Luku 1

Kohti olioita

Trurl päätti lopulta vaientaa hänet kerta kaikkiaan rakentamalla koneen, joka osaisi kirjoittaa runoja. Ensin Trurl keräsi kahdeksansataakaksikymmentä tonnia kybernetiikkaa käsittelevää kirjallisuutta ja kaksitoistatuhatta tonnia parasta runoutta, istahti aloilleen ja luki kaiken läpi. Aina kun hänestä alkoi tuntua, ettei hän pystyisi nielaisemaan enää ainuttakaan kaavakuvaa tai yhtälöä, hän vaihtoi runouteen, ja päinvastoin. Jonkin ajan kuluttua hänelle alkoi valjeta, että koneen rakentaminen olisi lastenleikkiä verrattuna sen ohjelmoimiseen. Keskivertorunoilijan päässä olevan ohjelmanhan on kirjoittanut runoilijan oma kulttuuri, ja tämän kulttuurin puolestaan on ohjelmoinut sitä edeltänyt kulttuuri ja niin edelleen aina aikojen aamuun asti, jolloin ne tiedonsirut, jotka myöhemmin osoittautuvat tärkeiksi tulevaisuuden runoilijalle, pyörteilivät vielä kosmoksen syövereiden alkukaaoksessa. Jotta siis voisi onnistuneesti ohjelmoida runokoneen, on ensin toistettava koko maailmankaikkeuden kehitys alusta pitäen — tai ainakin melkoinen osa siitä.

– Trurlin elektrubaduuri [Lem, 1965]

1.1 Ohjelmistojen tekijöiden ongelmakenttä

Tehokkaiden, luotettavien, halpojen, selkeiden, helppokäyttöisten,

ylläpidettävien, siirrettävien, pitkäikäisten — sanalla sanoen **laadukkaiden** ohjelmistojen tekijät ovat ammattilaisia, jotka joutuvat päivittäisessä työssään toimimaan taiteen ja tieteen rajamailla. Toisaalta ohjelmoijien on ymmärrettävä alansa formaalit periaatteet (ohjelmistojen suunnittelumenetelmistä tietorakenteiden käyttöön ja ohjelmointiin liittyvään matematiikkaan). Toisaalta ohjelmoijat ovat myös taiteilijoita, jotka muokkaavat olemassa olevista materiaaleista ja valmiista rakennuspalikoista ohjelmiston kokonaisuuden. Ohjelmien teon luova puoli on varsinkin alan palkkauksessa vähälle huomiolle jätetty puoli. Silti kaikki tuntevat alalla toimivia "taikureita", jotka hallitsevat bittejä ja niiden kokonaisuuksia muita paremmin.

1.2 Laajojen ohjelmistojen teon vaikeus

Jokainen tietokoneohjelmointia opiskellut on kirjoittanut ensimmäisenä ohjelmanaan korkeintaan muutaman kymmenen rivin ohjelman, jonka avulla on saanut tuntuman ohjelmoinnin käytännön puoleen (ohjelman kirjoittaminen syntaksisesti oikein koneen ymmärtämään muotoon, ohjelman suorittaminen tietokoneella ja mahdollisesti jonkin näkyvän tuloksen saaminen aikaan, ohjelman toimintalogiikan muuttaminen ja sovittaminen halutun ongelman ratkaisemiseksi jne.). Koska lyhyet ohjelmanpätkät saa useimmiten tekemään haluamiaan asioita kohtuullisen lyhyellä vaivannäöllä (muutamasta minuutista muutamaan päivään), on yleinen harhaluulo, että tästä voidaan yleistää suurempien ohjelmistojen valmistamisen olevan *vain* saman prosessin toistaminen isommalle rivimäärälle ohjelmakoodia.

Käytännössä jokaiselle ihmiselle tulee jossain vaiheessa vastaan raja tiedon hallinnassa, kun hallittava tietomäärä kasvaa liian suureksi. Ohjelmien teossa tämä raja tulee näkyviin ohjelman rivimäärän kasvaessa (muuttujista ei muista enää heti kaikkien käyttötarkoitusta, ehto- ja silmukkalauseiden logiikka hämärtyy, tietorakenteiden hallinta sekoaa, osoittimet eivät osoita oikeaan paikkaan jne.).

Ohjelmistotekniikan suurimpia ongelmia on suurten ohjelmistojen tekemisen vaikeus. Tämä ns. **ohjelmistokriisi** ("*software crisis*") on tavallaan tietotekniikan itsensä aiheuttama ongelma. Tietokonelaitteistojen mm. tallennus- ja prosessointitekniikan räjähdysmäinen kehitys on mahdollistanut jatkuvasti aikaisempaa laajempien ja mutkikkaampien ohjelmistojen suorittamisen, mutta näiden ohjelmisto-

jen tekemiseen tarvittavat menetelmät ja työkalut eivät ole kehittyneet läheskään yhtä nopeasti. Ohjelmistotuotannon professori *Ilkka Haikala* on sanonut aiheesta:

> "Ohjelmistokriisiä ratkomaan on kehitetty mm. uusia työkaluja ja työmenetelmiä. Tästä huolimatta ohjelmistotyön tuottavuuden kasvu on tilastojen mukaan ollut vuosittain vain noin neljän prosentin luokkaa. Muutamien vuosien välein kaksinkertaistuvaan ohjelmistojen keskimääräiseen kokoon verrattuna kasvu on huolestuttavan pieni." [Haikala ja Märijärvi, 2002]

Ongelma ei ole mikään uusi ilmiö eikä ole kyse siitä, että se olisi havaittu vasta viime aikoina. Vuonna 1972 vastaanottaessaan Turing Award -palkintoa *Edsger W. Dijkstra* puhui aiheesta:

> "As long as there were no machines, programming was no problem at all; when we had a few weak computers, programming became a mild problem and now that we have gigantic computers, programming has become an equally gigantic problem. In this sense the electronic industry has not solved a single problem, it has only created them — it has created the problem of using its product." [Dijkstra, 1972]

Suurten ohjelmistokokonaisuuksien hallintaan on kehitetty useita erilaisia menetelmiä, mutta mikään niistä ei ole osoittautunut muita selvästi paremmaksi viisasten kiveksi, joka ratkaisee kaikki ohjelmistotyön ongelmat. Näistä eri menetelmistä eniten huomiota ovat viime aikoina saaneet oliokeskeiset menetelmät, joissa nimen mukaisesti keskitytään olioihin. Mitä nämä oliot sitten ovat ja miten ne vaikuttavat ohjelmien suunnitteluun ja toteuttamiseen? Näihin kysymyksiin haemme vastausta seuraavissa luvuissa.

1.3 Suurten ohjelmistokokonaisuuksien hallinta

Seitsemänkymmentäluvulla ryhdyttiin kehittämään menetelmiä ohjelmistokriisin ratkaisemiseksi — tämä työ jatkuu yhä edelleen. Yksi

varhaisimmista ratkaisuperiaatteista on tuttu kaikesta inhimillisestä toiminnasta: ***ongelman jakaminen yhden ihmisen hallittaviin paloihin ja yksinkertaistaminen abstrahoimalla***. Tämä periaate on nähtävissä, kun tarkastellaan ohjelmoijien tärkeimpien työkalujen, ohjelmointikielten ja niiden periaatteiden kehitystä.

