Marko Hollanti

UUDELLEENKÄYTETTÄVIEN OHJELMISTOKOMPONENTTIEN RAJAPINTOJEN DOKUMENTOINTI

Tietojärjestelmätieteen
pro gradu –tutkielma
29.8.2000

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Hollanti, Marko Tapani
Uudelleenkäytettävien ohjelmistokomponenttien rajapintojen dokumentointi / Marko Hollanti.
Jyväskylä: Jyväskylän yliopisto, 2000.
114 s.
Tutkielma

Ohjelmistokomponenttien uudelleenkäytön ja ohjelmistokomponenttimarkkinoiden kehittymisen yhtenä edellytyksenä voidaan pitää ohjelmistokomponenttien rajapintojen riittävää dokumentointia. Tutkimuksessa rajaudutaan tarkastelemaan suorituskelpoisten ohjelmistokomponenttien sopimuksenmukaisten rajapintojen dokumentointia. Tutkimuksen kantavana ajatuksena on se, että rajapintojen kuvausten voidaan ajatella olevan ohjelmistokomponentin ja tämän asiakkaiden välisiä sopimuksia.

Tutkimus on luonteeltaan käsitteellisteoreettinen. Siinä esitetään kuvaus ohjelmistokomponenttien uudelleenkäyttöön perustuvasta ohjelmistonkehitysprosessista, ja selvitetään mitä sopimuksenmukaisissa rajapinnoissa on syytä kuvata ja miksi. Tämän lisäksi kuvataan ohjelmistojen dokumentointia yleisesti, sekä esitetään, mitkä ovat hyvän ohjelmistodokumentaation kriteerit.

Tutkimuksen keskeinen tuotos, ohjelmistokomponentin rajapintojen dokumentointimalli, perustuu edellä kuvattuihin asioihin. Lopuksi dokumentointimallia sovelletaan esimerkkinä käytettävän JavaBeans-ohjelmistokomponentin toiminnallisuuden dokumentoimiseksi.

AVAINSANAT: ohjelmistokomponentti, rajapinta, uudelleenkäyttö

1 JOHDANTO 1

2 OHJELMISTOKOMPONENTTEIHIN PERUSTUVA OHJELMISTOTUOTANTO 3

2.1 Keskeisten käsitteiden määrittely 3
2.1.1 Ohjelmistokomponentti 3
2.1.2 Rajapinta 6
2.1.3 Uudelleenkäyttö 8
2.2 Ohjelmistojen 'rakentaminen' ohjelmistokomponenteista 9
2.2.1 Ohjelmistokomponenttien uudelleenkäytön tehtävät 11
2.2.2 Uudelleenkäyttöön perustuva ohjelmistonkehitysprosessi 13
2.3 Yhteenveto 17

3 SOPIMUKSENMUKAISET RAJAPINNAT 19

3.1 Mitä sopimuksenmukaisilla rajapinnoilla tarkoitetaan? 19
3.1.1 Tarvittavan tiedon määrä 20
3.1.2 Korvattavuus 21
3.2 Mitä sopimuksenmukaiset rajapinnat sisältävät? 22
3.2.1 Ohjelmistokomponentin toiminta 22
3.2.2 Ohjelmistokomponentin laatu 29
3.3 Yhteenveto 32

4 OHJELMISTOJEN DOKUMENTOINTI 34

4.1 Ohjelmistodokumentaation kategoriat 34
4.1.1 Käyttäjille suunnattu dokumentaatio 35
4.1.2 Järjestelmädokumentaatio 37
4.1.3 Prosessidokumentaatio 39
4.1.4 Uudelleenkäytön dokumentaatio 40
4.2 Hyvän ohjelmistodokumentaation kriteerit 47
4.2.1 Oikeellisuus 49
4.2.2 Kattavuus 50
4.2.3 Käytettävyys 51
4.2.4 Laajennettavuus 52
4.3 Yhteenveto 53

5 RAJAPINTOJEN DOKUMENTOINTIMALLI 56

5.1 Dokumentointimallin rakenne 57
5.1.1 Yleinen osa 58
5.1.2 Yksittäisen palvelun kuvaava osa 58
5.2 Dokumentointimallin sisältö 60
5.2.1 Ohjelmistokomponentin toiminnan kuvaus 62
5.2.2 Ohjelmistokomponentin laadun kuvaus 71
5.3 Yhteenveto 87

6 DOKUMENTOINTIMALLIN SOVELTAMINEN 90
6.1 JavaBeans-määritys 90
6.1.1 Ohjelmistokomponenttimalli ja ohjelmistokomponentti 90
6.1.2 Esimerkkinä käytettävä JavaBeans-ohjelmistokomponentti 92
6.2 Dokumentointimallin soveltaminen ja arviointi 95
6.2.1 Soveltaminen 95
6.2.2 Arviointi 99
6.3 Yhteenveto 101
7 YHTEENVETO 103
LÄHTEET 105
LIITTEET 115

1 JOHDANTO

Idea ohjelmistokomponentteihin perustuvasta ohjelmistoteollisuudesta esitettiin jo 1960-luvun lopulla (McIlroy, 1969), kun haettin vastausta ohjelmistokriisiin. Vielä tänäkään päivänä ei voida kuitenkaan väittää, että ohjelmistokomponentteihin perustuva ohjelmistoteollisuus olisi todellisuutta, ainakaan laajassa mittakaavassa.

Yksi tekijä, jolla on suuri vaikutus tulevaisuuden ohjelmistokomponenttimarkkinoiden kehitykseen, on ohjelmistokomponenttien dokumentointi. Ilman riittävää dokumentaatiota ohjelmistokomponenttien löytäminen ja niiden oikeaoppinen uudelleenkäyttö on vaikeaa. Büchi ja Weck (1997) ovat todenneet, että ohjelmistokomponentit helpottavat ohjelmistotuotantoa vain, mikäli niiden käyttöönotto on helpompaa kuin se, että ohjelmistokomponentit tehtäisiin itse tai että niiden sisäinen toiminnallisuus ymmärrettäisiin täydellisesti. Ohjelmistokomponenttien dokumentoinnin yksi tärkeä tehtävä onkin mahdollistaa olemassaolevien ohjelmistokomponenttien uudelleenkäyttö ilman, että käyttäjän tarvitsee tietää miten ohjelmistokomponentti on toteutettu.

Ohjelmistokomponentin rajapinnat kuvaavat ohjelmistokomponentin tarjoamat palvelut, mutta eivät niiden toteutusta. Usein rajapinnat eivät kuitenkaan tarjoa riittävästi tietoa, jotta ohjelmistokomponentin käyttö olisi helppoa. Ohjelmistokomponenttien rajapintojen dokumentaation tuleekin tarjota riittävästi informaatiota, jotta ohjelmistokomponentin käyttö onnistuu, mutta toisaalta se ei saa paljastaa toteutusteknisiä yksityiskohtia, jottei ohjelmistokomponentin korvaaminen toisella, mahdollisesti saman ohjelmistokomponentin uudella versiolla, aiheuta ongelmia (ks. Büchi ja Weck, 1997).

Tutkimuksen kantavana ajatuksena on se, että rajapintojen kuvausten voidaan ajatella olevan ohjelmistokomponentin ja tämän asiakkaiden välisiä sopimuksia. Tämä on myös ohjelmistokomponentin rajapintojen dokumentointimallin kehittämisen lähtökohta. Tutkimuksen tavoitteena on selvittää, kuinka uudelleenkäytettävien ohjelmistokomponenttien rajapinnat voidaan dokumentoida. Kuvattavien asioiden selvittämisen lisäksi tutkimuksessa tarkastellaan ohjelmistokomponentin uudelleenkäyttöprosessia, yleisesti ohjelmistojen dokumentointia, sekä hyvän ohjelmistodokumentaation kriteereitä. Näiden merkitystä pohditaan mahdollisuuksien mukaan ohjelmistokomponentin rajapintojen dokumentointimallin sekä tähän perustuvan dokumentaation kannalta. Dokumentointimallia sovelletaan esimerkkinä käytettävän JavaBeans-ohjelmistokomponentin rajapintojen dokumentoimiseksi.

Tutkielman sisältö on seuraava. Kappaleessa 2 määritellään tutkimuksen kannalta keskeiset käsitteet, jotka ovat ohjelmistokomponentti, rajapinta ja uudelleenkäyttö. Tämän lisäksi kuvataan ohjelmistokomponenttien uudelleenkäyttöprosessin sisältämät tehtävät ja esitetään, kuinka ne vaikuttavat perinteiseen ohjelmistonkehitysprosessiin. Kappaleessa 3 tarkastellaan ohjelmistokomponentin sopimuksenmukaisia rajapintoja, erityisesti mitä niillä tarkoitetaan, sekä mitä ohjelmistokomponentin piirteitä niiden avulla tulee kuvata ja miksi. Kappale 4 käsittelee yleisesti ohjelmistojen dokumentointia. Lisäksi tässä kappaleessa esitetään hyvän ohjelmistodokumentaation kriteerit. Kappale 5 perustuu aikaisemmissa kappaleissa käsiteltyjen asioiden varaan, ja siinä esitetään malli ohjelmistokomponentin rajapintojen dokumentoimiseksi. Kappaleessa 6 dokumentointimallia sovelletaan esimerkkinä käytettävän JavaBeans-ohjelmistokomponentin toiminnallisuuden dokumentoimiseksi. Kappalessa 7 esitetään tutkimuksen yhteenveto.

2 OHJELMISTOKOMPONENTTEIHIN PERUSTUVA OHJELMISTOTUOTANTO

Tässä kappaleessa tarkastellaan ohjelmistokomponentteihin perustuvaa ohjelmistotuotantoa. Aluksi määritellään mitä ohjelmistokomponentilla tarkoitetaan tässä tutkimuksessa, ja mitä muita määritelmiä sille on kirjallisuudessa esitetty. Myös muut tutkielman kannalta keskeiset käsitteet ja niiden väliset suhteet määritellään. Lopuksi kerrotaan, kuinka ohjelmistoja 'rakennetaan' kokoamalla niitä ohjelmistokomponenteista, erityisesti mitä tehtäviä ohjelmistokomponenttien uudelleenkäyttöön liittyy ja kuinka ne vaikuttavat perinteiseen ohjelmistonkehitysprosessiin.

2.1 Keskeisten käsitteiden määrittely

Tämän tutkimuksen kannalta keskeiset käsitteet ovat ohjelmistokomponentti, rajapinta ja uudelleenkäyttö. Käsitteiden määrittelyn lisäksi kuvataan, kuinka ne liittyvät toisiinsa.

2.1.1 Ohjelmistokomponentti

Ohjelmistokomponenteille on esitetty kirjallisuudessa useita erilaisia määritelmiä. Tässä tutustutaan niistä muutamiin ja pohditaan hiukan, miten eri määritelmät eroavat toisistaan. Ensimmäiseksi esitetään Szyperskin (1997) määritelmä, jota käytetään myös tässä tutkimuksessa.

Ohjelmistokomponentti (software component) on itsenäinen, tietyn toiminnan omaava yksikkö, jolle on selkeästi määritelty sopimuksenmukaiset rajapinnat sekä riippuvuudet ympäristöönsä. Ohjelmistokomponentteja voidaan levittää

toisistaan riippumatta ja kolmannet osapuolet voivat käyttää niitä ohjelmistojensa rakentamiseen. (Szyperski, 1997, s. 34)

Ohjelmistokomponenteilla ei siis ole muita riippuvuuksia ympäristöönsä kuin ne, jotka sille on eksplisiittisesti määritelty. Sametinger (1997) käyttää termiä itseriittoinen (self-contained) kuvatessaan ohjelmistokomponentteja. Szyperski (1997) toteaa, että hänen määritelmänsä seurauksena ohjelmistokomponenttien voidaan ymmärtää olevan binaarisia yksiköitä. Jaaksin (1998) mukaan ohjelmistokomponentit ovat tuotteita, toisin kuin esimerkiksi C++-luokat. Tässä tutkimuksessa ohjelmistokomponentilla tarkoitetaan jatkossa Szyperskin (1997) määritelmän mukaisia, suorituskelpoisia ohjelmistokomponentteja.

Jacobsonin (1992, s. 289) mukaan ohjelmistokomponentit ovat jo olemassaolevia, toteutettuja yksiköitä, joita käytetään ohjelmointikielen rakenteiden ohella ohjelmoinnin aikana. Szyperski (1997) mainitsee makrot ja mallit (templates) Jacobsonin (1992) määritelmän mukaisina ohjelmistokomponentteina, ja jatkaa, että Jacobsonin (1992) määritelmä on kattavampi kuin hänen määritelmänsä. Jacobsonin (1992) määritelmä rajoittuu kuitenkin ohjelmointivaiheen aikana käytettäviin ei-binaarisiin yksiköihin, joten se eroaa huomattavasti Szyperskin (1997) esittämästä ja tässä tutkimuksessa käytettävästä ohjelmistokomponentin määritelmästä.

Sametinger (1997, s. 68) määrittelee ohjelmistokomponentin seuraavasti. Uudelleenkäytettävät ohjelmistokomponentit ovat itseriittoisia, selkeästi tunnistettavia (ohjelmiston) osia, jotka kuvaavat ja/tai suorittavat tiettyjä toimintoja, ja joilla on selkeästi määritellyt rajapinnat, riittävä dokumentaatio ja määritelty uudelleenkäytön status. Määritelty uudelleenkäytön status tarkoittaa mm. testauksen ja ylläpidon suhteen määriteltyä ohjelmistokomponentin laatuarvoa. Sametinger (1997) toteaa, että hänen määritelmänsä mukaiset ohjelmistokomponentit voivat olla hyvinkin erityyppisiä, kuten esimerkiksi dokumentteja tai lähdekoodia. Vaikka Sametingerin (1997) määritelmä onkin hyvin laaja, Szyperskin (1997)

mukaan siinä on myös monia yhteneväisyyksiä hänen määritelmäänsä. Itseriittoisuuden ja tunnistettavuuden voidaan katsoa epäsuorasti tarkoittavan sitä, että ohjelmistokomponentteja voidaan levittää toisistaan riippumatta (independent deployability). Myös selkeästi määritellyt rajapinnat ovat mukana molemmissa määritelmissä.

Orfalin, Harkeyn ja Edwardsin (1995, s. 34) määritelmä on sopusoinnussa tässä tutkielmassa käytetyn määritelmän kanssa. Heidän mukaansa ohjelmistokomponentit ovat riippumattomia, tunnistettavia ja markkinoinnin kohteena olevia kokonaisuuksia, joita käytetään uudelleen enemmän tai vähemmän ennustamattomissa tilanteissa ja odottamattomissa yhteyksissä. Tämän määritelmän perusteella ohjelmistokomponenttien voidaan ymmärtää olevan itsenäisiä tuotteita.

Usein kirjallisuudessa puhutaan ohjelmistokomponenteista monessa eri yhteydessä ilman, että niitä aina edes määritellään kovinkaan tarkasti. Yleisesti ohjelmistokomponenteilla voidaan kirjallisuudessa tarkoittaa ohjelmiston osia, olivatpa ne sitten luokkia, luokkakirjastoja, suunnittelumalleja (design patterns) tai muita ohjelmistoihin liittyviä artefakteja. Esimerkiksi Pressmanin (1994, s. 13) mukaan 1990-luvun uudelleenkäytettävät ohjelmistokomponentit sisältävät sekä tiedon että tiedon käsittelyyn tarvittavat operaatiot yhdessä ja samassa paketissa (kutsutaan usein *luokaksi* tai *olioksi*), mahdollistaen sen, että ohjelmiston kehittäjä voi luoda uusia sovelluksia uudelleenkäytettävistä osista. Pressmanin (1994) määritelmä on aika yleisluonteinen ja soveltuu kuvaamaan myös suorituskelpoisia ohjelmistokomponentteja, mutta Pressman (1994) viittaa määritelmällään kuitenkin luokkiin ja olioihin.

Kuten Pressmanin (1994) ohjelmistokomponentin määritelmästä voidaan havaita, on olioilla ja ohjelmistokomponenteilla monia yhteisiä tekijöitä. Molemmat esimerkiksi tarjoavat joukon palveluita, joita on mahdollista käyttää rajapintojen kautta, vaikkakin rajapinnat saattavat erota tekniseltä toteutukseltaan

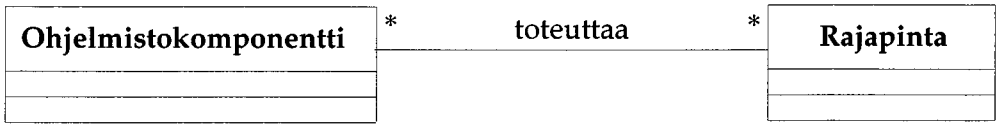
toisistaan. Toinen yhtäläisyys, joka liittyy rajapintoihin, on se, että sekä oliot että ohjelmistokomponentit kapseloivat (encapsulation) sisäisen toteutuksensa. Tämä liittyy tiedon piilottamisen (information hiding) periaatteeseen, joka on myös yhteinen piirre. Lisäksi oliojärjestelmien suunnittelu perustuu yleensä siihen, että oliot voivat toimia keskenään vuorovaikutuksessa. Tilanne on sama ohjelmistokomponenttien yhteydessä. Ohjelmistokomponentit kuitenkin myös eroavat olioista monella tapaa. Ensinnäkin, ohjelmistokomponenttien toteutus ei välttämättä perustu olioteknologiaan, vaikka tilanne usein onkin tämä. Toiseksi, ohjelmistokomponentit voivat koostua useista, keskenään vuorovaikutuksessa toimivista, olioista. Tällöin ohjelmistokomponentin voidaan ajatella vastaavan olio-ohjelmistoa tai sen osaa. Muitakin yhteneväisyyksiä ja eroavaisuuksia on varmasti olemassa. Ohjelmistokomponenttien ja olioiden yhteiset piirteet mahdollistavat sen, että tässä tutkimuksessa voidaan hyödyntää joitakin oliolähestymistavan yhteydessä esitettyjä ajatuksia. Toisaalta ohjelmistokomponenttien ja olioiden väliset erot rajoittavat edellä mainittua ajatusten uudelleenkäyttöä.

2.1.2 Rajapinta

Rajapinta (interface) on abstraktio, joka kuvaa tietyn palvelun käyttötavan, mutta ei sen toteutusta (Szyperski, 1997, s. 374). Tämän yleisen määritelmän lisäksi Szyperski (1997) toteaa, että teknisestä näkökulmasta tarkasteltuna rajapinta voidaan ymmärtää joukkona operaatioita, jotka ovat asiakkaiden kutsuttavissa. Rajapintoina voidaan ymmärtää esimerkiksi luokkien julkiset osat, jotka voidaan kuvata luokan yhteydessä tai erillisessä otsikkotiedostossa (header file), sekä täysin abstraktit luokat. Szyperski (1997) käyttää edellisistä termiä proseduraalinen rajapinta (procedural interface) ja jälkimmäisistä termiä oliorajapinta (object interface). Szyperskin (1997) mukaan ohjelmistokomponentin suoraan tarjoamat rajapinnat (interfaces directly provided by a component) vastaavat perinteisten ohjelmistokirjastojen proseduraalisia rajapintoja, kun taas välillisesti

toteutetut rajapinnat (indirectly implemented interfaces) vastaavat oliorajapintoja. Jälkimmäisten tapauksessa ohjelmistokomponentin palveluita käytetään välillisesti yhden tai useamman olion kautta (ks. Szyperksi, 1997). Rajapintojen kuvaamiseen on kehitetty myös omia kieliä. OMG:n (Object Management Group) IDL-kieli (Interface Definition Language) on yksi tällainen (ks. Vinoski, 1997).

