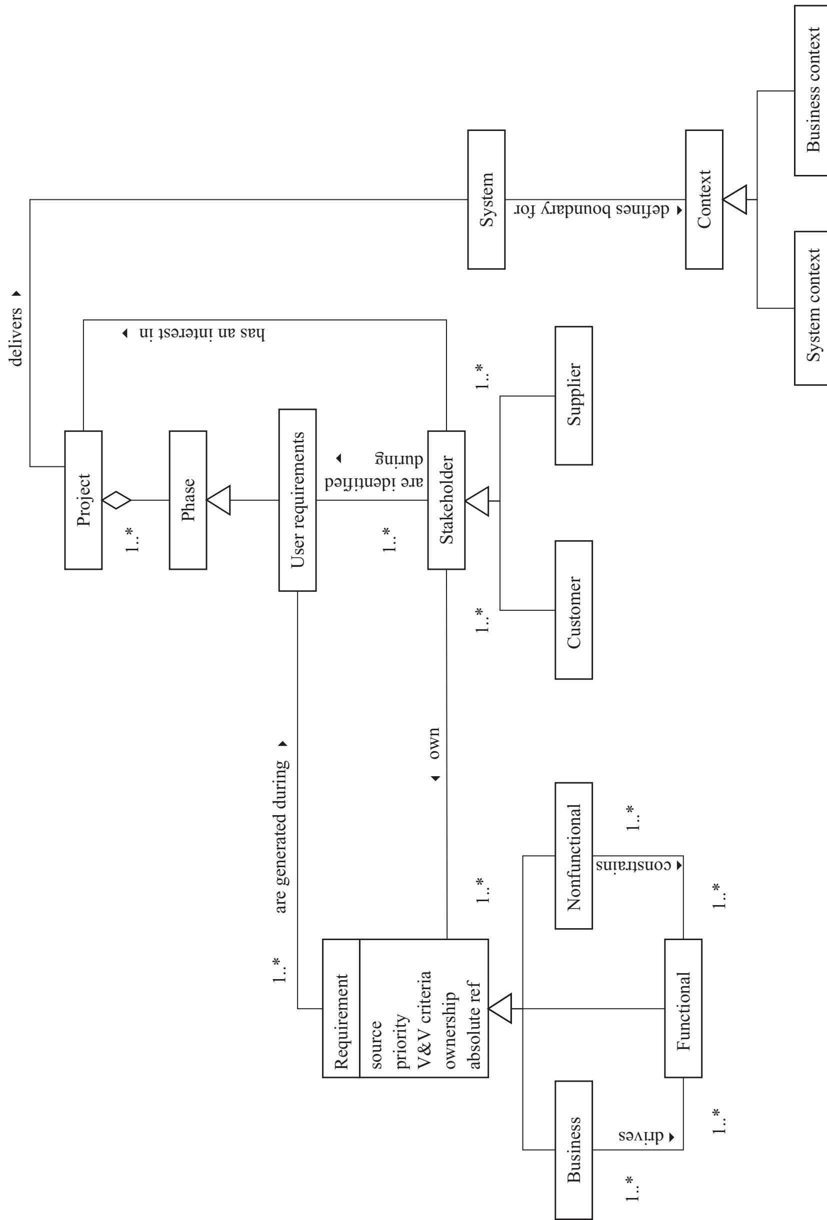


*Figure 7.32 Concepts introduced in this chapter*

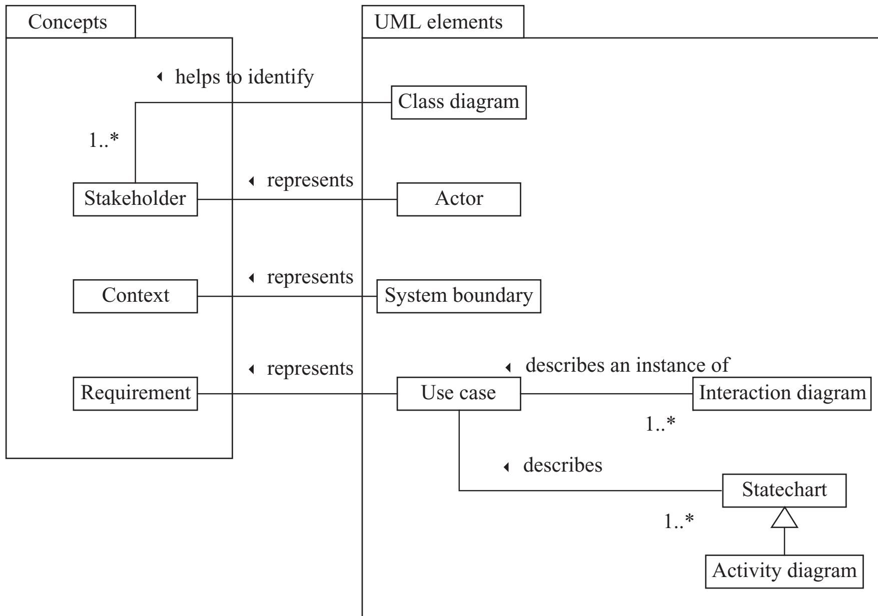


*Figure 7.33 Summary of relationships between concepts and UML elements*

differentiated from interaction diagrams that show scenarios, or instances, of a use case.

The best way to learn these concepts in more detail, like all aspects of the UML, is to take an example and model it for yourself. Section 7.9 gives some ideas about how these ideas may be taken further.

## 7.9 Further discussion

1. Consider a project that you are familiar with and try to identify a stakeholder model, using a class diagram. Does the generic model shown in Figure 7.7 work for your example, or does it need to be changed considerably? In either case, why is this? Which context will each stakeholder appear in: the system context, the business context, or both?
2. Take another look at the model shown in Figure 7.16 and select a different requirement. Try to decompose this requirement into lower-level requirements so that it still makes sense. Also, bear in mind that whatever requirements are generated must be kept in line with the original business requirements that were shown in Figure 7.4.
3. Look at the requirements shown in Figure 7.20 and select a different use case from the one chosen for the example. This time, try to create some scenarios that would represent typical system operation. Use both the sequence diagrams

and collaboration diagrams and compare and contrast them. Can you think of a situation where the collaboration diagram would be more useful than the sequence diagram and vice versa?

4. Choose any requirement that has been discussed in this chapter (whether functional, nonfunctional or business requirement) and document it as if it were part of a user requirements specification. Which part of the document would it be most applicable to?
5. Model the business requirements for another organisation (maybe your own) and see if they make sense to someone else. Choose a few existing projects or other pieces of work and see if they fit in with your business requirements model. Is the organisation's mission statement included as part of this model? Is it possible to demonstrate to a third party that the organisation is meeting its mission statement, or not?
6. Consider the system context shown in Figure 7.16. How would this be different if it were the system context from the customer's point of view, rather than the supplier's? Are there any new requirements that need to be added, or any existing ones that need to go? Are the stakeholders the same for both contexts and, if not, then why not? Does the new system context still meet the business requirements in the business context? Does it need to meet these business requirements?
7. Take one of the business requirements shown in the business context in Figure 7.4 and try to think of the type of project that may be associated with some of the other business requirements. Can you come up with a quick system context for such a model? Which stakeholders, if any, are relevant to the new system context and the business context? Are there any new stakeholders to be identified?
8. Complete the model in Figure 7.14 using the three basic and one stereotyped relationship introduced in this chapter.

## 7.10 References

1 JACOBSON, I.: 'Object-oriented software engineering: a use case-driven approach' (Addison-Wesley, Washington, 1995)

2 JACOBSON, I., BOOCH, G., and RUMBAUGH, J.: 'The unified software development process' (Addison-Wesley, Massachusetts, 1999)

3 SCHNEIDER, G., and WINTERS, J. P.: 'Applying use cases – a practical guide' (Addison-Wesley, Massachusetts, 1998)

4 JACKSON, M.: 'Software requirements and specifications' (Addison-Wesley Publishing, New York, 1995)

5 DAVIES, A. M.: 'Software requirements: analysis and specification' (Prentice-Hall International Editions, London, 1990)

6 SCHACH, S. R.: 'Software engineering with Java' (McGraw-Hill International Editions, Singapore, 1997)

7 STEVENS, R., BROOK, P., JACKSON, K., and ARNOLD, S.: 'Systems engineering, coping with complexity' (Prentice-Hall Europe, London, 1998)

8 O'CONNOR, J., and McDERMOTT, I.: 'The art of systems thinking, essential skills for creativity and problem solving' (Thorsons, London, 1997)
9 SKIDMORE, S.: 'Introducing systems design' (Blackwell Publishing, Oxford, 1996)
10 MAZZA, C., FAIRCLOUGH, J., MELTON, B., DE PABLO, D., SCHEFFER, A., and STEVENS, R.: 'Software engineering standards' (Prentice-Hall, Herfordshire, 1994)

*Chapter 8*

# Life cycle management

life is what happens to you whilst you're busy making other plans

*John Lennon*

## 8.1 Life cycles and life cycle models

Life cycles and life cycle models are very widely misunderstood concepts. Despite the fact that they are fundamental to almost every project, life cycles and life cycle models are often confused and the terms used (incorrectly) interchangeably. There is also a great deal of misunderstanding about how processes relate to both of these concepts and, once more, people often use the term 'process' interchangeably with both life cycle and life cycle model [7].

A life cycle describes the evolution of a system from conception to its ultimate demise. The most obvious analogy to this is a human life where the life cycle starts at the actual conception, rather than birth, and continues on until death.

A life cycle is made up of a number of phases, each of which will implement one or more processes. It follows, therefore, that in order to define a life cycle, its phases must be defined. In order to define a phase, its associated process, or processes, must also be defined. Bearing in mind that a set of processes has been defined in Chapter 6, it is possible to define a life cycle using these processes.

A life cycle model describes the configuration of a life cycle for a particular project, or to put it another way, a life cycle model defines the order in which the processes are carried out and is defined on a project by project basis. In UML terms, a life cycle states 'what' (a structural view), while the life cycle model states 'how'.

Figure 8.1 shows these definitions using the UML. From the model, the 'Life cycle' is made up of one or more 'Phase', each of which implements one or more 'Process'. Each 'Process', as before, is made up of one or more 'Practice'.

The 'Life cycle model' shows the implementation of the 'Life cycle' and the configuration of one or more 'Phase'. This is consistent with the process definitions that were stated previously.

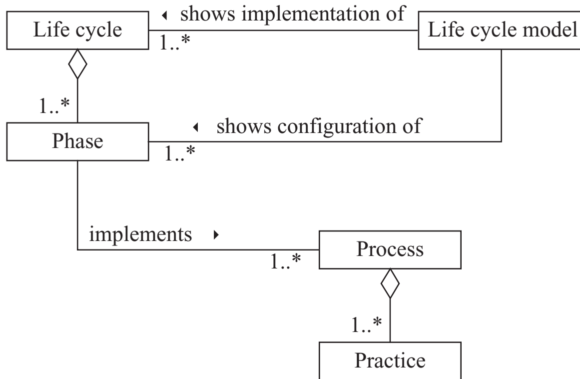


*Figure 8.1 Definition of life cycle*

Figure 8.2 shows a 'Life cycle' that is made up of one or more 'Phase'. The types of 'Phase' have been defined as 'User requirements', 'System specification', 'Planning', 'Design', 'Integration', 'Implementation', 'Installation' and 'Maintenance'.

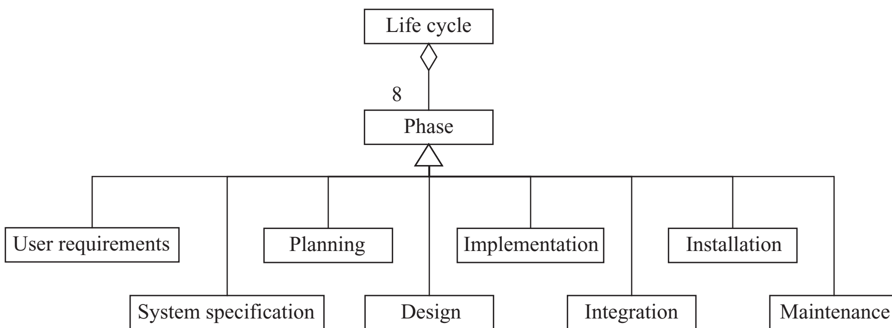


*Figure 8.2 A typical life cycle*

There is no single life cycle that will apply to all projects, as it will depend on the nature of the project and its application. Phases should be chosen depending on the type of project and the processes that are available to be implemented in each phase. For example, a project that is concerned with specifying a bespoke system may include a user requirements phase, a system specification phase and an installation phase only. Another example, however, would be a project to be developed in-house that would include all the phases shown in Figure 8.2.

The phases have been named, in this case, according to the processes that they implement. This can lead to some confusion as people often assume that 'phase' and 'process' are the same thing, which is not always the case. Consider, for example, the RUP, where each phase is made up of a number of iterations, each of which may

implement between one and nine core workflows. A core workflow in the RUP is equivalent to a process in ISO terms.

Note the inclusion of the phase 'Planning' here. This comes from the 'MAN' process group from Chapter 6, which demonstrates the 'pick 'n' mix' aspect of the life cycle construction.

As this is a structural view of the life cycle, the model shows what the life cycle looks like, but not how it behaves. The behaviour of the life cycle is shown by its 'Life cycle model', which may be realised in the UML using an interaction diagram.

## 8.2 Different life cycle models

### *8.2.1 Introduction*

Choosing a life cycle model is a very important step in any project and there are two main questions that must be asked when deciding – when to choose and which model to choose.

In terms of 'when to choose', this will depend on the type of project and the process being followed on the project, whereas 'which to choose' will depend on the nature of the project. Each of these points will be covered in more detail in this chapter.

### *8.2.2 When to choose*

In terms of when to choose the life cycle model, this is not as easy a question as it first appears since it will depend on the type of project being undertaken. For example, consider an organisation that develops a single product but, once or twice a year, they release a new version of the product. The nature of the project, in this example, is very well understood, therefore, it is possible and appropriate to make the decision of 'which life cycle?' right at the beginning of each project as each project is a new iteration of a well-understood cycle. Consider, on the other hand, a project that is completely new to an organisation in terms of its deliverables and the type of deliverables, then this introduces an element of doubt as to how the project will progress. In this example, the choice of the life cycle model will be left until later in the project.

### *8.2.3 Linear versus iterative models*

There are two main camps in the world of life cycle models – the linear and the iterative life cycle models. Linear life cycle models have evolved from the original 'Waterfall' model into a variety of shapes and sizes, whereas the iterative models represent the latest thinking in life cycle models. Despite claims from various sources, there is nothing really new about the iterative model, it does not represent a quantum leap in project management theory, it is simply a natural evolution of the linear techniques that involves a slightly higher level of thinking.

Choosing a life cycle model is relatively straightforward providing that the processes have been well defined. Bear in mind that a life cycle model defines

the behaviour of a life cycle which is made up of phases, each of which implements one or more process. Therefore, the first step is to identify which processes will be adopted for the project. These processes should then be collected into phases and then how the phases will be executed should be determined, which is the life cycle model.

### *8.2.4 The linear approach*

Depending on the type of project, a particular life cycle model may be selected. Many established life cycle models exist, such as 'Waterfall', 'Spiral', 'Incremental' and 'Iterative', or it may be a bespoke model. Any book on software or systems engineering will show these basic models and the way in which they are structured.

Whichever life cycle model is chosen, the phases defined in Figure 8.2 are executed. In real terms, this means executing or instantiating processes within the phase.

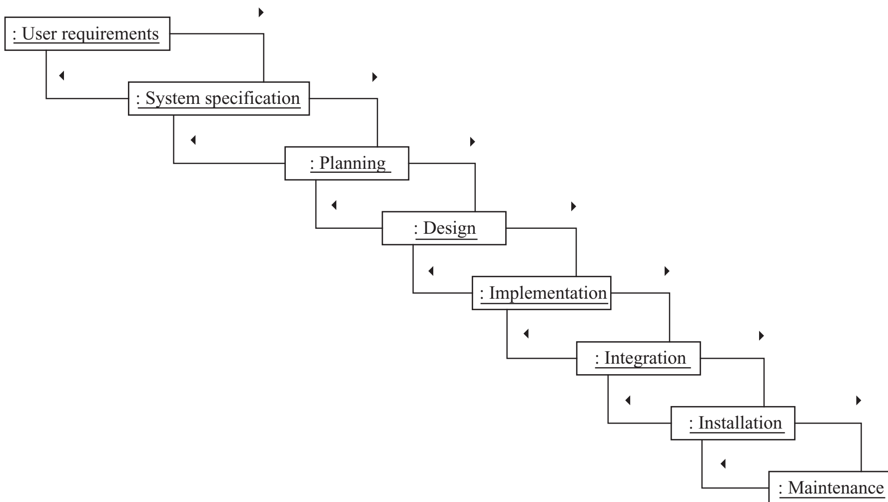


*Figure 8.3 The 'Waterfall' model*

Figure 8.3 here shows an example of a life cycle model known as the 'Waterfall' model. The Waterfall model allows a project to progress from one phase to the next or back to the previous phase. This sums up the life cycle model. This is perhaps the oldest of the established life cycle models and is one that has been used successfully for many years on many projects. However, the Waterfall model has also been used unsuccessfully on many projects as, like most things in real life, it has its own limitations. To summarise, the Waterfall model is regarded as very useful when project requirements are well defined, or for small projects.

Each phase is shown as an instance of a process defined previously in this chapter. It is interesting to note that one instance of a process is 'Planning', which was not defined as part of the 'ENG' process group, but is part of the 'MAN' process group from Chapter 6.

Any type of life cycle model whatsoever may be modelled in this way using interaction diagrams (in this case, a communication diagram). This will dictate the permissible navigation of phases throughout a project but will not necessarily show actual project progress without introducing sequence numbers and making the model far more complex. One way to show the actual progress on a project is to show the same information, but to use the other type of interaction diagram: the sequence diagram.

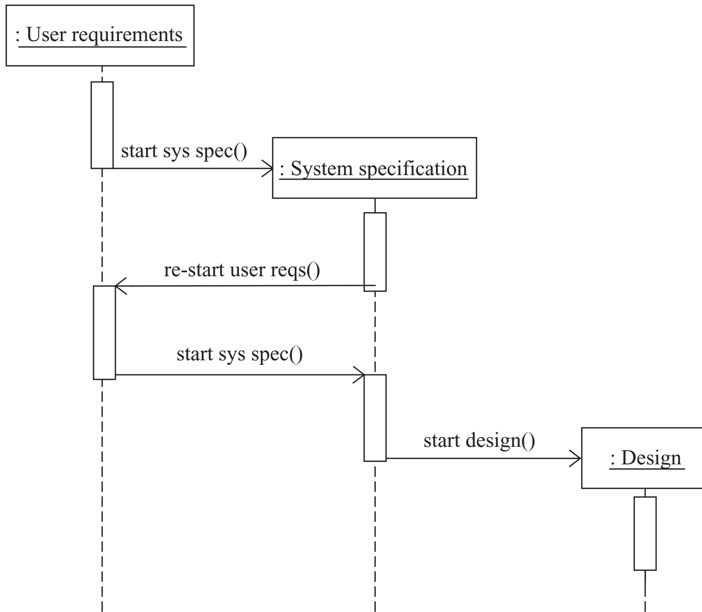


*Figure 8.4 Sequence diagram view of the Waterfall model*

Figure 8.4 shows a sequence diagram view of the Waterfall model. This model is more useful from a project monitoring point of view as it shows individual iterations of the life cycle far more clearly than the collaboration diagram model. It can be seen here that there has been an iteration in this life cycle where the progress has gone back from 'System specification' to 'User requirements', returning to 'System specification' before going on to 'Design'.

This linear approach reflects a common-sense approach to project development. However, there are a number of problems associated with the implementation of such linear approaches:

- *Scalability*: Linear approaches work very well for small, well-understood projects but tend not to scale up to be workable in large projects. For example, it is quite simple to imagine where an approach like the one shown in the previous diagram would be applicable to a small project with a single team of workers involved. However, imagine the same approach being applied to a large project involving many teams of people. The model simply will not work for such a project.

- *Frozen phases*: It is confidently assumed that once the requirements have been drawn up they are frozen for the remainder of the project and will not change. In real life, this is far from the truth as requirements tend to 'creep' or change altogether. It is also possible for new requirements to emerge from an existing requirements set, particularly for large projects.
- *One process for each phase*: In many cases, a single process will be implemented during each phase, which leads to a great deal of confusion regarding the two terms as well as making for a very rigid, inflexible life cycle model.

These are just a few of the problems that are often associated with linear approaches – for more detailed discussions, see References 1 to 4.

Rather than being completely negative about linear approaches, it should be remembered that they are good for small projects. Why then is it not possible to break up a large project into a number of small, well-understood mini-projects and then apply the same principles? The answer is that this is possible, and forms the basis for the concepts behind the iterative approach to project development.

### *8.2.5 The iterative approach*

Contrary to many opinions concerning iterative life cycle modelling, there is really nothing new or revolutionary about it. Rather, the iterative approach is simply an evolution of the existing linear approaches that is intended to make a simple yet flexible approach to system development. There are some key differences between the iterative and linear approaches:

- *Fewer phases*: Iterative life cycles tend to have fewer phases, each of which represents a more generic concept with particular aims and goals, rather than identifying which processes should be executed in it. For example, rather than saying 'the concept phase implements the requirements process' it would be more apt to say 'the concept phase aims to produce a complete set of requirements'.
- *Each phase implements a number of iterations*: Leading on from the previous point, each phase is made up of a number of iterations. A good analogy to consider when trying to understand an iteration conceptually is to consider a number of teams working on a single project, each of which may have to work in a parallel with one another. Imagine that each team represents an iteration within the phase and the analogy starts to work. Imagine now that each conceptual team could actually be a single person or a number of people, and that these iterations may be executed in sequence or parallel and a more complete picture starts to emerge. For example, consider a project where an initial investigation of the requirements shows that more information needs to be understood about the problem domain and the solution domain. It could be the case where there are three iterations here: iteration #1 would involve the initial requirements investigation and must precede the other two. Iteration #2 and iteration #3 are both conducted in parallel – one investigates the problem domain and the other the solution. All iterations are working towards the common goal of producing a single coherent set of requirements, but in a number of iterations rather than a single process.

- *Each iteration implements a number of processes*: Each of these iterations may execute any number of processes to produce mini-linear models. For instance, taking the example used in the previous point, the first iteration (concerning requirements) may execute a requirements process only. Iteration #1 and iteration #2, however, may execute some analysis and design processes respectively.

This iterative approach is advocated in a number of process models, including: the Rational Unified Process (RUP – see References 5 and 6 for more information) and both ISO/IEC 15288 and STUMPI which will be discussed later in this chapter.

#### 8.2.5.1 Introduction

This section briefly discusses some other iterative models that are in widespread use today. Both are based on standards – the Rational Unified Process, which is a software-based industry standard and ISO/IEC 15288, which is an international systems-based standard. Both standards use very different terminology but, as will be seen from the models, the patterns for both are very similar.

#### 8.2.5.2 ISO 15288

The standard ISO/IEC 15288 is entitled ‘Systems engineering – system life cycle processes’ and was released in its final version at the end of 2002. The standard itself has taken many other standards into account when being written, including: ISO 15504, IEEE 1220 and EIA 632 amongst others. One of the aims of the standard is to consolidate the existing systems-based standard into a single coherent source for systems engineering processes.

The standard advocates an iterative approach to systems life cycle processes, and a high-level view of this can be seen in Figure 8.5.

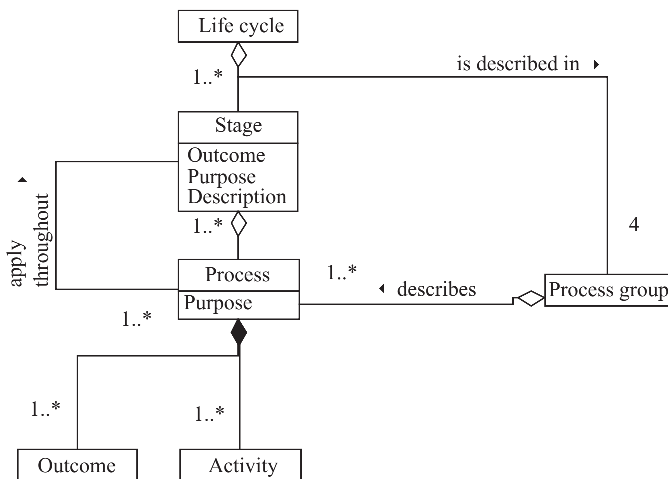


*Figure 8.5 High-level view of ISO 15288*

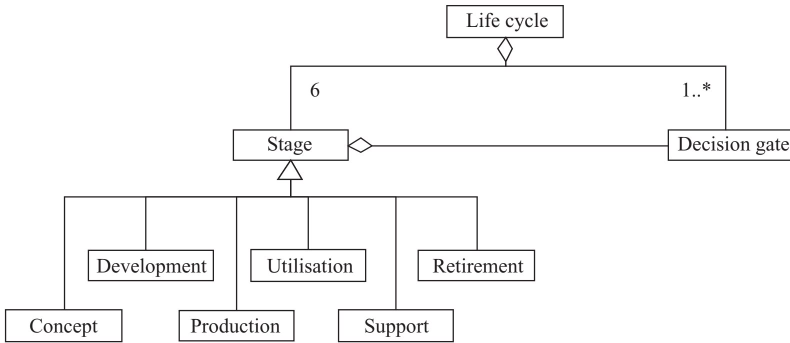


*Figure 8.6 Focus on the stages of ISO 15288*

The diagram in Figure 8.5 shows that the basic unit in ISO/IEC 15288 is the 'Life cycle', which is made up of a number of 'Stage'. Each 'Stage' is made up of a number of 'Process', each of which has a number of 'Outcome' and a number of 'Activity'. The life cycle is also described by four 'Process group', each of which describes a number of 'Process'.

It is now possible to focus on one part of this diagram, in this case the different types of 'Stage' involved in a life cycle.

The diagram in Figure 8.6 shows that there are six types of 'Stage': 'Concept', 'Development', 'Production', 'Utilisation', 'Support' and 'Retirement'. Each of these 'Stage' is made up of a 'Decision gate' that, basically, describes the test that must be made at the end of each stage to ascertain whether or not to proceed with the project and, if so, how. These decision gates are described in more detail in Figure 8.7.

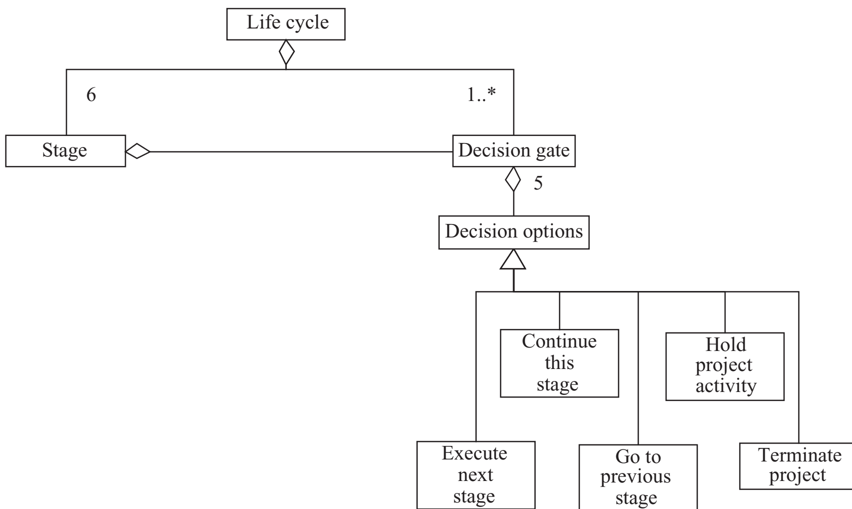


*Figure 8.7 Focus on the decision gates in ISO/IEC 15288*

The diagram in Figure 8.7 shows that each 'Decision gate' is made up of five possible 'Decision option':

- 'Execute next stage' that represents the scenario where all the stage goals have been met.
- 'Continue this stage' that represents the scenario where the stage goals are incomplete and the stage cannot yet be exited.
- 'Go to previous stage' that represents a scenario where something has been discovered that means that the previous stage must be revisited. For example, it may be that new requirements emerge during the development stage to such an extent that the concept stage must be revisited to adapt the requirements model.
- 'Hold project activity' that represents the scenario where something has happened that completely throws out the project schedule. For example, it may be that a major requirement has been omitted, or maybe legislation or technology changes in such a way as to have a large impact on the project.
- 'Terminate project' that represents the worst-case scenario where things have either gone so badly or have changed so much that there is more benefit in stopping the project than in continuing with it.

The decision gates represent the options that must be considered when transiting from one stage to another.

The actual work practices that are executed during particular stages are represented by the basic processes in ISO/IEC 15288 that are categorised according to four different process groups.

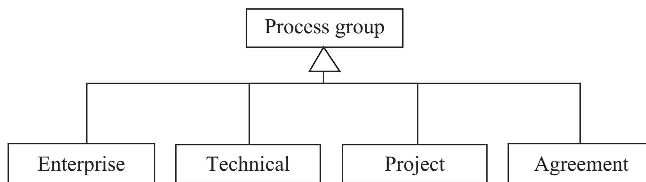


*Figure 8.8 Types of 'Process group' in ISO/IEC 15288*

The diagram in Figure 8.8 shows that there are four types of 'Process group' defined in the standard, which are:

- 'Enterprise' that represents the set of processes applying across the whole organisation, such as environment management, resource management and quality issues.
- 'Technical' that represents the set of processes traditionally associated with engineering activities, such as: stakeholder requirements, design, implementation, etc.

