Luku 8

Lisää rajapinnoista

"There are things known and things unknown and in between are The Doors."

– Jim Morrison [The Doors FAQ, 2002]

Rajapinnan tehtävä on kertoa siihen liittyvästä komponentista (moduuli tai olio) käyttäjälle vastaus kysymykseen "Miten tämän on tarkoitus toimia?". Kun käyttäjä tietää vastauksen tähän kysymykseen, hän pystyy hyödyntämään rajapinnan määrittelemää palvelua välittämättä sen taakse kapseloidusta toteutuksesta. Samalla komponentilla voi olla useita erilaisia rajapintoja. Oliolla voi olla rajapinta (tai sen osa) erikseen vakio-olioille, "tavallisille" instansseille ja aliluokille.

Rajapinnan toteuttajan pitäisi pitää mielessään lause "Tekeekö toteutus *täsmälleen* sen, mitä rajapinnan määrittely sanoo?". Taito ja kokemus ovat tässä työssä korvaamattomia. Ohjelmointikieli sekä suunnittelu- ja tyylisäännöt voivat ainakin ohjata kohti oikein toimivaa lopputulosta.

8.1 Sopimus rajapinnasta

Sopimussuunnittelu (*Design By Contract*, [Meyer, 1997]) on rajapintasuunnittelun ja -toteutuksen menetelmä, jossa palveluiden ominaisuuksia kuvataan matematiikan keinoin. Lakimiesten viilaaman kir-

jallisen kahden osapuolen sopimuksen tapaan sopimussuunnittelussa ajatellaan syntyvän rajapinnan käyttäjän ja sen toteutuksen välille sopimus, jossa kummallakin osapuolella on tarkkaan määritellyt vastuunsa:

- **Rajapinnan toteutus** lupaa jokaisen rajapinnan palvelun osalta toimia tietyllä tavalla, *kun rajapintaa käytetään sovitulla tavalla*. Määrittelyssä on siis mukana tieto siitä, mitkä ovat rajapinnan sallittuja käyttötapoja (funktioiden parametrit, kutsujärjestykset, tietotyyppien arvorajat yms.).
- **Rajapintaa käyttävä ohjelmoija** lupaa käyttää palveluita *ainoastaan* niiden määrittelyn mukaisesti.

Sopimussuunnittelu on nimensä mukaisesti ennen kaikkea rajapintojen suunnittelua auttava ajattelutapa, joka pakottaa ja ohjaa sekä miettimään että dokumentoimaan rajapinnan huolellisesti. Toissijaisena hyötynä menetelmä yksinkertaistaa komponenttien toteuttamista. Tämä saavutetaan siten, että rajapinnan toteutuksen ulkopuolelle on sopimuksessa rajattu hankalat ja ei-toivotut käyttötavat, jolloin niihin ei tarvitse toteutuksessa varautua. Voidaan ajatella että "villin lännen" rajapintaa saa käyttää miten huvittaa ja toteutuksen pitäisi osata toimia kaikissa tilanteissa jotenkin "oikein" (minimissään laadukas koodi ei ainakaan kaadu kummallisissa tilanteissa). Sopimussuunnittelussa jaetaan vastuuta toiminnasta kutsujan ja toteutuksen kesken, jolloin kokonaisohjelmakoodimäärä on yleensä pienempi kuin kaikkeen varautuneissa toteutuksissa.

8.1.1 Palveluiden esi- ja jälkiehdot

Rajapinnan yksittäisen palvelun sopimus tehdään määrittelemällä jokaiselle palvelulle **esiehto** (*precondition*, P) ja **jälkiehto** (*postcondition*, Q). Molemmat ovat (predikaattilogiikan) loogisia lausekkeita, joista esiehdon tulee olla totta ennen palvelun käynnistämistä ja jälkiehdon palvelun suorittamisen jälkeen:

$$\{P\} \quad \texttt{palvelu()} \quad \{Q\}$$

Tämän "oikeellisuuskaavan" (*correctness formula*) mukaisesti palvelu lupaa, että kaikki funktion `palvelu` suoritukset, joiden alkaessa ehto P on totta, päättyvät tilaan, jossa ehto Q on totta. Jos ehto P ei

toteudu, niin palvelu toimii määrittelemättömästi ja jälkiehdon ei tarvitse toteutua (haluttua palvelua ei saada).