Mitä abstrahointi tarkoittaa? Voimme aloittaa vaikka sivistyssanakirjan selvityksellä:

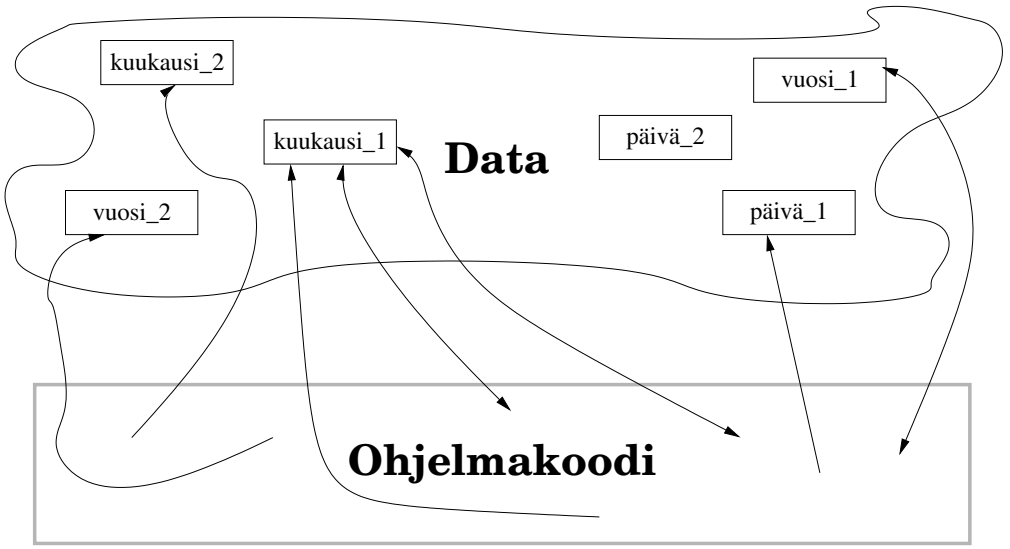
> **Abstraktio**. Ajatustoiminta, jonka avulla jostakin käsitteestä saadaan yleisempi käsite vähentämällä siitä tiettyjä ominaisuuksia. Myös valikointi, jossa jokin ominaisuus tai ominaisuusryhmä erotetaan muista yhteyksistään tarkastelun kohteeksi.
>
> **Abstrahoida**. Suorittaa abstraktio, erottaa mielessään olennainen muusta yhteydestä.
>
> **Abstrakti**. Abstrahoimalla saatu, puhtaasti ajatuksellinen, käsitteellinen. [Aikio ja Vornanen, 1992]

Kerätään siis yhteen toisiinsa liittyviä asioita ja nimitetään niiden kokonaisuutta jollain kuvaavalla “uudella” termillä tai kuvauksella. Ohjelmointi on täynnä tällaisia rakenteita. Otetaan esimerkiksi hyvin yleinen operaatio: tiedon tallentaminen massamuistilaitteelle. Sovellusohjelmoijan ei tarvitse tuntea kiintolevyn valmistajan käyttämää laitteen ohjauksen protokollaa, koodausta, ajoitusta ja bittien järjestystä, vaan hän voi käyttää erikseen määriteltyjä korkeamman (abstraktio)tason operaatioita (esim. `open`, `write` ja `close`).

Seuraavissa aliluvuissa näemme, miten abstrahoinnilla saadaan selkeämmäksi tietokoneohjelman rakenteen hallinta.

1.3.1 Ennen kehittyneitä ohjelmointikieliä: ei rakennetta

Ohjelmoinnin historian alkuhämärässä ohjelmoinnin tavan määräsi laitteisto. Prosessorien ymmärtämiä käskykoodeja kirjoitettiin suoraan joko niiden numerokoodeilla tai myöhemmin symboleja näiksi koodeiksi muuntavan assembler-ohjelman avulla. Tällaisessa laitteistonläheisessä näperryksessä kaikista hiemankin suuremmista ohjelmista tuli spagettiröykkiöitä, joissa tieto oli yksittäisiä muistipaikkoja, ja tietoa käsittelevää ohjelmakoodia oli ripoteltuna ympäriinsä. Kuvassa 1.1 seuraavalla sivulla on esimerkki ohjelmasta (koodi ja



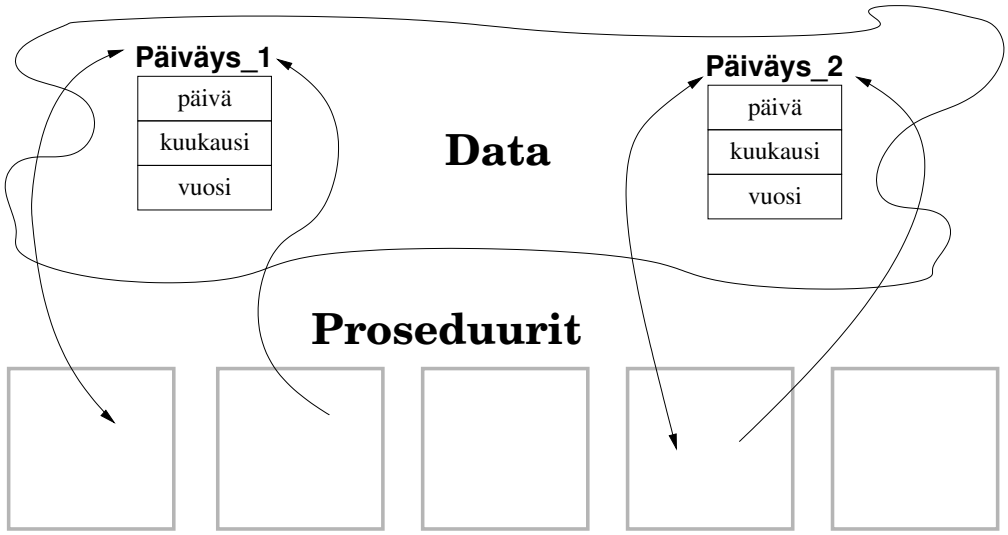
KUVA 1.1: Tyypillinen assembly-ohjelman spagettirakenne

data), joka käsittelee kahta päiväystä: ohjelmoija on varannut kolme muistipaikkaa yhdelle päiväykselle ja näitä tietoja käsitellään ohjelmakoodista suoraan muistipaikkojen osoitteilla. Rakennekuvasta näkee hyvin, että tällaisesta rakenteesta on esim. ylläpitovaiheessa hyvin vaikea selvittää, mitkä kaikki ohjelmakoodin kohdat viittaavat johonkin määrättyyn muistipaikkaan (esim. `kuukausi_1`).

1.3.2 Tietorakenteet: tiedon kerääminen yhteen

Kun ohjelmointikielten kehitys kuusikymmentäluvulla pääsi laajemmassa mitassa alkamaan, ryhdyttiin tekemään ohjelmoijien kokemusten pohjalta kieliä, joissa tietoa pystyttiin käsittelemään muistipaikkoja korkeammalla abstraktiotasolla — **tietorakenteina**.

