Osa VII

Kehittyneet aiheet

Oppitunnit

Osa

VII

22. oppitunti

Oliopohjainen analyysi ja suunnittelu

Ohjelmoinnin keskipisteeksi tulee helposti C++ -kielen syntaksi ja tällöin menetetään näkemys siitä, kuinka ja miksi näitä tekniikoita käytetään ohjelmien tekemiseen. Tässä luvussa käsitellään seuraavia aiheita:

- Kuinka ongelmia tarkastellaan oliopohjaisesta perspektiivistä
- Kuinka ohjelma suunnitellaan oliopohjaisesta perspektiivistä
- Kuinka otat suunnittelussa huomioon uudelleenkäytettävyyden ja laajennettavuuden

Kehitysvaiheet

Ohjelmien kehitysvaiheista on tehty runsaasti kirjoja. Jotkut asiantuntijat ehdottavat vesiputousmallin käyttöä, jossa suunnittelijat määrittävät, mitä ohjelman tulee tehdä, arkkitehtuurin suunnittelijat määrittävät, kuinka ja mistä luokista ohjelma rakennetaan, jne. Lopuksi ohjelmoijat toteuttavat suunnitelman. Kun suunnitelma tulee ohjelmoijille, on se jo valmis; ohjelmoijien on vain toteutettava määritelty toiminnallisuus.

Vaikka vesiputousmalli toimisikin, se olisi kuitenkin huono menettely hyvien ohjelmien aikaansaamiseksi. Ohjelmoijan jatkaessa työtään tulee suunnittelusta muutoksia prosessiin. Vaikkakin on totta, että hyvä C++ -ohjelma suunnitellaan hyvinkin tarkasti ennen koodin kirjoittamista, suunnitelma ei kuitenkaan pysy muuttumattomana koko kehitysjakson ajan.

Suunnitteluaika riippuu ohjelman koosta. Monimutkainen ohjelma, jota on tekemässä kymmeniä ohjelmoijia usean kuukauden ajan, vaatii hyvin toteutetun arkkitehtuurin, eikä siihen riitä mikään yksittäisen ohjelmoijan tekemä yhden päivän kirjoitelma.

Tämän luvun painopiste on laajojen, monimutkaisten ohjelmien suunnittelussa. Noita ohjelmia tulee voida laajentaa ja hyödyntää usean vuoden ajan. Useat ohjelmoijat nauttivat työskentelystä teknologian äärirajoilla; he tekevät mielellään ohjelmia, joiden monimutkaisuus on lähellä heidän työkalujensa ja ymmärtämyksensä rajoja. C++ suunniteltiin monipuoliseksi ja riittäväksi työkaluksi ohjelmoijille ja ohjelmoijaryhmille.

Tässä luvussa tutkitaan erilaisia suunnitteluongelmia oliopohjaisesta näkökulmasta. Tavoitteena on tutustua määrittelyprosessiin ja oppia sitten ymmärtämään, kuinka nuo suunnittelutavoitteet toteutetaan C++ -kielellä.

Hälytysjärjestelmän simulointi

Simulaatio on tietokonemalli osasta todellista järjestelmää. Simulaation rakentamiselle on monta syytä, mutta hyvän suunnittelun tulee alkaa tietämyksestä, mihin kysymyksiin simulaation tulisi vastata.

Tutkikaamme tässä lähtöpisteessä ongelmaamme: sinua on pyydetty simuloimaan talon hälytysjärjestelmää. Talo on kaksikerroksinen ja kellarikerroksessa on vielä autotalli.

Alakerrassa on seuraavat ikkunat: kolme keittiössä, neljä ruokailuhuoneessa, yksi pesuhuoneessa, kaksi olohuoneessa, kaksi perhehuoneessa ja kaksi pientä ikkunaa oven lähellä. Kaikki neljä makuuhuonetta ovat yläkerrassa. Jokaisessa makuuhuoneessa on kaksi ikkunaa, paitsi suurimmassa makuuhuoneessa, jossa on neljä ikkunaa.

Yläkerrassa on kaksi kylpyhuonetta, joissa on kummassakin yksi ikkuna. Kellarikerroksessa on neljä puoli-ikkunaa ja yksi ikkuna autotallissa.

Taloon mennään pääovesta. Keittiössä on lasinen liukuovi ja autotallissa on kaksi suurovea autoille ja yksi pikkuovi. Takaosassa on vielä kellarin ovi.

Kaikki ikkunat ja ovet ovat hälytysjärjestelmässä ja kullekin puhelimelle on paniikkinappi ja suurimmassa makuuhuoneessa on vielä yksi paniikkinappi vuoteen vierellä. Perustukset kuuluvat myöskin hälytysjärjestelmään, vaikkakin ne on säädetty karkeammin, etteivät pienet eläimet tai linnut aseta hälytyksiä päälle.

Kellarikerroksessa on hälytysjärjestelmän keskus. Siitä lähtee varoitusääni, kun hälytys on päällä. Jos hälytysjärjestelmä on otettu pois päältä, kutsutaan poliisi tietyn ajan kuluttua. Jos paniikkinappia painetaan, kutsutaan poliisi välittömästi.

Hälytysjärjestelmä langoitetaan takkaan ja savuilmaisimiin sekä sprinkler-järjestelmään. Itse hälytysjärjestelmässä on vianesto, oma sisäinen varmistustehonsyöttö ja se on sijoitettu tulenkestävään koteloon.

Käsitteellistäminen

Uusi käsite

Käsitteellistämisvaiheessa yritetään selvittää, mitä asiakas toivoo ohjelmalta: mitä varten ohjelma kehitetään? Mihin kysymyksiin simulaation tulisi vastata? Simulaatiota saatetaan käyttää vastaamaan esimerkiksi seuraaviin kysymyksiin: "Kuinka kauan voi anturi olla rikki ennen kuin joku huomaa sen?" tai "Voidaanko ikkunahälytys ottaa pois päältä ilman ilmoituksen menemistä poliisille?"

Käsitteellistämisvaiheessa on hyvä miettiä, mitä kuuluu ohjelman sisälle ja mitä jää ulkopuolelle. Kuuluuko poliisin esittäminen simulaatioon? Onko todellisen hälytyksen kontrolli itse järjestelmän sisällä?

Analyysi ja vaatimukset

Käsitteellistämisvaihe johtaa analyysivaiheeseen. Analyysin aikana on oliopohjaisen analysoijan tehtävänä auttaa asiakasta ymmärtämään, mitä hän vaatii ohjelmalta. Kuinka ohjelman tulee käyttäytyä? Millaisia toimintoja ohjelman tulee toteuttaa?

Uusi käsite

Vaatimukset tuottavat tyypillisesti joukon dokumentteja. Näissä dokumenteissa voi olla kuvauksia. Käyttökuvaus on esitys siitä, kuinka järjestelmää käytetään. Siinä kuvataan tapahtumat ja käytetään malleja, jotka auttavat ohjelmoijaa todentamaan itselleen järjestelmät tavoitteet.

Korkean tason ja matalan tason suunnittelu

Kun tuote ymmärretään täysin ja vaatimukset on kirjoitettu dokumentaatioon, on aika siirtyä korkean tason suunnitteluun. Tässä vaiheessa ei vielä välitetä alustasta, käyttöjärjestelmästä tai ohjelmointikielestä. Sen sijaan keskitytään siihen, kuinka ohjelma toimii: mitkä ovat pääkomponentit? Kuinka ne ovat yhteydessä toisiinsa?

Eräs keino lähestyä tätä ongelmaa on laittaa käyttöliittymään liittyvät asiat sivuun ja kiinnittää huomio itse ohjelman komponentteihin.

Uusi käsite Ohjelma-avaruus on ohjelmien ja asioiden joukko, jota yritetään ratkaista. Ratkaisuavaruus on mahdollisten ratkaisujen joukko.

Kun korkean tason suunnittelu aloitetaan, joudutaan miettimään esille tulleiden olioiden vastuualueet: mitä ne tekevät ja mitä tietoa niihin tallennetaan. Esille tulevat myös riippuvuudet: kuinka oliot ovat vuorovaikutuksessa toistensa kanssa.

Olettakaamme, että meillä on esimerkiksi eri tyyppisiä antureita, keskushälytysjärjestelmä, painonappeja, lankoja ja puhelimia. Myös huoneita, ehkäpä lattioita ja mahdollisesti ihmisryhmiä kuten omistajat ja poliisi, tulee simuloida.

