Sampo Kuusi

UUDELLEENKÄYTETTÄVÄT KOMPONENTIT OHJELMISTOTUOTANNOSSA

Tietojärjestelmätieteen
pro gradu -tutkielma
1.6.1999

Jyväskylän yliopisto
Tietojenkäsittelytieteen laitos
Jyväskylä

TIIVISTELMÄ

Kuusi, Sampo Sakari
Uudelleenkäytettävät komponentit ohjelmistotuotannossa / Sampo Kuusi
Jyväskylä: Jyväskylän yliopisto, 1999
99 s.
Tutkielma

Tässä tutkielmassa tarkastellaan ohjelmistojen rakentamista uudelleenkäytettävistä komponenteista kokoamalla. Tavoitteena on selvittää, mitä komponenteilla varsinaisesti tarkoitetaan, miten ohjelmistoja rakennetaan komponenteista ja mitä vaikutuksia komponenttien käytöllä on ohjelmistotuotantoprosessiin.

Aiheeseen on perehdytty sekä kirjallisuuden sekä osittain myös käytännön kokemuksen pohjalta. Tutkimusmenetelmä on käsitteellisteoreettinen, vaikka jotkin osat ovatkin lähinnä kirjallisuustutkimusta.

Tutkielmassa on esitetty kirjallisuuden perusteella komponenttien määrittelyyn liittyvä luokittelu, sekä komponenttien suhde komponenttimalleihin ja muihin läheisiin käsitteisiin. Lisäksi tutkielmassa verrataan komponenttien sopeuttamiseen käytettäviä menetelmiä ja esitetään arvio niiden vaikutuksesta komponenttipohjaiseen ohjelmistotuotantoprosessiin.

AVAINSANAT: ohjelmistokomponentti, komponenttimalli, ohjelmistotuotanto, uudelleenkäyttö

Kiitokset

Haluan kiittää työnantajaani Infomanager Oy:tä saamastani tuesta sekä ymmärtäväisestä asenteesta pro gradu –tutkielman tekemiseen.

Lisäksi suuri kiitos ohjaajilleni professori Markku Sakkiselle sekä KTM Petri Saloselle heiltä saamistani hyvistä neuvoista ja rakentavasta palautteesta.

1 JOHDANTO 1

2 OHJELMISTOJEN UUDELLEENKÄYTTÖ 3

2.1 OHJELMISTOKRIISI 3
2.2 TAIKALUODIT - VOIKO OHJELMISTOKRIISIÄ RATKAISTA? 5
2.2.1 Oliot ja komponentit 5
2.2.2 Olio-ohjelmoinnin puutteet 6
2.3 OHJELMISTOJEN UUDELLEENKÄYTÖN EDUT 6
2.4 UUDELLEENKÄYTTÖÖN LIITTYVÄT ONGELMAT 9
2.4.1 Tekniset ongelmat 9
2.4.2 Organisationaaliset ongelmat 10

3 MITÄ OHJELMISTOKOMPONENTIT OVAT 12

3.1 KIRJALLISUUDESSA ESITETYISTÄ MÄÄRITELMISTÄ 12
3.1.1 Komponentit uudelleenkäytettävinä rakenteina 14
3.1.2 Komponentit itsenäisinä rakennuspalikoina 16
3.1.3 Superkomponentit 17
3.1.4 Kirjallisuudessa esitettyjen määritelmien puutteita 19
3.2 ESIMERKKEJÄ KOMPONENTEISTA 20
3.3 MÄÄRITELMIEN LUOKITTELUA: ABSTRAKTIT JA KONKREETTISET KOMPONENTIT 21
3.4 KOMPONENTTIA LÄHELLÄ OLEVIA KÄSITTEITÄ 23
3.4.1 Olioluokat ja oliot 23
3.4.2 Suunnittelumallit 25
3.4.3 Sovelluskehykset 26
3.4.4 Arkkitehtuurit 28
3.5 KIRJALLISUUDESSA ESITETTYJÄ KOMPONENTTIEN LUOKITTELUJA 29
3.5.1 Sametingerin taksonomia 29
3.5.2 Boochin taksonomia 30
3.5.3 Haikalan ja Märijärven taksonomia 31
3.5.4 Chappellin taksonomia 31
3.5.5 Hoffmannin taksonomia 32
3.5.6 Kainin jaottelu 33
3.6 YHTEENVETO OHJELMISTOKOMPONENTTEIHIN LIITTYVÄSTÄ KÄSITTEISTÖSTÄ 33

4 KOMPONENTTIMALLIT 35

4.1 MITÄ KOMPONENTTIMALLILLA TARKOITETAAN 35
4.1.1 Komponenttimallin määritelmä 35
4.1.2 Komponenttimallin tarjoamat palvelut 36
4.2 TÄRKEIMMISTÄ NYKYISISTÄ KOMPONENTTIMALLEISTA 37
4.2.1 COM 37
4.2.2 CORBA 39
4.2.3 JavaBeans ja Enterprise JavaBeans 43
4.2.4 Yhteenveto komponenttimallien eroista 46

5 KOMPONENTTITEKNOLOGIASTA 48

5.1 KOMPONENTTIEN HANKINTA ... 48
5.1.1 Komponenttivarastot ... 48
5.1.2 Komponenttien arviointi ... 49
5.1.3 Komponenttien kuvaaminen rajapintojen avulla ... 50
5.1.4 Rajapinnat ja versiointi ... 51
5.2 KOMPONENTTIEN SOPEUTTAMINEN ... 52
5.2.1 Sopeuttamistekniikat ... 54
5.2.2 Mukauttaminen ja muokkaus ... 54
5.2.3 Leikkaa-liimaa-tekniikka ... 55
5.2.4 Periyttäminen ... 56
5.2.5 Periyttämiseen liittyvät ongelmat ... 62
5.2.6 Implisiittinen alityypitys ... 65
5.2.7 Ympäröiminen ... 66
5.2.8 Sekoitteiden käyttö ... 68
5.2.9 Binäärinen sopeuttaminen ... 69
5.2.10 Superimpositio ... 70
5.3 KOMPONENTTIEN YHDISTÄMINEN ... 71
5.3.1 Sisäinen ja ulkoinen kompositio ... 71
5.3.2 Komposition tasot Sametingerin mukaan ... 72
5.3.3 Visuaalinen kompositio ... 75
5.4 YHTEENVETO KOMPONENTTITEKNOLOGIOISTA ... 77

6 KOMPONENTTIPOHJAINEN OHJELMISTOTUOTANTO ... 80

6.1 SIIRTYMINEN KOMPONENTTIPOHJAISEEN OHJELMISTOTUOTANTOON ... 80
6.1.1 Menetelmätuki ... 81
6.1.2 Työkalutuki ... 81
6.2 OHJELMISTOTUOTANNON KAHTIAJAKO ... 82
6.3 KOMPONENTTITUOTANTO ... 83
6.3.1 Milloin komponentti kannattaa tuottaa itse? ... 84
6.3.2 Uudelleenkäytettävien komponenttien tuottamisesta ... 85
6.4 SOVELLUSTUOTANTO ... 85
6.4.1 Perinteinen vesiputousmalli ... 86
6.4.2 Kehittäminen uudelleenkäyttöä hyväksikäyttäen ... 87
6.4.3 Uudelleenkäyttöpohjainen kehittäminen ... 89
6.4.4 Spiraalimalli ... 90
6.5 VAIKUTUS ELINKAARIKUSTANNUKSIIN ... 91
6.6 YHTEENVETO KOMPONENTTIPOHJAISESTA OHJELMISTOTUOTANNOSTA ... 92

7 YHTEENVETO ... 94

LÄHTEET ... 96

1 JOHDANTO

Komponenttipohjainen ohjelmistotuotanto ja ohjelmointi ovat astuneet 1990-luvun jälkimmäisellä puoliskolla oliomenetelmien ja olio-ohjelmoinnin tilalle alan lehdistön huomion keskipisteeksi. Tästä huolimatta alalla ei vallitse yksimielisyyttä edes komponentteihin liittyvästä peruskäsitteistöstä - mitä komponentit itse asiassa ovat ja miten ne eroavat olioista? Vai onko kyseessä sama asia, jota vain tällä kertaa myydään eri nimellä? Koska puhuttaessa komponenteista tarkoitetaan hyvinkin erilaisia asioita, on komponenttien merkityksestä ja käytöstä esitetty monenlaisia arvioita.

Tämän tutkielman tavoitteena on selvittää, mitä ohjelmistokomponentit täsmällisesti ottaen ovat ja miten ohjelmistoja rakennetaan komponenteista. Lisäksi pyritään selvittämään, mitä vaikutuksia komponenttien käytöllä on ohjelmistotuotantoprosessiin. Tutkimusmenetelmä on käsitteellisteoreettinen. Tutkimusaineisto koostuu sekä kirjallisuudesta että kirjoittajan omista kokemuksista.

Tutkielman luvussa 2 käsitellään ohjelmistojen uudelleenkäyttöön liittyvää tutkimusta, joka on ohjelmistokomponenttien taustalla. Luvussa 3 käsitellään kirjallisuudesta löytyviä komponentin määritelmiä ja pyritään luokittelemaan niitä järkevästi. Lisäksi käsitellään joitain komponentteihin liittyviä käsitteitä sekä komponenteille tehtyjä taksonomioita. Luvussa 4 määritellään komponenttimallin käsite, joka liittyy keskeisesti nykyisiin komponentteihin. Lisäksi esitellään lyhyesti COM, CORBA ja JavaBeans-komponenttimalleja ja niiden välisiä eroja. Luvussa 5 käsitellään komponenttien hankintaa, sopeuttamista sekä yhdistämistä. Luvussa esitetään myös yhteenveto eri sopeuttamistekniikkojen eroista ja yhtäläisyyksistä.

Luvussa 6 käsitellään aikaisemmissa luvuissa esitettyjen asioiden vaikutusta koko ohjelmistotuotantoprosessiin. Luku 7 on tutkielman yhteenveto.

2 OHJELMISTOJEN UUDELLEENKÄYTTÖ

2.1 Ohjelmistokriisi

Ohjelmistotuotanto (software engineering) on muihin insinööritieteisiin verrattuna suhteellisen nuori ala. Tästä johtuen se ei ole ainakaan vielä saavuttanut samanlaista kypsyyttä, johon muut insinööritieteet ovat päässeet. Ohjelmistotuotanto muistuttaakin monessa suhteessa enemmän käsityötä tai jopa taidetta kuin teollista toimintaa. Kuitenkin rakennettavat ohjelmistot ovat kooltaan usein suuria, suurimmillaan jopa miljoonien ohjelmarivien pituisia [Haikala & Märijärvi 1997].

Vaikka tietotekniikan kehitys on ollut viimeisten vuosikymmenien aikana nopeaa, ohjelmistotuotantoa ovat koko ajan vaivanneet samankaltaiset ongelmat. Ohjelmistuotannon erityisongelmiin kiinnitettiin virallisesti huomiota ensimmäisen kerran 1960-luvun lopussa, jolloin NATO:n seminaarissa alettiin puhua ns. *ohjelmistokriisistä* (software crisis). Ohjelmistokriisillä tarkoitetaan ohjelmistojen tuottamiseen liittyviä erityisriskejä, jotka seuraavat tarvittavien resurssien arvioimisen vaikeudesta ja tuloksena syntyvän ohjelmiston vaihtelevasta laadusta. Ohjelmistotuotantoprosessin onnistuminen näyttäisi lisäksi riippuvan huomattavan paljon yksittäisten suunnittelijoiden kyvyistä [Haikala & Märijärvi 1997] [Gamma, Helm, Johnson & Vlissides 1995].

On arveltu ohjelmistotuotantoon liittyvien ongelmien johtuvan ainakin osaksi siitä, että muilla teollisuudenaloilla kyetään käyttämään hyväksi aikaisemmin tehtyä työtä, kun taas ohjelmistotuotannossa joudutaan jatkuvasti keksimään samoja asioita uudelleen. Erityisen usein ohjelmistotuotantoa on verrattu elektroniikkasuunnitteluun, jossa elektroniikkapiirien suunnittelu perustuu pitkälti valmiiden komponenttien hyväksikäyttöön. Jo

1980-luvulla Brad Cox esitti ohjelmistojen kokoamista uudelleenkäytettävistä ohjelmistokomponenteista. McIlroy kirjoitti ohjelmistokomponenttimarkkinoista jo vuonna 1969 [Sametinger 1997]. Komponenttipohjaisen ohjelmistotuotannon edut olisivat kiistattomat, mutta siitä huolimatta ohjelmistoja rakennetaan edelleen perinteisellä tavalla.

Toisaalta ohjelmistoilla on joitain perustavanlaatuisia eroja muiden teollisuudenalojen tuotteisiin verrattuna. Frederick Brooksin [Brooks 1987] mukaan tällaisia eroja ovat mm. ohjelmistojen luontainen *monimutkaisuus*, *näkymättömyys* ja *muunnettavuus*. Haikala ja Märijärvi [Haikala & Märijärvi 1997] lisäävät tähän vielä *ainutkertaisuuden*, *menetelmien skaalautumattomuuden* ja ohjelmistoihin perustuvien järjestelmien tietynlaisen *epäjatkuvuuden*. Clemens Szyperski [Szyperski 1998] huomauttaa, että kun muiden teollisuudenalojen tuotteiden toiminta perustuu fysikaalisiin tai kemiallisiin ilmiöihin, pohjautuu ohjelmistojen toiminta puhtaaseen matematiikkaan.

Ohjelmistojen luontainen monimutkaisuus tarkoittaa sitä, että ohjelmistot pyrkivät tyypillisesti ratkaisemaan reaalimaailmassa esiintyviä, hyvin laajoja ja monimutkaisia ongelmia. Näkymättömyys aiheuttaa ongelmia resurssien kulutuksen arvioinnin kannalta, koska rakennettavan ohjelmiston valmiusastetta on vaikea arvioida. Muunnettavuus tuottaa Haikalan ja Märijärven mukaan ongelmia erityisesti siksi, että ohjelmistoja on *näennäisen* helppo muuttaa. Ainutkertaisuus seuraa siitä, että - toisin kuin esimerkiksi siltaa rakennettaessa - samankaltaista ohjelmaa ei välttämättä ole koskaan ennen rakennettu. Skaalautumattomuus tarkoittaa sitä, että aiemmin hyväksi havaitut menetelmät eivät aina toimi projektin koon kasvaessa. Epäjatkuvuus tarkoittaa ohjelmien diskreettiä luonnetta - talo ei sorru yhden naulan katketessa, mutta avaruussukkula voi pudota, jos sitä ohjaavassa ohjelmistossa yhden muuttujan arvo on hetkellisesti väärä.

2.2 Taikaluodit - voiko ohjelmistokriisiä ratkaista?

Frederick Brooksin mukaan ennalta mainituista syistä johtuen ei ole odotettavissa mitään uutta menetelmää ("taikaluotia"), joka voisi ratkaisevasti parantaa ohjelmistotuotannon tuottavuutta tai luotettavuutta. Aikaisemmat ohjelmistotuotannon laatua ja tehokkuutta parantaneet keksinnöt, kuten korkean tason kielet, moniajo tai integroidut ohjelmointiympäristöt, eivät ole korjanneet ohjelmistotuotannon luonteesta johtuvia (essential) vaan voitettavissa olevia (accidental) ongelmia.

2.2.1 Oliot ja komponentit

Brooksin mukaan lupaavin ehdokas taikaluodiksi on olio-ohjelmointi (erityisesti abstraktit datatyypit ja hierarkkiset tyypit), mutta sekään ei vielä riitä parantamaan ohjelmistotuotannon tuottavuutta tai luotettavuutta riittävästi [Brooks 1987]. Jotkut tutkijat, kuten Szyperski kuitenkin näkevät komponenttipohjaisen ohjelmoinnin (COP, Component-Oriented Programming) ja komponenttipohjaisen ohjelmistonkehityksen (CBD, Component-Based Development) ohjelmistotuotannon seuraavana kehitysaskeleena olioparadigman jälkeen [Szyperski 1998]. Komponenttien käytön on väitetty parantavan ohjelmistotyön laatua ja nostavan tuottavuutta [Pree 1997] [Chappell 1997] [Veryard 1997]. Kirjallisuudessa ei kuitenkaan vielä vallitse edes yksimielisyyttä siitä, mitä oikeastaan tarkoitetaan ohjelmistokomponentilla. Laajimpien määritelmien mukaan kyse on mistä tahansa moduulista, tiukoimpien määritelmien mukaan ohjelmistokomponenttien käyttö ei ole edes vielä mahdollista nykyisillä ohjelmointikielillä ja välineillä.

2.2.2 Olio-ohjelmoinnin puutteet

Kun olio-ohjelmointi löi itsensä 1980-luvun lopussa läpi, odotettiin että seurauksena olisi uudelleenkäytön määrän moninkertaistuminen ohjelmistoprojekteissa. Näin ei kuitenkaan käynyt odotetussa määrin, ja onkin usein todettu, että olio-ohjelmointi ei ole täyttänyt uudelleenkäyttöön liittyviä lupauksiaan [Brown & Short 1997] [Pree 1997] [Szyperski 1995] [Jaaksi 1998]. Eräänä syynä olio-ohjelmoinnin epäonnistumiseen on esitetty mm. toteutusperinnän (implementation inheritance) liiallista käyttöä, koska tämä sotii kapselointiperiaatetta vastaan paljastamalla aliluokille yliluokkien toteutuksen. Lisäksi toteutuksen perintä tekee ohjelmakoodin toiminnasta vaikeampaa ymmärtää [Szyperski 1998] [Franz]. Puhe epäonnistumisesta liittynee kuitenkin enemmän liiallisiin odotuksiin kuin olioparadigman puutteisiin.

Uudelleenkäytettävistä komponenteista on puhuttu jo 1980-luvulla, mutta ne ovat tulleet voimakkaasti esiin vasta viime vuosina. Alan lehdistön kiinnostus on siirtynyt olioista komponentteihin. Kuvaavaa on, että entinen Object Magazine vaihtoi vuonna 1998 uudeksi nimekseen Component Strategies. Syitä viimeaikaiseen kiinnostuksen kasvuun ovat olleet mm. standardoitujen komponenttimallien yleistyminen (esim. COM, CORBA ja JavaBeans), olio-ohjelmoinnin tuomat mahdollisuudet, parantunut työkalutuki (esim. kaupallisesti hyvin menestynyt Visual Basic-ympäristö), ohjelmistojen elinkaaren lyheneminen sekä ohjelmistojen koon kasvaminen.

2.3 Ohjelmistojen uudelleenkäytön edut

Ohjelmistojen uudelleenkäyttö (software reuse) voidaan määritellä ohjelmistojen rakentamiseksi olemassaolevien ohjelmistojen pohjalta sen sijaan, että uudet ohjelmistot rakennettaisiin alusta alkaen itse [Sametinger 1997]. Uudelleenkäyttö voidaan jakaa *generatiiviseen* ja *koostavaan* (composi-

tional) uudelleenkäyttöön. Generatiivisessa uudelleenkäytössä työkalu generoi ohjelman tai sen osan sille annetun määrittelyn mukaan. Esimerkkejä tällaisista työkaluista ovat Lex ja Yacc. Ohjelmistojen kokoaminen komponentteja yhdistämällä liittyy puolestaan koostavaan uudelleenkäyttöön. Tästä syystä tässä tutkielmassa käsitellään vain koostavaa uudelleenkäyttöä.

Ohjelmoijat ovat jo kauan käyttäneet olemassaolevaa ohjelmakoodia uudelleen. Uudelleenkäyttö on kuitenkin pitkään ollut epäformaalia, ja vasta 1980-luvulla yrityksissä ryhdyttiin laajoihin ja järjestelmällisiin uudelleenkäyttöä edistäviin hankkeisiin [Sametinger 1997].

Tutkimukset ovat näyttäneet, että olemassaolevasta ohjelmistokoodista 40%-80% on uudelleenkäytettävää sovelluksesta toiseen [Sametinger 1997] [Haikala & Märijärvi 1997]. Vain 15% ohjelmistokoodista on täysin sovelluskohtaista [Tracz 1987].

Ohjelmistojen uudelleenkäyttö parantaa syntyvien ohjelmistojen laatua ja prosessin tuottavuutta [Sametinger 1997]. Laatu paranee, koska virheiden etsimiseen on käytetty aikaa jo aikaisemmin. Tuottavuus paranee, koska ohjelmistoja ei tarvitse rakentaa alusta asti itse. Muita etuja ovat mm. redundantin työn väheneminen, toiminnallisuuden yhtenäisyys (tällä on merkitystä erityisesti käyttöliittymien ohjelmoinnissa) ja kokeneiden suunnittelijoiden työstä oppiminen [Sametinger 1997][Haikala & Märijärvi 1997].

Ohjelmistonkehitys nopeutuu kun kehitettävät ohjelmistot koostuvat komponenteista, joita voidaan kehittää rinnakkain [Krajnc 1997] [Hannula 1998]. Työnjako voidaan organisoida komponenttien rajojen mukaan, esim. Ari Jaaksin mukaan komponenttipohjainen ohjelmistotuotanto tulisi organisoida siten, että yksi ohjelmistosuunnittelija voisi vastata yhden ohjelmistokomponentin kehityksestä ja ylläpidosta [Jaaksi 1998].

ISO 9126-standardi määrittelee ohjelmistojen laadun koostuvan seuraavista kuudesta ominaisuudesta: toiminnallisuus, ylläpidettävyys, käytettävyys, tehokkuus, luotettavuus ja siirrettävyys. Voidaan väittää, että komponenttipohjainen ohjelmistonkehitys auttaa kehittämään jokaista näistä laadun osa-alueista. Uudelleenkäytön lisääntyessä uutta toiminnallisuutta voidaan luoda nopeammin. Ylläpito helpottuu, kun modulaarinen rakenne mahdollistaa komponenttien korvaamisen toisilla. Käytettävyys parantuu, kun standardikomponenttien käyttö yhtenäistää käyttöliittymien ulkoasua. Tehokkuus parantuu, kun optimointi voidaan kohdistaa suorituskyvyn kannalta kriittisiin komponentteihin. Luotettavuus parantuu, kun voidaan luottaa siihen, että komponentit toimivat rajapintansa mukaisesti (joskin se, että kaikki komponentit toimivat luotettavasti, ei vielä takaa kokonaisen järjestelmän luotettavaa toimintaa). Järjestelmistä tulee myös siirrettävämpiä, jos komponenttien kehittämisessä käytetään ympäristöstä riippumattomia tekniikoita [Veryard 1997]. Voidaan havaita, että kaikki nämä edut seuraavat ylempänä esitetyistä ohjelmistojen uudelleenkäytön eduista: tuottavuuden ja laadun kohoamisesta.

Komponenteista koostuva ohjelmisto on joustavampi kuin perinteinen sovellus, koska toiminnallisuutta voidaan nopeasti muuttaa vaihtamalla yksi tai useampi komponenteista. Ohjelmiston joustavuus nopeuttaa organisaation toimintaympäristössä tapahtuviin muutoksiin reagoimista. Joustamaton ohjelmisto saattaa estää liiketaloudellisesti järkevien muutosten toteuttamisen.

Perinteisellä tavalla rakennetut ohjelmistot ovat sidottuja tiettyihin ohjelmointikieliin, käyttöjärjestelmiin, verkkoprotokolliin, ohjelmointirajapintoihin ja kehitysvälineisiin. Komponenttimallit kuten CORBA ja COM tuovat mukanaan ainakin osittaisen riippumattomuuden ohjelmointikielistä ja käyttöjärjestelmistä.

Eräs tietohallinnon perusongelmia on valinta itse tehdyn ja markkinoilta ostetun ohjelmiston välillä ("build vs. buy"). Itse tehty ohjelmisto on joustavampi ja huomioi paremmin paikalliset tarpeet, mutta maksaa selvästi enemmän. Markkinoilta ostettava pakettiohjelmisto on halvempi[1], mutta joustamattomampi organisaation tarpeiden kannalta. Szyperskin mukaan osaksi markkinoilta ostetuista ja osaksi itse tehdyistä komponenteista rakennettava ohjelmisto on välimuoto näiden äärimuotojen välillä ja useissa tapauksissa lähempänä optimia kustannustehokkuuden ja joustavuuden kannalta [Szyperski 1998].

Ohjelmistojen uudelleenkäyttö auttaa organisaatioita keskittymään ydinosaamiseensa. Ne ohjelmistokomponentit, jotka eivät suoranaisesti liity organisaation harjoittamaan liiketoimintaan, voidaan ostaa markkinoilta.

2.4 Uudelleenkäyttöön liittyvät ongelmat

Koska ohjelmistojen uudelleenkäytöstä ja komponenttien käytöstä on selviä etuja, on syytä kysyä miksi ohjelmistoja kehitetään vielä vanhoilla menetelmillä. Tämä johtuu siitä, että uudelleenkäyttöön liittyy vielä paljon ratkaisemattomia ongelmia. Esiintyvät ongelmat voidaan jakaa teknisiin ja organisationaalisiin.

2.4.1 Tekniset ongelmat

Teknisiä ongelmia ovat mm. sopivien komponenttien löytäminen, komponenttikirjaston perustaminen, komponenttien sopeuttamisen hankaluus ja komponenttien integrointiin liittyvät vaikeudet. Uudelleenkäytettävien oh-

jelmistokomponenttien luomisen on myös havaittu vaativan enemmän panostusta kuin vastaavan ei-uudelleenkäytettävän toiminnallisuuden luomisen [Sametinger 1997] [Haikala & Märijärvi 1997]. Ilman erillistä panostusta projektien sivutuotteena ei tavallisesti synny uudelleenkäytettäviä komponentteja.