Yksinkertaisimmillaan ehdoilla voidaan rajata parametrien arvoalueita:

$$\{\ \mathrm{p} >= \text{kirja.lainauspäivä}\ \}$$
$$\texttt{kirja.palauta(palautuspäivä p)}$$
$$\{\ \text{kirja.tila} = \text{PALAUTETTU}\ \}$$

Käytettäessä predikaattilogiikan ominaisuuksia voidaan kertoa esimerkiksi rutiinista, joka järjestää taulukon, joka ei sisällä yhtäkään tyhjää alkiota:

$$\{\forall\, i \mid 1 \leq i \leq n : \mathrm{ARRAY}[i] \neq \text{TYHJÄ}\}$$
$$\texttt{qsort(ARRAY)}$$
$$\{\forall\, i \mid 1 \leq i < n : \mathrm{ARRAY}[i] \leq \mathrm{ARRAY}[i+1]\}$$

Sopimussuunnittelun tärkein ominaisuus on palvelun kutsujan ja toteuttajan vastuiden määrittely — samalla suunnittelun suurin vaikeus on päättää, mitkä ominaisuudet kuuluvat vastuurajan millekin puolelle.

- **Kutsuja.** Palvelun kutsujan vastuulla on pitää huoli siitä, että esiehto toteutuu. Jos kirja yritetään palauttaa päivämäärällä joka on ennen kirjattua lainauspäivää, niin palvelun mukainen sopimus ei ole voimassa, ja suoritus voi tehdä mitä tahansa. Ohjelman *testausvaiheessa* voidaan esiehtoja tarkastaa, mutta päämääränä on, että näitä tarkastuksia ei ole enää lopullisessa ohjelmistossa mukana (huolellinen testaus on löytänyt esiehdon rikkovat palvelun kutsut).

- **Toteuttaja.** Palvelun toteuttajan vastuulla on pitää huoli siitä, että palvelun suorituksen jälkeen jälkiehto on aina voimassa (kunhan esiehto on ollut voimassa). Jos sopimuksessa on määritelty rutiinin parametripäiväyksen olevan jotain päivämäärää myöhemmän, niin palvelun toteutuksessa ***ei enää tehdä tarkastusta***, joka on määritelty kutsujan vastuulle. Jos palvelu ei pysty toteuttamaan sopimuksen mukaista palvelua (jälkiehto ei täyty) kyseessä on virhetilanne, joka on jollain tavalla ilmaistava. Tämä tehdään yleensä poikkeusten (luku 11) avulla.

Yleiskäyttöisten komponenttien rajapinnoissa on usein vaikea päättää, minkälainen sopimus halutaan muodostaa kutsujan ja toteutuksen välille. Joskus voi olla tarkoituksenmukaista tarjota samasta perustoiminnallisuudesta sopimukseltaan erilaisia versioita.

Yksi esimerkki tällaisesta toiminnasta on `vector`-taulukon indeksointi, josta löytyy kaksi versiota. Operaatio `at()` hyväksyy parametrinaan minkä tahansa kokonaisluvun ja tarkistaa toteutuksessaan osuuko tämä luku indeksinä tulkittuna taulukon sisälle. Jos indeksi on kelvollinen, niin kyseessä oleva alkio palautetaan, muutoin kutsujalle kerrotaan virheestä (poikkeusmekanismilla). Toinen versio indeksoinnista on hakasulkuoperaattori (`v[i]`), joka ei suorita indeksin laillisuustarkistusta. Sopimussuunnittelun kannalta tämä versio on määritellyt kutsujan vastuulle (esiehdoksi), että operaatiota kutsutaan ainoastaan laillisilla taulukon indekseillä. Jos kutsuja ei täytä esiehtoa, niin toteutus tekee määrittelemättömän operaation (joka ohjelmassa näkyy esimerkiksi sen kaatumisena tai muistin roskaantumisena).