KUVASSA 1 on esitetty UML-notaatiota (Unified Modeling Language, Booch, Rumbaugh ja Jacobson, 1998) käyttäen ohjelmistokomponenttien ja rajapintojen välinen käsitteellinen suhde. Yksi ohjelmistokomponentti voi toteuttaa useita rajapintoja ja toisaalta saman rajapinnan voi toteuttaa moni ohjelmistokomponentti (Jaaksi ja Laitkorpi, 1999). On kuitenkin syytä huomata, että vaikka edellä esitettiin ohjelmistokomponenttien toteuttavan rajapinnoissa kuvatut palvelut, niin rajapinnat eivät ole ohjelmistokomponenteista erillisiä osia, vaan kuuluvat olennaisena osana ohjelmistokomponentteihin. Yhtä hyvin voitaisiin ajatella, että ohjelmistokomponentti koostuu rajapinnoista ja niiden toteutuksista (ks. Szyperski, 1997). Itse asiassa tämä olisi selkeämpi tapa kuvata ohjelmistokomponentti. Rajapinnat voivat olla esimerkiksi erillisiä luokkia, joiden kautta ohjelmistokomponentin palveluita käytetään.



KUVA 1. Ohjelmistokomponenttien ja rajapintojen suhde

Rajapinta koostuu joukosta semanttisesti yhteenkuuluvia operaatioita, jotka tarjoavat tietyn palvelun mielivaltaisille asiakkaille. Ohjelmistokomponentin tehtävänä on toteuttaa yhden tai useamman rajapinnan määrittelemä palvelu. Rajapinnat on ainoa tapa päästä käsiksi ohjelmistokomponentin toteuttamiin palveluihin, joiden toteutuksiin liittyvät yksityiskohdat ovat piilossa rajapintojen takana. (Jaaksi ja Laitkorpi, 1999)

Tässä tutkimuksessa käytettävässä ohjelmistokomponentin määritelmässä todetaan rajapintojen olevan *sopimuksenmukaisia*. Sopimukseen liittyy aina vähintään kaksi osapuolta; tässä tapauksessa ohjelmistokomponentti ja asiakas. Ohjelmistokomponentti sitoutuu toteuttamaan rajapinnan kuvaaman palvelun ja asiakas odottaa pystyvänsä käyttämään rajapinnassa määriteltyä palvelua (ks. Jaaksi ja Laitkorpi, 1999 tai Szyperski 1997). Sopimuksenmukaiset rajapinnat on erittäin oleellinen asia ohjelmistokomponenttien uudelleenkäytön kannalta. Niiden merkitystä ja sisältöä käsitellään tarkemmin luvussa 3.

2.1.3 Uudelleenkäyttö

Uudelleenkäyttö (reuse) tarkoittaa lyhyesti ilmaistuna sitä, että jotakin asiaa käytetään uudelleen. Kruegerin (1992, s. 131) määritelmän mukaan uudelleenkäyttö ohjelmistotuotannossa tarkoittaa sitä, että ohjelmistoja tuotetaan mieluummin rakentamalla niitä olemassaolevista ohjelmiston osista kuin tekemällä kaikki alusta alkaen itse.

Ohjelmistotuotannossa nämä uudelleenkäytettävät asiat voivat olla abstrakteja tai konkreettisia. Suunnittelumallien uudelleenkäyttö on yksi esimerkki abstraktin kohteen uudelleenkäytöstä. Suunnittelumallilla tarkoitetaan tapaa suunnitella ohjelmiston osa tietyn usein esiintyvän ongelman ratkaisemiseksi (Koskimies, 1997). Tavallaan suunnittelumalleja hyödynnettäessä käytetään uudelleen aikaisempaa kokemustietoa. Konkreettisia uudelleenkäytön kohteita ovat vaikkapa ohjelmistokomponentit, luokkakirjastot, luokat ja funktiot.

Ohjelmistokomponentteja tulee voida käyttää uudelleen tietämättä niiden toteutuksesta mitään. Koskimies (1997) käyttää olioparadigman yhteydessä tällaisesta uudelleenkäytöstä termiä *kokoava uudelleenkäyttö* (black-box reuse), mutta se pätee yhtä hyvin myös ohjelmistokomponenteille. Tämä johtuu siitä,

että sekä ohjelmistokomponenteilla että olioilla on rajapinta, joka mahdollistaa sen, että molempia voidaan käyttää uudelleen ilman, että niiden sisäistä toteutusta tarvitsee ymmärtää. Kokoava uudelleenkäyttö perustuu täysin määriteltyyn rajapintaan sekä mahdollisesti muihin ohjelmistokomponentin määriteltyihin ominaisuuksiin. Ohjelmistokomponentilla voi olla esimerkiksi tiettyjä ympäristöriippuvuuksia, joiden tunteminen on edellytyksenä sen uudelleenkäytölle.

Koskimiehen (1997) mukaan periyttämiseen perustuva eli *muuntava uudelleenkäyttö* (white-box reuse) on toinen keskeinen uudelleenkäytön oliomekanismi: kun aliluokka perii yliluokan ominaisuudet, se voi muuttaa perittyjä operaatioita antamalla niille uuden toteutuksen. Periyttämiseen perustuva uudelleenkäyttö ei kuitenkaan päde ohjelmistokomponenttien yhteydessä, vaan se on puhtaasti luokkiin ja olioihin liittyvä mekanismi. Ohjelmistokomponentti ei voi periä toista ohjelmistokomponenttia (ks. Szyperski, 1997).

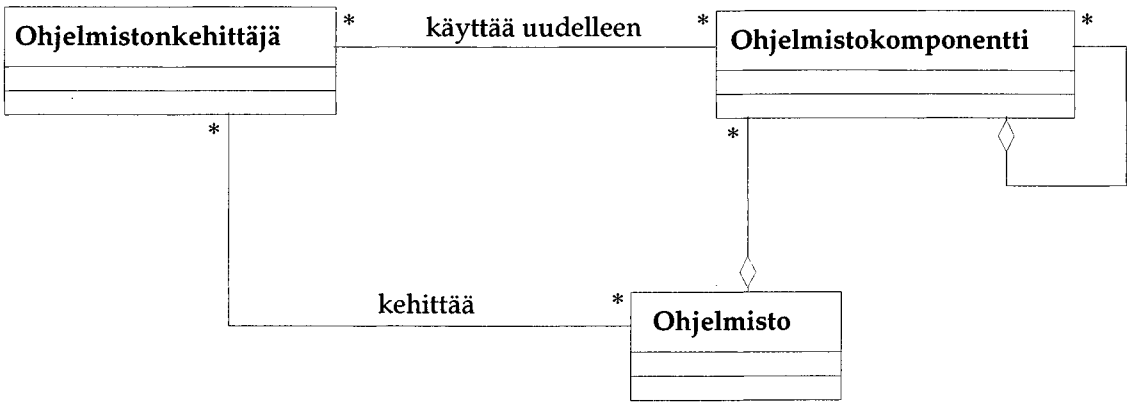
2.2 Ohjelmistojen 'rakentaminen' ohjelmistokomponenteista

Ohjelmistojen kehittämistä on usein verrattu muilla teollisuuden aloilla tapahtuvaan kehitystyöhön. Cox (1990) toteaa muiden teollisuuden alojen määrittelevän stadardoiduja tuotteita, joiden tuottamiseksi voidaan käyttää erilaisia prosesseja, kun taas ohjelmistotuotannon alueella määritellään standardoituja kieliä ja metodologioita, joiden avulla pyritään saamaan aikaiseksi ohjelmistokomponentteja. Coxin (1990) vertaus koskee siis uudelleenkäytön edistämiseksi tapahtuvaa kehitystyötä (development for reuse).

Sommerville (1995) puolestaan tarkastelee asiaa ohjelmistokomponenttien uudelleenkäytön kannalta (development with reuse). Hänen mukaansa useimpien insinöörialojen suunnitteluprosessit perustuvat komponenttien uudelleenkäyttöön, mutta ohjelmistoja kehitettäessä suurin osa ohjelmistokomponenteista

suunnitellaan ja toteutetaan kehitettävän ohjelmiston tarpeita ajatellen. Sommerville (1995) tarkoittaa ohjelmistokomponenteilla yleisesti vain ohjelmiston osia, joiden voidaan katsoa olevan tässä tutkimuksessa tarkasteltavien ohjelmistokomponenttien ylijoukko. Ohjelmistokomponentin määrittelyllä ei kuitenkaan ole vaikutusta Sommervillen (1995) havaintoon ohjelmistotuotannon luonteesta.

Tässä tutkimuksessa ymmärretty uudelleenkäytettäviin ohjelmistokomponentteihin perustuvan ohjelmistojen kehittämisen käsitteellinen malli on esitetty KUVASSA 2, joka noudattaa UML-notaatiota (Booch ym., 1998). Yleensä ohjelmiston kehittämiseen osallistuu joukko ohjelmistonkehittäjiä, jotka voivat olla esimerkiksi suunnittelijoita, ohjelmoijia tai projektipäälliköitä. Toisaalta yksi ohjelmistonkehittäjä voi olla mukana useammassa ohjelmistonkehittämishankkeessa. Ohjelmistonkehittäjät käyttävät uudelleen ohjelmistokomponentteja, joiden avulla he 'rakentavat' ohjelmistoja. Ohjelmisto koostuu ohjelmistokomponenteista, jotka voivat puolestaan koostua toisista ohjelmistokomponenteista (Jaaksi, 1998).



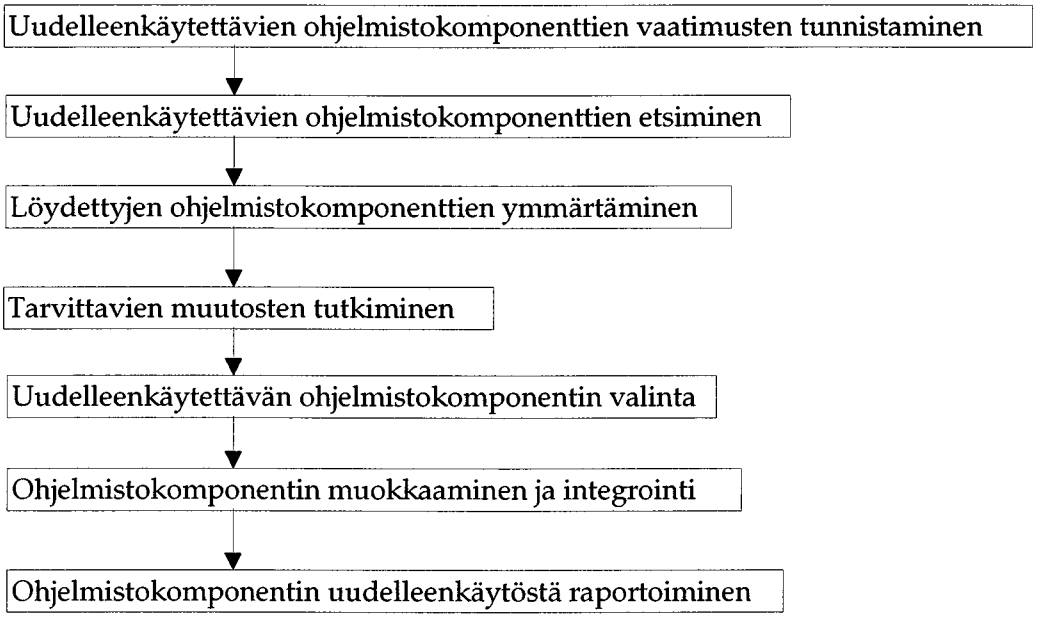
KUVA 2. Ohjelmistojen 'rakentaminen' ohjelmistokomponenteista

Seuraavaksi tarkastellaan mitä uusia tehtäviä uudelleenkäytettävien ohjelmistokomponenttien käyttö tuo perinteiseen ohjelmistonkehitysprosessiin. Tämän selvittämiseksi käydään läpi Karlssonin (1995) esittämät ohjelmistokomponenttien uudelleenkäytön tehtävät, Sommervillen (1995) kuvaus uudelleenkäyt-

töön perustuvasta ohjelmistonkehitysprosessista sekä Sametingerin (1997) esittämä uudelleenkäytön spiraalimalli, joka perustuu Boehmin (1988) spiraalimalliin. Kaikissa edellä mainituissa kuvauksissa ohjelmistokomponentti on määritelty eri tavalla kuin tässä tutkimuksessa. Sen vuoksi on tarpeen pohtia, miten Karlssonin (1995), Sommervillen (1995) ja Sametingerin (1997) kuvaukset soveltuvat suorituskelpoisille ohjelmistokomponenteille.

2.2.1 Ohjelmistokomponenttien uudelleenkäytön tehtävät

Karlssonin (1995) mukaan ohjelmistokomponenttien uudelleenkäytön prosessi on samanlainen kaikissa eri ohjelmistonkehitysvaiheissa. Hänen mukaansa ohjelmistokomponentteja on mahdollista tunnistaa yhtä hyvin niin analyysi-, suunnittelu- kuin testausvaiheessakin, mutta mitä aikaisemmassa vaiheessa ohjelmistokomponentti on käytettävissä uudelleen, sitä enemmän hyötyä siitä on seuraavien vaiheiden kannalta. Karlssonin (1995) yleiseen ohjelmistokomponenttien uudelleenkäytön prosessiin kuuluvat tehtävät on esitetty KUVASSA 3.



KUVA 3. Ohjelmistokomponenttien uudelleenkäytön tehtävät (mukaillen Karlsson, 1995, s. 346)

Karlsson (1995) on kuvannut tehtäviä seuraavasti. Uudelleenkäytettävien ohjelmistokomponenttien vaatimusten tunnistaminen edellyttää kohdealueen tuntemista. Vaatimusten ei tule olla liian yksityiskohtaisia, sillä tällöin uudelleenkäytettävien ohjelmistokomponenttien löytäminen saattaa olla vaikeaa. Ohjelmistokomponenttien etsiminen perustuu tunnistettuihin vaatimuksiin. Tulosjoukon tulee olla tarpeeksi iso, jotta eri vaihtoehtoja voidaan vertailla parhaan ohjelmistokomponentin valitsemiseksi. Ohjelmistokomponenttien etsiminen kohdistuu yleensä ohjelmistokomponenttivarastoihin, joiden tulee tarjota itse ohjelmistokomponenttien lisäksi tarpeelliset hakumekanismit sekä riittävästi tietoa ohjelmistokomponenteista. Löydettyjen ohjelmistokomponenttien ymmärtäminen edellyttää niiden toiminnallisten (functional) ja laadullisten (non-functional) piirteiden ymmärtämistä. Laadullisia piirteitä ovat esimerkiksi tehokkuus, siirrettävyys ja luotettavuus. Tarvittavien muutosten tutkimisen tarkoituksena on selvittää ne muutokset, jotka ohjelmistokomponentille täytyy tehdä, jotta se vastaisi sille esitettyjä vaatimuksia. Mikäli tarvittavia muutoksia on paljon, kannattaa miettiä, onko uuden ohjelmistokomponentin kehittäminen järkevämpää kuin tarvittavien muutosten tekeminen. Myös alkuperäisten vaatimusten muuttaminen ohjelmistokomponenttia vastaavaksi on mahdollista, mutta tällä on vaikutuksia ohjelmistonkehitysprosessin aikaisempiin vaiheisiin. Uudelleenkäytettävän ohjelmistokomponentin valinta sopivien joukosta voi riippua monista seikoista, kuten esimerkiksi taloudellisista ja laadullisista tekijöistä. Sopivien ohjelmistokomponenttien joukko koostuu sellaisista ohjelmistokomponenteista, jotka vastaavat niille esitettyihin vaatimuksiin. Ohjelmistokomponenttien muokkaaminen ja integrointi tarkoittaa aikaisemmin tutkittujen tarpeellisten muutosten toteuttamista ja ohjelmistokomponentin liittämistä osaksi ohjelmistoa. Ohjelmistokomponentin uudelleenkäytöstä raportoiminen ei vaikuta senhetkiseen ohjelmistonkehitystyöhön, mutta on tärkeää jatkossa tapahtuvaa ohjelmistokomponentin uudelleenkäyttöä varten. Raportissa on hyvä esittää kuvaus uudelleenkäytön ympäristöstä, mahdollisista ongelmista ohjelmistokomponentin uudelleenkäytön yhteydessä sekä uudelleenkäytön aiheuttamista kustannuksista.

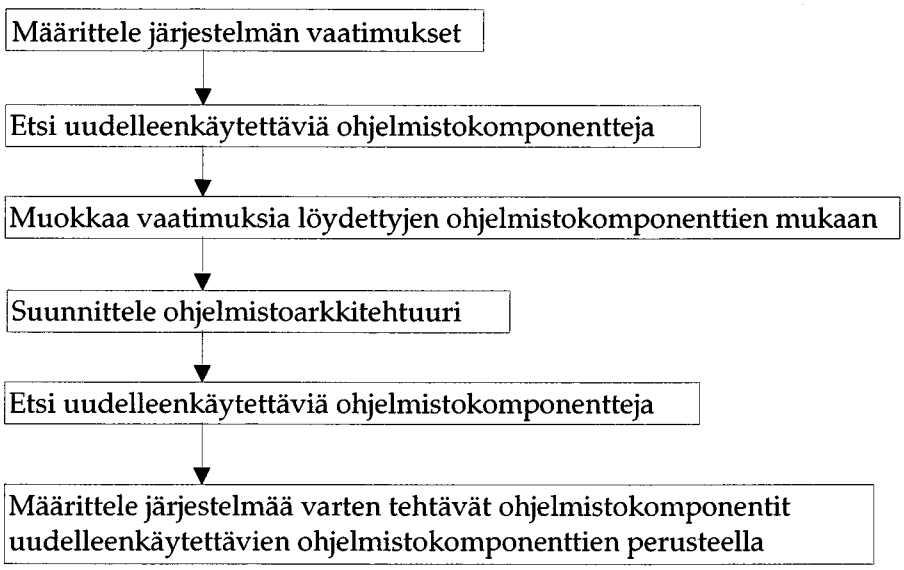
Karlssonin (1995) esittämä ohjelmistokomponenttien uudelleenkäytön prosessi soveltuu pienin tarkennuksin myös tämän tutkimuksen kohteena olevien suorituskelpoisten ohjelmistokomponenttien uudelleenkäyttöprosessin kuvaamiseen:

- Ohjelmistokomponenttien muokkaaminen täytyy ymmärtää ohjelmistokomponenttien mukauttamisena, sillä varsinaista lähdekoodin muuttamista ei suorituskelpoisille ohjelmistokomponenteille yleensä tehdä. Mukauttaminen tarkoittaa ohjelmistokomponentin julkisten ominaisuuksien muuttamista ilman, että halutut muutokset toteutetaan muokkaamalla suoraan ohjelmistokomponentin lähdekoodia (ks. Thomas, 1998). Julkisia ominaisuuksia ovat esimerkiksi käyttöliittymän painikkeen (button) näkyvä nimi tai tilienhallintaan käytetyn ohjelmistokomponentin (account management component) sisältämä tieto tietokannan sijainnista (ks. Thomas, 1998). Szyperski (1997) toteaa, että ohjelmistokomponentin ominaisuuksia voidaan mukauttaa sen toteutuksen aikana (assembly time), mutta myös myöhemmin, sen käytön aikana (runtime).
- Ohjelmistokomponentin uudelleenkäytöstä raportoiminen on uudelleenkäytön *mahdollistava* tehtävä eli se helpottaa uudelleenkäyttöä tulevaisuudessa, mutta se ei varsinaisesti ole ohjelmistokomponentin uudelleenkäytön *edellytyksenä* oleva tehtävä (Karlsson, 1995). Ohjelmistokomponentin uudelleenkäytöstä raportoiminen tapahtuu vasta sen uudelleenkäytön jälkeen, joten raportoimista ei voida pitää pakollisena ohjelmistokomponentin uudelleenkäytön tehtävänä.