Useat tutkijat, mm. Szyperski, pitävät eräänä suurimpana haasteena komponenttien välisten rajapintojen kuvauksien kehittämistä. Tällä hetkellä rajapinnat kuvaavat tyypillisesti vain joukon metodeja, joiden kuvaukset koostuvat nimestä, parametreistä ja palautustyypistä. Jotta rajapinta kertoisi enemmän komponentin toiminnasta, tulisi sen kyetä kuvaamaan komponentin käyttäytymistä nykyistä paremmin. Lisäksi rajapinnan tulisi kuvata myös komponentin toiminta uudelleenkutsujen yhteydessä (reentrance conditions) sekä ei-toiminnallisia ominaisuuksia, kuten ajan ja muistin kulutusta [Szyperski 1998]. Rajapintojen kuvauksen kehittyminen helpottaisi myös komponenttien etsimiseen liittyviä ongelmia.

2.4.2 Organisationaaliset ongelmat

Yleisimpiä organisationaalisia ongelmia ovat johdon tuen puute, perinteiset ohjelmistoprosessit, haluttomuus käyttää muiden tekemiä komponentteja (ns. "not invented here"-syndrooma) ja kannustimien puute. Perinteisillä ohjelmistoprosesseilla tarkoitetaan tässä sitä, ettei käytössä olevia ohjelmistoprosessin elinkaarimalleja ole suunniteltu uudelleenkäyttöä silmällä pitäen. Kannustimien puute tarkoittaa sitä, että suunnittelijoita palkitaan usein yksittäisten projektien tulosten perusteella, jolloin heidän ei välttämättä kannata käyttää resursseja komponenttien uudelleenkäytettävyyden parantamiseen [Haikala & Märijärvi 1997] [Pree 1997].

---

[1] Jos voidaan ostaa sellaisenaan, ilman räätälöintitarvetta.

Komponenttien ostamiseen markkinoilta liittyy myös erityisiä riskejä: komponentin valmistaja saattaa lopettaa toimintansa, lopettaa komponentin kehityksen tai tuki voi olla huonoa [Salonen 1998].

Universaalisti toimivien komponenttien rakentamista hankaloittaa se, että markkinoilla on useita kilpailevia komponenttimalleja. Mm. Orfali ym. pitävät *universaalin komponenttimallin* muodostumista edellytyksenä toimivien ohjelmistokomponenttimarkkinoiden muodostumiseksi [Orfali, Harkey & Edwards 1996]. Toisaalta komponenttimallien välinen kilpailu pakottaa komponenttimalleja kehittymään.

3 MITÄ OHJELMISTOKOMPONENTIT OVAT

Kirjallisuudessa on havaittavissa useita erilaisia näkemyksiä siitä, mitä ohjelmistokomponentit ovat. Kappaleessa 3.1 esitetään erilaisia kirjallisuudessa esitettyjä ohjelmistokomponentin määritelmiä ja luokitellaan ne kolmeen luokkaan. Kappaleessa 3.2 esitetään joitain esimerkkejä tyypillisistä ohjelmistokomponenteista eri määritelmätyyppien valossa. Kappaleessa 3.3 esitetään määritelmätyyppeihin kohdistuva luokittelu.

Viime aikoina on puhuttu paljon myös sovelluskehyksistä, suunnittelumalleista ja arkkitehtuureista samoissa yhteyksissä joissa on käsitelty komponenttipohjaista ohjelmointia ja ohjelmistotuotantoa. Lisäksi vallitsee edelleen epäselvyyttä siitä, mitä eroa komponenteilla ja olioilla on, vai onko kyseessä sama asia uudella nimellä. Kappaleessa 3.4 käsitellään näitä komponentteja enemmän tai vähemmän lähellä olevia käsitteitä.

Sen lisäksi, että kirjallisuudessa on esitetty toisistaan poikkeavia määritelmiä komponenteille, ovat eri kirjoittajat pyrkineet luokittelemaan komponentteja erilaisin perustein. Kappaleessa 3.5 käsitellään näitä luokitteluja eli taksonomioita. Taksonomioista käy ilmi, että kirjoittajat ovat lähestyneet aihetta hyvinkin erilaisista näkökulmista.

3.1 Kirjallisuudessa esitetyistä määritelmistä

Erään näkemyksen mukaan (esim. [Sametinger 1997]) ohjelmistokomponentit ovat mitä tahansa selkeästi tunnistettavissa olevia ohjelmiston osia, joita voidaan uudelleenkäyttää uusien ohjelmistojen rakentamisessa. Ohjelmistokomponentti olisi näin ollen mikä tahansa uudelleenkäytettävä rakenne, esimerkiksi luokka, funktio tai makro. Ohjelmistojen uudelleen-

käyttöön keskittyvän tutkimuksen parissa myös ideoita, kuten suunnittelumalleja ja ohjelmistotuotannossa syntyvää dokumentaatiota, kuten vaatimusmäärittelyjä voidaan pitää komponentteina.

Toisen näkemyksen mukaan (esim. [Szyperski 1998], [Pree 1997] ja [Vanhelsuwé 1997]) komponenttipohjainen ohjelmointi on ohjelmoinnin kehityksessä seuraava paradigma olio-ohjelmoinnin jälkeen. Ohjelmoinnin historiassa on havaittavissa, että uudet menestykselliset ohjelmointimenetelmät käyttävät korkeamman tason abstraktioita kuin aikaisemmat menetelmät. Tietokoneita ohjelmoitiin alunperin suoraan prosessorin ymmärtämällä konekielellä. Ensimmäiset korkean tason ohjelmointikielet sallivat siirtymisen lauseesta toiseen hyppykäskyillä. Seuraava vaihe oli strukturoitu ohjelmointi, jossa ohjelma koostui peräkkäisistä, ehdollisista ja toistettavista lauseista. Modulaarinen ohjelmointi yhdisti toisiinsa tiedon ja sitä käsittelevät toiminnot. Olio-ohjelmointi lisäsi tähän perinnän ja polymorfismin, jolloin viestin lähettäjän ja vastaanottajan välinen sidos saatiin löyhemmäksi. Komponenttipohjainen ohjelmointi nostaa abstraktiotasoa edelleen. Ohjelmistokomponentti on käsitteenä analoginen olioluokan kanssa, mutta korkeammalla tasolla.

Edelleen jotkut kirjoittajat, kuten Timothy Budd ja Bertrand Meyer pitävät ohjelmistokomponenttia suurin piirtein olioluokan synonyymina [Budd 1997] [Meyer 1997]. Onkin mielenkiintoista tarkastella, mitä eroa komponenttipohjaisella ohjelmoinnilla on olio-ohjelmoinnin kanssa - vai onko kyseessä sama asia uudella nimellä? Kysymykseen palataan kohdassa 3.4.1.

Jotkut korostavat komponenttimallin merkitystä ja ovat sitä mieltä, ettei komponenteista puhuminen ole mielekästä ilman komponenttimallia, joka mahdollistaa komponenttien yhdistämisen standardoimalla komponenttien rajapinnat ja tarjoaa niille erilaisia palveluita.

Joka tapauksessa komponenttipohjainen ohjelmointi pohjautuu hyväksi havaittuihin suunnittelun periaatteisiin: *kerroksellisuuteen*, *modulaarisuuteen* ja *tiedon piilottamiseen*. Tavoitteena on lego-palikoita muistuttava arkkitehtuuri, jossa uusien sovellusten rakentaminen olisi mahdollisimman pitkälle valmiiden komponenttien yhdistelyä.

3.1.1 Komponentit uudelleenkäytettävinä rakenteina

Uudelleenkäyttöön suuntautuneessa tutkimuksessa ohjelmistokomponentti on tyypillisesti laaja käsite, joka sisältää mm. dokumentit ja kaiken lähdekoodin. Joidenkin määritelmien mukaan komponentit voivat olla myös täysin abstrakteja ideoita.

J. Bradford Kain määrittelee ohjelmistokomponentin löyhästi. Hän määrittelee komponenteille seuraavat vaatimukset: komponentin tulee olla suoraan käytettävissä ohjelmistokehityksessä, komponentin tulee olla eksplisiittisesti määritelty ja diskreetti yksikkö, komponentin tulee olla erotettavissa alkuperäisestä kontekstistaan ja käytettävissä muissa konteksteissa [Kain 1998].

Kainin määritelmää voidaan pitää tietynlaisena ylijoukkona, joka sisältää valtaosan muista esitetyistä ohjelmistokomponentin määritelmistä. Kainin määritelmän mukaan komponentti voi olla uudelleenkäytettävä idea, dokumentti tai lähdekoodi- tai binäärimuotoinen rakenne.

Johannes Sametinger määrittelee ohjelmistokomponentin seuraavasti: "*Uudelleenkäytettävät ohjelmistokomponentit ovat itseriittoisia, selkeästi tunnistettavia artifakteja, jotka kuvaavat ja/tai suorittavat tiettyjä tehtäviä, joilla on selkeä rajapinta, riittävä dokumentaatio ja tarkoin määritelty status uudelleenkäytön kannalta.*" [Sametinger 1997]

Itseriittoisuus tarkoittaa Sametingerin mukaan uudelleenkäytettävyyttä ilman, että mukana tarvitaan joitain muita komponentteja. Näin ollen esim. funktio A on komponentti, jos sitä voidaan uudelleenkäyttää sellaisenaan, ilman muita funktioita. Jos funktio A käyttää funktioita B ja C, eivätkä nämä eivät käytä muita funktioita, funktiot A, B ja C muodostavat yhdessä ohjelmistokomponentin, jonka rajapinnan muodostaa funktio A. Sametinger toteaa kuitenkin, ettei aina ole järkevää sisällyttää kaikkia tarvittavia osia ohjelmistokomponenttiin, mutta tällöin on kuitenkin selkeästi ilmoitettava, mitä ulkoisia riippuvuuksia komponentilla on.

Sametingerin määritelmän mukaan komponentin mukana on toimitettava riittävä dokumentaatio, josta selviää, miten komponentti hankitaan varastosta, selviää sen soveltuvuus tietyn uudelleenkäyttökontekstin kannalta, sekä selviää miten sopeuttaa komponentti ja yhdistää se uuteen ympäristöönsä.

Tarkoin määritelty status uudelleenkäytön kannalta tarkoittaa sitä, että komponentin mukana on toimitettava tieto siitä kuka on komponentin omistaja, kuka ylläpitää sitä, keneen voi ottaa yhteyttä ongelmien ilmaantuessa ja mikä on komponentin laatustatus.

Sametinger toteaa ottavansa tarkoituksella mahdollisimman monet olemassaolevat abstraktiot mukaan ohjelmistokomponentin määritelmäänsä, koska järjestelmälliseen uudelleenkäyttöön on ryhdyttävä riippumatta siitä, millä tasolla komponenttiteknologia on tällä hetkellä.

Sametinger kuitenkin toteaa, ettei pidä ideoita kuten algoritmeja ja suunnittelumalleja komponentteina.

3.1.2 Komponentit itsenäisinä rakennuspalikoina

Toiset tutkijat kuitenkin rajaavat ohjelmistokomponentin käsitteen tiukemmin. Jos ohjelmistokomponentit nähdään ensisijaisesti uutena abstraktiona ja komponenttipohjainen ohjelmointi olio-ohjelmoinnin seuraajana, ohjelmistokomponentit määritellään yleensä binäärisiksi, selkeän rajapinnan omaaviksi ohjelmiston osiksi. Esimerkiksi Clemens Szyperski ja Holger Hoffmann ovat esittäneet tarkoin rajattuja määritelmiä.

Szyperski määrittelee ohjelmistokomponentin seuraavasti: "*Ohjelmistokomponentti on kompositioyksikkö, jolla on sopimuksellisesti määritelty rajapinta tai rajapintoja ja vain eksplisiittisesti ilmoitettuja ulkoisia riippuvuuksia. Ohjelmistokomponentti voidaan julkistaa itsenäisesti ja kehittäjät voivat käyttää sitä rakennuspalikkana.*" [Szyperski 1998]

Komponentin *rajapinnalla* (interface) tarkoitetaan epätäsmällisesti ilmoitettuna pääsyä komponentin tarjoamiin palveluihin. Rajapinnan sopimuksellisuudella tarkoitetaan sitä, että komponentin käyttäjät voivat luottaa komponentin toimivan rajapinnan lupaamalla tavalla. Myös komponentin tulevien versioiden tulisi toimia rajapinnan mukaisesti. Eksplisiittisesti ilmoitetut kontekstiriippuvuudet tarkoittavat komponentin itsensä käyttämiä ulkoisia rajapintoja ja muita riippuvuuksia.

Jotta komponentti olisi itsenäisesti jaeltavissa, sen on oltava hyvin eristetty ympäristöstään ja muista komponenteista. Komponenttia ei voida koskaan jakaa tai asentaa osittain. Komponentin käyttäjät voivat käyttää komponenttia rakennuspalikkana sellaisenaan, eli käyttäjän ei tarvitse nähdä miten komponentti on toteutettu sisäisesti. Komponenttia tarkastellaan siis mustana laatikkona. Szyperski toteaakin, että ohjelmistokomponentit ovat käytännössä aina binäärisiä [Szyperski 1998].

Holger Hoffmannin [Hoffmann 1997] mukaan ohjelmistokomponentti on "*...ohjelmiston osa jolla on yksi tai useampi hyvin määritelty rajapinta, joka on käyttäjän konfiguroitavissa ja integroitavissa muihin komponentteihin, mutta jota käyttäjä ei voi muokata*".

Hoffmannin määritelmä eroaa Szyperskin määritelmässä siten, että Hoffmann ei erikseen vaadi komponentilta ulkoisten rajapintojen riippuvuuden dokumentoimista. Hoffmann vaatii komponenteilta konfiguroitavuutta. Konfiguroitavuus tarkoittaa sitä, että käyttäjä voi asettaa parametreja, jotka vaikuttavat komponentin toimintaan tai ominaisuuksien arvoihin. Konfiguroitavuus ei saa edellyttää komponentin lähdekoodin muuttamista. Komponenttien tulee Hoffmannin mielestä olla konfiguroitavia, jotta niitä voitaisiin sopeuttaa käyttöympäristöönsä ja yhdistää toisiinsa. Hoffmann ei selkeästi erota toisistaan komponentin tarjoamia rajapintoja ja komponentin konfiguroitavuutta, sillä esimerkkinä konfiguroituvuudesta hän antaa Painonappi-komponentin tekstin värin asettamisen [Hoffmann 1997].

3.1.3 Superkomponentit

Jotkut tutkijat vaativat ohjelmistokomponenteilta myös arkkitehtuurillisia ominaisuuksia, kuten riippumattomuuden käytetystä ohjelmointikielestä. Tällaiset piirteet komponentille tarjoaa komponenttimalli, jolloin ohjelmistokomponentilta vaaditaan käytännössä jonkin komponenttimallin tukemista.

Andersen Consultingin omaa komponenttimallia kehittäneet Bronsard ym. asettavat komponenteille seuraavia vaatimuksia: *kontekstiriippumattomuus*, *sijaintiläpinäkyvyys* (location transparency), *heterogeeninen yhdistettävyys* (heterogeneus connection) ja *laajennettavuus* [Bronsard & Bryan & Kozaczynski & Liongosari & Ning & Olafsson & Wetterstrand 1997]. Kon-

tekstiriippumattomuus tarkoittaa että komponenttien tulisi selkeästi ilmoittaa kaikki oletukset joita ne tekevät ympäristöstään. Läpinäkyvyys sijainnin kannalta tarkoittaa, että komponentin ei tulisi olettaa toimivansa missään tietyssä paikassa. Komponenttia tulee voida ajaa saman prosessin eri säikeissä tai eri prosesseissa, jotka on hajautettu tietokoneverkon yli. Heterogeenisella yhteydellä tarkoitetaan sitä, että komponentit riippuvat joka tapauksessa jonkin verran ajonaikaisen ympäristön teknisistä ominaisuuksista, käytetystä ohjelmointikielestä, kommunikaatioprotokollista jne. Komponenttien tulisi olla heterogeenisiä eli tukea mahdollisimman monia eri standardeja. Laajennettavuus tarkoittaa sitä, että komponenttien lisäämisen, poistamisen ja korvaamisen tulisi olla helppoa myös järjestelmän asentamisen jälkeen.

Kontekstiriippumattomuus voidaan määritellä myös tiukemmin, kuten tekevät esim. Ciupke ja Schmidt. Heidän mukaansa kontekstiriippumattomuuteen kuuluu erillinen kääntäminen, riippumattomuus ohjelmointikielestä, tuki rinnakkaisille prosesseille, tuki hajautukselle, versionhallinta ja käyttöjärjestelmäriippumattomuus [Ciupke & Schmidt 1996].

Robert Orfali, Dan Harkey ja Jeri Edwards [Orfali & Harkey & Edwards 1996] ovat esittäneet *superkomponentin* määritelmän. He määrittelevät ensin minimalistisen komponentin, joka vastaa suurin piirtein Szyperskin kohdassa 3.1.2 esittämää määritelmää sillä erotuksella, että minimalistinen komponentti ei saa olla itsessään sovellus ja että minimalistisen komponentin on tuettava perintää ja polymorfismia (näin kirjoittajat sitovat ohjelmistokomponentit oliopohjaiseen paradigmaan). Tämän jälkeen he määrittelevät superkomponentin, joka on minimalistinen komponentti ja tukee lisäksi seuraavia piirteitä: turvallisuus, lisensointi, versiointi, elinkaaren hallinta, avoin tuki työkalupaleteille, tapahtumapohjainen kommunikointi (event notification), konfiguroinnin ja ominaisuuksien hallinta, tuki skriptikielille, metadata ja introspektio, transaktionhallinta ja lukitus, py-

syvyys, suhteiden hallinta (relationships), helppokäyttöisyys, itsensätestaus, semanttinen viestintä ja itseasentuvuus (self-installing) [Orfali & Harkey & Edwards 1996].

Käytännössä uusien ohjelmistokomponenttien tekeminen muodostuisi varsin työlääksi, mikäli jokaiseen komponenttiin olisi erikseen ohjelmoitava tuki yllä luetelluille piirteille. Lisäksi komponenteista tulisi melko kookkaita. Orfalin ym. ajatuksena onkin se, että yllä luetellut palvelut on standardoitu ja superkomponentit pyytävät niitä joltain yleisesti tunnetulta komponentilta, kuten CORBA-komponenttimallin mukaiselta oliokutsuvälittäjältä (ORB).

3.1.4 Kirjallisuudessa esitettyjen määritelmien puutteita

Kirjallisuudessa esiintyneet määritelmät eivät yleensä ole kovin täsmällisiä. Epätäsmällisyyksiä esiintyy tyypillisesti mm. siinä, voivatko komponentit olla itse sovelluksia, täytyykö komponenttien perustua oliopohjaiseen paradigmaan ja täytyykö komponentin sisältää dokumentaatiota.

Yksikään kirjallisuudessa vastaan tullut määritelmä ei ottanut kantaa siihen, millaisia ulkoisia riippuvuuksia ohjelmistokomponentille täsmällisesti ottaen sallitaan. On eri asia, sallitaanko komponentille ulkoinen viittaus tietyn käyttöjärjestelmän tai komponenttimallin tarjoamaan palveluun, tiettyyn rajapintaan vai peräti tiettyyn komponenttiin. Esimerkiksi sallitaanko komponenttien välinen periytyminen, mikäli selkeästi ilmoitetaan tällaisen ulkoisen riippuvuuden olemassaolo?

3.2 Esimerkkejä komponenteista

Tarkastellaan vielä eri määritelmätyyppejä muutamien esimerkkien valossa. Ovatko painonappi (Button)-toiminnallisuuden toteuttava Visual Basic-kontrolli, Microsoft Word, pinon toteuttavan C++-luokan lähdekoodi, C-kielinen kokonaisluvun kertoman laskeva funktio, Java-sovelma tai Tarkkailija (Observer) suunnittelumalli ohjelmistokomponentteja? Taulukossa 1 on esitetty nämä tapaukset eri määritelmätyyppien valossa.

Kandidaatti	Sametingerin määritelmä	Szyperskin määritelmä	Orfalin ym. superkomponentti
COM+ prosessinsisäinen palvelin-DLL, joka sisältää pankkitilin tiedot	on	on	on***
Painonappi-VBX	on	on	ei
MS Word	on	on	ei**
stack.cpp	on	ei	ei
int factorial(int nArg) (lähdekoodimuodossa)	on	ei	ei
MyApplet.class	on	on	ei
Tarkkailija - malli	ei*	ei	ei

TAULUKKO 1 Esimerkkejä ohjelmistokomponenteista eri määritelmien mukaan.

* Sametingerin mukaan suunnittelumallit (design patterns) eivät ole ohjelmistokomponentteja vaan pikemminkin ideoita, joita komponentit toteuttavat.

** MS Word tukee kyllä useita Orfalin ym. superkomponenteilta vaatimia piirteitä, kuten versiointi, lisensointi, skriptikielet, konfigurointi jne. muttei kuitenkaan läheskään niitä kaikkia. Lisäksi MS Word on sovellus.

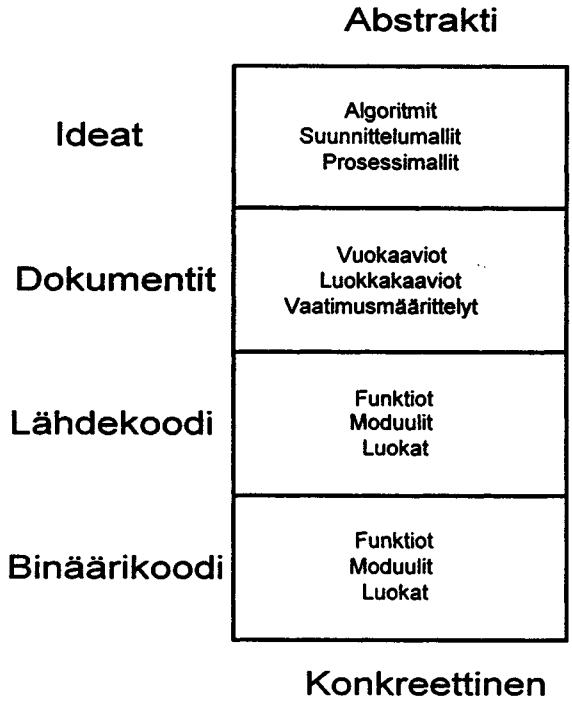
*** Ei tosin täytä Orfalin ym. asettamaa vaatimusta, jonka mukaan komponenttien on tuettava perintää.

3.3 Määritelmien luokittelua: Abstraktit ja konkreettiset komponentit

Ohjelmistokomponenteille esitettyjä määritelmiä voidaan luokitella sen mukaan, miten laajoja ne ovat ts. miten suuren osan potentiaalisista komponenteista ne sisältävät. Yksi dimensio jonka mukaan määritelmän laajuutta voidaan tarkastella on abstraktioaste. Esimerkiksi Kainin [Kain 1998], Hooperin ja Chesterin ja NATO:n määritelmien [Braun 1994] mukaan ohjelmistokomponentteja ovat kaikki ohjelmistotuotannossa syntyvät eristettävät (separable) ja uudelleenkäytettävät tuotteet, kuten suunnittelu(malli), ohjelmakoodi tai dokumentaatio. Useat muut määritelmät sisältävät kuitenkin vain osajoukon tästä. Esimerkiksi Sametingerin määritelmä sulkee pois abstraktit ideat, kuten suunnittelumallit. Edelleen Szyperskin ja Hoffmannin määritelmät sulkevat pois dokumentaation ja lähdekoodin (Szyperski ja Hoffmann eivät toki myöskään katso kaikkea binäärikoodia komponenteiksi).

Kuvassa 1 on esitetty ohjelmistotuotannossa syntyvät tuotteet jaoteltuna neljään luokkaan abstraktioasteen mukaan. Abstrakteimmalla tasolla ohjelmistotuotannossa voi syntyä uusia algoritmeja, suunnittelumalleja ja ohjelmistotuotantoprosessiin liittyviä innovaatioita, joita voidaan käyttää uudelleen tulevissa projekteissa. Tässä ei välttämättä ole kyse ainoastaan täysin uusista ideoista, vaan organisationaalisesta oppimisesta, eli ideoista joita kyseisessä organisaatiossa ei ole aikaisemmin osattu käyttää. Seuraavalla abstraktiotasolla on kyse dokumentaatiosta, jota ohjelmistotuotannossa syntyy. Dokumentaatiota ei tyypillisesti käytetä uudelleen aivan sellaisenaan, vaan usein otetaan vanha dokumentti pohjaksi ja muokataan siitä uusi dokumentti, tai sitten tehdään mallidokumentti (template), jonka mukaan vastaavat dokumentit jatkossa tehdään. Kolmas, edelleen konkreettisempi luokka on lähdekoodi. Konkreettisimmillaan ohjelmistotuotannossa syntyvät tuotteet ovat käännetyssä tai tulkatussa binäärikoodimuodossa.

Esimerkiksi lajittelu voi ideatasolla olla quicksort-algoritmi, dokumentaatiotasolla quicksortin kuvaus, lähdekooditasolla C-kielinen funktio ja binäärikooditasolla joukko konekielisiä käskyjä.



KUVA 1 Ohjelmistotuotannossa syntyvät tuotteet abstraktioasteen mukaan.

Taulukossa 2 on esitetty esimerkkejä komponenteista eri abstraktioasteilla. Taulukossa on esitetty kullekin abstraktioasteelle esimerkki sekä täysin yleiskäyttöisestä että tietylle sovellusalueelle suunnitellusta komponentista.

Abstraktioaste	**Yleiskäyttöinen**	**Sovellusaluekohtainen**
Ideat	Tarkkailija-malli	pivotointialgoritmi
Dokumentaatio	Tuntiraporttipohja	pivotointialgoritmin kuvaus
Lähdekoodi	quicksort.cpp	PieChart.java
Binäärikoodi	Button.class	peliohjaimen ajuri

TAULUKKO 2 Esimerkkejä eri abstraktioasteisista komponenteista.

3.4 Komponenttia lähellä olevia käsitteitä

Komponenttien yhteydessä puhutaan usein olioista, suunnittelumalleista, sovelluskehyksistä sekä ohjelmistojen arkkitehtuureista. Tämän luvun tarkoituksena on käsitellä näitä käsitteitä ja selvittää, miten ne liittyvät komponentteihin.