Anturit voidaan jakaa liikeilmaisimiin, kosketuslankoihin, ääni-ilmaisimiin, savuilmaisimiin jne. Kaikki nuo kohteet ovat jotain anturityyppiä, vaikkakaan ei ole olemassa mitään tiettyä anturipohjaa. Tämä osoittaa hyvin, että anturi voisi olla abstrakti tietotyyppi (ADT).

ADT-tyyppinen Sensor-luokka tarjoaisi kokonaisvaltaisen liittymän kaikentyyppisille antureille ja kullekin johdetulle tyypille luodaan toteutus. Eri antureiden asiakkaat käyttäisivät niitä tarvitsematta tietää, mitä tyyppiä ne ovat, ja kukin tekisi juuri sen tehtävän, joka niiden kuuluukin tehdä.

Hyvän ADT-luokan luomiseksi on ymmärrettävä täysin, mitä anturit tekevät (ei niinkään, kuinka ne toimivat). Ovatko anturit esimerkiksi passiivisia vai aktiivisia? Odottavatko ne, että jokin osa kuumenee, lanka katkeaa, jokin aine sulaa vai signaloivatko ne muuten ympäristöönsä. Ehkäpä joillakin antureilla on vain binäärinen tila (poissa tai päällä), mutta toiset taas ovat analogisempia (mikä on tämän hetken lämpötila). Liittymän abstraktiin tietoon tulee olla tarpeeksi hyvä käsittelemään kaikki johdettujen luokkien esille tulevat tarpeet.

Muut kohteet

Suunnittelu jatkuu tällä tavalla synnyttäen lukuisia uusia luokkia, joita tarvitaan vaatimusten täyttämiseksi. Esimerkiksi, jos pidetään lokikirjaa, tarvitaan ehkä ajastin: tulisiko ajastimen pollata kutakin anturia vai tulisiko jokaisen anturin raportoida itse omasta tilastaan säännöllisesti?

Käyttäjän tulee voida asettaa, ottaa pois päältä ja ohjelmoida järjestelmää, joten tarvitaan jonkinlainen käyttöjärjestelmä. Itse ohjelmalle tarvittaneen erillinen olionsa simulaatiossa.

Mitä luokkia tarvitaan?

Ongelmia ratkaistaessa aloitetaan luokkien suunnittelulla. Olet jo todennut, että esimerkiksi HeatSensor johdetaan Sensor-luokasta. Jos anturi raportoi säännöllisesti, se perii ominaisuuksia myös ajastimelta (Timer) tai ajastin on sen yhtenä jäsenmuuttujana.

Lämpöanturilla on luultavasti sellaisia jäsenfunktioita kuin CurrentTemp() ja SetTempLimit() ja se perinee sellaisia funktioita kuin SoundAlarm() perusluokaltaan Sensorilta.

Kapselointi kuuluu oliopohjaiseen suunnitteluun. Voisit kuvitella suunnitelmaa, jossa hälytysjärjestelmällä on MaxTemp-asetus. Hälytysjärjestelmä kysyy lämpöanturilta nykyistä lämpötilaa ja vertaa sitä maksimilämpötilaan ja suorittaa hälytyksen, jos se on liian korkea. Joku voi väittää, että tämä on ristiriidassa kapseloinnin kanssa. Olisi ehkä parempi, jos hälytysjärjestelmä ei tietäisi tai välittäisi lämpötila-analyysin yksityiskohdista - se voitaisiin asettaa HeatSensorissa.

Juuri edellä kerrottu tilanne on eräs päätöslaji, jota pohditaan ongelmia analysoitaessa. Jatkaaksemme analyysiä voisimme väittää, että vain anturin ja Log-olion tulisi tietää yksityiskohdat siitä, kuinka anturin toiminnot kirjataan; Alarm-olion ei tulisi tietää tai välittää siitä.

Hyvän kapseloinnin tunnuksia on se, että kullakin luokalla on tiukka ja täydellinen vastuualuejoukkonsa eikä muilla luokilla ole samoja vastuita. Jos Sensor-luokka on vastuussa nykyisen lämpötilan huomaamisesta, ei muilla luokilla tulisi olla sitä vastuuta.

Toisaalta toiset luokat voisivat auttaa välttämättömän toiminnallisuuden tarjoamisessa. Esimerkiksi, vaikka Sensor-luokan tulisi huomata ja kirjata nykyinen lämpötila, se voisi toteuttaa vastuunsa delegoimalla Log-oliolle todellisen tiedon tallentamisen.

Vastuiden jakaminen huolellisesti tekee ohjelmasta helpommin laajennettavan ja ylläpidettävän. Kun päätät muuttaa hälytysjärjestelmää tietyn moduulin kohdalla, on sen liittymä kirjaukseen ja antureihin kapea ja

hyvin määritelty. Hälytysjärjestelmän muutosten ei tulisi vaikuttaa Sensor-luokkiin ja päinvastoin.

Tulisiko HeatSensor-luokalla olla ReportAlarm()-funktio? Kaikkien antureiden tulee voida raportoida hälytys. Selvästikin ReportAlarm()-funktion tulisi olla Sensor-luokan virtuaalinen metodi ja Sensor-luokan kuuluu olla abstrakti perusluokka. On mahdollista, että HeatSensor liittyy Sensor-luokan yleisempään ReportAlarm()-metodiin; korvattu funktio voi täydentää yksityiskohdat vaaditulla tavalla.

Kuinka hälytykset raportoidaan?

Kun anturit raportoivat hälytystilanteen, ne antavat paljon tietoa oliolle, joka soittaa poliisille ja lokikirjaan. Ehkäpä on hyvä luoda Condition-luokka, jonka muodostin hoitaa useita mittauksia. Riippuen mittausten monimutkaisuudesta, ne voisivat myöskin olla olioita tai ne voisivat olla yksinkertaisia skalaariarvoja kuten kokonaislukuja.

On mahdollista, että Condition-oliot viedään keskushälytysjärjestelmälle; tai että Condition-oliot ovat Alarm-olioiden aliluokkia, jolloin ne voivat itse hoitaa hälyttämisen. Ehkäpä keskusoliota ei tarvita lainkaan; sen sijaan on antureita, jotka osaavat luoda Condition-olioita. Jotkut Condition-oliot tietäisivät, kuinka ne itse kirjataan; toiset taas osaisivat ottaa yhteyttä poliisiin.

Hyvin suunniteltu, tapahtumaohjattu järjestelmä ei tarvitse keskuskoordinaattoria. Voitaisiin kuvitella, että kaikki anturit ottavat itsenäisesti vastaan ja lähettävät viestiolioita toinen toiselleen asettaen parametreja, ottaen lukemia, valvoen taloa. Kun jokin vika havaitaan, luodaan Alarm-olio, joka kirjaa ongelman (lähettämällä viestin Log-oliolle?) ja suorittaa sopivan toiminnon.

Tapahtumasilmukat

Simuloidakseen tapahtumaohjattua järjestelmää ohjelmasi on luotava tapahtumasilmukka. Tapahtumasilmukka on yleensä päättymätön silmukka kuten while(1), joka saa viestejä käyttöjärjestelmältä (hiiren klikkaus, näppäinpainallus, jne) ja käsittelee ne yksi kerrallaan palaen silmukkaan kunnes lopettamisehto toteutuu. Listaus 22.1 esittää karkean tapahtumasilmukan.

Listaus 22.1. Yksinkertainen tapahtumasilmukka.

```
1:  // Listaus 22.1
2:
3:  #include <iostream.h>
4:
5:  class Condition
6:  {
```

```
7:  public:
8:    Condition() { }
9:    virtual ~Condition() {}
10:   virtual void Log() = 0;
11: };
12:
13: class Normal : public Condition
14: {
15: public:
16:   Normal() { Log(); }
17:   virtual ~Normal() {}
18:   virtual void Log() { cout << "Logging normal conditions...\n"; }
19: };
20:
21: class Error : public Condition
22: {
23: public:
24:   Error() {Log();}
25:   virtual ~Error() {}
26:   virtual void Log() { cout << "Logging error!\n"; }
27: };
28:
29: class Alarm : public Condition
30: {
31: public:
32:   Alarm ();
33:   virtual   ~Alarm() {}
34:   virtual void Warn() { cout << "Warning!\n"; }
35:   virtual void Log() { cout << "General Alarm log\n"; }
36:   virtual void Call() = 0;
37:
38: };
39:
40: Alarm::Alarm()
41: {
42:   Log();
43:   Warn();
44: }
45: class FireAlarm : public Alarm
46: {
47: public:
48:   FireAlarm(){Log();};
49:   virtual ~FireAlarm() {}
50:   virtual void Call() { cout<< "Calling Fire Dept.!\n"; }
51:   virtual void Log() { cout << "Logging fire call.\n"; }
52: };
53:
54: int main()
55: {
56:   int input;
57:   int okay = 1;
58:   Condition * pCondition;
59:   while (okay)
60:   {
61:     cout << "(0)Quit (1)Normal (2)Fire: ";
62:     cin >> input;
63:     okay = input;
64:     switch (input)
65:     {
66:       case 0: break;
67:       case 1:
```

```
68:         pCondition = new Normal;
69:         delete pCondition;
70:       break;
71:       case 2:
72:         pCondition = new FireAlarm;
73:        delete pCondition;
74:       break;
75:       default:
76:         pCondition = new Error;
77:         delete pCondition;
78:         okay = 0;
79:       break;
80:     }
81:   }
82:   return 0;
83: }
```