8.1.2 Luokkainvariantti

Luokkien tapauksessa voidaan palveluiden esi- ja jälkiehtojen lisäksi määritellä pysyväisväittämä eli **invariantti** (*invariant*), joka kertoo luokan olion ylläpitämän ehdon palvelukutsujen välillä. Esimerkiksi:

```
class KirjastonKirja {
  // Invariantti: sijaintitieto on aina
  // LAINASSA, PAIKALLA, HUOLTO tai POISTETTU
...
```

Esimerkin `KirjastonKirja`-luokan olioiden tilan tiedetään olevan aina invariantin mukainen silloin, kun ohjelman suoritus ei ole keskellä jotain luokan tarjoamaa palvelua.

Luokkainvariantin on oltava voimassa heti olion synnyttyä. Jos oliota luotaessa kutsutaan alustusrutiinia (kuten `C++`:n rakentajajäsenfunktio), invariantin on oltava voimassa, kun tämä rutiini on lopettanut suorituksensa. Tämän jälkeen invariantin on oltava voimassa aina ennen ja jälkeen jokaista julkisen rajapinnan rajapintafunktion kutsua:

$\{\text{LUOKKAINVARIANTTI} \wedge P\}$ `olio.palvelu()` $\{\text{LUOKKAINVARIANTTI} \wedge Q\}$

Kun suoritus on jäsenfunktion koodissa olion “sisällä”, puhutaan epästabiilista tilasta, jossa luokkainvariantti saa väliaikisesti olla epätosi. Tämän säännön noudattaminen ja tarkastaminen tulee hankalaksi silloin kun jäsenfunktio kutsuu toisia jäsenfunktioita osana omaa toiminnallisuuttaan.

8.1.3 Sopimussuunnittelun käyttö

Yksinkertaisissa esimerkeissä kauniilta näyttävä sopimussuunnittelu ei valitettavasti skaalaudu varsinkaan ohjelmiston ylimmän tason moduulien rajapintoihin. Esi- ja jälkiehtojen tarkkaan määrittelyyn kuuluu usein niin paljon informaatiota, että sen kuvaaminen matemaattisen tarkasti on ainakin tällä hetkellä liian työlästä. Formaaleihin määrittely- ja suunnittelumenetelmiin liittyvää tutkimusta tehdään paljon, ja sitä sovelletaan useissa erityisen suurta varmuutta vaativissa ohjelmistoprojekteissa. Tavallisimmissa ohjelmistoprojekteissa matemaattista määrittelyä ei kuitenkaan yleensä juuri käytetä. Sopimussuunnittelun periaatteiden tunteminen ja jopa osittainenkin noudattaminen on tietysti pelkkää sanallista rajapintakuvausta eksaktimpi menetelmä.

Olio-ohjelmointi aiheuttaa myös omat hankaluutensa sopimussuunnitteluun. Periytymisen yhteydessä rajapinnan sopimuksen pitäisi usein osata sanoa jotain myös aliluokan rajapinnasta, joka usein on hyvin hankalaa. Joskus kantaluokka ei tiedä edes karkeasti minkälaisia versioita siitä on tarkoitus periyttää, jolloin liian tiukasti määritellyt rajapintasopimukset voivat pahimmillaan tehdä periytettyjen versioiden tekemisen mahdottomiksi siten, että kantaluokan rajapintasopimus pidetään voimassa.

8.1.4 C++: luokan sopimusten tarkastus

Ohjelmiston kehitysvaiheessa on hyödyllistä tarkastaa, että rajapinnoille määriteltyjä sopimuksia noudatetaan. Väärinymmärryksistä tai huonosta dokumentoinnista johtuvat sopimuksen vastaiset kutsut saadaan heti näkyville ohjelmiston testausvaiheessa. Ylimääräiset testit hidastavat ohjelman toimintaa, mutta testausvaiheessa sillä ei useinkaan ole merkitystä. Poikkeuksena ovat reaaliaikavaatimuksia omaavat ohjelmistot, joissa ei voida käyttää testeissäkään ylimää-

räisiä tarkastuksia, jos niiden aiheuttama ylimääräinen prosessointi vaikuttaa ohjelmiston ajoituksiin.