Erityisesti oliot ja komponentit käsitetään usein samaksi asiaksi. Komponentit ovat viime aikoina astuneet olioiden paikalle alan lehdistössä ja on syytä tutkia, mitä eroa olioilla ja komponenteilla itse asiassa on.

3.4.1 Olioluokat ja oliot

Olioluokka on abstrakti tietotyyppi, joka sisältää toteutuksen, joka voi olla osittainen. Olio on luokan ajonaikainen ilmentymä [Meyer 1997]. Käsitteenä luokka on samankaltainen kuin komponentti. Samankaltaisuus ilmenee erityisesti siinä, että sekä olio- että komponenttipohjainen paradigma painottavat kapselointia, tiedon piilottamista ja monimuotoisuutta [Vayda 1999].

Jotkut kirjoittajat, kuten J. Bradford Kain [Kain 1998] ja Tom Vayda [Vayda 1999] näkevät eron olioluokkien ja komponenttien granulariteetissa. Kun olioluokka voi olla hyvinkin pieni ja kokonaisuuden kannalta merkityksetön osa, on komponentti heidän mukaansa aina merkittävä osa sovellusta. Komponentti voi olla yksi luokka, mutta usein komponentti muodostuu useasta olioluokasta. On huomattava, että olioluokat eivät useinkaan ole itseriittoisia, jolloin ne eivät yksinään muodosta komponenttia [D'Souza & Wills 1998].

Ohjelmistokomponentti ei välttämättä sisällä lainkaan olioluokkia, vaan se voidaan toteuttaa myös perinteisemmillä ohjelmointimenetelmillä [Szy-

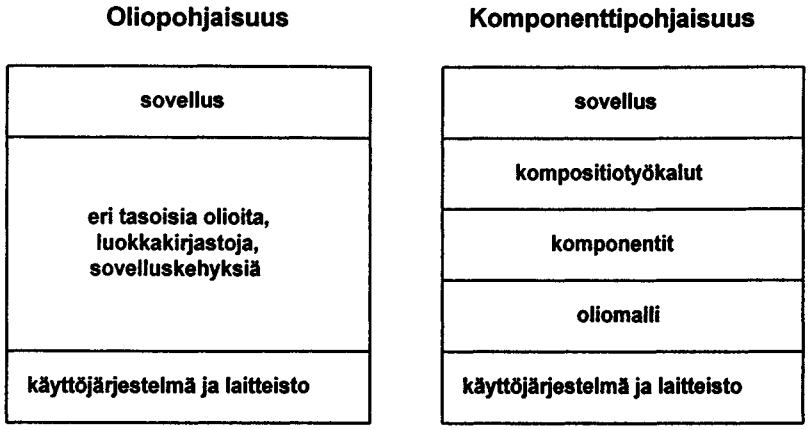
perski 1998] [Kain 1998] [Vayda 1999]. Komponentin rajapinta on määritelty komponenttimallin tai ainakin tietyn ohjelmointikielen asettaman standardin mukaisesti, mutta sisäinen toteutus voi perustua esimerkiksi C-kielisiin funktioihin.

Olioluokkien välillä toteutuksen perintää käytetään yleisesti. Mm. Michael Franzin [Franz] mukaan komponenttien välillä ei tulisi olla periytymissuhteita (erityisesti ei toteutuksen periytymistä). Tällä hetkellä osa tärkeimmistä komponenttimalleista sallii komponenttien välisen toteutusperinnän (esim. JavaBeans), osa estää sen (esim. COM).

Wolfgang Pree [Pree 1997] näkee komponenttipohjaisen ohjelmistonkehityksen olioparadigman seuraajana. Hänen mukaansa komponenttiarkkitehtuuri on selkeämmin *kerroksellinen* kuin olioarkkitehtuuri. Kuvassa 2 on esitetty oliopohjaisen ja komponenttipohjaisen järjestelmän kerrokset Preen mukaan [Pree 1997]. Oliopohjaisessa järjestelmässä sovellus koostuu olioista, joista osa kuuluu erilaisiin luokkakirjastoihin ja sovelluskehyksiin. Osa olioista käyttää suoraan hyväkseen käyttöjärjestelmän tarjoamia palveluita, osa pyytää näitä palveluita toisilta olioilta. Komponenttipohjaisessa järjestelmässä sovellukset koostuvat komponenteista, jotka on liitetty toisiinsa kompositiotyökalujen avulla. Komponentit käyttävät hyväkseen komponenttimallin[2] tarjoamia palveluita. Komponenttimalli taas käyttää hyväkseen käyttöjärjestelmän tarjoamia palveluita. Näin komponentit voidaan ohjelmoida täysin riippumattomiksi alla olevasta käyttöjärjestelmästä, joskin komponenttimallit voivat käytännössä olla sidottuja tiettyyn käyttöjärjestelmään.

---

[2] Mm. lähteissä [Pree 1997] ja [Sametinger 1997] puhutaan komponenttimallin sijasta oliomallista.



KUVA 2 Oliopohjaisen ja komponenttipohjaisen järjestelmän ero. Lähde: Wolfgang Pree [Pree 1997]

3.4.2 Suunnittelumallit

Suunnittelumallit ovat saaneet paljon huomiota osakseen sen jälkeen, kun Gamma ym. julkaisivat aiheesta kirjan (Design Patterns - Elements Of Reusable Object-Oriented Software) vuonna 1995. Gamman ym. [Gamma, Helm, Johnson & Vlissides 1995] mukaan suunnittelumalli (design pattern) koostuu yksikäsitteisestä nimestä, ongelman kuvauksesta, mallin tuomasta ratkaisusta ja ratkaisun tuomista seurauksista mahdollisine kompromisseineen. Ongelman kuvaukseen sisältyvät ne ehdot, joiden on oltava voimassa jotta mallia voidaan käyttää. Ratkaisun kuvaus koostuu mukana olevista elementeistä (Gamman ym. kirjassa olioluokista), niiden rooleista, vastuista ja niiden välisistä suhteista. Ratkaisuna ei anneta konkreettista suunnittelua tai toteutusta. Ratkaisun tuomissa seurauksissa käsitellään ratkaisun tehokkuus suoritusajan ja muistin kannalta, kielikohtaiset seikat ja vaikutukset järjestelmän joustavuuteen, laajennettavuuteen ja siirrettävyyteen.

Joidenkin kirjoittajien mukaan suunnittelumallit ovat sinänsä ohjelmistokomponentteja. Kappaleessa 3.3 esitetyn luokittelun mukaan suunnittelumallit ovat ideoita, joiden sisällyttäminen komponenteiksi johtaa laajimpaan mahdolliseen ohjelmistokomponentin määritelmään. Kuitenkin tiu-

kempien määritelmien mukaan suunnittelumallit vain kuvaavat hyväksi todettuja ratkaisuja komponenttien väliseen kanssakäymiseen.

3.4.3 Sovelluskehykset

Sovelluskehys (framework) on joukko keskenään yhteistoiminnassa toimiva luokkia, joista osa voi olla abstrakteja. Sovelluskehys on usein, muttei aina sovellusaluekohtainen. Osa luokista on abstrakteja ("hot spots") sitä varten, että käyttäjä voisi korvata ne periyttämällä, dynaamisella sidonnalla tai kompositiolla [Szyperski 1998] [Koskimies 1998]. Tyypillisin ratkaisu on antaa sovelluskehyksen käyttäjän periyttää omat konkreettiset aliluokkansa[3]. Sovelluskehys toteuttaa yhden tai useita suunnittelumalleja.

Esimerkkejä tunnetuista sovelluskehyksistä ovat mm. Microsoftin MFC-kirjasto Windows-ohjelmien luomiseen, Inprisen OWL-kirjasto Windows-käyttöliittymien tekemiseen sekä `java.applet`-pakkaus, jota käytetään selaimessa toimivien sovelmien tekemiseen.

Gamman ym. [Gamma, Helm, Johnson & Vlissides 1995] mukaan sovelluskehykset ovat konkreettisempia ja erikoistuneempia elementtejä kuin suunnittelumallit. Sovelluskehykset sisältävät osittaisen toteutuksen, kun taas suunnittelumallit eivät sisällä toteutusta lainkaan. Suunnittelumallit ovat Gamman mukaan myös arkkitehtuurillisesti pienempiä elementtejä kuin sovelluskehykset.

Koska sovelluskehyksiä käytetään rakennuspalikoina uusien ohjelmistojen kehittämisessä ja niillä on tarkoin määritellyt rajapinnat, täyttävät ne osan komponenteilta vaadituista ominaisuuksista. Sovelluskehys voidaan myös jakaa binäärimuotoisena. Toisaalta sovelluskehykset eivät kuitenkaan ole

itseriittoisia, koska niiden käyttäjän on aina lisättävä toiminnallisuutta sovelluskehykseen.

Sovellusten rakentaminen sovelluskehysten ympärille on kuitenkin luonteeltaan erilaista kuin sovellusten rakentaminen erillisiä komponentteja yhdistämällä. Mm. Franz on kritisoinut sovelluskehysten käyttöön perustuvaa lähestymistapaa [Franz]. Hänen kritiikkinsä kohdistuu erityisesti siihen, että sovelluskehykset suunnitellaan yleensä siten, että tietyt luokat jätetään abstrakteiksi ja odotetaan sovelluskehyksen käyttäjän periyttävän niistä oman toteutuksensa. Tällöin luokkien välille syntyy keskinäisten riippuvuuksien muodostama verkko, erityisesti jos käytettävä ohjelmointikieli tukee moniperintää. Luokan toimintaa ei voi täysin ymmärtää ymmärtämättä täysin kaikkien sen yliluokkien toimintaa. Käytännössä sovelluskehykset toimitetaan tästä syystä usein lähdekoodimuodossa - vaikka niitä voi käyttää binäärimuodossa, on toimivan aliluokan luominen periyttämällä usein hyvin vaikeaa ilman tietoa sovelluskehyksen luokkien sisäisestä toteutuksesta.

Franz esittää myös, että sovelluskehyksiä ei tulisi nähdä komponentteina, koska ne eivät täytä vaatimusta jonka mukaan komponentin on oltava korvattavissa toisella saman rajapinnan toteuttavalla komponentilla. Sen sijaan että uudelleenkäytettävä toiminnallisuus pakattaisiin vapaasti vaihdettavissa oleviin komponentteihin, se pakataan yhden ja saman sovelluskehyksen sisään. Sovelluskehys on itsessään suuri ja "monoliittinen" ohjelmisto, joka sisältää paljon sisäisiä riippuvuuksia - ja juuri tästä komponentteja käyttämällä pitäisi päästä eroon.

---

[3] Abstraktien luokkien käyttö ei ole välttämätöntä. Korvattavaksi tarkoitetut metodit voidaan myös määritellä oletuksena tyhjiksi.

3.4.4 Arkkitehtuurit

Arkkitehtuuri kuvaa ohjelmiston kokonaisrakenteen. Arkkitehtuuri voidaan määritellä esim. ohjelmiston keskeisten komponenttien välisten suhteiden ja ohjelmiston rakentamisessa käytettyjen keskeisten periaatteiden kuvaukseksi [Garlan & Perry]. Toisen määritelmän mukaan arkkitehtuuri määrittelee joukon moduuleja ja niiden välisen interaktion rajapintojen, yhteyksien ja rajoitteiden avulla [Luckham, Vera & Meldal 1995]. Rajoitteet ja ohjelmiston rakentamisessa noudatettavat keskeiset periaatteet voivat koskea esimerkiksi ohjelmiston kohdeympäristöä, suorituskykyä, skaalattavuutta, tai laajennettavuutta [Vayda 1999].

Menestyksekkäät arkkitehtuurit ovat yleensä hierarkisia ja kerrostettuja [Szyperski 1998]. Hyvän arkkitehtuurin olisi syytä olla helposti ymmärrettävissä. Ivar Jacobson näkee hyvän arkkitehtuurin ominaisuuksina muunnettavuuden, yksinkertaisuuden, selkeän vastuun jakamisen alijärjestelmien välillä ja osien uudelleenkäytettävyyden [Jacobson 1999].

Kuten suunnittelumallitkin, myös arkkitehtuurit antavat ohjelmistosuunnittelijoille mahdollisuuden kuvata laajaa kokonaisuutta yhdellä sanalla. Hyvä esimerkki tästä on termi asiakas-palvelin-arkkitehtuuri. Toisaalta tämä ei vielä kerro kovin täsmällistä tietoa arkkitehtuurista. Arkkitehtuuria voidaan kuvata tarkemmin esimerkiksi UML-mallin mukaisen arkkitehtuurikuvauksen avulla. Arkkitehtuurikuvaus sisältää käyttötapausten (use cases), analyysimallin, suunnittelumallin, toteutusmallin ja sijoitusmallin (deployment model) "arkkitehtuurillisesti merkittävimmät" osat [Jacobson 1999].

Vaikka arkkitehtuuri periaatteessa onkin ohjelmistotuotannossa syntynyt uudelleenkäytettävä idea, arkkitehtuureja ei laajimpienkaan määritelmien mukaan ole järkevää katsoa komponenteiksi. Sen sijaan arkkitehtuurit ku-

vaavat komponenttien välisiä suhteita, asettavat niille rajoituksia ja sisältävät periaatteita, joiden mukaan ohjelmistoa kehitetään.

3.5 Kirjallisuudessa esitettyjä komponenttien luokitteluja

Ohjelmistokomponenteille on kirjallisuudessa esitetty erilaisia luokitteluja. Luokitteluperusteet vaihtelevat keskenään hyvinkin paljon. Tähän lukuun on pyritty löytämään eri lähtökohdista tehtyjä luokitteluja.

3.5.1 Sametingerin taksonomia

Sametinger jaottelee ohjelmistokomponentteja neljän dimension mukaan, jotka ovat *käyttäjärajapinnan taso, datarajapinnan taso, kompositiorajapinnan taso* ja *komponenttialustan* (component platform) *taso*. Sametinger esittää kullekin dimensiolle numeerisen asteikon, jonka alin taso alkaa nollasta [Sametinger 1997].

Käyttäjärajapinnalla Sametinger tarkoittaa komponentin ja loppukäyttäjän välistä rajapintaa. Datarajapinnalla tarkoitetaan komponentin ja sen käyttämän datan välistä rajapintaa. Kompositiorajapinnalla tarkoitetaan komponentin ja muiden komponenttien välistä rajapintaa. Komponenttialustalla tarkoitetaan kaikkea sitä ohjelmistoa, jota komponentti tarvitsee toimiakseen.

Sametinger jakaa komponenttien tarjoaman käyttäjärajapinnan kolmeen tasoon: ei käyttäjärajapintaa (taso 0) , tekstipohjainen käyttöliittymä (taso 1) ja graafinen käyttöliittymä (taso 2). Datarajapinta jakautuu neljään tasoon: ei datarajapintaa (taso 0), tekstuaalinen datarajapinta (taso 1), tiedostorajapinta (taso 2) ja tietokantarajapinta (taso 3).

Kompositiorajapinta jakautuu kahdeksaan tasoon, jotka ovat tekstuaalinen kompositio, funktionaalinen kompositio, modulaarinen kompositio, oliopohjainen kompositio, alijärjestelmätason kompositio, komponenttimallitason kompositio, tietyn alustan kompositio sekä avoimen alustan kompositio. Komponenttialustat (component platforms) jakautuvat viiteen tasoon, jotka ovat laitteisto (taso 0), käyttöjärjestelmä (taso 1), ohjelmointijärjestelmä (taso 2), kirjastot (taso 3) ja ohjelmointikielet (taso 4). Komposition tasoja ja komponenttialustoja käsitellään tarkemmin kohdassa 5.3.2.

Ohjelmistokomponentit ovat sitä uudelleenkäytettävämpiä, mitä alhaisempi käyttäjä- ja datarajapinnan taso niillä on ja mitä korkeampi kompositiorajapinnan ja komponenttialustan taso niillä on.

Sametingerin taksonomia koskee vain lähdekoodi- tai binäärimuodossa olevia komponentteja, vaikka Sametinger laskee ohjelmistotuotantoprosessiin liittyvät dokumentitkin komponenteiksi.

3.5.2 Boochin taksonomia

Grady Booch [Booch 1987] jakaa ohjelmistokomponentit *rakenteisiin*, *välineisiin* ja *alijärjestelmiin*. Rakenteet ovat abstrakteja tietotyyppejä eli olioita, luokkia tai moduuleja. Rakenteet koostuvat perustietotyypeistä, jotka voivat olla atomisia tyyppejä (bittejä, tavuja, kokonaislukuja jne), koostettuja tyyppejä (taulukot jne) tai osoitintyyppejä. Rakenteet voivat olla monoliittisia (monolithic), jolloin niitä käsitellään aina yhtenä kokonaisuutena tai polyliittisiä (polylithic), jolloin myös niiden osia voidaan käsitellä (esim. puurakenne). Välineet ovat algoritmisia abstraktioita, jotka kohdistuvat rakenteisiin. Alijärjestelmät ovat loogisesti yhteenkuuluvia ja yhdessä toimivia rakenteita ja välineitä. On huomattava, että välineet ovat ideoita, eivätkä siis useimpien määritelmien mukaan komponentteja.

3.5.3 Haikalan ja Märijärven taksonomia

Haikala ja Märijärvi [Haikala & Märijärvi 1997] jakavat ohjelmistokomponentit yleiskäyttöisyyden mukaan *täysin yleiskäyttöisiin*, *sovellusaluekohtaisiin* ja *sovelluskohtaisiin* uudelleenkäytettäviin komponentteihin. Esim. ikkunointiin tarkoitetut komponentit ovat yleiskäyttöisiä ja multidimensionaalisen aineiston pivotointiin kykenevä komponentti sovellusaluekohtainen. On vaikeampaa keksiä esimerkkiä sovelluskohtaisesta komponentista, koska jotta se voitaisiin määritellä komponentiksi se ei saisi olla riippuvainen muista komponenteista ja jos se ei ole riippuvainen muista komponenteista, on vaikea keksiä mitään syytä miksi sen käyttö olisi rajattu tiettyyn sovellukseen. Haikala ja Märijärvi itse antavat esimerkkinä tähän kategoriaan sovelluksen oman käyttöliittymäkirjaston.

Haikalan ja Märijärven esittämä taksonomia on yleiskäyttöinen sikäli, että se soveltuu kaikkiin tässä työssä viitattuihin ohjelmistokomponentin määritelmiin.

3.5.4 Chappellin taksonomia

David Chappell [Chappell 1997] on esittänyt luokittelun asiakas-palvelin-arkkitehtuurin puitteissa toimiville komponenteille. Chappell luokittelee ohjelmistokomponentteja kahden ulottuvuuden mukaan: komponentin itsenäisyys ja asema client/server-arkkitehtuurissa. Näin muodostuu neljä luokkaa: *prosessin sisäinen asiakas* (in-process client), *itsenäinen asiakas* (standalone client), *prosessin sisäinen palvelin* (in-process server) ja *itsenäinen palvelin* (standalone server). Prosessin sisäiset asiakkaat toimivat jonkin toisen komponentin tai sovelluksen, esimerkiksi WWW-selaimen sisällä. Tyypillinen esimerkki on Active-X-kontrolli tai Java-sovelma. Itsenäinen asiakas voi olla lähes mikä tahansa sovellus, joka paljastaa rajapintansa standardoidulla tavalla, esimerkiksi Microsoftin Word tai Excel. Itsenäinen

palvelin voi myöskin olla mikä tahansa sovellus, jolla on standardoitu rajapinta ja johon pääsee käsiksi esimerkiksi RPC-kutsulla tai muulla tavalla tietokoneverkon ylitse (tässä on ilmeisesti määritelty palvelin ohjelmaksi, joka palvelee muilla prosessoreilla sijaitsevia asiakkaita). Tämän mahdollistavia teknologioita ovat RPC:n lisäksi mm. DCOM (Distributed Component Object Model), CORBA (Common Object Request Broker Architecture) ja RMI (Remote Method Invocation). Prosessin sisäinen palvelin toimii jonkin muun komponentin tai sovelluksen (container) sisällä, esimerkkinä Microsoft Transaction Server. Sama komponentti voi toimia sekä asiakkaana että palvelimena.

David Chappellin esittämä taksonomia luokittelee vain binäärimuotoisia ohjelmistokomponentteja.

3.5.5 Hoffmannin taksonomia

Holger Hoffmann [Hoffmann 1997] jakaa ohjelmistokomponentit niiden granulariteetin mukaan neljälle tasolle. Alimmalla tasolla sijaitsevat *alkeistason komponentit*, jotka ovat horisontaalisesti yleiskäyttöisiä, esim. käyttöliittymäkirjastot tai niiden osat. *Palvelutason komponentit* ovat vertikaalisesti tai horisontaalisesti yleiskäyttöisiä palveluita. Hoffmann ei kuitenkaan tee kovin selvää eroa alkeis- ja palvelutason komponentin välille. *Liiketoimintapalvelutason komponentit* vastaavat suoraan jotain liiketoimintakäsitettä ja ovat siinä mielessä sovellusaluekohtaisia. liiketoimintakomponentti kuvaa siis jonkin liiketoimintakäsitteen nimen, määritelmän, ominaisuudet, käyttäytymisen ja suhteen muihin liiketoimintakomponentteihin. Ylimpänä ovat *sovellustason komponentit*, jotka ovat fyysisesti olemassa sovelluksina.

Hoffmannin jako on siis kerroksellinen ja sen voi ajatella tarkentavan kuvaa 2 (katso alakohta 3.4.1) siten, että kuvan 2 keskimmäinen kerros (komponentit) jaetaan vielä neljään ohuempaan kerrokseen.

3.5.6 Kainin jaottelu

J.Bradford Kain [Kain 1998] jakaa komponentit neljän ulottuvuuden mukaan. Ensinnä komponentit voidaan jakaa määrittely- ja toteutuskomponentteihin. Määrittelykomponentit ovat ideoita ja dokumentteja, kuten suunnittelumalleja ja vaatimusmäärittelyjä. Toteutuskomponentit ovat lähde- tai binäärikoodia. Tämä jako vastaa kappaleessa 3.3 esitettyä jakoa abstrakteihin ja konkreettisiin ohjelmistokomponentteihin. Toinen ulottuvuus on komponentin tarkoitus (purpose). Tämä jakaa ohjelmistokomponentit sovellusalueen ongelmia ratkoviin ja ohjelmistoteknisiin komponentteihin. Kolmas ulottuvuus on komponentin koko (granularity). Neljäs ulottuvuus on komponentin yleisyys, eli miten laajalla sovellusalueella sitä voidaan käyttää (vrt. kohdassa 3.5.3 esitetty Haikalan ja Märijärven taksonomia).

3.6 Yhteenveto ohjelmistokomponentteihin liittyvästä käsitteistöstä

Ohjelmistokomponentin määritelmästä ei olla päästy vielä yksimielisyyteen. Valtaosa komponenteille esitetyistä määritelmistä sisältää kuitenkin seuraavat vaatimukset: komponentin tulee olla kokonaisuus, jolla on selkeä ulkoinen rajapinta, komponentin tulee olla itseriittoinen ja komponentin tulee olla yhdistettävissä muihin komponentteihin ja käytettävissä uudestaan erilaisessa ympäristössä.

Ohjelmistokomponenteille esitetyt määritelmät eroavat siinä, miten abstrakteja käsitteitä ne laskevat komponenteiksi. Tiukimmat määritelmät hy-

väksyvät vain binäärisen koodin, kun löyhemmät määritelmät hyväksyvät myös lähdekoodin, dokumentaation ja jopa abstraktit ideat - kaikki ohjelmistotuotannossa syntyvät, uudelleenkäytettävissä olevat tuotokset.

Komponentteihin enemmän tai vähemmän liittyviä käsitteitä ovat oliot, sovelluskehykset, suunnittelumallit ja arkkitehtuurit. Komponentit eivät ole sama asia kuin oliot tai olioluokat, mutta komponentit voivat koostua yhdestä tai useammasta olioluokasta. Oliopohjaisuus on vain yksi osa komponenttipohjaista kehittämistä. Suunnittelumalleja voi hyödyntää komponenttien sisäisessä suunnittelussa sekä komponenttien välisen toiminnan suunnittelussa. Sovelluskehykset ovat valmiita sovellusrunkoja, jotka voivat koostua komponenteista. Sovelluskehyksestä saadaan valmis sovellus lisäämällä tai sopeuttamalla joitain komponentteja. Arkkitehtuurilla tarkoitetaan ohjelmiston tai järjestelmän komponentteja, niiden välistä interaktiota, ohjelmiston rakentamisessa sovellettavia rajoitteita ja yleisiä periaatteita.

Kirjallisuudessa on esitetty hyvin erilaisista lähtökohdista tehtyjä komponenttien luokitteluja. Mm. Sametinger jakaa komponentit eri rajapintojen tason mukaan ja Haikala ja Märijärvi yleiskäyttöisyyden mukaan.

4 KOMPONENTTIMALLIT

Tämän luvun tarkoituksena on määritellä komponenttimallin käsite, esitellä lyhyesti kolme tällä hetkellä tärkeintä komponenttimallia sekä tehdä lyhyt yhteenveto näiden komponenttimallien keskeisimmistä eroista.

4.1 Mitä komponenttimallilla tarkoitetaan

Valtaosa ohjelmistokomponentin määritelmistä sisältää vaatimuksen, jonka mukaan komponentilla on oltava selkeä rajapinta. Rajapinta kertoo komponentin käyttäjille (toisille komponenteille) mitä palveluita komponentti tarjoaa ja miten näitä palveluita käytetään. Jotta komponentit voisivat käyttää toistensa tarjoamia palveluita (ts. komponenttien käyttäjät voisivat yhdistää ne toisiinsa), tulee rajapintojen noudattaa tiettyjä standardeja.

Katsaus eritasoisiin ohjelmistokomponenttien yhdistämistapoihin suoritetaan kohdassa 5.3.2. Vielä viime aikoihin saakka ohjelmistojen osien (komponenttien) yhdistäminen on rajoittunut binääritason standardeihin aliohjelmakutsujen suoritustavasta [Szyperski 1998].