Tietorakenteet ovat nimettyjä kokonaisuuksia, jotka koostuvat muista tietorakenteista tai ns. kielen perustyypeistä. Näiden hierarkkisten rakenteiden avulla saatiin kerättyä yhteen kuuluvat tiedot saman nimikkeen alle. Esim. päiväystieto voitaisiin haluta käsitellä kolmena kokonaislukuna, jotka kuvaavat vuotta, kuukautta ja päivää. Näistä voidaan muodostaa ohjelmointikielen tasolla yksi päiväykseksi nimetty tietorakenne. Vastaavasti ohjelman toiminnallisuutta voi-



KUVA 1.2: Päiväykset tietorakenteina

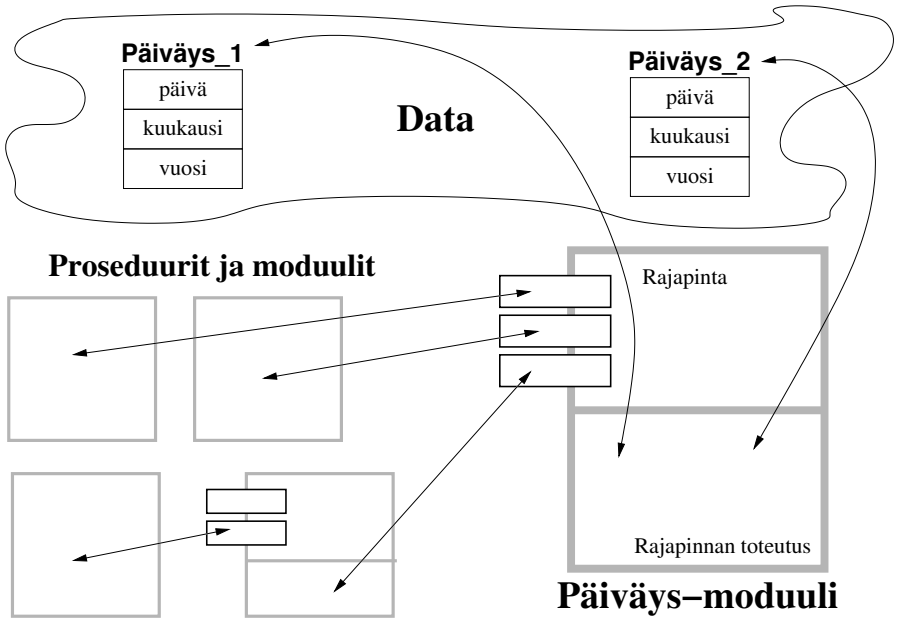
daan jakaa yhden toiminnon toteuttaviin paloihin, joita nimitetään funktioiksi tai proseduureiksi.

Kuvassa 1.2 on esimerkki kahdesta päiväyksestä ja niiden tietorakenteista. Tässä mallissa päiväyksiä käsitellään ohjelmakoodissa edelleen missä kohtaa tahansa aivan kuten spagettimallissakin. Tietorakenteita ja funktioita tukevia ohjelmointikieliä ovat mm. Pascal ja C.

1.3.3 Moduulit: myös ohjelmakoodi kerätään yhteen

Vuosi 2000 -ongelma oli tietotekniikassa paljon puhetta ja työtä synnyttänyt aihe. Siinä ohjelmistoista oli tarkastettava, että ne eivät oleta kaikkien vuosilukujen olevan muotoa 19xy. Näiden kohtien löytäminen on edellisten käytäntöjen mukaan tehdyissä ohjelmistoissa hyvin vaikeata, sillä mikä tahansa ohjelmiston osa voi suoraan käsitellä päiväyksen sitä kokonaislukua, joka kuvaa vuosilukua.

Jotta tällaiset toiminnalliset viittaukset tietorakenteisiin saataisiin hierarkkiseen hallintaan, päätettiin määritellä joukko määrättyyn tietorakenteeseen liittyviä funktioita ja sopia, että ***vain näillä funktioilla saa käsitellä kyseistä tietorakennetta*** (esim. päiväys, katso kuva 1.3 seuraavalla sivulla). Tätä funktiokokoelmaa nimitetään **rajapinnaksi** (*interface*) ja sen funktioita **rajapintafunktioiksi**.



KUVA 1.3: Päiväyksiä käsittelevän moduulin rakenne

Ohjelmointikielten rakenteita, joissa kootaan tietorakenteet ja niitä käsittelevä ohjelmakoodi samaan kokonaisuuteen sekä erikseen määritellään tietorakenteiden käsittelyyn toiminnallinen rajapinta, nimitetään **moduuleiksi** (*module*). Koska moduulit kätkevät niiden rajapinnan toiminnallisuuden toteutuksen, moduuleja tulee tarkastella kahdesta näkökulmasta:

- **Moduulin käyttäjä.** Moduulin käyttäjä on ohjelmoija, joka tarvitsee moduulin tarjoamaa palvelua oman koodinsa osana. Tässä ulkopuolisessa näkökulmassa meitä kiinnostaa vain moduulin ulkoinen eli **julkinen rajapinta**: miten sitä tulee käyttää ja saammeko sen avulla toteutettua haluamamme toiminnan? (Moduulilla voi olla myös ns. sisäinen rajapinta, joka on tarkoitettu vain moduulin tekijän käyttöön.)

- **Moduulin toteuttaja.** Moduulin suunnittelijan ja toteuttajan vastuulla on määritellä ja dokumentoida moduulille rajapinta, joka on yksinkertainen, helppokäyttöinen, selkeä ja käyttökelpoinen. Vain moduulin toteuttajan täytyy miettiä ja toteuttaa rajapinnan takana oleva toiminnallisuus — tämä osa on muilta ohjelmoijilta kätkettynä siinä mielessä, että heidän ei vält-

tämättä tarvitse tuntea toteutuksen yksityiskohtia. Tätä nimitetään **tiedon kätkennäksi**. Siinä **kapseloidaan** käytön kannalta turha tieto moduulin sisälle (*encapsulation*). Kyseessä on abstrahoinnin tärkeimpiä työkaluja ohjelmoinnissa.

Käytännössä moduuliajattelu voi olla vain sopimus määrätyn rajapinnan noudattamisesta (esim. Pascal ja C), tai ohjelmointikieli voi jopa kieltää (ja tarkastaa) rajapinnan takana olevien tietorakenteiden käsittelyn moduulin ohjelmakoodin ulkopuolelta (Ada, Modula).