- ‘Project’ that represents the set of processes associated with project management on a project by project basis and also the support services required by other processes, such as risk management, version control, and so on.
- ‘Agreement’ that represents the set of processes representing the nature of the agreement between the customers and suppliers involved in the project.

Each of these process groups is made up of a number of processes and, just to get a feel for the level of information in the standard, Figure 8.9 shows the processes defined for the ‘Enterprise’ process group.

This figure shows a detailed view of the processes that exist in the ‘Enterprise’ process group. Each process is represented as a class and the activities are represented by operations on each class, whereas the outcomes are represented as attributes.

#### 8.2.5.3 The Rational Unified Process (RUP)

The RUP – Rational Unified Process – is a proprietary process that is owned by the Rational Software Corporation and is often sold alongside the UML. The RUP is the proprietary incarnation of the UP – Unified Process – that was developed by Ivar Jacobson as a best-practice software engineering process model.

The RUP process model is intended for use with software development projects and has an emphasis on the technical activities of the development.

The diagram in Figure 8.10 shows a high-level view of the RUP. It can be seen from the diagram that the basic construct in the RUP is the ‘Cycle’ that is made up of four ‘Phase’, of which there are four types: ‘Inception’, ‘Elaboration’, ‘Construction’ and ‘Transition’. Each ‘Phase’ is made up of a number of ‘Iteration’ and between one and nine ‘Core workflow’ take place over each ‘Iteration’.

The term ‘Core workflow’ is equivalent to a process in most other process models and, hence, it is the core workflows that form the basis of the process model. An expansion of the types of core workflow (or process) can be seen in Figure 8.11.

This figure shows that there are two types of ‘Core workflow’: ‘Engineering’ and ‘Support’. There are six types of ‘Engineering’ core workflow, which are: ‘Business modelling’, ‘Requirements’, ‘Analysis & design’, ‘Implementation’, ‘Test’ and ‘Deployment’. Three types of ‘Support’ core workflow: ‘Project management’, ‘Environment’ and ‘Config & change man’.

It can be demonstrated from this diagram that the RUP is strong where technical activities are concerned, as illustrated by the six ‘Engineering’ core workflows, but that it is quite weak when it comes to any of the other activities, represented here by ‘Support’. For example, compare this process structure to that of ISO 15288 in Figure 8.8 and it is clear that the three ‘Management’ core workflows from the RUP cannot hope to have the coverage of the three types of process (each containing a number of processes – as seen in Figure 8.9) contained within ISO 15288. However, it should be stressed that the RUP is intended to be focused on development rather than management, so that it can be easily integrated into existing management systems.

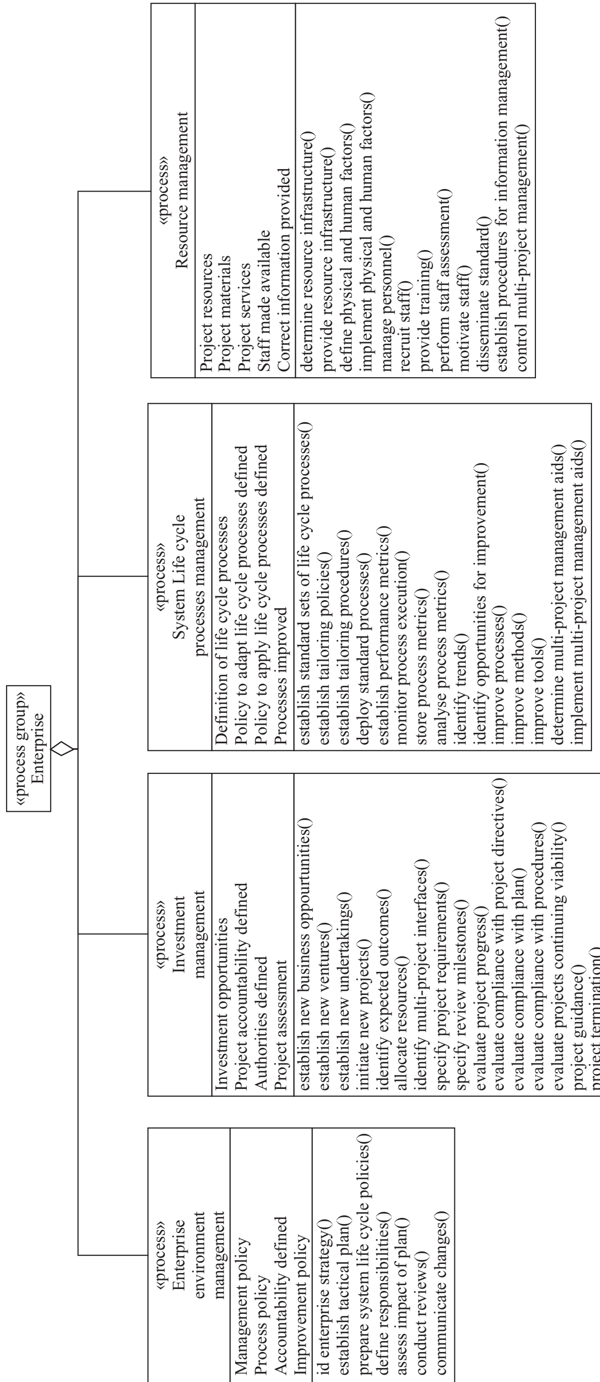


*Figure 8.9 Detailed view of the 'Enterprise' processes in ISO 15288*

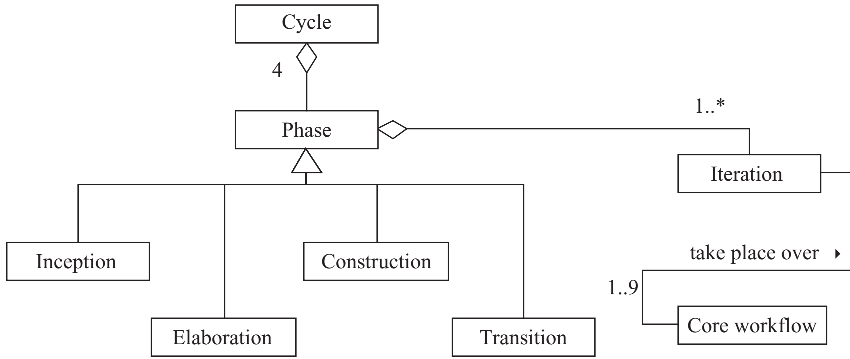


*Figure 8.10 High-level view of the RUP*

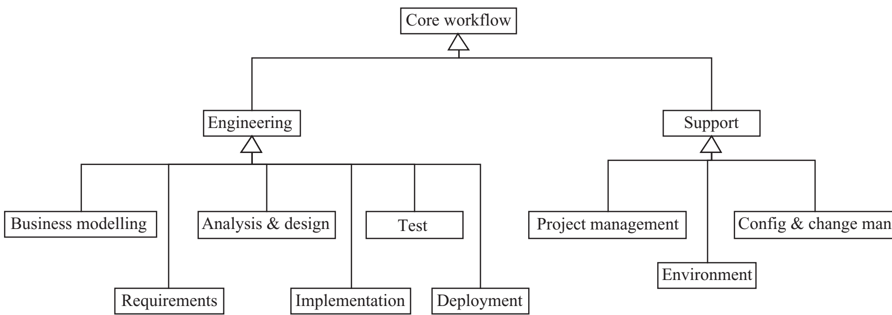


*Figure 8.11 Types of 'Core workflow' in the RUP*

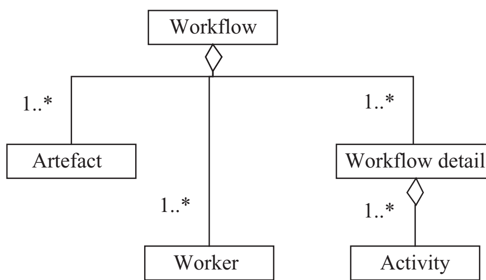


*Figure 8.12 Breakdown of a 'Core workflow' in the RUP*

The diagram in Figure 8.12 shows the breakdown of a single core workflow from the RUP. It can be seen that the 'Core workflow' is made up of a number of 'Artefact', a number of 'Worker' and a number of 'Workflow detail' – each of which is made up of a number of 'Activity'. There is a clear mapping here between the structure of a

core workflow from the RUP and a process from ISO/IEC 15288 – compare diagrams Figure 8.5 and Figure 8.12.

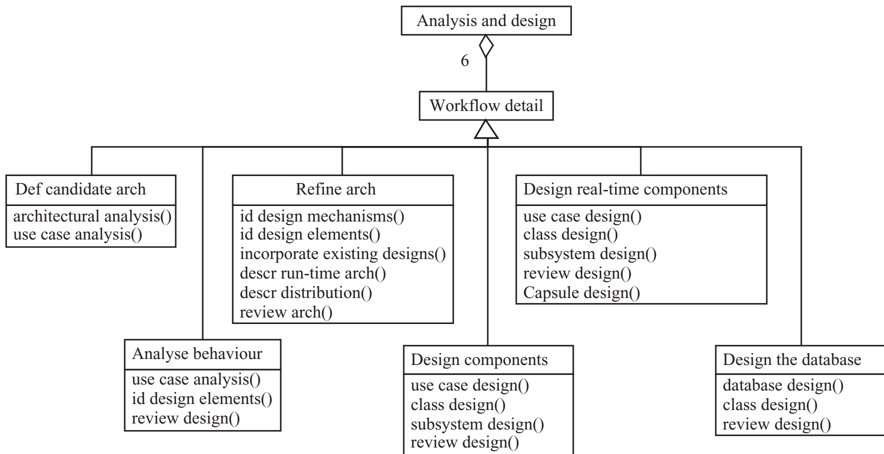


*Figure 8.13 Detailed view of an individual core workflow*

The diagram in Figure 8.13 shows a detailed view of the 'Analysis and design' core workflow. It can be seen that the 'Analysis and design' core workflow is made up of six 'Workflow detail', which are: 'Def (define) candidate architecture', 'Analyse behaviour', 'Refine architecture', 'Design components', 'Design real-time components' and 'Design the database'.

## 8.3 STUMPI – an example of life cycle management

### *8.3.1 Introduction*

This section introduces an example of a life cycle and process model that is based on an iterative approach and is fully traceable back to an international standard – in this case ISO/IEC 15288.

One of the challenges facing anyone involved in running a successful project is that of managing the project effectively and efficiently. There are many approaches to project management in existence at the moment, many of which are very effective. The aim of this chapter is to consider where and how the use of modelling can help with this management. Therefore, the process model that is being followed is irrelevant it is rather the relationship between the life cycle, life cycle model, process model and project schedule that is being emphasised here.

The model used in this section is known as STUMPI – Students Managing Processes Intelligently. The STUMPI model is intended to be applied to student projects for higher education. Such projects present a number of problems, which give rise to a number of specific requirements for any sort of management model, including:

- Student projects have a varied time frame. Many only last a few weeks, whereas others may last up to three years, therefore it is essential that the model reflects this.

- On the whole, there are not many people involved directly in student projects. In many cases, it may be a single individual doing all the work and taking on all the roles.
- Most students would not have been exposed to any sort of project management or process modelling, so it is essential that the model is easy to understand and, hence, use.
- Students are often a fickle bunch, therefore it is crucial that they can see immediate benefits in applying the model.
- There should be a minimum overhead involved with any sort of approach taken by students. For example, in terms of deliverables, most students simply have a log book used to record all project information that then forms an essential input to the formal report-writing at the end of the project. It is crucial, therefore, that all deliverables are recordable in such a fashion whilst still maintaining the current best-practice approach as advocated by the source standard.
- The application areas of the projects will vary wildly and, therefore, the model should be flexible enough to allow the project to run in any application domain.
- From an academic point of view, the students must be able to learn about particular concepts by applying the model. In this case, STUMPI forms the basis for an entire academic module and introduces students to key systems engineering concepts, such as: life cycles and life cycle models, processes and standards, project schedules, stakeholders and people, and so on.

Now that the requirements have been established, it is possible to take a look at the STUMPI model itself and see how it may be applied to life cycle management.

### *8.3.2 The STUMPI life cycle*

The STUMPI model defines a structural life cycle that can be used as a basis for projects.

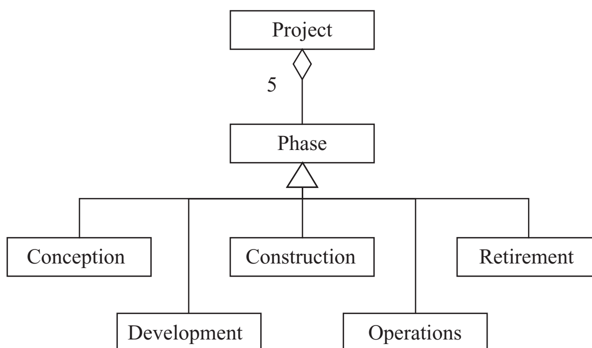


*Figure 8.14 The STUMPI life cycle*

The diagram in Figure 8.14 shows the life cycle defined as part of the STUMPI model. It can be seen that each ‘Project’ is made up of five ‘Phase’, which are: ‘Conception’, ‘Development’, ‘Construction’, ‘Operations’ and ‘Retirement’. It is

interesting to note that many student projects never actually implement all of these phases. For example, many projects are mainly concerned with the conception and development phases where the end result of the project is a deliverable. It may also be that a project is a feasibility study, in which case the life cycle may only have a single conception phase. Of course, some projects do have a complete life cycle – it depends on the nature of the project and this forces people to think about the nature of the project at the outset and ask a few important questions, such as 'how far through the life cycle are we going with this project?' and 'will we be doing all the work?'.

The life cycle shows the 'what' of the project, but it is also vital to show the 'how' of the project using a behavioural view. This behavioural view is known as the life cycle model and will be discussed in the next section.

### *8.3.3 The STUMPI life cycle model*

The life cycle model shows how the life cycle is executed over the course of a project. It shows which phases are executed and in what order. The actual order of the execution of these phases will vary depending on the type of project being run, which goes to show that it is possible to have many different life cycle models relating to a single life cycle.

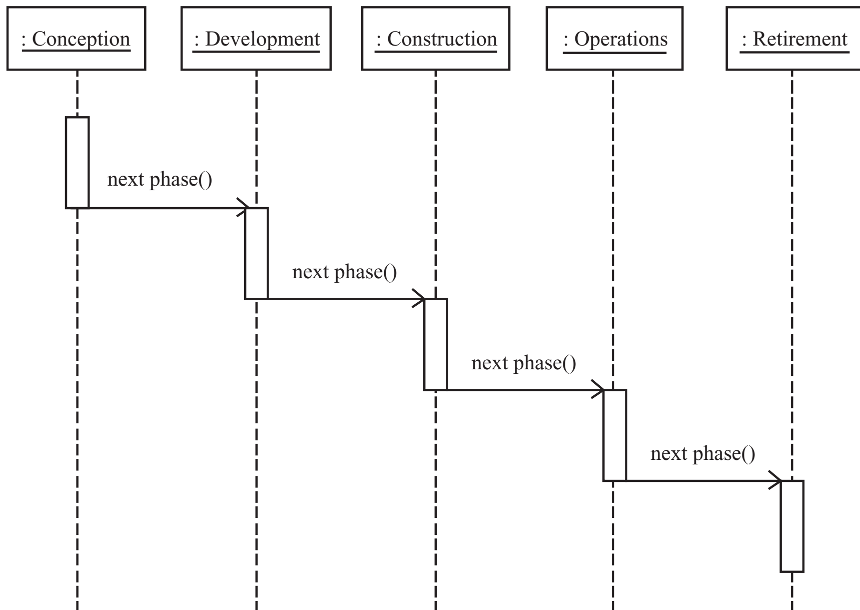


*Figure 8.15 An example STUMPI life cycle model*

The diagram in Figure 8.15 shows an example of a STUMPI life cycle model in the form of a sequence diagram. This could easily have been realised using other

types of interaction diagrams but the sequence diagram has been chosen because of its relationship to project management diagrams, shown later in this chapter.

It can be seen in this example that the life cycle model is following a simple linear execution of phases with no examples of iteration between phases. Iteration can be shown easily on such a diagram by adding more messages between phase instances.

It is important here not to confuse this sort of life cycle model with a more traditional linear life cycle model, as each phase here will be made up of a number of iterations that is not necessarily true for linear models. This can be seen more clearly in Figure 8.16.

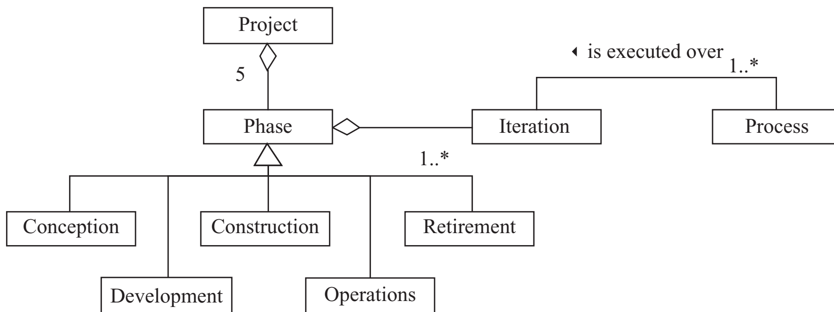


*Figure 8.16 Relating phases to iterations and processes*

This figure shows how phases are related to other STUMPI concepts – the iteration and the process. This diagram is an extended version of the diagram shown in Figure 8.14 but, this time, it can be seen that each ‘Phase’ is made up of one or more ‘Iteration’, each of which has of a number of ‘Process’ executed over it. This is a crucial relationship to identify as it defines the conceptual link between the phases and the processes that will be executed on a project – something that is very often confused.

These processes will now be defined in the next section.

## *8.3.4 The STUMPI process model*

The structure of the processes is key to understanding the nature of life cycle management so it is important that this structure is defined and well understood by all people involved in the project. The process structure for STUMPI is based on that for ISO/IEC 15288 but has been quite severely cut down to enable its use for student projects.

The diagram in Figure 8.17 shows the process structure for the STUMPI model. This is based directly on ISO/IEC 15288 and it can be seen that the ‘STUMPI process model’ is made up of two ‘Process group’, each of which is made up of a number of ‘Process’. Each ‘Process’ is made up of a number of ‘Outcome’, a number

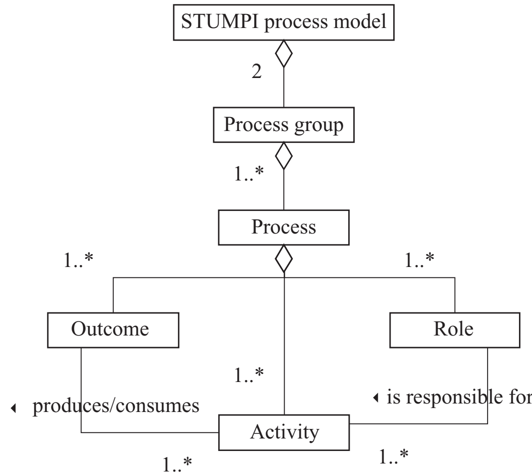


*Figure 8.17 The STUMPI process structure*

of 'Activity' and a number of 'Role'. Note how the pattern here is the same as the pattern introduced in Chapter 6 on process modelling, even though the terminology is slightly different. When the patterns are the same, it often means that both diagrams are trying to describe the same concepts – hence the same or similar pattern.

The source standard, ISO/IEC 15288, identifies four different process groups, which are 'Technical', 'Enterprise', 'Project' and 'Agreement'. The STUMPI model, on the other hand, only has two process groups identified, which can be seen in Figure 8.18.

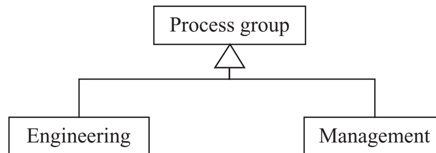


*Figure 8.18 Types of process group*

This figure shows that there are two types of 'Process group' – 'Engineering' and 'Management'. The STUMPI 'Engineering' process group maps directly onto the 'Technical' process group from ISO 15288, whereas the STUMPI 'Management' process group maps to the 'Project' process group in ISO 15288.

The processes for each of these process groups have been defined using the approach described in Chapter 6.

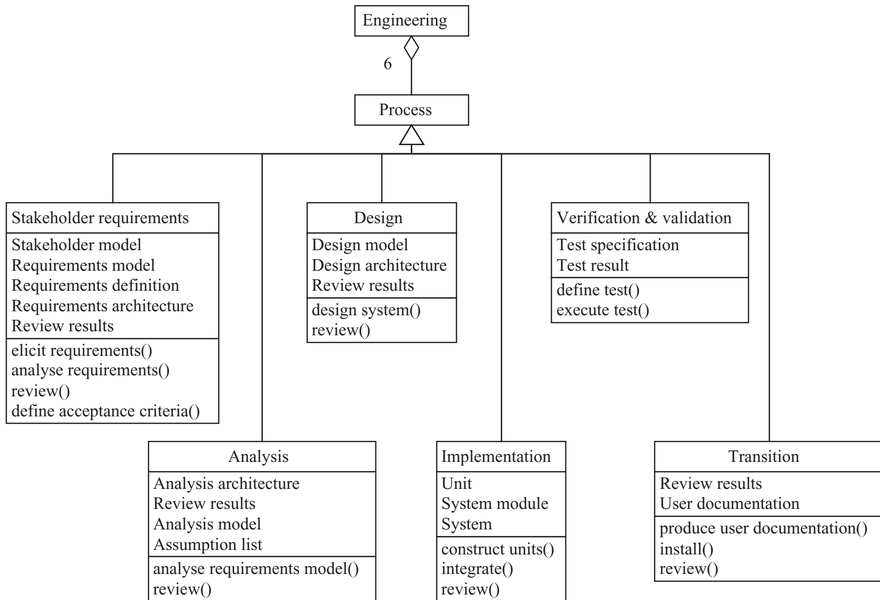


*Figure 8.19 The 'Engineering' processes*

This figure shows the processes that have been identified for the 'Engineering' process group. Each class that represents a process has three things defined:

- The process name, shown by the class name.
- The process outcomes, shown as attributes on the process class. These represent the sort of information that must be recorded in a log book to demonstrate that the activities associated with this process have been executed.
- The process activities, shown as operations on the process class. These represent the things that must be done in order to successfully execute this process.

It can be seen that six processes have been identified here: 'Stakeholder requirements', 'Analysis', 'Design', 'Implementation', Verification & validation' and 'Transition'.

These processes are based directly on those in ISO/IEC 15288 but each one has been simplified in order to make it more usable on small-scale projects. Some of the names have been changed to maintain consistency with concepts that are taught within other subject areas, such as 'Analysis' and 'Design'.

The diagram in Figure 8.20 shows the processes associated with the 'Management' process group, which are: 'Project planning', 'Project monitoring', 'Project write-up' and 'Literature review'. These processes are also based on the source standard but, in this case, two processes have been explicitly introduced since they relate specifically to students. These processes are; 'Project write-up' where a thesis or dissertation is produced at the end of a project and 'Literature review' where all

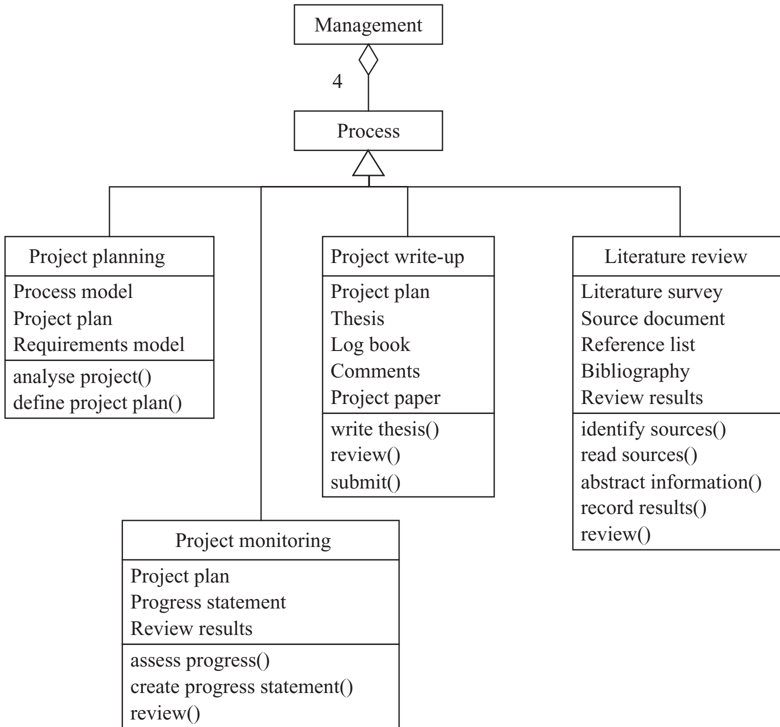


*Figure 8.20 The 'Management' processes*

current literature in the project application domain is reviewed to demonstrate expert knowledge. Although these two processes do not explicitly exist with ISO/IEC 15288, it is still possible to trace individual activities back to activities from various processes within ISO 15288.

It should be stressed that the processes identified here are intended to be used as a starting point for identifying processes for a particular process. Clearly, in the event that a particular project requires a specific process, it must be defined in the same way as those shown here.

The only area that has not yet been covered by the process model, is that of the stakeholder diagram, which can be seen here.

The diagram in Figure 8.21 shows the stakeholders who have been identified within the STUMPI model. There are three main types of 'Stakeholder'.

- The 'Customer' stakeholder who represents the roles that want something from the project and includes: 'User', 'Operator', 'University' and 'Industry'. Again, there may be no industrial involvement, in which case there will be one less stakeholder shown here. Conversely, it may be the case that several more stakeholders are identified here.

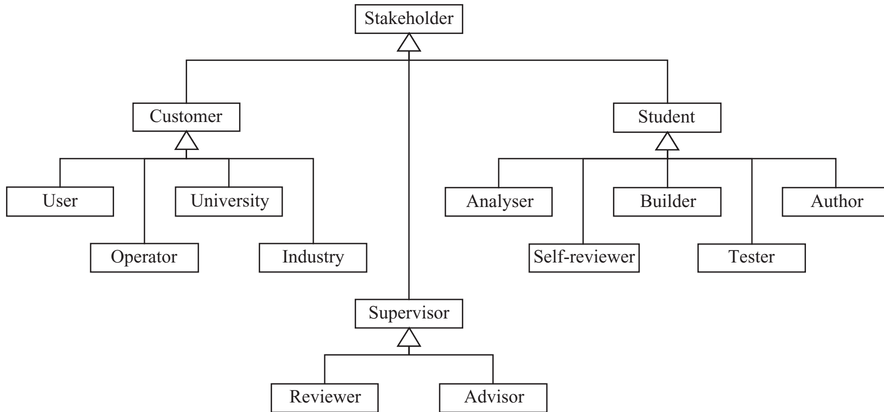


*Figure 8.21 Stakeholder diagram for STUMPI*

- The 'Supervisor' role representing the roles taken on by a supervisor of the project. In this case, there are two roles identified: 'Reviewer' and 'Advisor'. It may be the case that both of these roles are taken by a single individual or, in another case, it may be that there are several supervisors (perhaps both academic and industrial) involved in the project.
- The 'Student' role representing the various roles that the student, or group of students, will take on the project. Again, consider a project with a single student, in which case all roles will have one name associated with them or, on the other hand, it could be a group project where the roles can be 'shared out' amongst the project team.

This stakeholder diagram should be tailored for a particular project and then the responsibility for each role considered. It is known from the process structure in Figure 8.17 that each 'Role' is responsible for a number of 'Activity' and this is shown using an activity diagram to describe the behaviour of each process, as shown here.

The diagram in Figure 8.22 shows the behaviour for a single process from the STUMPI process model, in this case the 'Stakeholder requirements' process. This diagram ties together the role responsible for each activity using swim lanes and also shows the information flow, as well as the ordering of the activity execution.

Now that the life cycle has been identified and the process model has been defined, it is possible to go back to the life cycle model and consider how each one behaves when it is executed. This is done by considering 'iterations' within each phase and their associated process execution.

### *8.3.5 Phase iterations*

Each phase is made up of a number of iterations, as specified in Figure 8.16. Each of these iterations consists of the execution of a sequence of processes, as can be seen in Figure 8.23, which shows an iteration taking place in the conception phase.