4.1.1 Komponenttimallin määritelmä

Komponenttimallilla tarkoitetaan *standardia, joka määrittelee komponenttien rajapintojen rakenteen ja tavan jolla komponentti kommunikoi muiden komponenttien kanssa tai tälläisen standardin toteutusta. Lisäksi komponenttimalli voi tarjota komponenteille erilaisia palveluita.* Kirjallisuudessa on esitetty useita termejä kuvaamaan samaa asiaa, mm. komponenttijärjes-

telmä (component system) [Harmon 1998], komponenttimaailma (component world) [Szyperski 1998] ja olioväylä (object bus) [Orfali, Harkey & Edwards 1996].

4.1.2 Komponenttimallin tarjoamat palvelut

Tyypillisiä komponenttimallin komponenteille tarjoamia palveluita ovat *sijaintiläpinäkyvyys*, *ohjelmointikieliriippumattomuus* ja *käyttöjärjestelmäriippumattomuus*. Sijaintiläpinäkyvyydellä tarkoitetaan sitä, että komponentit voivat käyttää toisilla prosessoreilla sijaitsevien komponenttien palveluita samalla tavalla, kuin ne käyttäisivät paikallisia palveluita. Ohjelmointikieliriippumattomuudella tarkoitetaan sitä, että eri kielillä kirjoitetut komponentit voivat kommunikoida keskenään. Käyttöjärjestelmäriippumattomuudella tarkoitetaan sitä, että sama komponentti toimii useissa eri käyttöjärjestelmissä ilman, että komponenttia tarvitsee muuttaa. Kaikki komponenttimallit eivät tarjoa komponenteille kaikkia näitä palveluita.

Komponenttimalli voi tarjota komponenteille myös muita palveluita, kuten komponenttien nimeämisen, tuen visuaaliselle yhdistämiselle tai tuen komponenttien käsittelemiseen transaktioiden yhteydessä.

Ensimmäisen laajalle levinneen komponenttimallin muodostivat vuonna 1992 Visual Basicin kanssa käytettävät VBX-laajennukset [Harmon 1998] [Udell 1994]. Tällä hetkellä markkinoilla on kolme vahvaa komponenttimallia: OMG:n CORBA, Microsoftin COM ja Sun Microsystemsin JavaBeans [Szyperski 1998] [Harmon 1998]. Tässä kappaleessa esitellään näiden kolmen komponenttimallin pääpiirteitä. Erityistä huomiota kiinnitetään siihen, miten komponenttimallit määrittelevät komponentin ja mitä palveluita komponenttimallit tarjoavat komponenteille.

4.2 Tärkeimmistä nykyisistä komponenttimalleista

Tässä kappaleessa esitellään kolme tällä hetkellä kaupallisesti tärkeintä komponenttimallia: Microsoftin COM, Object Management Groupin CORBA sekä Sun Microsystemsin JavaBeans. Kustakin komponenttimallista pyritään tuomaan esille keskeiset piirteet. Tarkkaan ja yksityiskohtaiseen käsittelyyn ei pyritä, sillä kustakin komponenttimallista on saatavilla paljon kirjallisuutta. Luvun lopussa esitetään yhteenveto, jossa verrataan näiden komponenttimallien keskeisimpiä piirteitä.

4.2.1 COM

COM (Component Object Model) on Microsoftin komponenttimalli. COM yhdistää Microsoftin aikaisemmin julkistamat OCX-komponentit ja OLE2-arkkitehtuurin. COM on samalla sekä spesifikaatio että toteutus.

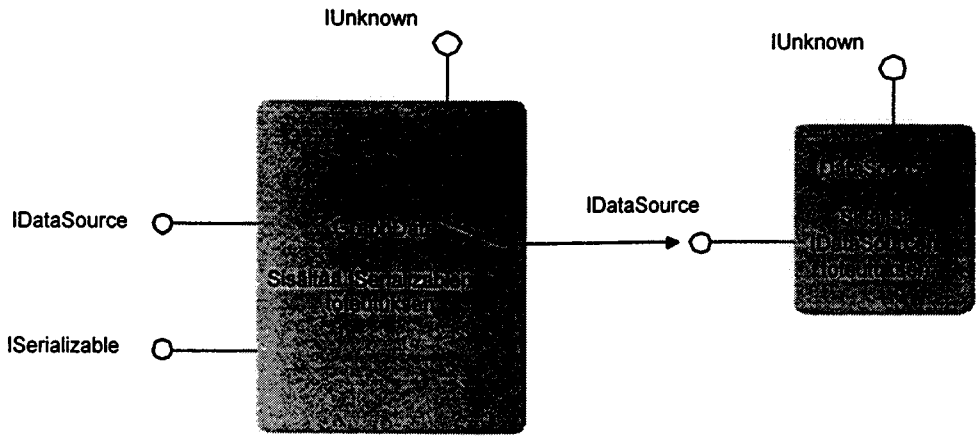
COM on binäärinen standardi, jonka mukaan rajapinnat toteutetaan. Rajapinta määritellään COM:issa osoittimeksi osoittimeen, joka osoittaa virtuaalitauluun, jossa määritellään joukko metodeita [Szyperski 1998].

COM ei varsinaisesti määrittele, mikä komponentti on [Szyperski 1998]. COM-mallissa puhutaan usein komponenteista ja olioista ja tarkoitetaan samaa asiaa. Käytännössä komponentti toteuttaa yhden tai useamman COM-luokan, jotka kukin toteuttavat useita COM-rajapintoja.

Kaikkien komponenttien on tuettava ainakin `IUnknown`-rajapintaa [Szyperski 1998] [Harmon 1998]. `IUnknown`-rajapinta sisältää kolme metodia, jotka ovat `QueryInterface`, `AddRef` ja `Release`. `QueryInterface` kertoo, tukeeko COM-komponentti tiettyä rajapintaa ja jos tukee, se palauttaa osoittimen kyseiseen rajapintaan. `AddRef` ja `Release` liittyvät roskienkeruuseen (garbage collection), joka perustuu COM:issa viitteiden laskentaan.

`AddRef` lisää viittauksen komponenttiin ja `Release` vapauttaa sen [Szyperski 1998].

Komponentti voi sisältää tukemiensa rajapintojen toteutuksen itse (aggregaatio) tai se voi sisältää osoittimen toiseen komponenttiin, joka tukee rajapintaa (sisältäminen)[4] [Szyperski 1998]. Kuvassa 3 on esitetty COM-komponentti, joka tukee kolmea rajapintaa: `IUnknown`, `IDataSource` ja `ISerializable`. Komponentti sisältää osoittimen `IUnknown`-rajapinnan toteutukseen, osoittimen rajapinnan `IDataSource` toteutukseen ja rajapinnan `ISerializable` toteutuksen. Käytännössä rajapinnan toteutus voi olla proseduraalinen moduuli, olioluokka, joukko olioluokkia tai kokonainen sovellus [Harmon 1998].



KUVA 3 COM-komponentti `GraphData` tukee rajapintoja `IUnknown`, `IDataSource` ja `ISerializable`. `GraphData` sisältää `ISerializablen` toteutuksen (aggregaatio) ja osoittimen toiseen komponenttiin, joka tukee rajapintaa `IDataSource` (containment).

COM-komponenttimallin suurin etu ja pahin rajoite on se, että se on tiukasti integroitu Windows-käyttöjärjestelmään. COM-komponentit voivat käyttää hyväkseen kaikkia Windowsin tarjoamia palveluita ilman rajoituksia. Toisaalta COM-komponentit eivät toimi muissa käyttöjärjestelmissä. COM

---

[4] Sekä aggregaatio (aggregation) että sisältäminen (containment) ovat komponentin ympäröimistä (wrapping) toisella komponentilla. Tätä käsitellään alakohdassa 5.2.7.

ei siis tarjoa komponenteille käyttöjärjestelmäriippumattomuutta. Tosin ainakin Software AG on jo julkaissut COM-toteutuksen UNIX-käyttöjärjestelmään.

DCOM on COM:iin tehty laajennus, joka mahdollistaa eri prosessoreilla sijaitsevien komponenttien väliset kutsut. Toisella prosessorilla sijaitsevaa rajapintaa voidaan kutsua täysin samalla tavalla kuin samalla prosessorilla sijaitsevaa rajapintaa. DCOM luo tarvittavan proxy-olion asiakaspäähän ja tynkäolion palvelinpäähän sekä huolehtii mahdollisista tietotyyppimuunnoksista (marshalling). Näin COM tarjoaa komponenteille sijaintiläpinäkyvyyden (location transparency).

Koska COM on binääritason standardi, voidaan COM-komponentit periaatteessa kirjoittaa lähes millä tahansa käännettävällä ohjelmointikielellä. Näin COM tarjoaa komponenteille ohjelmointikieliriippumattomuuden.

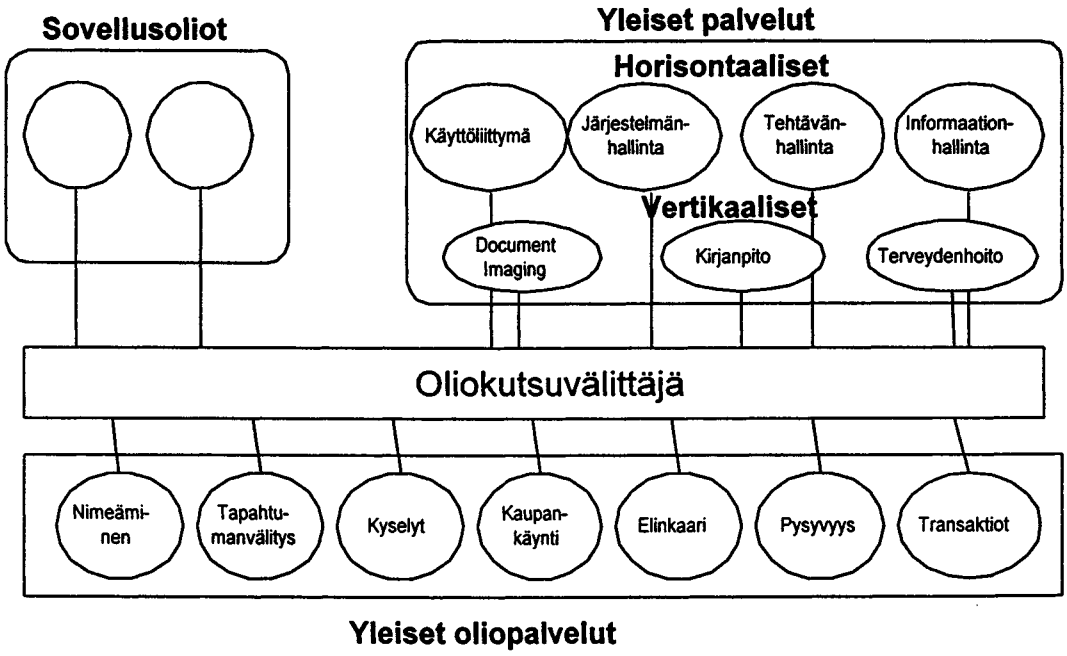
4.2.2 CORBA

CORBA (Common Object Request Broker Architecture) on OMG:n (Object Management Group) kehittämä standardi hajautettujen oliopohjaisten järjestelmien rakentamista varten. OMG on aloittanut CORBA-standardin kehittämisen vuonna 1989, ja vuonna 1995 jäsenyrityksiä oli yli 500 [Orfali, Harkey & Edwards 1996].

Vuonna 1989 termiä komponentti ei käytetty yhtä laajalti samassa merkityksessä kuin nykyään. CORBA:n yhteydessä puhutaan yleisesti ottaen olioista. Oliolla (tai komponentilla) tarkoitetaan CORBA:ssa IDL-kielisen rajapinnan toteuttavaa koodia, joka voi olla yksi olioluokka, koostua useasta olioluokasta tai voi olla toteutettu ei-oliopohjaisella kielellä.

Toisin kuin COM, CORBA on standardi eikä toteutus. Eri ohjelmistotalot, kuten IBM, Inprise ja Iona toteuttavat ja myyvät CORBA-standardin mukaisia ORB:eja.

CORBA koostuu neljästä pääelementistä: *Oliokutsuvälittäjä* (Object Request Broker, ORB), *Yleiset oliopalvelut* (Common Object Services), *Yleiset palvelut* (Common Facilities) ja *Sovellusoliot* (Application Objects). Näistä ORB käsittelee matalimman tason asioita ja sovellusoliot korkeimman tason asioita. Kuvassa 4 on esitetty nämä CORBA-arkkitehtuurin keskeiset osat.



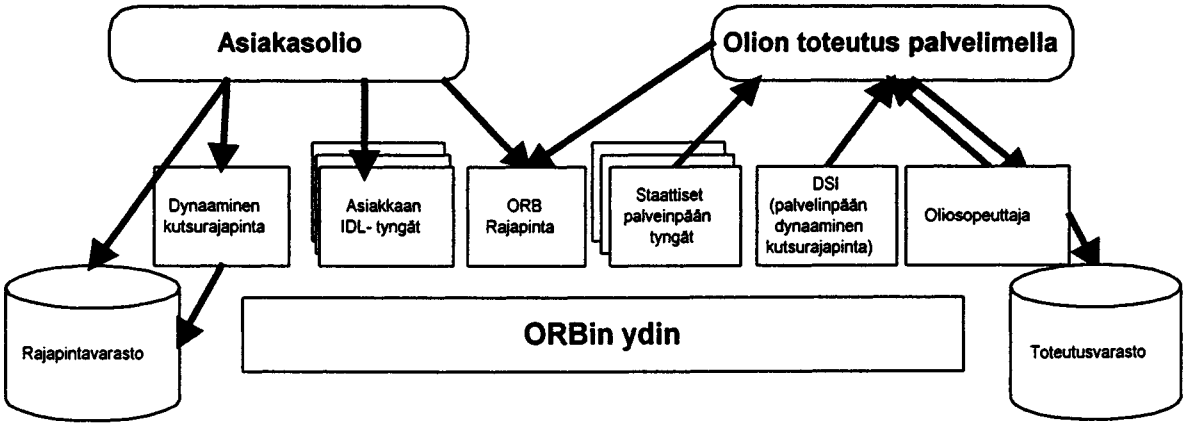
KUVA 4 Yleisnäkymä CORBA-arkkitehtuurista. Kuva mukaeltu Orfalin ym. kirjasta [Orfali, Harkey & Edwards 1996].

Oliokutsuvälittäjä mahdollistaa sen, että oliot voivat kutsua eri osoiteavaruudessa sijaitsevien olioiden metodeja ja vastaanottaa näiden metodien palauttamat arvot. Tämä mekanismi on kutsuvan olion kannalta näkymätön.

Metodikutsut voivat olla *staattisia* tai *dynaamisia*. Staattiset metodikutsut tarkoittavat sitä, että metodikutsujen oikeellisuus tarkistetaan käännösaikana. Dynaamiset metodikutsut voidaan rakentaa ajonaikaisesti.

ORB mahdollistaa myös eri kielillä tehtyjen olioiden välisen kommunikaation, kun oliolle on kirjoitettu IDL-kielinen (Interface Description Language) rajapinta.

Kuvassa 5 on esitetty CORBA-oliokutsuvälittäjän rakenne.



KUVA 5 Oliokutsuvälittäjän rakenne. Kuva mukaeltu Orfalin ym. kirjasta [Orfali, Harkey & Edwards 1996].

Asiakaspään IDL-tyngät (IDL-stubs) ovat palvelinpään olioiden valmiiksi käännettyjä rajapintoja. Nämä määrittelevät, miten palvelinpään olioiden palveluja (metodeja) kutsutaan. (Palvelinpäällä tarkoitetaan tässä sitä puolta, jossa sijaitsevat ne oliot, joiden metodeja kutsutaan. Käytännössä sama olio voi toimia samassa järjestelmässä sekä asiakkaana että palvelimena.) Nämä rajapinnat on määritelty IDL-kielellä ja käännetty tyngiksi IDL-kääntäjällä. Asiakkaalla pitää olla siis tynkä jokaista palvelinpään oliota kohden. Tynkä sisältää myös koodin, jolla se muuttaa palvelinolion metodille lähetettävät parametrit käyttöjärjestelmäneutraaliin muotoon (marshalling). Vastaavasti palvelimella olio purkaa parametrit ymmärtä-

määnsä muotoon. Näin oliokutsuvälittäjän käyttäjiltä piilotetaan eri käyttöjärjestelmien toisistaan eroavat tietotyypit.

Dynaaminen kutsurajapinta (DII, Dynamic Invocation Interface) mahdollistaa rajapintavaraston metodien ajonaikaisen tutkimisen. Asiakkaan ei siis välttämättä tarvitse käännösaikana tietää, mitä kaikkea palvelimella on tarjolla.

Rajapintavarasto (interface repository) on tietokanta, joka sisältää tietoa palvelimella sijaitsevien olioiden rajapinnoista. Interface Repository API tarjoaa olioille mahdollisuuden tutkia ja tarpeen vaatiessa myös päivittää tätä metadataa. Rajapintavarasto toteutetaan joukkona olioita, jotka esittävät rajapintavaraston sisältämiä rajapintoja.

Olioviite (object reference) yksilöi olion hajautetussa järjestelmässä. CORBA ei määrittele millaisia näiden viittausten tulisi olla. CORBA 2.0:an kuuluva IIOP-protokolla määrittelee kuitenkin millaisessa muodossa näiden viittausten tulee kulkea eri ORB:ien välillä. Oliokutsuvälittäjän rajapinta (ORB Interface) tarjoaa palveluita joilla voi esim. muuttaa olioviitteen merkkijonoksi tai toisinpäin [Orfali, Harkey & Edwards 1996].

Palvelinpään IDL-tyngät ("skeletons") ovat palvelimelta löytyvien olioiden IDL:llä määriteltyjä rajapintoja, jotka on luotu IDL-kääntäjällä (vastaavasti kuin asiakaspään IDL-tyngät). Nämä tyngät kutsuvat varsinaisten palvelinpään olioiden metodeita.

Palvelinpään dynaaminen kutsurajapinta (DSI, Dynamic Skeleton Interface) mahdollistaa myös sellaisten palvelimella sijaitsevien olioiden metodien kutsumisen, joita varten ei ole luotu IDL-tynkää. DSI tutkii metodikutsun parametreja ja päättelee kohdeolion ja -metodin.

Oliosopeuttaja (object adapter) käsittelee palvelimelle tulevia kutsuja. Se mahdollistaa palvelinpään olioiden ajonaikaisen käynnistyksen kutsun yhteydessä (voidaan tarvittaessa luoda uusi säie tai prosessi) ja antaa käynnistetyille olioille yksilöllisen olioviitteen. Oliosopeuttaja huolehtii toteutusspesifisten ja ORB-spesifisten olioviitteiden yhteensovittamisesta. Oliosopeuttaja vastaa myös *toteutusvaraston* (implementation repository) ylläpidosta.

Toteutusvarasto (implementation repository) on tietokanta joka sisältää olioviitteet käynnissä oleviin ilmentymiin ja tietoa palvelimen tällä hetkellä tarjoamista olioluokista. Toteutusvarasto voi myös sisältää muuta ORB:n toimintaan liittyvää tietoa (esim. lokitiedot).

CORBA-komponentteja voi tehdä millä tahansa ohjelmointikielellä, jolle on määritelty kuvaus IDL-kielelle. Näin ollen CORBA tarjoaa komponenteille ohjelmointikieliriippumattomuuden. CORBA tarjoaa komponenteille myös käyttöjärjestelmäriippumattomuuden sikäli, kun ORB-toteutuksia on tehty eri käyttöjärjestelmille.

4.2.3 JavaBeans ja Enterprise JavaBeans

JavaBeans on Sun Microsystemsin komponenttimalli, joka perustuu Java-ohjelmointikieleen ja JDK 1.1 tai uudempaan Java-ympäristöön. Java-kielen spesifikaatio edellyttää vähintään seuraavien kolmen luokkakirjaston olemassaoloa: `java.lang`, `java.util` ja `java.io` [Szyperski 1998]. JavaBeans edellyttää lisäksi `java.beans`-luokkakirjaston mukanaoloa. Näin ollen JavaBeans-komponenttimalli on tiukasti sidoksissa Sunin luokkakirjastoiden muodostamaan sovelluskehykseen [Franz]. Toisin kuin COM ja CORBA, JavaBeans (ainakin ensimmäinen versio) on tarkoitettu lähinnä graafisten käyttöliittymäkomponenttien rakentamiseen [Vanhelsuwé 1997].

Myöhemmin julkaistu Enterprise JavaBeans on puolestaan tarkoitettu palvelimella transaktioympäristössä toimivien komponenttien rakentamiseen.

Yksittäistä JavaBeans-mallin komponenttia kutsutaan *JavaBean*iksi [Vanhelsuwé 1997]. Komponentti koostuu yhdestä tai useammasta luokasta (binäärimuodossa olevia välikielisiä `.class`-tiedostoja) sekä niiden tarvitsemista resursseista. Resurssit ovat komponentin tarvitsemia tiedostoja, kuten kuvatiedostoja, HTML-kielisiä dokumentaatiotiedostoja, tai sarjallistettuja olioita, joita käytetään prototyyppi-ilmentymien luontiin [Szyperski 1998]. Komponentti paljastaa käyttäjilleen joukon *ominaisuuksia* (properties), *tapahtumia* (events) ja *metodeita* (methods). JavaBean jaellaan JAR-pakatussa tiedostossa, joka sisältää tarvittavat tiedostot.

Ominaisuudet ovat attribuutteja, jotka komponentti paljastaa käyttäjilleen. Käytännössä ominaisuudet ovat komponentin sisältämien luokkien julkisiksi (public) määriteltyjä attribuutteja, joille on määritelty ns. *saantimetodit* (accessor methods). Saantimetodit on nimettävä JavaBeans-standardin mukaisesti.

Komponentin tarjoamat metodit ovat muita komponentin sisältämien luokkien määrittelemiä julkisia metodeja.

Tapahtumat ovat komponentin sisäisiä tilamuutoksia, joista komponentti informoi muita komponentteja. Tapahtumien käsittely kuuluu oleellisesti graafisten käyttöliittymien rakentamiseen, koska käyttöliittymän on kyettävä reagoimaan käyttäjän tekemiin asioihin. Tapahtumia voidaan käyttää kuitenkin myös muiden komponenttien väliseen kommunikaatioon.

Tapahtuman käsittelyn kannalta komponentit ovat joko *tapahtumien lähteitä* (event sources) tai *tapahtumien kuuntelijoita* (event listeners). Komponentti voi olla myös sekä tapahtumien lähde että kuuntelija. JavaBeans-

mallissa tapahtumat ovat olioita, joiden luokat periytyvät `java.awt.AWTEvent`-luokasta. Näin ollen tapahtumat voivat sisältää informaatiota tapahtuman luonteesta sekä metodeja joilla niiltä voi tiedustella tätä informaatiota.

JavaBeans-malli erottaa toisistaan komponenttien *suunnittelunaikaisen* (design-time) ja *ajonaikaisen* (run-time) käytön. Suunnittelunaikainen käyttö tarkoittaa komponenttien yhdistelyä visuaalisessa ympäristössä, kuten Inprisen JBuilderissa. Ajonaikainen käyttö tarkoittaa komponentin toimintaa valmiissa sovelluksessa. Komponentin ilmentymä voi esim. piirtää itsensä eri tavalla suunnittelun aikana kuin ajonaikaisesti.[5]

JavaBeans-malliin kuuluu *introspektio* (introspection), joka tarkoittaa sitä, että komponentin ilmentymän tarjoamia ominaisuuksia, tapahtumia ja metodeja voidaan tarkastella ajonaikaisesti. Tämä helpottaa mm. graafisten JavaBeans-ohjelmankehitysympäristöjen toteutusta.

Hajautettujen komponenttien käsittely hoituu JavaBeans-mallissa RMI-palvelun (Remote Method Invocation) sekä olioiden sarjallistamisen avulla. Kummatkin piirteet esiteltiin jo JDK 1.1:ssä, eivätkä sinänsä kuulu ainoastaan JavaBeans-malliin. Toisella prosessorilla sijaitsevaan olioon voidaan viitata sen rajapinnan kautta. Rajapintojen, joihin voidaan viitata verkon yli, tulee periytyä `java.rmi.Remote` -rajapinnasta. Javaan sisältyvä automaattinen roskienkeruu toimii myös hajautettujen komponenttien yhteydessä [Szyperski 1998].

Enterprise JavaBeans on komponenttimalli palvelimella ajettavia komponentteja varten. Enterprise JavaBeans määrittelee erilaisia palveluja näille komponenteille, mm. transaktiotuen.

Javaa sanotaan käyttöjärjestelmäriippumattomaksi, koska välikielelle käännetty Java-ohjelma vaatii toimiakseen vain Java-virtuaalikoneen. Virtuaalikone tulkkaa välikielisen Java-ohjelman kohdeprosessorin ymmärtämälle konekielelle. Java-virtuaalikoneita on toteutettu kaikille yleisimmille nykyisille käyttöjärjestelmille. Käytännössä kuitenkin pieniä yhteensopimattomuuksia esiintyy jo samassa käyttöjärjestelmässä eri virtuaalikonetoteutusten välillä - vaikka ne toteuttaisivatkin saman spesifikaation, toteutuksessa olevat virheet ovat erilaisia ja eri paikoissa. Tästä huolimatta Javan avulla voidaan saavuttaa merkittäviä säästöjä, jos komponentin tai ohjelmiston on toimittava useissa eri käyttöjärjestelmissä.

JavaBeans-komponenttimallin heikkona puolena on pidetty sitä, että komponenttien kirjoittamisessa on rajoituttava vain yhteen ohjelmointikieleen.

4.2.4 Yhteenveto komponenttimallien eroista

Taulukossa 3 esitetään joitain COM, CORBA ja JavaBeans-komponenttimallien keskeisimpiä eroja.