Rajapinta-ajattelu pakottaa suunnittelemaan tarkemmin ennalta, miten määrättyä tietorakennetta on tarkoitus käyttää eli mitä palveluita tietorakenteen lisäksi halutaan siihen liittyen tarjota. Tiedon kätkennän ja rajapinta-ajattelun haittapuolina voidaankin pitää suunnittelun vaikeutumista. Moduulin tekijälle on helppo sanoa päämääräksi yksinkertainen ja käyttökelpoinen (eli täydellinen?) rajapinta. Näiden vaatimusten toteuttamiseen sitten tarvitaankin todellista ohjelmointitaikuria. Käytännön ohjelmistotyössä rajapintoja joudutaan tarkastamaan ja tarkentamaan ohjelmiston kokonaissuunnittelun edetessä ja jatkossa ohjelmistoa ylläpidettäessä. [Sethi, 1996]

Rajapinta edistää merkittävästi ohjelmistojen ylläpidettävyyttä, koska ne rajapinnan “takana” tehdyt muutokset, jotka eivät vaikuta rajapinnalle määriteltyyn toiminnallisuuteen, eivät aiheuta mitään muutoksia muuhun ohjelmistoon. Näin saadaan abstrahoitua laajan ohjelmiston kokonaisuutta rajapinnoilla toisistaan eroteltuihin mahdollisimman itsenäisiin paloihin.

Vuosi 2000 -ongelman tapauksessa modulaarisessa ohjelmistossa on ensin tarkastettava, että rajapinnan kautta ei päästä käsittelemään vuosilukuja väärällä tavalla. Jos vuosisatojen 1900 ja 2000 väliseen käsittelyyn liittyvät rakenteet ovat mahdollisia vain moduulin sisällä, niin riittää, että tarkastamme kaiken moduulin koodin ja varmistamme sen toimivan myös vuoden 1999 jälkeen. On tärkeätä huomata, että jos rajapinnan määrittely “paljastaa” vuosiluvun tietorakenteen ulkopuolelle esimerkiksi 0–99 välille määriteltynä kokonaislukuna, niin joudumme edelleen tarkastamaan myös kaiken moduulin ulkopuolella päiväyksiä käsittelevän ohjelmakoodin. Tällaista rajapintaa voidaan pitää huonosti suunniteltuna vuosi 2000 -ongelman kannalta. Kyse on kuitenkin jälkiviisaudesta, jos rajapinnan suunnittelun aikaan ohjelmiston arvioitu elinkaari ei ole yltänyt vuoden 1999 ylitse. Olisiko rajapinnan määrittelijän pitänyt ottaa huomioon ongelma-

kentän ulkopuolisia asioita? Ja mitä kaikkia niistä? Rajapinta-ajattelu ei ole inhimillisiä virheitä korjaava tai estävä menetelmä, mutta se on ihmisille luontaista hierarkkista ajattelua tukeva menetelmä, joka selkeyttää suurten ohjelmistojen rakennetta.

Modulaarista ohjelmointia tukevia ohjelmointikieliä ovat mm. Ada ja Modula. Listauksessa 1.1 on esimerkki päiväys-moduulin tietorakenteesta ja osasta rajapintaa Modula-3-kielellä.

1.3.4 Oliot: tietorakenteet, rajapinnat ja toiminnot yhdessä

Kun tarkastelemme päiväysrajapinnan käyttöä (listaus 1.2 seuraavalla sivulla), niin huomaamme heti, että rajapintafunktioille on aina välitettävä parametrina se tietorakenne, johon operaatio halutaan kohdistaa. Tästä käytännön tarpeesta on lähtöisin ajattelumalli, jossa ei haluta korostaa pelkästään moduulin toiminnallista osaa (rajapintafunktiot) vaan ajatellaan rajapintaa tietorakenteen “osana”. Tämän mallin mukaisia alkioita, jotka yhdistävät tietorakenteet ja rajapinnan, nimitetään olioiksi (kuva 1.4 seuraavalla sivulla). Uudessa ajattelumallissa kohdistetaan rajapintakutsu määrättyyn olioon, ja tämä näkyy myös olio-ohjelmointikielen tasolla (listaus 1.3 sivulla 36).

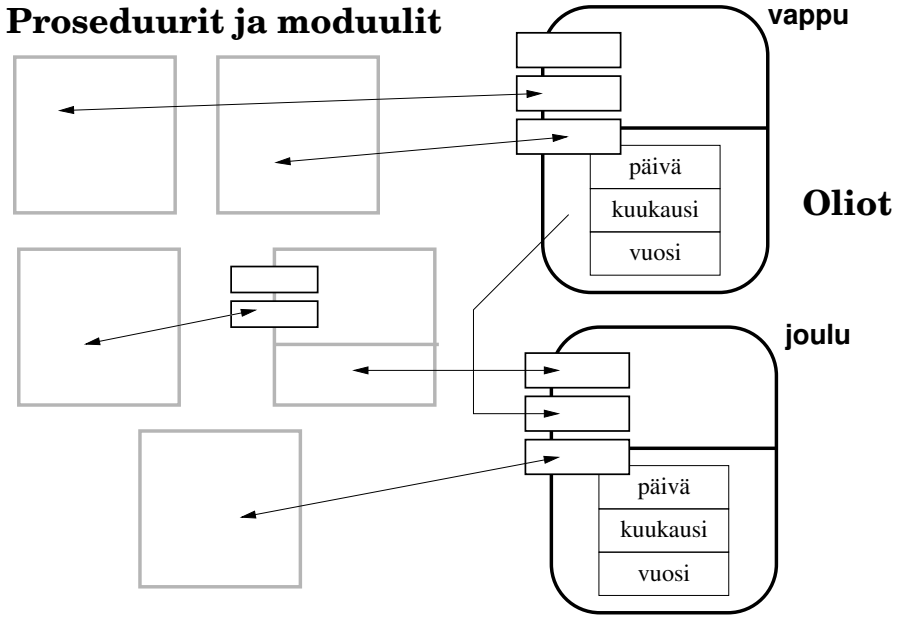
Oliomallin mukaisen ohjelmistojen suunnittelun ja toteutuksen katsotaan sopivan paremmin ihmisten luontaiseen tapaan tarkastella ongelmia. Oli toteutettavana järjestelmänä lentokoneen toiminnan valvonta tai šakkipeli, niin järjestelmän katsotaan koostuvan osista ja näiden osien toisiinsa ja ulkomaailmaan kohdistamista toiminnoista.

```
1  INTERFACE päiväys;
2
3  TYPE
4    PVM : RECORD
5      päivä, kuukausi, vuosi : INTEGER;
6    END;
7
8  PROCEDURE Luo( päivä, kuukausi, vuosi : INTEGER ) : PVM;
9  PROCEDURE PaljonkoEdellä( eka, toka : PVM ) : INTEGER;
10 PROCEDURE Tulosta( kohde : PVM );
```

LISTAUS 1.1: Päiväysmoduulin tietorakenne ja osa rajapintaa

```
1  IMPORT päiväys;
2
3  VAR vappu : päiväys.PVM;
4
5  BEGIN
6    vappu := päiväys.Luo( 1, 5, 2001 );
7    päiväys.Tulosta( vappu );
8  END;
```

LISTAUS 1.2: Päiväysmoduulin käyttöesimerkki



KUVA 1.4: Rajapinta osana tietorakennetta (oliot)

Tämä uusi ajattelumalli vaikuttaa useisiin ohjelmistojen tekemisen osa-alueisiin:

- **Määrittely.** Käytettäessä olioita on määrittelyssä pidettävä mielessä tämä päämäärä (päädytään oliorakenteeseen).
- **Suunnittelu.** Oliosuunnittelussa on tunnettava olioiden ja luokkien ominaisuuksia (mm. periytyminen ja geneerisyys), jotta jo suunnitteluvaiheessa pystytään hyödyntämään uuden ajattelumallin kaikki (tarvittava) teho.

```
1  PROCEDURE TulostaAikaisempi( päivä_A, päivä_B : PVM )
2  BEGIN
3    IF päivä_A.PaljonkoEdellä( päivä_B ) > 0 THEN
4      päivä_A.Tulosta();
5    ELSE
6      päivä_B.Tulosta();
7    END;
8  END;
9
10 BEGIN
11   vappu := PVM( 1, 5, 2000 );
12   joulu := PVM( 24, 12, 1999 );
13   TulostaAikaisempi( vappu, joulu );
14 END.
```

LISTAUS 1.3: Päiväysolioiden käyttö

- **Ohjelmointi.** Erityiset olio-ohjelmointikielet helpottavat olioita sisältävän suunnitelman toteuttamista huomattavasti. Työkaluina nämä kielet ovat aikaisempia mutkikkaampia, koska ne tuovat lisää uusia laajoja ominaisuuksia aikaisempiin (proseduraalisiin) kieliin verrattuna.

Perinteisessä ohjelmistosuunnittelussa on otettu lähtökohdaksi osat (data, tietorakenteet) *tai* toiminnot (funktiot ja rajapinnat) ja pyritty rakentamaan koko ohjelmisto tästä yhdestä jaottelusta lähtien. Olio-ohjelmoinnin yksi tärkeimmistä suunnitteluperiaatteista on jatkuvasti yrittää pitää näitä molempia näkökantoja mukana suunnitteluprosessissa: järjestelmästä tunnistetaan sekä olioita että niiden välisiä toiminnallisuuksia.

Ongelmalähtöinen ja laitteistoläheinen näkökulma olioihin

Olion voidaan katsoa olevan osa sen ongelman ratkaisua, johon ohjelmistolla pyritään. Tämä on ulkoinen tai ongelmalähtöinen näkökulma oliohin. Ohjelmiston suunnittelija pyrkii tunnistamaan esim. šakkipelistä pelin mallintamiseen tarvittavat oliot: erilaiset nappulat, lauta, peliajan mittaava kello jne.

Olioista koostuvan ohjelmiston toiminnallisuus on olioiden välistä kommunikointia. Oliot kutsuvat toistensa rajapintojen kautta tarvitsemiaan toimintoja. Esimerkiksi šakkinappulaolio voi pyytää šak-

kilautaa mallintavalta oliolta tietoa siitä, voiko se siirtyä määrättyyn ruutuun. Kyselyn seurauksena lautaolio voi taas vuorostaan kysyä muilta nappuloilta niiden nykyisiä paikkoja jne. Ohjelmiston ylimmän tason "logiikka" voidaan näin määritellä olioiden rajapintojen ja olioiden välisen kommunikaation avulla — tässä suunnitteluvaiheessa ei tarvitse ottaa kantaa siihen, miten nämä toiminnot tullaan yksityiskohtaisesti toteuttamaan.

Toisaalta jokaisen olion voi ajatella edustavan erillistä tietokonetta, jolla on oma ohjelmakoodi (rajapinnan takainen toteutus) ja muisti (olion tietorakenteen arvot eli olion tila). Nämä "minikoneet" kommunikoivat lähettämällä toisilleen viestejä, joilla pyydetään jonkin toiminnon suorittamista toisessa koneessa eli oliossa. Tämä ajattelumalli on sisäinen tai laitteistonläheinen näkökulma olioihin. Malli voi olla hyödyllinen esimerkiksi hajautetuissa järjestelmissä, joissa osa ohjelmiston olioista tulee lopullisessa toteutuksessa todellisuudessakin sijaitsemaan eri fyysisissä tietokoneissa. [Sethi, 1996]

Ohjelmiston staattiset ja dynaamiset osat

Oliomallia ei ole tarkoitettu syrjäyttämään modulaarisuutta vaan täydentämään sitä. Moduulien avulla ohjelmistoon voidaan tehdä yleensä ylimmillä tasoilla olevaa jakoa osiin ja niiden välisiin rajapintoihin: käyttöliittymän näkymät, näyttölaitteiden rajapinnat, tietokanta jne. Koska tämä jaottelu osiin on ohjelmassa pysyvä, sitä nimitetään staattiseksi jaotteluksi.

Dynaamiset osat tarkoittavat tietorakenteiden ja rajapintojen kokoelmia, joissa yhden mallin mukaisia olioita voi esiintyä ohjelman ajoaikana useita. Esim. päiväysoliolla on aina sama toiminnallinen rajapinta ja sama tietorakenne (kolme kokonaislukua). Eri päiväyksillä voi olla erilaiset arvot tietorakenteessa. Saman mallin (päiväys) eri ilmentymillä (oliot) sanotaan olevan erilainen tila (tietorakenteen arvot). Olioiden dynaamisuutta on myös se, että niitä voi syntyä ja tuhoutua ohjelman suorituksen aikana — kutakin moduulia taas on olemassa yksi kappale koko ohjelmiston ajossaolon ajan.

Olio on itsenäinen kokonaisuus

Oliokeskeisessä ajattelumallissa olioiden katsotaan olevan itsenäisiä kokonaisuuksia, joille on määritelty tarkka **vastuualue** (*responsibil-*

ity) ohjelmistossa [Budd, 2002]. Vastuualue on nimitys kaikelle sille, mitä olion on määrätty toteuttavan. Esim. päiväysoliolla on määriteltynä vastuu tiedon taltioinnista (olio kuvaa yhtä päivämäärää) ja julkinen rajapinta (tarjotut palvelut). Vastuualue kattaa muutakin kuin ohjelmointikielen muuttujia ja funktioita. Se voi esimerkiksi määritellä että jokin olio on vastuussa määrättyjen muiden olioiden luomisesta ja tuhoamisesta eli niiden elinkaaresta.

1.3.5 Komponentit: moduulit ja oliot yhdessä

On tärkeätä huomata, että moduulit ja oliot voivat muodostaa hierarkioita ja kokonaisuuksia yhdessä. Ohjelmistossa voidaan esim. ylimmällä tasolla määritellä ajanlaskusta vastuussa oleva moduuli, jonka tarjoamista palveluista yksi on päiväyksiä kuvaavat oliot.

Hyvin suunniteltu rajapintojen, olioiden ja toteutusten (samalla rajapinnalla voi olla esimerkiksi vaihtoehtoisia toteutuksia) kokoelmaa on viime aikoina ryhdytty nimittämään **komponentiksi** (*component*). Ideana olisi tarjota ohjelmistojen valmistajille hieman samanlainen tilanne kuin tällä hetkellä on mm. elektroniikkasuunnittelijoilla — he voivat suunnitella ja valmistaa tuotteitaan hyvin laajan valmiin komponenttivalikoiman avulla. Tuote luodaan yhdistelemällä elektroniikkakomponentit uusiksi mutkikkaammiksi tuotteiksi. Vastaavalla tavalla uudelleenkäyttöä varten suunniteltuja ohjelmakomponentteja voitaisiin pitää uusien ohjelmistojen pohjana.