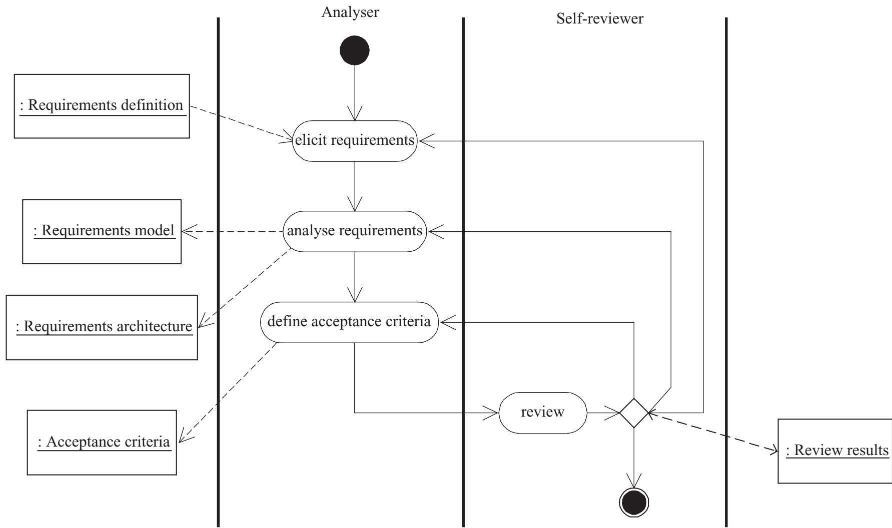


*Figure 8.22 Behaviour of the 'Stakeholder requirements' process*

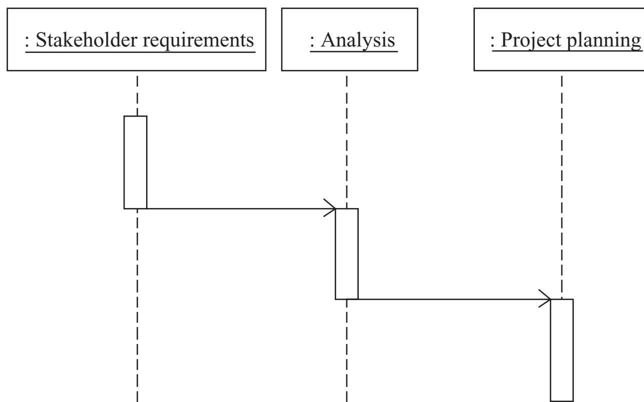


*Figure 8.23 An example of an iteration*

The diagram in Figure 8.23 shows an example of a typical iteration taking place inside a phase – in this case the conception phase. The diagram shows that three processes are being executed: 'Stakeholder requirements', 'Analysis' and 'Project planning'. In this scenario, each process is being executed in a simple sequential manner, but there is no reason why there would not be more iterations between the process execution or, indeed, more processes. In the case shown here, a simple iteration is represented at the beginning of a project when some requirement engineering takes place, followed by analysis of the requirements model. This then enables some project management to take place based on the requirements and analysis models.

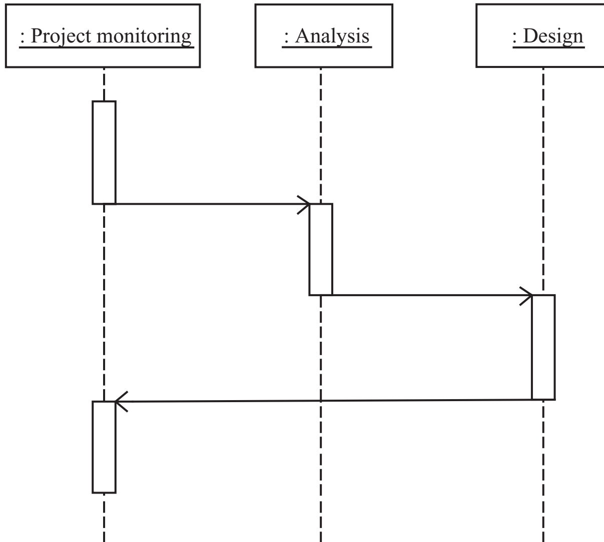


*Figure 8.24 Another example of an iteration*

Another example of an iteration is shown in Figure 8.24, but this time, there is more of an emphasis on analysis and design.

The diagram in Figure 8.24 shows another iteration that has come, this time once again, from the conception phase. Note how, in both examples, there is a mixture of processes from both the 'Management' and 'Engineering' which is typical of an everyday project. Seeing these two iterations, however, raises quite a fundamental point associated with iterations, which is 'why do we need iterations'? If these two iterations were being executed in a sequential order, then why bother showing them as separate iterations, rather than just a larger, single iteration? There are several answers to this but, to start with, if the iterations were considered as being executed in parallel, rather than in sequence, then this would make perfect sense. It may very well be that two teams are working on the project at the same time in which case the iterations must be considered separately rather than as a single iteration. In fact, thinking about the way that projects run in real life, it would be unusual if everyone involved in the project works on exactly the same process at all times and these processes are operated in a sequential manner. This, actually, is one of the reasons that the iterative approach has evolved from the linear approach of life cycle modelling. Therefore, a good way to think about iterations is to consider different groups of people having their own iterations that can be executed sequentially or in parallel. To relate this back to the sort of project that STUMPI is designed for, it is easy to imagine how an individual project with a single worker would have a number of sequential iterations, whereas a project with a team of people working on it would suit a set of parallel iterations.

### *8.3.6 Process behaviour*

So far, the behaviour of the life cycle (the life cycle model) has been discussed along with phases and their associated iterations. It is now time to look at a lower level behavioural view of an individual process by defining an activity diagram for each process, as introduced in Chapter 6.

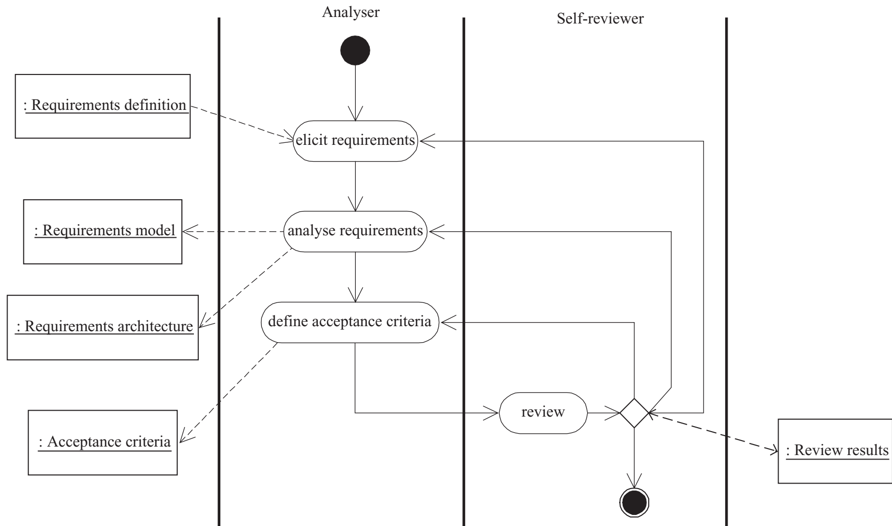


*Figure 8.25 Behavioural view of the 'Stakeholder requirements' process*

The diagram in Figure 8.25 shows a behavioural view of one of the STUMPI processes, in this case the 'Stakeholder requirements' process. The activity diagram shows the order of execution of the activities that are carried out in the process, along with the production and consumption of the outcomes for each activity. Also shown here is the stakeholder responsibility for each activity by the use of swim lanes.

An activity such as the one shown here should be created for each of the processes in the STUMPI model which will lead to a fully-defined set of basic processes. Clearly, these processes may be tailored to suit individual processes using the approach shown in Chapter 6.

## 8.4 Project planning

### *8.4.1 Introduction*

It is all very well modelling an effective set of processes and a life cycle, but how does this tie into the actual project itself? Let's face it, the reality is that whoever is running the project will only be interested in working with traditional project management diagrams such as GANTT and PERT charts, rather than a UML-based

model. Unfortunately, many project schedules that are produced could be filed under the heading of 'fiction' as they are often based on discussions between teams and customers, and dates frequently seem to be plucked from thin air. It is clear that these project estimations should be based on past experiences and future predictions based on actual work activities or, to put it another way, the processes that are executed on a project.

Schedules are, therefore, concerned with a few key estimates:

- What needs to be done, that are indicated on GANTT charts as tasks, sub tasks, sub-subtasks, etc.
- What needs to be produced, in terms of key deliverables and milestones.
- The responsibility for each of the tasks.
- Time estimates for each of the tasks.
- The order of execution for each task.

These concepts can be seen illustrated in Figure 8.26, which shows a typical GANTT chart that is split into two main sections. The main section on the left contains the names of the tasks, milestones, responsibilities and dates. The right-hand section, which is often very long, shows the order of execution of these tasks along with their durations in a graphical form. Time flows from left to right on the time line and the granularity of time is shown along the top of the time line.

The UML life cycle and process models can be used as bases for most of this information, as follows:

- The highest-level tasks map directly to the phases in the life cycle. Therefore, the information needed to start populating the GANTT chart exists within the life cycle model that shows the behaviour of the life cycle.
- The next-level tasks map directly to iterations within each phase. Therefore, the subtasks within each task can be identified and their order known.
- The next-level tasks map directly onto processes within each iteration. Therefore, the sub-subtasks can be identified and their order known from the iteration behavioural model.
- The lowest-level tasks map directly onto the activities that exist within each process. Again, the order of execution is known, which provides the basis for the sub-sub-subtasks.
- A number of milestones are identified for the project that are based on key deliverables already identified within the UML process model. The outcomes in the process model identify what needs to be produced and when (according to the activity diagrams for each process) and also the relationships between them (shown on the information model for the process model).
- The responsibilities for each task, subtask and the like need to be identified which, again, can come directly from the model. All roles on the project have been identified in the stakeholder model and their individual responsibilities have been identified in the activity diagrams for each process. The stakeholder model can

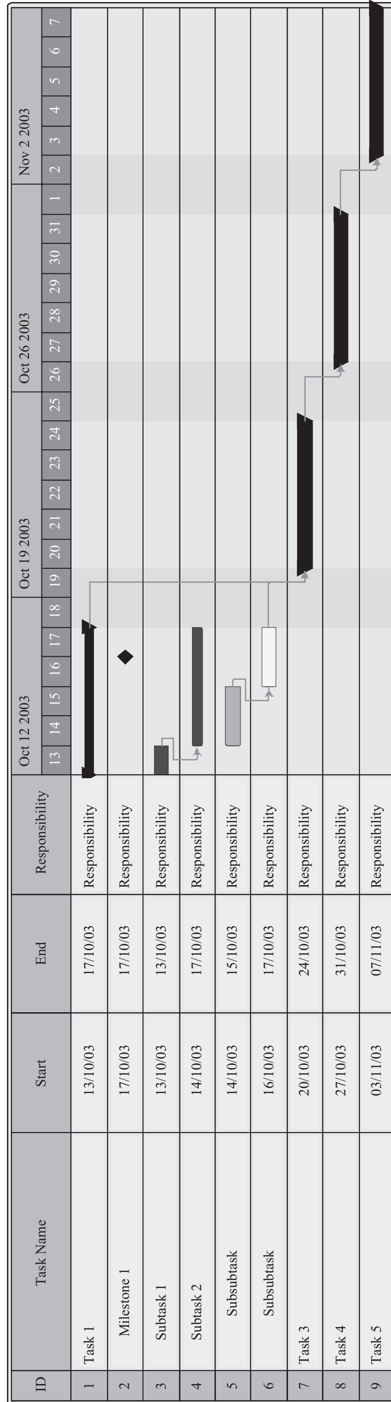


*Figure 8.26 Example GANTT chart*

also be used as a basis for resource allocation for each task, as well as a basis for identifying key skills required for each team member.

All this information is shown on the GANTT chart in Figure 8.27, which shows the same GANTT chart as in Figure 8.26 except this time, instead of using GANTT chart terminology, the terms from the STUMPI process model have been included. Therefore, the relationship between the process model and the GANTT chart is clearly seen.

There is, however, some crucial information missing from this diagram, which is the timing information, and this is often where producing realistic estimates can prove to be difficult. Consider the different levels of detail of behaviour where timing durations need to be estimated – each phase, iteration, process and activity – and then consider the likelihood of being able to estimate, or predict these accurately.

- The time frame for the overall project will typically be known, which provides an overall deterministic time for the entire project. Although it is difficult to accurately predict the timings for each phase, these are often imposed at this stage as key points in the project. It is not unusual, therefore, to specify that the conception phase, for example, should be complete within two months, the development phase within another three months, and so on. In fact, such timings are constraints rather than predictions and it is important that all project activities can be completed within these time limits.
- The time frames for each iteration will typically be unknown and it is very difficult to put any sort of accurate figure.
- The time frames for individual processes may be more possible to estimate but still without much accuracy.
- The time frames for the individual activities, however, should be quite possible to guess accurately, particularly if they are from mature processes that have been executed before. Therefore, this is the most realistic level to start entering timing values into the GANTT chart.

Once the times have been estimated for each activity it is then possible to aggregate these into a duration for each process and hence for each iteration and, finally, for each phase. In many cases, there will be a degree of changing the order of execution of iterations and processes to ensure that the estimated timings will meet the overall timing constraints from the overall project.

It is possible, therefore, to use the process model as a basis for accurate project timings for the project schedule. Clearly, there is still a lot of work involved with getting the timing estimates and the project time constraints to match up, but then by using the process model, there is a realistic basis for predicting the schedule for projects, rather than generating a work of fiction based on, at best, vague empirical knowledge.

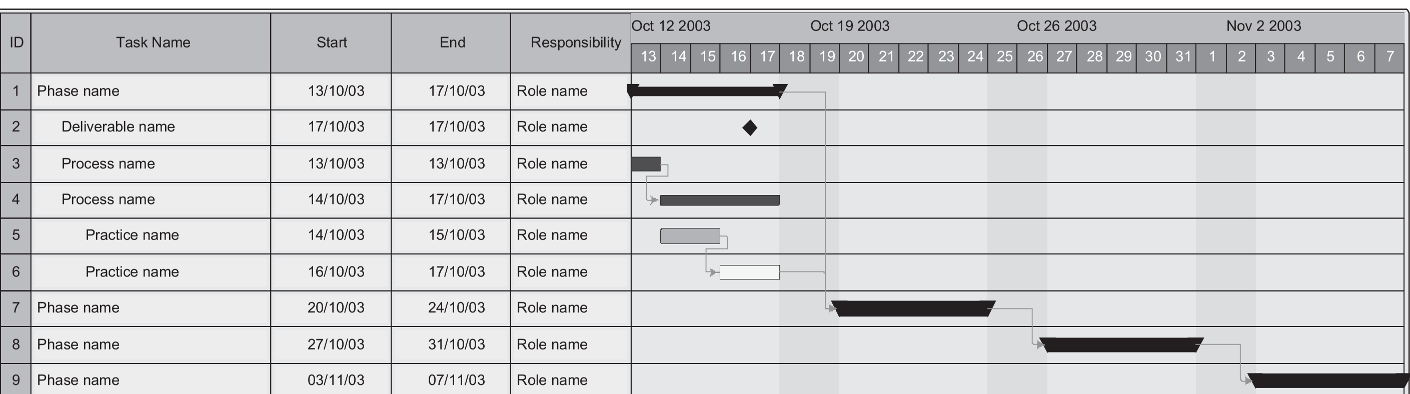


*Figure 8.27 Example GANTT chart with process model concepts*

## 8.5 Summary

To summarise life cycles and life cycle models, the following points should be remembered:

- A life cycle is a static representation of a collection of phases that describe the execution of a project from its initial conception to its ultimate demise.
- A phase implements one or more process. In many life cycles, this will be a one to one relationship, while in others, such as iterative life cycles, this is a many to one relationship using iterations to group processes. It should be stressed that a process and phase are not the same thing.
- A life cycle model describes the behaviour of a life cycle over the course of a specific project.
- Each phase has a number of iterations within it and each of these iterations is made up of a number process executions.
- Each process can be defined in terms of its own behaviour using activity diagrams.
- All the information from the UML model can be used as a basis for the project schedule, which will result in a realistic estimation of the project timings.

Using the UML to represent life cycles and life cycle models is a natural progression from the process modelling that was carried out previously and is a very powerful tool for project management.

## 8.6 Exercise

1. Create a mapping between the concepts in ISO/IEC 15288 and those in the RUP. Are there any differences between the two and, if so, what are they? How does the software-based standard hold up to comparison with the systems-based standard?

## 8.7 References

1 SCHACH, S. R.: ‘Classical and object-oriented software engineering’ (Irwin, Singapore, 1996)
2 STEVENS, R., BROOK, P., JACKSON, K., and ARNOLD, S.: ‘Systems engineering, coping with complexity’ (Prentice Hall, Europe, London, 1998)
3 PRESSMAN, R.: ‘Software engineering: A practitioner’s approach: European adaptation’ (McGraw Hill Publication, Europe, London, 2000)
4 KRUCHTEN, P.: ‘The rational unified process: an introduction’ (Addison-Wesley Publishing, Massachusetts, 1999)
5 INTERNATIONAL STANDARDS ORGANISATION: ‘ISO 15288, Lifecycle management – system life cycle processes’ (ISO)
6 INTERNATIONAL STANDARDS ORGANISATION: ‘ISO 12207, Software lifecycle processes’ (ISO)
7 LOCK, D.: ‘Project management’ (Gower Publishing, London, 2003)

# *Chapter 9*

# System architectures

I never knew what my house was like, until I took a walk outside

*Jools Holland*

## 9.1 Introduction

### *9.1.1 Introduction to architectures*

The term 'architecture' to most people conjures up images of fine buildings and drawings and plans of houses but, as systems engineers, we also use the same term. The term itself may be applied to any system that we develop but, the question is, exactly what is meant by an architecture? Also, once we have a clear understanding of what an architecture actually is, then at what phase of the system life cycle should it be produced and how should it be produced? This chapter looks at several definitions of the term 'architecture' and observes how this has evolved over time to be more encompassing than most people would think. Standards for architectures are also considered and, throughout, the use of the UML is discussed to see how it may help us to define, understand and validate our systems.

### *9.1.2 Architecture – definitions*

There are many definitions of the term architecture. For the purposes of this book, those from two different disciplines – software engineering and systems engineering – are compared and contrasted. The term is used widely throughout each discipline but, as with the scope of the two disciplines, an architecture in systems engineering is wider than one in software engineering.

First of all, consider the following definitions of 'architecture' taken from the world of software engineering:

- 'modules and how they are interconnected' [1].
- 'relationships among major components of the program' [2].
- 'a hierarchy of components according to a partitioning method' [3].

- 'the set of significant decisions about the organisation of a software system, the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour, as specified in the collaboration among the elements, the composition of these structural and behavioural elements into progressively larger subsystems and the architectural style that guides this organisation' [4].

Each of these definitions is primarily concerned with the major elements in a system and their interrelationships and interactions. In fact, the first two definitions are very similar, the only real difference being that the second one explicitly mentions a program, hence relating it directly to the software world, whereas the first one could apply equally to software or systems.

The third definition starts to introduce the concept of a 'method' – in this case a partitioning method, which indicates the need for partitions, assuming that the method here refers to the rationale and approach according to which the partitions are defined.

The fourth, and by far the longest, definition starts to introduce the concept of evolution, by mentioning 'progressively larger subsystems' which implies some sort of natural progression, or evolution, over time. This definition also introduces the concept of 'architectural style' or, to put it another way, the way in which the architecture is developed.

The next question that is addressed is that of how the UML relates to these concepts. Clearly, the first three definitions refer to a structural representation of the software. In this case, 'modules' may be represented as classes or components and the term 'major element' may refer to just about any structural element within the UML. It could be argued here that the concept of a 'method' implies some sort of behaviour according to which the architecture is defined, but this is by no means clear.

The fourth, and last, definition, however, is the first to explicitly mention behaviour and identifies the need for collaborations (upon which, in the UML, all interactions are based) and interfaces (and associated behaviours).

Consider now, two more definitions of 'architecture' this time, however, taken from the world of systems engineering:

- 'the structure of components, their relationships, and the principles and guidelines governing their design & evolution over time' [5].
- 'the fundamental organisation of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution' [6].

These two definitions have much in common with those identified for software engineering – both talk about the 'components' in a system and their 'relationships' and both explicitly mention 'evolution', however, the scope of the definition here is far wider. There is explicit mention, in both definitions, of 'principles' governing how the architecture is developed. This means that the systems definition of architecture is also concerned not just with what is produced, but also how the architecture is produced, something that was only really hinted at in the software engineering

definition. In fact, if one considers each of these definitions in order, they represent an evolution of the term 'architecture' from a simple, narrow definition, to a more complete and wider definition of the term.

### *9.1.3 Not an architecture*

As well as being able to understand what an architecture is, it is very important to be able to recognise what an architecture isn't. Too many people just see an architecture as a simple block diagram with a few lines on it but, in many cases, there is simply not enough information there to justify the use of the word architecture.

One of the main problems with these 'block diagrams' is that there is no actual definitive interpretation for each block. For example, consider the case of any simple block diagram and ask yourself the following questions:

- What do the blocks represent? They represent almost anything, ranging from runtime components to abstract concepts associated with the solution. Also, at which point in the life cycle does this diagram become relevant?
- What do the lines mean? Again, they could represent almost anything, from control flow, to data flow to logical relationships between concepts.
- Is the layout on the page relevant? Some diagrams attribute no relevance to the location of different blocks on a page, whereas some do. Consider, for example, the communication diagram in the UML, where the actual layout of the graphical nodes on the page has quite an impact on the meaning of the diagram.
- Is the diagram representing a structural or behavioural aspect of the model? Or, even worse, is it combining the two aspects of the model on a single diagram? Quite often people mix up both aspects of the model and this can lead to a massive degree of lack of understanding and communication when trying to read the diagram.
- Should the diagram be read from left to right or *vice versa*? Or, is there an obvious flow throughout the diagram that will enable the model to be read and easily understood.
- What level of abstraction is the diagram at? Is this a system-level view, modules-level view or unit-level view? Again, there is simply not enough information on the diagram to make it easily understood.

Many, if not all, of these arguments could be countered by saying 'but I put a key, or legend, on each of my diagrams so that they can be understood by anyone'. This is, of course a valid argument but is also one of the fundamental driving forces behind the creation of the UML. Why create a new graphical language for each model or project when one already exists that has taken many thousands of man years to create and is being constantly evolved to match current technology?

In fact, many of these questions actually relate to basic modelling concepts and just happen to apply to architectures as well. If one takes the latest definitions of the term 'architecture' as taken from current standards, then there is a clear relationship between the architecture of the system and the model of the system.

Of course, by using an established graphical notation such as the UML it is possible to immediately recognise what many of the symbols represent. The UML has a finite set of graphical nodes and graphical paths, each of which has its own distinct meaning and, perhaps just as importantly, they each have established relationships with other elements in the language, hence the basis for consistency exists.

### *9.1.4 Architecture versus model*

There are three underlying reasons why an architecture must exist, which are:

*To aid understanding*: To enable whoever is looking at the system, at whatever point in the life cycle, to understand the what and how of the system.

*To highlight and manage complexity*: Complexity, as has been established already, cannot be eliminated altogether, therefore it is important to be able to spot complexity and, hence, attempt to control it.

*To communicate ideas and data*: Quite often, the architecture is viewed as a way to communicate ideas to other members of the project team, or any other stakeholders and will be used as a basis for any sort of review.

These three ideas do not just apply to architectures, but also are the same three reasons why we want to model in the first place. Therefore, there is a clear link between the architecture and the model as the architecture may be viewed as a special type of model, though with an emphasis on the major aspects of the system.

### *9.1.5 Architectural views*

An architecture may be modelled from any number of perspectives, or 'views'. It is important to be able to understand what viewpoint is being taken when any modelling is being performed as the models for different views can vary dramatically. Just to confuse matters even further, there are also two dimensions to the categorisation of views, as a view may be based on a process or based on a frame of mind. Before a full discussion can take place, consider the following two lists of typical views, bearing in mind that these lists are nonexhaustive and also that some of the terms for each view may differ depending on the reader's background (or viewpoint!). Process-based views include, but are not limited to:

*An analysis or design*: An analysis view shows a model of the problem whereas the design view will show the solution to that problem. The analysis view has a far wider scope than the design, as the analysis view includes elements that are outside the scope of the system, such as stakeholders and their interactions. Indeed, it may be necessary to model interactions that are completely outside the scope of a system in order to understand the behaviour of the system under analysis. The design view, however, has a narrower scope as its boundaries stop at the system interfaces and it is solely concerned with what goes on inside the system.

*An operational view*: An operational view shows the 'how to' of a system and is used as part of the user documentation or instructions for a system. In such a view,

a system will typically be represented at a very abstract level, with only the major components and interfaces of the system represented, as the emphasis will usually be on the user interactions with the system.

*Verification and validation views*: These views are used to check that the system works (verification) and also to ensure that the system does what it is supposed to do (validation). The emphasis for a validation view is focused on a high-level representation of the system and is traceable back to the source requirements of the system. The emphasis for a verification view depends on the type of verification, or test, being carried out. For example, unit-level tests require detailed views of the systems, whereas integration and system tests will require higher-level models.

The other dimension for views concerns the conceptual frame of mind of the stakeholder that requires the view and may include, but is not restricted to:

*A conceptual view*: A conceptual view may be used for helping to gain an understanding of a system. A conceptual view may differ enormously from any other view and may be greatly simplified compared to other views. Conceptual views are particularly useful in the early phases of a project and also when presenting information to third parties.

*A process view*: The process view of a system ties together the systems themselves with their underlying processes. This helps in terms of schedules and resources. For example, the human and physical resources required to implement a particular part of an architecture, along with any timing, or scheduling, requirements of the view (including human resources).

*A physical view*: A physical view will be an accurate representation of the real life, physical components of a systems. This is useful for later life cycle stages and processes that involve delivery, transition, installation, acceptance, and so on. Such views show a high-level representation of the system and typically focus on a structural aspect of the model.

It is possible for a view to combine two of these views, for example, consider the case where a legacy system exists that needs to be built into a new project architecture. It would be possible here to generate an operational (process) view of the system, but this view could be represented physically or conceptually, leading to a variety of diagrams being used to represent the view. It is important to be able to differentiate between different views as, otherwise, looking at several views can lead to a great deal of lack of understanding, hence complexity and hence communication problems.

A key issue here, with regard to applying the UML, is which diagrams to use where? This is discussed in more detail in Section 9.6 where a general appreciation for the types of diagram and their application is provided.

There are many ideas and theories concerning what should be contained within an architecture. Indeed, just by looking at the definitions discussed so far in this chapter, it can be seen that there are many viewpoints on exactly 'what' an architecture is, before even looking at how it should be represented. This section takes a look at two different architectures that have been defined in different standards. These two

standards are not included to provide any sort of recommendation as to how to model an architecture, but are included for discussion only and are here to make the reader think about what should be within and without an architecture.

## 9.2 Example of architecture – Department of Defense Architectural Framework (DoDAF)

The first standard to be considered is DoDAF, which is 'a framework for command, control, communication, computers, intelligence, surveillance and reconnaissance (C4ISR) architecture development' within the US military and is being increasingly used as reference for military organisations in other countries.

This standard defines an architecture as 'the structure of components, their relationships, and the principles and guidelines governing their design & evolution over time' as was previously mentioned. It is not, however, until we actually look at the standard itself that we can gain an appreciation of what exactly is meant by this statement. The actual standard is a hefty beast, weighing in at over 100 pages, which at first impression can be slightly daunting. In order to understand this standard, it is necessary to simplify it and, by definition, in order to simplify something, it should be modelled.

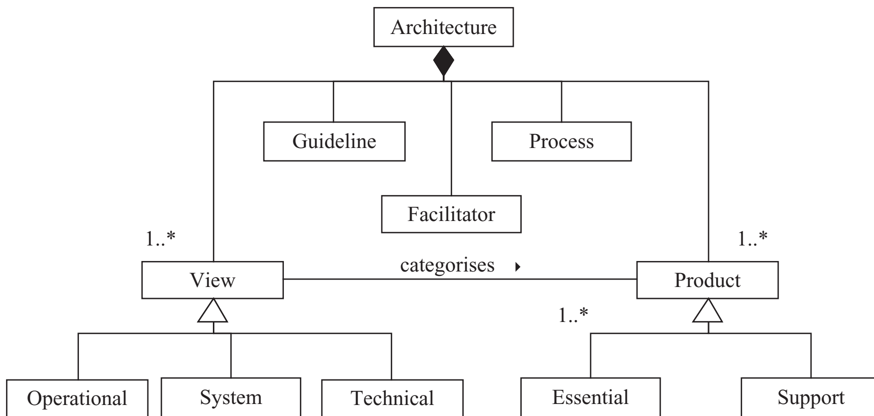


*Figure 9.1 Overview of the framework of an architecture*

The diagram shown in Figure 9.1 shows that an 'Architecture' is composed of a number of 'View', a number of 'Guideline', a number of 'Facilitator', a number of 'Process' and a number of 'Product'.

Some key points to note here are the explicit inclusion of 'Guideline' and 'Process' which ties in with the definition of the architecture provided previously. Also, note the term 'Facilitator', a direct parallel to the term stakeholder as used in this book. The next important issue here is the concept of the 'View' which has three types: 'Operational', 'System' and 'Technical'. Views have already been introduced in this

chapter and are discussed in more detail later. The main point in identifying the various views is so that each 'product' is categorised, which will be revisited in Figure 9.3.

The standard then goes on to discuss different types of architecture that exist within the standard, as illustrated in Figure 9.2.

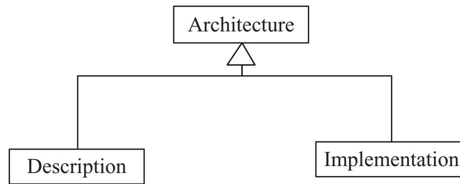


*Figure 9.2 Types of architecture in DoDAF*

The diagram in Figure 9.2 shows that there are two types of 'Architecture' – 'Description' and 'Implementation'. This standard is mainly concerned with the 'Description' architecture that refers to the architecture used during the system development, which is the main focus of this chapter.

As an interesting point concerning specialisation, the entire structure below the class 'Architecture' from Figure 9.1 is inherited by both types of architecture shown here.

The next class to focus on from Figure 9.1 is the one concerning the 'Product' that had two types: 'Essential' and 'Support'.