---

[5] Beaneja voidaan toki käyttää muuallakin kuin visuaalisessa JavaBeans-ympäristössä, esimerkiksi perinteisessä tekstipohjaisessa ohjelmointiympäristössä. Tällöin niitä käsitellään, kuten mitä tahansa muutakin Java-luokkaa tai luokkia.

	COM	CORBA	JavaBeans
Komponenttimallin luonne	Binääritason standardi ja toteutus sekä joukko palveluja.	Komponenttien rajapintoja ja komponenteille tarjottavia palveluja koskeva standardi.	Komponenttien rajapintoja koskeva standardi sekä joukko luokkia, jotka toteuttavat palveluita.
Mikä on komponentti?	Useiden rajapintojen toteutus.	IDL-rajapinnan toteutus.	Yksi tai useampi Java-luokka ja niiden tarvitsemat resurssitiedostot.
Sijaintiriippumattomuus	Toteutuu DCOM-laajennuksen avulla.	Toteutuu eri valmistajien ORB:ien avulla.	Toteutuu RMI-pakkauksen avulla.
Käyttöjärjestelmäriippumattomuus	Ei toteudu, COM on sidottu Windows-käyttöjärjestelmään.	Toteutuu, mikäli käyttöjärjestelmälle on olemassa ORB-toteutus.	Toteutuu, mikäli käyttöjärjestelmälle on toteutettu Java-virtuaalikone.
Ohjelmointikieliriippumattomuus	Toteutuu, komponenttien rajapinnat toteuttavat COM-standardin binääritasolla.	Toteutuu, jos ohjelmointikielen ja IDL:n välille on määritelty kuvaus.	Ei toteudu. JavaBeans on sidottu Java-kieleen ja useisiin Sunin pakkauksiin.
Komponenttimallin tarjoamat muut keskeiset palvelut.	Useat Windows-käyttöjärjestelmän tarjoamat palvelut.	OMG:n määrittelemät yleiset oliopalvelut, hierarkiset ja vertikaaliset sovellusoliopal-velut.	Introspektio, reflektio, Sunin Java-kirjastojen muodostama sovelluskehys.

TAULUKKO 3 Yhteenveto komponenttimallien keskeisimmistä eroista.

5 KOMPONENTTITEKNOLOGIASTA

Jotta komponenttipohjaisessa ohjelmistotuotannossa olisi järkeä, komponenttien hankinnan, sopeuttamisen ja yhdistämisen tulisi olla helpompaa kuin uusien komponenttien rakentaminen. Tässä luvussa tarkastellaan erilaisia menetelmiä, joita voidaan käyttää komponenttien hankintaan, sopeuttamiseen ja yhdistämiseen.

5.1 Komponenttien hankinta

Komponenttien hankinta edellyttää sitä, että voidaan löytää joukko potentiaalisia sopivia komponentteja ja kyetään jotenkin arvioimaan näiden komponenttien sopivuutta projektin tarpeiden kannalta. Komponentteja voidaan etsiä markkinoilta tai oman organisaation aikaisempien projektien tuotoksista. Komponenttivarastot ovat joko organisaation ulkopuolisia tai sisäisiä järjestelmällisesti ylläpidettyjä tietokantoja, jotka auttavat komponenttien etsinnässä. Komponenttien arviointi tapahtuu tavallisesti komponenttien rajapintoja tutkimalla ja tarvittaessa kokeilemalla komponentin käyttöä prototyyppisovelluksessa.

5.1.1 Komponenttivarastot

Komponenttivarastot (component repositories) sisältävät uudelleenkäytettäviä komponentteja ja metatietoa, jonka avulla komponenttien hakeminen varastosta on helppoa. Komponenttipohjaisessa ohjelmistotuotannossa komponenttivarasto on linkki komponenttituotannon ja sovellustuotannon välillä.

Komponenttivarasto voi sisältää yleisiä komponentteja, sovellusaluekohtaisia komponentteja tai viittauksia muihin komponenttivarastoihin ("keltaiset sivut") [Sametinger 1997].

Sametinger [Sametinger 1997] kertoo Bankerin ym. [Banker, Kauffmann & Zweig 1993] huomanneen seuraavia seikkoja tutkiessaan komponenttivarastojen käyttöä: varastoitujen komponenttien määrän kasvaessa uudelleenkäytön määrä ei nouse samassa suhteessa, suurin osa komponenteista käytetään uudelleen samoissa projekteissa joissa ne on tehty, ja ohjelmoijat uudelleenkäyttävät mieluiten itse tekemiään komponentteja. Sametingerin johtopäätös Bankerin tutkimuksesta on se, että hakumenetelmät, joilla komponentteja haetaan varastoista, ovat puutteellisia.

5.1.2 Komponenttien arviointi

Eri lähteistä löytyneitä komponentteja on arvioitava, jotta saataisiin selville soveltuvatko ne projektin käyttöön sellaisenaan tai sopeutettuna. Komponenttien arvioinnissa on huomioitava komponentin toiminnallisuus ja laatu. Lisäksi mikäli komponentti ostetaan markkinoilta, saattaa olla syytä arvioida komponentin valmistajan tulevaisuudennäkymiä. Jos ne ovat heikot, komponentin tuki ja tulevat päivitykset ovat vaarassa.

Komponentin toiminnallisuudella tarkoitetaan niitä palveluita, joita komponentti tarjoaa käyttäjilleen. Komponentin rajapinnat kuvaavat nämä palvelut, mutta yleensä ylimalkaisella tasolla (rajapinta kertoo kunkin operaation nimen, palautustyypin, parametrien lukumäärän sekä kunkin parametrin nimen ja tyypin). Komponentin mukana saattaa tulla dokumentaatiota, joka kuvaa toiminnallisuutta tarkemmin.

Komponenttien laadulla tarkoitetaan mm. komponentin luotettavuutta, käytettävyyttä sekä suorituskykyä. Usein puhutaan ei-toiminnallisista

ominaisuuksista. Komponentin rajapinnat eivät tyypillisesti kerro komponentin laadusta mitään.

Komponentin arviointiin on käytössä periaatteessa kolme keinoa: arviointi rajapintojen avulla, arviointi dokumentaation perusteella sekä komponentin kokeileminen. Arviointi rajapintojen avulla on halvin ja kokeileminen kallein vaihtoehto.

5.1.3 Komponenttien kuvaaminen rajapintojen avulla

Komponentin tulisi sisältää jonkinlaista metatietoa omasta käyttäytymisestään, jonka avulla suunnittelijat voivat päättää, soveltuuko komponentti heidän käyttöönsä. Tätä tietoa kutsutaan komponentin *rajapinnaksi*[6] (interface). Rajapinta siis kuvaa, mitä palveluja komponentti tarjoaa, mutta ei kerro miten komponentti toteuttaa nämä palvelut.

Komponentin rajapinnat voidaan jakaa *sisääntuleviin* ja *ulosmeneviin* [Hoffmann 1997]. Sisääntulevat rajapinnat paljastavat ne operaatiot, joita komponentin käyttäjät voivat kutsua. Ulosmenevät rajapinnat kertovat, mitä operaatioita komponentti itse käyttää. Tässä kohdassa käsitellään pelkästään sisääntulevia rajapintoja.

Brown ja Short esittävät, että rajapinnan tulisi kertoa ainakin sen tarjoamien operaatioiden nimet, palautustyypit ja parametrilistat, kuvata epäformaalisti komponentin toiminnallisuus ja muutama toimintaskenaario, kuvata epäformaalisti komponentin toimintaympäristö (vaadittavat käyttöjärjestelmät, muut ohjelmistot ja tunnetut rajoitteet) sekä sisältää tietoa komponentin suorituskyvystä ja luotettavuudesta [Brown & Short 1997]. Komponentti voi tarjota käyttäjälleen useita rajapintoja.

---

[6] Tämä tarkoittaa nimenomaan sisääntulevia rajapintoja.

Szyperskin mukaan rajapintoja tulisi tarkastella ensisijaisesti *sopimuksina* (contracts). Sopimuksen osapuolina voidaan pitää toisaalta komponenttia, joka tarjoaa palveluita ja toisaalta käyttäjiä (jotka voivat olla muita komponentteja), jotka käyttävät komponentin tarjoamia palveluita. On tärkeää, ettei komponentti koskaan riko sopimustaan, sillä tämä voi johtaa siihen että se *rikkoo* käyttäjänsä eli sopimukseen luottavat, komponentin palveluita rajapinnan kautta käyttävät muut komponentit eivät enää toimi. Sopimuksen tulee määritellä mitä palveluja tarjotaan sekä esi- ja jälkiehdot niiden käyttöä varten. Esiehdot (preconditions) ovat ehtoja, joiden on oltava voimassa, jotta palvelua voisi käyttää. Jälkiehdot (postconditions) ovat ehtoja, joiden komponentti takaa olevan voimassa palvelun käytön jälkeen. Toteutus voi heikentää esiehtoja ja/tai vahvistaa jälkiehtoja, muttei päinvastoin. Sopimus voi myös sisältää invariantteja, eli ehtoja joiden taataan pysyvän voimassa operaation suorituksen ajan. Sopimus voi myös sisältää ei-funktionaalisia vaatimuksia kuten tietoa suorituskyvystä, esimerkiksi operaatioiden aikakompleksisuudesta tai muistinkäytöstä [Szyperski 1998].

Bertrand Meyerin suunnittelema Eiffel-kieli sisältää esi- ja jälkiehdot sekä invariantit kielellisinä rakenteina [Hoffmann 1997]. Vaikka ohjelmointikieli ei tukisikaan näitä rakenteita suoraan, voidaan ne kuitenkin toteuttaa valintalauseiden avulla.

5.1.4 Rajapinnat ja versiointi

Yksinkertaisin tapa varmistaa etteivät sopimukset muutu, on kieltää niiden muuttaminen. Todellisuudessa komponentteja kuitenkin kehitetään eteenpäin ja niistä julkaistaan uusia versioita. Kehitystyön yhdeydessä voi tulla esiin uusia tarpeita, joita ei osattu huomioida alkuperäistä rajapintaa suunniteltaessa. Jos vain komponentin rajapinta muuttuu, puhutaan syntaktisesta muutoksesta. Mikäli komponentin toiminnallisuus muuttuu, ky-

seessä on semanttinen muutos. Sopimusten muuttuminen voi johtaa siihen, että vanhaan sopimukseen luottavat käyttäjät eivät voi enää käyttää komponentin uutta versiota ts. uusi versio rikkoo olemassaolevat käyttäjät.

Erityisen suuria ongelmia uuden version julkistaminen aiheuttaa silloin, kun komponentin valmistaja ei kontrolloi komponentin käyttäjiä eli yleensä ainakin silloin, kun komponenttia myydään markkinoilla. On tärkeää varmistaa, ettei uusi versio riko käyttäjiä. Mahdollisia ratkaisuja ovat mm. muuttumattomat rajapinnat, versionumeroiden käyttö sekä sopimusten määräaikaisuus.

Rajapintojen muuttumattomuudella tarkoitetaan sitä, että kun uuden version yhteydessä rajapintaan halutaan tehdä lisäyksiä, vanhaa rajapintaa ei muuteta, vaan tehdään uusi rajapinta joka sisältää samat operaatiot kuin vanha rajapinta ja lisäksi uusia operaatiota. Komponentin uusi versio tukee sekä uutta että vanhaa rajapintaa. Versionumeroiden käyttö tarkoittaa sitä, että samasta rajapinnasta tehdään useita eri versioita. Rajapinnan käyttäjät tietävät mitä versiota ne osaavat käyttää ja näin huomataan, mikäli komponentin tarjoaman rajapinnan ja käyttäjän olettaman rajapinnan versioiden välillä on ristiriita. Sopimusten määräaikaisuus tarkoittaa sitä, että rajapinnan tekijä takaa, että sopimus on voimassa (ts. rajapinta ei muutu) määräaikaan mennessä, mutta tämän jälkeen käyttäjät eivät voi enää luottaa rajapintaan.

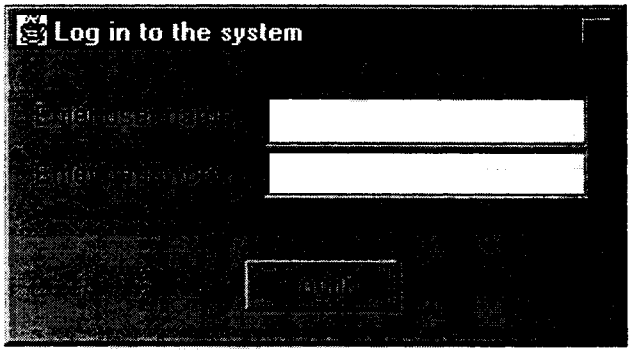
Esimerkiksi Microsoftin COM-komponenttimalli käyttää muuttumattomia rajapintoja ja IBM:n SOM-komponenttimalli versionumeroita.

5.2 Komponenttien sopeuttaminen

Ohjelmistokomponentteja voidaan harvoin käyttää uudelleen sellaisenaan [Bosch]. Komponenttien *sopeuttamisella* tarkoitetaan niitä toimenpiteitä,

joita ohjelmistokomponentille joudutaan tekemään, jotta sitä voitaisiin käyttää tietyn ohjelmiston rakentamisessa. Sopeuttamisen tavoitteena on usein tehdä pieniä toiminnallisia muutoksia ohjelmistokomponenttiin.

ESIMERKKI 1: Olkoon esimerkkikomponenttina graafinen ikkuna `LoginDialog` (katso kuva 6), jota käytetään palvelimelle sisään kirjauduttaessa. Ikkuna sisältää kaksi tekstikenttää, joihin käyttäjä kirjoittaa käyttäjätunnuksensa ja salasanansa, sekä painonapin jonka painamiseen ikkuna osaa reagoida. Rakennettavassa ohjelmistossa ilmenee kuitenkin tarve muuttaa ikkunaa siten, että siihen lisätään mahdollisuus valita, halutaanko asiakas- ja palvelinohjelmistojen välinen yhteys kryptata vai ei. Uuden toiminnallisuuden lisäämiseksi komponenttia voidaan sopeuttaa.



KUVA 6 Alkuperäinen komponentti `LoginDialog`: Ikkuna, jota käytetään järjestelmään sisäänkirjautumisessa.

Sopeuttamista voidaan tarvita myös tilanteissa, joissa yhteensovitettavat komponentit käyttäytyvät halutulla tavalla, mutta niiden rajapinnoissa on syntaktisia eroavaisuuksia. Hölzle [Hölzle 1993] väittää varsin uskottavasti, että vapailla komponenttimarkkinoilla syntyy tällaisia tilanteita.

ESIMERKKI 2: Olkoon meillä piirto-ohjelma, joka osaa piirtää COM-komponentteja, jotka toteuttavat rajapinnan `IDrawable`. `IDrawable` sisältää metodin `draw`. Markkinoilta voi löytyä sinänsä toiminnallisesti sopivia komponentteja, jotka toteuttavat esimerkiksi rajapinnan `IGraphicalOb-`

`ject` ja metodin `paint` ja ovat siten yhteensopimattomia piirto-ohjelman kanssa.

Mikäli alkuperäistä komponenttia ei suoraan tarvitse muuttaa lainkaan, komponenttia uudelleenkäytetään mustana laatikkona (black box). Mikäli sopeuttaminen edellyttää komponentin muuttamista, komponenttia uudelleenkäytetään valkoisena laatikkona (white box). Mikäli sopeuttaminen edellyttää komponentin toteutuksen tarkastelua mutta ei sen muuttamista, komponenttia uudelleenkäytetään lasilaatikkona (glass box) [Sametinger 1997].

5.2.1 Sopeuttamistekniikat

Tässä luvussa tarkastellaan eri sopeuttamistekniikoita. Yleisimmin käytössä oleva tekniikka, jossa komponenttia sopeutetaan valkoisena laatikkona on ns. leikkaa-liimaa-tekniikka. Mikäli käytössä on oliopohjainen toteutuskieli, komponentteja voidaan sopeuttaa mustana tai lasilaatikkona käyttäen periyttämistä tai ympäröimistä. Mikäli ohjelmointikieli tukee moniperintää, nämä tekniikat voidaan yhdistää käyttämällä yhdisteitä. Suurempien kokonaisuuksien sopeuttaminen kerralla olisi mahdollista superimposition avulla.

Sopeuttamistekniikka on *läpinäkyvä* (transparent), mikäli sopeutetun komponentin ilmentymä voidaan sijoittaa alkuperäisen komponentin ilmentymän paikalle, eli korvattavuusperiaate toteutuu.

5.2.2 Mukauttaminen ja muokkaus

Johannes Sametinger [Sametinger 1997] jakaa komponenttien sopeuttamisen kahteen osaan: *mukauttamiseen* (customization) ja *muokkaamiseen*

(modification). Kyse on mukauttamisesta, mikäli komponentin kehittäjä on etukäteen määritellyt optioita, joiden avulla komponentin toimintaa voi konfiguroida. Esimerkiksi musta laatikko voidaan mukauttaa parametrisoinnilla ja valkoinen laatikko ehdollisella kääntämisellä. Mikäli sopeuttaminen taas tapahtuu täysin käyttäjän toimesta ja perustuu käytettävissä olevaan komponenttiteknologiaan, kyse on muokkaamisesta (esim. periyttäminen). Tässä luvussa käsitellään muokkaamistekniikoita.

5.2.3 Leikkaa-liimaa-tekniikka

Leikkaa-liimaa-tekniikka (copy-paste) tarkoittaa olemassaolevan toteutuksen (esim. lähdekoodin) kopioimista ja muuttamista. Tätä tekniikkaa voidaan käyttää, mikäli komponentin toteutus on saatavilla ja muutettavissa, eli komponenttia tarkastellaan valkoisena laatikkona. Ohjelmakoodia voidaan muokata leikkaa-liimaa-tekniikalla, vaikka kyse ei olisikaan komponenteista edes löyhimpien määritelmien mukaan (esim. mitä tahansa dokumenttia voidaan muokata tällä tavalla). Leikkaa-liimaa-tekniikan etuna on se, että komponenttia voidaan muuttaa täysin mielivaltaisesti. Toisaalta leikkaa-liimaa-tekniikan onnistunut soveltaminen edellyttää, että sopeuttaja ymmärtää miten alkuperäinen toteutus toimii. Ylläpito muodostuu rasitteeksi, koska komponentin sopeuttaja ei yleensä voi tietää, mikä osa komponentin toteutuksesta tulee pysymään samanlaisena tulevissa versioissa [Buchi & Weck 1997], joten tällä tavalla syntyvät uudet versiot eivät juuri koskaan ole korvattavissa toisillaan tai tulevilla versioilla.

Leikkaa-liimaa ei ole läpinäkyvä sopeutustekniikka, koska sopeutetun komponentin ilmentymää ei välttämättä voi käyttää alkuperäisen komponentin ilmentymän paikalla.

5.2.4 Periyttäminen

Periyttämistä (inheritance) voidaan käyttää, kun käytössä oleva ohjelmointikieli tukee sitä. Tässä luvussa rajaudutaan luokkapohjaisen periyttämisen käsittelyyn ja suljetaan pois prototyyppipohjaiset oliokielet.

Periyttäminen on yliluokan A ja aliluokkien B_0 ... B_n välinen suhde, jossa aliluokat perivät yliluokan attribuutit ja metodit ts. aliluokat sisältävät automaattisesti kaikki samat attribuutit ja metodit, jotka yliluokalla on. Luokkapohjainen periyttäminen voidaan jakaa *toteutuksen periyttämiseen* (implementation inheritance, subclassing) ja *rajapintojen periyttämiseen* (interface inheritance, subtyping) [Szyperski 1997]. Jos kyse on toteutuksen periyttämisestä, aliluokat perivät yliluokan metodien toteutuksen, muussa tapauksessa vain rajapinnan.

Periyttämistä käytetään usein silloin, kun yliluokan ja aliluokan välillä on ns. *is-a*-suhde ts. voidaan sanoa, että aliluokka on yliluokan erikoistapaus [Budd 1997]. Markku Sakkisen mukaan tällöin on kyse *olennaisesta* (essential) periyttämisestä. Jos taas periyttämistä käytetään pelkästään metodien toteutuksen uudelleenkäyttämiseksi, vaikka is-a-sääntö ei ole voimassa, kyse on *satunnaisesta* (incidental) periyttämisestä [Sakkinen 1992].

Joissain oliokielissä osa luokan metodeista ja attribuuteista voi olla suojattu siten, etteivät ne näy eivätkä periydy lainkaan aliluokille. Luokkapohjaisissa oliokielissä periyttämissuhteet määritellään käännösaikaisesti, eikä niitä voi muuttaa ajonaikaisesti.

Periyttämiseen liittyy olennaisesti *monimuotoisuus* (polymorphism) ja *korvattavuusperiaate* (principle of substitutability) [Budd 1997]. Mikäli viitesemantiikkaa noudattava muuttuja X on määritelty viittaamaan tyyppiä T olevaan olioon, X voidaan asettaa viittamaan mihin hyvänsä olioon, jonka

tyyppi on T:n alityyppi. Näin ollen periyttäminen on läpinäkyvä sopeuttamistekniikka.

Joissain oliokielissä on mahdollista, että luokalla on useampi kuin yksi yliluokka. Tällöin kyseessä on *moniperintä* (multiple inheritance).

Sopeuttaminen periyttämällä voi parhaimmillaan olla musta-laatikkouudelleenkäyttöä, koska sopeuttajan ei välttämättä tarvitse tuntea yliluokan toteutusta. Käytännössä periyttäminen on yleensä lasilaatikkouudelleenkäyttöä - yliluokan lähdekoodia on tutkittava, jotta aliluokka saadaan toimimaan halutulla tavalla.

ESIMERKKI 3: Palataan esimerkin 1 sisäänkirjautumisikkunaan. Kun halutaan sopeuttaa ikkuna sellaiseksi, jossa on mahdollisuus valita kryptataanko yhteys palvelimeen, voimme periyttää alkuperäisestä komponentista `LoginDialog` uuden komponentin `CryptLoginDialog`. Käytämme JavaBeans-komponenttimallia[7] ja näin ollen luonnollisesti Java-kieltä. Oletetaan, että käytössä on komponentin LoginDialog lähdekoodi:

```
package logindialog;

import java.awt.*;
import java.awt.event.*;
import borland.jbcl.layout.*;

public class LoginDialog extends Dialog
{
  XYLayout xYLayout1 = new XYLayout();
  Panel panel1 = new Panel();
  Button buttonOK = new Button();
  Label labelUserName = new Label();
  Label labelPassword = new Label();
  TextField textFieldUserName = new TextField();
```

[7] Tässä esitetyt luokat eivät kelpaa sellaisenaan JavaBeans-komponentiksi, koska luokalla `CryptLoginDialog` ei ole parametritonta konstruktoria. Sitä ei ole, koska luokasta `java.awt.Dialog` periytyvät luokat tarvitsevat `Frame`-tyyppisen olion luontinsa yhteydessä. Jotta dialogista saisi tehtyä JavaBeans-komponentin, on käytettävä ympäröimistä (kts. luku 5.2.7).

```
  TextField textFieldPassword = new TextField();

  public LoginDialog(Frame frame, String title, boolean mo-
dal)
  {
    super(frame, title, modal);
    try
    {
      jbInit();
      add(panel1);
      pack();
    }
    catch (Exception e)
    {
      e.printStackTrace();
    }
  }

  public LoginDialog(Frame frame)
  {
    this(frame, "Log in to the system", false);
  }

  public LoginDialog(Frame frame, String title)
  {
    this(frame, title, false);
  }

  public void jbInit() throws Exception
  {
    xYLayout1.setWidth(279);
    xYLayout1.setHeight(133);
    buttonOK.setLabel("Log in");
    buttonOK.addActionListener(new Login-
Dialog_buttonOK_actionAdapter(this));
    labelUserName.setText("Enter user name:");
    labelPassword.setText("Enter password:");
    panel1.setLayout(xYLayout1);
    panel1.add(buttonOK, new XYConstraints(108, 98, 72,
24));
    panel1.add(labelUserName, new XYConstraints(11, 21,
106, 20));
    panel1.add(labelPassword, new XYConstraints(11, 46,
106, 24));
    panel1.add(textFieldUserName, new XYConstraints(117,
21, 149, 24));
    panel1.add(textFieldPassword, new XYConstraints(117,
46, 149, -1));
  }

  public Panel getPanel()
  {
   return panel1;
```

```
  }

  void buttonOK_actionPerformed(ActionEvent e)
  {
   // some code to handle this
  }
}

class LoginDialog_buttonOK_actionAdapter implements ja-
va.awt.event.ActionListener{
  LoginDialog adaptee;

  LoginDialog_buttonOK_actionAdapter(LoginDialog adaptee) {
    this.adaptee = adaptee;
  }

  public void actionPerformed(ActionEvent e) {
    adaptee.buttonOK_actionPerformed(e);
  }
}
```

Periyttäminen voitaisiin toteuttaa vaikkapa seuraavasti:

```
package logindialog;

import java.awt.Checkbox;
import java.awt.Dimension;
import java.awt.event.*;
import borland.jbcl.layout.*;


public class CryptLoginDialog extends LoginDialog
{

 private Checkbox      m_CryptCheckbox;
 private boolean       m_bCryptConnection = false;

 public CryptLoginDialog(Frame frame, String title, boolean
      modal)
  {
    super(frame, title, modal);
  }

  public CryptLoginDialog(Frame frame)
  {
    this(frame, "Log in to the system", false);
  }

   public void jbInit() throws Exception
   {
    super.jbInit();
```

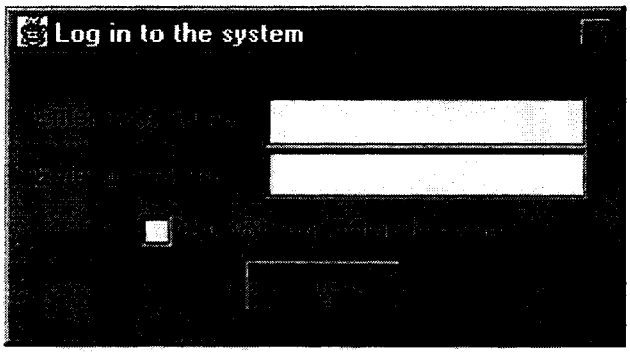
```
    m_CryptCheckbox = new Checkbox();
    m_CryptCheckbox.setSize(new Dimension(244, 318));
    m_CryptCheckbox.setLabel("Encrypt connection to
      server");
    m_CryptCheckbox.addItemListener(new
      CryptLoginDialog_m_CryptCheckbox_itemAdapter(this));
   panel1.add(m_CryptCheckbox,new XYConstraints(60, 72,
      200, 24));
  }

  public boolean isCryptOn()
  {
     return m_bCryptConnection;
  }

  void m_CryptCheckbox_itemStateChanged(ItemEvent e)
  {
       m_bCryptConnection = m_CryptCheckbox.getState();
  }
}

class CryptLoginDialog_m_CryptCheckbox_itemAdapter imple-
ments java.awt.event.ItemListener
{
  CryptLoginDialog adaptee;

  CryptLogin-
Dialog_m_CryptCheckbox_itemAdapter(CryptLoginDialog adap-
tee)
{
    this.adaptee = adaptee;
}

  public void itemStateChanged(ItemEvent e)
  {
    adaptee.m_CryptCheckbox_itemStateChanged(e);
  }
}
```

Uusi komponentti perii kaikki `LoginDialogin` ominaisuudet, sekä määrittelee lisäksi operaation `isCryptOn()`, joka kertoo valitsiko käyttäjä kryptauksen. Lisäksi uuden komponentin toteutukseen lisätään käyttöliittymäkomponentti (checkbox), josta käyttäjä voi valita kryptauksen päälle tai pois sekä boolean-tyyppinen muuttuja `bCryptOn`, johon käyttäjän tekemä valinta talletetaan. Edelleen komponentti on saatava reagoimaan hiiren liikkeisiin siten, että se huomaa kun käyttäjä painaa valintakomponenttia. Esimerkiksi JavaBeans-mallissa tämä voitaisiin tehdä siten, että määri-

tellään uusi luokka CryptLoginDialog_m_CryptCheckbox_itemAdapter, joka toteuttaa rajapinnan java.awt.event.ItemStateListener, luodaan tämän luokan ilmentymä ja asetetaan se kuuntelemaan valintakomponenttia. Kun tämä ilmoittaa kuuntelijalle, että sen tila on muuttunut, kutsuu kuuntelija ikkunakomponentin tiettyä metodia, jossa asetetaan bCryptOn-muuttujan arvo vastaamaan valintakomponentin tilaa. Viimeisen funktion toteutusta lukuunottamatta JavaBeans-komponenttimallia tukevat kehitysympäristöt, kuten JBuilder ja VisualCafe luovat tarvittavan koodin automaattisesti.