Tulostus

```
(0)Quit (1)Normal (2)Fire: 1
Logging normal conditions…
(0)Quit (1)Normal (2)Fire: 2

General Alarm log
Warning!
Logging fire call.
(0)Quit (1)Normal (2)Fire: 0
```

Analyysi

Rivien 59-80 silmukka sallii käyttäjän syötön, joka simuloi joko normaalia tai palosta varoittavaa, anturin antamaa ilmoitusta. Huomaa, että raportin tarkoituksena on luoda Condition-olio, jonka muodostin kutsuu eri jäsenfunktioita.

Virtuaalisten jäsenfunktioiden kutsuminen muodostimesta voi aiheuttaa sekaannuttavia tuloksia, jos et muista olioiden muodostamisjärjestystä. Esimerkiksi, kun FireAlarm-olio luodaan rivillä 72, on muodostamisjärjestys Condition, Alarm, FireAlarm. Alarm-muodostin kutsuu Log-funktiota, mutta siinä käytetään Alarm-olion Log()-funktiota, ei FireAlarm-olion Log-funktiota, vaikka Log() on esitelty virtuaalisena. Tämä johtuu siitä, että Alarm-muodostimen ajon aikana ei vielä ole FireAlarm-oliota. Myöhemmin, kun FireAlarm on itse muodostettu, sen muodostin kutsuu Log()-funktiota uudelleen ja tällä kertaa kutsutaan funktiota FireAlarm::Log().

Postimestari: tapaustutkimus

Seuraavassa esitellään toinen ongelma, jossa voimme harjoitella oliopohjaista analyysiä. Olet töissä Acme Software Inc. -yrityksessä aloittamassa uutta ohjelmistoprojektia ja menet C++ -tiimin toteuttamaan ohjelmaasi. Jim Grandiose on tuotekehityksen päällikkö ja sinun pomosi. Hän haluaa suunnitella ja rakentaa PostMaster-ohjelman, jolla luetaan

sähköpostia monesta erillisestä sähköpostisovelluksesta. Mahdollinen asiakas on liikemies, joka käyttää useampaa kuin yhtä email-ohjelmaa, esimerkiksi CompuServe, America Online, Internet Mail, Lotus Notes jne.

Asiakkaan tulee voida opettaa PostMaster-ohjelmalle, kuinka kuhunkin email-ohjelmaan kytkeydytään. Ohjelma ottaa sähköpostin ja esittää sen tietyllä tavalla sallien asiakkaan organisoida posti, lähettää kirjeet eteenpäin jne.

PostMaster Professional, joka jaetaan versiona 2, on jo suunniteltu. Siinä on mukana Administrative Assistant, joka sallii käyttäjän valtuuttaa toisen henkilön lukemaan osan postista tai koko postin, käsittelemään rutiiinikirjeenvaihdon jne. Markkinointiosasto haluaisi vielä lisätä ohjelmaan keinoälykomponentin, joka esilajittelee ja priorisoi postin aiheen ja sisällön avainsanojen sekä sanayhdistelmien mukaan.

Myös muista laajennuksista on keskusteltu: esimerkiksi mahdollisuudesta käsitellä postin lisäksi keskusteluryhmiä kuten Interchange-keskustelut, CompuServe-foorumit, Internet-keskusteluryhmät jne. On selvää, että Acmella on suuria toiveita PostMasterin suhteen ja se on tuotava markkinoille tiukan aikataulun mukaan, vaikkakaan budjettirajoituksia ei näytä olevan.

Mittaa kahdesti, leikkaa kerran

Laitat toimistosi kuntoon ja tilaat tarvikkeet. Ensimmäisenä virallisena työnäsi on nyt saada hyvät määrittelyt tuotteesta. Markkinoiden tutkimisen jälkeen päätät suositella kehittämistä aluksi yhteen ympäristöön ja otat mukaan UNIX-, Macintosh- ja Windows NT -ympäristöt.

Sinulla on monta tuskaisaa keskustelutilaisuutta Jim Grandiosen kanssa. Näyttää siltä, että mitään varmaa vaihtoehtoa ei ole ja päätät erottaa käyttöliittymän itse taustaohjelmista - tiedonsiirrosta ja tietokantaosasta. Saadaksesi asiat sujumaan nopeammin päätät aloittaa kehittämisen NT-ympäristöön ja myöhemmin UNIX- ja ehkä myös Mac-ympäristöihin.

Tuolla päätöksellä rajaat projektia voimakkaasti. Pian ilmenee, että tarvitset luokkakirjaston tai joukon kirjastoja muistin hallintaan, useisiin eri käyttöliittymiin ja ehkä myös tiedonsiirtoon ja tietokantakomponentteihin.

Herra Grandiose uskoo, että projektin onnistuminen on kiinni yhden henkilön selvästä visiosta, joten hän pyytää sinua laatimaan esiarkkitehtuurin ja suunnittelun ennen lisäohjelmoijien hankintaa. Alat analysoida ohjelmaa.

Hajota ja hallitse

Nyt sinulla on enemmän kuin yksi ongelma ratkaistavana. Jaatkin projektin seuraaviin merkittäviin osaprojekteihin:

- ❏ Tiedonsiirrot: ohjelmiston kyky soittaa email-sovellukseen modeemin kautta tai ottaa yhteys verkon kautta.
- ❏ Tietokanta: kyky tallentaa tietoa ja ladata tietoa levyltä.
- ❏ Email: kyky lukea erilaisia email-muotoja ja kirjoittaa uusia viestejä kuhunkin järjestelmään.
- ❏ Muokkaukset: ohjelmassa tulee olla kehittyneitä editoreja viestien lukemiseen ja muokkaamiseen.
- ❏ Alustakohtaiset seikat: useita erilaisia käyttöliittymiä kullekin alustalle.
- ❏ Laajennettavuus: kasvun ja laajennusten suunnittelu.
- ❏ Organisaatio ja aikataulutus: useiden kehittäjien ja heidän koodinsa välisten riippuvuuksien hallinta. Kunkin ryhmän tulee laatia ja julkaista aikataulunsa ja toimia niiden mukaan. Johdon ja markkinoinnin tulee tietää projektin kehittymisestä.

Päätät palkata johtohenkilön hoitamaan organisaation ja aikataulutuksen. Palkkaat sitten kokeneita kehittäjiä tukemaan analysointia ja suunnittelua ja hoitamaan jäljellejääneiden alueiden toteutuksen. Nämä kokeneet kehittäjät luovat seuraavat ryhmät:

- ❏ Tiedonsiirrot: Vastuussa sekä soittokytkentäisestä että verkkopohjaisesta tiedonsiirrosta. He käsittelevät paketteja, virtoja ja bittejä pikemminkin kuin itse sähköpostiviestejä.
- ❏ Viestimuodot: Vastuussa viestien muuntamisesta kustakin alkumuodosta tiettyyn yhteismuotoon (Postmaster-standardiin) ja takaisin. Tämän ryhmän tehtävänä on myös kirjoittaa nämä viestit levylle ja ladata niitä sieltä tarvittaessa.
- ❏ Viestieditorit: Tämä ryhmä on vastuussa tuotteen koko käyttöliittymästä kaikilla alustoilla. Heidän tehtävänään on taata, että edusta- ja taustaohjelmien välinen liittymä on tarpeeksi kapea, jottei tuotteen laajentaminen vaadi toistuvaa koodia.

Viestimuoto

Aiot kiinnittää huomion aluksi viestimuotoon jättäen sivuun tiedonsiirtoon ja käyttöliittymiin liittyvät seikat. Niitä käsitellään sitten, kun ymmärretään paremmin, mistä on kysymys. Ei ole juurikaan järkeä murehtia sitä, kuinka tieto esitetään käyttäjälle ennen kuin tiedetään, mistä tiedosta on kysymys.