2.2.2 Uudelleenkäyttöön perustuva ohjelmistonkehitysprosessi

Sommervillen (1995) esittämä ohjelmistokomponenttien uudelleenkäyttöön perustuva ohjelmistonkehitysprosessi kuvaa varsin yleisellä tasolla miten uudelleenkäytettävät ohjelmistokomponentit vaikuttavat perinteiseen ohjelmistonkehitysprosessiin. Uudelleenkäytettävien ohjelmistokomponenttien etsiminen on

ainoa suoranaisesti ohjelmistokomponenttien uudelleenkäyttöön liittyvä tehtävä. Sen on määrä tapahtua kehitettävälle ohjelmistolle määriteltyjen vaatimusten tai suunnitellun ohjelmistoarkkitehtuurin perusteella. Merkittävää on, että Sommervillen (1995) mukaan löydettyjen ohjelmistokomponenttien perusteella tulee muuttaa ohjelmistolle asetettuja vaatimuksia. Ohjelmistokomponenttien muokkaamista ei esitetä edes vaihtoehtoisena tehtävänä. Sommerville (1995) toteaa vain, että uudelleenkäytettävien ohjelmistokomponenttien mukanaan tuomat hyödyt, kuten pienemmät ohjelmistonkehittämiskustannukset ja luotettavampi ohjelmisto, kompensoivat vaatimusten muuttamisen sellaisiksi, että löydettyjä ohjelmistokomponentteja on mahdollista käyttää ohjelmiston kehittämisessä. Sommervillen (1995) uudelleenkäyttöön perustuva ohjelmistonkehitysprosessi on esitetty kokonaisuudessaan KUVASSA 4.



KUVA 4. Uudelleenkäyttöön perustuva ohjelmistonkehitysprosessi (mukaillen Sommerville, 1995, s. 399)

Sommervillen (1995) kuvaamassa ohjelmistonkehitysprosessissa uudelleenkäytettävien ohjelmistokomponenttien etsiminen tapahtuu kahdessa vaiheessa; ensin ohjelmistolle määriteltyjen vaatimusten ja sitten suunnitellun ohjelmistoarkkitehtuurin perusteella. Tämä tukee Karlssonin (1995) havaintoa uudel-

leenkäytettävien ohjelmistokomponenttien tunnistamisesta missä tahansa ohjelmiston kehittämisen vaiheessa.

Sametinger (1997) on kuvannut uudelleenkäyttöön perustuvaa ohjelmistonkehitysprosessia yhdistämällä Hooperin ja Chesterin (1991) esittämät ohjelmistokomponenttien uudelleenkäyttöön liittyvät tehtävät Boehmin (1988) spiraalimalliin. Tuloksena on uudelleenkäytön spiraalimalli (reuse spiral), jonka neljännekset sisältävät seuraavat ohjelmistokomponenttien uudelleenkäyttöön liittyvät tehtävät:

1. neljännes: tavoitteiden, vaihtoehtojen ja rajoitteiden asettaminen

Tehtäviä ovat kohdealueen ymmärtäminen, kehitettävän ohjelmiston vaatimusten esittäminen, uudelleenkäytettävien ohjemistokomponenttien etsiminen, ohjelmistokomponenttien arviointi ja ohjelmiston vaatimusten muuttaminen löydettyjen ohjelmistokomponenttien perusteella. Ensimmäisen neljänneksen tuloksena ovat ehdokkaat uudelleenkäytettäviksi ohjelmistokomponenteiksi sekä ohjelmiston vaatimukset.

2. neljännes: vaihtoehtojen arviointi; riskien tunnistaminen ja niihin varautuminen

Tehtävät koskevat ohjelmistokomponenttien valintaa ja niiden muokkaamista. Tavoitteena on selvittää ohjelmistokomponenteille tehtävät tarpeelliset muutokset, kuinka paljon työtä muutokset aiheuttavat, ja mitä riskejä niiden toteuttamiseen liittyy. Eri ohjelmistokomponenttivaihtoehtoja arvioidaan vaadittavien työmäärien ja riskien perusteella. Riskeihin voidaan varautua esimerkiksi hankkimalla ohjelmistokomponenteista riittävästi tietoa tai kokeilemalla niitä. Toisen neljänneksen tuloksena on päätös siitä, kuinka kehittämisen kohteena oleva ohjelmiston osa tulee toteuttaa eli mitä ohjelmistokomponentteja käytetään uudelleen ja mitä muutoksia niille on syytä tehdä.

3. neljännes: kehittäminen

Tehtäviä ovat ohjelmistokomponenttien muokkaaminen ja niiden integroiminen kehittämisen kohteena olevaan ohjelmiston osaan. Kolmannen neljänneksen tuloksena on kehittämisen kohteena olevan ohjelmiston osan toteutus.

4. neljännes: seuraavan vaiheen suunnittelu

Tehtävänä on arvioida muokattujen tai uusien ohjelmistokomponenttien uudelleenkäyttömahdollisuuksia jatkossa, sillä mahdollisesti uudelleenkäytettävät ohjelmistokomponentit kannattaa tallettaa erityiseen ohjelmistokomponenttien kuvauskantaan (repository).

Uudelleenkäytön spiraalimalli ei korvaa alkuperäistä Boehmin (1988) spiraalimallia, vaan kuvaa yhden mahdollisen tavan kehittää ohjelmistoa spiraalimallin mukaisesti. Vaikka ohjelmistoa lähdettäisiin kehittämään uudelleenkäytettävien ohjelmistokomponenttien avulla, voidaan kuitenkin päätyä perinteiseen ohjelmistonkehitysratkaisuun, jossa kaikki tehdään alusta alkaen itse. (Sametinger, 1997)

Sametingerin (1997) esittämässä uudelleenkäytön spiraalissa on mukana sekä ohjelmiston vaatimusten muuttaminen löydettyjen ohjelmistokomponenttien perusteella että ohjelmistokomponenttien muokkaaminen. Sommervillen (1995) kuvaamassa uudelleenkäyttöön perustuvassa ohjelmistonkehitysprosessissa puolestaan muutetaan ohjelmistolle asetettuja vaatimuksia eikä huomioida lainkaan ohjelmistokomponenttien muokkaamisen mahdollisuutta. Sommerville (1995) tosin mainitsee yhtenä uudelleenkäyttöön perustuvan ohjelmistonkehitysprosessin vaatimuksena sen, että ohjelmistokomponenttien mukana täytyy tulla dokumentaatio, joka auttaa käyttäjää ymmärtämään ja muokkaamaan ohjelmistokomponentteja uuden ohjelmiston vaatimuksia vastaaviksi. On tärkeää, että ohjelmistokomponentteja voidaan jollakin tapaa sovittaa kehitettävän ohjelmiston tarpeisiin siten, että niitä voidaan käyttää ilman, että ohjelmistolle asetettuja vaatimuksia joudutaan aina muuttamaan.

Sametingerin (1997) uudelleenkäytön spiraalimalli sisältää samat tehtävät kuin Karlssonin (1995) ohjelmistokomponenttien uudelleenkäytön prosessi, jonka todettiin soveltuvan pienin muutoksin myös suorituskelpoisten ohjelmistokomponenttien uudelleenkäyttöön. Uudelleenkäytön spiraalimalli sisältää lisäksi Boehmin (1988) spiraalimallin tavoin riskien hallinnan osana ohjelmistonke-

hitysprosessia. Sametingerin (1997) uudelleenkäytön spiraalimalli tarjoaa siis hyvän tuen ohjelmistokomponenttien uudelleenkäyttöön perustuvalle ohjelmistonkehitysprosessille. Sommervillen (1995) kuvaus on esitetty niin yleisellä tasolla, ettei siitä juurikaan ole apua käytännön ohjelmistonkehitystyöhön.

2.3 Yhteenveto

Tässä kappaleessa kuvattiin tutkimuksen kannalta keskeiset käsitteet, jotka ovat ohjelmistokomponentti, rajapinta ja uudelleenkäyttö, sekä käsitteiden väliset suhteet. Tämän lisäksi esitettiin mitä tehtäviä ohjelmistokomponenttien uudelleenkäyttöön liittyy sekä kuinka nämä tehtävät vaikuttavat perinteiseen ohjelmistonkehitysprosessiin.

Ohjelmistokomponenteille on esitetty kirjallisuudessa paljon erilaisia määritelmiä, jotka saattavat poiketa toisistaan huomattavastikin. Laajimmassa merkityksessä ohjelmistokomponentti voidaan ymmärtää yleisesti ohjelmiston osana, joka voi tarkoittaa esimerkiksi ohjelmiston dokumentaatiota, lähdekoodia tai suunnittelumalleja. Tässä tutkimuksessa noudatetaan Szyperskin (1997) esittämää ohjelmistokomponentin määritelmää, joka rajoittaa tarkasteltavan joukon suorituskelpoisiin ohjelmistokomponentteihin.

Rajapinnat kuvaavat ohjelmistokomponentin tarjoamat palvelut, mutta eivät niiden toteutusta. Ohjelmistokomponentti voi toteuttaa useita rajapintoja ja toisaalta saman rajapinnan voi toteuttaa moni ohjelmistokomponentti. Ohjelmistokomponenteilla ja olioilla on monia yhteisiä piirteitä, mutta ne myös eroavat monessa suhteessa toisistaan. Esimerkiksi ohjelmistokomponenttien uudelleenkäyttö rajoittuu kokoavaan uudelleenkäyttöön, kun taas olioluokkien yhteydessä voidaan soveltaa myös muuntavaa uudelleenkäyttöä.

Karlssonin (1995) esittämä yleinen ohjelmistokomponenttien uudelleenkäytön prosessi soveltuu seuraavin pienin muutoksin myös suorituskelpoisten ohjelmistokomponenttien uudelleenkäytön kuvaamiseen. Ensinnäkin, suorituskelpoisten ohjelmistokomponenttien lähdekoodia ei voida muuttaa, joten muuttaminen tulee ymmärtää mukauttamisena. Toiseksi, ohjelmistokomponentin uudelleenkäytöstä raportoimista ei voida pitää pakollisena uudelleenkäytön tehtävänä, koska se tapahtuu vasta ohjelmistokomponentin uudelleenkäytön jälkeen. Jälkimmäistä tehtävää koskeva huomio ei liity pelkästään suorituskelpoisten ohjelmistokomponenttien uudelleenkäyttöprosessiin, vaan sen voidaan ajatella pätevän yleisesti erityyppisten ohjelmistokomponenttien uudelleenkäytön yhteydessä.

Sekä Sommerville (1995) että Sametinger (1997) ovat kuvanneet ohjelmistokomponenttien uudelleenkäyttöön perustuvaa ohjelmistonkehitysprosessia. Sommervillen (1995) kuvaus on esitetty niin yleisellä tasolla, ettei siitä ole apua käytännön ohjelmistonkehitystyöhön. Sametingerin (1997) uudelleenkäytön spiraalimalli sen sijaan on huomattavasti seikkaperäisempi ja sisältää Karlssonin (1995) esittämät ohjelmistokomponenttien uudelleenkäytön tehtävät jaettuna eri ohjelmistonkehitysvaiheille. Sametingerin (1997) uudelleenkäytön spiraalimalli perustuu Boehmin (1988) spiraalimalliin, joten siihen liittyy olennaisesti riskien hallinta osana ohjelmistonkehitysprosessia. Näiden huomioiden perusteella Sametingerin (1997) uudelleenkäytön spiraalimalli tarjoaa hyvän pohjan ohjelmistokomponenttien uudelleenkäyttöön perustuvalle ohjelmistonkehitysprosessille.

3 SOPIMUKSENMUKAISET RAJAPINNAT

Tässä kappaleessa tarkastellaan mitä ohjelmistokomponentin sopimuksenmukaisilla rajapinnoilla tarkoitetaan sekä mitä tietoja tällaiset rajapinnat sisältävät. Tarkoituksena on selvittää miksi sopimuksenmukaisissa rajapinnoissa tulee kuvata tietyt asiat, kuten ohjelmistokomponentin toiminta ja laatu, sekä mitä seikkoja näiden asioiden kuvaamiseen liittyy. Tämän tutkimuksen kantavana ajatuksena on se, että ohjelmistokomponentin sopimuksenmukaisten rajapintojen tulee antaa mahdollisimman oikea kuva ohjelmistokomponentin toiminnallisuudesta. Edellisessä kappaleessa esitetyistä Karlssonin (1995) kuvaamista ohjelmistokomponenttien uudelleenkäytön tehtävistä ohjelmistokomponenttien ymmärtämisen voidaan ajatella kuuluvan keskeisimmin tämän tutkimuksen piiriin.

Koska tässä kappaleessa käsitellään ohjelmistokomponenttien ominaisuuksia teknisestä näkökulmasta, on syytä ottaa kantaa myös ohjelmistokomponenttien toteutukseen liittyviin seikkoihin. Tarkastelun kohteena ovat olioperustaiset ohjelmistokomponentit eli sellaiset ohjelmistokomponentit, joiden toteutus perustuu olioteknologioiden hyödyntämiseen. Vaikka tämän kappaleen tarkastelunäkökulma on tekninen, edellyttää järjestelmällinen ohjelmistokomponenttien uudelleenkäyttö myös organisatorisen näkökulman huomioimista (ks. Tracz, 1995 tai Braun, 1994).

3.1 Mitä sopimuksenmukaisilla rajapinnoilla tarkoitetaan?

Rajapintojen määrityksien voidaan ajatella olevan ohjelmistokomponentin asiakkaiden ja toteuttajien välistä sopimuksia (Szyperski, 1997). Asiakkaat ja toteuttajat voidaan ymmärtää ohjelmistokomponentin toteutuksen aikana esimerkiksi ohjelmistonkehittäjinä, jolloin rajapinnan toteuttaja on vastuussa siitä ohjelmis-

tokomponentista, jonka tulee toteuttaa sovitussa rajapinnassa määritellyt palvelut. Asiakas voi puolestaan kehittää toista ohjelmistokomponenttia, joka käyttää rajapinnassa määriteltyjä palveluita eli sovitun rajapinnan toteuttavaa ohjelmistokomponenttia. Tällä tavalla sopimalla ohjelmistokomponentin rajapinnasta voidaan ohjelmiston kehittäminen jakaa useisiin rinnakkaisiin tehtäviin (ks. Jaaksi ja Laitkorpi, 1999). Ohjelmistokomponentin käytön aikana asiakkaiden voidaan ajatella olevan toisia ohjelmistokomponentteja tai mahdollisesti ohjelmiston käyttäjiä, mikäli ohjelmistokomponentti tarjoaa palveluita suoraan käyttäjille. Toteutuksen aikaisena toteuttajana voidaan puolestaan ymmärtää rajapinnan toteuttava ohjelmiston osa eli ohjelmistokomponentin toteutus. Tässä tutkimuksessa ohjelmistokomponentteja tarkastellaan uudelleenkäytön näkökulmasta, joten toteuttajana voidaan jatkossa ymmärtää ohjelmistokomponentin toteutus.

3.1.1 Tarvittavan tiedon määrä

Edellä kuvattiin, mitä sopimuksenmukaisilla rajapinnoilla tarkoitetaan ohjelmistokomponenttien yhteydessä, mutta ei sitä, kuinka paljon tietoa näiden rajapintojen tulee käyttäjilleen tarjota. Kuten aikaisemmin todettiin, rajapinnan tulee mahdollistaa ohjelmistokomponentin käyttö ilman, että käyttäjän tarvitsee tietää, miten ohjelmistokomponentti on toteutettu. Tarvittavan tiedon määrän selvittämiseksi käytetään Parnasin (1972) periaatteita:

- Moduulin käyttäjälle tulee tarjota kaikki se tieto, joka on tarpeen moduulin oikeaoppisen käytön kannalta, eikä muuta tietoa.
- Moduulin toteuttajalle tulee tarjota kaikki se tieto, joka on tarpeen moduulin toteuttamisen kannalta, eikä muuta tietoa.

Parnasin (1972) periaatteita voidaan soveltaa hyvin myös ohjelmistokomponenteille, jotka voidaan mieltää eräänlaisiksi moduuleiksi. Ohjelmistokomponentin käyttäjälle tuleekin tarjota vain se tieto, joka on edellytyksenä ohjelmistokompo-

nentin oikeaoppiselle käytölle. Edelleen, ohjelmistokomponentin toteuttajalle tulee tarjota vain se tieto, joka on edellytyksenä ohjelmistokomponentin toteuttamiselle. Budd (1997) toteaa, että ohjelmistokomponenttien onnistuneen uudelleenkäytön kannalta on tärkeää, että ohjelmiston eri osien välillä on minimaaliset ja hyvin ymmärretyt yhteydet. Ohjemiston eri osilla tarkoitetaan tässä yhteydessä tietyn toiminnallisuuden sisältäviä osia. Esimerkiksi ohjelmistokomponentit tai oliot lukeutuvat näihin, mutta ohjelmiston dokumentaatio ei.

3.1.2 Korvattavuus

Kun ohjelmistokomponentti korvataan toisella, ohjelmistokomponentin rajapinta voi joko säilyä samana tai muuttua. Mikäli rajapinta säilyy samana toteutuksen muuttuessa, on kyse toteutuksen korvaamisesta. Jos taas rajapinta muuttuu ohjelmistokomponentin myötä, kyse ei ole korvaamisesta, vaan normaalista ohjelmistonkehitystyöstä, jonka vaikutukset eivät ulotu pelkästään uuteen ohjelmistokomponenttiin vaan myös muuhun ohjelmistoon.

Rajapinnan ja toteutuksen erottaminen toisistaan mahdollistaa sen, että rajapinnan toteutus voidaan vaihtaa toiseen ilman, että se vaikuttaa toisiin ohjelmistokomponentteihin millään tavalla (Budd, 1997). Korvattavuus tarkoittaa juuri tätä. Jotta rajapinnan uusi toteutus todella korvaisi alkuperäisen toteutuksen ilman, että ohjelmistokomponentin asiakkaat huomaavat eroa toteutuksien välillä, korvaavan toteutuksen täytyy toimia ulkoisesti samalla tavalla kuin alkuperäisen toteutuksen. Ei siis riitä, että uuden toteutuksen operaatiot täsmäävät syntaktisesti rajapinnan operaatioiden kanssa. Mikäli korvaava toteutus on sopimuksenmukaisen rajapinnan mukainen, voidaan olettaa, että uusi toteutus ei aiheuta ongelmia ohjelmistokomponentin asiakkaille. Korvattavuudesta ja sen edellytyksistä on kirjoitettu lähinnä olio-ohjelmoinnin yhteydessä (ks. Budd, 1997 tai Coplien, 1992).

Ohjelmistokomponentin rajapintoihin kohdistuvat muutokset aiheuttavat ongelmia, sillä tällöin ohjelmistokomponenttia ei voida enää käyttää kuten aikaisemmin. Tämä tarkoittaa sitä, että myös ohjelmistokomponentin asiakkaita tulee muuttaa, jotta ne voisivat edelleen käyttää ohjelmistokomponentin palveluita. Yhdeksi ratkaisuksi ohjelmistokomponentin rajapintojen muutosten aiheuttamiin ongelmiin on ehdotettu rajapintojen kuvaamista uudelleenkäyttösopimuksina (reuse contracts), joiden avulla on mahdollista esittää ohjelmistokomponentin rajapintojen evoluutio (ks. Steyaert, Lucas, Mens ja D'Hondt, 1996 tai Hondt, Lucas ja Steyaert, 1997). Ohjelmistokomponentin rajapintojen muutosten tutkiminen ei kuitenkaan kuulu tämän tutkimuksen piiriin.