Esimerkissä 3 on yksinkertaisuuden vuoksi käytetty JBCL-kirjaston mukana tulevia XYLayout ja XYConstraints-luokkia komponenttien sijoitteluun ikkunan sisällä absoluuttisin koordinaatein.

CryptLoginDialog kyettiin luomaan muuttamatta yliluokkaa LoginDialog. Yliluokan toteutuksen tunteminen oli silti välttämätöntä, joten kyse oli uudelleenkäytöstä lasilaatikkona. Oli tiedettävä, että yliluokka järjestää ikkunan sisällön JBCL-kirjaston XYLayout ja XYConstraints-luokkien avulla, mikä ei ilmene LoginDialog-luokan rajapinnasta. Uuden komponentin sijoittaminen ikkunaan oli lisäksi ongelmallista, koska yliluokassa käyttöliittymäkomponentit sijoitettiin ikkunaan absoluuttisia koordinaatteja käyttäen, ja uusi komponentti oli sijoitettava niiden väliin. Tässä tapauksessa se sattui mahtumaan tekstikenttien ja painonapin väliin (katso kuva 7). Vaikka yliluokka olisi käyttänyt java.awt-pakkauksesta löytyviä luokkia java.awt.LayoutManager-rajapinnan toteuttavia luokkia komponenttien sijoitteluun (kuten olisi suositeltavaa, jotta käyttöliittymäkomponentit voitaisiin sijoitella suhteellisesti toisiinsa nähden), edellyttäisi uusien käyttöliittymäkomponenttien lisääminen aliluokkaan yliluokan lähdekoodin tutkimista.



KUVA 7 `CryptLoginDialog`-komponenttissa ikkunaan on lisätty valintakomponentti, josta voi valita kryptataanko yhteys palvelimelle.

5.2.5 Periyttämiseen liittyvät ongelmat

Komponenttien välinen periyttäminen muodostaa riippuvuussuhteen komponenttien välille. Näin ollen on kyseenalaista, tulisiko periyttämistä käyttää lainkaan komponenttien välillä. Kolme merkittävintä komponenttimallia eroavat tässä asiassa toisistaan. CORBA ja JavaBeans mahdollistavat komponenttien välisen periyttämisen. JavaBeans ei kuitenkaan mahdollista toteutuksen moniperintää, luokka voi periä toteutuksen vain yhdeltä luokalta. Luokka voi kuitenkin toteuttaa monta rajapintaa. COM mahdollistaa periyttämisen vain komponenttien sisällä, mikäli ne on toteutettu jollain oliokielellä.

M. Franzin mukaan oliokeskeiset kielet, joissa käytetään toteutusperintää johtavat uudelleenkäyttöön sovelluskehysten avulla ts. uusia ohjelmistoja rakennetaan periyttämällä muutama sovelluskehyksen luokka ja käyttämällä uudelleen sovelluskehyksen muita osia sellaisenaan. Sovelluskehys on kuitenkin itsessään monimutkainen sovellus, jonka sisällä (ja sovelluskehyksen sekä rakennettavan sovelluksen välissä) voi olla monimutkaisia riippuvuussuhteita, joten sovelluskehyksen vaihtaminen on vaikeaa. Franzin mukaan komponenttien välillä ei tulisi olla periytymissuhteita [Franz].

Lisäksi periyttäminen voi johtaa ns. *särkyvän yliluokan ongelmaan* (the fragile base class problem). Tämä tarkoittaa sitä, että kun yliluokka muuttuu, se voi rikkoa olemassa olevat aliluokat ts. olemassa olevat aliluokat eivät enää toimi muuttuneen yliluokan kanssa. Ongelma voidaan jakaa syntaktiseen ja semanttiseen osaan [Szyperski 1998]. Ongelma ei varsinaisesti liity pelkästään periyttämiseen. Syntaktinen ongelma liittyy yleisesti rajapintojen muuttumiseen ja semanttinen ongelma rajapinnan takana olevan toiminnallisuuden muuttumiseen.

Syntaktinen ongelma seuraa yliluokan muuttuneen rajapinnan binääritason epäyhteensopivuudesta olemassa olevien aliluokkien kanssa. Yhteensopimattomuutta aiheuttavia muutoksia ovat mm. metodien nimien tai parametrilistojen muuttuminen tai metodien poistuminen yliluokasta.

Jotkut muutokset, kuten metodien siirtyminen luokkahierarkiassa ylöspäin tai uusien metodien lisääminen, voitaisiin aina selvittää kääntämällä aliluokka uudestaan uuden yliluokan kanssa. Tavoitteena on kuitenkin, että aliluokka toimii uuden yliluokan kanssa ilman että sitä tarvitsee kääntää uudestaan mm. siksi, että aliluokan käyttäjällä ei välttämättä ole käytettävissään yliluokan lähdekoodia. Ratkaisuna tähän esim. IBM:n komponenttimalli System Object Model (SOM) alustaa metoditaulut (method dispatch tables) ohjelmakoodin latauksen yhteydessä. Tämä ei luonnollisestikaan auta kaikkien syntaktisten muutosten kohdalla. Esimerkiksi yliluokassa tapahtuva metodin jakaminen kahdeksi metodiksi tai metodin parametrilistan muuttaminen voivat edelleen rikkoa olemassaolevat aliluokat [Szyperski 1998].

Semanttinen ongelma seuraa, jos yliluokan käyttäytyminen muuttuu odottamattomalla tavalla eli yliluokan toteutus muuttuu. Jos rajapinta pysyy kuitenkin samana, voi komponentti jatkaa toimintaansa mutta saattaa toimia virheellisesti. Tätä ongelmaa on vaikea välttää muutoin kuin huo-

lellisella suunnittelulla. Osaksi juuri tämän ongelman vuoksi komponenttien välinen periyttäminen on ongelmallista. Ongelmaan on kuitenkin esitetty joitain ratkaisuehdotuksia.

Mikhaljov ja Sekerinski esittävät joukon sääntöjä sovelluskehysten ja komponenttien suunnittelijoille, joita noudattamalla yliluokkaan tehdyt muutokset eivät riko aliluokkia [Mikhajlov & Sekerinski 1997].

Kiczales ja Lamping ovat esittäneet ongelman ratkaisuksi *erikoistumisrajapintoja* luokan ja sen aliluokkien välille. Esim. C++ ja Java-kielissä tämän voi toteuttaa määrittelemällä metodit `protected`-avainsanaa käyttäen. Lamping on myös ehdottanut metodien jakamista *ryhmiksi*: kaikki toisistaan riippuvat metodit muodostavat ryhmän ja jos riippuvuudet ovat asyklisiä, ryhmät muodostavat *kerroksia* [Szyperski 1998].

Stata ja Guttag esittävät, että luokka jaetaan ryhmiin, jotka koostuvat luokan metodeista ja attribuuteista. Ryhmät ovat kuin luokan sisäisiä erillisiä luokkia, paitsi että niillä on yhteinen *self*. Mikäli ryhmistä A, B ja C koostuvan luokan M aliluokka N haluaa korvata ryhmässä A olevan metodin, sen täytyy toteuttaa itse kaikki ryhmän A metodit. Ryhmän metodien toteutuksen voi siis periä vain, jos ei korvaa yhtään niistä. Tarkoituksena on, että luokan suunnittelija jakaa metodit ryhmiin siten, että semanttisesti vaaralliset metodien korvaamiset aliluokkien toimesta voitaisiin estää [Stata & Guttag 1995]. Clemens Szyperski huomauttaa, että mikäli luokka koostuu vain yhdestä ryhmästä, toteutuksen perintää ei voida käyttää luokkaa sopeutettaessa. Luokkaa joko käytetään uudelleen sellaisenaan, tai sitten toteutus korvataan kokonaan (jolloin luokka N kannattaa periyttää suoraan M:n yliluokasta). Szyperski huomauttaa edelleen, että useasta ryhmästä koostuvan luokan voi purkaa useaksi luokaksi käyttämällä kompositiota [Szyperski 1998].

Steyaert, Lucas, Mens ja D'Hondt ovat Szyperskin mukaan ehdottaneet särkyvän yliluokan ongelman semanttisen osan ratkaisuksi *uudelleenkäyttösopimuksia* (reuse contracts). Uudelleenkäyttösopimukset määrittelevät, mitä metodeja aliluokat voivat käyttää tai syrjäyttää. Uudelleenkäyttösopimukset voivat myös sisältää *muokkausoperaattoreita* (modification operators), jotka määrittelevät millaisia muutoksia metodeihin voidaan tehdä. Mahdollisiin operaattoreihin kuuluvat konkretisointi/abstraktio, laajennus/poisto (extension/cancellation) ja tarkentaminen/karkeuttaminen (refinement/coarsening). Steyaert väittää, että olemassaolevien aliluokkien yhteensopivuus yliluokkaa muutettaessa selviää automaattisesti käytettyjen uudelleenkäyttöoperaattorien sarjan perusteella. Szyperski huomauttaa kuitenkin, että koska aliluokat voivat luottaa myös yliluokan metodikutsujen järjestykseen, uudelleenkäyttösopimukset Steyaertin ym. esittämässä muodossa eivät voi kokonaan ratkaista särkyvän yliluokan ongelman semanttista puolta [Szyperski 1998].

Särkyvän yliluokan ongelmasta johtuen mm. Szyperski suosittelee toteutusperinnän liiallisen käytön välttämistä. Toteutusperinnän sijasta voidaan usein käyttää ympäröimistä (katso kohta 5.2.7) [Szyperski 1998]. Erityisesti näin voisi ajatella olevan, kun kyse on satunnaisesta periyttämisestä [Sakkinen 1992], jolloin komponenttien mallintaminen ilmiöiden välillä ei reaalimaailmassa ole *is-a*-suhdetta.

5.2.6 Implisiittinen alityypitys

Jotkin oliokielet tukevat *implisiittistä alityypitystä* (implicit subtyping). Implisiittinen alityypitys tarkoittaa sitä, että tyyppiä A voi olla tyypin B laillisena ylityyppinä ilman eksplisiittisesti määriteltyä periytymissuhdetta, mikäli rajapinnat täsmäävät. Implisiittinen alityypitys ei kuitenkaan ratkaise väärin nimettyjen tai jaettujen metodien tuomia ongelmia. Implisiittinen alityypitys on myös teoreettisesti ongelmallinen, koska se voi luoda

alityyppisuhteen, vaikka semanttista suhdetta ei ole. Esimerkiksi tyypeille `Cowboy` ja `Drawable` voi olla määriteltynä metodi `draw`, joka tarkoittaa täysin eri asiaa [Ciupke & Schmidt 1996].

5.2.7 Ympäröiminen

Ympäröiminen (wrapping) tarkoittaa sitä, että sopeutettavan komponentin ympärille rakennetaan ympäröijäkomponentti (wrapper), jonka tehtävänä on tarjota halutunlainen rajapinta ja siirtää (forward) metodikutsut ympäröivästä rajapinnasta ympäröidylle komponentille. Ympäröimistä kutsutaan joskus myös *kompositioksi* (object composition) tai *aggregaatioksi* (aggregation). Ympäröivää komponenttia kutsutaan myös ulommaksi komponentiksi ja ympäröityjä komponentteja sisemmiksi komponenteiksi [Szyperski 1998]. Sisemmät komponentit voidaan tietyissä tapauksissa ajatella ulomman komponentin osiksi. Tällöin kyseessä on *has-a* suhde ulomman ja sisempien komponenttien välillä [Budd 1997].

Käytännössä ympäröiminen voidaan toteuttaa joko siten, että ulommassa komponentissa on viittaus sisempiin komponentteihin tai siten, että sisemmät komponentit ovat ulomman komponentin ilmentymämuuttujina.

ESIMERKKI 4: Edellisessä esimerkissä komponenttia sopeutettiin periyttämällä. Sama sopeutus voidaan tehdä myös ympäröimällä:

```
package logindialog;

import java.awt.Checkbox;
import java.awt.Dimension;
import java.awt.event.ItemEvent;
import borland.jbcl.layout.XYConstraints;

public class CryptLoginDialog2 {

  private LoginDialog m_loginDialog = null;
  private Checkbox      cryptCheckbox;
```

```
  private boolean        cryptConnection = false;
  private Frame          frame;

  public CryptLoginDialog2()
  {
    cryptCheckbox = new Checkbox();
    cryptCheckbox.setEnabled(true);
    cryptCheckbox.setSize(new Dimension(244, 318));
    cryptCheckbox.setLabel("Encrypt connection to server");
    cryptCheckbox.addItemListener(new CryptLogin-
Dialog2_m_CryptCheckbox_itemAdapter(this));

  }

  public void show()
  {
   m_loginDialog = new LoginDialog(frame);
   m_loginDialog.getPanel().add(cryptCheckbox,new XYConst-
raints(60, 72, 200, 24));
   m_loginDialog.show();
  }

  void m_CryptCheckbox_itemStateChanged(ItemEvent e)
  {
   cryptConnection = cryptCheckbox.getState();
  }

  public void setFrame(Frame frame)
  {
   this.frame = frame;
  }
}

class CryptLoginDialog2_m_CryptCheckbox_itemAdapter imple-
ments java.awt.event.ItemListener{
  CryptLoginDialog2 adaptee;

  CryptLogin-
Dialog2_m_CryptCheckbox_itemAdapter(CryptLoginDialog2 adap-
tee) {
    this.adaptee = adaptee;
  }

  public void itemStateChanged(ItemEvent e) {
    adaptee.m_CryptCheckbox_itemStateChanged(e);
  }
}
```

Yksi ympäroimisen eroista periyttämiseen verrattuna on sen dynaaminen luonne. Ympäröiminen on ilmentymien ja periytyminen tyyppien välinen

suhde. Ympäroimisellä voidaan ratkaista sellaisiakin ongelmia, joita ei voi ratkaista periyttämisen avulla.

ESIMERKKI 5: Olkoon käytettävissä kolme eri valmistajilta ostettua graafista komponenttia. Ne olisi sopeutettava piirto-ohjelmaan, joka olettaa piirrettävien olevan tyyppiä `IDrawable`. Komponentit eivät sellaisinaan kelpaa piirto-ohjelmalle ja ne ostettiiin ilman lähdekoodia, joten niitä ei saada periytymään tyypistä `IDrawable` (tai toteuttamaan rajapintaa `IDrawable`). Ongelma voidaan ratkaista luomalla kolme ympäröijää, jotka toteuttavat rajapinnan `IDrawable`. Kukin ympäröijä välittää sille lähetetyt viestit (metodikutsut) ympäröimälleen komponentille.

Ympäröiminen ei ole läpinäkyvä sopeuttamistekniikka. Toisin sanoen korvattavuusperiaate ei toteudu ympäröijän ja ympäröidyn komponentin välillä.

Ympäröijät tuovat myös redundanssia, koska kun komponentin rajapinta muuttuu, on ympäröijän rajapintaa muutettava vastaavasti. Toisaalta ongelma on luonteeltaan melko yksinkertainen: ohjelmointiympäristöt voisivat luoda ja päivittää ympäröijiä automaattisesti.

Ympäröintiin liittyy kuitenkin muitakin ongelmia. Mikäli ohjelmointikieli ei tue delegaatiota, ympäröijät eivät voi syrjäyttää (override) ympäröidyn komponentin metodeja, paitsi jos näitä metodeita kutsutaan ainoastaan ympäröidyn komponentin ulkopuolelta. Ympäröidyn komponentin kutsuessa omia metodejaan nämä kutsut eivät kulje ympäröijän rajapinnan kautta.

5.2.8 Sekoitteiden käyttö

Ympäröiminen ja periyttäminen voidaan yhdistää käyttämällä moniperintää. Tällöin komponentti A sopeutetaan periyttämällä siitä uusi kompo-

nentti B, joka perii myös halutun rajapinnan. Näin uusi komponentti *sekoittaa* (mixes in) halutun rajapinnan sopeutetun komponentin rajapintoihin. Tämä toimii, jos kyseisen komponentin A ilmentymiä luodaan ainoastaan sopeutetun ohjelmistokomponentin ulkopuolella. Kuitenkin jos sopeutetussa koodissa luodaan sopeutetun komponentin A ilmentymiä, ne pitäisi taas sopeuttaa tyypin B ilmentymiksi. Yleisesti käytössä olevissa oliokielissä eksplisiittinen tyyppimuunnos B:ksi ei auta, koska ajonaikainen järjestelmä testaa luotiinko olio B:ksi eikä sitä, kelpaisiko se B:ksi.

Tämä ongelma voidaan osittain ratkaista *oliotehtailla* (object factories). Tällöin jokaisen ilmentymän luonnin yhteydessä kutsutaan oliotehdasta, jonka tehtävänä on oikeantyyppisten olioiden luonti. Komponentin sopeuttaja voi korvata oliotehtaan omalla versiollaan tai sopeuttaa oliotehdasta. Esimerkiksi kun alkuperäinen oliotehdas loi A-olioita, voidaan se korvata sellaisella, joka palauttaa B-olioita.

Kuitenkin nyt jos A:n `get()`-kutsu palauttaa A:n ilmentymän, joudumme syrjäyttämään sen B:ssä `get()`-metodilla, joka palauttaa B:n ilmentymän. Jotkut kielet kuitenkin estävät syrjäytetyn funktion palautustyypin muuttamisen, jolloin menetelmä ei toimi. Lisäksi jos komponenttivalmistaja X on korvannut komponenttivalmistaja Y:n oliotehtaan omallaan ja me haluamme edelleen korvata sen, yksi oliotehdas ei riitä. Tästä voidaan tehdä se johtopäätös, että oliotehtaisiin perustuva lähestymistapa ei näyttäisi toimivan suuremmassa mittakaavassa [Ciupke & Schmidt 1996].

5.2.9 Binäärinen sopeuttaminen

Ralph Keller ja Ulf Hölzle [Keller & Hölzle 1997] esittävät *binääristä sopeuttamista* (binary adaptation), joka perustuu binääristen tiedostojen suoraan muokkaamiseen. Kyse on ilmeisesti lähinnä komponenttien rajapinnan sopeuttamisesta. Tämä on mahdollista, jos binäärimuodossa oleva

komponentti sisältää riittävästi informaatiota rajapinnastaan. Riittävästi informaatiota sisältää esimerkiksi tavukoodiksi käännetty Java-luokka. Keller ja Hölzle ovat kehittäneet ohjelmiston, joka osaa lukea muutostietoja sisältäviä *delta-tiedostoja*, jotka yhdistetään Java-kääntäjän kääntämään luokkaan. Tuloksena syntyy edelleen tavukoodimuodossa oleva sopeutettu luokka. Keller ja Hölzle eivät kuitenkaan tuo esiin sitä, ettei tässä ole käsitteellisesti paljoa eroa lähdekoodin suoraan muokkaamiseen. Mikäli binääritiedosto sisältää tarpeeksi informaatiota binääriseen sopeuttamiseen, riittänee sama informaatio myös lähdekoodin generointiin binääritiedostosta.

5.2.10 Superimpositio

Jan Bosch esittää joukon vaatimuksia komponenttien sopeutukseen käytettäville tekniikoille. Komponentin sopeutuksen tulisi olla läpinäkyvä, sen tulisi käsitellä komponenttia mustana laatikkona, sen tulisi olla yhdistettävissä muihin sopeutuksiin, konfiguroitava ja uudelleenkäytettävä. Mikään perinteisistä sopeuttamismenetelmistä (leikkaa-liimaa, periyttäminen, ympäröiminen) ei täytä kaikkia näitä vaatimuksia ja Bosch esittääkin *superimpositiomenetelmän*, joka täyttää kaikki hänen esittämänsä vaatimukset. Superimpositio käsittelee komponentteja mustina laatikoina ja mahdollistaa valmiiksi määritellyn, mutta muokattavissa olevan toiminnallisuuden (superimpositio-entiteetti) lisäämisen komponenttiin. Täsmällisemmin superimpositio S muodostuu toiminnallisuuden B lisäämisestä komponenttiin O, jolloin komponentin O toiminnallisuus muuttuu B:n osalta. Superimpositio eroaa periyttämisestä siten, että yksittäinen S voi muuttaa komponenttia useasta eri kohdasta (esim. jos komponentti koostuu useasta eri luokasta, useampi luokka voi muuttua). Superimpositio mahdollistaa myös useiden superimpositio-entiteettien yhdistämisen samaan komponenttiin. Kunkin superimpositio-entiteetin aikaansaama sopeutus muodostaa käsitteellisesti uuden kerroksen komponentin ympärille [Bosch].

Superimpositiosta ei ole tietääkseni muita käytännön toteutuksia kuin Boschin Layered Object Model. Käsitteellisesti superimpositio ei kuitenkaan ilmeisesti ratkaise periyttämiseen liittyvää rikkonaisen yliluokan ongelmaa. Koska uudet toiminnallisuudet muodostavat kerroksia sopeutetun komponentin ympärille, superimpositio voi myös tulla kalliiksi suorituskyvyn kannalta.

5.3 Komponenttien yhdistäminen

Kun tarvittavat komponentit on hankittu ja tarvittaessa sopeutettu kohdeympäristöönsä, voidaan ne yhdistää toimivaksi järjestelmäksi. Komponenttien yhdistäminen edellyttää, että komponentit voivat kommunikoida toistensa kanssa. Tavoitteena on, että komponenttien yhdistäminen olisi mahdollisimman joustavaa eli komponentit olisi sidottu mahdollisimman löyhästi toisiinsa. Siksi komponenttien ei tulisi viitata toisiin konkreettisiin komponentteihin, vaan ainoastaan niiden rajapintoihin.

Tässä kohdassa pyritään esittelemään erilaisia tekniikoita komponenttien yhdistämiseksi ja esitetään Sametingerin [Sametinger 1997] esittämä luokittelu.

5.3.1 Sisäinen ja ulkoinen kompositio

Komponenttien yhdistäminen eli *kompositio*[8] voidaan jakaa ulkoiseen ja sisäiseen [Sametinger 1997]. *Sisäisellä kompositiolla* tarkoitetaan komponenttien integrointia järjestelmän osaksi. Esimerkiksi lähdekoodin kääntäminen ja linkittäminen ohjelmiston osaksi on sisäistä kompositiota. *Ulkoisessa kompositiossa* puolestaan komponentit toimivat itsenäisesti ja

---

[8] Sanaa kompositio käytetään tässä eri merkityksessä kuin ympäröimisen yhteydessä.

kenne ei estä ketä tahansa lisäämästä pakkauksiin omia luokkiaan. Szyperski esittääkin Java-kieleen lisäykseksi *suljettua* pakkaus (closed package) -käsitettä. Lisäksi Javan pakkauksilla ei ole erikseen määriteltyjä rajapintoja, vaan niiden sisään- ja ulostulevat rajapinnat koostuvat pakkauksen sisältämien luokkien sisään- ja ulostulevista rajapinnoista.