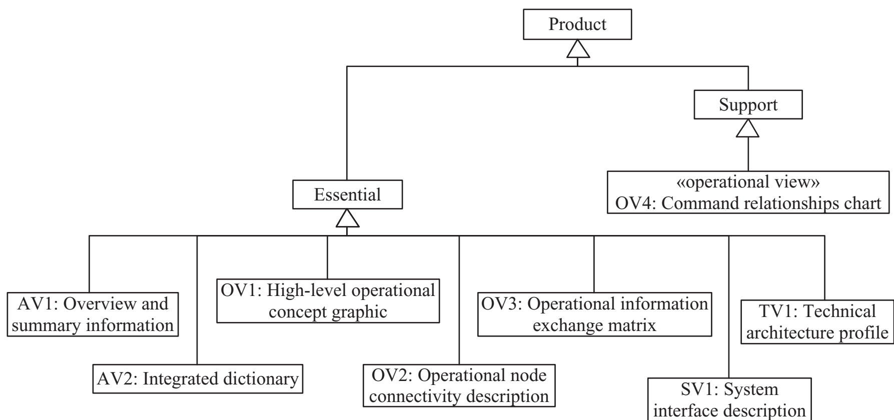


*Figure 9.3 Types of product identified in DoDAF*

The diagram in Figure 9.3 shows the various types of products that have been identified in the standard. Remember from Figure 9.1 that each product was categorised by a particular 'View' that manifests itself in this diagram by the numbering convention adopted at the beginning of each product name. Therefore, products

starting with ‘OV’ refer to operational views, products starting with ‘SV’ refer to system views and products starting with ‘TV’ refer to technical views. So far, so good, but what about the products that start with ‘AV’, as this is not obviously covered by the three types of view? The answer to this lies in the text – as products starting with ‘AV’ refer to all views!

The products themselves are:

- ‘AV1 Overview and summary information’ that applies to all views and provides general information concerning the entire architecture.
- ‘AV2 Integrated dictionary’ that defines all the terms and nomenclature used in the standard and, again, applies to all views.
- ‘OV1 High-level operational graphic’ that refers to a high-level model of the system architecture from an operational point of view.
- ‘OV2 Operational node connectivity description’ that describes major elements within the system and their interrelationships.
- ‘OV3 Operational information exchange matrix’.
- ‘SV1 System interface description’ that describes all the interfaces to the outside world in terms of their structure and behaviour.
- ‘TV1 Technical architecture profile’ that provides the architecture for the solution to be implemented.
- ‘OV4 Command relationship chart’ that describes all stakeholders and their relationships in terms of command structure.

Note how many of the terms and concepts introduced in this standard have been discussed frequently in this book – interfaces, elements and their relationships, stakeholders, and so on.

The model, quite obviously for a relatively large standard, goes into far more detail and consists of many more diagrams covering different levels of abstraction, but the key concepts have been introduced and this is enough to start to draw comparisons between this standard and others.

## 9.3 Example architecture – IEEE 1471

The next standard concerning architectures to be investigated is ‘IEEE 1471 Recommended Practice for Architectural Description of Software-Intensive Systems’. Although the standard is concerned with ‘software-intensive systems’ the principle can be applied to almost any system. In fact, the term ‘software-intensive systems’ is very often used to describe the context of UML which, it has already been established, can be used to model any sort of system and is not just limited to software systems. This also leads to another question, since the same term is being used, does that mean that there will be UML elements within this standard, or is it a coincidence. The answer to this will become all too clear in the following section.

The definition of an architecture in IEEE 1471 is ‘the fundamental organisation of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution’. Here the term

'fundamental organisation' can be taken to refer to the elements in the system and their interrelationships. This definition also alludes to processes through the term 'guidelines governing its design'. Although limiting the guidelines to design is a somewhat blinkered view – this would have been better written as 'development' rather than 'design'. Also, this definition includes mention of 'evolution' implying the growth and change of the architecture over the life cycle of the development of the project.

### *9.3.1 Conceptual model of architectural description*

The standard itself contains a UML description of the conceptual model of the architecture. However, the model itself contains a few fundamental UML errors and, hence, has been redrawn correctly. These errors are mainly concerned with the way that relationships are expressed and the use of role names. The diagram presented in Figure 9.4 is, therefore, a slightly modified version of the original, but at least it is correct.

The standard should be applauded despite the minor technical niggles with the diagram, as being brave enough to use a UML diagram within itself. This is certainly an encouraging occurrence and proves, once again, that the UML is not just used for software development but, as in this case, can be used to help to model standards and architectures.

The diagram in Figure 9.4 shows a high-level conceptual model of the standard. It can be seen that a 'System' fulfils one or more 'Mission' and that the 'System' also inhabits an 'Environment', which, in turn influences the 'System'. Each 'System has an 'Architecture' and has one or more 'Stakeholder'. Each 'Stakeholder' has

*Figure 9.4 Correct version of the conceptual model*

a number of ‘Concern’, each of which is important to one or more ‘Stakeholder’. Each ‘Architecture’ has an ‘Architectural description’ that is made up of number of ‘Model’ and one or more ‘View’, each of which confirms to a ‘Viewpoint’. Each ‘View’ is also composed of a number of ‘Model’ that participate in a ‘View’.

This diagram sums up, rather neatly, all the major elements within the standard itself. Note the use of some of the terms that have been used earlier both in the UML and in DoDAF. One interesting definition here is that of views and models as it is the complete opposite of how they are defined within the UML itself! In the UML, a model is made up of a number of views, whilst here it is the views that are made up of models. This is not necessarily a problem, providing that we know about the difference in terminology. If this difference is not known, then it has the potential to lead to an absolute disaster on the project!

When comparing the terminology with the DoDAF standard, some other parallels emerge. Common terms are used, such as view, whereas in other places, different terms are used, such as ‘stakeholder’ here and ‘facilitator’ in DoDAF.

When looking at the standard further, another term emerges, that is not mentioned on this conceptual model (although it is fairly straightforward to see where it fits in) which is the term ‘intended user’. This is expanded upon in Figure 9.5.

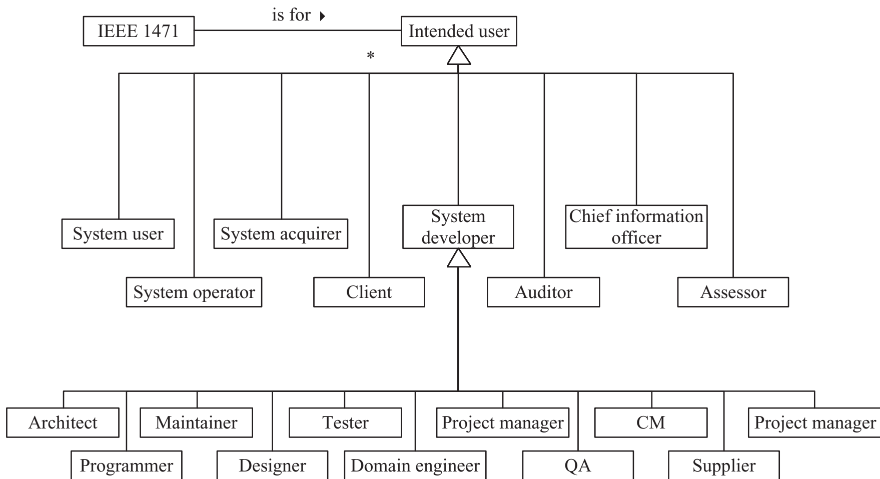


*Figure 9.5 Intended users for IEEE 1471*

The diagram in Figure 9.5 has taken rather the brave step of trying to define all the intended users of this standard. Although theoretically not a bad idea, in practice it is very unlikely that all intended users will be covered. Also, just by looking at the diagram it can be seen that there is a massive imbalance in the model. Notice how one term ‘System Developer’ sits at the same level as all the other intended users, yet has a further 11 types defined for it. It would be more readable and consistent if more levels of abstraction were introduced for the other intended users – similar to when defining a stakeholder view using a class diagram – by adding more structure. It will

also be seen that, when compared with other diagrams in the standard, this diagram will lead to further inconsistencies.

Figure 9.6 focuses on the 'Architectural description' from Figure 9.4 and examines what information is required according to the standard.

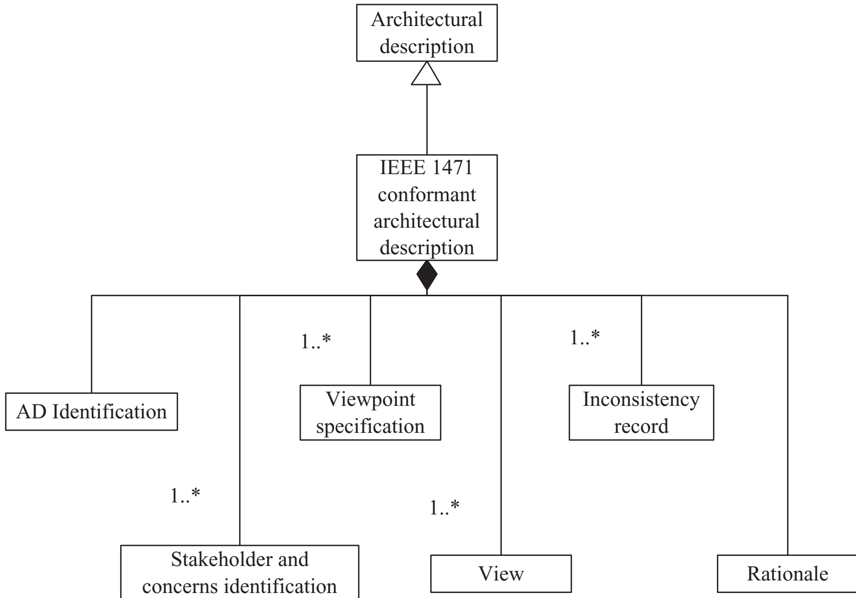


*Figure 9.6 Architectural description for IEEE 1471*

The diagram in Figure 9.6 shows an 'IEEE Conformant Architectural Description' composed of an 'AD identification', one or more 'Stakeholder and concerns identification', one or more 'Viewpoint specification', one or more 'View', one or more 'Inconsistency record' and a single 'Rationale'.

In the standard itself, each of these elements is described in more detail but, for the sake of this example, one of these elements will be expanded upon and the resultant diagram discussed. The element to be investigated is 'Stakeholder and concerns identification', which is shown in Figure 9.7.

The diagram in Figure 9.7 shows that there are two main elements that make up the 'Stakeholder and concerns identification' which, quite unsurprisingly are: 'Minimum stakeholder identification' and 'Minimum concern identification'. If one considers the 'Minimum stakeholder identification' in more detail, it can be seen that it is made up of a number of elements that relate directly back to the 'Intended users' diagram shown in Figure 9.5. However, now there are problems. Note how this diagram looks simple to understand and quite well balanced, but caution must be exercised due to the element 'System developer'. Bear in mind that there are 11 types of 'System developer' as defined in Figure 9.5, so now the diagram becomes far more complex. Also, and more worryingly, one of the elements here refers to 'Maintainer',

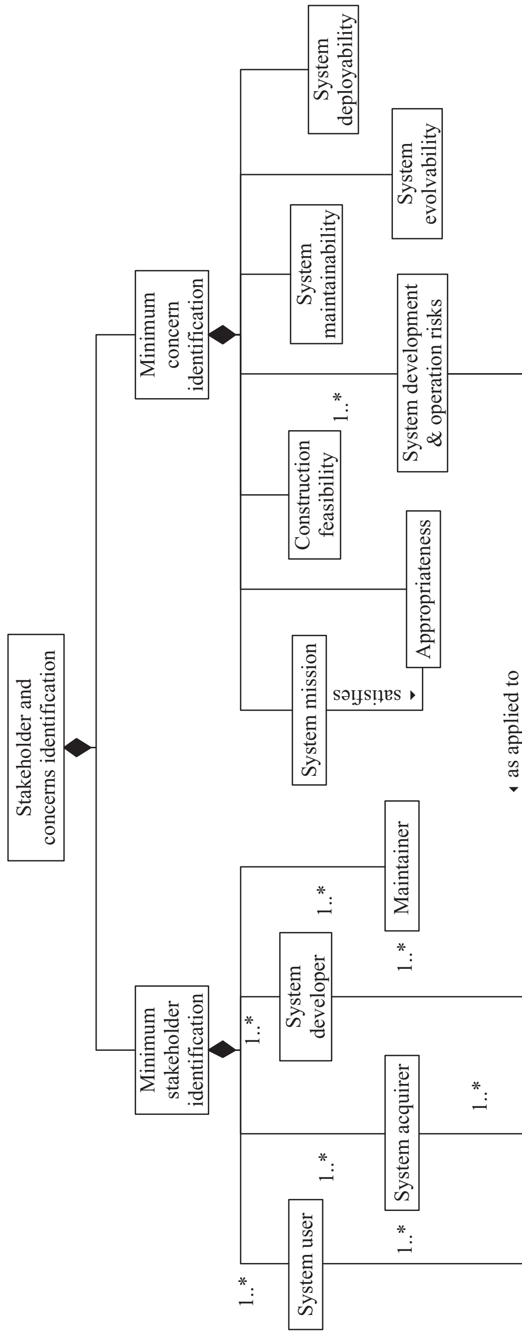


*Figure 9.7 Stakeholder and concerns identification*

which has been defined previously as a 'System developer'. Does this mean that the 'Maintainer' is more important than all the other system developer users, or has someone misunderstood the term 'System developer'. These are problems that are not very easy to pick up from the text, but that are relatively easy to visualise through use of appropriate UML modelling.

## 9.4 Common themes

Having looked at the two standards, albeit at a very high level, the question that comes up is how do the two relate? Clearly they are both aimed at similar audiences and, indeed, situations have arisen where it is necessary to meet both standards, so what is the relationship between them?

There are some common terms, such as 'View'. There are also some different terms for the same concept, such as 'Facilitator' in DoDAF and 'Stakeholder' in IEEE 1471. Any of these common themes that occur lead to a rather sensible assumption that there must be some relevance to these terms, as they are being used in the same context (architectures) in two separate documents. Consider now the scenario where an organisation exists that has its own standard for architectures and also needs to meet both of the standards introduced here, how is one to demonstrate compliance between them all? The answer to this problem lies in the modelling. It has already been shown that modelling, even at a very high level, can throw up inconsistencies within a single standard and that it can also be used as a basis for mapping between several standards, the techniques introduced in Chapter 7 are applied here to demonstrate this.

Standard is really just a guideline and will need to be tailored to meet the individual needs of a particular organisation but, by appropriate modelling, it is possible to draw out common themes and concepts that can be used a basis for an organisation's architectural process or standard.

Bearing in mind the concepts introduced by these two standards, it is now time to discuss how it is possible to use an architecture to add more value to a project by considering project validation.

## 9.5 Validation with architectures

### *9.5.1 Introduction*

In this section the concept of using architectures as part of a validation process is discussed. Validation is concerned with ensuring that the system meets the original requirements or, to put it another way, that the system does what it is supposed to do. Clearly, if this validation is borne in mind throughout the life cycle of the project, then success is far more likely. Whenever the architecture is generated during the life cycle it can be used as part of the validation effort. In order to illustrate this, let us consider an example project and see where the architecture fits in. Before this, however, it is worth considering when, exactly, the architecture should be generated.

### *9.5.2 Generating the architecture*

One issue that is a hot topic of debate is that of when to generate the architecture, or at which point in the development life cycle. There is no simple answer to this question as, like many other hot topics of debate, the answer depends on the specific project or system that is being work on.

It is possible to start to generate the architecture during the requirements stage of the project. In fact, some processes, such as the RUP, actually state this as part of the process model. Other people say that it is best to start to think about the architecture during specification, or analysis of the project. After all, until the problem is understood, so the argument goes, how is it possible to start to define the architecture.

Another school of thought recommends waiting until the problem has been fully understood and is ready to be solved, or as part of the design, before an architecture is generated. In this way, the architecture will be solely concerned with the solution and, hence, provide an optimum solution to the problem.

Unfortunately, there is no clear answer to this and, to illustrate this further, consider the following examples:

- A clean-sheet project that is left entirely up to the project team to define the requirements, perform the analysis and design the solution. In such a case it maybe that the team are looking at cutting-edge, unproven technology to solve the problem, hence it may be that not enough information is available to start to generate the project until the design is being performed.
- A clean-sheet project, but in a domain that is very well understood. It may be that there have been similar projects in the past and a particular approach has proven to be very successful, therefore, it may be that the architecture is started during earlier stages of the project, such as during requirements or analysis activities in a conception phase.
- A project where the actual architecture is specified in the requirements as a constraint. Whether this is right or wrong is not the issue here, but this sort of thing does go on. It may be the scenario where the customer has read about certain best-practice solutions to a problem and has decided to minimise risk by applying a well-proven architecture. Of course, it may also be the case that the customer has ‘buzzword fever’ and has come across a term that is difficult to fully understand and decides to include it in the requirements. In the case of the latter scenario, this can be quite dangerous to the project by constraining the solution very early on in the project without proper justification.
- A project with a large element of legacy systems, where it may be necessary to abstract the architecture of the legacy components of the system and ensure that the new system integrates properly. In such a case, it may be that the original system actually limits the way that the new system may be developed.

It can be seen, therefore, that there is no strict right or wrong answer to the question of when the architecture should be generated, but there are a few pros and cons associated with the choice. The earlier the architecture is generated the more likely it is that the solution will be influenced by the early definition of this architecture. This

can be a good or bad thing depending on the project. Also, if the architecture is left until too late in the development cycle then there is a danger that it may not be easily traceable back to the project requirements. An early definition, on the other hand, can be ever useful for traceability and, hence, validation of the project.

Validation is key to the success of the project as it is the validation that ensures that all requirements have been met. The architecture of a system can fulfil a very useful role in the project validation activity, which is discussed in detail in the following section.

### *9.5.3 Using an architecture for project validation*

For the purposes of this example, consider a development project that is being carried out according to a particular process and particular life cycle model. The actual process and life cycle model being followed are not too important for this example as the emphasis will be on the artefacts of the project and the relationships between them. It is crucial, however, that the artefacts have been identified and defined in an information model that can be represented as a class diagram.

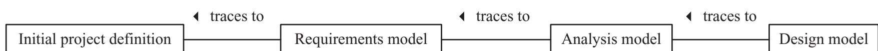


*Figure 9.8 High-level view of a simple information model*

Consider Figure 9.8, which shows four main classes and three relationships between them to represent a partial information model for the project. Each class represents an artefact for the project, in this example they are:

- The 'Initial project definition' that is the rough and ready source of the requirements and is the output of one of the customer–supplier processes. This is not a formal requirements document and is more like a set of statements concerning the project.
- The 'Requirements model' that is the formal statement of requirements for the project and is one of the outcomes of some sort of requirements process. The 'Requirements model' must be traceable back to the 'Initial project definition'.
- The 'Analysis model' that is the formal definition of the analysis, or specification, of the project and describes the problem, based on the requirements. This artefact is the outcome of some sort of analysis or specification process and the 'Analysis model' must be traceable back to the 'Requirements model'.
- The 'Design model' that is the formal definition of the design, based on the analysis and describes the solution to the problem, based on the requirements. This 'Design model' must be traceable back to the 'Analysis model'.

The traceability paths between each artefact are shown as simple association on the diagram.

This diagram may now be looked at in more detail, by looking at each of the artefacts in turn and looking at the lower-level relationships between different parts of each artefact. It has already been established that there is a high-level relationship

between each artefact, but now it is time to look at a lower level and explore exactly what the nature of the relationships is.

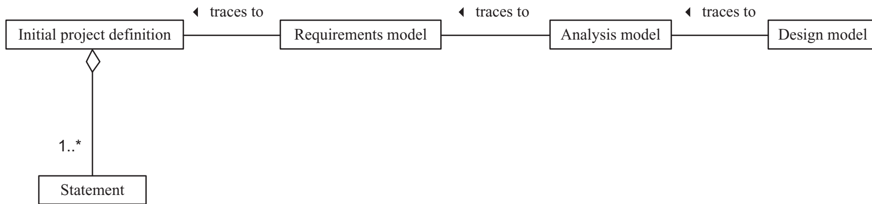


*Figure 9.9 Lower-level view of the 'Initial project definition'*

The diagram in Figure 9.9 is very similar to that shown in Figure 9.8 except, this time, it can be seen that the 'Initial requirements definition' is made up of a number of 'Statement'. Each statement will be analysed and, eventually, will evolve into the full, formal 'Requirements model'. However, there is more to the 'Requirements model' than a fuller description of each statement, so this can now be looked at in more detail and is shown in Figure 9.10.

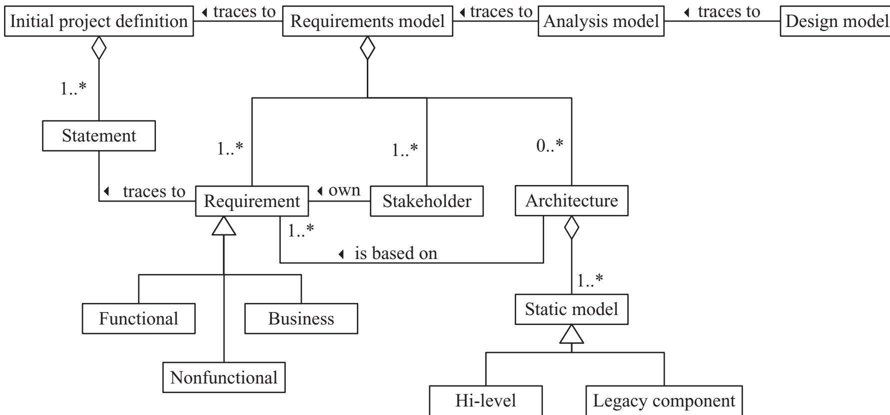


*Figure 9.10 Lower-level view showing the structure of the 'Requirements model'*

Figure 9.10 shows the same diagram as the previous two, but this time the 'Requirements model' has been broken down to provide a lower-level description. This is not intended to be a definitive model of what constitutes a requirements model, but is based on good practice and is shown here for illustration purposes only.

The 'Requirements model' is made up of a number of 'Requirement' which, as described elsewhere in this book, has three types: 'Functional', 'Nonfunctional' and 'Business' and a number of 'Stakeholder', each of which owns a number of

'Requirement'. The focus here, however, is on the 'Architecture' which has appeared as part of the 'Requirements model'. The argument concerning whether the architecture should be present at this early stage of the development process has already been discussed and is addressed in this diagram by indicating a multiplicity of '0..*' which shows that the architecture may, or may not, be present.

At this early stage, the architecture itself is relatively simple and consists of a 'Structural model' that has two types: 'High-level' and 'Legacy component'. The 'Structural model' itself is based on a number of requirements. For example, it may be that the system is an evolution of a preexisting system, therefore there will be legacy parts of the system that must exist in the new version. Also, it may be that a certain amount of partitioning can be carried at this point, maybe in terms of functionality, which may lead to major components of the system being identified.

The important aspect of this diagram, with regard to validation, is the relationship between the architecture and the source information for the project – the 'Initial requirements definition' – that can be traced via the requirements. Clearly, it is crucial that the architecture is related directly back the requirements of the system at this stage to ensure that the elements within the architecture are valid. One common mistake that is often made on projects is to introduce too much information into the architecture that cannot be traced back to requirements – very often because this is how work has been carried out before or maybe people start to think about solutions far too early in the project. A crucial part of any process, therefore, is to validate each element in the architecture at this point.

The architecture will evolve as the project progresses and it is also important to capture this evolution through effective configuration control of the models. This point can be seen more clearly in Figure 9.11, as another evolution of the architecture is introduced that must be traceable back to the previous architecture.

Figure 9.11 shows the breakdown of the 'Analysis model' that can be seen to be made up of one or more 'Scenario' and one or more 'Analysis architecture' that in turn is made up of a number of 'View'. It can also be seen here that each 'Scenario' validates a number of 'Requirement' and also the 'Analysis architecture'. Again, the architecture is a key part of the validation process and is being validated consistently throughout the development life cycle.

The final diagram, Figure 9.12, shown here for completeness takes the architecture through to the 'Design model' and, again shows traceability back to other artefacts.

It shows that the 'Design architecture' is made up of a number of 'View' that has three types, 'Physical', 'Process' and 'Operational'. The '{incomplete}' constraint here shows that there are more views that this – bearing in mind that this is a simple example.

It is important to bear in mind that there may be any number of views to represent different aspects or contexts of an architecture.

### *9.5.4 Conclusion*

To conclude, therefore, it should be made clear that the architecture of a system can be a very important factor when considering the validation of the system. As the

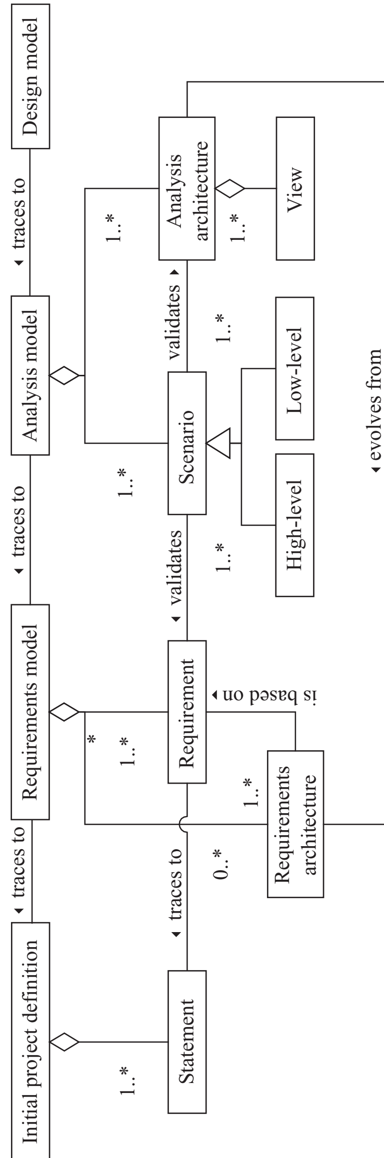


*Figure 9.11 Lower-level view focusing on analysis*

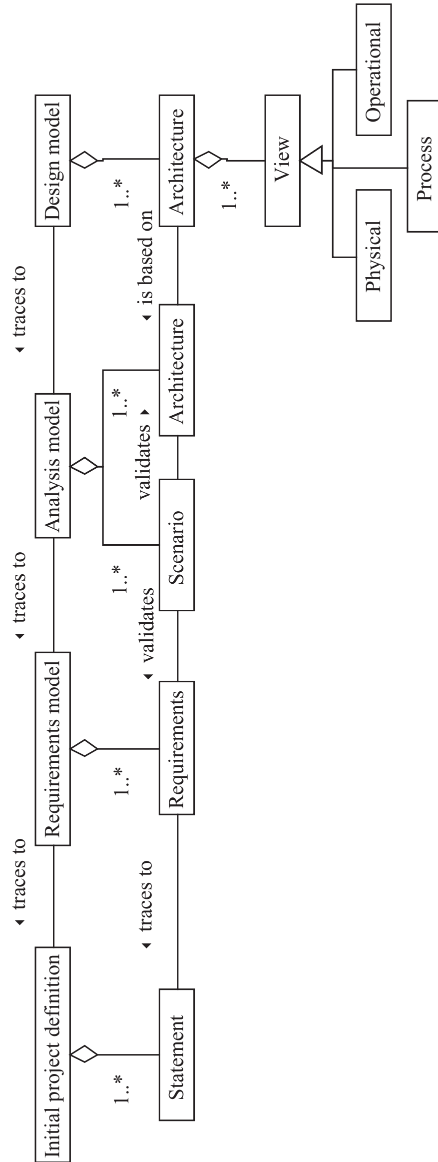


*Figure 9.12 Lower-level view focusing on the design model*

validation of a system is inexorably linked to the quality of a system (see Chapter 7 for more details) it follows on that the architecture is an important part of system quality. Although having a system architecture does not necessarily guarantee the quality of a project, it is significantly more difficult to demonstrate quality without one.

It should also be remembered that when the architecture is generated, is very much dependent on the type of project, but caution must be exercised, otherwise the architecture may constrain the actual solution to the problem.

## 9.6 Applying the UML

### *9.6.1 Introduction*

The concepts discussed so far are concerned very much with the essence of an architecture and very little has been said about how exactly to apply the UML to generating a system architecture. This section looks at the different aspects of the architecture that have been discussed so far and suggests some diagrams that may be appropriate for representing the architecture. As with all applications of the UML, there are very few hard and fast rules regarding which diagrams to use under which circumstances, so the following information should be used as a guide.

The breakdown of the architecture is based on the typical information associated with architectures discussed in the previous section concerning project validation. As such, the following headings are based on the various types of architecture that have already been discussed. It should be stressed, once again, that these are guidelines only and are not carved in stone.

### *9.6.2 The requirements model architecture*

The requirements model architecture is generated based on the requirements and stakeholders of the project. The requirements and stakeholders may be represented by use case diagrams, with use cases to represent requirements and actors to represent stakeholders and class diagrams and with a hierarchy of classes to represent stakeholders.