Ohjelmistokomponentti on itsenäinen kokonaisuus, jota sen tarjoamien julkisten rajapintojen avulla voidaan hyödyntää osana suurempaa kokonaisuutta. Komponentti kerää yhteen määrätyn vastuualueen toteuttamiseen tarvittavat ohjelmiston moduulit ja oliot (olio-ohjelmana toteutus on usein kokoelma luokkia). Komponenttimarkkinoita on tällä hetkellä olemassa erityisesti graafisten käyttöliittymien alueella, mutta vasta tulevaisuus tulee näyttämään, onko kyseessä ohjelmistojen tekemistä suuremmassa mittakaavassa muokkaava ajattelumalli.

Aivan viime vuosien ohjelmistokomponentit sisältävät usein rajapintadokumentaation ja lähdekoodin lisäksi myös valmiin binäärimuotoisen toteutuksen. Tällöin komponentti otetaan sellaisenaan (valmiiksi konekoodiksi käännettynä) käyttöön omaan ohjelmistoon. Yksi tällainen komponenttiarkkitehtuuri on JavaBeans [JavaBeans, 2001].

Luku 5

Oliosuunnittelu

> *Koska eräät suunnittelun piirteet eivät ole analyyttisiä tai tieteellisiä, suunnittelun opetus ei yleensä sovi hyvin arvovaltaisiin teknisiin korkeakouluihin, joiden työntekijät tuntevat olonsa kotoisammaksi opettaessaan matematiikkaa ja tieteitä kuin kertoessaan, miten yksilön fysikaalisen maailman ilmiöitä koskevaa arvostelukykyä, estetiikkaa, luovuutta ja herkkyyttä kehitetään. Monet tekniikan professorit toivovat, että suunnittelu olisi tieteellisempää. Itse asiassa tekniikan piirissä on ajoittain liikkeitä, joiden tavoitteena on tehdä suunnittelusta analyyttisempää. Ne epäonnistuvat suunnittelun pehmeämpien osien suhteen. Käytännössä suunnittelu on huomattavasti kriittisempää ja arvostetumpaa toimintaa kuin sen asema tekniikan koulutuksessa antaa ymmärtää.*
>
> *– Insinöörin maailma [Adams, 1991]*

Ohjelmistomäärittely ja -suunnittelu on hyvin laaja-alainen alue, johon on olemassa erilaisia teorioita, menetelmiä ja kansanperinnettä. Kokonaiskuvan alueesta saa esim. teoksesta “Ohjelmistotuotanto” [Haikala ja Märijärvi, 2002]. Kannattaa myös pitää aina mielessä, että ohjelmistojen valmistaminen ei ole ainoastaan tietojenkäsittelyä — ohjelmiston “sieluna” olevan toimintojen filosofian määrittelee aina ihminen ja lähes aina ihmisiä varten [Roszak, 1992]. Tässä luvussa esitellään muutamia oliosuunnitteluun sopivia yleisiä periaatteita ja kuvaustapoja.

5.1 Oliosuunnittelua ohjaavat ominaisuudet

Ohjelmiston jako moduuleihin ja luokkiin on ratkaisu, jota ohjaavat monet tekijät: ongelman ratkaisu, käytetyt työkalut, menetelmät, kokemus, "talon perinteet", reunaehdot, jatkokehityksen suunnitelmat jne. Luokkahuone-esimerkkejä laajemmissa ohjelmistoissa ei koskaan ole olemassa vain yhtä ainoata oikeata tapaa tehdä moduuli- ja luokkajakoa. Seuraavat aliluvut esittelevät niitä asioita, joita oliosuunnittelussa tulisi osata ottaa huomioon [Booch, 1987].

5.1.1 Mitä suunnittelu on?

Kaikessa insinöörityössä suunnittelulla pyritään löytämään ratkaisu johonkin ongelmaan. Ratkaisun "rakennepiirustusten" avulla voidaan valmistaa haluttu tuote. Suunnittelu on reitti ongelman kuvauksen ja määrittelyn sekä lopullisen tuotteen välillä.

Suunnittelun tarkoituksena on saada aikaan järjestelmä, joka

- toteuttaa ongelman (toiminnallisen) kuvauksen
- voidaan toteuttaa käytössä olevilla raaka-aineilla ja resursseilla
- sopii implisiittisiin ja eksplisiittisiin resurssirajoihin [Booch, 1991, s. 20]
- erityisesti ohjelmistojen tapauksessa lopputuloksen tulisi olla varautunut jatkokehitykseen ja ylläpitoon (esimerkiksi laadukkaan dokumentaation avulla).

Varsinkin järjestelmän "piilo-oletusten" kaivaminen esille osaksi suunnitelmaa on yksi vaikeimmista suunnittelutyön osista.

Ohjelmistosuunnittelu on termi, joka voidaan määritellä tarkoittamaan ohjelmiston vaatimusten ja toiminnallisuuden suunnittelua menetelmällä, jossa lopputuloksena saadaan mahdollisimman helposti toteutettava (ohjelmoitava) suunnitelma (esimerkiksi suunnitteludokumentti). Oliosuunnittelussa tämä tarkoittaa sitä, että *jo suunnitteluvaiheessa on otettava huomioon valitun menetelmän (olioiden) ominaisuudet ja pyrittävä tekemään suunnitelma, joka tukee olioiden käyttöä toteutusvaiheessa*.

Ohjelmistotuotteiden tapauksessa lopputulos on äärimmäisen harvoin kerralla valmis (ja heti perään unohdettu) kokonaisuus vaan

tuotteen julkistuksen jälkeen sitä kehitetään eteenpäin lisäämällä ominaisuuksia ja (valitettavan usein) korjaamalla aikaisempiin versioihin jääneitä virheitä.

5.1.2 Abstraktio ja tiedon kätkentä

Oliosuunnittelun tärkeimpiä työkaluja on jo aikaisemmin esitelty moduulien esittelyn yhteydessä (katso luku 1.3.3). Modulaarisuus on ohjelmiston jakoa paremmin hallittaviin kokonaisuuksiin abstrahoinnin ja tiedon kätkennän avulla. Suunnitteluvaiheessa tämä tarkoittaa ohjelmiston jakamista paloihin joko osittavalla tai kokoavalla jaottelulla.

- **Osittava** (*top-down*). Haetaan järjestelmän suurimmat kokonaisuudet, jotka usein ovat toiminnallisia, kuten käyttöliittymä, tietokanta, syöttö ja tulostus, tietoliikenne ja rajapinnat muihin ohjelmiin. Näitä osakokonaisuuksia jaetaan taas vuorostaan pienempiin paloihin, joista lopulta muodostuu moduuleja ja luokkia.

- **Kokoava** (*bottom-up*). Jos suunnittelun alussa tunnetaan parhaiten joidenkin osajärjestelmien toiminta, niin moduulisuunnittelu voidaan aloittaa niistä ja myöhemmin kerätä näitä osia suuremmiksi kokonaisuuksiksi.