Jotta eri ohjelmointikielillä kirjoitetut ja eri käyttöjärjestelmissä toimivat komponentit voisivat kommunikoida keskenään, on niillä oltava jokin yhteinen tekijä. Oliomallitason kompositio perustuu komponenttimalleihin, jotka standardoivat komponenttien rajapinnat ja niiden välisen kommunikoinnin ja näin mahdollistavat komponenttien yhdistämisen. Komponentin rajapintojen on oltava tietyn standardin mukaisia, jotta se voisi käyttää hyväkseen komponenttimallin tarjoamia palveluita. Object Management Groupin CORBA-standardissa komponenttien rajapinta tulee määritellä IDL-kielellä (Interface Description Language) [Orfali, Harkey & Edwards 1996]. Microsoftin COM puolestaan edellyttää tiettyä binääritason standardia, jonka mukaan komponentit voivat kutsua toistensa rajapintoja [Szyperski 1998]. JavaSoftin JavaBeans-standardi edellyttää komponenttien kirjoittamista Java-kielellä ja metodien nimeämistä tietyn nimeämisstandardin mukaisesti [Vanhelsuwé 1997].

Yhdistäminen voi myös perustua tiettyyn komponenttialustaan (component platform). Komponenttialustalla tarkoitetaan ohjelmistoa, jonka varassa komponentti toimii [Sametinger 1997]. Komponenttialusta voi olla esim. käyttöjärjestelmä, ikkunointijärjestelmä tai joukko muita komponentteja. Komponenttialustat voidaan jakaa suoritusaikaisiin alustoihin sekä kompositioalustoihin.

Jotta kaksi komponenttia voitaisiin yhdistää toisiinsa, on niillä oltava samat komponenttialustat. Ohjelmistokomponentit ovat sitä uudelleenkäytettävämpiä, mitä vähemmän komponenttialustoja niillä on ja mitä siirrettä-

vämpiä nämä komponenttialustat ovat [Sametinger 1997]. On epäselvää, miksi Sametinger sijoittaa tiettyyn komponenttialustaan perustuvan yhdistämisen esim. komponenttimalliin perustuvan yhdistämisen yläpuolelle, koska komponenttimallit tarjoavat nimenomaan riippumattomuutta ainakin joistain komponenttialustoista ja toisaalta Sametinger pitää avoimen alustan kompositiota komposition ylimpänä tasona.

Avoimen alustan (open platform) kompositiota tukeva komponentti on periaatteessa mahdollista yhdistää vapaasti mihin tahansa muuhun komponenttiin [Sametinger 1997]. Ainakaan toistaiseksi komponenttiteknologia ei kuitenkaan ole vielä edennyt näin pitkälle ja on kyseenalaista, onko tämä tulevaisuudessakaan teknisesti mahdollista. Mikäli avoimen alustan kompositio määritellään niin kuin Sametinger sen tekee, se ei voi perustua siihen, että komponentit tukevat jotain standardia, koska tällöin komponentteja ei voisi yhdistää niihin komponentteihin, jotka eivät tue tätä standardia eikä kyse siis olisi avoimen alustan kompositiosta.

5.3.3 Visuaalinen kompositio

Viimeisen vuosikymmenen aikana markkinoille on tullut useita *visuaaliseen kompositioon* perustuvia ohjelmankehitysympäristöjä, kuten Microsoftin Visual Basic ja Visual J++, Inprisen Delphi ja JBuilder sekä IBM:n VisualAge. Nämä mahdollistavat ohjelmistokomponenttien ilmentymien yhdistämisen toisiinsa graafisesti hiirtä käyttäen. Ilmentymien yhdistäminen toisiinsa perustuu yleensä ulosmeneviin ja sisääntuleviin rajapintoihin sekä tapahtumaohjattuun (event-based) kommunikaatioon. Tavoitteena on usein ollut minimoida sen ohjelmakoodin määrä, joka ohjelmointiympäristön käyttäjän pitää kirjoittaa. Visuaalisen komposition yleisin käyttötarkoitus on edelleen graafisten käyttöliittymien rakentaminen, mutta sitä käytetään myös esim. tietokantayhteyksien luontiin.

Visuaalisessa ohjelmankehitysympäristössä komponentin ilmentymä voidaan yhdistää toiseen, vaikka komponentin toteutus olisikin piilotettu. Ohjelmistokomponentista näkyy vain sen rajapinta, esim. Inprisen JBuilder paljastaa yksittäisestä komponentista sen ominaisuudet, metodit ja tapahtumat sekä komponenttia kuvaavan ikonin. Mikäli komponentin lähdekoodi on saatavilla, sen tarkastelu ja muokkaaminen on toki mahdollista.

Nämä ympäristöt perustuvat yleensä johonkin tiettyyn komponenttimalliin. Esimerkiksi Visual J++ perustuu COM:iin ja JBuilder JavaBeansiin.

Clemens Szyperskin mukaan visuaalinen kompositio sopii käytettäväksi kun käsiteltävien komponenttien määrä on suhteellisen pieni ja komponenttien väliset suhteet ja kanssakäyminen yksinkertaista. Muussa tapauksessa abstraktimpien menetelmien (perinteinen ohjelmointi) käyttö on taloudellisempaa tilankäytön kannalta, ohjelmia on helpompi muuttaa ja komponenttien väliset suhteet voidaan esittää täsmällisemmin. Kun visuaalista kompositiota käytetään muuhun kuin selkeästi visuaalisten asioiden (esim. käyttöliittymät) luontiin, lähestymistavan rajoitukset tulevat vastaan. Ongelmat johtuvat siitä, että abstraktien käsitteiden käsittely rajoitetulla määrällä tai ilman symboleita on vaikeaa [Szyperski 1998].

Tämänhetkisten visuaalisten kompositiotyökalujen käyttö edellyttää myös, että rakennettava ohjelmisto on riittävän staattinen, koska kompositio voi kohdistua vain niihin komponenttien ilmentymiin, joiden olemassaolo tiedetään käännösaikana [Szyperski 1998].

On myös huomattava, että vaikka visuaaliseen kompositioon perustuvien ympäristöjen myyjät väittävät välineiden nostavan ohjelmistotyön tuottavuutta kirjoitettavien koodirivien määrän vähetessä, ei tämä välttämättä ole merkittävä säästö ohjelmiston elinkaarikustannusten kannalta. Tutkimusten mukaan ohjelmiston elinkaarikustannuksista vain 6% muodostuu ohjelmointivaiheessa ja peräti 66% ohjelmiston ylläpitovaiheessa [Schach

1990, katso kuva 11]. Näin ohjelmistovaiheen kustannussäästöt voivat kostautua myöhemmässä vaiheessa, jos seurauksena muutosten teko ohjelmakoodiin on vaikeampaa.

5.4 Yhteenveto komponenttiteknologioista

Komponenttipohjaisen ohjelmistotuotannon järkevä soveltaminen edellyttää, että olemassaolevan komponentin hankkiminen on helpompaa, kuin uuden komponentin kirjoittamisen. Komponentteja voidaan hankkia organisaation sisäisistä komponenttivarastoista sekä markkinoilta. Tehokas haku edellyttäisi keinoja kuvata komponentin toiminnallisuus sekä eitoiminnallisia ominaisuuksia.

Komponentteja ei yleensä voida uudelleenkäyttää sellaisenaan. Komponenttien sopeuttaminen tarkoittaa komponentin toiminnallisuuden muuttamista käyttäjän tarpeiden mukaisesti. Sopeuttamiseen voidaan käyttää erilaisia tekniikoita, kuten leikkaa-liimaa-tekniikkaa, ympäröimistä ja periyttämistä. Käytetty komponenttimalli ja ohjelmointikieli voivat rajoittaa sopeuttamistekniikan valintaa. Sopeuttamistekniikka on läpinäkyvä, jos sopeutetun komponentin ilmentymä voidaan sijoittaa alkuperäisen paikalle sellaisenaan. Toiset sopeuttamistekniikat tarkastelevat sopeutettavaa komponenttia mustana laatikkona, jolloin lähdekoodia ei tarvita sopeuttamiseen. Jotkut sopeuttamistekniikat, kuten periyttäminen, vaativat yleensä käytännössä alkuperäisen komponentin lähdekoodin tarkastelua, jolloin alkuperäistä komponenttia tarkastellaan lasilaatikkona. Leikkaa-liimaa-tekniikka tarkastelee alkuperäistä komponenttia valkoisena laatikkona eli lähdekoodia muutetaan suoraan. Sopeuttamistekniikat voivat johtaa yhteensopivuusongelmiin myöhemmin, kun sopeutettavasta komponentista julkaistaan uusi versio. Yhteensopivuusongelmat voivat olla syntaktisia tai semanttisia. Syntaktinen yhteensopivuusongelma johtaa helposti havaittavaan ajonaikaiseen virheeseen (sopeutettu komponentti ei enää toimi, koska se ei ole binääritasolla yhteensopiva sopeutettavan komponentin raja-

pinnan muututtua). Semanttiset yhteensopivuusongelmat ovat vaikeampia havaita, koska sopeutettu komponentti voi jatkaa toimintaansa, mutta toimia virheellisesti. Taulukossa 4 esitetään yhteenveto tässä luvussa käsitellyistä tekniikoista edellämainittujen ominaisuuksien valossa.

Sopeuttamistekniikka	**Läpinäkyvä**	**Lähdekoodi**	**Yhteensopivuus jatkossa**
Leikkaa-liimaa	ei	muutetaan	ei
Periyttäminen	kyllä	oltava tutkittavissa(toteutuspe-rintä)	ei välttämättä
Ympäröiminen	ei	ei tarvita	ympäröijää päivitettävä
Sekoitteet	kyllä	ei tarvita	ei välttämättä
Binäärinen sopeuttaminen	kyllä	ei tarvita	ei välttämättä
Superimpositio	kyllä	ei tarvita	ei välttämättä

TAULUKKO 4 Sopeuttamistekniikoiden vertailua

Voidaan huomata, että mikäli komponentin lähdekoodia ei ole käytettävissä, mahdolliset tekniikat komponentin sopeuttamiseen ovat rajallisemmat. Tällöin on rajattu pois toisaalta paljon ongelmia aiheuttava, mutta toisaalta paljon käytetyt sopeuttamisen muoto eli leikkaa-liimaa-tekniikka. Lisäksi periyttäminen on vaikeaa.

Kun tarvittavat komponentit on löydetty ja niitä on tarvittaessa sopeutettu, ne voidaan yhdistää toimivaksi sovellukseksi. Sovellusta ei välttämättä koota pelkästään komponentteja yhdistämällä, vaan osa sovelluksesta voidaan tehdä perinteisin menetelmin. Komponenttien yhdistäminen voi tapahtua useilla eri tavoilla. Käytettävä ohjelmointikieli ja komponenttimalli voivat rajoittaa yhdistämistavan valintaa, kuten sopeuttamistekniikoidenkin tapauksessa.

Sametinger on jakanut komponenttien yhdistämisen kahdeksaan eri tasoon, jotka ovat tekstuaalinen, proseduraalinen, modulaarinen, oliopohjainen, alijärjestelmätasoinen, oliomallitasoinen, komponenttialustatasoinen ja avoimen alustan kompositio [Sametinger 1997]. Jaottelun yksityiskohtia voidaan kritisoida, mutta pääpiirteittäin se kuvaa asiaa hyvin. Yhdistämistekniikkaa voidaan pitää sitä korkeatasoisempana, mitä löyhemmän siteen se jättää komponenttien välille.

6 KOMPONENTTIPOHJAINEN OHJELMISTOTUOTANTO

Tässä luvussa tarkastellaan aikaisemmissa luvuissa esitettyjen teknologioiden vaikutusta ohjelmistotuotantoprosessiin.

Komponenttipohjainen ohjelmistotuotanto (component-based development, CBD) voidaan määritellä tavaksi tuottaa ohjelmistoja, jossa ohjelmistoja tuotetaan sopeuttamalla ja yhdistämällä komponentteja. Komponentit voivat olla edellisten projektien tuotoksia, ne voidaan ostaa markkinoilta tai viime kädessä ne voidaan tehdä itse. Komponentin määritelmiä on esitetty luvussa 3.

6.1 Siirtyminen komponenttipohjaiseen ohjelmistotuotantoon

On huomattu, että uudelleenkäytettävät komponentit eivät synny itsestään projektien sivutuotteina, vaan niiden tuottamiseen tulee käyttää ylimääräisiä resursseja [Sametinger 1997]. Kun ohjelmistotuotannon tuloksia mitataan tyypillisesti projektikohtaisesti, muodostuu ongelmaksi suunnittelijoiden ja projektien vetäjien motivointi. Yksi mahdollisuus on palkita suunnittelijoita sen mukaan, miten paljon heidän tekemiään komponentteja käytetään myöhemmissä projekteissa. Jos organisaation liikeidea on myydä komponentteja, motivointi on helpompaa koska komponentit ovat itsessään tuotteita.

Mm. Tom Vayda suosittelee pilottiprojektin käynnistämistä siirtymävaiheessa. Pilottiprojektin tulisi olla kooltaan suhteellisen pieni mutta merkitykseltään suuri. Johdon tuen hankkiminen on tärkeää, kuten kaikissa muissakin prosessiin liittyvissä muutoksissa. Erityisen tärkeää on mitata

pilottiprojektin onnistumista. Odotuksia ei tulisi asettaa liian korkealle pettymysten välttämiseksi [Vayda 1999].

Komponenttipohjaiseen kehittämiseen siirtymisen katsotaan edellyttävän myös kehittäjien koulutusta, työkalujen ja menetelmien valintaa sekä komponenttivaraston perustamista [D'Souza & Wills 1998].

6.1.1 Menetelmätuki

Komponenttipohjaisen ohjelmistokehityksen tueksi on tarjolla ainakin kaksi tunnettua menetelmää, jotka ovat UML (Unified Modeling Language) sekä Catalysis.

Tom Vaydan mukaan UML on kuitenkin tarkoitettu enemmänkin olio- kuin komponenttipohjaiseen ohjelmistonkehitykseen. UML:n puutteista hän mainitsee mm. keskittymisen fyysisiin komponentteihin, rajoitetut mahdollisuudet semantiikan kuvaamiseen, puutteet komponenttien välisten suhteiden kuvaamisessa sekä rajoitetut mahdollisuudet loogisten komponenttien, sisään- ja ulostulevien rajapintojen, introspektion ja reflektion sekä useiden rajapintojen per komponentti kuvaamiseen. Catalysis korvaa hänen mukaansa osan puutteista tuomalla mukaan mm. OCL (Object Constraints Language)-kielen, jolla on mahdollista kuvata esi- ja jälkiehtoja sekä invariantteja ja sisääntulevat rajapinnat [Vayda 1999].

6.1.2 Työkalutuki

Tällä hetkellä markkinoilla olevat komponenttipohjaista ohjelmistotuotantoa tukevat työkalut ovat lähinnä tiettyyn komponenttimalliin liittyviä toteutusvaiheen välineitä. Komponenttipohjaista ohjelmistotuotantoprosessia laajemmalti tukevat työkalut (CASE-välineet, Computer-Aided Software

Engineering tools) ovat vielä keskeneräisiä, mutta jonkinlaista tukea tarjoavat mm. Rational Rose, SELECT ja Platinumin Enterprise Modeling Suite [D'Souza & Wills 1998]. Tom Vaydan mukaan tällaisen CASE-välineen tulisi tukea mm. mallintamista, ohjelmakoodin generointia malleista sekä integraatiota eri sovelluskehysten sekä ohjelmistonkehitysympäristöjen kanssa [Vayda 1999].

6.2 Ohjelmistotuotannon kahtiajako

Karlsson ym. jakavat komponenttipohjaisen ohjelmistotuotannon kahteen osaan: *kehittäminen uudelleenkäyttöä varten* ja *kehittäminen uudelleenkäytettäviä ohjelmistokomponentteja hyväksikäyttämällä* [Karlsson 1995]. Kehittäminen uudelleenkäyttöä varten tarkoittaa uudelleenkäytettävien ohjelmistokomponenttien kehittämistä. Kehittäminen uudelleenkäytettäviä ohjelmistokomponentteja hyväksikäyttämällä tarkoittaa ohjelmistojen rakentamista ohjelmistokomponenttien avulla. Sametinger kutsuu kehittämistä uudelleenkäyttöä varten *komponenttituotannoksi* (component engineering) ja kehittämistä uudelleenkäytettävillä komponenteilla *sovellustuotannoksi* (application engineering) [Sametinger 1997].

Mm. Richard Veryard näkee komponenttipohjaisen ohjelmistotuotannon johtavan ohjelmistotuotannon eri haarojen eriytymiseen, eli jatkossa yksittäisen suunnittelijan on entistä vaikeampi hallita ohjelmistotuotannon koko kenttä [Veryard 1997].

Boehms ja Scherlis esittävät Garlanin ja Perryn mukaan, että komponenttipohjainen ohjelmistotuotanto tulee synnyttämään useita uusia rooleja. Ohjelmistosuunnittelijat tulevat jakautumaan *komponenttien tuottajiin*, jotka kehittävät, testaavat ja sopeuttavat ohjelmistokomponentteja sekä *komponenttien käyttäjiin* (component assemblers), jotka etsivät ohjelmistokomponentteja ja yhdistävät niitä toimiviksi ohjelmistoiksi. Boehms ja

Scherlis näkevät yhtenä roolina myös *välittäjät* (brokers), jotka ylläpitävät komponenttikirjastoja [Garlan & Perry].

Jako komponenttien ja sovellusten tuottamiseen on erittäin merkittävä ohjelmistotuotannon tuottavuuden kannalta. Kirjallisuudessa on korostettu sitä, että ohjelmistokomponentit lisäävät tuottavuutta siksi, että voidaan käyttää hyväksi aikaisemmin tehtyä työtä. Vähintään yhtä suuri merkitys tulee olemaan sillä, että komponenttipohjainen ohjelmistotuotanto *tehostaa ohjelmistotuotannon työnjakoa.* Työnjaon kehittyminen eli erikoistuminen lisää kaiken taloudellisen toiminnan tehokkuutta. Tästä johtuen eri tuottajien luontaisista eroista, erikoistumisen mahdollistamasta harjaantumisesta sekä mahdollisista suurtuotannon eduista [Pekkarinen & Sutela 1989]. Komponenttien ja sovellusten kehittäminen vaativat erilaisia taitoja. Lisäksi eri tyyppisten komponenttien tuottaminen vaatii tietämystä eri sovellusalueista. Näin komponenttien tuottajat voivat erikoistua tuottamaan sitä, mitä he parhaiten osaavat. Sovellusten kokoajat voivat puolestaan keskittyä mm. ohjelmistojen arkkitehtuurillisiin kysymyksiin sekä käyttöliittymiin.

6.3 Komponenttituotanto

Komponenttituotanto on uudelleenkäytettävien ohjelmistokomponenttien kehittämistä. Uudelleenkäytettävän komponentin kehittäminen vaatii yleensä enemmän resursseja kuin vastaavan toiminnallisuuden kehittäminen ilman uudelleenkäytettävyyttä [Sametinger 1997][Haikala & Märijärvi 1997]. Uudelleenkäytettävät komponentit eivät yleisesti ottaen synny kehitettävien sovellusten oheistuotteina ilman erillistä panostusta, koska tällöin kehityksessä otetaan huomioon vain kehitettävän sovelluksen erityistarpeet [Sommerville 1998].

6.3.1 Milloin komponentti kannattaa tuottaa itse?

Ennen uuden komponentin tuottamista tulee miettiä, onko se välttämätöntä tuottaa itse. Uuden komponentin tekeminen itse on kannattavaa vain, jos:

$$T < E + O + S$$

missä

T = komponentin tuotantokustannukset
E = vastaavan komponentin etsinnästä ja potentiaalisten komponenttien arvioinnista aiheutuvat kustannukset[9]
O = vastaavan komponentin hinta
S = vastaavan komponentin mahdollisesta sopeuttamisesta aiheutuvat kustannukset

Jos vastaava komponentti löytyy oman organisaation sisältä aikaisempien projektien tuotoksena, O on 0. Mikäli vastaavaa komponenttia ei tarvitse sopeuttaa, S on 0.

Mikäli komponenttimarkkinat olisivat täydelliset, olisi komponenttien tekeminen itse yleensä kannattavaa vain kahdessa erikoistapauksessa:

1. Komponentti liittyy organisaation erityisosaamiseen, jolloin organisaation suhteellinen etu kyseisen komponentin tuottamisessa on niin suuri, ettei markkinoilta löydy vastaavaa riittävän laadukasta komponenttia kilpailukykyiseen hintaan.

2. Komponentti on niin pieni (esim. voidaan tehdä yhdessä tai muutamassa päivässä), että etsimiseen, arviointiin ja ostamiseen liittyvät kiinteät

---

[9] Nämä kustannukset eivät ole mitättömiä, koska ainakin toistaiseksi komponenttien formaali kuvaaminen rajapintojen avulla tuottaa vielä suuria ongelmia, joten komponenttien vertaaminen edellyttää usein niiden kokeilemista.

kustannukset muodostavat suuren osan E+S+O:sta. Tällöin T voi helposti jäädä pienemmäksi kuin E+S+O.

6.3.2 Uudelleenkäytettävien komponenttien tuottamisesta

Uudelleenkäytettävien komponenttien on oltava *tarpeeksi yleisiä*, jotta niitä voitaisiin käyttää useissa eri ympäristöissä ja tarpeeksi *hyvin dokumentoituja*, jotta useat eri käyttäjät osaisivat käyttää niitä. Ennen kaikkea komponenttien tulisi olla *itseriittoisia* tai ainakin ulkoisten riippuvuuksien pitäisi olla selkeästi dokumentoituna.

Komponenteista saadaan yleiskäyttöisempiä mm. lisäämällä komponenteille asetettuja toiminnallisia vaatimuksia tai mahdollistamalla komponentin konfigurointi käyttäjän asettamilla optioilla [Sametinger 1997]. Komponenttien ei tulisi sisältää viittauksia konkreettisiin ulkoisiin komponentteihin, vaan viittauksia rajapintoihin [Weck].

6.4 Sovellustuotanto

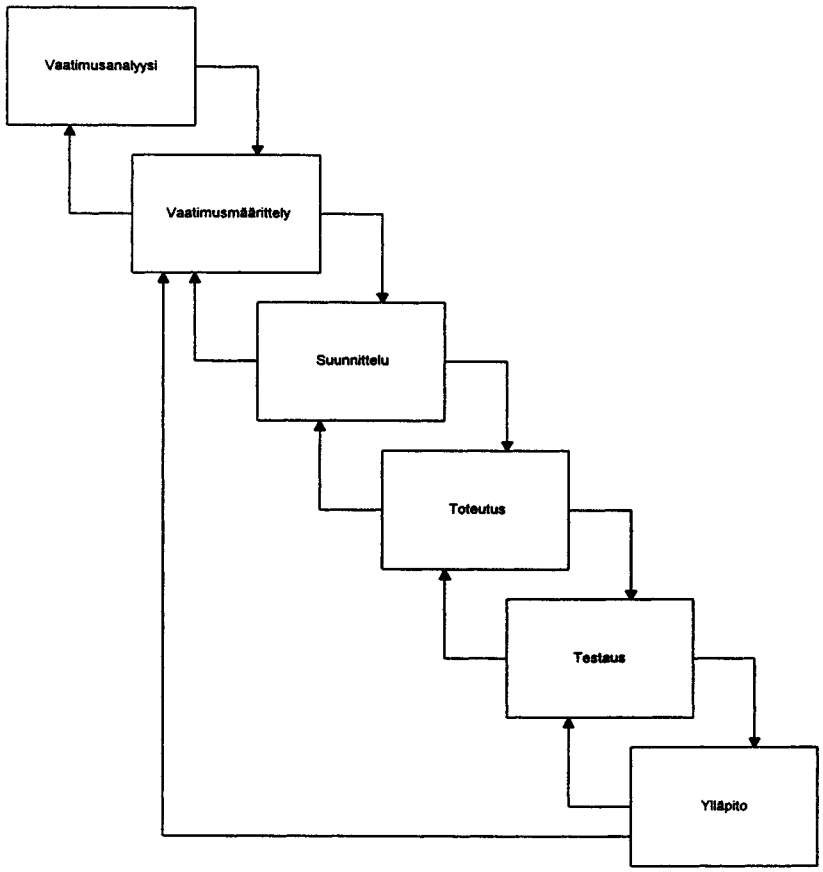
Tässä kappaleessa esitetään kaksi elinkaarimallia, joita voidaan soveltaa sovellustuotannossa. Mallien kehittämisessä lähdetään liikkeelle perinteisestä vesiputousmallista. Toinen malleista kuvaa kehittämistä uudelleenkäyttöä hyväksikäyttäen ja toinen uudelleenkäyttöpohjaista kehittämistä. Edellisessä vaatimusmäärittely ja suunnittelu määräävät millaisia komponentteja järjestelmän rakentamisessa käytetään. Jälkimmäisessä olemassaolevat ohjelmistokomponentit asettavat rajoituksia kehitettävälle ohjelmistolle vaatimusmäärittely- ja suunnitteluvaiheessa.

6.4.1 Perinteinen vesiputousmalli

Ohjelmistotuotannossa on jo pitkään pyritty soveltamaan *vaihejakomalleja*, jotka jakavat ohjelmistotuotantoprosessin selkeisiin, toisistaan erillisiin vaiheisiin. Perinteisen *vesiputousmallin* (waterfall model) mukaan ohjelmistotuotantoprosessin vaiheet ovat peräkkäisiä eli jokainen vaihe on suoritettava loppuun ennen seuraavaan vaiheeseen siirtymistä. Ohjelmisto on sitä parempi, mitä paremmin se vastaa ensimmäisessä vaiheessa luotua vaatimusmäärittelyä, joka ei muutu ohjelmistoprosessin aikana.

Vesiputousmallin etuna voidaan pitää sen yksinkertaisuutta projektinhallinnan kannalta [Sametinger 1997]. Haittana voidaan pitää epärealistisuutta, sillä varsin harvoin rakennettavalle ohjelmistolle esitettävät vaatimukset pysyvät samana koko projektin ajan. Vaikka olosuhteet pysyisivätkin samana, vaatimuksia ei ole välttämättä osattu ajatella oikein vaatimusmäärittelyä ensimmäistä kertaa laadittaessa.