3.2 Mitä sopimuksenmukaiset rajapinnat sisältävät?

Szyperskin (1997) mukaan sopimuksenmukaisten rajapintojen tulee sisältää riittävästi tietoa ohjelmistokomponentin toiminnasta ja laadusta. Molemmat ovat tärkeitä asioita ohjelmistokomponentin uudelleenkäytön kannalta, sillä uudelleenkäytettävän ohjelmistokomponentin tulee täyttää ympäristön asettamat niin toiminnalliset kuin laadullisetkin vaatimukset. Seuraavaksi tarkastellaan yksittäisen ohjelmistokomponentin toiminnallisuutta sekä ohjelmistokomponenttien väliseen vuorovaikutukseen liittyviä asioita. Tämän jälkeen esitetään, miksi myös ohjelmistokomponentin laatu on tärkeä asia uudelleenkäytön kannalta, sekä mitkä ohjelmistokomponentin laadulliset piirteet sopimuksenmukaisissa rajapinnoissa tulee kuvata ja miksi.

3.2.1 Ohjelmistokomponentin toiminta

Ohjelmistokomponentin uudelleenkäytettävyyden perustana voidaan pitää sitä, että ohjelmistokomponentti toimii oikein. Mikäli ohjelmistokomponentti ei toimi odotetulla tavalla, ei muillakaan tekijöillä, kuten laadulla tai muokattavuudella,

ole merkitystä. Toisaalta ohjelmistokomponentin toiminnan oikeellisuus ei ole riittävä tae sille, että ohjelmistokomponentti sopii uudelleenkäytettäväksi.

Ohjelmistokomponentin toiminnan tarkastelussa tulee huomioida itsenäisen toiminnan lisäksi ohjelmistokomponenttien vuorovaikutus. Seuraavaksi esitetään, kuinka esi- ja jälkiehdot soveltuvat ohjelmistokomponentin toiminnan oikeellisuuden osoittamiseen, sekä miten ulkoiset – ja takaisinkutsut vaikuttavat ohjelmistokomponentin toiminnan oikeellisuuden arviointiin.

Ohjelmistokomponentin toiminnan oikeellisuutta voidaan tarkastella *esi- ja jälkiehtojen* (pre- and postconditionds, Dijkstra, 1976) avulla. Esi- ja jälkiehtojen toteutumista on mahdollista seurata yksittäisten operaatioiden tasolla tai yleisesti ohjelmistokomponenttien välillä vallitsevien kausaalisten riippuvuuksien tarkastelussa. Krasnerin ja Popen (1988) esittämää MVC-paradigmaa (Model, View, Controller) esimerkkinä käyttäen voidaan todeta yleinen olioiden välinen kausaalisuus, jonka mukaan mallin (model) muuttaminen johtaa näkymän (view) päivitykseen. Mallilla tarkoitetaan tässä jotakin liiketoiminta-alueen kohdetta kuvaavaa oliota, kuten esimerkiksi pankkitiliä. Näkymällä tarkoitetaan puolestaan käyttöliittymän tai sen osan esittävää oliota, esimerkiksi pankkitilin saldon kertovaa näyttöä. Samoin voidaan tarkastella ohjelmistokomponenttien välisiä riippuvuuksia.

Yksittäisen operaation tapauksessa asiakkaan täytyy muodostaa sopimuksen mukainen esiehto ennen operaation kutsumista, ja toteuttaja voi luottaa siihen, että esiehto on voimassa aina kun operaatiota kutsutaan. Toteuttajan täytyy puolestaan muodostaa sopimuksen mukainen jälkiehto ennen operaation päättymistä, ja asiakas voi luottaa siihen, että jälkiehto on voimassa aina operaation päättyessä. Asiakkaan ja toteuttajan täytyy siis täyttää sopimuksessa määritellyt ehdot, mutta kumpikin voi vapaasti ylittää velvollisuutensa. Tämä tarkoittaa sitä, että asiakas voi vahvistaa esiehtoja tai heikentää jälkiehtoja, mutta ei päin-

vastoin. Toteutus voi puolestaan heikentää esiehtoja tai vahvistaa jälkiehtoja, mutta ei päinvastoin. (Szyperski, 1997)

Esi- ja jälkiehtojen lisäksi ohjelmistokomponentin toiminnan oikeellisuutta voidaan arvioida muiden käsitteiden avulla. Tällaisia käsitteitä ovat *invariantit* ja ohjelmistokomponentin *tila*. Invarianteilla (ks. Szyperski, 1997 tai Helm, Holland ja Gangopadhyay, 1990) tarkoitetaan tässä tapauksessa operaation suorituksen ajan voimassaolevia ehtoja. Esimerkiksi ehto, jonka mukaan operaatio ei saa muuttaa ohjelmistokomponentin attribuuttien arvoja, on invariantti. Esimerkin kaltainen invariantti voidaan toteuttaa myös ohjelmointikielen tasolla. Esimerkiksi C++-ohjelmointikielessä tämä tapahtuu operaation eteen liitettävän `const`-avainsanan avulla, jolloin operaatiota estetään muuttamasta olion attribuuttien arvoja (ks. Stroustrup, 1997). Ohjelmistokomponentin tilan (ks. Szyperski, 1997 tai Büchi ja Weck, 1997) voidaan puolestaan ymmärtää muodostuvan ohjelmistokomponentin attribuuttien tietyn ajanhetken mukaisista arvoista. Koska ohjelmistokomponentti voi koostua useista olioista, on ohjelmistokomponentin attribuutteina ymmärrettävä tässä tapauksessa kaikki ohjelmistokomponentin sisältämien olioiden attribuutit.

Büchi ja Weck (1997) sekä Wegner (1997) toteavat, että ohjelmistokomponentin toiminnan arviointi pelkkien syötteiden ja tulosteiden (input-output behaviour) avulla ei ole riittävää, mikäli ohjelmistokomponentti on vuorovaikutuksessa muiden ohjelmistokomponenttien kanssa. Syötteillä ja tulosteilla he viittaavat edellä esiteltyihin esi- ja jälkiehtoihin. Büchin ja Weckin (1997) mukaan ohjelmistokomponentin käyttöönottajan tulee saada tietää esi- ja jälkiehtojen lisäksi osa ohjelmistokomponentin sisäisestä toteutuksesta, erityisesti mitkä ovat ne tilat, joissa ohjelmistokomponentti tekee *ulkoisia kutsuja* (external calls) sekä mikä on näiden kutsujen suoritusjärjestys. Ulkoisilla kutsuilla tarkoitetaan ohjelmistokomponentin tekemiä kutsuja, joita se joutuu tekemään voidakseen toteuttaa rajapinnassaan määritellyn palvelun. Nämä kutsut voivat kohdistua esimerkiksi olioihin tai toisiin ohjelmistokomponentteihin. Büchi ja Weck (1997)

perustelevat ulkoisiin kutsuihin liittyvien tilojen kuvaamista sillä, että ulkoisen kutsun toteutus on usein riippuvainen sen tekevän ohjelmistokomponentin tilasta. Tämä tarkoittaa sitä, että ohjelmistokomponentin uudelleenkäyttäjä saattaa itse toteuttaa tai uudelleenmääritellä (override, ks. Budd, 1997) jonkin toisen olion tai ohjelmistokomponentin metodin, jota uudelleenkäytettävä ohjelmistokomponentti kutsuu. Büchi ja Weck (1997) perustelevat väitettään esi- ja jälkiehtojen riittämättömyydestä ohjelmistokomponenttien toiminnan määrittelyssä sillä, että ohjelmistokomponentin tekemien ulkoisten kutsujen toteuttamisessa saatetaan tarvita tietoa ohjelmistokomponentin sisäisestä toteutuksesta.

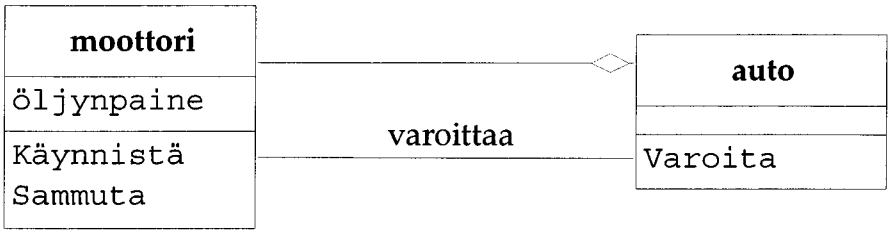
Staunstrupin (1997) mukaan rajapinta kuvaa ohjelmistokomponentin ulkoisesti näkyvän käyttäytymisen, kuten syntaksin sekä semanttisen tulkinnan ohjelmistokomponentin kommunikoinnista ympäristönsä kanssa. Staunstrupin (1997) määritelmä tukee Büchin ja Weckin (1997) huomiota siitä, että esi- ja jälkiehdot ovat riittämättömiä ohjelmistokomponentin toiminnan kuvaamiseen. Itse asiassa ohjelmistokomponentin ulkoisiin kutsuihin liittyvien seikkojen kuvaamisen voidaan ajatella kuuluvan Staunstrupin (1997) esittämään semanttiseen tulkintaan ohjelmistokomponentin kommunikoinnista ympäristönsä kanssa.

Huomattavaa on, että hyödynnettäessä tietoa ohjelmistokomponentin sisäisestä toteutuksesta esimerkiksi toisen ohjelmistokomponentin suunnittelun ja toteutuksen yhteydessä, on olemassa vaara, että ulkoisen kutsun toteutuksen tarjoava ohjelmistokomponentti tulee riippuvaiseksi sitä käyttävästä ohjelmistokomponentista. Büchi ja Weck (1997) tyrmäävätkin ajatuksen siitä, että ohjelmistokomponentin sisäinen toteutus olisi kokonaisuudessaan toisten ohjelmistokomponenttien suunnittelijoiden ja toteuttajien tarkasteltavissa. Edellä esiteltyjä Parnasin (1972) periaatteita soveltaen voidaan todeta, että mikäli ohjelmistokomponentin toteutuksessa tarvitaan tietoa sitä käyttävien ohjelmistokomponenttien sisäisestä toiminnasta, tässä tapauksessa tilat, joissa ulkoiset kutsut tehdään sekä näiden kutsujen suoritusjärjestys, on tämä tieto oltava ul-

koiset kutsut toteuttavien ohjelmistokomponenttien suunnittelijoiden ja toteuttajien käytettävissä.

Jaaksi ja Laitkorpi (1999) sekä Olafsson ja Doug (1997) erottavat sisääntulevat ja ulosmenevät rajapinnat (incoming and outgoing interfaces) toisistaan. Sisääntulevilla rajapinnoilla tarkoitetaan ohjelmistokomponentin palveluita kuvaavia rajapintoja ja ulosmenevillä rajapinnoilla riippuvuuksia toisista rajapinnoista. Ulosmenevät rajapinnat vastaavat Büchin ja Weckin (1997) kuvaamia ulkoisia kutsuja. Jaaksi ja Laitkorpi (1999) toteavat, että ohjelmistokomponenttien väliset suhteet voidaan kuvata sisääntulevilla ja ulosmenevillä rajapinnoilla.

Takaisinkutsut (callbacks, ks. Szyperski, 1997 tai Jaaksi, 1996b) eroavat ulkoisista kutsuista ainakin kahdella tavalla. Ensinnäkin, takaisinkutsut kohdistuvat ohjelmistokomponentin asiakkaaseen. Toiseksi, takaisinkutsut saattavat tapahtua asynkronisesti. Takaisinkutsut eivät siis välttämättä tapahdu ulkoisten kutsujen tapaan asiakkaan tekemän palvelupyynnön perusteella, vaan esimerkiksi jonkin ehdon täyttymisen tai ohjelmistokomponentin tilan muutoksen seurauksena. Jaaksi (1996b) käyttää olioiden yhteydessä esimerkkinä asynkronisesta takaisinkutsusta tilannetta, jossa moottorin tulee varoittaa autoa, mikäli moottorin öljynpaine laskee alle hyväksytyn rajan. KUVASSA 5 on esitetty UML-notaation (Booch ym., 1998) mukainen luokkakaavio, joka voidaan tulkita siten, että auton osana on moottori ts. auto koostuu moottorista. Koska auto koostuu moottorista, sillä on viite moottoriin ja se voi näin kutsua moottorin `Käynnistä`- ja `Sammuta`-metodeja. Moottorin täytyy puolestaan voida kutsua auton `Varoita`-metodia, mikäli moottorin öljynpaine laskee alle sallitun rajan. Jotta moottori voisi kutsua autoa, sen täytyy saada viite autoon esimerkiksi konstruktorissaan. Moottorin kutsuessa auton `Varoita`-metodia öljynpaineen laskemisen seurauksena, on kyse asynkronisesta takaisinkutsusta. Jatkossa takaisinkutsut erotellaan synkronisiin ja asynkronisiin takaisinkutsuihin vain, jos tähän on erityistä tarvetta. Mikäli takaisinkutsujen välillä ei tehdä eroa, pätee esitetty asia sekä synkronisten että asynkronisten takaisinkutsujen osalta.

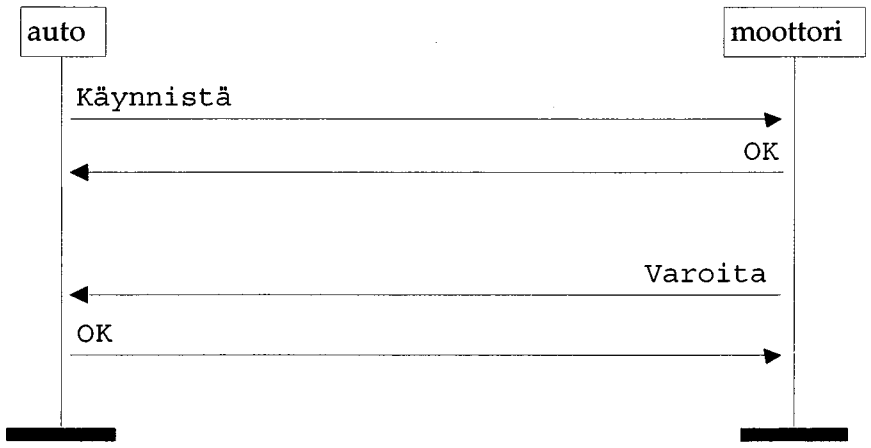


KUVA 5. Auto koostuu moottorista (mukaillen Jaaksi, 1996b, s. 94)

Myös ohjelmistokomponenttien yhteydessä asiakkaan on jollakin tapaa rekisteröidyttävä ohjelmistokomponentille, jotta takaisinkutsut olisivat mahdollisia. Olioperustaisten ohjelmistokomponenttien yhteydessä tämä tarkoittaa sitä, että takaisinkutsuja suorittavalle ohjelmistokomponentille tulee toimittaa asiakkaan viite tai viitteet, joiden avulla ohjelmistokomponentti voi tarvittaessa suorittaa takaisinkutsuja. Takaisinkutsuista voi aiheutua myös erinäisiä ongelmia, jotka aiheuttavat hankaluuksia ohjemistokomponentin toiminnan oikeellisuuden kannalta. Szyperski (1997) on käsitellyt näitä ongelmia proseduraalisen ohjelmistokirjaston (procedural library) tapauksessa.

Tiukassa proseduraalisen ohjelmistokirjaston mallissa (strict procedural library model) asiakas kutsuu aina ohjelmistokirjastoa eikä päinvastoin. Tällöin ohjelmistokirjasto saa suorittamansa toimenpiteet päätökseen ennen kontrollin palaamista takaisin asiakkaalle. Asiakas ei siis koskaan havaitse niitä välillisiä tiloja (intermediate states), joissa ohjelmistokirjasto asiakkaan palvelupyyntöä toteuttaessaan käy. Takaisinkutsujen yhteydessä tilanne on kuitenkin toinen, sillä tällöin ohjelmistokirjaston välillinen tila saattaa olla asiakkaan havaittavissa. Ohjelmistokirjaston tuleekin olla 'oikeellisessa' (valid) tilassa, ainakin siltä osin kuin tila on asiakkaan havaittavissa, ennen takaisinkutsun tekemistä. Ohjelmistokirjaston tila voi myös muuttua takaisinkutsun ollessa vielä aktiivisena, mikäli ohjelmistokirjasto saa uusia palvelupyyntöjä takaisinkutsun aikana. Ohjelmistokirjaston havaittavan tilan tuleekin säilyä 'oikeellisena' niin kauan kuin sen tekemät takaisinkutsut ovat aktiivisia. (Szyperski, 1997)

Takaisinkutsujen problematiikan voidaan ajatella pätevän myös ohjelmistokomponenttien yhteydessä. Ohjelmistokomponentin tilan tulee myös säilyä 'oikeellisena' aina, kun sen tekemät takaisinkutsut ovat aktiivisia. KUVASSA 6 on esitetty sekvenssikaavion (sequence diagram, Booch ym., 1998) avulla Jaaksin (1996b) esimerkkiin perustuen, kuinka moottori tekee takaisinkutsun eli kutsuu auton `Varoita`-metodia. Moottorin tilan tulee pysyä 'oikeellisena' `Varoita`-metodin kutsumisesta asti siihen saakka kunnes kutsu palaa. KUVAN 6 kutsujärjestys ja kuvitteelliset tapahtumat etenevät seuraavasti. Auto kutsuu ensin moottorin `Käynnistä`-metodia, minkä jälkeen moottori on käynnissä. Jonkin ajan kuluttua moottorin öljynpaine laskee alle sallitun rajan. Sen vuoksi moottori joutuu varoittamaan autoa liian alhaisesta öljynpaineesta eli kutsumaan auton `Varoita`-metodia. Tällöin auton on mahdollista esimerkiksi varoittaa auton käyttäjää moottorin liian alhaisesta öljynpaineesta. Edellä kuvattu on vain yksi mahdollinen tapahtumasarja. Yhtä hyvin moottori voidaan sammuttaa ennen kuin sen öljynpaine ehtii laskea alle sallitun rajan. Tällöin moottorin tekemä takaisinkutsu jää suorittamatta.



KUVA 6. Moottori tekee takaisinkutsun eli kutsuu auton `Varoita`-metodia

Szyperski (1997) ei ota kantaa siihen, kuinka ohjelmistokomponentti voi saada uusia palvelupyyntöjä takaisinkutsun aikana. Mahdollisuuksia on ainakin kaksi. Ensinnäkin, takaisinkutsun kohteena oleva asiakas saattaa kutsua ohjelmistokomponenttia takaisinkutsun perusteella, sillä kontrolli on tällöin kyseisellä asi-

akkaalla. Tällainen takaisinkutsusta aiheutuva palvelupyyntö voi olla esimerkiksi ohjelmistokomponentin tilan tiedustelu. Toinen mahdollisuus on se, että jokin toinen ohjelmistokomponentin asiakas kutsuu ohjelmistokomponenttia takaisinkutsun aikana. Tällöin ohjelmistokomponentin tulee kuitenkin olla sellainen, että se pystyy palvelemaan useita samanaikaisia asiakkaita. Samanaikaisten asiakkaiden tapauksessa täytyy huomata, että ne voivat aiheuttaa ongelmia myös tavallisten kutsujen yhteydessä, sillä ohjelmistokomponentti voi saada palvelupyynnön, kun se on vielä suorittamassa aikaisemman asiakkaan palvelupyyntöä. Szyperski (1997) käyttää tällaisesta tilanteesta termiä *uudelleenkäynnistettävyys* (re-entrance). Tämä aiheuttaa samanlaisia ongelmia ohjelmistokomponentin tilan suhteen kuin takaisinkutsujen tapauksessa.