Käytännön suunnittelutyö ei tietenkään seuraa pelkästään jompaa kumpaa näistä tavoista vaan on niiden yhdistelmä. Jaottelun yhteydessä tunnistetaan ohjelmiston staattista rakennetta, josta tulee moduulijako ja dynaamisia rakenteita (olioita), jotka kuvataan luokkina. Koska luokka pystyy ilmaisemaan kaikki moduulin tärkeimmät ominaisuudet, useat oliosuunnittelumenetelmät käyttävät suunnitteluvaiheessa vain luokkia.

5.1.3 Osien väliset yhteydet ja lokaalisuusperiaate

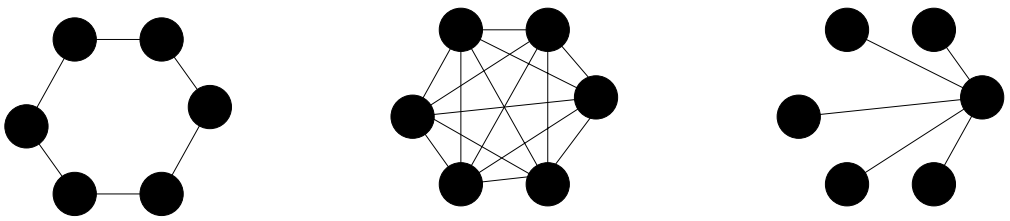
Jotta ohjelmiston moduulien välinen kommunikaatio ei muistuttaisi spagettikoodin aikaisia sotkuja, suunnitteluvaiheessa on pyrittävä pitämään moduulien väliset viittaukset mahdollisimman pieninä. Moduulin A katsotaan viittaavan moduuliin B, kun A tarvitsee jotain

palvelua B:n julkisesta rajapinnasta. Kokoavassa suunnittelussa pyritään keräämään kaikki läheisesti toisiinsa kuuluvat moduulit samaan ylemmän tason kokonaisuuteen (esimerkiksi kaikki ajanlaskuun ja kalenteriin liittyvät moduulit). Tämä vahvasti kytkeytyneiden moduulien paketoiminen uuden pelkistetymmän rajapinnan taakse on esimerkki lokaalisuuden säilyttämisestä suunnittelussa (**lokaalisuusperiaate**).

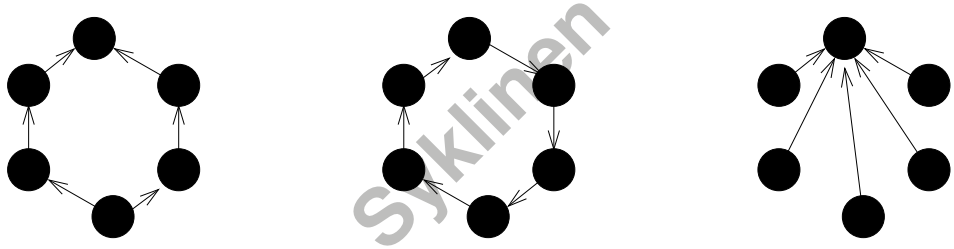
Kuvassa 5.1 on erilaisia riippuvuusvaihtoehtoja moduulien välillä. Jos alijärjestelmässä on n moduulia, niiden välillä on vähintään $n - 1$ riippuvuutta (jokainen osa tietää kokonaisuuden julkisen rajapinnan, joka on yksi osa). Maksimissaan kaikki moduulit tietävät kaikkien muiden olemassaolosta, jolloin riippuvuuksien lukumäärä on $\frac{n(n-1)}{2}$ [Meyer, 1997].

Lokaalisuusperiaate pyrkii minimoimaan ohjelmakomponenttien välisiä yhteyksiä ja näin pitämään kokonaisuuden kompleksisuutta paremmin hallinnassa. Osien välisten riippuvuuksien lukumäärän vähenemisen lisäksi ohjelmiston rakenne yksinkertaistuu, jos riippuvuudet pidetään aina mahdollisuuksien mukaan yksisuuntaisina. Päiväysmoduulin osat eivät tiedä (eivätkä välitä siitä), että niitä käytetään suuremman kokonaisuuden osina. Jos riippuvuudet muodostavat syklisiä silmukkaviittauksia, mutkistuu rakenteiden toteuttaminen: jos A tarvitsee B:tä ja B tarvitsee A:ta, niin kuinka A voidaan toteuttaa ennen kuin on esitelty millainen on B, jota taas ei voida toteuttaa ennen kuin A:n toteutus on valmis jne. (C++:n ratkaisu tähän “muna-kana” -ongelmaan on esitetty aliluvussa 4.4.)

Kuvassa 5.2 seuraavalla sivulla on esimerkkejä erilaisista yksisuuntaisista riippuvuuksista, joista yhdessä riippuvuudet muodostavat silmukan.



KUVA 5.1: Erilaisia yhteysvaihtoehtoja kuuden moduulin välillä



KUVA 5.2: Moduulien välisiä yksisuuntaisia riippuvuuksia

5.1.4 Laatu

Laadukkaisiin suunnitelmiin, moduuleihin ja luokkiin kuuluvia ominaisuuksia on helppo luetella ([Booch, 1987], [Liberty, 1998], [Meyer, 1997]), mutta käytännössä vaikea toteuttaa kaikilta osiltaan:

- **Oikeellisuus.** Ohjelmiston komponenttien (moduuli, luokka tai koko ohjelmisto) tulee toteuttaa kaikki toiminnot suunnitelman määrittelemällä tavalla.
- **Virheitä sietävä.** Komponenttien tulee varautua virhetilanteisiin (väärä syöte, muistin loppuminen jne.) ja osata toimia virhetilanteen havaitessaan etukäteen suunnitellulla tavalla (*robustness*).
- **Selkeys.** Ohjelmiston toiminnallisten yksiköiden rajapintojen dokumentointi ja toteutus on tehtävä yhtenäisellä ja selkeällä menetelmällä, jotta niiden muuttaminen ja uudelleenkäyttö olisi helppoa.
- **Etukäteissuunnittelu.** Moduulien ja luokkien suunnittelussa tulee pyrkiä ottamaan huomioon kohdeprojektin tarpeiden lisäksi yleisempiä näkökohtia, jotta syntyvä komponentti olisi suuremmalla todennäköisyydellä myös käyttökelpoinen osa tulevia projekteja.
- **Tehokkuus.** Komponenttien tulee käyttää tuhlailematta hyväkseen järjestelmän resursseja (prosessointiaika ja muisti).

- **Siirrettävyys.** Komponenttien suunnittelussa ja toteutuksessa tulisi ottaa huomioon, että osia mahdollisesti käytetään tulevaisuudessa niiden kehitysympäristöstä poikkeavassa ympäristössä.
- **Elinkaaren huomioiva.** Laadukkaan ohjelmiston rakenteen ja dokumentaation tulisi tukea jatkokehitystä, jota kaikille "todellisille" ohjelmistoille tullaan jossain vaiheessa tekemään.

"Rational design process and how to fake it"

Tämä *David Parnasin* [Clements ja Parnas, 1986] artikkelin otsikko kuvaa käytännön ohjelmistotyön ristiriitaa usein hyvinkin ylevien suunnitteluperiaatteiden kanssa.