Erilaisten email-muotojen tutkiminen paljastaa, että monet asiat ovat yhteisiä huolimatta useista eroista. Jokaisessa viestissä on alkuperätieto, kohdetieto sekä luomispäivämäärä. Lähes kaikissa viesteissä on otsikko tai aiherivi ja runko, jossa on tavallista tekstiä, muotoiltua tekstiä, grafiikkaa ja ehkä myös ääniä tai muita erikoislisäyksiä. Useimmat email-palvelut tukevat myös liitteitä, joiden avulla voidaan lähettää erikoistiedostoja sähköpostin mukana.

Päätät pysyä päätöksessäsi, että muutat jokaisen viestin PostMaster-muotoon. Sillä tavoin on tarvetta tehdä tallennus vain yhteen tietuemuotoon ja levynkäyttö yksinkertaistuu. Päätät myös erottaa otsikkotiedot (lähettäjä, vastaanottaja, päivämäärä, aihe, jne) viestin rungosta. Käyttäjä haluaa usein tutkia otsikkoluetteloa lukematta heti itse viestin sisältöä. Ehkäpä tulee aika, että käyttäjä haluaa ladata vain viestien otsikko-osat ja jättää itse viestin lukematta. Tässä PostMaster-versiossa saadaan kuitenkin aina koko viesti, vaikkakaan sitä ei ole pakko esittää käyttäjälle.

Alustava luokkien suunnittelu

Viestien tutkiminen johtaa Message-luokan suunnitteluun. Koska ohjelmaa tulee voida laajentaa käsittelemään muutakin kuin email-viestejä, johdat EmailMessage-luokan abstraktista Message-perusluokasta. EmailMessage-luokasta johdat sitten PostMasterMessage-, InterChangeMessage-, CISMessage-, ProdigyMessage-, yms. luokat.

Viestit ovat luonnollinen valinta kohteille ohjelmassa, jossa käsitellään sähköpostiviestejä, mutta oikeiden olioiden löytäminen monimutkaisessa järjestelmässä on suurimpia yksittäisiä oliopohjaisen ohjelmoinnin haasteita. Joissakin tapauksissa, kuten viestien kohdalla, ensisijaiset oliot näyttävät mukautuvan hyvin ongelmakenttään. Useimmiten on kuitenkin mietittävä pitkään ja hartaasti toteutettavia asioita jotta oikeat oliot löytyisivät.

Älä masennu. Kukaan ei ole seppä syntyessään. Yksi lähtökohta on kuvata ongelma kieliopin avulla. Tee luettelo kaikista substantiiveista ja verbeistä, joita projektin kuvauksessa käytetään. Substantiivit ovat hyviä olioehdokkaita. Verbit voivat olla noiden olioiden metodeja (tai ne voivat olla itse olioita). Tämä ei ole mikään idioottivarma menetelmä, mutta hyvä tekniikka aloittaa suunnittelu.

Siinä olikin helpoin osa kehitystä. Nyt esille nousee kysymyksiä: "Tulisiko viestin otsikon olla erillinen luokka suhteessa runkoon?" Jos niin on, tarvitaan rinnakkaiset hierarkiat - CompuServeBody ja CompuServeHeader sekä ProdigBody ja ProdigHeader?

Rinnakkaiset hierarkiat ovat usein huonon suunnittelun merkkejä. Oliopohjaisen suunnittelun yleinen virhe on laittaa osa olioista yhteen hierarkiaan ja vastaavat hallintaoliot toiseen hierarkiaan. Näiden hierarkioiden ylläpito ja yhteistyö voi olla hankalaa: oikea klassinen ylläpidon painajainen.

Mitään selviä sääntöjä ei kuitenkaan ole ja joskus rinnakkaiset hierarkiat ovat tehokkain keino ratkaista jokin ongelma. Kuitenkin, jos suunnittelusi näyttää menevän tuohon suuntaan, sinun tulisi miettiä ongelmaa uudelleen; hienompi ratkaisu voi olla löydettävissä.

Kun viestit saapuvat email-ohjelmasta, niiden otsikot ja rungot erotetaan toisistaan; moni viesti on yksi suuri tietovirta, jota ohjelman tulee tutkia. Ehkäpä hierarkiasi tulisi heijastua suoraan juuri tuosta ajatuksesta. Nyt voit luetella näiden viestien ominaisuuksia ajatellen toimintojen ja tietovarastojen esittämistä abstraktion oikealla tasolla. Olioiden ominaisuuksien luetteleminen on hyvä keino löytää tietojäsenet sekä havaita muut tarvittavat oliot.

Sähköpostiviestit tulee voida tallentaa, kuten myös käyttäjän tiedot, puhelinnumerot, jne. Tallennuksen tulee olla korkealla hierarkiassa. Tulisiko viestien välttämättä jakaa perusluokan ominaisuudet?

Juurelliset hierarkiat vastaan juurettomat hierarkiat

Periytymishierarkioihin liittyy kaksi yleistä lähestymistapaa: kaikki tai lähes kaikki luokat voivat periytyä jostakin yhteisestä juuriluokasta tai sitten periytymishierarkioita on enemmän kuin yksi. Yhteisen juuriluokan etuna on se, että voit usein välttää moniperiytyvyyden; haittana on se, että toteuttaminen on usein kiinni perusluokassa.

Uusi käsite Luokkajoukko on juurellinen, jos kaikilla luokilla on yhteinen esi-isä. Juurettomilla hierarkioilla ei ole yhteistä perusluokkaa.

Koska tiedät, että tuotteesi kehitetään monelle alustalle ja koska moniperiytyvyys on monimutkaista eivätkä kaikki kääntäjät tue sitä välttämättä hyvin kaikilla alustoilla, on ensimmäisenä päätöksenäsi käyttää juurellista hierarkiaa ja yksittäistä periyttämistä. Päätät identifioida ne kohdat, joissa moniperiytyvyyttä käytetään tulevaisuudessa ja tehdä suunnittelun niin, että hierarkian pirstominen ja moniperiytyvyyden lisääminen myöhemmin on mahdollista.

Päätät käyttää sisäisten luokkien nimessä etuliitettä p, jotta voit helposti ja nopeasti kertoa, mitkä luokat ovat omiasi ja mitkä tulevat muista kirjastoista.

Juuriluokkasi nimi on pObject. Kaikki luotavat luokat johdetaan tästä luokasta. Itse pObject on suppea; siinä on vain tiedot, joita jokainen luokka tarvitsee.

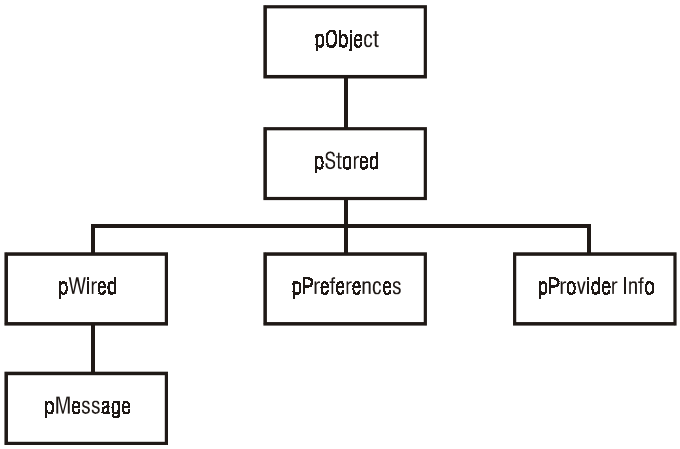
Yleensä juuriluokalle annetaan yleisluontoinen nimi (kuten pObject) ja vähän ominaisuuksia. Juuriluokan avulla luodaan luokkakokoelmia ja niihin viitataan pObject-luokan ilmentyminä. Huonona puolena on se, että juurelliset hierarkiat välittävät liittymänsä aina juuriluokkaan saakka. Se on hyväksyttävä. Muilla johdetuilla luokilla on liittymiä, jotka eivät niille sovellu. Ainoa hyvä ratkaisu tähän ongelmaan on mallien käyttäminen. Malleja käsitellään seuraavassa luvussa.

Seuraavaksi todennäköisimmät ehdokkaat hierarkian huipulle ovat pStored ja pWired. pStored-oliot tallennetaan levylle eri aikoina (esimerkiksi, kun ohjelma ei ole käytössä) ja pWired-oliot lähetetään modeemin tai verkon kautta. Koska melkein kaikki oliot tulee tallentaa levylle, on järkevää laittaa tämä toiminto hierarkian huipulle. Koska kaikki modeemilla lähetettävät oliot täytyy tallentaa, mutta kaikkia tallennettavia olioita ei lähetetä puhelinverkkoon, on järkevää johtaa pWired pStored-luokasta.