Ulkoiset kutsut eivät yleensä aiheuta samanlaisia ongelmia kuin takaisinkutsut, sillä kommunikointi on tällöin tavallisesti yksisuuntaista ts. ulkoisen kutsun kohde ei ole tietoinen ohjelmistokomponentista, joka käyttää sen palveluita. Kuitenkin, mikäli ulkoisen kutsun kohde voi käyttää ohjelmistokomponentin palveluita, on tilanne sama kuin takaisinkutsujen yhteydessä. Tässäkin tapauksessa Büchin ja Weckin (1997) huomio siitä, että ohjelmistokomponentin toiminnan arviointi pelkkien syötteiden ja tulosteiden perusteella ei aina ole riittävää, pitää paikkansa.

3.2.2 Ohjelmistokomponentin laatu

Ohjelmistokomponentin laatu kuuluu myös olennaisena osana sopimuksenmukaisten rajapintojen määrityksiin, vaikka sen merkitystä ei olekaan korostettu ohjemistokomponentin toiminnan tapaan. Esimerkiksi Fischer ja Snelting (1997) sekä Büchi ja Weck (1997) jättävät laatunäkökulman kokonaan huomioimatta kuvatessaan sopimuksenmukaisiin rajapintoihin perustuvaa uudelleenkäyttöä.

Ohjelmistokomponentin laadun kuvaaminen on kuitenkin tärkeää uudelleenkäytön kannalta. Franch (1997) toteaa, että ohjelmistokomponentit, jotka valitaan käytettäviksi uudelleen pelkkien toiminnallisten vaatimusten perusteella, eivät välttämättä täytä käyttöympäristön laadullisia vaatimuksia, jolloin niiden käyttö on hankalaa tai jopa mahdotonta. Szyperski (1997) tarkastelee asiaa uudelleenkäytettävän ohjelmistokomponentin asiakkaiden kannalta. Hänen mukaansa ohjelmistokomponentin rajapinnassa määritellyistä laatuvaatimuksista poikkeaminen voi 'rikkoa' ohjelmistokomponentin asiakkaat yhtä helposti kuin toiminnallisten vaatimusten täyttämättä jättäminen.

Kirjallisuudessa käytetään usein termiä 'ei-toiminnalliset vaatimukset' (non-functional requirements) laatuvaatimusten synonyyminä (ks. esimerkiksi Jaaksi, Aalto, Aalto, Vättö, 1999 tai Szyperski, 1997). Franch (1997) sen sijaan huomauttaa, että kaikki ohjelmiston ei-toiminnalliset piirteet, kuten esimerkiksi käyttöliittymän tyyppi, eivät liity itse ohjelmiston laatuun. Sen vuoksi Franch (1997) tarkastelee mieluummin ohjelmiston ei-toiminnallisia piirteitä kuin laatua. Tässä tutkimuksessa keskitytään tarkastelemaan ohjelmistokomponenttien laatua osana sopimuksenmukaisia rajapintoja. Laatuun liittymättömät ei-toiminnalliset piirteet eivät ole kiinnostuksen kohteena, sillä ne eivät vaikuta laatua kuvaavien piirteiden tapaan ohjelmistokomponentin toiminnallisuudesta saatavaan oikeaan kuvaan.

Franch (1997) toteaa, että ohjelmistokomponenttien ei-toiminnallisia piirteitä voidaan tarkastella joko prosessi- tai tuotekeskeisen lähestymistavan näkökulmasta. Franchin (1997) mukaan prosessikeskeiset lähestymistavat (ks. Mylopoulos, Chung ja Nixon, 1992) näkevät ei-toiminnallisen informaation ohjelmistojen kehittämistä ohjaavana tekijänä, kun taas tuotekeskeiset lähestymistavat pyrkivät määrittelemään ohjelmistokomponentin ei-toiminnalliset piirteet, jotta on mahdollista tutkia, täyttääkö jokin ohjelmistokomponentti tietyt ei-toiminnalliset vaatimukset. Tässä tutkimuksessa ohjelmistokomponentin ei-toiminnal-

lisia, laatua kuvaavia piirteitä tarkastellaan tuotekeskeisen lähestymistavan näkökulmasta.

Ohjelmistokomponentin laatua voidaan arvioida monien eri ominaisuuksien perusteella. Esimerkiksi Franch (1997) mainitsee suorituskyvyn, uudelleenkäytettävyyden ja luotettavuuden ohjelmistokomponentin laatua kuvaavina piirteinä. Szyperski (1997) sen sijaan käsittelee ohjelmistokomponentin laatua yksittäisten operaatioiden vaatimien resurssien, kuten ajan ja muistinkäytön, suhteen. Jaaksin ym. (1999) mukaan ei-toiminnalliset vaatimukset ovat pääasiassa teknisiä vaatimuksia, jotka kertovat kuinka hyvin toiminnalliset vaatimukset tulee suorittaa. Szyperskin (1997) tarkastelemat ohjelmistokomponentin operaatioiden vaatimat resurssit voidaan ajatella Jaaksin ym. (1999) kuvaamina toiminnallisten vaatimusten laatua määrittävinä ei-toiminnallisina vaatimuksina.

Kaikkia ohjelmistokomponentteja ei kuitenkaan voida arvioida samojen laatukriteerien perusteella. Esimerkiksi kaikki ohjelmistokomponentit eivät kykene palvelemaan useita samanaikaisia asiakkaita. Tällöin niiden laatua ei ole mielekästä arvioida sillä perusteella, että kuinka monta samanaikaista asiakasta ohjelmistokomponentilla voi olla tai kuinka samanaikaisten asiakkaiden määrän lisääntyminen vaikuttaa ohjelmistokomponentin käyttäytymiseen.

Szyperskin (1997) mukaan ohjelmistokomponentin operaatioiden vaatimien aika- ja muistiresurssien määrittäminen nostaa huomattavasti sopimuksenmukaisten rajapintojen käyttöarvoa. Tällöin voidaan arvioida, täyttääkö jokin ohjelmistokomponentti tietyt laatuvaatimukset. Toisaalta, ohjelmistokomponentin uudelleenkäytettävyyden ja luotettavuuden sisällyttäminen osaksi sopimuksenmukaisia rajapintoja ei ole hyvä ajatus, sillä niiden sopimuksenmukaisuus on vaikeasti käsitettävä asia. Uudelleenkäytettävyys, luotettavuus sekä muut ohjelmistokomponentin laatua kuvaavat ominaisuudet, joiden sisällyttäminen osaksi ohjelmistokomponentin sopimuksenmukaisia rajapintoja ei

tunnu mielekkäältä, kannattaa kuitenkin liittää osaksi ohjelmistokomponentin muuta dokumentaatiota. Usein tällaisten ominaisuuksien dokumentaatioon voidaan liittää myös ohjelmistonkehitysprosessin aikana tuotettuja dokumentteja. Esimerkiksi ohjelmistokomponentin testausvaiheen aikana syntyneiden dokumenttien avulla ohjelmistokomponentin voidaan osoittaa täyttävän tietyt luotettavuuden kriteerit.

3.3 Yhteenveto

Tässä kappaleessa esitettiin, mitä sopimuksenmukaisilla rajapinnoilla tarkoitetaan sekä mitä ohjelmistokomponenttien piirteitä sopimuksenmukaisten rajapintojen tulee kuvata ja miksi. Sopimuksenmukaisilla rajapinnoilla tarkoitetaan sitä, että rajapinnat voidaan nähdä ohjelmistokomponentin asiakkaan ja toteuttajan välisinä sopimuksina. Tällöin ohjelmistokomponentin asiakkaan tulee voida luottaa siihen, että ohjelmistokomponentti toimii sopimuksessa määritellyllä tavalla. Toisaalta sopimus edellyttää asiakkaan täyttävän tietyt esiehdot, jotta sopimus olisi voimassa.

Ohjelmistokomponenttien rajapintojen tulee tarjota vain tarpeellinen tieto, joka on edellytyksenä ohjelmistokomponentin uudelleenkäytölle. Tämä edistää osaltaan ohjelmistokomponentin korvaamista toisella. Olennaista korvattavuuden kannalta on se, että ohjelmistokomponentin rajapinnan uusi toteutus toimii ulkoisesti samalla tavalla kuin alkuperäinen eli täsmää semanttisesti alkuperäisen kanssa.

Sopimuksenmukaisten rajapintojen tulee antaa oikea kuva ohjelmistokomponentin toiminnallisuudesta, jonka voidaan ymmärtää muodostuvan ohjelmistokomponentin toiminnasta ja toiminnan laadusta. Ohjelmistokomponentin itsenäistä toimintaa voidaan tarkastella esi- ja jälkiehtojen avulla. Tilanne kuitenkin muuttuu, kun huomioidaan ohjelmistokomponenttien välinen vuorovaiku-

tus, jolla tarkoitetaan ohjelmistokomponentin tekemiä ulkoisia – ja takaisinkutsuja. Tällöin esi- ja jälkiehtoja ei voida pitää riittävinä ohjelmistokomponentin toiminnan kuvaamisessa.

Ohjelmistokomponentin laadun kuvaaminen on tärkeää, sillä pelkkä toiminnan kuvaus ei anna takeita siitä, täyttääkö ohjelmistokomponentti käyttöympäristön asettamat laadulliset vaatimukset. Ohjelmistokomponentin laadulla voidaan tarkoittaa monia asioita, kuten esimerkiksi ohjelmistokomponentin luotettavuutta, uudelleenkäytettävyyttä tai ohjelmistokomponentin tarjoamien palveluiden vaatimia resursseja. Sopimuksenmukaisisten rajapintojen kuvauksissa on syytä ottaa kantaa jälkimmäiseen. Sen sijaan muut, vaikeasti sopimuksenmukaisina ymmärrettävät asiat, kannattaa liittää osaksi ohjelmistokomponentin muuta dokumentaatiota.

4 OHJELMISTOJEN DOKUMENTOINTI

Tässä kappaleessa käsitellään yleisesti ohjelmistojen dokumentointia, jonka yhdeksi osa-alueeksi voidaan laskea ohjelmistokomponenttien dokumentointi. Kappaleessa kuvataan, mihin eri kategorioihin ohjelmistodokumentaatio jakautuu, ja selvitetään lyhyesti näiden kategorioiden sisältöä. Tämän tarkoituksena on tarjota riittävä yleiskuva ohjelmistojen dokumentoinnista, sekä esittää uudelleenkäytön dokumentaation tarve ohjelmistokomponenttien yhteydessä. Lisäksi tarkastellaan hyvän ohjelmistodokumentaation kriteereitä, sekä pohditaan, kuinka hyvin ne soveltuvat ohjelmistokomponentin dokumentaation arvioimiseen. Syynä tällaisen tarkastelun tekemiselle on tarve tutkia dokumentaation sisällön ohella dokumentaation laatua kuvaavia piirteitä.

Spraguen (1995, s. 32) määritelmän mukaan dokumentti on joukko tiettyyn aiheeseen liittyvää informaatiota, joka on jäsennetty ihmisen käsityskykyä vastaavaksi, esitetty erityyppisillä symboleilla, sekä tallennettu ja käsitelty yhtenä yksikkönä. Tämä määritelmä sopii yleisluonteisuutensa vuoksi käytettäväksi myös tässä tutkimuksessa. Se ei esimerkiksi ota kantaa siihen, esitetäänkö dokumentti elektronisessa muodossa vai ei. Erityyppiset symbolit viittaavat siihen, että dokumentti voi koostua esimerkiksi tekstistä, kuvista ja taulukoista. Ohjelmisto- ja myös muun dokumentaation voidaan ajatella muodostuvan yhdestä tai useammasta edellä mainitun määritelmän kaltaisesta dokumentista.

4.1 Ohjelmistodokumentaation kategoriat

Ohjelmistodokumentaatio jaetaan yleensä käyttäjille suunnattuun, järjestelmä- ja prosessidokumentaatioon (ks. Sametinger, 1994 tai Pomberger, 1984). Mikään edellämainituista kategorioista ei kuitenkaan sisällä tietoja, joiden tuntemista voidaan pitää edellytyksenä ohjelmistokomponenttien uudelleenkäytölle. Sen

vuoksi yhdeksi ohjelmistodokumentaation kategoriaksi on ehdotettu uudelleenkäytön dokumentaatiota (ks. Sametinger, 1997 tai Braun, 1994), joka sisältää ohjelmistokomponenttien uudelleenkäytön kannalta tarpeelliset tiedot. Seuraavaksi käydään läpi kaikki edellä mainitut ohjelmistodokumentaation kategoriat.

4.1.1 Käyttäjille suunnattu dokumentaatio

Käyttäjille suunnatun dokumentaation (user documentation) tarkoituksena on opastaa eritasoisia loppukäyttäjiä järjestelmän tehokkaassa käyttämisessä ja hallinnoimisessa. Dokumentaation tulee olla riittävä näihin tarkoituksiin, eli järjestelmän käytön ja hallinnoinnin tulee onnistua ilman käyttäjän opastusta tai mahdollista lisätietoa. (Sametinger, 1997)

Pombergerin (1984) mukaan käyttäjille suunnattu dokumentaatio koostuu kolmesta osasta, jotka ovat

- järjestelmän yleinen kuvaus (general system description),
- asennus- ja käyttöopas (installation and user manual), ja
- järjestelmän ylläpitäjän opas (operator manual).

Loppukäyttäjät eivät yleensä ole suoranaisesti tekemisissä ohjelmistokomponenttien kanssa, joten ohjelmistokomponentit eivät tavallisesti ole käyttäjille suunnatun dokumentaation piirissä. Sametinger (1997) toteaa, että poikkeuksen edellä mainittuun muodostavat loppukäyttäjien kanssa vuorovaikutuksessa toimivat ohjelmistokomponentit. Tällöin käyttäjille suunnattu dokumentaatio saattaa koostua osittain yksittäisten ohjelmistokomponenttien dokumentaatioista. Kokonaisuuden tulee kuitenkin olla eheä loppukäyttäjän näkökulmasta katsottuna.

Carroll ja Rosson (1987) pitävät merkittävänä käyttäjille suunnattuun dokumentaatioon liittyvänä ongelmana sitä, että aikuiset suoranaisesti vastustavat

uusien asioiden oppimista. Uuden järjestelmän käytön oppiminen ei tuota poikkeusta. Carrollin (1990) mukaan aikuiset oppijat (adult learners)

- ovat kärsimättömiä oppijoita ja haluavat päästä nopeasti tekemään jotain tuottavaa,
- harvoin lukevat dokumentaatiota läpikotaisin,
- tekevät virheitä, mutta oppivat usein virheistään,
- pysyvät parhaiten motivoituneina saadessaan tutkia asioita oma-aloitteisesti, ja
- lannistuvat laajoista oppaista, joissa jokainen tehtävä on jaettu yksityiskohtaisiin osatehtäviin.

Käyttäjille suunnatusta dokumentaatiosta ei ole hyötyä, jos käyttäjät eivät suostu sitä lukemaan. Minimalistinen lähestymistapa (ks. Brockmann, 1990 tai Carroll, 1990) tarjoaa vaihtoehdon perinteiselle käyttäjille suunnatulle dokumentaatiolle, sillä sen pääajatuksena on tekemällä oppiminen (learning by doing) lukemalla oppimisen (learning by reading) sijaan. Carroll (1990) toteaa, että minimalistinen lähestymistapa ei sisällä varsinaisia ohjeita minimalistisen dokumentaation tekoon. Seuraavassa kuitenkin joitakin Carrollin (1990) mainitsemia asioita, jotka yleensä liittyvät minimalistisen lähestymistavan soveltamiseen dokumentoinnin yhteydessä:

- Kaikki toissijaiset asiat, kuten johdannot ja yhteenvedot, jätetään oppaista ja dokumenteista pois.
- Dokumentaation suunnittelussa keskitytään siihen, mitkä tiedot ovat oleellisia lukijoiden tekemän tuottavan työn kannalta.
- Lukijoita rohkaistaan aktiiviseen järjestelmän tutkimiseen tarjoamalla heille tarkoituksellisesti epätäydellisiä tietoja.

Brockmann (1990) toteaa, että minimalistisen lähestymistavan soveltamiseen liittyy myös monia ongelmia. Nämä ongelmat liittyvät dokumentaation suunnitteluun sekä dokumentaation perusteella tapahtuvaan oppimiseen. Jälkimmäiseen voidaan laskea seuraavat Brockmannin (1990) mainitsemat ongelmat:

- Oma-aloitteisen, järjestelmän tutkimiseen perustuvan oppimisen tulokset ovat vaikeasti ennustettavissa.
- Tekemällä oppiminen edellyttää motivoitunutta kohderyhmää.

Loppukäyttäjien kielteinen suhtautuminen dokumentaatioon on ollut lähtökohtana myös Wilkinin ja Wulffin (1990) esittämälle lähestymistavalle, jonka mukaan dokumentaatio on olennainen osa itse tuotetta (product-as-document approach). Perinteisen käsityksen mukaan dokumentaatio nähdään erillisenä tuotteesta, jota se kuvaa (document-as-product approach, Wilkin ja Wulff, 1990). Esimerkkejä tuotteeseen integroidusta dokumentaatiosta löytyy monelta eri teollisuuden alalta. Wilkin ja Wulff (1990) toteavat, että pankki- ja vakuutustoimialalla suurin osa saatavilla olevista 'tuotteista' on dokumentteja, kuten esimerkiksi lainahakemukset ja vakuutuskaavakkeet, joiden täyttäminen edellyttää niiden lukemista. Tällöin dokumentaation voidaan katsoa kuuluvan olennaisena osana itse tuotteeseen. Toinen selkeä Wilkinin ja Wulffin (1990) mainitsema esimerkki tuotteeseen integroidusta dokumentaatiosta on tuotteet, joilla on interaktiivinen käyttöliittymä. Pankkiautomaattien voidaan ajatella kuuluvan tähän kategoriaan. Esimerkiksi rahan nostamiseen ei tarvita erillistä dokumentaatiota, sillä pankkiautomaatti ohjaa tapahtumaa antamalla käyttäjälle ohjeita ja pyytämällä käyttäjää toimimaan tilanteen edellyttämällä tavalla.

4.1.2 Järjestelmädokumentaatio

Järjestelmädokumentaatio (system documentation) sisältää kaikki järjestelmän kehittämiseen liittyvän tiedon. Sen tulee sisältää riittävästi tietoa, jotta uusi kehittämis- tai ylläpitoryhmän jäsen kykenee muuttamaan tai laajentamaan järjestelmää ilman muiden, jo aikaisemmin järjestelmän kehittämis- tai ylläpitotehtävissä toimineiden henkilöiden apua. (Sametinger, 1997)

Pombergerin (1984) ja Sommervillen (1995) mukaan järjestelmädokumentaation tulee sisältää seuraavat järjestelmää kuvaavat tiedot:

- vaatimukset (requirements),
- kokonaissuunnitelma ja rakenne (overall design and structure),
- toteutuksen yksityiskohdat (implementation details),
- testisuunnitelmat ja raportit (test plans and reports), ja
- lähdekoodi (source code listings).