assert-makro

Ohjelmissa olevilla sopimustarkastuksilla on pitkät perinteet jo ajalta ennen olio-ohjelmoinnin yleistymistä. C-kielessä on määriteltynä **varmistusrutiini** `assert` [Kerninghan ja Ritchie, 1988], joka ilmoittaa virheestä ja pysäyttää ohjelman suorituksen, jos sen parametrina oleva lauseke on arvoltaan epätosi. Kun ohjelmiston julkaisuversio käännetään siten, että esikääntäjäsymboli `NDEBUG` on määriteltynä, `assert`-makron arvo asettuu tyhjäksi, jolloin nämä ***kehitysvaiheen tarkastukset eivät ole mukana lopullisessa ohjelmistossa***. Tämä käytäntö (assert-testit ovat mukana vain ohjelman testausvaiheessa) noudattaa sopimussuunnittelun periaatetta, jonka mukaan oikein toimivassa ohjelmassa sekä rajapinnan kutsu että toteutus noudattavat *aina* sopimusta.

Koska `assert` on määritelty makroksi, se *ei* ole `C++`:n `std`-nimiavaruuden sisällä. Rajapintafunktion alussa voidaan testauksessa varmistaa kutsujan noudattavan sopimusta funktion parametreista:

```
#include <cassert>
     ⋮
int palvelu(int a, int b)
{
  assert( a < 2 && b > 40 );
     ⋮
```

Funktion väärä käyttö voi näkyä testaajalle esimerkiksi seuraavassa muodossa:

```
assertkoe.cc:4: failed assertion 'a < 2 && b > 40'
(program aborted)
```

Tässä `assertkoe.cc` on lähdekooditiedoston nimi. `C++`-standardi jättää tarkoituksella määrittelemättä tulostettavan virheen muodon. Jossain ympäristössä tällainen virheilmoitus voidaan ilmaista esimerkiksi käyttöliittymän virheikkunalla.

Koska `assert`-tarkastukset eivät ole mukana lopullisessa ohjelmakoodissa, on oltava tarkkana, että käytetty ***varmistusehto ei sisällä***

ohjelman toiminnallisuutta:

```
1   Paivays* pNyt = 0;
2   assert(pNyt = Paivays::NytOsoitin()); // Sijoitus assertissa!
3   pNyt->Tulosta();
```

Vaikka `assert`-makron parametrina oleva sijoitus (rivi 2) saakin C++:ssa arvon, joka tulkitaan myös totuusarvoksi, kyseessä on silti väärä tapa käyttää varmistusrutiinia.[†] Edellinen koodinpätkä voi toimia täysin oikein ohjelmiston testausvaiheessa, mutta kun lopullisessa versiossa `assert`-makro määritellään tyhjäksi, rivin 2 sijoitus jää kokonaan suorittamatta ja koodin toiminta muuttuu.

C-kielen `assert` on määritelty siten, että ohjelman suoritus keskeytetään kutsumalla kielen funktiota `abort`. Tätä funktiota ei pidetä C++:ssä suositeltavana, koska sitä kutsuttaessa ei suoriteta mitään lopetustoimenpiteitä. Erityisesti ohjelmassa kutsuhetkellä olevien olioiden purkajat jäävät suorittamatta (niissä voi olla resurssien vapautukseen liittyviä toimintoja). C++:ssä suositeltavampi tapa on käyttää poikkeusmekanismia (luku 11) myös "assertiovirheen" ilmaisemiseen. Listauksessa 8.1 seuraavalla sivulla on esimerkki poikkeuksilla toteutetusta `Assert`-makrosta C++:lla.

Luokkainvariantti

Koska luokkainvariantin tarkastuksessa on tarkoituksena tarkastaa luokan vastuualueen sopimus jokaisen rajapintafunktion suorituksen jälkeen, kannattaa tämä tarkastus kirjoittaa omaksi jäsenfunktiokseen, jota kutsutaan aina muiden jäsenfunktioiden alussa ja lopussa. (Myös alussa, jotta voidaan varmistua siitä, että invariantin määräämä sopimus on voimassa myös rutiinin suorituksen alkaessa, katso ohjelmalistaus 8.2 sivulla 248).

Tässä esitetty suoraviivainen toteutus ei huomioi mitenkään sitä tilannetta, että invariantti saa olla epätosi jos tarkistuksen kohteena olevaa jäsenfunktiota on kutsuttu toisesta (saman olion) jäsenfunktiosta. Tällainen kutsuketjun seuraaminen mutkistaisi ohjelmakoodia jonkin verran, ja sen toteuttamiseen ei C++:ssä valitettavasti ole valmiita apukeinoja.