Jokainen johdettu luokka saa kaiken tiedon ja toiminnallisuuden perusluokaltaan ja kullakin tulisi olla yksi erillinen lisäominaisuus. Siten pWired-luokkaan pitänee lisätä erilaisia metodeja, mutta kaikkien noiden metodien on suunniteltu siirtävän tietoa modeemilla.

On mahdollista, että kaikki lähetettävät oliot tallennetaan tai että kaikki tallennettavat oliot lähetetään tai että kumpikaan määrittely ei ole tosi. Jos vain joitakin lähetettäviä olioita tallennetaan ja vain muutama tallennettava olio lähetetään, on pakko käyttää joko moniperiyttämistä tai muokata ongelmaratkaisua. Ongelman voisi ratkaista johtamalla esimerkiksi pWired pStored-luokasta ja poistaa sitten tallentavien metodien toiminnallisuus tai palauttaa virhe niistä olioista, jotka lähetetään modeemilla, mutta ei tallenneta.

Itse asiassa havaitsemme, että joitakin tallennettavia olioita kuten käyttäjäominaisuudet ei todellakaan lähetetä. Kaikki lähetettävät oliot kuitenkin tallennetaan, joten periytymishierarkia on kuvan 22.1 mukainen.



Kuva 22.1. Alustava hierarkia.

Liittymien suunnittelu

Tässä suunnittelun vaiheessa on tärkeää, ettei huomiota kiinnitetä toteuttamiseen. Kaikki energia on suunnattava luokkien välisten liittymien suunnitteluun ja sitten suunniteltava kaikki luokkien tarvitsemat jäsenet.

Usein on hyvä ymmärtää tarkasti perusluokat ennen johdettujen luokkien suunnittelua, joten päätät tutkiskella ensin luokkia pObject, pStored ja pWired.

pObject-juuriluokalla tulee olla vain ne tiedot ja metodit, jotka ovat yhteisiä koko järjestelmälle. Ehkäpä jokaisella oliolla tulisi olla uniikki tunnistenumero. Voisitkin luoda pID-numeron (PostMaster ID) ja tehdä siitä pObject-jäsenen; mutta ensin sinun on kysyttävä itseltäsi "Tarvitseeko jokainen olio, joka tallennetaan, mutta jota ei lähetetä modeemilla, sellaisen numeron?" Siitä poikii toinen kysymys "Onko olemassa olioita, joita ei tallenneta, mutta jotka ovat osa hierarkiaa?"

Jos sellaisia olioita on, saatat harkita pObject- ja pStored-luokkien yhdistämistä; kuitenkin, jos kaikki oliot tallennetaan, kuinka eriyttäminen tapahtuu? Kun ajattelet tätä tarkemmin, havaitset, että on joitakin olioita kuten osoitteet, jotka olisi edullista johtaa pObject-luokasta, mutta joita ei koskaan tallenneta yksinään; jos ne tallennetaan, ne tallennetaan osana muita olioita.

Havaitsetkin nyt, että erillinen pObject-luokka olisi hyödyllinen. Voit kuvitella, että käytössä on osoitekirja, joka olisi kokoelma pAddress-olioita ja kun yhtään pAddress-oliota ei tallenneta yksinään, olisi hyvä, jos jokaisella olisi oma uniikki ID-numeronsa. Sijoitat siis pID-jäsenen pObject-luokkaan; nyt pObject näyttäisi minimimuodossaan seuraavalta:

```
class pObject
{
public:
  pObject();
  ~pObject();
  pID GetID()const;
  void SetID();
private:
  pID itsID;
}
```

Luokan esittelyssä on useita seikkoja, jotka vaativat huomiota. Ensiksikin tätä luokkaa ei ole esitelty johdettavaksi jostakin muusta luokasta; se on juuriluokka. Toiseksi toteuttamista ei näytetä, vaikka esimerkiksi GetID()-metodin kohdalla on todennäköistä, että se määritellään inline-tyyppisenä.

Kolmanneksi on const-metodit jo identifioitu; se on osa liittymää, ei toteutusta. Lopuksi luokassa on mukana uusi tietotyyppi, pID. Määrittelemällä pID tyyppinä esimerkiksi tyypin unsigned long sijaan antaa suunnitteluun enemmän joustavuutta.

Jos osoittautuu, että unsigned long -tyyppiä ei tarvita tai että unsigned long ei ole tarpeeksi suuri, voit modifioida pID-määrittelyä. Tuo muutos vaikuttaa kaikkialle sinne, missä pID on käytössä eikä sinun tarvitse jäljittää kohtia ja muokata tiedostoja, joissa pID esiintyy.

Toistaiseksi käytät typedef-lausetta esittelemään pID ULONG-tyyppisenä, joka on esittelyn mukaan tyyppiä unsigned long. Tästä herää kysymys: minne nämä esittelyt kuuluvat?

Laajassa projektissa on tarvetta tiedostojen yleissuunnitelmalle. Yleinen lähestymistapa, jota tässäkin projektissa käytetään, on se, että kukin luokka sijoitetaan omaan otsikkotiedostoonsa ja luokan metodien toteuttaminen on vastaavassa .CPP-tiedostossa. Siten projektissa on tiedosto OBJECT.HPP ja OBJECT.CPP. Myöhemmin siihen tulee muita tiedostoja kuten MSG.HPP, jossa on luokan pMessage esittely ja MSG.CPP, jossa ovat luokan metodien määrittelyt.

Hankkia vai kehittää -päätös

Ostaa valmis vai luoda itse? Yksi kysymys on aina esillä ohjelmistojen suunnitteluvaiheessa: mitä aliohjelmia ja rutiineita ostetaan ja mitä kirjoitetaan itse. On täysin mahdollista hyödyntää olemassa olevia kaupallisia kirjastoja ratkaisemaan joitakin tai kaikki tiedonsiirron toiminnot. Lisenssimaksut ja muut ei-tekniset seikat tulevat tällöin ratkaistaviksi.

Usein sellainen kirjasto kannattaa hankkia ja kiinnittää huomio itse ohjelmaan kuin alkaa keksiä pyörää uudelleen. Saatat haluta ostaa myös kirjastoja, joita ei ole välttämättä tarkoitettu käytettäväksi C++ -kielen

kanssa, jos ne tarjoavat perustoiminnot, jotka joutuisit muutoin itse toteuttamaan. Se voikin tukea aikataulun pitämistä.

Prototyypin rakentaminen

Niinkin laajan projektin kuin PostMasterin kohdalla on epätodennäköistä, että alustava suunnitelma olisi kokonainen ja täydellinen. Jos projektin laajuutta ei ymmärretä ja yritetään luoda kaikki luokat ja toteuttaa niiden liittymät ennen ensimmäistäkään koodiriviä, voivat seuraukset olla tuhoisat.

Monestakin syystä kannattaa yrittää suunnitella ensin prototyyppi - nopeasti kasattu ja toimiva esimerkki ydinideoista. Prototyyppejä voidaan luoda useaan eri tarkoitukseen.

Käyttöliittymäprototyyppi antaa mahdollisuuden testauttaa tuotteen ulkoasua ja tuntumaa ohjelman käyttäjällä.

Toimintoprototyypissä ei ehkä ole lopullista käyttöliittymää, mutta sen avulla käyttäjät voivat kokeille eri piirteitä, kuten viestien lähettämistä tai tiedostojen liittämistä kiinnittämättä huomiota käyttöliittymään.

Voidaan kehittää myös arkkitehtuuriprototyyppi, joka olisi suppea versio ohjelmasta ja jonka avulla voisi saada tietoa päätöksentekoon ohjelman laajentuessa.

On tärkeää pitää protoilun tavoitteet selkeinä. Tutkitko nyt käyttöliittymää, toiminnallisuutta vai rakennatko suppean mallin lopullisesta tuotteesta? Hyvässä arkkitehtuuriprototyypissä on heikko käyttöliittymä ja kääntäen. On myös tärkeää, ettei prototyyppi ole liian laaja tai joudut miettimään protoilun kustannuksia ja mahdollisia negatiivisia vaikutuksia koko projektiin.

80/80-sääntö

Eräs hyvä peukalosääntö tässä vaiheessa on suunnitella ne asiat, joita 80 % ihmisistä haluaa tehdä 80 % ajastaan ja laittaa sivuun loput 20 %. Reunaehdot tulee asettaa ennemmin tai myöhemmin, mutta suunnittelun ydinseikat tulisi asettaa 80/80-säännön mukaan.