Alongside the requirements model sits the requirements architecture model. In the example shown here, two elements have been identified; the high-level structural model and the legacy components, which may be realised in the UML as follows:

*High-level structural model*: This will typically contain the high-level elements of the architecture. As the name implies, we are very much looking at structural, or static, diagrams for this part of the model. Regarding structural diagrams, almost any could be used. The class diagram is a main contender and forms the heart of most UML models. The class diagram represents the system as a whole and then some of the subsystems as classes, with an emphasis on the composition of each class. Alongside this class structure may sit a composite structure diagram. Remember that a composite structure diagram is very much like a class diagram but with an emphasis on the logical relationships between different elements in the hierarchy, whereas the

class diagram emphasised composition. The composite structure diagram is also used to identify the major collaborations at this point, which will be used as a basis for the scenarios later on. It is also possible that a package diagram may be used at this point with each package representing a subsystem of the main system. This is one approach that is favoured by some people, but not others. An object diagram, although possible, is not very likely to be used at this point. Component diagrams may be utilised, especially when used in conjunction with package diagrams. Caution must be observed when starting with component diagrams on a clean-sheet project as it is easy to slip into a functional-decomposition style approach. Several methodologies advocate this component-based approach, including Cheeseman [7]. Deployment diagrams are far more likely to be used for legacy components, rather than the high-level structure.

*Legacy component*: Many real-life systems projects will not start with a clean sheet of paper and will have to integrate with an existing environment in some way. In the case of a system upgrade or a new subsystem being developed, then the project will include some legacy components. It is important to have an understanding of the workings of any legacy components within the system, even if it is only at a high level. In many cases, it may only be necessary to have an understanding of what the legacy elements are and what they do in terms of inputs and outputs, often referred to as the terminal conditions of a legacy element. In such cases, component diagrams make ideal candidates for the modelling. Also, particularly when the environment is important, it is useful to look at deployment diagrams to model how the system fits into its target environment.

It should be stressed that the example shown here has only included structural elements of the architecture and, as has been stated many times in this book, it is essential to have both structural and behavioural aspects of the model considered if one is to get anything that approaches a correct model. There are two answers to this – one is that these diagrams could be traced back to the requirements model, represented as use case diagrams hence providing a behavioural aspect to the model. The other answer is, of course, that there may be some high-level behaviour defined at this point. This is completely feasible but it should be remembered that the more 'how' included in a model (behavioural aspects) then the more constrained the architecture will become. It is often desirable to know 'what' needs to be done, with the 'how' left until later in the development.

### *9.6.3 The analysis model architecture*

The main aim behind any sort of analysis is to understand and in the context of system development, the analysis, or specification, is all about understanding the problem. It is almost impossible (without relying on luck) to solve a problem properly and elegantly if it is not understood in the first place. The following information was identified in the example shown in this chapter:

*High-level scenarios*: A high-level scenario will basically consist of a lifeline that represents the system along with several other lifelines that represent stakeholders in

the system. Before we even start thinking about the scenario itself here, we should be thinking about consistency with the rest of the model. How should the players (stakeholders, lifelines, etc.) in each scenario be identified and how does this relate to the rest of the model? If composite structure diagrams have been used effectively, the several collaborations will already have been identified in the structural aspect of the model that can be used as a basis for the scenarios. If this is not the case, then the stakeholders should be consistent with both the stakeholder view (realised by a class diagram) and the actors in the use case diagrams. The main emphasis of a high-level scenario is on the behaviour between the system, treated as a single entity and the outside world, represented by actors. Another consistency check that can be performed relates back to the context of the system (realised by a use case diagram with a system boundary) where a number of interfaces will already have been identified by associations crossing the system boundary. Each of these interfaces will be realised by a set of interactions that describe the behaviour of each interface. The diagram that can be used to realise these scenarios will be any, or any combination, of the interaction diagrams – sequence, collaboration, timing and interaction overview diagrams. Each scenario should then be traceable back to a use case, as each scenario is an instance of a use case.

*Low-level scenarios*: Low-level scenarios obey the same rules as the high-level scenarios in terms of consistency and their usage, except that this time the emphasis is on what goes on inside the system. Rather than a lifeline representing the entire system, this time each lifeline will represent a component or lower-level element within the system. Of course, it is essential that the low-level scenarios are consistent with the high-level scenarios which will help towards the entire traceability of the project.

*Analysis architecture*: The analysis architecture will show a high-level representation of not only the system, but also any thing else that is contained with the problem domain. For example, it may be that some system behaviours need to be modelled for legacy systems, to see exactly how they interact with the current system. It may also be, and is often the case, that some of the human stakeholders' behaviours also need to be modelled. This modelling of stakeholders, whether they are legacy systems or people, will typically use lower-level behaviour modelling techniques, such as state machine diagrams and activity diagrams. Once each has been modelled, it is then possible to ensure that their behaviours are consistent with the high-level behaviour of the system, as described in the high- and low-level scenarios.

*Views*: Various views will be discussed in more detail in the next section.

The main aim of the analysis is to understand the nature of the problem before it can be solved. Any or all of the above techniques may be used to realise this, including any from the requirements architecture model. Remember that the architecture should evolve over time and there is no strict order to the way that this evolution will progress. Some systems will have more of an emphasis on behaviour, such as communication systems, whereas others will have more of an emphasis on static aspects of the model.

### *9.6.4 The design model architecture*

The main aim of the design architecture is to solve the problem based on the analysis model. The solution domain, or the context of the solution, is actually far smaller than the problem domain, as its context stops at the system boundary and is only interested in the interfaces, rather than what is on the other side of the interfaces. The design model architecture may use any or all of the techniques mentioned previously in this section but, in the example shown previously, the design architecture model is focussed on the various views of the system. There are many views that may exist and the following is just a selection of some possible options.

*Conceptual view*: The conceptual view is very important to understanding the nature of the whole problem and is, therefore, fundamental to the analysis architecture. It is also important in the design architecture as it may be that the design view is more concerned with concepts of various aspects of the solution, rather than the general appreciation of the problem. When modelling concepts, it is a good idea to make the models as simple as possible. It is often the case that people represent the entire system as just a few classes and then represent the behaviour of each class as simple state machine diagrams or activity diagrams. It is almost always the case that such simple conceptual diagrams actually evolve into a more complete set of scenarios and structural diagrams, but if the simple view aids understanding of the basics of the system, then it is valid. Remember, keep asking yourself if you are adding value and, as long as the answer is yes, then the modelling is valid.

*Physical view*: The physical view is a real-life representation of the major elements in the system. This is generally used for one of two purposes – to model what is delivered to the customer in a 'what's in the box' fashion and also to start to analyse legacy systems by trying to model the major elements in a preexisting system. The main types of diagram used here are the deployment, component and, where applicable, object diagrams. If the behaviour needs to be modelled, then it will usually be a higher-level behavioural diagram, such as any of the interaction diagrams.

*Process view*: The process view is concerned with how the system will operate within the real world. How is the system operated, what resources are required, and so on. The diagrams used here will be those used for any type of process modelling as described in Chapter 6. Therefore, the relevant diagrams would be: class diagram (for structural process views, information view and stakeholder view) and state machines and/or activity diagram (behaviour of each process). These processes must then be consistent with the overall system behaviour as defined in the various scenarios previously generated.

*Operational view*: The operational view is concerned mainly with how the system is operated from the point of view of users and operators of the system. Again, the diagrams used here are the same as those used for process modelling, with the addition of interaction diagrams to show the high-level behaviour of the system. In terms of knowing if the operational view covers all aspects of user operation, this can be validated quite simply. Refer back to the original use case diagram and

identify all use cases that interact with stakeholders at the user/operator level, which may be identified from the stakeholder model. Then, ensure that enough scenarios are generated to ensure coverage of each of these particular use cases. Again, consistency of the diagrams is being applied to add yet more value to the model and to increase confidence that the model is correct.

*Verification and validation view*: These two views are crucial to the success of a system and for demonstrating its quality. The verification views demonstrate that the system works, whereas the validation views demonstrate that the system does what it is supposed to do. As verification is more concerned with demonstrating that the system works, the diagrams used will tend to be lower-level diagrams, such as state machine/activity diagrams in conjunction with structural diagrams. The validation view, on the other hand, is more concerned with meeting the original requirements, therefore the diagrams used will be interaction diagrams that trace directly back to the use cases (and, hence, requirements) of the system. Of course, now that we have the lower-level view (verification) and higher-level view (validation) it is then possible to ensure that the two are consistent.

The views shown for the design architecture are just examples and, naturally, many more views may be used. It may also be possible that within the context of a particular process or methodology, the same name will be used for views with completely different aims and purposes. Again, it should be stressed that the information here is for guidance only.

## 9.7 Conclusions

System architectures are about much more than just elements in a system and how they relate to one another. The architecture is a living beast that will evolve over the course of a development life cycle. Like all beasts, it must be understood and controlled in some way, hence the application of the modelling. Also included in the architecture is not just what is in the system, but also the approach or process to how the system is developed. The process then ties together all the project artefacts, or deliverables, and identifies their interrelationships and how they should be used. The process is also key to establishing traceability, validation and hence, quality.

There are many sources of information for defining architectures, ranging from traditional software architectures that tend to have quite a narrow scope towards large, system-based architecture models that are intended to be applied to massive projects. Two examples are looked at (DoDAF and IEEE 1471) and, although neither is perfect, they each represent a good starting point for architectural modelling and identify several key points and common issues between the two. Any other approach may also be modelled in the same way as these two standards, simply by applying the techniques discussed in Chapter 6 to any source standard or model. The resultant UML models may then be used as a basis for quality checking back to the source and may form the basis of an assessment or audit.

One of the main aims of the chapter is to demonstrate how an architecture may evolve over the course of a system development and to identify some possible elements at various points of the project. This simple example was then discussed in more detail in terms of which UML diagrams could be applied to what aspects of the architecture model. The example shown here is just that – an example – and the information concerning the diagrams is simply a guideline that should provide the basis for thinking about which diagrams apply where.

Finally, an architectural model will not ensure quality, but quality will be significantly more difficult to demonstrate without an architecture.

## 9.8 References

1 SCHACH, S. R.: 'Software engineering with Java' (McGraw-Hill International Editions, New York, 1997)
2 PRESSMAN, R.: 'Software engineering: A practitioner's approach: European adaptation' (McGraw-Hill Publications, Europe, 2000)
3 SANDERS, J., and CURRAN, E.: 'Software quality: Framework for success in software development and support' (ACM Press Series, Addison Wesley, 1994)
4 OMG: 'Unified modelling language superstructure – Version 2.0' (Object Management Group)
5 DoDAF: A framework for command, control, communication, computers, intelligence, surveillance and reconnaissance (C4ISR) architecture development
6 INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS: 'IEEE 1471, Recommended practice for architectural description of software-intensive systems' (IEEE, New York)
7 CHEESEMAN, J.: 'UML components: A simple process for specifying component-based software (Component-based Development)' (Addison Wesley, 2000)

*Chapter 10*

# Extending the UML

there is nothing permanent, except change

*Heraclitus*

## 10.1 Introduction

This chapter looks at how the UML can be extended by defining a user-defined 'profile'. In order to illustrate the use of profiles, an example will be considered that continues the discussion concerning process modelling from Chapter 6.

The UML is intended to be a general-purpose modelling language and, as such, it is inevitable that situations will occur where the UML does not suit the application very well due to the fact that applications are specific, whereas the UML is generic. In such situations, it is possible to tailor the UML language by defining a 'profile'. A profile is a declaration of a set of extensions to the UML language that will enable the UML to become more application- or domain-specific. These profiles may be defined by an individual or organisation, usually for a specific application, in what is known as a user-defined profile, or it may be a well-accepted, widely-used profile for a specific domain. These domain-specific profiles may be used on an international level and, indeed, the OMG is currently working towards developing a standard set of profiles for specific domains. Examples of domains that may be considered include: real-time application, systems engineering (the so-called SysML – not yet developed enough for inclusion here), process modelling etc.

The use of profiles is, however, a double-edged sword as the whole point of the UML is to enable people to speak a common language. As soon as this language is altered, or tailored, for a specific usage then it is assumed that whoever is reading the model also knows about these extensions. Caution must be exercised otherwise it is possible to end up with a language that bears no resemblance whatsoever to the original UML – including the symbols!

## 10.2 Creating a profile

### *10.2.1 Introduction*

In order to create a profile, it is necessary to create a new package that will contain all the tailored aspects of the UML, or extension mechanisms. There are three such extension mechanisms:

*Stereotypes*: Stereotypes are, by far, the most widely-used of all the UML extension mechanisms and represent a powerful way to define new UML elements by tailoring the UML meta-model.

*Constraints*: Constraints allow new rules to be created as part of a model, that add more precision to different modelling elements. There are three main types of constraint: invariants, preconditions and postconditions, each of which will be elaborated upon later in this chapter.

*Tagged values*: Tagged values exist outside the actual model itself and allow extra information, usually management information, to be defined for a model of its parts.

The example chosen here will continue the example of process modelling that has been introduced previously in this book by defining a 'process profile'.

### *10.2.2 Background to process modelling example*

Imagine that the process model, as introduced in Chapter 6, is to be introduced into an organisation. This is an excellent example of a potential use of a profile. What we have here is a defined set of knowledge, in this case concerning processes, that it is possible to embed into the UML language itself. The structure of the process model is well defined and can be seen in the following diagram.

The diagram in Figure 10.1 shows the overall structure of the process model, as defined previously. It can be seen that the 'Process model' is made up of five 'Process group' and that each is made up of a number of 'Process'. Each 'Process' is

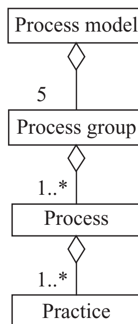


*Figure 10.1 Process model structure*

made up of a number of 'Practice', 'Deliverable' and 'Role'. Each 'Role' is responsible for a number of 'Practice' and each 'Practice' produces or consumes a number of 'Deliverable'.

The information that is modelled here is actually knowledge concerning the nature of the process model and this will be relatively static and tend not to change as time goes on. It is possible to embed this knowledge into the UML and make it part of the language.

### *10.2.3 Identifying and reading stereotypes*

In order to use stereotypes effectively, it is first necessary to be able to spot one within a model. Visually, this is very simple, as stereotypes are indicated by enclosing the name of the stereotype within a set of double chevrons. The name of the stereotype is displayed either inside the symbol (in the case of the stereotype applying to a graphical node) or above the symbol (in the case of the stereotype applying to a graphical path).

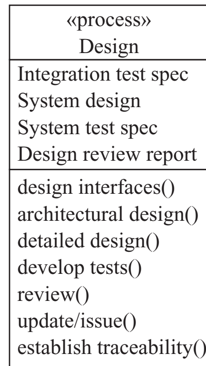


*Figure 10.2 Stereotyped class*

The diagram in Figure 10.2 shows a UML class with a number of attributes and operations. However, in this case the class shown here is 'not' a regular UML class, but it is a stereotyped class. This fact can be established because there is a word 'process' inside the chevrons above the class name. The English phrase that should be used when reading stereotypes is 'that happens to be a'. Therefore, to read the diagram in Figure 10.2, one would say 'there is a class called design, that happens to be a process'. Of course, we are assuming here that the person reading the diagram actually understands what the term 'process' means in the context of this model.

A stereotype can be defined as a tailored version of any type of UML element that exists within the UML meta-model. A stereotype 'must' be based on an existing UML element and it is 'not possible' to define a brand new element from scratch, it must be based on an existing element. Some stereotypes are already defined within the UML, such as «includes», «extends» that are defined as special types of dependency and «component» that is defined as a special type of class.

### *10.2.4 Declaring and defining stereotypes*

Stereotypes must be declared as part of a profile and this is done by creating a package to represent the profile and putting all the profile declaration information within it.

The stereotypes are then defined by tailoring the UML meta-model. Elements from the UML meta-model are taken and then special types among them are declared that represent the stereotype, as seen in Figure 10.3.

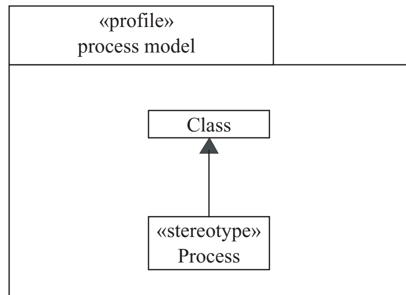


*Figure 10.3 Declaring and defining a stereotype*

The diagram in Figure 10.3 shows a declaration of a profile and the definition of a stereotype. It can be seen from the diagram that there is a package called 'Process model' that happens to be a 'profile'. Note how the way that a profile is declared actually involves using a stereotype called 'profile'. Within this profile, there are two classes, 'Class' and 'Process' that are related by a particular type of specialisation known as an extension. An extension is used specifically when defining stereotypes. An extension is represented graphically by a filled-in triangle – very similar to the generalisation symbol.

The element that is being stereotyped, in this case, is 'Class'. The most important point to realise at this point is that 'Class' is taken 'from the meta-model'. The element 'Class' here represents the abstract notion of a class that is found in the meta-model, or to put it another way, it is the representation of class from the UML language itself.

The new UML element, in this case 'Process', is shown in exactly the same way as a child class on a specialisation but, this time, it happens to be a stereotype, as indicated by the chevrons.

Now, whenever we see a UML class that happens to be a process, as shown in Figure 10.3, we know that it is not a regular UML class, but a special type of class, as defined in a profile – in this case the 'Process model' profile.

Practically, we now have a short-hand way to show that a particular class is actually a process, but it is possible to embed a lot more knowledge within the profile itself and, hence, add far more value to the use of the profile.

### *10.2.5 Adding more knowledge*

It has already been established, in Figure 10.1, that the 'Process' is in itself part of a larger structure and this information can also be embedded into the UML language.

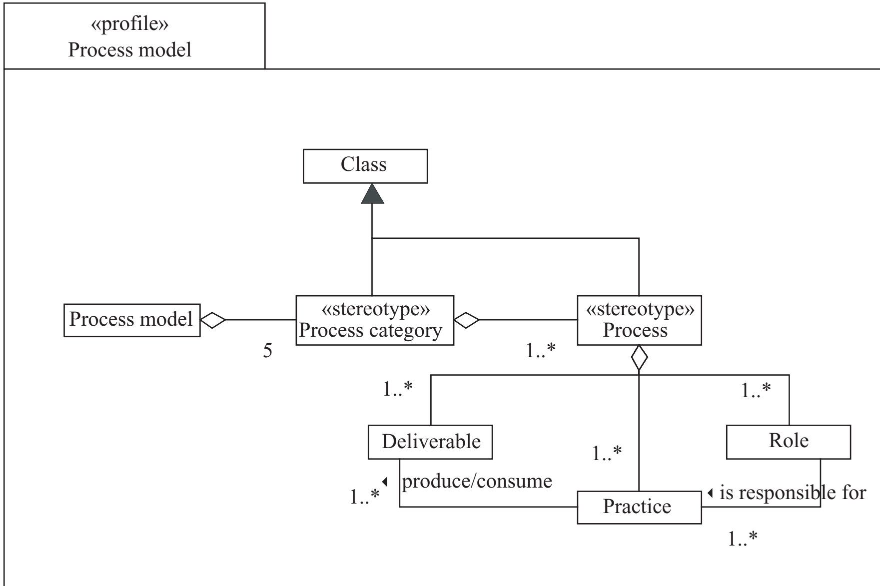


*Figure 10.4 A more detailed profile*

Figure 10.1 can now be used as the basis for the new 'Process model' profile, as shown in Figure 10.4.

The diagram in Figure 10.4 shows that the stereotyped class named 'Process' now has a structure below it. This means that whenever a class that is stereotyped as 'Process' is encountered on a model, it also has that same structure below it. In the same way, it is also possible to specify behaviour for any classes that have operations (using, say, state machine diagrams or activity diagrams) and this behaviour will also be embedded into the stereotype. The more information that is embedded into a stereotype, on the one hand, the more powerful it becomes. On the other hand, however, any such complex stereotype will require more expert interpretation and runs the risk of becoming completely unreadable. This also reinforces the importance of having a well-defined and well-declared profile.

## *10.2.6 Declaring a profile*

When using any profile, it is important that the process is not only well defined, but also that it is openly declared to any readers of the diagram. A profile is declared on a project by treating the project as a package and then relating the stereotyped profile package to it. It is also possible to relate more than one profile to a project or, indeed, profiles may be related to one another.

The diagram in Figure 10.5 shows an example of profiles being declared for a particular project. In the example seen here, the project itself is represented by the package named 'Process model' and it can be seen that this project 'applies'

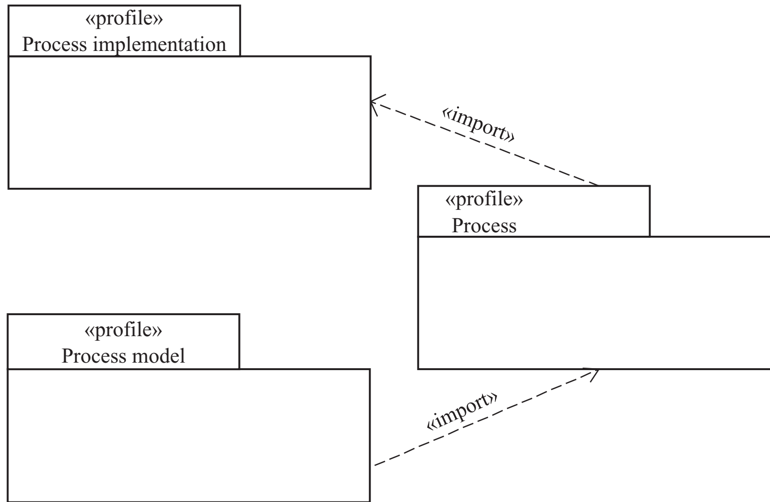


*Figure 10.5 Declaring profiles*

a package called 'Process' that happens to be a 'profile'. This example also takes the concept of profiles one step further by showing that the 'Process' package also applies a package named 'Process implementation' that happens to be a profile. This diagram actually represents not only the process structure to be used on all projects, but also the way that the processes are to be implemented using a process implementation profile. Such an implementation profile may be defined for a specific commercial tool and, taking this further, profiles may be created that apply to a number of tools. By taking this approach of defining a separate profile for each commercial tool, it ensures that the process remains robust even in the event of having to change tools between commercial vendors. Of course, this means that should any project require the use of a different tool for its process implementation, the original process model profile remains unchanged, hence ensuring a robust process model, regardless of the tool that is being used to implement it.

### *10.2.7 Stereotyping symbols*

It is possible to stereotype the actual symbols that represent UML elements to whatever icon is desired. The theory behind this is that some people will find the UML element symbols too difficult to read and that stereotyping icons will make it easier for everyone who reads this model. Although both of these points are true, there is a massive danger when doing this as the assumed knowledge that is required in order to read the diagram correctly increases dramatically. If this stereotyping of symbols is taken to extremes, it is almost impossible to guess even the type of UML diagram that the model is representing, let alone what the diagram actually means.

As an example of this, consider the model in Figure 10.6 and then try to answer the questions posed below.

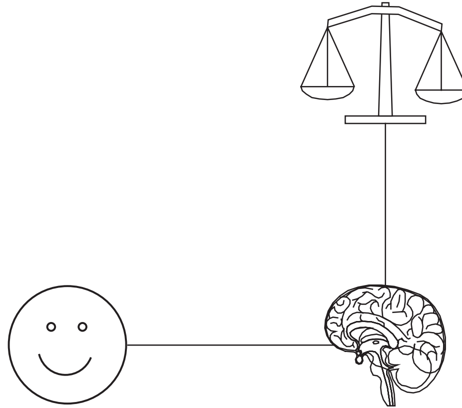

*Figure 10.6 Stereotyped icons*

Figure 10.6 shows an extreme example of stereotyped symbols, but try to answer the following questions:

- What type of UML diagram is it?
- Which UML element does each symbol represent? (Providing that you can answer the first question correctly.)

The actual answer to both questions is that it could be any diagram! Each UML diagram is made up of a number of symbols that are nodes with arcs drawn between them. The model shown here also shows some symbols (although strange) with arcs that join them. Therefore, it is almost impossible to read the diagram without some serious assumptions being made to interpret the new symbols. This may seem like an extreme example, but take a look at some examples shown in UML literature and it will become clear that this model is not as extreme as it first appears.

Having said all this, stereotyping symbols can be very useful and add value to the models; however, much caution must be exercised when using them, even more so than with standard stereotypes.

### *10.2.8 Summary*

In summary, therefore:

- Stereotypes refine existing UML elements from the UML meta-model. It is not possible to define completely new elements.
- Stereotypes may be used to refine any UML element from the meta-model, not just classes. For example, several stereotypes are already defined within the UML language that apply to dependencies, such as «includes» and «extends».
- Whenever stereotypes are used, they must be defined in a profile. This acts like a legend to a map.
- It is important to avoid over use of stereotypes, particularly when it comes to icons, as stereotypes by their very nature assume that the reader has a certain amount of knowledge.

Stereotypes are the first of the three UML extension mechanisms that exist. The next type is 'constraints'.

### *10.2.9 Introduction to constraints*

This section looks at the second of the UML extension mechanisms, that of constraints.

Constraints allow the creation of new rules in the UML that are a semantic restriction represented by a text expression. The constraint itself is shown between two curly brackets and may be represented either informally by simply enclosing text within the curly brackets, or more formally using a specific language, such as a constraint language or a programming language. These may include: OCL (object constraint language), programming languages (such as C++ or Java), pseudocode or, in some case, natural language.

Constraints are declarative in nature and appear on UML diagrams as restrictions on one or more value of a UML element.

Constraints are limited to being added to only a few types of UML elements:

*As an addition to certain UML elements*: Although constraints may be associated with any UML element, the use of the OCL limits their use to a subset of UML elements (defined later).

*Attached to a dependency*: This is often used, for example, to express a binary relationship between two UML entities, such as an 'XOR' function between two classes.

*Attached to a note*: Constraints are often written in notes and associated with UML elements using a dependency relationship.

Constraints are used to express aspects of the UML model that cannot ordinarily be represented using the UML.

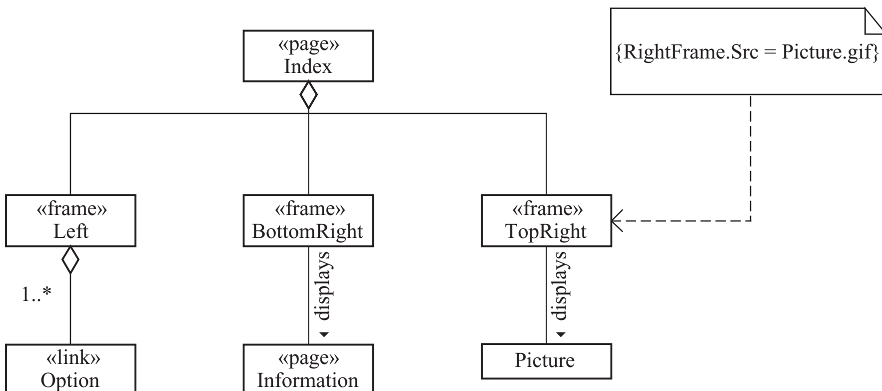


*Figure 10.7 Example of constraint*

Figure 10.7 shows how a constraint has been applied to the class 'TopRight', indicating that the '.Src' value of 'RightFrame' is always set to the value 'Picture.gif'.

This ensures that the bottom right-hand frame is always set to a value that refers to a picture page. The constraint itself is written inside a note and its relationship with a class is indicated with a dependency, as defined previously.

### *10.2.10 Formal constraints – the object constraint language (OCL)*

#### 10.2.10.1 Overview

The structure and contents of a constraint can be relatively informal, providing that it makes sense to the reader and adds value to the model. However, for more formal, precise UML modelling there is a bespoke language that is defined as part of the UML standard, known as the object constraint language, or OCL. The language itself is fairly simple yet powerful and can be used to express aspects of a UML model that are not covered by the basic UML syntax. An in-depth discussion concerning the OCL is outside the scope of this book; however, this section gives a very high-level view of the structure and syntax of the OCL. For a complete definition of the OCL and its use, see Reference 4. This information is presented in the same way as the UML is presented in previous chapters, by using the concept of a meta-model.

The OCL is used to represent constraints that are then attached to particular elements of a UML model. There are three basic types of constraint, as shown in Figure 10.8.

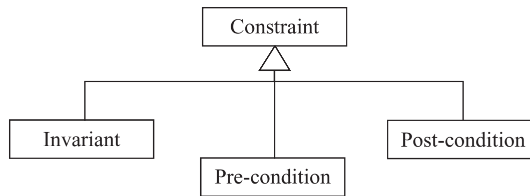


*Figure 10.8 Types of constraint*

Figure 10.8 shows that there are three types of 'Constraint': 'Invariant', 'precondition' and 'postcondition'. These will be described in more detail in due course.

#### 10.2.10.2 Invariants

An 'Invariant' describes something that cannot change, such as an attribute value. This is often useful when describing class hierarchies using the specialisation and generalisation relationships, where the difference between two child classes may be the fact that one inherited attribute is always set to a certain value. In order to illustrate this, consider the example shown in Figure 10.9.

Figure 10.9 shows a possible use for an invariant constraint. In this example, the class 'Mammal' is shown as having three attributes – 'number of legs', 'gender' and 'age' – which will be inherited by its two subclasses 'Dolphin' and 'Cat'. Obviously, there are many differences between a dolphin and a cat, but for the purposes of this

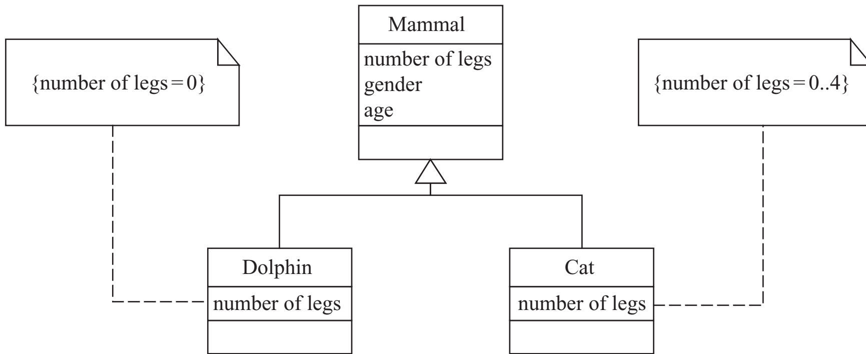


*Figure 10.9 Possible use for a constraint*

example we shall consider how we may differentiate between them based on the information contained in the parent class. Consider the attribute ‘number of legs’, which is applicable to both dolphins and cats. In each case, there is a value assigned to this attribute that (in most cases) does not change. A ‘Cat’ will always have ‘number of legs’ set to the value ‘4’ (unless it has been in some sort of accident, in which case set it to between zero and four) whereas the ‘Dolphin’ will always have its ‘number of legs’ attribute set to the value zero. This concept of a value never varying is the ‘invariant’ in the OCL.