[†] Hyvin tyypillinen virhe C++:ssa on käyttää yhtäsuuruutta tarkoitettaessa kielen (huonosti valittua) sijoitusmuotoa [`assert( x = 7 )` vs. `assert( x == 7 )`].

```
1  #ifndef ASSERT_HH
2  #define ASSERT_HH
3
4  #include "varmistuspieleen.hh"          /* poikkeusluokan esittely */
5  #include <iostream>
6
7  #ifdef NDEBUG
8  #define Assert(x) /* tyhja */
9  #else
10 #define Assert(x) Assert_toteutus( x, #x, __FILE__, __LINE__ )
11 #endif
12
13 inline void Assert_toteutus( bool varmistus, char const* lauseke,
14                                  char const* tiedosto,
15                                  unsigned int rivinumero )
16 {
17   if( varmistus == false ) {
18     std::cerr << tiedosto << ":" << rivinumero << ":";
19     std::cerr << "Varmistus epäonnistui: " << lauseke << std::endl;
20     throw VarmistusPieleen(lauseke, tiedosto, rivinumero);
21   }
22 }
23
24 #endif /* ASSERT_HH */
```

LISTAUS 8.1: Varmistusrutiinin toteutus

Koska C++ ei automaattisesti tue invarianttien tarkastusta, niiden käyttö on täysin ohjelmoijan vastuulla. Tämä tulee ottaa huomioon myös periytymishierarkioissa, joissa on pidettävä huoli siitä, että aliluokan jäsenfunktiot tarkastavat myös kantaluokan invariantin säilymisen:

```
// Kun järjestetty taulukko on periytetty luokasta Taulukko
inline void JärjestettyTaulukko::Invariantti()
{
#ifndef NDEBUG
  Taulukko::Invariantti();
     ⋮
#endif
}
```

```
1  inline void JarjestettyTaulukko::Invariantti()
2  {
3  #ifndef NDEBUG
4     // Invariantti: alkiot ovat aina suuruusjärjestyksessä, siten että
5     // indeksissä 1 on pienin alkio ja indeksissä KOKO suurin.
6     for( int i = 1; i < KOKO; i++ )
7     {
8        if( alkio[ i ] > alkio[ i+1 ] )
9           throw JarjestettyTaulukko::InvarianttiRikottu();
10    }
11 #endif
12 }
13
14 void JarjestettyTaulukko::EtsiJaMuutaAlkio(
15    Alkio const& etsittava, Alkio const& korvaava )
16 {
17    Invariantti();
18    // Jäsenfunktion toteutus
19    Invariantti();
20 }
```

LISTAUS 8.2: Esimerkki luokkainvariantin toteutuksesta jäsenfunktiona

8.2 Luokan rajapinta ja olion rajapinta

Luokkasuunnittelussa tulee silloin tällöin vastaan tilanne, jossa jokin luokan vastuualueeseen kuuluva asia ei oikeastaan kuulu minkään luokan olion tehtäviin vaan ikään kuin “koko luokalle”, toisaalta kaikille, toisaalta ei millekään oliolle. Esimerkkejä tällaisista luokan “yhteisistä” asioista ovat esim. päiväysluokalla taulukko, jossa pidetään muistissa kuukausien pituudet, tai merkkijonoluokalla tieto siitä, mikä on merkkijonojen maksimipituus. Samoin ohjelman testausvaiheessa saattaisi olla mukavaa pitää kirjaa siitä, montako jonkin luokan oliota ohjelman ajon aikana on luotu, sekä tarjota funktio, jolla tuon tiedon saa tulostettua.

Tällaiset luokalle yhteiset asiat pitäisi tietysti jotenkin sitoa itse luokkaan ja kapseloida niin, että vain käyttäjälle tarkoitettu “luokan rajapinta” näkyy ulospäin. Tämän saavuttamiseen eri oliokielissä on käytetty erilaisia mekanismeja. Tässä teoksessa tutustutaan lyhyesti kahteen eri mekanismiin: Smalltalkin metaluokkiin ja luokkaolioihin sekä C++:n luokkafunktioihin ja -muuttujiin. Jotkin ohjelmointi-