Vesiputousmalliin katsotaan tyypillisesti kuuluvan seuraavat vaiheet: vaatimusanalyysi, vaatimusmäärittely, suunnittelu, toteutus, testaus ja ylläpito. Vesiputousmallin kulku on esitetty kuvassa 8.

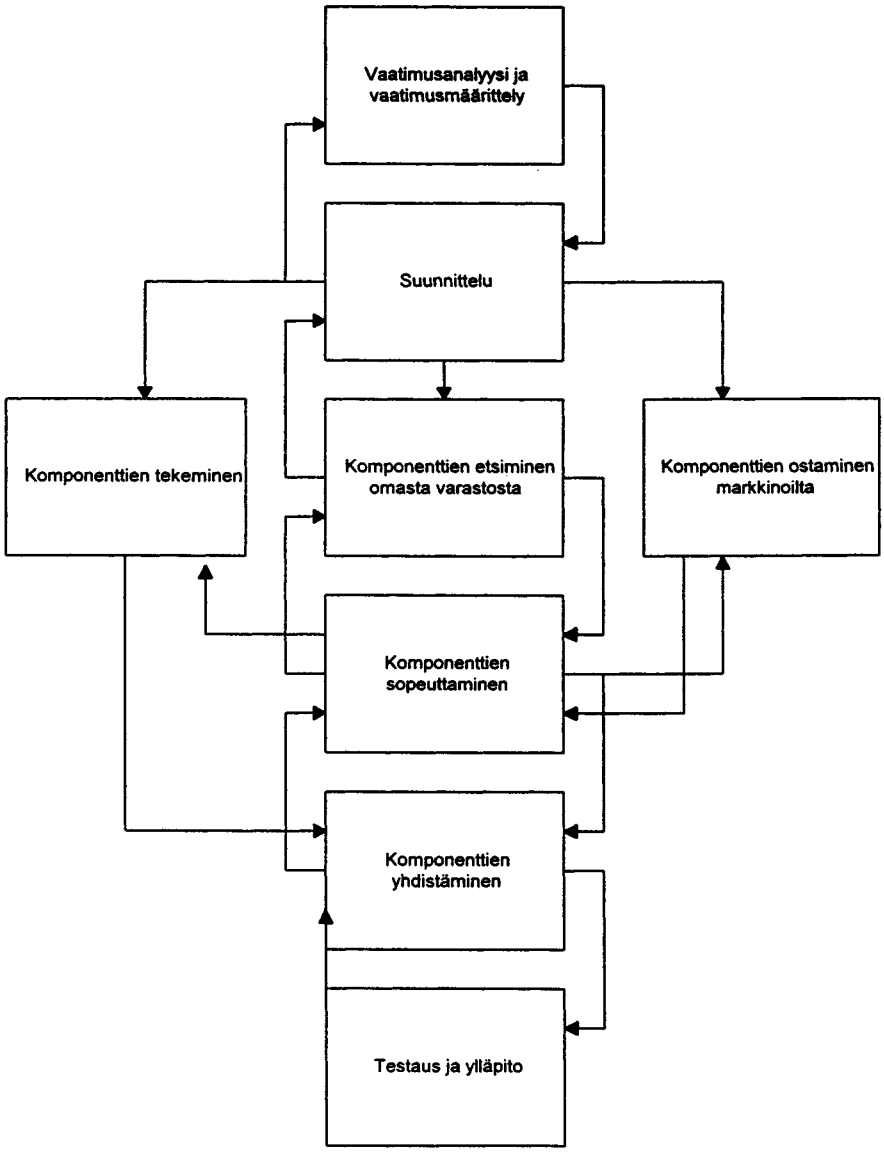


KUVA 8 Perinteinen vesiputousmalli.

6.4.2 Kehittäminen uudelleenkäyttöä hyväksikäyttäen

Kuvassa 9 on esitettynä vesiputousmalli muokattuna komponenttipohjaiselle ohjelmistotuotannolle sopivaksi. Vaatimusanalyysi ja vaatimusmäärittely ovat kuten perinteisessäkin vesiputousmallissa. Suunnitteluvaihe eroaa perinteisestä vesiputousmallista siten, että tässä vaiheessa pyritään määrittelemään, minkälaisista komponenteista järjestelmä koostuu. Tarkemmin ottaen pyritään määrittelemään minkälaiset rajapinnat näillä komponenteilla tulisi olla ja mitä suunnittelumalleja ja sovelluskehyksiä käytetään. Tämän jälkeen seuraa komponenttien hankintavaihe, jonka aikana pyritään hankkimaan komponentteja, jotka toteuttavat suunnitteluvaiheessa määritellyt rajapinnat. Osa komponenteista voidaan tehdä itse,

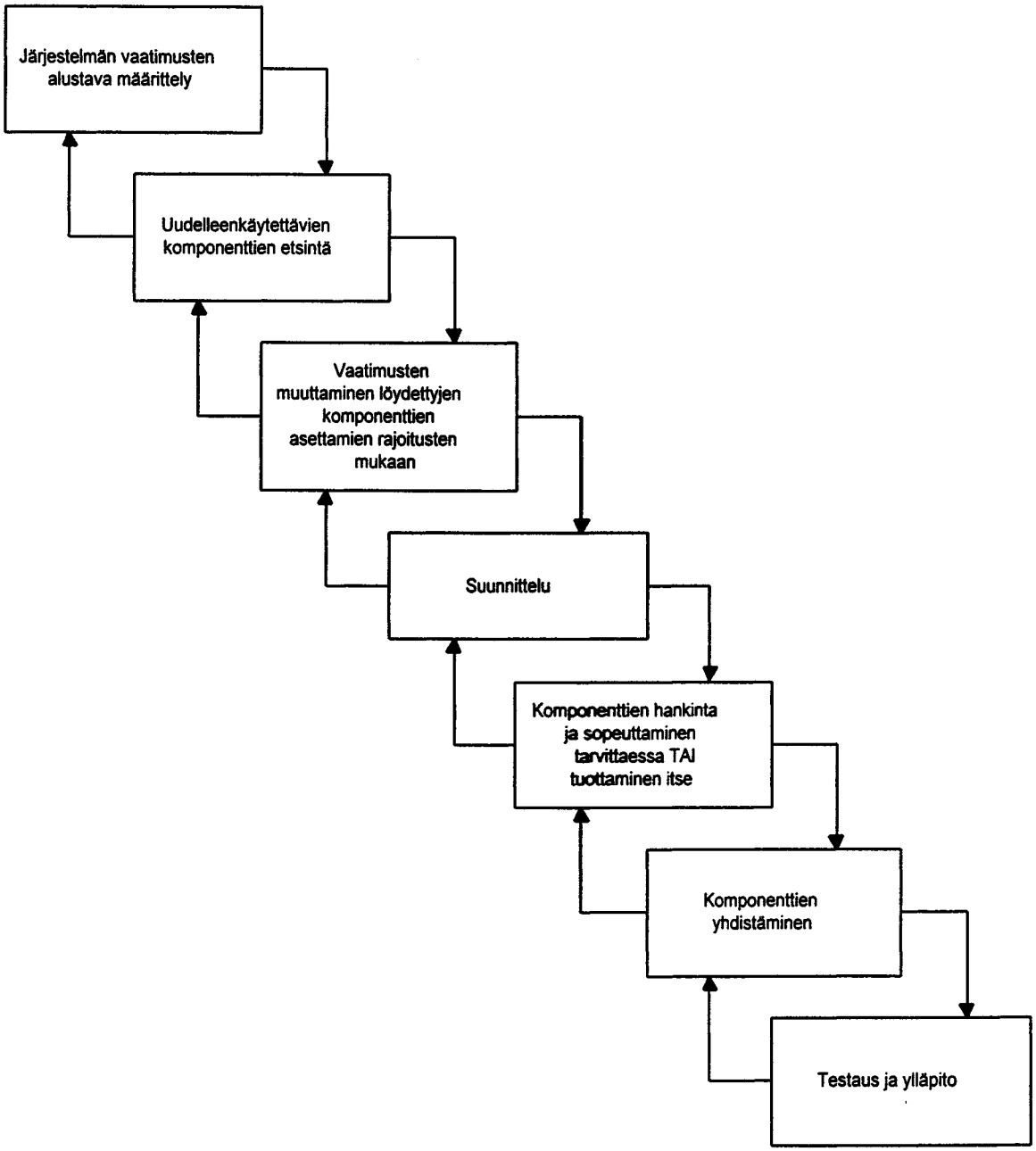
osa saattaa löytyä aikaisempien projektien tuotoksina organisaation ylläpitämästä komponenttivarastosta ja osa voidaan hankkia markkinoilta. Komponenttien hankinta markkinoilta edellyttää potentiaalisten komponenttien hankintaa arviointia varten ja arvioinnin suorittamista esimerkiksi tutustumalla dokumentaatioon tai rakentamalla prototyyppi. Seuraava vaihe on (tarvittaessa) komponenttien sopeutus. Sopeutuksen jälkeen komponentit voidaan yhdistää keskenään toimivaksi järjestelmäksi. Tämän jälkeen seuraavat järjestelmätestaus- ja ylläpitovaiheet ovat kuten perinteisessäkin vesiputousmallissa.



KUVA 9 Komponenttipohjainen vesiputousmalli, mukaeltu Sametingerin kuvasta [Sametinger 1997].

6.4.3 Uudelleenkäyttöpohjainen kehittäminen

Uudelleenkäyttöpohjaisessa kehittämisessä olemassaolevat ohjelmistokomponentit rajoittavat kehitettävän sovelluksen vaatimusmäärittelyä. Tämä johtaa luonnollisesti kompromisseihin järjestelmälle asetettavissa vaatimuksissa, mutta alentaa järjestelmän kehityskustannuksia ja lisää luotettavuutta [Sommerville 1998]. Kuvassa 10 on esitetty elinkaarimalli uudelleenkäyttöpohjaiselle kehittämiselle.



KUVA 10 Uudelleenkäyttöpohjaisen kehittämisen elinkaarimalli Sommervillen mukaan.

6.4.4 Spiraalimalli

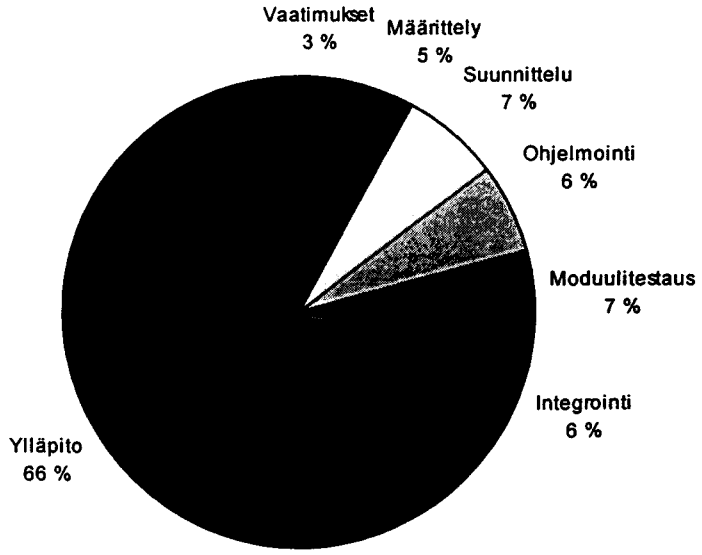
Vesiputousmallia realistisemman kuva ohjelmistoprojektin kulusta antaa alunperin Boehmin 1988 esittämä spiraalimalli. Malli on inkrementaalinen, eli projekti koostuu useista peräkkäisistä kierroksista. Kukin kierros koos-

tuu samoista vaiheista, joita Boehmin mallissa ovat prosessin suunnittelu, tekninen suunnittelu, toteutus ja testaus sekä tulosten arviointivaihe. Arviointivaiheessa arvioidaan myös projektiin jatkossa kohdistuvia riskejä.

Komponenttipohjaisessa ohjelmistotuotannossa kukin kierros koostuisi prosessin suunnittelusta, teknisestä suunnittelusta, uudelleenkäytettävien komponenttien hankinnasta, sopeuttamisesta, yhdistämisestä, testauksesta sekä tulosten arvioinnista.

6.5 Vaikutus elinkaarikustannuksiin

Kuvassa 11 on esitetty perinteisten ohjelmistoprojektien elinkaarikustannusten jakautuminen. Kuten kuvasta näkyy, noin kaksi kolmasosaa elinkaarikustannuksista aiheutuu ohjelmistojen ylläpidosta.



KUVA 11 Elinkaarikustannusten jakautuminen ohjelmistoprojekteissa [Schach 1990].

Olemassaolevien komponenttien käyttö vähentää komponenttien suunnitteluun, ohjelmointiin sekä moduulitestaukseen kuluvaa aikaa. Nämä säästöt eivät kuitenkaan ole kokonaisuuden kannalta ratkaisevan suuria. Lisäksi kaikkien näiden osa-alueiden kustannuksia lisää hieman mahdollinen komponenttien sopeuttaminen.

Kuvan 11 perusteella on selvää, että suurimmat potentiaaliset säästöt voidaan saavuttaa, mikäli tarve ohjelmistojen ylläpitoon vähenee. Komponenteista koostuvaa ohjelmistoa on helpompi ylläpitää mm. siksi, että komponentteja on testattu jo aikaisemmissa ympäristöissä [Veryard 1997] [Sametinger 1997]. Lisäksi mikäli komponentit on ostettu markkinoilta, huolehtii komponentin toimittaja tämä komponenttien ylläpidosta: virheiden korjaamisesta ja uusien piirteiden lisäämisestä.

Ylläpidon tarve ei kuitenkaan katoa kokonaan omasta organisaatiosta kahdesta syystä. Ensinnäkin, vaikka komponentit sinänsä toimisivatkin oikein, se ei takaa että niistä yhdistetty ohjelmisto toimisi kokonaisuudessaan oikein. Toiseksi vaikka markkinoilta ostettujen komponenttien toimittaja huolehtii niiden ylläpidosta, aiheuttaa tämä töitä myös ostajalle, jos komponentteja on itse sopeutettu. Käytetyllä sopeuttamistekniikalla on merkitystä syntyvien kustannusten määrään. Leikkaa-liimaa-tekniikalla tehty sopeuttaminen aiheuttaa eniten ylläpitokustannuksia, koska uuden version tullessa sopeuttaminen täytyy käytännössä tehdä kokonaan uudestaan. Myös periyttäminen saattaa tuottaa ongelmia (katso alakohta 5.2.5).

6.6 Yhteenveto komponenttipohjaisesta ohjelmistotuotannosta

Komponenttipohjaiseen ohjelmistotuotantoon siirtyminen edellyttää organisaatiolta johdon tukea, henkilökunnan koulutusta, menetelmien ja työkalujen valintaa ja komponenttien varastoinnin aloittamista. Lisäksi erityistä huomiota on kiinnitettävä suunnittelijoiden ja ryhmänvetäjien motivoimi-

seksi, jotta heidän kannattaisi käyttää resursseja uudelleenkäytettävien komponenttien luomiseen.

Suunnittelijat tulevat jatkossa jakautumaan komponenttien tuottajiin ja käyttäjiin, jotka yhdistävät komponenteista sovelluksia. Toisaalta sovelluksia ei välttämättä tulevaisuudessakaan koota pelkästään komponenteista, joten jako ei välttämättä ole täydellinen. Joka tapauksessa lisääntyvä erikoistuminen tulee johtamaan ohjelmistotuotannon tuottavuuden kasvuun.

Ohjelmistokomponentteja kannattaa tehdä itse vain silloin, kun komponentti liittyy organisaation erityisosaamiseen tai silloin, kun sen toteuttaminen on triviaalia. Jos komponentteja tehdään itse, tulee komponenttien yleiskäyttöisyyteen, itseriittoisuuteen ja dokumentaatioon kiinnittää erityistä huomiota.

Komponenttipohjainen ohjelmistotuotanto tuo mukaan ohjelmistotuotannon vaihejakomalleihin uusia vaiheita, joita ovat komponenttien hankinta ja mahdollisesti niiden sopeuttaminen.

Ohjelmistojen elinkaarikustannusten kannalta komponenttien käyttö tuo pieniä säästöjä suunnittelu-, toteutus- ja testausvaiheissa, mutta merkittävimmät säästöt voidaan saavuttaa ylläpitovaiheessa. Toisaalta tiettyjen sopeuttamistekniikoiden (leikkaa-liimaa ja toteutusperintä) laaja käyttö voivat vähentää syntyviä säästöjä.

7 YHTEENVETO

Tämän tutkielman tavoitteena on ollut selvittää, mitä ohjelmistokomponentit ovat, miten ne eroavat aikaisemmista ratkaisuista, miten ohjelmistoja tehdään komponenteista, ja millaisia vaikutuksia komponenttipohjaisella ohjelmistotuotannolla on ohjelmistotuotannon vaihejakoon.

Tutkielman luvussa 3 on esitetty kirjallisuudesta löytyneitä määritelmiä ohjelmistokomponenteille. Määritelmissä on yhteisiä piirteitä. Valtaosa ohjelmistokomponenteille esitetyistä määritelmistä vaatii komponenteilta selkeästi dokumentoituja rajapintoja, itseriittoisuutta sekä yhdistettävyyttä muihin komponentteihin. Määritelmät kuitenkin eroavat toisistaan siinä, miten abstrakteja käsitteitä ne laskevat komponenteiksi. Määritelmät on jaettu luvussa 3 neljään luokkaan, joista tiukimmat määritelmät hyväksyvät vain binäärimuotoiset komponentit, kun löyhemmät määritelmät hyväksyvät myös lähdekoodin, dokumentaation ja jopa abstraktit ideat - kaikki ohjelmistotuotannossa syntyvät, uudelleenkäytettävissä olevat tuotokset. Luvussa 3 on myös esitetty, miten komponentit eroavat olioluokista ja mikä on niiden suhde sovelluskehyksiin, suunnittelumalleihin ja arkkitehtuureihin.

Luvussa 4 on esitetty komponentteihin ainakin tiukempien määritelmien mukaan oleellisesti liittyvä komponenttimallin käsite, sekä kuvattu kolmen tällä hetkellä tärkeimmän komponenttimallin perusteita. Lisäksi luvussa 4 olevassa taulukossa esitetään joitain komponenttimallien keskeisiä eroja.

Luvussa 5 on kuvattu komponenttipohjaisen ohjelmistotuotannon vaiheita: komponenttien hankinta, sopeuttaminen ja yhdistäminen. Hankinta ja yhdistäminen kuvataan lähinnä kirjallisuuden perusteella. Hankinnan yhteydessä käsitellään mm. komponenttien rajapinnoille asetettavia vaatimuksia

ja yhdistämisen yhteydessä yhdistämisen tasoja ja visuaalista yhdistämistä. Luku kuitenkin painottuu komponenttien sopeuttamisen käsittelyyn. Esitettyjä sopeuttamistekniikoita verrataan seuraavien kriteerien valossa: onko se läpinäkyvä, edellyttääkö se komponenttien toteutuksen paljastamista (musta laatikko vs. valkoinen laatikko) ja millaisia ongelmia komponenttien sopeuttaminen aiheuttaa, kun komponentista julkaistaan uusi versio.

Tutkielman luvussa 6 käsitellään vielä komponenttipohjaisen ohjelmistotuotannon vaihejakoa kokonaisuutena. Komponenttipohjaisuus tuo ohjelmistotuotannon vaihejakomalleihin uusia vaiheita, joita ovat komponenttien hankinta, sopeuttaminen ja yhdistäminen. Suurimmat hyödyt komponenttipohjaisesta ohjelmistotuotannosta saadaan, jos komponentteja voidaan käyttää sellaisenaan, koska tällöin ylläpidossa saavutettavat säästöt ovat suurimmillaan. Tämä on kuitenkin käytännössä harvinaista, joten komponentteja on yleensä sopeutettava jotain sopeuttamistekniikkaa käyttäen. Eniten ongelmia ohjelmistojen ylläpidon kannalta aiheuttaa komponenttien lähdekoodin suora editointi (leikkaa-liimaa-tekniikka). Periytymisen käyttö voi johtaa särkyvän yliluokan ongelmaan.

Tutkielman keskeiset kontribuutiot ovat siis: 1) komponenttien määrittelyyn liittyvä kirjallisuustutkimus ja luokittelu, 2) esitys komponenttien suhteesta komponenttimalleihin, olioihin, suunnittelumalleihin, sovelluskehyksiin ja arkkitehtuureihin, 3) sopeuttamistekniikoiden esitys ja vertailu ja 4) esitys edellisten vaikutuksesta komponenttipohjaiseen ohjelmistotuotantoprosessiin.

LÄHTEET

Banker R.D., Kauffmann R.J. ja Zweig D.: Repository Evaluation Of Software Reuse, IEEE Transactions on Software Engineering, 19, April 1993

Booch Grady: Software Components with Ada: Structures, Tools and Subsystems, Benjamin/Cummings Publishing Company Inc., 1987

Bosch Jan: Superimposition - A Component Adaptation Technique, Information and Software Technology, 1999

Braun Christine: NATO Standard For The Development Of Reusable Software Component Library, 1994

Brooks Frederick Jr: No Silver Bullet - Essence and Accidents of Software Engineering, IEEE Computer, Vol. 20, No. 4, 1987

Bronsard Francois, Bryan Douglas, Kozaczynski W., Liongosari Edy, Ning Jim, Olafsson Asgeir ja Wetterstrand John: Toward Software Plug-and-Play, Software Engineering Notes, Vol. 22, No. 3, Andersen Consulting, 1997

Brown Alan ja Short Keith: On Components and Objects: The Foundations of Component-Based Development, Texas Instruments Software, 1997

Buchi Martin ja Weck Wolfgang: A Plea for Grey-Box Components, Turku Centre for Computer Science, 1997

Budd Timothy: An Introduction To Object-Oriented Programming, 2. painos, Addison-Wesley, 1997

Chappell David: Component Software Enters The Mainstream, The Next Wave, Rational Software Corporation, April 1997

Ciupke Oliver ja Schmidt Rainer: Components As Context-Independent Units Of Software, Forschungszentrum Informatik, 1996

D'Souza D.F. ja Wills A.: Objects, Components And Frameworks With UML: The Catalysis Approach, Addison-Wesley, 1998

Franz Michael: On The Architecture Of Software Component Systems, Systems Implementation 2000, Chapman & Hall, 1998

Gamma Erich, Helm Richard, Johnson Ralph ja Vlissides John: Design Patterns - Elements Of Reusable Object-Oriented Software, Addison-Wesley, 1995

Garlan David ja Perry Dewayne: Introduction To The Special Issue On Software Architecture (DRAFT)

Haikala Ilkka ja Märijärvi Jukka: Ohjelmistotuotanto, 4.painos, Suomen ATK-kustannus, 1997

Hannula Esko: New Trends Of Software Technology, Merito Forum Oy, 1998

Harmon Paul: Components, Component Development Strategies July 1998, Cutter Information Corporation,1998

Hoffmann Holger: Componentware - Integration Of Software Components In Distributed Computing Environments, Master's Thesis, 1997

Hölzle Urs: Integrating Independently-Developed Components In Object-Oriented Languages, ECOOP'93 Proceedings, Springer-Verlag Lecture Notes on Computer Science, 1993

Jaaksi Ari: From Objects To Components, Merito Forum Oy, 1998

Jacobson Ivar: Architecture Is About Everything - But Not Everything Is Architecture, Component Strategies, March 1999

Kain Bradford: Software Components As Application Building Blocks, QUOIN Inc., 1998

Karlsson Even-André, editor: Software Reuse: A Holistic Approach, John Wiley & Sons, 1995

Keller Ralph ja Hölzle Urs: Supporting The Integration And Evolution Of Components Through Binary Component Adaptation, University of California, Department of CS, 1997

Koskimies Kai: Framework technology, Merito Forum Oy, 1998

Krajnc Marco: What Is The Difference Between COP And OOP?, Oda Team, 1997

Luckham David, Vera Javes and Meldal Sigurd: Three Concepts of System Architecture, 1995

Meyer Bertrand: Object-Oriented Software Construction, 2. painos, Prentice Hall, 1997

Mikhajlov Leonid ja Sekerinski Emil: The Fragile Base Class Problem And Its Solution, Turku Centre for Computer Science, 1997

Orfali Robert, Harkey Dan ja Edwards Jeri: The Essential Distributed Objects Survival Guide, John Wiley & Sons, 1996

Pekkarinen Jukka ja Sutela Pekka: Kansantaloustiede I, 5.painos, WSOY, 1989

Pree Wolfgang: Component-Based Software Development - A New Paradigm In Software Engineering?, Software - Concepts and Tools, 1997

Sakkinen Markku: Disciplined Inheritance, ECOOP'89 Proceedings, Cambridge University Press, 1989

Salonen Petri: Quality Of Objects And Components, Merito Forum Oy, 1998

Sametinger Johannes: Software Engineering With Reusable Components, Springer-Verlag, 1997

Schach S.: Software Engineering, Aksen Associates Incorporated Publishers, 1990

Sommerville Ian: Software Engineering, Addison-Wesley, 1998

Stata R. ja Guttag J.: Modular Reasoning In The Presence Of Subclassing, OOPSLA'95 Proceedings, ACM Press ja Addison-Wesley, 1995

Szyperski Clemens: Component-Oriented Programming - A Refined Variation on Object-Oriented Programming, The Oberon tribune Vol. 1, No. 2, December 1995

Szyperski Clemens: Component Software - Beyond Object-Oriented Programming, Addison-Wesley, 1998

Tracz Will: Software Reuse: Motivators And Inhibitors, Tutorial: Software Reuse: Emerging Technology, 1987

Udell Jon: Componentware, Byte Magazine, May 1994

Vanhelsuwé Laurence: Mastering JavaBeans, Sybex, 1997

Vayda Tom: Organizing For Components, Component Strategies, February 1999

Veryard Richard: Component-Based Development, Veryard projects, 1997

Weck Wolfgang: Inheritance Using Contracts & Object Composition, Turku Center for Computer Science, 1997