Näitä ristiriitoja on useita. Mikä on oikein toimiva ohjelma? (Asiakkaan ja määrittelijän kanta ei välttämättä ole aina sama.) Mihin kaikkiin mahdollisiin (ja mahdottomalta tuntuviin) virhetilanteisiin ohjelmiston tulisi varautua — ja miten? Kun aikataulut, kiire ja stressi painavat päälle, niin kuka jaksaa ajatella ohjelmiston tulevia ylläpitäjiä ja koodin selkeyttä?

Vaikka käytännön työ aina välillä onkin kaukana akatemian ylevistä päämääristä, niin harva silti väittää, että huolellisesta ohjelmistojen suunnittelusta tulisi luopua kokonaan. Vaikka joissain osissa joudutaankin joustamaan, niin jo pitämällä aina mielessään ylevämpiä päämääriä pystyy varmasti pitämään laatunsa "aloittelijan spagettihäkkyrää" parempana.[†]

5.2 Oliosuunnittelun aloittaminen

Alku aina hankalaa. Ensimmäisenä täytyy pitää mielessä kaikki kokonaisuuteen liittyvät asiat. Oliosuunnittelun päävaiheet ovat (luettelossa tarkoitetaan komponentilla ohjelmiston moduuleja ja luokkia) [Booch, 1987]

1. komponenttien tunnistaminen
2. komponenttien vastuualueiden määrittely

[†] Kun joku keksii ideointiin, suunnitteluun, ohjelmointiin ja ihmisten johtamiseen menetelmän, jolla saa ylivoimaista laatua täysin aikataulujen ja budjetin mukaisesti, niin pyydämme nöyrästi kertomaan siitä meille ja muullekin maailmalle.

3. komponenttien välisten suhteiden määrittely (keskinäinen näkyvyys)

4. komponenttien rajapintojen määrittely (esimerkiksi formaalisti matemaattisella notaatiolla, mahdollisimman yksikäsitteisesti ohjelmointikielen rakenteilla tai sanallisella kuvauksella)

5. viimeisenä vaiheena edellä määriteltyjen luokkien ja moduulien sekä niiden muodostaman kokonaisuuden (=ohjelmisto) toteutus.

Luettelo on hyvä esimerkki säännöistä, joissa käytännön suunnittelutyössä ei edetä yksi kohta kerrallaan — "Tämä sääntö mietitään nyt loppuun ennen kuin jatketaan seuraavaan vaiheeseen". Tärkeä osa suunnittelua on ideointi, jota ei kannata rajoittaa vain yhteen osa-alueeseen kerrallaan. Kun johonkin kohtaan sopiva idea tulee mieleen, niin se kannattaa kirjoittaa muistiin ja luottaa siihen, että myöhemmin suunnittelun viimeistelyssä mahdolliset turhat tai mahdottomat ideat karsiutuvat pois.

5.2.1 Luokan vastuualue

Jokaisesta olio-ohjelmassa olevasta luokasta pitäisi olla olemassa selkeä ja kattava kuvaus. Yhdellä sanalla ilmaistuna on määriteltävä luokan **vastuualue** [Budd, 2002]. Vastuualue määrittelee sen, mitä kyseisen luokan olioiden on tarkoitus mallintaa ohjelmistossa. Vastuualueeseen kuuluvat luokan tarjoamat **palvelut** (jotka ovat käytettävissä luokan julkisen rajapinnan kautta) sekä luokkaan kuuluvan olion toiminnan ymmärtämisen kannalta oleellinen tilatieto eli **attribuutit**. Käytännön luokissa tilatietoon kuuluu usein muutakin kuin yksittäisiä ohjelmointikielen muuttujia. Luokka voi muun ohella omistaa toisia olioita (esim. Päiväys voi sisältyä jonkin toisen luokan tilaan).

Erityisesti kannattaa ottaa huomioon, että suunnitteluvaiheessa on tarkoitus kirjata vain "julkisen" toiminnallisuuden ymmärtämisen kannalta oleellista informaatiota. Esimerkiksi näyttölaitteella olevaa grafiikkapistettä kuvaava luokka sisältää suunnitteluvaiheessa attribuutin "sijainti", ja tätä tietoa voidaan käsitellä rajapinnan tarjoamien palveluiden avulla. Suunnitteluvaiheessa *ei* ole oleellista määritellä sitä, onko järjestelmässä sijainti-informaation toteutus x- ja y-koordinaatit kokonaislukuina vaiko esimerkiksi polaarikoordinaatiston kul-

ma- ja sädetiedot. Tämä sijaintiattribuutin toteutus on luokan sisäinen asia, ja siihen liittyvät päätökset tulisi tehdä luokan toteutusvaiheessa. (Edelleen käytännön työssä on usein hyödyllistä miettiä myös toteutukseen liittyviä asioita suunnittelun aikana, mutta niiden ei tulisi päästä hallitsemaan ja sotkemaan ylemmän tason suunnittelua.)

5.2.2 Kuinka löytää luokkia?

Helpommin sanottu kuin tehty. Useat oliomenetelmät lähtevät ongelman kuvauksesta. Ohjelmistosuunnittelun käynnistyessä pitäisi olla olemassa vähintään toiminnallinen kuvaus siitä, mitä ollaan tekemässä (sama pätee kaikkeen insinööritoimintaan [Adams, 1991]). Tämän niin sanotun määrittelyvaiheen dokumentista etsitään (vaikkapa alleviivaamalla) substantiivit, joista sitten *kenties* tehdään olioita. Yksi mahdollinen lähtökohta tälle "oliometsästykselle" on **käyttötapaukset** (*use case*), [Jacobson *ja muut*, 1994], joissa on pyritty kuvaamaan yksittäinen ohjelmiston osa **käyttäjäroolin** (*actor*) näkökulmasta. Käyttötapaus pyrkii kuvaamaan lyhyesti, mutta mahdollisimman kattavasti, yksittäiset ohjelmiston toimintaan liittyvät osat. (Näyttävät keskenään ristiriitaisilta vaatimuksilta — ja ovatkin sitä!) Näistä "näytellyistä" ohjelman tärkeimmistä toiminnoista muodostuu määrittely koko ohjelmiston toiminnallisista vaatimuksista. Kuvassa 5.3 seuraavalla sivulla on esimerkki kirjaston tietojärjestelmän yhdestä käyttötapauksesta.

Substantiivien hakumenetelmän hyvänä puolena on, että mallinnettavan ongelman alueelta otetaan sopivan kokoisia kokonaisuuksia ohjelman rakenteeseen. Toisaalta ohjelma tarvitsee luultavasti toimiakseen muitakin osia kuin ne, jotka löytyvät suoraan ongelman kuvauksesta. Emme voi siis olla varmoja, että saamme laadukkaan oliosuunnitelman vain pelkän määrittelyn substantiivien avulla. Esimerkkinä käyttämämme kirjaston toiminnallisuuden kuvauksessa esiintyy varmasti tietoa lainausajoista ja muista vastaavista ajankohdista ja ajanjaksoista. Pelkällä substantiivihaulla emme silti välttämättä keksi, että saatamme tarvita päiväyspalveluita varten oman olion tai moduulin ohjelmistoomme.