Sametinger (1997) toteaa, että lähdekoodi on usein ainoa ajantasainen ja kattava kuvaus järjestelmästä. Lähdekoodi ei kuitenkaan tarjoa riittäviä tietoja järjestelmän ylläpitämiseksi. Esimerkiksi järjestelmän suunnittelun yhteydessä tehdyt suunnittelupäätökset tai järjestelmän arkkitehtuuri eivät ole johdettavissa lähdekoodista. Lähdekoodin perusteella on myös vaikea saada riittävää kokonaiskuvaa järjestelmästä. Sametingerin (1997) mukaan ajalliset paineet aiheuttavat yleensä sen, että järjestelmädokumentaatio ei ole lähdekoodia lukuunottamatta ajan tasalla. Muitakin syitä dokumentoinnin laiminlyömiseen varmasti on. Järjestelmänkehittäjät saattavat esimerkiksi kokea dokumentoinnin ylimääräisenä työnä eivätkä varsinaisena järjestelmän kehittämiseen liittyvänä tehtävänä. Tällöin järjestelmädokumentaation päivittäminen saattaa helposti jäädä tekemättä. Järjestelmän ylläpidon kannalta on kuitenkin tärkeää, että järjestelmädokumentaatio on ajan tasalla. Pressman (1994) esimerkiksi toteaa, että dokumentaatio, joka ei vastaa järjestelmän nykyistä tilaa, on huonompi vaihtoehto kuin se, että dokumentaatiota ei ole ollenkaan. Pressmanin (1994) mukaan tämä johtuu siitä, että dokumentaatio, joka ei ole ajan tasalla, antaa virheellisen kuvan järjestelmästä ja voi siten aiheuttaa merkittäviä ongelmia.

Järjestelmädokumentaatiota on usein syytä päivittää järjestelmän toteutuksen jälkeen vastaamaan järjestelmän todellista tilannetta. Dokumentaation päivittämisen yhteydessä voidaan joissakin tapauksissa puhua *käänteisestä suunnittelusta* (reverse engineering), jolla tarkoitetaan järjestelmän analysointiprosessia, jonka tarkoituksena on tuottaa järjestelmästä korkeamman tason kuvaus kuin

mitä lähdekoodi edustaa (ks. Pressman, 1994 tai Sommerville, 1995). Järjestelmän analysointia ei tarvita, mikäli järjestelmän kehittäjät osaavat suoraan päivittää järjestelmädokumentaation tarpeellisilta osilta vastaamaan järjestelmän nykyistä tilannetta. Esimerkki tällaisesta päivityksestä on toteutuksen yhteydessä suunnitelmista tehtyjen poikkeuksien huomioiminen järjestelmädokumentaatiossa.

4.1.3 Prosessidokumentaatio

Prosessidokumentaatio (process documentation) eroaa käyttäjille suunnatusta - ja järjestelmädokumentaatiosta siinä, että se ei kuvaa itse tuotetta eli ohjelmistoa, vaan prosessia, jossa ohjelmisto on kehitetty. Prosessidokumentaation tarkoituksena on tukea projektin johtamista ja hallintaa. Sen avulla voidaan myös suunnitella tulevia ohjelmistonkehitysprojekteja ja -prosesseja. Prosessidokumentaatiota ei pidetä käyttäjille suunnatun - ja järjestelmädokumentaation tavoin ajan tasalla. (Sametinger, 1997)

Pombergerin (1984) mukaan prosessidokumentaatioon kuuluvat

- projektisuunnitelma (project plan),
- organisointisuunnitelma (organization plan),
- resurssisuunnitelma (resource plan),
- projektin standardit (project standards),
- projektin aikana syntyneet dokumentit (working papers),
- projektin jäsenten välinen kommunikointi (log book) , ja
- lukemista helpottavat dokumentit (reading aids).

Projektin aikana syntyneneiden dokumenttien tarkoituksena on tallentaa ideat, strategiat ja tunnistetut ongelmat, jotta tehdyille suunnittelupäätöksille voidaan tarvittaessa löytää perusteet (Pomberger, 1984). Tällaisesta tiedosta on hyötyä esimerkiksi silloin, kun jossakin toisessa projektissa on ratkaistavana saman-

tyyppinen ongelma. Tällöin voidaan tarkistaa, mitä aikaisemmin on tehty vastaavanlaisessa tilanteessa ja millä perusteella.

Edellä esitetty Pombergerin (1984) lista prosessidokumentaatioon kuuluvista dokumenteista ei ole kattava. Esimerkiksi Blokdijk ja Blokdijk (1987) esittävät, että projektissa tulee tuottaa 17 erilaista suunnitelmaa. Tämän lisäksi Blokdijk ja Blokdijk (1987) ehdottavat, että itse suunnitelmien tekemistä varten tarvitaan erillinen suunnitelma (planning plan), josta tulee ilmetä esimerkiksi se, milloin tietyn suunnitelman tekemisen voi aloittaa. Kaikkiaan projektissa tulee näin ollen tuottaa 18 suunnitelmaa. Projektissa tuotettavat suunnitelmat ja muut dokumentit sekä niiden laajuus kannattaa kuitenkin mitoittaa kulloisenkin projektin tarpeita vastaaviksi, sillä usein suunnitelmia joudutaan niiden tekemisen lisäksi myös päivittämään projektin edetessä.

4.1.4 Uudelleenkäytön dokumentaatio

Braunin (1994) mukaan dokumentaatio muodostaa merkittävän osan ohjelmistokomponentin uudelleenkäyttöarvosta (reuse value), sillä dokumentaatio toimii perinteisen roolinsa lisäksi eksplisiittisenä oppaana ohjelmistonkehittäjille, joiden tarkoituksena on käyttää ohjelmistokomponenttia uudelleen. Eksplisiittisellä oppaalla Braun (1994) tarkoittaa nimenomaan ohjelmistokomponentin uudelleenkäytön dokumentaatiota. Dokumentaation perinteistä roolia Braun (1994) ei kuvaile tarkemmin, mutta sen voidaan ajatella viittaavan esimerkiksi järjestelmädokumentaatioon, joka on osa perinteisten ohjelmistojen dokumentaatiota, mutta jonka voidaan katsoa kuuluvan yhtä hyvin myös osaksi ohjelmistokomponentin dokumentaatiota.

Ohjelmistokomponentin uudelleenkäytön dokumentaation kohderyhmänä on ohjelmistonkehittäjät, joiden tulee voida päättää, sopiiko tietty ohjelmistokomponentti heidän tarpeisiinsa, ja jotka tarvitsevat kattavasti tietoa ohjelmistokom-

ponentin rajapinnoista (Sametinger, 1997). Ohjelmistokomponentin käyttäjät eroavatkin tavallisista loppukäyttäjistä, jotka muodostavat käyttäjille suunnatun dokumentaation kohderyhmän.

Karlsson (1995) esittää Braunin (1994) tavoin, että ohjelmistokomponentin dokumentaatio tulee jakaa kahteen osaan, jotka ovat

- uudelleenkäytön dokumentaatio, joka tukee ohjelmistokomponentin uudelleenkäyttöä, sekä
- muut ohjelmistokomponenttia kuvaavat tiedot, jotka kuuluvat sen tuotteen, jonka osana ohjelmistokomponentti on, dokumentaation piiriin.

Tässä yhteydessä tutustutaan ohjelmistokomponentin uudelleenkäytön dokumentaatioon. Muu ohjelmistokomponentin dokumentaatio kattaa ainakin ohjelmistokomponentin järjestelmädokumentaation eli ohjelmistokomponentin kehittämiseen ja ylläpitämiseen liittyvät tiedot. Sametinger (1997) tosin toteaa, että järjestelmädokumentaatio ei ole merkittävä ohjelmistokomponentin uudelleenkäytön kannalta, mikäli ohjelmistokomponentti on käytettävissä uudelleen sellaisenaan (reused as black box), mutta liittää sen kuitenkin osaksi ohjelmistokomponentin uudelleenkäytön dokumentaatiota. Tämän tutkimuksen kohteena olevat ohjelmistokomponentit ovat käytettävissä uudelleen sellaisinaan, joten ohjelmistokomponentin järjestelmädokumentaatio ei tässä tapauksessa ole osa ohjelmistokomponentin uudelleenkäytön dokumentaatiota. Myös käyttäjille suunnattu dokumentaatio voi olla tietyiltä osin kohdistettavissa yhdelle ohjelmistokomponentille, mutta prosessidokumentaatiota on jo vaikeampi osittaa yksittäiselle ohjelmistokomponentille, paitsi siinä erityistapauksessa, että ohjelmistokomponentin tuottamista varten on perustettu oma projektinsa. Tämä lienee kuitenkin harvinaista.

Sekä Braun (1994) että Sametinger (1997) toteavat, että ohjelmistokomponentin dokumentaation tulee ohjelmistokomponentin tavoin olla itseriittoinen. Tällä he tarkoittavat sitä, että kullekin ohjelmistokomponentille tulee olla oma doku-

mentaationsa, jolla on mahdollisimman vähän riippuvuuksia muuhun dokumentaatioon. Erityisen tärkeää dokumentaation itseriittoisuus on uudelleenkäytön dokumentaation osalta, koska se on muuta ohjelmistokomponentin dokumentaatiota, kuten esimerkiksi suunnittelu- ja testausdokumentaatiota, useammin käytössä.

Ohjelmistokomponentin voidaan ajatella koostuvan uudelleenkäytettävästä ja uudelleenkäytön mahdollistavasta osasta (Forsell ja Päivärinta, 2001). Tämä ajatus on eksplikoitu

- Braunin (1994) ja Sametingerin (1997) huomiosta, jonka mukaan kullekin ohjelmistokomponentille tulee olla oma dokumentaationsa,
- Braunin (1994) ajatuksesta, jonka mukaan ohjelmistokomponentin dokumentaation tulee ohjelmistokomponentin tavoin olla uudelleenkäytettävä, sekä
- Braunin (1994) ja Karlssonin (1995) esittämästä ohjelmistokomponentin dokumentaation jaottelusta kahteen osaan.

Forsellin ja Päivärinnan (2001) mukaan ohjelmistokomponentin uudelleenkäytettävä osa syntyy ohjelmistokomponentin tuottamisen yhteydessä, ja sen dokumentaation on täytettävä samat vaatimukset, kuin ohjelmistotuotantoprosessin eri vaiheissa syntyneiden tulosten. Uudelleenkäytettävän osan dokumentaatiolla he siis tarkoittavat ainakin ohjelmistokomponentin järjestelmädokumentaatiota. Lisäksi itse ohjelmistokomponentti kuulunee tähän osaan. Forsell ja Päivärinta (2001) toteavat, että uudelleenkäytön mahdollistavasta osasta on löydyttävä ohjelmistokomponentin välittämistä, käyttämistä ja hallinnointia tukeva dokumentaatio. Tämän voidaan ajatella tarkoittavan ainakin ohjelmistokomponentin uudelleenkäytön dokumentaatiota.

Sametingerin (1997) mukaan useat kirjoittajat, kuten Braun (1994), Karlsson (1995), Krueger (1992) sekä Meyer (1994), ovat kuvanneet, mitkä tiedot ovat edellytyksenä ohjelmistokomponenttien uudelleenkäytölle. Näitä kuvauksia ei

kuitenkaan ole tarkoituksenmukaista esittää yksityiskohtaisesti, sillä ne sisältävät osittain samoja asioita. Seuraavaksi kuvataan Sametingerin (1996) esittämä ohjelmistokomponentin uudelleenkäytön opas (reuse manual), joka jakaantuu seuraaviin osiin:

- yleinen informaatio (general information),
- uudelleenkäytön informaatio (reuse information),
- hallinnollinen informaatio (administrative information),
- arviointia tukeva informaatio (evaluation information) , ja
- muu informaatio (other information).

Yleisen informaation tarkoituksena on antaa ohjelmistokomponentista riittävä yleiskuva, jonka perusteella ohjelmistonkehittäjät voivat arvioida ohjelmistokomponentin sopivuutta heidän tarkoituksiinsa. Yleiskuvan perusteella ei yleensä kuitenkaan voida tehdä lopullista päätöstä ohjelmistokomponentin käytöstä tietyssä yhteydessä, mutta sen avulla voidaan rajata tarkasteltavien ohjelmistokomponenttien joukkoa. Yksityiskohtaisempi, muuhun kuin yleiseen informaatioon perustuva, tarkastelu voidaan sitten tehdä tämän joukon sisältämille ohjelmistokomponenteille. Yleinen informaatio koostuu seuraavista:

- Johdanto (introduction): selkeä ja tiivis kuvaus, joka kattaa ohjelmistokomponentin nimen, tunnisteen ja yleiskuvauksen.
- Luokitus (classification): ohjelmistokomponentin luokittelemiseen tarvittavat tiedot, kuten esimerkiksi ohjelmistokomponentin toiminnallisuus tai soveltuvat alustat, jotka esiintyvät muualla uudelleenkäytön oppaassa.
- Toiminnallisuus (functionality): ohjelmistokomponentin yleinen toiminnallisuus, joka käsittää mm. kaikki ulkoiset operaatiot.
- Alustat (platforms): alustat, kuten esimerkiksi C++ tai OpenDoc, joilla ohjelmistokomponentti soveltuu käytettäväksi.
- Uudelleenkäytön tila (reuse status): eri tekijöiden, kuten esimerkiksi testauksen ja ylläpidon, suhteen määritelty ohjelmistokomponentin laatuarvo.

Uudelleenkäytön informaatio sisältää olennaiset tiedot varsinaisen uudelleenkäytön kannalta. Näitä tietoja tarvitaan lopullisen päätöksen tekemiseen ohjelmistokomponentin uudelleenkäytöstä, sekä silloin, kun ohjelmistokomponentti liitetään osaksi muuta ohjelmistoa. Uudelleenkäytön informaation tulee sisältää yksityiskohtaiset tiedot ohjelmistokomponentin asennusta, uudelleenkäyttöä ja muokkaamista varten:

- Asennus (installation): vaiheet, jotka tarvitaan ohjelmistokomponentin liittämiseksi osaksi muuta järjestelmää.
- Rajapintojen kuvaukset (interface descriptions): yksityiskohtaiset rajapintojen määritykset, jotka kattavat kaiken ohjelmistokomponentin toiminnallisuuden.
- Integrointi ja käyttö (integration and usage): ohjelmistokomponentin oikeaoppisen uudelleenkäytön kannalta välttämättömät tiedot, kuten rajapinnat, malliskenaariot ja diagnosoivat menettelytavat, jotka kertovat mitä tehdä ongelmatilanteiden yhteydessä.
- Muokkaus (adaptation): yksityiskohtaiset tiedot siitä, kuinka ohjelmistokomponentti voidaan mukauttaa tiettyjä tarpeita varten.

Hallinnollinen informaatio sisältää ohjelmistokomponentin hankkimisen ja käytön kannalta tarpeelliset tiedot, mutta hiukan eri näkökulmasta kuin esimerkiksi uudelleenkäytön informaation yhteydessä:

- Hankinta ja tuki (procurement and support): yhteystiedot, joista selviää mistä voidaan pyytää apua esimerkiksi ohjelmistokomponentin muokkaamisen yhteydessä, sekä saatavuuteen ja omistajuuteen liittyvät tiedot.
- Kaupalliset ja lailliset rajoitukset (commercial and legal restrictions): ohjelmistokomponentin käyttöön liittyvät kaupalliset ja lailliset rajoitukset, kuten esimerkiksi lisenssit tai muut luvat.
- Historia ja versiot (history and versions): ohjelmistokomponentin versiohistoria, joka sisältää tiedot kunkin version tekijöistä ja julkaisupäivistä, sekä pääasialliset eroavaisuudet eri versioiden välillä.

Arviointia tukevan informaation tarkoituksena on tarjota ohjelmistokomponentista yksityiskohtainen kuvaus, jonka perusteella voidaan päättää, soveltuuko ohjelmistokomponentti todella käytettäväksi tietyssä yhteydessä. Arviointia tukeva informaatio antaa ohjelmistokomponentista tarkemman kuvauksen kuin yleinen informaatio, jonka avulla voidaan rajata mahdollisesti sopivien ohjelmistokomponenttien joukkoa. Tämä yksityiskohtaisempi kuvaus koostuu seuraavista:

- Määritys (specification): ohjelmistokomponentin toiminnallisuuden yksityiskohtainen määrittely, johon sisältyvät toiminnalliset ja ei-toiminnalliset vaatimukset.
- Laatu (quality): tiedot, joiden perusteella voidaan tehdä päätelmiä ohjelmistokomponentin laadusta, kuten sovelletut testit ja niiden tulokset.
- Suorituskyky- ja resurssivaatimukset (performance and resource requirements): tiedot ohjelmistokomponentin vaatimista järjestelmäresursseista, kuten muistin ja prosessoriajan käytöstä, sekä suorituskykyä kuvaavista piirteistä.
- Vaihtoehtoiset ohjelmistokomponentit (alternative components): tiedot samankaltaisista ohjelmistokomponenteista, jotka soveltuvat käytettäviksi dokumentaation kohteena olevan ohjelmistokomponentin sijaan.
- Tiedostetut viat tai ongelmat (known bugs/problems): raportit ratkaisemattomista ongelmista, kuten havaituista vioista tai halutuista parannuksista.
- Rajoitteet ja rajoitukset (limitations and restrictions): ohjelmistokomponentin käyttöön liittyvät tekniset rajoitteet ja rajoitukset, kuten riippuvuudet tietystä ohjelmointikielestä tai käyttöjärjestelmästä.
- Mahdolliset parannukset (possible enhancements): tiedot mahdollisista parannuskohteista, jotka voivat liittyä esimerkiksi ohjemistokomponentin ylläpidettävyyteen tai uudelleenkäytön laaja-alaisuuteen.
- Testaustuki (test support): tiedot ohjelmistokomponenttiin liittyvistä testitapauksista ja testausympäristöstä.

- Riippuvuussuhteet (interdependencies): tiedot siitä, voidaanko ohjelmistokomponenttia käyttää itsenäisesti vai vaatiko sen käyttö myös muiden ohjelmistokomponenttien käyttöönottoa.

Muu informaatio kattaa kaikki muut ohjelmistokomponentin uudelleenkäyttöön liittyvät tiedot, jotka eivät sisälly edellä lueteltuihin uudelleenkäytön oppaan osa-alueisiin:

- Järjestelmädokumentaatio (system documentation): mikäli ohjelmistokomponentti ei ole käytettävissä uudelleen sellaisenaan, saattaa olla hyödyllistä liittää mukaan ohjelmistokomponentin toteutuksen tiedot, jotka kattavat ohjelmistokomponentin kehittämisen eri vaiheiden, kuten vaatimusmäärittelyn, suunnittelun ja toteutuksen, dokumentit.
- Viitteet (references): viitteet kirjallisuuteen tai muuhun dokumentaatioon, jotka sisältävät hyödyllistä tietoa ohjelmistokomponentin uudelleenkäyttöä ajatellen.
- Lukemista helpottavat dokumentit (reading aids): hakemisto, sisällysluettelo sekä listat kuvista ja taulukoista, jotka helpottavat laajan dokumentaation läpikäymistä.

Sametingerin (1997) kuvaama ohjelmistokomponentin uudelleenkäytön opas tarjoaa hyvän perustan uudelleenkäytettävien ohjelmistokomponenttien dokumentoimiselle, sillä se kattaa varsin monia asioita, ja eri näkökulmista tarkasteltuna. Teknisen näkökulman lisäksi mukana on kaupallinen ja hallinnollinen näkökulma, joka on selvimmin havaittavissa hallinnollisen informaation osuudesta. Kaupallisen näkökulman mukanaolo perustuu siihen, että ohjelmistokomponentti on itsenäinen tuote, jonka hankkimiseen liittyy monia tekijöitä, kuten esimerkiksi kaupalliset ja lailliset rajoitukset, joita ei voida pitää teknisinä asioina.