#### 10.2.10.3 preconditions and postconditions

The other two types of constraint are the ‘precondition’ and ‘postcondition’, which are usually associated with operations. In order to understand why preconditions and postconditions occur, one must consider the concept of ‘design by contract’. A ‘contract’ is defined as an unequivocal agreement between two parties in which both parties accept obligation. The two parties, in this case, are the consumer and the supplier, which will be classes in a UML model. The services, or operations, that are available to a consumer are defined by the interface to the class. This is shown visually in Figure 10.10.

Figure 10.10 shows the concepts concerning preconditions and postconditions related to the previous paragraph. It can be seen in the model, therefore, that:

- The ‘Interface’ defines the access available to the ‘Consumer’ of a particular class. This ‘Interface’ is made up of one or more ‘Operation’ that will be made publicly available to the outside world. Each ‘Operation’ is made up of zero or more ‘precondition’ and zero or more ‘postcondition’.
- A ‘precondition’ states what must be made available to the ‘Supplier’ when an operation is invoked. To put this another way, the ‘precondition’ describes the conditions that must be met, including any information that must be made available, in order for the operation to execute correctly.

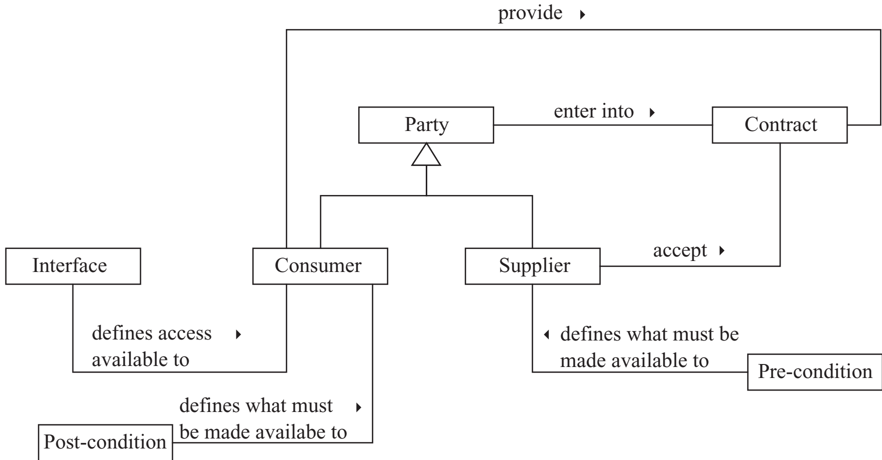


*Figure 10.10* *Concepts concerning preconditions and postconditions*

- A 'postcondition' states what must be made available to the 'Consumer' once an operation has been executed. To put this another way, the 'postcondition' describes what is expected as the outcome of the execution of the operation.

These three types of 'Constraint' are represented using OCL expressions. Each OCL expression has the same structure, illustrated in Figure 10.11.

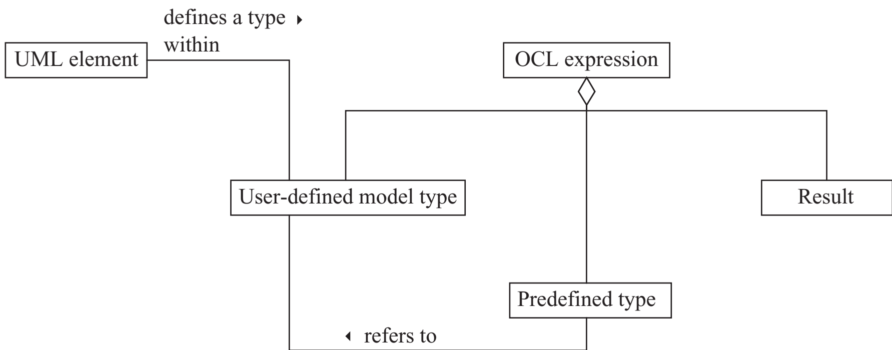


*Figure 10.11* *Structure of an OCL expression*

Figure 10.11 shows that an 'OCL expression' is made up of a 'User-defined model type', a 'Predefined model type' that refers to the 'User-defined model type' and a 'Result'. The 'UML element' defines a type within a 'User-defined model type' and may be further described using the model in Figure 10.12.

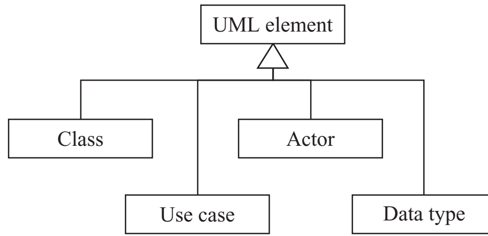


*Figure 10.12 Types of UML element*

Figure 10.12 shows the four basic types of 'UML element' that are used in the OCL: 'Class', 'Use case', Data type' and 'Actor'. It is these UML elements that define the 'User-defined model type'. This relationship effectively allows a constraint to be associated with a UML element by stating that the UML element defines the type required in the OCL expression. Effectively, this means that the constraint can only be associated with the UML element (and any other of their subelements, such as attributes and operations).

The 'Predefined types' from Figure 10.11 may also be defined in more detail, as shown in Figure 10.13.

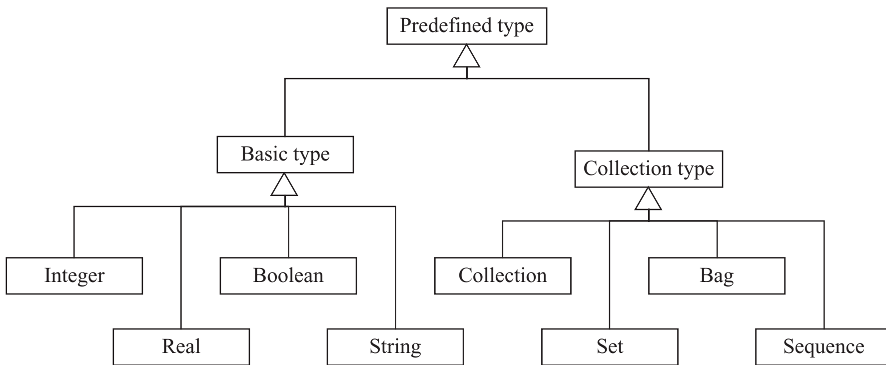


*Figure 10.13 Description of user-defined types*

Figure 10.13 shows that 'Predefined type' has two subclasses: 'Basic type' and 'Collection type'. 'Basic type' has types 'Integer', 'Real', 'Boolean' and 'String', which are the same as their mathematical namesakes. 'Collection type' has four subclasses, which require a little more description:

*Collection*: A 'Collection' is a generic name for 'Set', 'Bag' and 'Sequence', which contains their common operations.

*Set*: A 'Set' is a type of 'Collection' that contains instances of OCL types. Each instance can only be present once in the collection and the instances are not ordered.

*Bag*: A 'Bag' is a type of 'Collection' that contains instances of OCL types. Each instance can be present more than once in the collection and the instances are not ordered.

*Sequence*: A 'Sequence' is a type of 'Collection' that contains instances of OCL types. Each instance can be present more than once in the collection, but this time the instances are ordered.

Each 'Predefined type' has a set of operations associated with it that have a name, an expression and a result type. The model in Figure 10.14 shows the operations associated with the type 'String'.

| String |
| --- |
| |
| concat()<br>size()<br>toLower()<br>toUpper()<br>substring()<br>equals()<br>notEquals() |

*Figure 10.14 Operations of 'String'*

Figure 10.14 shows that the 'String' type has seven operations: 'concat', 'size', 'toLower', 'toUpper', 'substring', 'equals' and 'notEquals'.

Let us now consider some examples of OCL expressions. The constraint expressed visually in Figure 10.9 may be shown in OCL as:

*Cat*
*Number of legs = 4*

Where '*Cat*' is the class name and '*number of legs*' the attribute to be constrained. The attribute is of type integer, therefore it may use any of the operations defined on the predefined type 'Integer'. The equals sign '=' is one such operation and its value ('Value' from Figure 10.11) is assigned as '4'.

The 'Collection' type deals with single-valued or multi-valued attributes. As an example of this, imagine that we have added a new attribute called 'name' and we would like to define its value as a set of three names (all cats have three names):

*Cat*
*name = Set {name}*

Where '*Cat*' is the class name and '*name*' is the attribute to be constrained. The attribute is of type set, therefore it may use any of operations defined on the predefined type 'Set'. The value ('Value' from Figure 10.11) of the set is assigned as all values that can be found for '*name*'.

For a complete and far more detailed description of the OCL, see Warner's definitive book on the subject.

### *10.2.11 Introduction to tagged values*

This section looks at the third UML extension mechanism, that of tagged values. Tagged values allow extra information to be added to any UML element and are shown as tag-value pairs. This means that each tagged value is made up of a tag string, a value string and an equals expression.

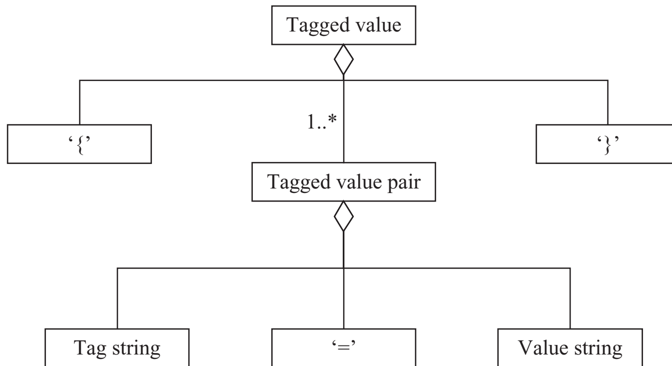


*Figure 10.15 Tagged value syntax*

Figure 10.15 shows that a 'tagged value' is made up of a '{', one or more 'Tagged value pair' and a '}'. Each 'Tagged value pair' is made up of a 'Tag string', a '=' and a 'Value string'.

Tagged values are typically used to show extra information, such as project management information or configuration management information. Tagged values are not part of UML semantics, but are information about the model itself.

*Figure 10.16 Example tagged value*

It can be seen from the model in Figure 10.16 that extra information has been added to two classes, in the form of a tagged value that contains two tagged value pairs. The tag strings are 'Author' and 'Date' and the value strings that define the value for each.

Sometimes it is desirable to define a set of tagged values that may be added to a particular element. In such a case, they should be defined on an assumption model by defining types of 'tag string' as defined in Figure 10.15

## 10.3 Conclusions

This chapter introduces three basic extension mechanisms that exist in the UML:

- *Stereotypes*: These are used to define new UML elements based on existing ones. Specialist knowledge is required to understand a model that uses stereotypes, which must be represented on an assumption model.
- *Constraints*: There are three types of constraints: invariants that restrict the value of a UML element, preconditions and postconditions, that define what happens before and after the execution of an operation.
- *Tagged values*: The use of tagged values allows extra information such as project management and configuration management details to be added to UML elements.

All these extension mechanisms should be used with caution, as it is very easy to make a diagram more confusing than it might otherwise be. Remember, if you are unsure about whether to use stereotypes or not, ask yourself whether it adds value to the model or in fact detracts from it.

## 10.4 Further discussion

1. Revisit the new process that was modelled as part of the process modelling exercise in Chapter 6. Consider the high-level structure model and apply stereotypes to it. Where can stereotypes be used here? How will this affect other models?
2. Define the assumption model for your stereotypes from question 1.
3. Find out more information about the OCL and model some of its predefined types and their associated operations, as shown in Figure 10.14.
4. Why would you prefer to use OCL over informal constraint modelling, or vice versa?
5. Create a set of tagged values that may be applicable to UML components from any component diagram from Chapter 5. Are there any other UML elements to which your tagged values may be applied?

## 10.5 References

1 RUMBAUGH, J., JACOBSON, I., and BOOCH, G.: 'The unified modelling language reference manual' (Addison Wesley, 1998)
2 BOOCH, G., RUMBAUGH, J., and JACOBSON, I.: 'The unified modelling language user guide' (Addison Wesley, 1998)
3 JACOBSON, I., BOOCH, G., and RUMBAUGH, J.: 'The unified software development process' (Addison Wesley, 1999)
4 WARMER, J., and KLEPPE, A.: 'The object constraint language, precise modelling with the UML' (Addison-Wesley, 1999)

# *Chapter 11*

# Tools

hi, I'm Plenty
*Plenty O'Toole*

## 11.1 Introduction

This chapter discusses tools that may be used for UML modelling and, in some cases, for general systems engineering tasks. The main aim of this chapter is to provide information and guidelines that describe how to make an informed decision when choosing a tool.

At the end of the day, if you are carrying out any form of UML modelling, you will need to consider obtaining some sort of tool for assistance. Tools range from the very basic, such as a simple PAPS (paper and pen system) tool, to a full-blown, interactive 'environment', but which sort of tool will be right for you? Tools can be expensive, very expensive! It is therefore absolutely essential to know your requirements when choosing a tool and to make sure that your requirements drive the choice of tool, rather than the other way around.

No specific tools will be mentioned explicitly in this chapter as the tool market fluctuates very quickly. The functionality of tools changes enormously from one version to another, or from one configuration to another. The marketplace itself is still quite unstable with larger companies swallowing up smaller ones in order to take over their market share or, in a slightly more sinister way, to remove a competitor from the market altogether.

Some companies are quick to dispel their competition with swift statements, while others produce reports and comparison tables between their tools and their competitors'. It is worth remembering where tool vendors actually make their money – is it from the sale of the tools or from the maintenance costs that go along with it? Is it in their interest to provide a tool that is simple and easy to learn when they provide training courses and consultancy to help the way you work with the tool? This may seem like a particularly cynical view, but when spending such potentially large sums of money a healthy dose of cynicism can save a significant amount.

This chapter is split into three main sections:

- The first section gives a brief overview of CASE (computer aided/assisted software engineering) tools and discusses what a typical tool can do. Perhaps even more important than this is to realise what a tool cannot do and to have a good picture of what its limitations are.
- The lion's share of this chapter is devoted to providing a number of requirements to bear in mind when choosing a tool. The guidelines in this section are based on many years' personal experience dealing with both tools and their vendors and also seeing the result of what making the wrong decision can have on an organisation or project.
- Finally, this chapter winds up with some conclusions concerning tools.

All of these points are designed to be generic so that they may be applied to the selection of any CASE tool whatsoever, rather than trying to compare specific tools on the market. Clearly, the latter approach would render this chapter obsolete even before the book was published, bearing in mind the speed at which tools change and the rapid growth in the number of CASE tools on the market.

## 11.2 CASE tools

The tools that are discussed in this chapter may be split into two broad categories – CASE tools and systems engineering tools – although generally speaking the guidelines in this section are equally applicable to both.

The acronym CASE stands for either 'Computer Aided Software Engineering tools' or 'Computer Assisted Software Engineering tools'. Although both definitions are correct, the true definition is lost in the mists of time and nobody is really sure which of the two definitions is the true one. CASE tools are generally associated with a particular modelling language, method or methodology. Obviously, the tools this book is aimed at are the UML-based CASE tools, although the guidelines are applicable to any type of CASE tool.

Systems engineering tools are often similar to CASE tools, but they do not generally adhere to a particular approach. Their functionality tends to be aimed at a higher level – dealing with process-oriented functions rather than modelling *per se*.

Some of today's tools actually do both, or may be integrated into a 'suite' of tools that are purported to make life far easier. Beware of words such as 'seamless', 'transparent' and 'single-stop solution' when considering any tool as they are concepts that are much vaunted yet rarely exist in real life.

## 11.3 Selecting a tool

### *11.3.1 What tools can do*

Before entering the world of CASE tools, it is very important to take a reality check and to sum up what these tools are capable of. Generally, CASE tools can help

a project in three ways:

- The main use of CASE tools is to help the people who want to model, to use the language in a consistent and constructive way. The tool should aid a newcomer, yet at the same time not hinder an experienced user. Most CASE tools will make modelling easier in that: models will be quick and efficient to produce, models will look good (hence aiding communication) and they will help to store the diagrams in the model in a simple and efficient manner. As a basic rule of thumb, imagine creating UML diagrams using a simple drawing package and then make comparisons based on its usage compared to a CASE tool. This is the benchmark that is used throughout this section.
- The second way that CASE tools can make life easier is, like all computer applications, to take the drudgery out of a particular task, in this case by helping with checking and verification. This becomes very important with regard to syntax and semantics and can, theoretically, be used as part of a quality control process as a first step towards assessing models.
- The third major use of CASE tools is to help with documentation. In an ideal world, documentation should be fully automated and a simple keystroke should automatically produce any report that is required. The reality of this is somewhat different, but this is discussed further in due course.

CASE tools do have many other functions but these may be grouped into the 'bells and whistles' category from a systems engineering viewpoint. Code generation is covered in this section but remains simply one of main capabilities that may be required by an engineer, rather than being the main aim of the tool, as in a software engineering context.

Systems engineering tools generally work at a higher level, being more concerned with process activities than actual modelling. The typical functionality of a systems engineering tool includes:

- Traceability throughout life cycle phases. Being able to trace individual requirements forward through the project life cycle is very important. The opposite is also essential, for example, starting at one point in the design and trying to trace back to its originating requirement. This is the main focus of many systems engineering tools.
- Information management, where information pertaining to a project (that should be based on a process) can be stored in a central repository across all phases of the systems life cycle. This includes managing the implementation model if the approach suggested in Chapter 6 is followed.
- Process definition and help. This is potentially the most useful aspect yet also potentially the most dangerous aspect of tool functionality. It is stated (over and over) in this book that the process must drive the tool and not the other way around. By letting a tool effectively organise your process, it is relatively easy to fall into this trap.

This chapter focuses on CASE tool functionality with regard to the UML, rather than general systems engineering functionality. It should not be too difficult, however, to

extract a few rules from the ones presented here that would be applicable to almost any tool.

### *11.3.2 What tools cannot do*

The previous section introduced some high-level features of what CASE tools can do, and now this section looks at the opposite of this, which is, what CASE tools cannot do. The nonfunctionality of CASE tools may be summarised as follows:

*Validation*: Despite what people may say, there is no inherent mechanism for current tools to validate designs. Designs may be verified using the mechanisms mentioned previously, but there is no way to validate whether these designs actually meet their original requirements or not.

*Ensure correct models*: There is no way that a model may be automatically checked to ensure that it makes sense. Taking an example that has been used previously in this book, there is nothing to stop someone creating a model that reads 'cat' chases 'dog', which in our stereotypical world cannot happen. A tool can check the syntax of the model, but cannot guarantee that the overall model is correct.

*Another function that CASE tools cannot perform is to implement a generic process*: Many tools implement a process, but this will probably not be the process that a given organisation would want to implement. This can be very dangerous, as adopting a proprietary process can lock an organisation into a single vendor's product, which can turn out to be very expensive. This is a situation that has been encountered all too often in the past. The problem starts when an organisation buys a proprietary tool that implements a proprietary process. The organisation is then convinced that this process is an appropriate approach for their particular set up, without any sort of analysis or assessment by the organisation. Quite often, this buy-in will occur when the organisation is shown a ream of case studies and an example of how the process works that has been implemented using the tool. The next step is that the organisation, when it comes down to implementing the process on a real project, does not have the practical knowledge needed to use the process properly, and thus calls in consultants from the company that produces the tool. The next step is that training courses are required in order to achieve the full benefits of the tools and the process. A few months later, the price of all consultancy and training goes up significantly, but the organisation is now locked into using a particular tool and approach and has invested too much time and money to pull out at this stage. Another eventuality is that the tool vendor then goes out of business or is bought up by a larger company, effectively reducing the resources required for the project.

This section may seem rather negative, but it is important to approach assessing any tool with both eyes wide open.

## 11.4 A requirements model for CASE tool evaluation

### *11.4.1 Requirements model overview*

This section introduces a simple requirements model for tool selection. A number of different high-level requirements have been created, each of which has several lower-level requirements that are discussed in more detail in their relevant section of text.

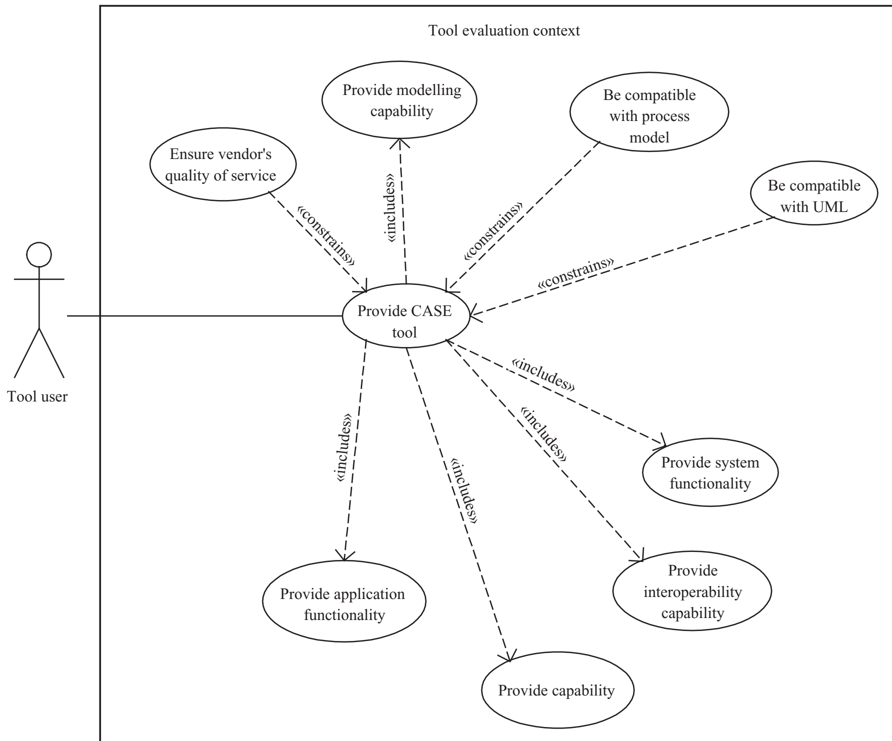


*Figure 11.1 High-level view of the requirements for CASE tool evaluation*

The diagram in Figure 11.1 shows that there is a single main requirement to ‘Provide CASE tool’ that includes five high-level requirements, that are:

- ‘Provide modelling capability’ that describes the drawing, management and usability of a tool.
- ‘Provide system functionality’ that describes the actual system requirements in terms of any hardware and software requirements along with networking capabilities.
- ‘Provide interoperability capability’ that describes how a tool may be required to operate with other tools, technologies or clients.

- 'Provide capability' that describes the sort of tool capabilities that may be used by a systems engineer, including: report generation, extension functions, and so on.
- 'Provide application functionality' that describes how the use that the tool may be put to will generate requirements for the tool.

The main requirement of 'Provide CASE tool' also has several constraints that will limit how the choice of tool can be made, as follows:

- 'Ensure vendor's quality of service' that describes how to ensure that the vendors can meet all of your requirements.
- 'Be compatible with process model' that describes how the tool may be required to fit in with a particular approach to working, which may put additional requirements on the tool.
- 'Be compatible with UML' that describes how to ascertain what your UML-related compliance requirements are and how this may form an input to tool selection.

Each of these areas is covered in more detail later in this chapter.

Different tools offer different capabilities, but there is a core set of functionality that is offered by many of the tools. Many specialist capabilities will not be discussed here, such as specific real-time extensions and language-specific constructs. Remember that the model presented here is a set of requirements for tool selection, not the actual process that must be followed. Of course, it is possible to generate a set of processes meeting these requirements that can form the basis for a consistent tool evaluation service.

### *11.4.2 Modelling capability provision*

Perhaps the main reason to obtain a CASE tool is to actually help with the modelling of a system. This is far more complex than it first seems and there are several considerations to be borne in mind when considering this requirement, as shown in Figure 11.2.

Perhaps the main reason to buy a CASE tool is to help with editing the actual diagrams themselves, as indicated in Figure 11.2 (by the 'Edit models' requirement). While it may be practical to create initial models using a paper and pencil or a white board with sticky notes, there comes a point in every project where it becomes too impractical to continue using a hard medium, due to the constant changes that must be made to each model, particularly in the early stages of a project. These constant changes are due, in part, to the iterative nature of UML development.

It may also be tempting to use a standard drawing package to create UML diagrams – after all, one of the requirements that was used when creating the UML itself was to make it simple to implement on an automated tool. Any drawing package, no matter how basic, is capable of creating any UML diagram, which must be seriously borne in mind when looking at some of the more expensive CASE tools that are around today. One of the problems that existed with the older modelling techniques was that the symbols chosen were a little esoteric at times. Anyone who has ever tried to draw a Booch class, represented by a 'cloud' shape, will know the

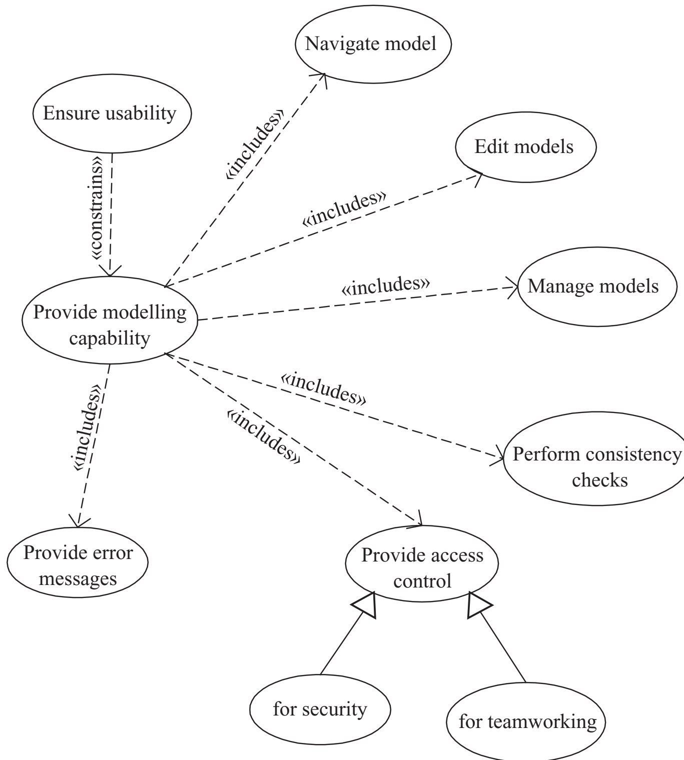


*Figure 11.2 Requirements for provision of modelling capability*

problem. Anyone who has ever tried to draw a 'dashed' cloud either by hand or with a tool will also know frustration!

Some sophisticated drawing packages allow the creation of special templates that allow specific shapes to be created automatically, which means that it is possible to completely configure a CASE tool out of a simple drawing package! This may seem totally impractical, but almost all of the tools around at the moment do not meet the UML 100 per cent when it comes to drawing the diagrams! It is important, therefore, to ensure that you are getting more out of the CASE tool than just a bespoke drawing package, especially bearing in mind the difference in cost between a simple drawing package and a full-blown CASE tool.

Following from the basic editing requirement, comes the issue of how the model is stored and how all aspects of the model are managed. Indeed, the 'Manage models' requirement may be the essential factor in choosing between two tools. It should be borne in mind that different tools use different formats for storing the core data. Is the tool using a data base, or a simple ASCII text file? Is the format a standard format, such as HTML, or a bespoke format that can only be read by other copies of the same tool? Can the source files be emailed, or do they rely on a particular instance of

license in order to be read? These may seem like trivial questions, but are well worth investigating since what use is a tool, if you are working in a distributor team and that tool will not allow models to be emailed? This relates directly to another requirement, that of 'Provide access control'. If you are working within a team, then can different members of the same team access the same information at the same time? If so, it needs to be determined how the tool gives protection against two people changing the same data simultaneously. Most tools allow the model to be partitioned in some way in order to allow distributed working. This may then throw up security issues as some information in the model may be classified and unavailable to all members of the team.

Another aspect of using a CASE tool that is worth considering is how the tool navigates between the different diagrams, indicated in Figure 11.2 by the 'Navigate model' requirement. Some tools may allow any diagram to be created outright, while others may insist that one diagram exists before another is created. As an example of this, consider the strong relationships between the nine diagrams that have been discussed so far in this book. State machine diagrams, it has been stated, should be associated with classes or use cases; therefore, some tools will only allow a statechart to be created if it has a class or use case associated with it. Another example is where a tool has adopted a particular process that insists on the diagrams being created in a strict, predefined order such as: use cases first, interaction diagrams (collaboration and/or sequence diagrams) and then class diagrams. Another tool may insist, for example, that class diagrams should be the first diagrams to be created and all other diagrams should relate back to this.