Päätätkin nyt aloittaa pääluokkien suunnittelun ja jättää toissijaisten luokkien suunnittelun sivuun. Edelleen, jos esille tulee useita luokkia, joiden rakenteet ovat pieniä eroja lukuunottamatta samanlaiset, voit poimia yhden esimerkkiluokan työn alle ja suunnitella lähiserkut myöhemmin.

Huom! On olemassa myös toinen yleissääntö, 80/20-sääntö, jonka mukaan "ensimmäiset 20 % ohjelmastasi vievät 80 % ajastasi; loput 80 % ohjelmasta vievät toisen 80 % ajastasi!"

PostMasterMessage-luokan suunnittelu

Edellä olevat ohjeet mielessäsi päätät aloittaa PostMasterMessage-luokan suunnittelun. Juuri tämä luokka on eniten suoran kontrollisi alaisuudessa.

Osana liittymäänsä PostMasterMessage-luokan on tietenkin keskusteltava muiden tyyppisten viestien kanssa. Toivot voivasi työskennellä läheisesti muiden viestitoimittajien kanssa ja saavasi heiltä viestimuotojen määritykset, mutta toistaiseksi teet joitakin arvailuja sen mukaan, mitä heidän palvelunsa tuottavat omalle koneellesi.

Joka tapauksessa tiedät, että jokaisessa PostMasterMessage-viestissä on lähettäjä, vastaanottaja, päivämäärä ja aihe sekä viestin runko ja ehkä liitetiedostoja. Tarvitset siis käsittelymetodit kullekin ominaisuudelle sekä metodit raportoimaan liitetiedostojen koot, viestien koot, jne.

Jotkut viestitoimittajat käyttävät muotoiltua tekstiä. Toiset taas eivät tue tätä formaattia ja ne, jotka tukevat saattavat käyttää tai olla käyttämättä omaa menettelyään muotoillun tekstin hallintaan. Luokallasi tulee olla muunnosmetodeita muuntamaan muotoiltu teksti puhtaaksi tekstiksi ja ehkäpä muuntamaan myös muita formaatteja PostMaster-muotoon.

API-liittymä

API (Application Programming Interface) on kokoelma dokumentaatiota ja rutiineita palvelun käyttämiseksi. Monet email-sovellustoimittajat antavat sinulle oman APInsa, jotta PostMaster voisi hyödyntää niiden kehittyneitä piirteitä kuten muotoiltu teksti ja upotetut tiedostot. Saatat haluta julkaista myös APIn PostMaster-ohjelmalle, jotta toiset toimittajat voivat suunnitella PostMasterin käyttöä tulevaisuudessa.

PostMasterMessage-luokallasi on oltava hyvin suunniteltu, julkinen liittymä ja muunnosfunktiot ovat PostMasterin APIn pääkomponentti. Listaus 22.2 esittää, miltä PostMasterMessagen liittymä näyttää toistaiseksi.

Listaus 22.2. PostMasterMessagen liittymä.

```
1:  class PostMasterMessage : public MailMessage
2:  {
3:  public:
4:    PostMasterMessage();
5:    PostMasterMessage(
6:    pAddress Sender,
7:    pAddress Recipient,
8:    pString Subject,
9:    pDate creationDate);
10:
11: // muut muodostimet
12: // muista kopiomuodostin
13: // sekä tallennuksen muodostin
```

```
14: // sekä formaattimuodostin
15: // Muista myös muiden formaattien tallennus
16:   ~PostMasterMessage();
17:   pAddress& GetSender() const;
18:   void SetSender(pAddress&);
19: // muut metodit
20:
21: // operaattorimetodit
22: // sekä muunnosmetodit
23: // viestien muuntamiseen.
24:
25: private:
26:   pAddress itsSender;
27:   pAddress itsRecipient;
28:   pString  itsSubject;
29:   pDate itsCreationDate;
30:   pDate itsLastModDate;
31:   pDate itsReceiptDate;
32:   pDate itsFirstReadDate;
33:   pDate itsLastReadDate;
34: };
```

Tulostus
Ei tulostusta.

Analyysi
PostMasterMessage-luokka johdetaan MailMessage-luokasta. Mukana on joukko muodostimia, joiden avulla voidaan luoda PostMasterMessage-viestejä muista viestityypeistä.

Luokassa on joukko metodeita jäsentietojen lukemiseen ja kirjoittamiseen sekä operaattoreita muuntamaan osa tai koko viesti toisiin muotoihin. Viestit olisi hyvä tallentaa levylle ja ne luetaan puhelinverkosta, joten mukana on metodeja myös näihin toimintoihin.

Suurissa ryhmissä ohjelmointi

Jo tämä esiarkkitehtuuri riittää osoittamaan, kuinka useat eri kehitysryhmät tulevat jatkamaan työtä. Tiedonsiirtoryhmä alkaa työstää tiedonsiirron taustaohjelmia jättäen kapean liittymän Viestimuoto-ryhmään.

Viestimuoto-ryhmä suunnittelee luultavasti yleisliittymän Message-luokille, mikä aloitettiin aiemmin ja kiinnittää sitten huomionsa siihen, kuinka tieto tallennetaan levylle ja kuinka se luetaan levyltä. Kun tämä levyliittymä tunnetaan hyvin, voi ryhmä alkaa suunnitella liittymää tiedonsiirtotasolle.

Viestieditori-ryhmää houkuttelee editorien luominen Message-luokan sisäisen käyttäytymisen pohjalta, mutta se olisi paha virhe. Myös tämän ryhmän on suunniteltava kapea liittymä Message-luokkaan; viestieditorin olioiden tulee tietää hyvin vähän viestien sisäisestä rakenteesta.

Harkintaa vaativia seikkoja

Projektin jatkuessa joudutaan toistuvasti seuraavan perusongelman eteen: mihin luokkaan tulisi sijoittaa tietty toiminnallisuus (tai tiedot)? Olisiko Message-luokalla tuo funktio vai Address-luokalla? Tulisiko editorin tallentaa informaatio vai tallentaako viesti sen itseensä?

Luokkien tulisi toimia välttämättömän tiedon perusteella kuin salaiset agentit. Niiden ei tulisi saada tietoa enempää kuin on välttämätöntä.

Suunnittelupäätökset

Ohjelman kehittyessä eteen tulee sadoittain suunnittelupulmia. Ne ulottuvat globaalisista kysymyksistä "Mitä haluamme tämän tekevän?" tarkempiin kysymyksiin "Kuinka saamme tämän toteutettua?"

Vaikkakin toteutuksen yksityiskohdat eivät tule viimeistellyiksi ennen koodin viimeistelyä ja jotkut liittymät muuttuvat työn aikana, on sinun varmistettava, että suunnitelma ymmärretään hyvin jo prosessin aikaisessa vaiheessa. On tärkeää, että ymmärrät, mitä aiot kehittää ennen koodin kirjoittamista. Yleisin yksittäinen syy ohjelmistoprojektin epäonnistumiseen on siinä, ettei ymmärretty riittävän hyvin ja tarpeeksi aikaisessa vaiheessa, mitä oltiin kehittämässä.

Päätöksiä, päätöksiä

Saadaksesi tuntumaa siitä, millainen suunnitteluprosessi on, tutki seuraavaa kysymystä: "Mitä valikossa tulee olemaan?" PostMaster-ohjelmassa on ensimmäinen kohta luultavasti New Mail Message ja tämä herättää heti uuden suunnitteluaiheen: kun käyttäjä valitsee New Mail Message, mitä tapahtuu? Luodaanko editori, joka taas luo viestin vai luodaanko uusi viesti, joka taas luo editorin?

Komento, jota työstät, on New Mail Message, joten uuden viestin luominen näyttäisi olevan se toiminto, joka toteutetaan. Mutta, mitä tapahtuu, jos käyttäjä klikkaa Cancel alettuaan kirjoittaa uutta viestiä? Olisikin ehkä parempi luoda ensin editori ja antaa sen luoda (ja omistaa) uusi viesti.

Tämän lähestymistavan ongelmana on se, että editorin on toimittava eri tavalla, jos se luo viestin kuin editoinnin aikana. Jos viesti taas luodaan ensin editorilla käsiteltäväksi, tarvitaan vain yksi koodijoukko, koska kaikki on vain olemassa olevan viestin editointia.

Jos viesti luodaan ensin, kuka luo sen? Luodaanko se valikkokomennolla? Jos niin tehdään, kertooko valikko siis viestille, että sen on muokattava itseään vai onko se osa viestin muodostimen toimintaa?