Hyvänä puolena Sametingerin (1997) kuvaamassa ohjelmistokomponentin uudelleenkäytön oppaassa voidaan pitää myös sitä, että se huomioi ohjelmisto-

komponentin uudelleenkäyttöprosessin (ks. Karlsson, 1995) etenemisen. Tämä on havaittavissa siinä, että oppaan yleinen osuus on tarkoitettu tukemaan sopivien ohjelmistokomponenttiehdokkaiden löytämistä kaikkien mahdollisten ohjelmistokomponenttien joukosta, kun taas muut oppaan osuudet sisältävät yksityiskohtaisempia tietoja ohjelmistokomponentista, ja soveltuvat sen vuoksi paremmin tarkasteltaviksi pienemmälle ohjelmistokomponenttijoukolle. Tämän vuoksi esimerkiksi ohjelmistokomponentin toiminnallisuuden kuvaaminen on hajautettuna tarkkuustason mukaan eri osiin ohjelmistokomponentin uudelleenkäytön opasta.

Toisaalta Sametingerin (1997) esittämä ohjelmistokomponentin uudelleenkäytön opas on hiukan hajanainen, sillä monet toisiinsa liittyvät asiat on kuvattu toisistaan erillisinä. Hyvänä esimerkkinä on suorituskyky- ja resurssivaatimusten sekä vuorovaikutussuhteiden erottaminen ohjelmistokomponentin toiminnallisuudesta. Sametinger (1997) kyllä mainitsee arviointia tukevan informaation osuudessa, että ohjelmistokomponentin toiminnallisuuden määrittelyyn liittyvät sekä toiminnalliset että ei-toiminnalliset vaatimukset, mutta ei sitä, mitä ei-toiminnalliset vaatimukset tässä yhteydessä tarkoittavat. Mikäli ei-toiminnallisiin vaatimuksiin sisältyvät myös esimerkiksi operaatioiden vaatimat muistiresurssit, voidaan Sametingerin (1997) esittämää ohjelmistokomponentin uudelleenkäytön opasta pitää toisteista tietoa sisältävänä. Ohjelmistokomponentin ja siihen liittyvän dokumentaation ylläpitämisen kannalta toisteinen tieto ei ole hyvä asia, sillä tällöin on olemassa vaara, että tiedon päivitys tapahtuu vain yhteen osaan dokumentaatiota, jolloin ohjelmistokomponentin dokumentaatiosta muodostuu ajan mittaan sisäisesti ristiriitainen.

4.2 Hyvän ohjelmistodokumentaation kriteerit

D'Souza ja Wills (1998) esittävät hyvän dokumentaation perustuvan taitoon jättää asioita pois; kuitenkin johdonmukaisella ja käytännöllisellä tavalla, jotta do-

kumentaatio on hyvin jäsentynyt, tiivis sekä luettava. Carrollin (1990) mukaan asioiden poisjättäminen on yksi minimalistisen lähestymistavan soveltamisen yhteydessä usein esiintyvä piirre, joten edellä mainitun D'Souzan ja Willsin (1998) huomion voidaan todeta tukevan ajatusta, jonka mukaan soveltamalla minimalistista lähestymistapaa dokumentoinnin yhteydessä voidaan parantaa dokumentaation luettavuutta.

Arthur ja Stevens (1992) toteavat tekemänsä kirjallisuuskatsauksen (US Army, 1984; US Air Force, 1987; Collofello ja Bortman, 1986; Horowitz ja Williamson, 1986a, 1986b; Murine, 1986; Sneed ja Merey, 1985) perusteella, että hyvän dokumentaation ensisijaiset, korkean tason laatua kuvaavat piirteet (high-level qualities) ovat *oikeellisuus* (accuracy), *kattavuus* (completeness), *käytettävyys* (usability) ja *laajennettavuus* (expandability). Tässä tutkimuksessa käytetään Arthurin ja Stevensin (1992) esittämiä piirteitä ohjelmistokomponentin dokumentaation riittävyyden tarkastelussa.

Arthurin ja Stevensin (1992) mukaan edellä mainitut korkean tason piirteet ovat suoraan liitettävissä dokumentaation riittävyyteen (adequate documentation), joka on vielä edellä lueteltuja korkean tason laatua kuvaavia piirteitä abstraktimpi käsite. Arthurin ja Stevensin (1992) tekemän tutkimuksen tarkoituksena on ollut selvittää, mistä asioista riittävä dokumentaatio koostuu, sekä kuinka dokumentaation riittävyyttä voidaan mitata. Näiden asioiden tarkastelemiseksi he ovat tekemänsä kirjallisuuskatsauksen lisäksi esittäneet arviointia tukevan taksonomian, jonka tehtävänä on liittää dokumentaation laatua kuvaavat abstraktit piirteet yksittäisiin tekijöihin (factors), ja nämä tekijät edelleen mitattavissa oleviin yksiköihin (measurable quantifiers). Seuraavaksi kuvataan Arthurin ja Stevensin (1992) esittämät ohjelmistodokumentaation laatua kuvaavat piirteet, ja tarkastellaan niitä ohjelmistokomponentin dokumentaation näkökulmasta.

4.2.1 Oikeellisuus

Arthur ja Stevens (1992) toteavat, että ohjelmistodokumentaation yhteydessä oikeellisuudella tarkoitetaan ohjelmiston lähdekoodin ja lähdekoodia kuvaavan dokumentaation yhteneväisyyttä. Myös Braun (1994) huomauttaa, että dokumentaation tulee säilyä yhdenmukaisena (consistent) lähdekoodin kanssa. Ei siis riitä, että ohjelmistodokumentaatio on hetkellisesti, ohjelmiston julkistamisen hetkellä, ajan tasalla. On voitava olettaa, että ohjelmistodokumentaatio on ohjelmistolle tehtävistä muutoksista huolimatta yhdenmukainen lähdekoodin kanssa. Tämä on tulkittavissa myös Arthurin ja Stevensin (1992) huomiosta, jonka mukaan oikeellisen ohjelmistodokumentaation tulee heijastaa ohjelmiston todellista tilaa. Ohjelmistokomponentin dokumentaation tulee muun ohjelmistodokumentaation tavoin säilyä yhdenmukaisena ohjelmistokomponentin lähdekoodin kanssa. Braun (1994) toteaa tämän olevan tärkeää, sillä ohjelmistokomponentin dokumentaatiota lukevien ohjelmistonkehittäjien, joiden tarkoituksena on käyttää ohjelmistokomponenttia uudelleen, tulee saada oikeelliset tiedot ohjelmistokomponentista. Erityisen tärkeää oikeellisuus on ohjelmistokomponentin uudelleenkäytön dokumentaation osalta, mutta myös muun ohjelmistokomponenttiin liittyvän dokumentaation, erityisesti järjestelmädokumentaation, tulee olla yhdenmukainen ohjelmistokomponentin lähdekoodin kanssa.

Arthur ja Stevens (1992) mainitsevat, että epäjohdonmukaisuudet ohjelmistodokumentaation ja toteutetun järjestelmän välillä voivat ilmetä monella tavalla, kuten esimerkiksi ohjelmointivaiheen aikaisina virheinä tulkittaessa ja toteutettaessa aikaisemmin tehtyjä suunnitelmia. Tällöin ohjelmistodokumentaation epäjohdonmukaisuus ei johdu dokumentoinnin laiminlyömisestä, vaan yksinkertaisesti ihmisen erehtyväisyydestä. Arthurin ja Stevensin (1992) mukaan epäjohdonmukaisuuksia saattaa toisaalta syntyä myös ohjelmiston päivittämisen yhteydessä, mikäli ohjelmistodokumentaation päivittämisestä ei huolehdita vastaavilta osin. Tässä tapauksessa kyse on selkeästi dokumentoinnin laimin-

lyömisestä. Dokumentaation ja lähdekoodin epäyhteneväisyyden seurauksena ohjelmistokomponentin uudelleenkäyttö muodostuu hankalaksi, sillä dokumentaatio ei enää tarjoa kaikilta osin oikeellisia tietoja ohjelmistokomponentista.

4.2.2 Kattavuus

Arthurin ja Stevensin (1992) mukaan ohjelmistodokumentaation voidaan todeta olevan kattava, mikäli se sisältää kaiken tarvittavan informaation. Esimerkiksi Sametingerin (1997) ja Braunin (1994) kuvaamat ohjelmistokomponentin uudelleenkäytön oppaat tarjoavat tietyt kriteerit ohjelmistokomponentin uudelleenkäytön yhteydessä tarvittavalle informaatiolle. Ohjelmistokomponentin tyypistä riippuen eri asiat voidaan nähdä tarpeellisina. Sametinger (1997) esimerkiksi toteaa, että sellaisenaan uudelleenkäytettävien ohjelmistokomponenttien tapauksessa järjestelmädokumentaatiota ei ole tarpeen liittää osaksi uudelleenkäytön dokumentaatiota.

Jotta voidaan tarkastella ohjelmistodokumentaation kattavuutta, täytyy ensin tarkastella itse ohjelmistoa, sillä eri ohjelmistot saattavat poiketa toisistaan huomattavasti. Eri ohjelmistotyypeille on olemassa standardeja, joiden perusteella voidaan arvioida kunkin ohjelmistotyypin dokumentaation kattavuuutta. Tämän seurauksena ohjelmistodokumentaation kattavuus voidaan määritellä edellä mainitun lisäksi myös siten, että se tarkoittaa kaikkien ohjelmistotyypille soveltuvien standardien määrittelemien dokumenttien olemassaoloa. (Arthur ja Stevens, 1992)

Ohjelmistotyyppien tavoin myös eri ohjelmistokomponenttityypeille voidaan helposti kuvitella muodostuvan standardeja, tai ainakin jonkinlaisia ohjeita tarpeellisesta informaatiosisällöstä. Esimerkiksi Haikala ja Märijärvi (1997) jakavat ohjelmistokomponentit täysin yleiskäyttöisiin, sovellusaluekohtaisiin ja sovelluskohtaisiin ohjelmistokomponentteihin. Myös muut kirjoittajat, kuten esimer-

kiksi Sametinger (1997), Booch (1987), Chappell (1997) ja Kain (1998), ovat esittäneet erilaisia ohjelmistokomponenttien luokitteluja. Jaettaessa ohjelmistokomponentit eri tyyppeihin myös ohjelmistokomponentin dokumentaation kattavuus voidaan määritellä kaikkien ohjelmistokomponenttityypille soveltuvien standardien määrittelemien dokumenttien olemassaolona. Sametingerin (1997) ja Braunin (1994) esittämät uudelleenkäytön oppaat eivät kuitenkaan ole sidoksissa mihinkään ohjelmistokomponenttityyppiin, vaan soveltuvat yleisesti käytettäväksi erityyppisten ohjelmistokomponenttien dokumentoimiseksi.

4.2.3 Käytettävyys

Arthur ja Stevens (1992) esittävät, että ohjelmistodokumentaatioon liittyen käytettävyys määritellään dokumentaation sopivuutena (suitability), jota voidaan tarkastella suhteessa tarvittavan informaation löytämisen helppouteen. Tällöin erilaiset dokumentaation lukemista helpottavat osat, kuten esimerkiksi sisällysluettelo, hakemisto sekä listat kuvista ja taulukoista, parantavat dokumentaation käytettävyyttä. Myös Braun (1994) on kiinnittänyt huomiota ohjelmistokomponentin dokumentaation käytettävyyteen. Hänen mukaansa johdonmukainen lähestymistapa dokumentaation organisoinnin ja muodon (format) suhteen on eduksi ohjelmistokomponentin uudelleenkäyttäjän kannalta. Tämän voidaan ajatella perustuvan siihen, että tietyllä tavalla organisoidut ja muotoillut dokumentit parantavat niiden käytettävyyttä, eli helpottavat tarvittavan informaation löytymistä, sellaisten lukijoiden kannalta, joille tämäntyyppiset dokumentit ovat ennestään tuttuja. Arthurin ja Stevensin (1992) mukaan ohjelmistodokumentaation käytettävyyttä voidaan mitata esimerkiksi arvioimalla dokumentaation loogista jäljitettävyyttä, jolloin yksi arvioitava asia on se, kuinka helposti jokin kohta tai käsite on löydettävissä dokumentaatiosta.

Braun (1994) toteaa myös, että dokumentoinnissa kannattaa noudattaa tietyssä käyttäjäyhteisössä mahdollisesti hyväksyttyjä dokumentointistandardeja, koska

tämä parantaa merkittävästi dokumentaation uudelleenkäytön mahdollisuutta. Tällä Braun (1994) viittaa siihen, että dokumentaation tulee ohjelmistokomponentin tavoin olla uudelleenkäytettävä. Dokumentaation uudelleenkäytettävyys ei kuitenkaan suoranaisesti liity Arthurin ja Stevensin (1992) esittämän dokumentaation käytettävyyden määritelmään, joten sen ei voida katsoa parantavan dokumentaation käytettävyyttä. Dokumentointistandardeilla saattaa kuitenkin olla merkitystä käytettävyyyden parantamisen kannalta, mikäli ne kohdistuvat uudelleenkäytön huomioimisen sijasta juuri käytettävyyteen liittyviin seikkoihin.

4.2.4 Laajennettavuus

Arthurin ja Stevensin (1992) mukaan yleinen määritelmä laajennettavuudelle on kyky lisätä kohteen laajuutta (extent), lukumäärää (number), tilavuutta (volume) tai ulottuvuutta (scope). Ohjelmisto- ja ohjelmistokomponentin dokumentaation laajennettavuuden voidaan ajatella tarkoittavan sitä, että sen muuttaminen tai täydentäminen on tarpeen vaatiessa mahdollista. Arthur ja Stevens (1992) perustelevat laajennettavuuden ottamista yhdeksi ohjelmistodokumentaation riittävyyttä kuvaavaksi piirteeksi sillä, että ohjelmistodokumentaatiota on usein syytä päivittää ohjelmistolle tehtävien muutosten seurauksena. Jaaksi ym. (1999) toteavat, että ohjelmistoja kehitetään usein lisäämällä ohjelmiston toiminnallisuutta vähitellen, jolloin jokainen uusi julkaisu (release) tuo huomattavan toiminnallisen lisän (increment) olemassaolevan toiminnallisuuden päälle. Tällaista ohjelmistonkehitystä kutsutaan *inkrementaaliseksi ohjelmistonkehitykseksi* (ks. Jacobson ym., 1999 tai Jaaksi ym., 1999), ja sen yhteydessä on tärkeää, että myös ohjelmistoa koskeva dokumentaatio on tarpeen mukaan laajennettavissa. Arthur ja Stevens (1992) esittävätkin, että laajennettavuus voidaan määritellä ohjelmistodokumentaation yhteydessä sen muokattavuutena, jolla tarkoitetaan ohjelmistolle tehtyjen muutosten huomioimisen helppoutta ohjelmistodokumentaatiossa.

Sametingerin (1997) mukaan ohjelmistokomponentin dokumentaation tulee olla muokattavissa oleva (adaptable) sekä laajennettava (extensible). Arthur ja Stevens (1992) kuitenkin määrittelevät laajennettavuuden siten, että se kattaa myös muokattavuuden. Tämän tutkimuksen kannalta ei ole tärkeää, kattaako laajennettavuuden määritelmä muokattavuuden vai ei. Tärkeää on ymmärtää se, että ohjelmistodokumentaation muokkaaminen tai laajentaminen on merkittävä asia käytännön ohjelmistonkehityksen, erityisesti inkrementaalisen lähestymistavan yhteydessä esiintyvien evolutionaaristen ohjelmistojen (ks. Jaaksi ym., 1999), kannalta.

Ohjelmistodokumentaation laajennettavuuteen liittyy oleellisesti se, voidaanko dokumentaatiota käsitellä elektronisesti vai ei. Braun (1994) toteaa, että kaikki dokumentaatio tulee toimittaa elektronisessa muodossa (machine-readable form) siten, että dokumenttien esitysmuoto ei ole riippuvainen tietyistä tekstinkäsittelyohjelmista. Paperimuotoisen dokumentaation muokkaamista ja laajentamista ei voitane pitää kovinkaan järkevänä vaihtoehtona, sillä paperilla olevat dokumentit eivät yksinkertaisesti tarjoa kovinkaan hyviä edellytyksiä edellä mainituille toimenpiteille. Toinen syy elektronisen dokumentaation suosimiseen on mahdollisuus liittää ohjelmistodokumentaatio osaksi yrityksen elektronisen dokumenttien hallinnan (electronic document management, EDM) järjestelmää, mikäli sellainen on olemassa. Elektronisen dokumenttien hallinnan on todettu mm. lisäävän yritysten liiketoimintaprosessien tuottavuutta (ks. Sprague, 1995).

4.3 Yhteenveto

Tässä kappaleessa kuvattiin yleisesti ohjelmistojen dokumentointia sekä ohjelmistodokumentaation laatuun liittyviä piirteitä. Perinteinen ohjelmistodokumentaation yhteydessä esitetty jako sisältää käyttäjille suunnatun -, järjestelmä- ja prosessidokumentaation. Näistä yksikään ei kuitenkaan sisällä oh-

jelmistokomponenttien uudelleenkäytön kannalta tarpeellisia tietoja. Ohjelmistokomponenttien dokumentoinnin yhdeksi osa-alueeksi onkin ehdotettu uudelleenkäytön dokumentaatiota, joka kannattaa ymmärtää ohjelmistokomponenttikohtaisena uudelleenkäytön oppaana. Tämän oppaan tulee huomioida teknisen näkökulman lisäksi kaupallinen ja hallinnollinen näkökulma, sillä ohjelmistokomponenttien voidaan ajatella olevan itsenäisiä tuotteita. Ohjelmistokomponentin uudelleenkäytön oppaan on myös syytä tukea ohjelmistokomponentin uudelleenkäyttöprosessia. Tämä tarkoittaa sitä, että uudelleenkäytön oppaasta tulee löytyä tiedot uudelleenkäyttöprosessin eri tehtävien, kuten esimerkiksi ohjelmistokomponenttiehdokkaiden löytämisen sekä näiden sopivuuden arvioinnin, suorittamiseksi.

Ohjelmistodokumentaation laatua kuvaavia piirteitä ovat oikeellisuus, kattavuus, käytettävyys, ja laajennettavuus. Oikeellisuus tarkoittaa ohjelmiston lähdekoodin ja sitä kuvaavan dokumentaation yhteneväisyyttä. Ohjelmistokomponentin dokumentaatioon liittyen oikeellisuus on erityisen tärkeää uudelleenkäytön dokumentaation osalta, sillä se on muuta ohjelmistokomponentin dokumentaatiota useammin käytössä, ja siinä esiintyvät epäyhteneväisyydet ohjelmistokomponentin toiminnallisuuden kanssa saattavat vaikeuttaa tai jopa kokonaan estää ohjelmistokomponentin uudelleenkäytön. Ohjelmistodokumentaation voidaan todeta olevan kattava, mikäli se kattaa kaiken tarvittavan informaation. Kattavuus riippuu kuitenkin ohjelmiston tyypistä. Sen vuoksi kattavuus voidaan määritellä myös ohjelmistotyypille soveltuvien standardien määrittelemien dokumenttien olemassaolona. Ohjelmistokomponenteille on esitetty kirjallisuudessa erilaisia luokitteluja, joiden perusteella myös erityyppisille ohjelmistokomponenteille voidaan kuvitella muodostuvan erilaisia standardeja tai ohjeita tarpeellisesta informaatiosisällöstä. Käytettävyys määritellään ohjelmistodokumentaation sopivuutena, jota voidaan tarkastella suhteessa tarvittavan informaation löytämisen helppouteen. Tällöin esimerkiksi johdonmukainen dokumentointitapa saattaa lisätä dokumentaation käytettävyyttä, ainakin tiettyyn esitystapaan tottuneiden lukijoiden osalta. Laajennettavuus tarkoittaa

sitä, että ohjelmistodokumentaatiota voidaan täydentää tai muuttaa tarpeen vaatiessa. Erityisen tärkeää se on inkrementaalisen ohjelmistonkehityksen yhteydessä, sillä tämä perustuu ohjelmiston toiminnallisuuden asteittaiseen lisäämiseen.