Another requirement is to be able to 'Perform consistency checks'. Every CASE tool on the market will offer some form of verification that can be automatically performed on the UML models and this, in many cases, will add the most value to the modelling process. Invariably, this turns out to be some sort of syntax checking that can be carried out on diagrams, and consistency checks between diagrams. These checking facilities may be automatic, where the syntax is checked as the model is entered, or they may have to be invoked explicitly. The actual level of checking varies enormously and the following list gives some indication of the range of the verification facilities of tools, based on the relationships between all types of UML diagrams:

- One of the simplest checks is to ensure that there are no name clashes in a model. A simple example of this may be to check that no two classes have the same name. This is a feature that is often automatic and, in many cases, the tool may actually forbid a name clash, rather than just providing a warning.
- Another popular check is to ensure that send and receive events are consistent in some way. There are many ways to ensure that events are consistent, ranging from simply checking that each existing send event has a related receive event on the model, to ensuring that arguments passed between objects match up.
- Some tools ensure that any activities used on a statechart are chosen from a list of operations from their associated class diagram. In fact, some tools actually add any activities that do not exist as operations to their associated class diagram.

As can be seen from this list, the functionality of checking facilities varies, but there is a more subtle danger with using a tool's checking facilities and that is how the

checks are actually performed. This may seem trivial, but if the checking facilities are being used as part of a quality system, or as part of a defined process that will, in some way, demonstrate the correctness of a model, it is absolutely essential that the underlying rules of each check are explicitly defined. This is a problem that rears its ugly head in two of the other aspects of the assessment criteria laid out in this chapter.

The final point to bear in mind with verification is what actually happens to the results. Are they displayed on the screen or are they saved in a file somewhere? Will they still exist once the display window has been closed down or will they be deleted, never to be seen again? It may even be desirable to associate them with a particular version of a model (which relates to one of the other points here: version control).

The following list summarises the basic assessment criteria for model verification facilities in a tool:

- What verification facilities are offered by the tool?
- Are these verification facilities defined in some way, or are they nondeterministic?
- How are the results displayed and stored?

If the tool cannot offer adequate checking facilities, what exactly is it offering over and above a standard drawing package?

One requirement that will relate to all aspects of using the tool is ‘Provide error messages’ because, as when using any tool, there will be a learning curve where mistakes are made or, as is often the case with any software package, things go inexplicably wrong. Everyone must have, at some point in their lives, come across an error message that goes along the lines of ‘error X07708’ which is of no use to man or beast for any practical purpose. Error messages need to be intelligible to anyone using the tool and it should also be immediately apparent whether the error message relates to the tool itself or the underlying operating system. It is very frustrating to come across an error message and contact the vendor, only to be told that the error has nothing to do with their tool!

The final requirement, which is also the most difficult to quantify in any way, is that of ‘Ensure usability’. There are obvious criteria here, such as the look and feel, compared to, for example, a standard Windows environment, but other aspects are not so easy to evaluate. It is always worth reminding oneself that a tool is supposed to make life easier and quicker that doing the task by hand, so if you find that it takes an hour to enter a single diagram, then maybe that tells you something about the usability of the tool. Of course, there will be a learning curve involved, but all tools should be intuitive to use and, after all, the basic functionality should be similar to that of a drawing package which, it should be remembered, any primary-school child can successfully operate. If a tool requires extensive training and consultancy to use, it should be questioned whether it is usable or not.

### *11.4.3 Provision of interoperability capability*

No project is an island and, as such, projects often work alongside other projects or a disparate range of clients. Also, the project may be constrained by the other tools that are already being used, so any tool may have to be compatible with other,

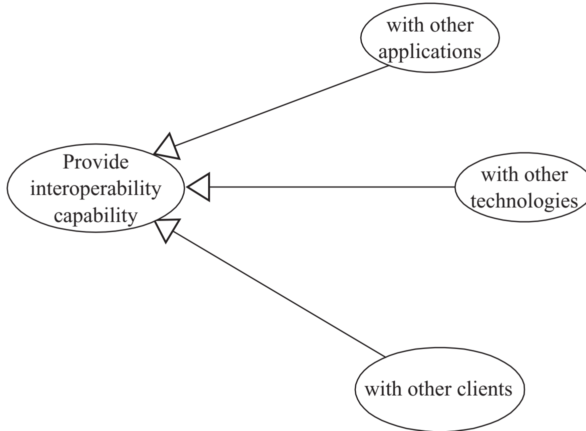


*Figure 11.3 Requirements for provision of interoperability capability*

nonengineering related packages. This section addresses these issues and starts by considering the requirements in Figure 11.3.

In many real-life scenarios, the CASE tool that is chosen will probably have to integrate with an established working environment or, to put it another way, to 'Provide interoperability capability with other applications'. This will entail the CASE tool output being used by other tools that may be used, for example, for aspects of project management. Some CASE tools vendors, sensing this, have geared their product towards becoming a single component of a complete solution that means that the actual UML CASE tool is part of an overall suite of tools that will perform every single task that your organisation could possibly want, which is good.

Some tools will offer extra functionality bundled into the CASE tool itself. This is all well and good, but will it fit into the existing working environment, or will it actually force people to work in a way that the tool wants people to? Extreme caution must be exercised here, as it is another example of the tool driving the process, rather than the other way around.

Some of the functions that may be worth thinking about are listed below. Bear in mind, however, that some of these may be included in the CASE tool or may be available as part of the same suite, or may even (the most likely case) be tools that are already being used within the organisation.

*Project management tools*: The CASE tool may interact with such tools in order to make a link between life cycle phases and aspects of project management.

*Configuration management and change control*: This is essential for any type of real project where the evolution of models must be recorded. This is achieved, in reality, by baselining and version control.

*Requirements management tools*: With requirements driving the whole project and defining the basic quality of a project, it is important that they are well defined and well managed.

*Life cycle support tools*: Some tools help with the development process and thus the life cycle of a project. Again, this may tie both project management and engineering activities together, but the tool must be flexible enough to allow any process to be adopted, rather than simply some proprietary process that is put forward by the tool.

*Animation tools*: Some tools will animate different aspects of models. This may range from highlighting states that are active when a system is being executed, to executing 'what if?' situations in the form of scenarios.

*Office suite packages*: Some packages are now 'fully integrated' into a complete office suite so that there is no need to ever look at another tool, as this one will do absolutely everything that anyone could possibly want. Silver bullets, indeed!

Integration with your existing environment may well be the crucial factor when buying a tool. It is important, therefore, to take stock of the current working environment before choosing a tool.

The working environment may also include working 'with other technologies'. This may include requiring an XML or XMI language output capability. A particular project may use different CASE tools from different vendors, hence an XMI output may be a prerequisite. Modern projects are increasingly using technologies like HTML to provide a common, and easily accessible output medium for all project deliverables, so this may be an issue.

One final requirement to bear in mind is that it may be necessary to work 'with other clients' who may be using a particular set of tools or technologies, which may have a massive impact on the final choice of tool.

### *11.4.4 Provision of other capabilities*

Many tools offer capabilities that are over and above those of a simple drawing package. This section looks at some of the more common capabilities that are offered and is not intended to be an exhaustive list.

The diagram in Figure 11.4 shows the sort of requirements that should be considered when looking into the provision of other tool capabilities. Two of the requirements here relate directly to software engineering and provide capability 'for code generation' and for 'reverse engineering'. It must be remembered that the UML's background is primarily derived from software-based methods and methodologies, rather than generic systems modelling; therefore, almost every CASE tool will come with some type of code generation function.

Most major languages are supported by CASE tools and some offer support for more than one language. The number of languages that are supported, however, will often have quite an impact on the price paid for the tool. For example, some tools have different editions, depending on what you want them to do. They have names such as 'basic', 'modeller', 'professional', 'designer' and 'enterprise'. Some of these editions will not allow any code generation, whereas others will allow a single language, or full support for all languages that are available.

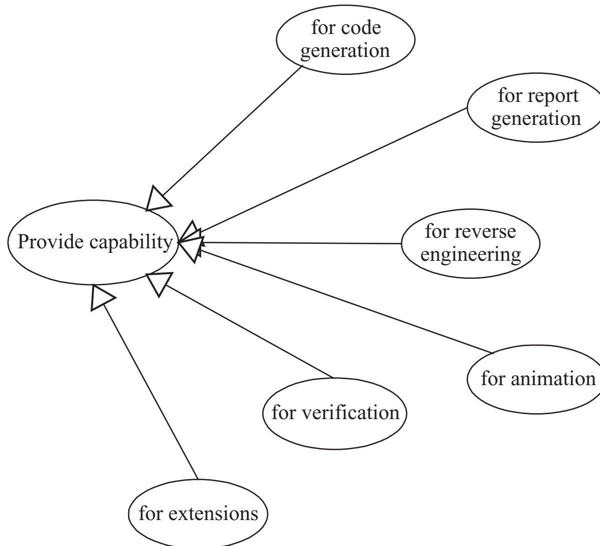


*Figure 11.4 Requirements for provision of tool capability*

It is also worth checking how the tool actually integrates with the target language's development environment – or, indeed, whether it does at all. Some tools interact with a fourth-generation programming language by controlling it in some way, whereas other tools simply create text files that need to be explicitly imported into a development environment.

Perhaps the most fundamental aspect of code generation to be looked at is whether it actually saves any time and effort. This may seem like a strange statement to make, but many people use code generation once and never touch it again. It is worth asking what code is actually created. In most cases, this will almost certainly contain definitions of, for example, classes, attributes and operations, along with stubs for each operation. Although useful, is it worth the extra money in licence costs to have the ability to create code stubs? Which diagrams are actually used for the code generation? The main contenders for this are those of the static model, as, by their very nature, they represent a snapshot of the software that is relatively simple to implement. Now think of the behavioural model and of what aspects of this model are used, if any. Also, remember that any information that is generated in the code must exist somewhere in the model. Imagine, for example, that the code that is generated contains information to define attribute types, default value, visibility, and so on, and this information must be entered (typed) into the model. At the end of the day, how much time does this save, if the level of detail of the model is so low that the model is almost a code itself?

There is a more dangerous aspect to code generation, which is the same point that was raised in the previous section concerning verification. How exactly is the code generated? What are the rules followed when going from a class diagram to a piece

of code? If this process of transforming a design into code is not fully defined, how far can it be trusted? History and fiction have taught us that we simply cannot trust any system that is not fully deterministic. In reality, this is impossible on all but the most trivial of systems, but that does not mean that we can be complacent and allow an unknown, undefined process to create (potentially) safety-critical applications. Remember the disaster that occurred in the Hollywood movie *Westworld*, where human-like android slaves turned into psychotic killing machines simply because they had the capability to change their own programming, which put them out of direct control of their human creators.

Code generation can be very useful and very productive, but, if not assessed and understood properly, it can also be a complete waste of time, money and disk space.

The following criteria should be borne in mind when assessing code generation facilities in a tool:

- Do you actually want code generation?
- What are the extra costs for code generation functions compared, for example, to the basic modeller-only edition of the tool?
- Which languages are supported by the tool and which development environments? In addition, which manufacturers and exact version numbers are supported?
- How much time and effort is actually being saved by using the code generation functions? Put it to the test before spending any money.
- What code exactly is generated? Is it just code stubs? Which diagrams are used to generate the code? Some tools claim 100 per cent code generation, which is quite a bold claim, but what does this mean exactly? In some cases, this means that 100 per cent of the model is used to generate the code rather than being able to generate 100 per cent of the actual final code.
- How, exactly, (by what process) is the code generated? Are the rules defined anywhere or are they nondeterministic?

These guidelines are applicable to forward engineering only and not reverse engineering, which is covered in the remainder of this section.

Reverse engineering is, unsurprisingly, the opposite of forward engineering. If forward engineering is going from designs to code, then reverse engineering is taking existing code and generating designs from it. This may be a particularly attractive function, especially where a great deal of legacy code exists that is difficult to make sense of.

Clearly, the output from such a function will be the models themselves, so it is important to stop and think for a moment about what types of model are created. First, take this literally and ask which of the nine diagrams are created from the code. Perhaps the most obvious diagram is the class diagram, but what about the others? It is only possible to reverse engineer between the implementation (code) phase and the low-level design phase, then how much practical use will the tool be?

Much of the focus of this book is how to communicate effectively by creating simple, clear and easy to understand models. Much of the art and skill of UML modelling is getting these models at an appropriate level of abstraction and ensuring that they have a clear connection to reality. The connection to reality should not really

be a problem with regard to the reverse engineering tool, as it is taking the reality and creating diagrams from it. But what about the level of abstraction of the models that are created? What about similar views of the same system? How can an automated tool possibly define, for example, sensible scenarios based on the code? The simple answer is, of course, that a tool cannot! It is well worth having a look at the quality of diagrams that are the output from the tool, as in some cases they contain very large classes that are full of attributes and operations at a very low level of abstraction, which, for all intents and purposes, are useless.

An obvious aspect of reverse engineering to look at is which languages, environments and versions of these are supported by the tool? Choosing an incorrect version number or the right language from the wrong manufacturer can be very costly.

The final point is a recurring one, which is, how are the models generated and what rules are followed to generate the models? Again, it is a question of having a deterministic system or a nondeterministic one.

The assessment criteria for reverse engineering may, therefore, be summed up as follows:

- Which of the 13 UML diagrams are generated from the code?
- How are the diagrams generated from the code – are the rules defined?
- Which languages, environments and their associated versions are supported?
- How useful is this function?

Above all, remember that reverse engineering will not be a feature that is wanted or needed by everyone, and is it worth assessing exactly how much it would be used before taking it as a basis for tool selection.

One of the potentially most useful functions offered by CASE tools is report generation, indicated by ‘for report generation’ in the diagram. Report generation is particularly attractive when demonstrations are shown where a complete phase is deliverable, that is produced at the touch of a single key.

The first criterion is to look at the output of the report generation and ask whether the tool produces its own files or whether it produces files that are compatible with another product. Many tools will offer both facilities: a simple text-only output that may be generated quickly and simply from the data dictionary of the tool, or a complex file that may be used by a proprietary word-processing package. In the case of a simple text output, any text editor will suffice, ranging from the most sophisticated office product to the simplest of line editors. However, what most people will want to see is a completely formatted, beautifully laid-out document that can be printed out directly from a favourite word-processing tool. This is a great idea, but the format of the document will be whatever the tool vendor thinks the format should be. This may be avoided by some type of editing function that will allow templates to be devised to create bespoke documents. The amount of effort involved in this may vary, however, from a simple ‘tick the box’ approach, to having to have intimate knowledge of a particular programming language or script language that may be required to create such templates. This is particularly important when it comes to following, and thus complying with, a process. It would be very useful and efficient to have document

templates defined that are compatible with a defined process, which would be very beneficial indeed to any project.

Another related note that is worth considering is which word-processing packages is the tool compatible with and is another licence required? This is one of the ways that extra costs can be sneaked on to the bill without anyone realising it. This is not much of a problem with PC-type systems, where most word processors seem to be converging in terms of format and compatibility, or that are relatively inexpensive anyway. However, consider, for example, Unix-based systems where the software (for dark reasons known only to software companies) seems to cost orders of magnitude more than their PC counterparts.

If the CASE tool is interacting with another package, it is worth considering how efficient the interaction between the two is. In some cases, there is no interaction whatsoever, as a file is created which must then be opened from the appropriate word-processing package. Another option is that the word-processing package is actually controlled directly from the CASE tool. This may have serious effects on the efficiency of the system. In one example, a particular CASE tool took over ten hours to generate a report from a model consisting of less than 50 classes! This turned out to be because the tool was, quite literally, cutting and pasting each diagram to a clipboard and, from there, copying it into the actual word-processing package.

The final point to bear in mind is the old chestnut of how the reports are generated. One particularly ambitious CASE tool was able to not only put the diagrams from the tool into the word-processing package, but also to generate English sentences from the model elements that could be used to explain the model to readers of the report. Unfortunately for the CASE tool company, the aggregation and specialisation interpretations had been confused. Therefore, whenever the model said aggregation, the accompanying text read 'has types' and the specialisation relationship read 'is made up of'. Of course, this could lead to considerable confusion especially if this error is carried over into the code generation or reverse engineering functions of the tool.

The assessment criteria for report generation may, therefore, be summarised in the following list:

- Does the tool use text-only outputs or is the output compatible with a proprietary word-processing tool?
- Are extra licences required to see the output of the report generation and, if so, how much extra is this?
- How easy or difficult is it to create bespoke templates for special report formatting?
- How efficient is the report generation in terms of time?
- What are the rules for report generation and how are they defined?

Report generation can be a very quick and efficient solution to many problems and may benefit a project enormously. However, as with all aspects of CASE tools, caution must be exercised when assessing this aspect of the tool.

Another capability that is often offered is 'for animation' where the behaviour of a model may be animated or simulated using, quite often very fancy, animation techniques. The same sort of questions must be asked here in terms of whether or not the animator is part of the standard tool package or part of another commercial

product since there could be serious financial considerations. From a UML point of view, the animation can only be applied to behavioural diagrams, so it is worth looking at what behavioural diagrams are supported by the tool and then comparing this set of diagrams to the ones supported by the animation tool. Common sense would dictate that the animation tool would animate all of the behavioural diagrams but this is often not the case. For example, some tools will only support the animation of state machine diagrams whereas others concentrate on scenarios. Animation techniques can be very powerful and very useful 'for verification' but, when used in this context, we once again have the issue of how the animation is being carried out and whether or not its level of determinism is good enough for forming part of a formal verification procedure.

The final requirement dealt with in this section covers providing capability 'for extensions'. This is a difficult area to classify as most tools allow a certain degree of extension and custom capabilities. Things to look out for include: writing bespoke consistency checking rules, tailoring the use of UML syntax, defining traceability, defining templates, etc.

### *11.4.5 Provision of application capability*

The UML is a general purpose modelling language, therefore it may be applied to almost any type of application. Although there are no solid rules with regard to which diagrams to use for a particular application, some diagrams do lend themselves more to certain applications than others. It is important, therefore, to try to ascertain why the tool is being bought and what use it will be put to. Figure 11.5 shows some of the areas forming the requirements for this section.

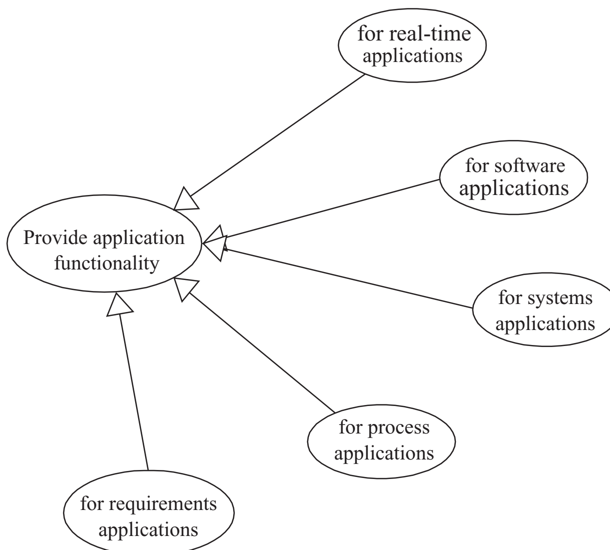


*Figure 11.5 Requirements for provision of application functionality*

There are five main requirements that represent application areas shown here. This is simply meant to give an indication of the sort of applications to consider and is not intended to cover every eventuality. The diagram covers providing application functionality:

*For real-time applications*: Clearly, some diagrams lend themselves more to real-time applications than others. For a start, timing attributes will apply directly to the behavioural aspect of the system and, with the new capability offered by UML 2.0, there are now mechanisms to apply timing constraints to (unsurprisingly) timing diagrams and sequence diagrams. Although the UML now supports the assignment of timing information to a model, it does not show how this timing information is to be used, as there is no inherent process in the UML. Many tools offer real-time support and it is, therefore, important to look at which methodology, if any, is supported by the tool. Some tools will support academically and industrially proven approaches, whereas other tools make up their own approaches.

*For software applications*: This is where most tools should excel and will depend on the process being followed by a particular organisation. For example, the RUP is related directly to the UML and it is, therefore, a simple task to identify which of the 13 diagrams are used (all of them!). Some other processes may require a subset of the diagrams, depending on the sort of information that is required to be delivered.

*For systems applications*: Again, what is required here in terms of the process deliverables and how is the UML being used? The rest of this book should give an idea about what sort of diagram may be used for different applications, but it is equally important to look at the type of work being carried out. For example, the use of legacy systems in a project will often show a use for component and deployment diagrams, as well as interaction diagrams. A project that starts with a clean sheet of paper may start with use cases, and so on.

*For process applications*: If the UML is going to be used exclusively for process modelling, then it is important that the tool will support certain diagrams. For example if the approach taken in this book is followed, then the minimum subset of diagrams would be: class diagrams, activity diagrams and sequence diagrams.

*For requirement applications*: Once more, depending on the sort of process being followed, different diagrams would be needed. If a similar approach is taken to the one shown in this book, then the minimum set of diagrams would be: use case diagrams, class diagrams and interaction diagrams.

It is worth restating at this point that there are no solid rules for which diagrams to use for particular applications as the choice of diagram should be left to the modeller and their knowledge and experience.

### *11.4.6 Ensuring process compatibility*

It is important to ensure that a tool will support any processes that are being used on a project and that the tool will not try to drive the process. Although it is difficult to

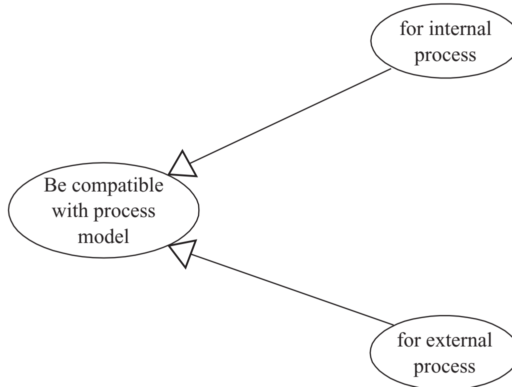


*Figure 11.6 Requirements for process compatibility*

discuss this in any detail, Figure 11.6 shows a simple breakdown of requirements for ensuring process compatibility.

The diagram in Figure 11.6 shows that there are two main requirements to meet in order to 'Be compatible with process model', which are 'for internal process' and 'for external process'.

Ensuring compliance with an internal process will depend entirely upon the approach being taken by the organisation for system development. The issue of external process, however, covers wider constraints, such as any standards or guidelines that the organisation has no control over (for example, ISO standards, industry best-practice guidelines, etc.) that may impact the project.

### *11.4.7 Provision of system functionality*

This section covers the pragmatic issues of tools selection concerning existing hardware and software requirements.

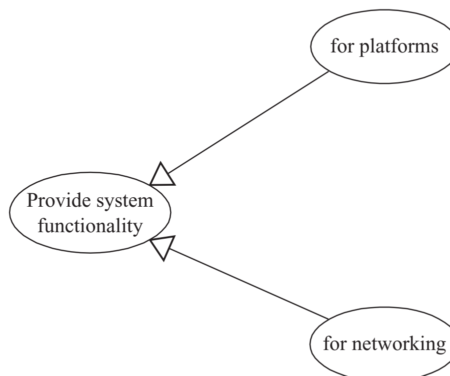


*Figure 11.7 Requirements for provision of system functionality*

The diagram in Figure 11.7 shows that there are two main requirements to 'Provide system functionality', which are: 'for platforms' and 'for networking'.

The issue of platforms raises questions such as: what are the minimum hardware requirements? What are the operating system requirements? Are there any other software requirements? It is also important to establish whether or not the tool will run on older operating systems since it is quite often the case with very large organisations, that they are using operating systems several versions behind the current market leaders.

It is also worth considering the networking requirements for the tool. This will cover different aspects such as access and security (as previously discussed) but also of licenses. Are floating licenses available, or do they require one per machine?

These very practical requirements can often mean that one tool that is suitable in terms of the functionality offered may not be so ideal in terms of being able to run on an existing network configuration.

### *11.4.8 Ensuring compatibility with the UML*

This question of compliance with the UML is often overlooked as being too obvious but can be quite a frightful, eye-opening experience at times! Although Figure 11.8 looks very simple, these two very simple requirements can often spell the doom of a tool.

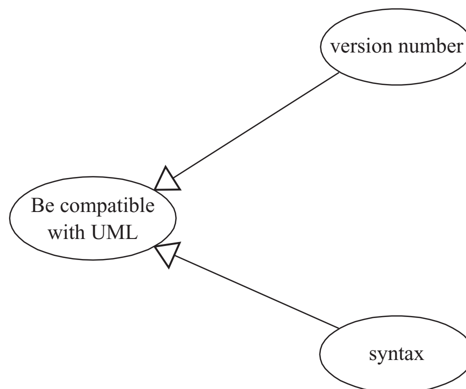


*Figure 11.8 Requirements for UML compatibility*

The diagram in Figure 11.8 shows that there are two main requirements associated with the compatibility of the tool with regard to the UML 'version number' and the 'syntax'.

The issue of the UML version number is quite straightforward – do you want version 1.x or version 2.0? Do you care? Some people will prefer version 1.3 as this is the version that will become the ISO standard, whereas other people will want to use the latest, most flexible version that is UML 2.0. One important point to make is you don't really need to make a black and white decision at this point, as version 2.0

is an evolution of 1.x and there is no reason why the two versions cannot be mixed and matched.

The second main requirement refers to the syntax of the diagram produced by the tool. When it comes to assessing the drawing capabilities of a CASE tool, the first and most obvious question (indeed, so obvious that most people do not bother to find it out) is to find out how many of the 13 diagrams the tool will allow to be created and whether or not the diagrams actually match the UML standard itself! Whenever this question is raised in a group discussion, the initial reaction is usually one of scorn and disbelief that anyone could produce a tool that does not match the UML. Some tools have the capability to model as few as four of the 13 diagrams, while others offer a facility to allow up to 17 different diagrams to be created – all of which are allegedly UML diagrams.

Once it has been established which of the 13 diagrams can be created, the next step is to see how each of the diagrams relates to the UML. Some diagrams follow the UML precisely, while others only bear a passing resemblance to their UML counterparts.

It is quite bizarre that so many tools do not measure up to the UML, particularly bearing in mind that the UML is very cleanly defined and has a single source document as its reference. The following list suggests reasons why some CASE tools may not be as UML-compliant as perhaps they should:

- Many CASE tools are an evolution of existing, or legacy, tools. The methodologies behind these legacy tools often provided an input to the UML, and thus many vendors may just assume that the diagrams used (for example, OMT) are the same as the diagrams in the UML. Unfortunately, these diagrams tend to be slightly different from their original language.
- Many companies are reluctant to throw away existing resources, so they may deem that it makes more business sense to reuse an older tool, rather than to redevelop it properly for a new market.
- It is also highly possible that some CASE tools are sometimes developed by people who do not actually use the technique themselves. An outsider's viewpoint of what may be useful may be completely different to what a practising UML specialist may deem as being useful.
- It is often the case that tool companies are looking for a market 'edge' and will focus their tool towards a specific market, such as real-time systems or safety-critical systems. In addition, the tools may be aimed purely at a single target, such as a particular programming language. Sometimes, this specialisation of a tool can be to the detriment of its compatibility with the UML.

On a more practical note, it is worth having a test model in mind that uses a few types of diagram that can be used to test the compatibility of a tool with the UML. A good example of this is the chess example used throughout Chapters 3 and 4 of this book that is simple, yet uses all 13 of the UML diagrams. Another useful thing to do is to enter the model yourself, rather than relying on a demonstrator to do it for you. This approach will provide a better indication of how easy the tool is to use and may very well throw up some quirks of the tool itself.

In summary, therefore, the following points should be looked at when assessing the drawing capabilities of a tool:

- Does the tool allow all 13 UML diagrams to be created and are the correct names used for each diagram?
- Does each of the diagrams that may be created actually match the syntax that is laid out in the UML standard?
- How does the tool allow navigation between different types of diagram?
- Have a sample model that may be input into the tool by way of a simple test. Choose something simple that will not take long to input and put the tool through its paces (such as the chess example in this book).

Remember that drawing is fundamental to UML and thus it is important that the tool can perform well.

### *11.4.9 Ensuring vendor's quality of service*

It is important that the tool is not the only thing taken into consideration, but also the vendor's quality of service. Bear in mind that there is a potential long-term relationship being started up between yourself and the vendor, so it is important that you are happy with both the people and the services offered by the vendor.

The diagram in Figure 11.9 shows the requirements to be considered when considering the vendor's quality of service. The first issue is that of the vendor's pedigree. It is worth considering how long the vendor has been in business and asking yourself will he still be there in five years? Also, look at the list of existing clients.

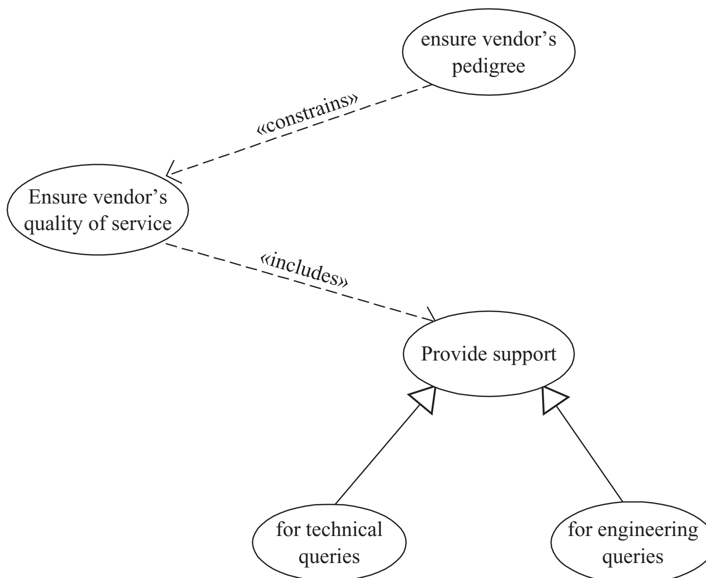


*Figure 11.9 Requirements for ensuring vendor's quality of service*

These lists always look very impressive but it is worth ascertaining where these clients are geographically. A list of clients may look very impressive to someone in, for example, the United Kingdom, but if the list is composed of 90 per cent of companies in other countries, then how good will the service be in the United Kingdom? The coverage of industries is important, as well as trying to get a feel for the types of applications that the tools are being used on. The quality of the vendor's service may also depend heavily on the individuals involved in the company. If a company is simply a reseller who only employs salespeople, then maybe their technical support is weak. If the people making the sales are, or are supported by, experienced engineers then this can be a great advantage in terms of the quality of responses to queries.