Ensi silmäyksellä tehtävä sopisi muodostimelle; jokaista viestiähän luultavasti aina editoidaan. Kuitenkaan kyseessä ei ole hyvä suunnitteluidea. Ensiksikin on hyvin mahdollista, että oletus on oikein: yhtä hyvin saatetaan luoda poikkeusviestejä (eli järjestelmäoperaattorille tarkoitettuja virheviestejä), joita ei sijoiteta editoriin. Toiseksi, mikä on vielä tärkeämpää, muodostimen työnä on luoda olio; sen ei tulisi tehdä mitään muuta. Kun viesti on luotu, on muodostimen tehtävä tehty; editorikutsun lisääminen sotkee muodostimen roolin ja tekee viestistä haavoittuvan johtuen editorivirheistä.

Ja vielä pahempaa, muokkausmetodi kutsuu toista luokkaa, editoria, aiheuttaen sen muodostimen kutsumisen. Editori ei ole viestin perusluokka eikä se sisälly viestiin, joten olisi onnetonta, jos viestin muodostuminen riippuisi editorin muodostimen onnistumisesta.

Lopuksi editoria ei kutsuta ollenkaan, jos viestiä ei voida luoda onnistuneesti. Ja onnistunut luonti riippuu tässä skenaariossa editorin kutsumisesta! On selvästi hyvä palata viestin muodostimesta ennen metodin Message::Edit() kutsumista.

Pääohjelman työstäminen

Eräs lähestymistapa päästä tutkimaan suunnittelun pulmia on luoda pääohjelma jo aikaisessa prosessin vaiheessa. Pääohjelma on toiminto, jonka tarkoituksena on demonstroida tai testata muita funktioita. Esimerkiksi PostMasterin pääohjelma on hyvin yksinkertainen valikko, joka luo PostMasterMessage-olioita, muokkaa niitä ja kokeilee muutoin joitakin suunnitelman osia.

Listaus 22.3 havainnollistaa hieman täydellisempää PostMasterMessage-luokan määrittelyä ja suppeaa pääohjelmaa.

Listaus 22.3. Testiohjelma PostMasterMessagelle.

```
1:  #include <iostream.h>
2:  #include <string.h>
3:
4:  typedef unsigned long pDate;
5:  enum SERVICE { PostMaster, Interchange, CompuServe, Prodigy, AOL, Internet };
6:
7:  class String
8:  {
9:  public:
10: // muodostimet
11:   String();
12:   String(const char *const);
13:   String(const String &);
14:   ~String();
15:
16:   // ylimääritellyt operaattorit
17:   char & operator[](int offset);
18:   char operator[](int offset) const;
19:   String operator+(const String&);
20:   void operator+=(const String&);
```

```
21:   String & operator= (const String &);
22:   friend ostream& operator<<( ostream& theStream,String& theString);
23:   // yleiset käsittelijät
24:   int GetLen()const { return itsLen; }
25:   const char * GetString() const { return itsString; }
26:   // static int ConstructorCount;
27:
28: private:
29:   String (int);
30:   char * itsString;
31:   int itsLen;
32:
33: };
34:
35:   // oletusmuodostin
36:   String::String()
37: {
38:   itsString = new char[1];
39:   itsString[0] = '\0';
40:   itsLen=0;
41:   // cout << "\tDefault string constructor\n";
42:   // ConstructorCount++;
43: }
44:
45: // tukimuodostin
46: // merkkijonojen muodostamiseen
47: // required size.  Null filled.
48: String::String(int len)
49: {
50: itsString = new char[len+1];
51: int i;
52: for ( i = 0; i<=len; i++)
53: itsString[1] = '\0';
54: itsLen=len;
55: // cout << "\tString(int) constructor\n";
56: // ConstructorCount++;
57: }
58:
59: // Converts a character array to a String
60: String::String(const char * const cString)
61: {
62: itsLen = strlen(cString);
63: itsString = new char[itsLen+1];
64: int i;
65: for ( i = 0; i<itsLen; i++)
66: itsString[i] = cString[i];
67: itsString[itsLen]='\0';
68: // cout << "\tString(char*) constructor\n";
69: // ConstructorCount++;
70: }
71:
72: // copy constructor
73: String::String (const String & rhs)
74: {
75: itsLen=rhs.GetLen();
76: itsString = new char[itsLen+1];
77: int i;
78: for (i = 0; i<itsLen;i++)
79: itsString[i] = rhs[i];
80: itsString[itsLen] = '\0';
81: // cout << "\tString(String&) constructor\n";
```

```
82: // ConstructorCount++;
83: }
84:
85: // destructor, frees allocated memory
86: String::~String ()
87: {
88: delete [] itsString;
89: itsLen = 0;
90: // cout << "\tString destructor\n";
91: }
92:
93: String& String::operator=(const String & rhs)
94: {
95: if (this == &rhs)
96: return *this;
97: delete [] itsString;
98: itsLen=rhs.GetLen();
99: itsString = new char[itsLen+1];
100: int i;
101: for (i = 0; i<itsLen;i++)
102: itsString[i] = rhs[i];
103: itsString[itsLen] = '\0';
104: return *this;
105: // cout << "\tString operator=\n";
106: }
107:
108: //ei-vakio operaattori
109: // palauttaa viittauksen merkkiin
110: char & String::operator[](int offset)
111: {
112:   if (offset > itsLen)
113:     return itsString[itsLen-1];
114:   else
115:     return itsString[offset];
116:   }
117:
118: // vakioindeksioperaattori
119: // toimii vakio-olioiden kanssa
120: char String::operator[](int offset) const
121: {
122:   if (offset > itsLen)
123:     return itsString[itsLen-1];
124:   else
125:     return itsString[offset];
126: }
127:
128: // luodaan +-operaattori
129: // kopioi
130: String String::operator+(const String& rhs)
131: {
132:   int  totalLen = itsLen + rhs.GetLen();
133:   String temp(totalLen);
134:   int i,j;
135:   for (i = 0; i<itsLen; i++)
136:     temp[i] = itsString[i];
137:   for (j = 0; j<rhs.GetLen(); j++, i++)
138:     temp[i] = rhs[j];
139:   temp[totalLen]='\0';
140:   return temp;
141: }
142:
```

```
143: // muuttaa merkijonoa
144: void String::operator+=(const String& rhs)
145: {
146:   int rhsLen = rhs.GetLen();
147:   inttotalLen = itsLen + rhsLen;
148:   String  temp(totalLen);
149:   int i,j;
150:   for ( i = 0; i<itsLen; i++)
151:     temp[i] = itsString[i];
152:   for ( j = 0; j<rhs.GetLen(); j++, i++)
153:     temp[i] = rhs[i-itsLen];
154:   temp[totalLen]='\0';
155:   *this = temp;
156: }
157:
158: // int String::ConstructorCount = 0;
159:
160: ostream& operator<<( ostream& theStream,String& theString)
161: {
162:   theStream << theString.GetString();
163:   return theStream;
164: }
165:
166: class pAddress
167: {
168: public:
169:   pAddress(SERVICE theService,
170:   const String& theAddress,
171:   const String& theDisplay):
172:   itsService(theService),
173:   itsAddressString(theAddress),
174:   itsDisplayString(theDisplay)
175: {}
176:   // pAddress(String, String);
177:   // pAddress();
178:   // pAddress (const pAddress&);
179:   ~pAddress(){}
180:   friend ostream& operator<<( ostream& theStream, pAddress& theAddress);
181:   String& GetDisplayString() { return itsDisplayString; }
182: private:
183:   SERVICE itsService;
184:   String itsAddressString;
185:   String itsDisplayString;
186: };
187:
188: ostream& operator<<( ostream& theStream, pAddress& theAddress)
189: {
190:   theStream << theAddress.GetDisplayString();
191:   return theStream;
192: }
193:
194: class PostMasterMessage
195: {
196: public:
197:   // PostMasterMessage();
198:
199:   PostMasterMessage(const pAddress& Sender,
200:   const pAddress& Recipient,
201:   const String& Subject,
202:   const pDate& creationDate);
203:
```

```
204:   ~PostMasterMessage(){}
205:
206:   void Edit(); // editori mukaan
207:
208:   pAddress& GetSender() const { return itsSender; }
209:   pAddress& GetRecipient() const { return itsRecipient; }
210:   String& GetSubject() const { return itsSubject; }
211:   //  void SetSender(pAddress& );
212:   // muut metodit
213:
214: // operaattorimetodit tänne
215: // sekä muunnos-
216: // metodit
217:
218: private:
219:   pAddress itsSender;
220:   pAddress itsRecipient;
221:   String  itsSubject;
222:   pDate itsCreationDate;
223:   pDate itsLastModDate;
224:   pDate itsReceiptDate;
225:   pDate itsFirstReadDate;
226:   pDate itsLastReadDate;
227: };
228:
229: PostMasterMessage::PostMasterMessage(
230:   const pAddress& Sender,
231:   const pAddress& Recipient,
232:   const String& Subject,
233:   const pDate& creationDate):
234:   itsSender(Sender),
235:   itsRecipient(Recipient),
236:   itsSubject(Subject),
237:   itsCreationDate(creationDate),
238:   itsLastModDate(creationDate),
239:   itsFirstReadDate(0),
240:   itsLastReadDate(0)
241: {
242:   cout << "Post Master Message created. \n";
243: }
244:
245: void PostMasterMessage::Edit()
246: {
247:   cout << "PostMasterMessage edit function called\n";
248: }
249:
250:
251: int main()
252: {
253:   pAddress Sender(PostMaster, "jliberty@PostMaster", "Jesse Liberty");
254:   pAddress Recipient(PostMaster, "sliberty@PostMaster","Stacey Liberty");
255:   PostMasterMessage PostMasterMessage(Sender, Recipient, "Saying Hello", 0);
256:   cout << "Message review... \n";
257:   cout << "From:\t\t" << PostMasterMessage.GetSender() << endl;
258:   cout << "To:\t\t" << PostMasterMessage.GetRecipient() << endl;
259:   cout << "Subject:\t" << PostMasterMessage.GetSubject() << endl;
260:   return 0;
261: }
```