Overall, it is very important to make an informed decision when selecting a CASE tool for purchase. CASE tools can be very expensive and take up quite a percentage of the total projects costs, so it is imperative that one is chosen wisely. The following list contains a number of very high-level questions that you should ask when considering buying a CASE tool:

- Will the tool be used on all projects, or just one? Can the cost, therefore, be spread among the whole organisation? Will the organisation be adopting a single tool across the whole organisation, or will the current market be assessed and different tools chosen where appropriate? Clearly, there are pros and cons with each approach, but 'which is the most appropriate?' should be the main question.
- What will the total cost of the tool be in terms of the number of licences and the maintenance costs? The maintenance costs for a CASE tool when viewed over a five-year period can easily be as much as the tool cost in the first place. Bear this in mind and consider the whole life of the project, rather than the initial outlay for the tool.
- What other packages are required to use the tool effectively? This may include simple, inexpensive packages, such as word processors, to extra licences for a third-party database.
- What functionality will you actually use? If you have used the assessment criteria outlined previously in this chapter, you may have come to the conclusion that all that is required is the basic drawing functionality of the tool. In such a case, there is little point spending extra money on code generators that will never be used, when the basic package or a simple drawing package will suffice. If, on the other hand, you feel that you would make extensive use of these extra functions, it would be well worth spending extra money and buying a more complete tool than a simple drawing package.
- What level of support do the vendor's offer, in terms of upgrades to versions, documentation and on-line or telephone support? This can be critical, as having to bring in the vendor's own consultants for the day can be very expensive indeed.
- Does your process drive the tool? (This may have been mentioned previously in this book!)

Above all, like any type of project, it is vital that you know your requirements for the tool before talking to any vendors. Draw up a list of requirements and stick to it.

## 11.5 Conclusions

This chapter has introduced some generic criteria for assessing the suitability of a CASE tool. Although the tone has generally been quite negative, this is not necessarily the case. The right tool can save you time and effort and hence money, and it can make life far easier. By the same token, however, the wrong tool will cost you time and effort and hence money! The overall suggestion here is one of caution, rather than a single negative view.

Therefore, to summarise the chapter, bear the following points in mind:

- Draw up a list of requirements for the tool. Bear in mind that the process that you follow should drive the tool and not the other way around, as this can turn out to be very expensive.
- Check how many of the 13 UML diagrams the tool can create and whether they are correct or not. A great help in determining this is to have a simple model at hand that can be quickly entered into the tool. An example of this is the chess example introduced in Chapters 3 and 4 of this book. This will not only prove which diagrams can be created and how accurate they are, but will also give an idea about the use of tool and navigation between diagrams. Do not buy a tool on the basis of an example, preentered model, but try it for yourself!
- Check the licensing agreements fully and read all the small print. It is essential to buy the right edition of the tool and to be sure that you have an appropriate level of support from the vendors.
- Shop around between different vendors and obtain independent advice from existing users, as any company that refuses to give you the contact details of an existing user may have something to hide. Remember that you are the customer and should rightly have the upper hand in any negotiations.

One final point to remember is that CASE tools are not magic, but they can help out, and that the output of these tools is only as good as the engineer behind the tool.

## 11.6 Further discussion

1. Draw up a list of requirements for a CASE tool that would suit your exact needs. Make this as fanciful as possible but always ensure that there is a need behind each wish on the list. Now create a UML model that represents this list of requirements. As a starting point, consider either a use case model (both business and system contexts will be important) and also a class diagram that will represent the tool itself.
2. Obtain demonstration copies of two or more different CASE tools and compare them. What can either offer that the other does not? What do they both offer and, hence, which is the better or best?

3. Compare each tool with the model created in discussion point 1.
4. Take the chess example from this book and enter it into any CASE tool. Can it model 100 per cent of the UML syntax used in the model? Remember that this book only introduces approximately 15 per cent of the total UML syntax.
5. Perform a quick Internet search and see how many free CASE tools are available for download. How do these relate to the commercial tools?

# Bibliography

## Systems engineering references

These references are solely concerned with systems engineering.


BERTALANFFY, L.: ‘General systems theory – foundations, development, applications’ (George Brazillier, New York, 1968)

BIGNELL, V., and FORTUNE, J.: ‘Understanding systems failures’ (Open University Press, London, 1984)

COLLINS, T., and BICKNELL, D.: ‘Crash, ten easy ways to avoid a computer disaster’ (Simon and Schuster, London, 1997)

LEVESON, N. G.: ‘Safeware, system safety and computers’ (Addison-Wesley Publishing Inc., Reading, MA, 1995)

LEWIN, R.: ‘The major new theory that unifies all sciences, complexity, life on the edge of chaos’ (Phoenix Paperbacks, London, 1995)

O’CONNOR, J., and McDERMOTT, I.: ‘The art of systems thinking, essential skills for creativity and problem solving’ (Thorsons, London, 1997)

SKIDMORE, S.: ‘Introducing systems design’ (Blackwell Publishing, Oxford, 1996)

STEVENS, R., BROOK, P., JACKSON, K., and ARNOLD, S.: ‘Systems engineering, coping with complexity’ (Prentice Hall, Europe, 1998)


## Software-related references with systems content

The following references have a heavy systems content, despite the fact that they are primarily aimed at the software market.


AYRES, R.: ‘The essence of professional issues in computing’ (Prentice Hall, London, 1999)

BAINBRIDGE, D.: ‘Computer law’ (Pitman Publishing, London, 1996)

BROOKS, F. P.: ‘The mythical man-month’ (Addison-Wesley Publishing Inc., New York, 1995)

DAVIES, A. M.: 'Software requirements: analysis and specification' (Prentice-Hall International Editions, London, 1990)

FLOWERS, S.: 'Software failure: management failure, amazing stories and cautionary tales' (John Wiley and Sons Ltd, Chichester, 1997)

JACKSON, M.: 'Software requirements and specifications' (Addison-Wesley Publishing, Reading, MA, 1995)

SCHACH, S. R.: 'Classical and object-oriented software engineering' (Irwin, Singapore, 1996)

## Software engineering references

The following references are concerned only with software engineering and are included here for completeness sake.

BOOCH, G.: 'Object oriented analysis and design, with applications' (The Benjamin/ Cummings Publishing Company Inc., California, 1994)

COAD, P., and YOURDON, E.: 'Object-oriented analysis' (Yourdon Press, Englewood Cliffs, NJ, 1991)

GRAHAM, I.: 'Object oriented methods' (Addison-Wesley Publishing Company, London, 1994)

JACOBSON, I.: 'Object-oriented software engineering, a use case-driven approach' (Addison-Wesley Publishing Company, Washington, 1992)

PRESSMAN, R.: 'Software engineering: A practitioner's approach: European adaptation' (McGraw-Hill Publications Europe, 2000)

RUMBAUGH, J., BLAHA, M., PREMERLANI, W., EDDY, F., and LORENSON, W.: 'Object-oriented modelling and design' (Prentice-Hall, Englewood Cliffs, NJ, 1991)

SANDERS, J., and CURRAN, E.: 'Software quality: Framework for success in software development and support' (ACM Press Series, Addison Wesley, 1994)

SCHACH, S. R.: 'Software engineering with Java' (McGraw-Hill International Editions, Singapore, 1997)

SCHNEIDER, G., and WINTERS, J. P.: 'Applying use cases – a practical guide' (Addison Wesley, Reading, MA, 1998)

SHLAER, S., and MELLOR, S. J.: 'Object-oriented systems analysis – modelling the world in data' (Yourdon Press, Englewood Cliffs, NJ, 1988)

WARD, P. T., and MELLOR, S.: 'Structured development for real-time systems' (Yourdon Press, New York, 1985)

WIRFS-BROCK, R., WILKERSON, B., and WIENER, L.: 'Designing object-oriented software' (Prentice-Hall, Englewood Cliffs, NJ, 1990)

YOURDON, E., and CONSTANTINE, L. L.: 'Structured design: fundamentals of a discipline of computer program and systems design' (Prentice-Hall, Englewood Cliffs, NJ, 1979)

## UML references

These references are some of the books (the number is increasing constantly) that are concerned with the UML.

BOOCH, G., RUMBAUGH, J., and JACOBSON, I.: 'The unified modelling language user guide' (Addison-Wesley, Massachusetts, 1999)

CANTOR, M. R.: 'Object-oriented project management with UML' (John Wiley and Sons Inc., New York, 1998)

CHEESEMAN, J.: 'UML components: A simple process for specifying component-based software (Component-based Development)' (Addison Wesley, Reading, MA, 2000)

CONALLEN, J.: 'Building web applications with UML' (Addison-Wesley, Reading, MA, 1999)

DOUGLASS, B. P.: 'Real-time UML' (Addison Wesley, Massachusetts, 1998)

DOUGLASS, B. P.: 'Doing hard time' (Addison Wesley, Massachusetts, 2000)

ERIKSSON, H., and PENKER, M.: 'UML 2 toolkit' (John Wiley and Sons Inc., New York, 2003)

FOWLER, M., and SCOTT, K.: 'UML distilled (applying the standard object modelling language)' (Addison Wesley, Boston, 3rd edn, 2003)

LARMAN, G.: 'Applying UML and patterns – an introduction to object-oriented analysis and design' (Prentice Hall Inc., New Jersey, 1998)

LEE, R. C., and TEPFENHART, W. M.: 'UML and C++, a practical guide to object-oriented development' (Prentice Hall, New Jersey, 1997)

MULLER, P.: 'Instant UML' (Wrox Press Ltd, Paris, 1997)

ROYCE, W.: 'Software project management – a unified framework' (Addison Wesley, Massachusetts, 1998)

SCHMULLER, J.: 'SAMS teach yourself UML in 24 hours' (SAMS, Indiana, 1999)

SI ALHIR, S.: 'UML in a nutshell – a desktop quick reference' (O'Reilly and Associates Inc., Cambridge, 1998)

STEVENS, P., and POOLEY, R.: 'Using UML' (Addison Wesley, Harrow, 1999)

RADER, J. R.: 'Advanced software design techniques' (Petrocelli Books, London, 1978)

RUMBAUGH, J., JACOBSON, I., and BOOCH, G.: 'The unified modelling language reference manual' (Addison Wesley, Massachusetts, 1998)

## Standards

The following are references either to actual standards or to books that describe standards in detail.

DoDAF: 'A framework for command, control, communication, computers, intelligence, surveillance and reconnaissance (C4ISR) architecture development' (US Department of Defense)

EL EMAM, K., DROUIN, J. N., and MELO, W.: 'SPICE, the theory and practice of software process improvement and capability determination' (IEEE Computer Society, Washington, 1998)

ELECTRONIC INDUSTRIES ASSOCIATION: 'EIA 632, Processes for engineering a system' (EIA)

ELECTRONIC INDUSTRIES ASSOCIATION: 'EIA 731, Systems engineering capability model' (EIA)

HOYLE, D.: 'ISO 9000, pocket guide' (Butterworth Heineman, Oxford, 1998)

INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS: 'IEEE 1220, Application and management of the systems engineering process' (IEEE)

INSTITUTE OF ELECTRICAL AND ELECTRONIC ENGINEERS: 'IEEE 1471, Recommended practice for architectural description of software-intensive systems' (IEEE)

INTERNATIONAL ELECTROTECHNICAL COMMISSION: 'IEC 61508, Functional safety of electrical/electronic/programmable electronic safety-related systems' (IEC)

INTERNATIONAL STANDARDS ORGANISATION: 'ISO 12207, Software life-cycle processes' (ISO)

INTERNATIONAL STANDARDS ORGANISATION: 'ISO 15288, Lifecycle management – system life cycle processes' (ISO)

INTERNATIONAL STANDARDS ORGANISATION: 'ISO 15504, Software process improvement and capability determination (SPICE)' (ISO)

INTERNATIONAL STANDARDS ORGANISATION: 'ISO 9000-3, Guidelines for the application of 9001 to the development supply and maintenance of software' (ISO)

INTERNATIONAL STANDARDS ORGANISATION: 'ISO 9001, Model for quality assurance in design, development, production, installation and servicing' (ISO)

JACOBSON, I., BOOCH, G., and RUMBAUGH, J.:, 'The unified software development process' (Addison Wesley, Massachusetts, 1999)

KRUCHTEN, P.: 'The rational unified process: an introduction' (Addison-Wesley Publishing, Massachusetts, 1999)

MAZZA, C., FAIRCLOUGH, J., MELTON, B., DE PABLO, D., SCHEFFER, A., and STEVENS, R.: 'Software engineering standards' (Prentice Hall, Herfordshire, 1994)

OMG: 'Unified modelling language superstructure – Version 2.0' (Object Management Group)

## Other useful resources

The following is a short list of some very useful resources on the Internet. Although the list is brief, the addresses form an excellent starting point for looking for any sort of information on systems engineering, or the UML.

WWW.INCOSE.ORG (The official website for the International council on Systems Engineering who have an excellent set of links to almost all aspects of systems engineering)

WWW.IEE.ORG (The official website for the Institution of Electrical Engineers, who provide many systems engineering resources including access to the Professional Network for Systems Engineers)

WWW.OMG.ORG (The official website for the Object Management Group who have control over the UML standard. The actual standard may be downloaded from this site along with a lot of other UML information)

# Index


activity diagram 59, 72, 80, 131–8
  elements 131–4
  examples and modelling 134–7
  graphical representation of elements in an 133
  partial meta-model for 131
  swimlanes and object flow 187
  using 137–8
alternative state machine modelling 70–2
  comparing approaches 71
  actions and activities 70
  *see also* state machine modelling
analysis model architecture
  high-level scenarios 299–300
  low-level scenarios 300
  views 300
animation tools and CASE tools 331
ANSI (American National Standards Institute) 18
application domains and UML 36, 39
approach to systems engineering, good 12–13
architecture
  definitions 279–82
  examples 286–91
  generating 292–3
  for project validation, using an 293–5
  requirements model 298–9
    lower-level view showing structure 294
  simple information model, high-level view 293
  validation with 291–5
  verification and validation views 283
  versus model 282
  views 282–4
    conceptual 283
    lower-level initial project definition 294
    operational 282
    physical 283
    process 283
assessment of organisation 20
associated class name 202
audit of organisation 20
autopilots for aeroplanes and ships 16

behavioural modelling 59–73
  adding more detail 64–6
  class diagram, changing the 68–9
  diagrams
    levels of abstraction 60
    types of 72
  ensuring consistency 66–7
  example 63–7
  simple behaviour 63–4
  solving inconsistency 67–9
  state machine diagrams
    changing 68
    using 60–2
bibliography 345–9
Booch 37–8
  example of modelling 26
    doghouse 25
    house 26–7
    office block 27–8
braking systems in cars 16
British Standard 14
BSI (British Standards Institution) 18
business context 220–1

C4ISR *see* command, control, communication, computers, intelligence, surveillance and reconnaissance
capability profile 20
CASE (computer aided/assisted software engineering) tools 322
  for ensuring process compatibility 337–8
  evaluation, requirements model for 325–43
  nonfunctionality of 324
  UML-compliant 39
  *see also* tools
case diagram 59, 72, 79
class diagrams 43–4, 79
  adding more detail 47–8
  aggregation and composition 48–50
  basic modelling 44–7
  example diagrams and modelling 84–8
  instantiation 53–4
  meta-model for 56
  specialisation and generalisation 50–3
  structural 82–8
    elements 82–4
    graphical symbols for elements 84
  using 88
classes
  and relationships, modelling 44
  and state machine diagrams, meta-model relationship between 107
closed loop control systems 15
code generation 331–3
  facilities, assessing 333
collaboration diagram 80–1
command, control, communication, computers, intelligence, surveillance and reconnaissance
(C4ISR) architecture development 284
common language, advantages 12
communication diagram 80, 113–17, 241
  elements 113–17
    graphical representation of 114
  examples and modelling 115
  and other UML elements, meta-model showing relationship between 114
  partial meta-model for 114
  using 117
communications 9–12
  levels of 9–10
  versus sequence diagrams 240–1
complexity
  accidental 7–9
  communication and understanding 7–13
  essential 7–8
  minimising 9
component diagrams 43, 79, 145–51, 299
  elements 146
    graphical representation of 147
  examples and modelling 148–51
  partial meta-model for 146
  using 151
composite structure diagrams 43, 94–101, 299
  elements 95–6
    graphical symbols for the 96
  examples and modelling 97–100
  meta-model of the 96
  and other parts of the UML, relationships between 96
  using 100–1
configuration management and change control and CASE tools 330
constraints 312–13, 319
  formal object constraint language (OCL) 313
  in profile creation 306
  types of 313
context modelling 219–26
  practical 222–6
  types of 219–22
CORBA (Common Object Request Broker Architecture) 18
CRC 38
cross-boundary communication 12

deliverables, defining 194–7
  by process 195
  by structure 196
  by type 194
Department of Defense Architectural Framework (DoDAF) 284–6
  types
    of architecture in 285
    of product identified in 285
deployment diagram 43, 79, 152–7, 299
  elements 152–4
    graphical representation 153
    and rest of UML, relationship between 154
  examples and modelling 154–7
  partial meta-model for 152
  using 157
DERA (Defence Evaluation and Research Agency) 18

design model architecture 301–2
  conceptual view 301
  operational view 301
  physical view 301
  process view 301
  verification and validation view 302
development
  environment and UML 36
  life cycles and UML 39
  processes and UML 39
diagrams, types of
  grouped into types of models 160
disasters and failures in systems 3–4
DoDAF *see* Department of Defense Architectural Framework

EIA 632 173
  comparing models 173–4
  high-level structure of 169
  highlighting complexity 174
  processes for engineering a system 164, 168–70
  types of process in 169
EIA 731 standard for systems engineering capability 18
  model 164, 181–2
EN (European Normative) 18
EN 50128 Railway applications 19
Europe 180
  quality mark 'CE' symbol 14
European Space Agency (ESA) 18

frozen phases in linear models 256
Fusion method 38

GANTT chart 122–3, 274
  in life cycle management 275
global market, increased size of 2
global village 2

Hewlett-Packard 38
historical methods and notations and UML 39

IBM 38
ICON Computing 38
IEC (International Electrotechnical Commission) 18
IEC 61508 functional safety of electrical/electronic/programmable electronic safety-related systems 164, 178
IEE (Institution of Electrical Engineers) 18
IEEE (Institute of Electrical and Electronic Engineers) 18
IEEE 1220 application and management of systems engineering process 165, 179
IEEE 1471 286–91
  conceptual model of architectural description 287
  stakeholder and concerns identification 290
I-Logix 38
implementation languages and UML 39
implementation model 199, 201–2
industry standards 18
information
  management in CASE tools 323
  model 199–200
in-house standards 18
integration of systems 3
Intellicorps 38
interaction diagrams 72, 81, 112–30
  communication, timing, interaction overview and sequence diagrams 59
  graphical representation of elements in an 124
  meta-model showing types of 112
  partial meta-model for the 123
interaction overview diagrams 123–6
  elements 123–4
  examples and modelling 124–6
  using 126
International standards 18
invariants as type of constraint 313–14
ISO (International Standards Organisation) 17–18
  work practice, activity diagram for an 136
ISO 12207 software life cycle processes 165
ISO 15288 257–60
  detailed view of 'Enterprise' processes in 261
  focus on stages of 258
  high-level view of 257
  life cycle management, system life cycle processes 165, 178
ISO 15504 software process improvement and capability determination (SPICE) 18, 165, 168, 171, 173
  comparing models 173–4
  high-level structure of 171
  highlighting complexity 174
  types of process category in 172

ISO 9000-3 guidelines for application of 9001 to development, supply and maintenance of software 164
ISO 9001 model for quality assurance in design, development, production, installation and servicing 19, 164
ISO/IEC 15288 207, 257, 266
  focus on decision gates in 258
  types of 'process group' in 258
Ivar Jacobson 260

James Martin and Company 38

kennel (doghouse) model 25–6
'kennel' mentality 28
knowledge, adding more to UML 308–9

languages 11
  common 12
  modelling 11
  programming 11
life cycle management 251–78
  GANTT chart with process model concepts 275
  project planning 273–7
life cycle models
  different 253–63
  iterative 256–7
  linear 254
    versus iterative 253
  process behaviour 273
life cycles
  creating 193
  definition of 252
  and life cycle models 251–3
  phases and UML 36
  support tools and CASE tools 331
  typical 252
Life line 80
link name 202

maintenance feedback requirements 209
MCI Systemhouse 38
model
  abstraction of 32
  aspects of 33
  behavioural aspect, seven diagrams for realising 59
  choice of 31
  defining 30–1
modelling 25–40
  connection to reality 32
  importance of 25
  independent views of same system 32
  need for 29–30
  principles of 31–2
modelling requirements 205–48
  document, populating 243
  of system 227–32
modelling standards processes and procedures 163–203
  and defining new processes using UML 182–91
  tailoring process 191–4
modelling techniques
  traditional 37
  unification of different 38

need for systems engineering 2–3

object constraint language *see* OCL
object diagrams 43, 79, 88–94
  elements 89–90
  example diagrams and modelling 90–4
  using 94
object flow 187
Object Management Group (OMG) 38, 305
object modelling technique (OMT) 37–8
object name 202
Object Time 38
object-oriented (OO) modelling techniques 37
  systems analysis 37
object-oriented software engineering (OOSE) 38
OCL (object constraint language)
  bag 317
  collection 316
  description of user-defined types 316
  expression, structure of 315
  sequence 317
  set 316
  types of UML elements 316
Open Data Base Connectivity (ODBC) 18
office suite packages and CASE tools 331
Oracle 38

package diagrams 43, 79
  elements
    graphical symbols for 102
    and rest of UML, relationships between 102
  examples of 103
  partial meta-model for 101

structural 101–4
using 104
Pert charts 122–3
Petri nets 80
Platinum Technology 38
postconditions 314–18
concepts concerning 315
preconditions
concepts concerning 315
types of constraint 314–18
PRINCE system 213
problems in systems 1–6
causes of 4–5
procedure, defined types of process group for 184
process
analysis 202
applications, CASE tools for 337
defined high-level process structure model for 183
definition and help in CASE tools 323
implementation profile 201
model and UML 36
processes in organisations 21–2
profile
background to process modelling example 306
creating 306–18
declaring 309–10
definition 200–1
project failures 4
reasons 29
project management tools and CASE tools 330
prototype 210
publicly-available specification (PAS) 18

quality 3, 14–22
need for 15
processes and 17

Rational Software Corporation 38, 260
Rational Unified Process (RUP) 136, 180, 226, 257, 260–3
high-level view of 262
types of 'Core workflow' in 262
real-time applications, CASE tools for 337
relationship, types of 230
report generation 334
assessment criteria for 335
and CASE tools 334
requirements
applications, CASE tools for 337
business 209, 211
capturing 209–10
concepts and UML elements, summary of relationships between 247
documenting 241–4
functional 213
implementation issues 213
life cycle issues 213
management tools and CASE tools 330
modelling 226–35
concepts 246
stakeholders 215–19
non-functional 213
phase 206–9
properties of 213–15
quality issues 213
types of 211, 230
user document, typical contents of 242
requirements architecture model
high-level structural model 298
legacy component 299
requirements engineering 205
basics 206–19
requirements problems
duplication 215
irrelevancies 215
overdescription 215
responsibility
complex systems 15
level of 15
reverse engineering 331
assessment criteria for 334
and CASE tools 333–4
Rumbaugh's object modelling technique (OMT) 226
RUP *see* Rational Unified Process

saving on training 12
scalability in linear models 255
scenarios
example 237–40
modelling 236–41
sequence diagram 81, 117–23, 240–1
elements 118
examples and modelling 118–22
using 122
view of Waterfall model 255
software
applications, CASE tools for 337
engineering references 346
related references with systems content 345

stakeholder
  requirements 206
    process 207–8
  types of 215–18
standards 164–7, 347
  analysing 167–82
  aspects of modelling using UML 167
  compliance with 20–1
  consequences of problems with 20
  modelling example 168
  problems with 18–19
  and quality 17–18
  requirements for using 166–7
  types of process comparison 175
state machine diagrams 59–60, 72, 80
  action-based 70
  activity-based 70
  alternative 70–2
    comparing approaches 71
  basic modelling 61–2
  behavioural 104–12
  element 104–8
    graphical representation 107
  examples and modelling 108–11
  partial meta-model for 106
  using 112
statechart diagram 80
stereotypes 319
  declaring and defining 308
  icons 311
  identifying and reading 307
  in profile creation 306
  name 202
  symbols 310–11
structural diagrams 54
structural modelling 43–57
  types of 43
    package of 55
  using class diagrams 44
Structure Analysis and Structured Design (SASD) 37
Structured Analysis for Real-time Systems 37
STUMPI (Students Managing Processes Intelligently) 263
  example of life cycle management 263–73
  life cycle 264–5
    model 265–6
  phase iterations 270–2
  process group, types of 267
  process model 206–7, 266–70
  stake holder diagram for 270
swimlanes 80, 133, 187
SysML 305
systems
  applications, CASE tools for 337
  context 221–2
  defining 5–6
  tool, use for 198–9
systems architecture 279–303
  analysis, lower-level view focusing on 296
  applying UML 298–302
  common themes 291
  design model, lower-level view focusing on 297
systems engineering 1–23
  defining 2, 6
  definitions of 22
  issues affecting 22
  process, implementing using tool 197–202
  references 345
  tool, model of popular 198
systems professionals
  quality policies, standards and procedures 16
  requirements for 16
  roles and responsibilities 16
  safety policies 16

tagged values 319
  in profile creation 306
  as UML extension mechanism 318
tailoring processes using a specialisation 192
target environment, working requirements 209
Taskon 38
technological performance, increased 2
Texas Instruments 38
timing diagrams 81, 126–30
  elements 127–30
    graphical representation of 128
  examples and modelling 128
  partial meta-model for 127
  using 130
tools 321–44
  functions of 322–4
  selecting 322
  and UML compliance 340–1
  *see also* CASE tools
tools, requirements
  additional capabilities 331
  application capability 336–7

application functionality 336
ensuring vendor's quality of service 341–3
generic criteria for assessing suitability of a 343–4
interoperability capability 329–31
modelling capability provision 326–9
process compatibility 338
system functionality 338–9
tool capability 332
UML compatibility 339–41
*see also* CASE tools
traceability
in CASE tools 323
paths 293

UML 18, 33–9
advantages 40
background of the 35
basic extension mechanisms 319
diagrams 33, 75–161
13 types of 159
behavioural aspect of model in 34
comparison of the types of 78–81
seven types of 59
extending 35
history of 36
meta-model 81–2
modelling techniques 37–8
models and their associated diagrams, summary of 55
as non-proprietary language 39
provision of unification in 39–40
references 347
rules, when modelling using 158
scenarios in 236
scope of 36
UML 1.x and UML 2.0
comparison of behavioural diagrams 80
comparison of structural diagrams 78
conceptual changes between 76
UML 2.0
component diagrams in 146, 151
composite structure diagram 94
diagrams, structure of 77
new graphical notation in state machine diagram 109
package diagrams 101
terminology 76–7
Unified Method 38
Unified Modelling Language *see* UML
Unified Process (UP) 260
Unisys 38
United States 180
use case diagrams 138–45
elements 139–41
graphical representation of 140
examples and modelling 141–5
partial meta-model for 139
using 145, 219
use cases
descriptions 232–5
confusion with visual 233–4
relationships, defining new stereotypes for 231
statechart description of a 234
text descriptions 234
and visual descriptions 235
and other UML elements, meta-model showing relationships between 141
users
group requirements 209
modifications 210
and stakeholders, relationship between 216

validation of designs 324
VDM or Z (formal specification languages) 35
views versus models 34

Waterfall model 254–5
websites
www.iee.org 349
www.incose.org 348
www.omg.org 349
word-processing packages and CASE tools 335
workflow 136

# UML for Systems Engineering: watching the wheels

## Second Edition

Up until a few years ago there were many different modelling languages available to software developers. However, this vast array of choice only served to hinder communication and as a result the Unified Modelling Language (UML) was born. Although the UML has its roots firmly in the software world, the benefits of adopting a standard visual notation have been recognised in many other fields, not least of which is the field of systems engineering. This book concentrates on systems-based applications, rather than the traditional software applications that are more usually associated with the UML. Now fully updated to reflect the changes to UML for its version 2.0 release, this new edition has been substantially re-written and includes new material on systems architectures and life cycle management.

**Jon Holt** is the founder-director of Brass Bullet Ltd, a company specialising in systems engineering consultancy, services and training. He is actively involved in promoting systems engineering through the IET and other professional bodies.

**The Institution of Engineering and Technology**
**www.theiet.org**
0 86341 354 4
978-0-86341-354-4