Jos saat virheilmoituksen muuntamisen epäonnistumisesta, poista const-sanat riveiltä 207-209.

Tulostus

```
Post Master Message created.
Message review…
From:   Jesse Liberty
To: Stacey Liberty
Subject:    Saying Hello
```

Analyysi

Rivillä 4 esitellään pDate unsigned long -tyyppiseksi. Ei ole lainkaan epätavallista, että päivämäärät tallennetaan long-tyyppisinä, koska sekuntien määrä lasketaan jostakin aloituspäivämäärästä kuten 1.1.1900. Tässä ohjelmassa kyseessä on paikanpitäjä, pDate on hyvä muuttaa todelliseksi luokaksi.

Rivillä 5 on määritelty lueteltu vakio, SERVICE, jolla osoiteoliot seuraavat osoitetyyppiään, joita ovat PostMaster, CompuServe, jne.

Rivit 7-163 edustavat liittymää String-luokkaan sekä String-toteutusta. Rivit ovat paljolti samanlaisia kuin aiempien lukujen listauksissa. String-luokkaa käytetään useiden jäsenmuuttujien kohdalla kaikissa viestiluokissa ja lukuisissa muissa viestien käyttämissä luokissa ja se onkin tärkeällä paikalla ohjelmassa. Kokonainen ja ehyt String-luokka on olennainen osa täydentämään viestiluokkia.

Riveillä 165-185 esitellään pAddress-luokka. Mukana on vain luokan perustoiminnallisuus ja määrittelyä tulisi täydentää, kun ohjelma tunnetaan paremmin. Seuraavat kohteet ovat olennaisia osia jokaisessa viestissä: lähettäjän ja vastaanottajan osoite. Täysin toimivan pAddress-olion tulee osata lähettää viesti eteenpäin, vastata, jne.

pAddress-olion tehtävänä on valvoa sekä esitettävää merkkijonoa että sisäistä reititysmerkkijonoa palvelulleen. Eräs suunnittelun avoin kysymys on se, tulisiko käytössä olla yksi pAddress-olio vai tulisiko se jakaa aliluokkiin kullekin palvelutyypille. Toistaiseksi palveluun viittaa lueteltu vakio, joka on pAddress-olion jäsenmuuttuja.

Rivit 193-226 sisältävät PostMasterMessage-luokan liittymän. Tässä listauksessa tämä luokka on erillään, mutta siitä tulisi piakkoin tehdä osa hierarkiaa. Kun luokka suunnitellaan uudelleen periytymään Message-luokasta, siirtyvät jotkut jäsenmuuttujat perusluokkaan ja jotkut jäsenfunktiot joudutaan korvaamaan.

Tarvitaan vielä muita muodostimia, käsittelyfunktioita ja muita jäsenfunktioita, jotta luokka olisi täysin toimiva. Huomaa (kuten listaus osoittaa), että luokan ei tarvitse olla täydellinen, jotta voisit kirjoittaa testiohjelman joidenkin olettamusten testaamiseen.

Riveillä 244-247 on Edit()-funktio, joka sisältää tarvittavan määrän ominaisuuksia, jotta se voisi osoittaa, mihin muokkaustoiminto sijoitetaan tämän luokan ollessa täysin toimiva.

Rivit 250-260 sisältävät testiohjelman. Nyt ohjelma ei tee muuta kuin kokeilee muutamia käsittelyfunktioita ja operator<< ylikuormitusta. Kuitenkin se antaa lähtökohdan PostMasterMessage-viestien tutkimiseen ja kehyksen, jossa näitä luokkia voidaan muokata ja tutkia sitten muutosten vaikutuksia.

Yhteenveto

Tässä luvussa näit, kuinka C++ -kielen monia ominaisuuksia sovelletaan oliopohjaisessa analyysissä, suunnittelussa ja ohjelmoinnissa. Kehitysjakso ei ole lineaarista edistymistä puhtaasta analyysistä suunnitteluun ja sieltä ohjelmointiin; pikemminkin se on vaiheittaista. Ensimmäisenä vaiheena on ongelman analysointi, jonka tulokset muodostavat alustavan suunnittelun pohjan.

Kun alustava suunnittelu on tehty, ohjelmointi voi alkaa, mutta ohjelmoinnin aikana opitut asiat syötetään takaisin analyysiin ja suunnitteluun. Kun ohjelmointi edistyy, alkaa testaaminen ja vianhaku. Iterointi jatkuu eikä pääty koskaan, vaikkakin joitakin erillisiä tavoitteita saavutetaan. Tulee kuitenkin aika, kun tuote on valmis jakeluun. Älä epäröi, laita menemään!

Laajaa ongelmaa analysoitaessa oliopohjaisesta näkökulmasta ongelman toisiinsa vaikuttavista osista tulee alustavan suunnittelun kohteita (olioita). Suunnittelija toivoo voivansa kapseloida erilliset toiminnot olioihin aina kun mahdollista.

Luokkahierarkia on suunniteltava ja toteutettava toisiinsa vuorovaikutuksessa olevien osien perustavat suhteet. Alustavaa suunnittelua ei ole tarkoitettu lopulliseksi ja toiminnallisuutta on lisättävä olioihin suunnittelun tarkentuessa.

Oliopohjaisen analyysin perustavoite on kätkeä niin paljon kuin mahdollista tiedosta ja toteutuksesta ja muodostaa itsenäisiä olioita, joilla on kapea ja hyvin suunniteltu liittymä. Olion asiakkaiden ei tule ymmärtää toteutuksen yksityiskohtia, joilla oliot täyttävät velvollisuutensa.

Kysymyksiä ja Vastauksia

K
Millä tavoin oliopohjainen analyysi ja suunnittelu eroavat pääpiirteissään muista lähestymistavoista?
V
Ennen näitä oliopohjaisia tekniikoita ohjelmoijilla oli taipumus ajatella ohjelmaa funktioina, jotka käsittelivät tietoa. Oliopohjaisen ohjelmoinnin ytimenä on yhdistää tiedot ja toiminnallisuus erillisiin yksiköihin, joilla on siis sekä tietämys (tieto) että kyvyt (funktiot). Proseduraalisissa ohjelmissa on toisaalta painopiste funktiossa ja siinä, kuinka ne käsittelevät tietoa. On sanottu, että Pascal- ja C-ohjelmat ovat proseduurikokoelmia ja C++ -ohjelmat taas luokkakokoelmia.

K
Onko oliopohjainen ohjelmointi todellakin se hopealuoti, joka ratkaisee kaikki ohjelmointiongelmat?
V
Ei, eikä niin ollut tarkoituskaan. Laajojen, monimutkaisten ongelmien kohdalla voi oliopohjainen analyysi, suunnittelu ja ohjelmointi tarjota ohjelmoijalle työkalut selvitä suunnattomasta monimutkaisuudesta tavoilla, joita ei aiemmin voitu käyttää.