5 RAJAPINTOJEN DOKUMENTOINTIMALLI

Tässä kappaleessa esitetään malli ohjelmistokomponentin rajapintojen dokumentoimiseksi. Tämä perustuu suurelta osin kappaleessa 3 tehtyihin havaintoihin ohjelmistokomponentin uudelleenkäytön yhteydessä tarvittavista tiedoista. Dokumentointimallia arvioidaan myös kappaleessa 2 esitetyn ohjelmistokomponentin uudelleenkäyttöprosessin sekä kappaleessa 4 kuvattujen hyvän ohjelmistodokumentaation kriteerien näkökulmista. Tarkoituksena on luoda sellainen dokumentointimalli, johon perustuva dokumentaatio tarjoaa mahdollisimman oikean kuvan ohjelmistokomponentin toiminnallisuudesta, ja joka lisäksi täyttää tässä tutkimuksessa käytettävät riittävän ohjelmistodokumentaation kriteerit mahdollisimman hyvin.

Dokumentointimallin 'kehittäminen' etenee seuraavasti. Ensin esitetään dokumentointimallin rakenne, joka perustuu kappaleessa 3 tehtyihin yleisiin havaintoihin ohjelmistokomponentin toiminnasta. Tämän jälkeen pohditaan sitä, kuinka erilaiset ohjelmistokomponentin toimintaan ja laatuun liittyvät seikat tulee kuvata, jotta dokumentointimalliin perustuva ohjelmistokomponentin dokumentaatio tarjoaisi uudelleenkäytön kannalta kaikki tarvittavat tiedot, mutta ei kuitenkaan paljastaisi ohjelmistokomponentin sisäiseen toiminnallisuuteen liittyviä asioita liian yksityiskohtaisella tasolla. Dokumentointimallin sisältöön ja rakenteeseen liittyviä asioita tarkastellaan tarpeen mukaan myös ohjelmistokomponentin uudelleenkäyttöprosessin sekä riittävän ohjelmistodokumentaation näkökulmista. Ohjelmistokomponentin uudelleenkäyttäjän kannalta kommentoidaan myös dokumentointimalliin perustuvan ohjelmistokomponentin dokumentaation käytettävyyttä.

5.1 Dokumentointimallin rakenne

Ohjelmistokomponentin rajapintojen dokumentointimalli jakaantuu yleiseen ja ohjelmistokomponentin toteuttaman yksittäisen palvelun kuvaavaan osaan. Edellinen sisältää ohjelmistokomponentin toiminnallisuuden yleisen kuvauksen. Jälkimmäinen puolestaan kattaa yhden ohjelmistokomponentin rajapinnassa määritellyn palvelun eli operaation kuvauksen.

Dokumentointimallin jakaminen kahteen osaan on perusteltua, sillä kaikkea ohjelmistokomponentin toiminnallisuutta, kuten esimerkiksi ohjelmistokomponentin tekemiä asynkronisia takaisinkutsuja, ei voida sisällyttää rajapinnoissa määriteltyjen yksittäisten palveluiden kuvauksiin. Kuitenkin myös ohjelmistokomponentin yleinen toiminnallisuus on syytä liittää dokumentointimallin osaksi, sillä sen huomioimisen voidaan katsoa olevan osa ohjelmistokomponentin rajapintojen tarjoamien palveluiden käyttöä. Esimerkiksi uudelleenkäytettävän ohjelmistokomponentin palveluita käyttävän olion tulee huomioida ohjelmistokomponentin tekemien asynkronisten takaisinkutsujen mahdollisuus. Siitä huolimatta, että ohjelmistokomponentin rajapinta voidaan teknisesti ymmärtää joukkona operaatioita (ks. Szyperski, 1997), on sen toiminnallisuuden dokumentoinnissa kuvattava muutakin kuin yksittäisiin operaatioihin liittyvät seikat.

Dokumentointimalliin perustuvan ohjelmistokomponentin dokumentaation voidaan ajatella koostuvan yleisen toiminnallisuuden sekä kaikkien ohjelmistokomponentin toteuttamien rajapintojen sisältämien palveluiden kuvauksista. Tämän perusteella dokumentaation ei voida kuitenkaan todeta olevan ohjelmistokomponentin toiminnallisuuden kattava kuvaus, sillä dokumentaation kattavuutta ei voida arvioida sen rakenteen perusteella. Dokumentaation kattavuuden arvioinnissa täytyy tarkastella sitä, sisältääkö dokumentaatio kaiken tarvittavan informaation (ks. Arthur ja Stevens, 1992). Tätä pohditaan doku-

mentointimallin sisällön kuvauksen yhteydessä. Seuraavaksi esitellään tarkemmin dokumentointimallin yleinen ja yksittäisen palvelun kuvaava osa.

5.1.1 Yleinen osa

Dokumentointimallin yleinen osa kattaa kaikki sellaiset ohjelmistokomponentin toiminnallisuuteen liittyvät asiat, joiden ei voida katsoa selkeästi kuuluvan minkään ohjelmistokomponentin toteuttaman yksittäisen operaation osaksi. Tällaisia asioita ovat ainakin ohjelmistokomponentin lähettämät herätteet sekä asynkroniset takaisinkutsut. Lisäksi tässä osassa voidaan kuvata esimerkiksi Sametingerin (1997) esittämän ohjelmistokomponentin uudelleenkäytön oppaan yleisen ja hallinnollisen informaation osien sisältämät tiedot. Näitä ei kuitenkaan esitetä dokumentointimallissa, sillä tässä tutkimuksessa keskitytään kuvaamaan ohjelmistokomponentin toiminnallisuuteen liittyvät asiat.

Braun (1994) ja Sametinger (1997) toteavat, että kullekin ohjelmistokomponentille tulee olla oma dokumentaationsa. Ohjelmistokomponentin eri versiot voidaan mieltää erillisiksi ohjelmistokomponenteiksi, joten myös niille tulee olla oma dokumentaationsa. Dokumentointimallin yleisessä osassa on selvyyden vuoksi syytä mainita ohjelmistokomponentin nimi ja versio. Jos ohjelmistokomponentin uusi versio korvaa aikaisemman toteutuksen aiheuttamatta syntaktisia muutoksia sen rajapintoihin, voidaan edellisen version dokumentaatiota muokata semanttisten muutosten osalta siten, että se kuvaa ohjelmistokomponentin uuden version toiminnallisuuden.

5.1.2 Yksittäisen palvelun kuvaava osa

Dokumentointimallin yksittäisen palvelun kuvaava osa kattaa ohjelmistokomponentin totetuttaman yksittäisen operaation toiminnallisuuden kuvauksen. Tässä osassa voidaan kuvata esimerkiksi operaation esi- ja jälkiehdot, invarian-

tit, sekä toiminnan ulkoinen kuvaus sekvenssikaavion avulla. Tarkempi selvitys kuvattavista asioista esitetään kuitenkin dokumentointimallin sisällön kuvauksen yhteydessä.

Dokumentointimallin yksittäisen palvelun kuvaavassa osassa on otettava kantaa siihen, minkä ohjelmistokomponentin toteuttaman rajapinnan palvelusta on kysymys, sillä eri rajapinnat saattavat sisältää samannimisiä palveluita. Proseduraalisia rajapintoja ei tosin ole tapana nimetä erikseen, mutta ohjelmistokomponentin proseduraaliset rajapinnat eivät voine sisältää useita samannimisiä palveluita.

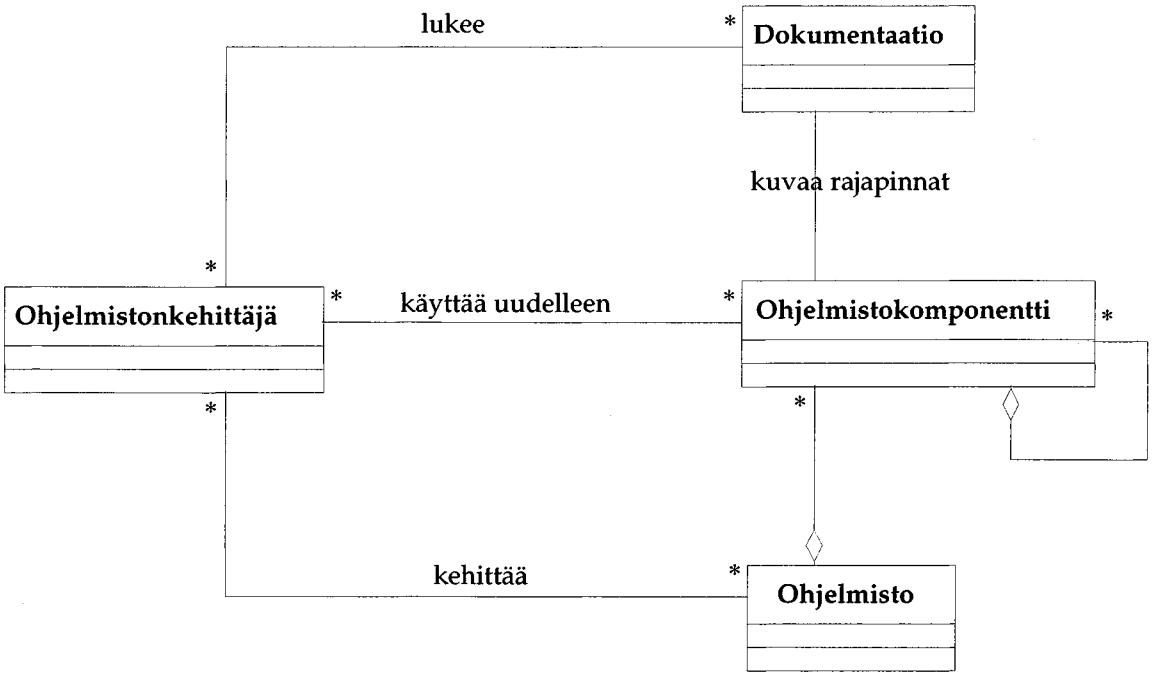
Ohjelmistokomponentin dokumentaation luettavuutta parantaa se, että kunkin rajapinnan tarjoamien palveluiden kuvaukset esitetään loogisesti yhteenkuuluvana joukkona. Tähän ei dokumentointimallissa voida ottaa kantaa, sillä siinä esitetään vain tapa kuvata ohjelmistokomponentin toteuttama yksittäinen operaatio, eikä kiinnitetä huomiota siihen, missä järjestyksessä rajapinnan sisältämät palvelut tulee dokumentaatiossa esittää. Ohjelmistokomponentin dokumentaatio kannattanee kuitenkin järjestää siten, että tietyn rajapinnan sisältämien palveluiden kuvaukset löytyvät dokumentaatiosta suhteellisen helposti.

Dokumentointimalliin perustuvan ohjelmistokomponentin dokumentaation laajentamisen voidaan todeta onnistuvan joissakin tapauksissa ilman suuria ongelmia. Mikäli ohjelmistokomponentti esimerkiksi toteuttaa jonkin uuden rajapinnan, ei dokumentaation täydentäminen aiheuta ongelmia, sillä yksittäisten palveluiden ja siten myös rajapintojen kuvaukset ovat toisistaan suhteellisen riippumattomia.

5.2 Dokumentointimallin sisältö

Dokumentointimallin sisällöllä tarkoitetaan ohjelmistokomponentin toiminnallisuuden kuvausta, joka jakaantuu toiminnan ja sen laadun kuvauksiin. Ohjelmistokomponentin rajapintojen dokumentointimallin tarkoituksena on tarjota yleinen malli suorituskelpoisten ohjelmistokomponenttien rajapintojen dokumentoimiseksi. Siihen perustuvan ohjelmistokomponentin dokumentaation voidaan ajatella vastaavan pääpiirteittäin Sametingerin (1997) esittämän uudelleenkäytön oppaan arviointia tukevan sekä uudelleenkäytön informaation osuutta, vaikka dokumentointimallissa ei otetakaan kantaa kaikkiin näiden osuuksien sisältämiin asioihin.

KUVA 7 täydentää KUVASSA 2 esitettyä käsitteellistä mallia, joka kuvaa ohjelmistojen 'rakentamisen' ohjelmistokomponenteista, lisäämällä mukaan käsitteen dokumentaatio, jonka tehtävänä on KUVAN 7 mukaisesti kuvata ohjelmistokomponentin rajapinnat ja siten tarjota ohjelmistonkehittäjille tarvittavat tiedot ohjelmistokomponentin uudelleenkäyttämiseksi. Dokumentaatiolla on toki muitakin tehtäviä, mutta edellä mainittu kuvaa dokumentaation roolin tämän tutkimuksen näkökulmasta uudelleenkäytettäviin ohjelmistokomponentteihin perustuvan ohjelmistojen kehittämisen yhteydessä. KUVASSA 7 on kardinalisuuksien avulla eksplisiittisesti esitetty myös se, että jokaisen ohjelmistokomponentin rajapinnat on kuvattu vain yhdessä dokumentaatiossa ja toisaalta tietty dokumentaatio kuvaa vain yhden ohjelmistokomponentin rajapinnat. Jokaiselle ohjelmistokomponentille on siis oma dokumentaationsa (ks. Braun, 1994 tai Sametinger, 1997).



KUVA 7. Dokumentaation rooli ohjelmistokomponenttien uudelleenkäyttöön perustuvassa ohjelmistojen kehittämisessä

On tärkeää, että dokumentaatio antaa ohjelmistokomponentin toiminnallisuudesta oikean kuvan. Tähän on tosin vaikea ottaa kantaa dokumentointimallin tasolla, sillä se ei tarjoa keinoja dokumentaation oikeellisuuden todentamiseen. Ohjelmistonkehittäjien kannalta hyödyllistä on myös se, että dokumentaatio tukee ohjelmistokomponentin uudelleenkäyttöprosessia mahdollisimman hyvin. Tämä tarkoittaa sitä, että dokumentaatio on jäsennelty siten, että sen avulla voidaan esimerkiksi ensin löytää sopivien ohjelmistokomponenttiehdokkaiden joukko, ja tämän jälkeen voidaan tutkia kyseisen joukon sisältämien ohjelmistokomponenttien ominaisuuksia tarkemmin. Esimerkiksi Sametingerin (1997) esittämä ohjelmistokomponentin uudelleenkäytön opas tukee uudelleenkäyttöprosessia edellä mainitulla tavalla. Tässä tutkimuksessa esitettävä ohjelmistokomponentin rajapintojen dokumentointimalli rajoittuu kuitenkin antamaan tarkat tiedot ohjelmistokomponentin toiminnallisuudesta, eikä siihen perustuva ohjelmistokomponentin dokumentaatio tämän seurauksena tue uudelleenkäyttöprosessia kovinkaan hyvin. Seuraavaksi

tarkastellaan sitä, kuinka ohjelmistokomponentin toiminta ja toiminnan laatu voidaan dokumentointimallissa kuvata.

5.2.1 Ohjelmistokomponentin toiminnan kuvaus

Ohjelmistokomponentin toiminnan kuvaamisessa keskitytään pääasiassa yksittäisten operaatioiden toiminnan kuvaamiseen, mutta myös yleisemmät kuvaukset ovat mahdollisia. Esimerkiksi ohjelmistokomponentin tilan muuttuminen saattaa toimia herätteenä jonkin toisen ohjelmistokomponentin toiminnalle. Tämä ajatus perustuu Tarkkailija-suunnittelumalliin (Observer design pattern, Gamma, Helm, Johnson ja Vlissides, 1995, s. 293 – 303). Edellä mainitun kaltaiset herätteet (events) perustuvat kuitenkin normaaliin olioiden viestinvälitysmekanismiin (ks. esimerkiksi Budd, 1997), joten tässä tutkimuksessa tarkastelun kohteena olevien olioperustaisten ohjelmistokomponenttien herätteiden kuvaaminen onnistuu myös kuvaamalla näiden yksittäisten operaatioiden toiminta. Tosin esimerkin kaltaisten herätteiden esittäminen abstraktimmalla kuin yksittäisten operaatioiden tasolla saattaa helpottaa ohjelmistokomponenttien välisten riippuvuuksien ymmärtämistä. Ohjelmistonkehittäjät tarvitsevat yksityiskohtaista tietoa ohjelmistokomponentin toiminnasta, joten operaatioiden kuvauksia ei voida korvata kuvaamalla ohjelmistokomponenttien riippuvuudet yleisellä tasolla. Sen sijaan tällaisten kuvausten voidaan ajatella täydentävän ohjelmistokomponentin toiminnan määrittelyä, ja ne voidaan liittää dokumentointimallin yleiseen osaan. Vaikka kaikki ohjelmistokomponentit eivät lähetäkään herätteitä, on kyseessä kuitenkin niin yleinen oliomekanismi, että herätteiden liittäminen dokumentointimallin osaksi on perusteltua.

Ohjelmistokomponentti voi toimia itsenäisesti tai vuorovaikutuksessa joko olioiden tai muiden ohjelmistokomponenttien kanssa. Itsenäisen toiminnan kuvaaminen on varsin suoraviivaista verrattuna ohjelmistokomponenttien tai ohjelmistokomponentin ja yhden tai useamman olion välisen vuorovaikutuksen

kuvaamiseen, sillä ulkoiset – ja takaisinkutsut eivät ole mahdollisia silloin, kun ohjelmistokomponentti ei palvelupyynnön suorittaakseen joudu kommunikoimaan olioiden tai muiden ohjelmistokomponenttien kanssa. Kaikenlaisten ohjelmistokomponenttien operaatioiden toiminnan kuvaamiseen voidaan aina liittää osaksi invarianttien ja ohjelmistokomponentin tilan tarkastelu, vaikkakin jälkimmäisen merkitys korostuu vuorovaikutteisten ohjelmistokomponenttien toiminnan kuvaamisen yhteydessä.

Jaaksi ym. (1999) kuvaavat OMT++-menetelmän analyysivaiheessa kehitettävän järjestelmän tukemat toiminnot ohjelmistokomponenttien yhteistoimintana (component collaboration). Nämä toiminnot ovat loppukäyttäjän kannalta merkityksellisiä ja sisällöltään ei-triviaaleja. Ohjelmistokomponenttien yhteistoiminnan kuvaukseen Jaaksi ym. (1999) liittävät kuvattavan toiminnon nimen, tarvittavat esiehdot, yhteistoimintaa kuvaavan sekvenssikaavion, mahdolliset poikkeukset, sekä jälkiehdot, jotka ovat voimassa toiminnon päätyttyä.

Jaaksin ym. (1999) mukaan sekvenssikaavioita voidaan käyttää sekä suoritettavien (executable) että toteutettavien (implementable) ohjelmistokomponenttien yhteistoiminnan kuvaamiseen. Näistä ensimmäisellä he tarkoittavat itsenäisiä prosesseja ja laitteita, jälkimmäisellä puolestaan itsenäisesti kehitettäviä ohjelmistokomponentteja. Edellä mainittu jaottelu perustuu ohjelmistoarkkitehtuurin kuvaamiseen eri näkökulmista (ks. Kruchten, 1995). Jaaksin ym. (1999) mukaan suoritettavat ohjelmistokomponentit ovat osana ohjelmistoarkkitehtuurin suorituksen aikaista näkökulmaa (run-time view) ja toteutettavat ohjelmistokomponentit ovat osana ohjelmistoarkkitehtuurin kehittämisen näkökulmaa (development view). Edellä esitettyjen Jaaksin ym. (1999) kuvaamien erityyppisten ohjelmistokomponenttien perusteella voidaan jälleen todeta, kuinka ohjelmistokomponentille voidaan eri yhteyksissä antaa varsin erilaisia merkityksiä.

Jaaksin ym. (1999) käyttämä tapa kuvata järjestelmän toiminnot OMT++-menetelmän analyysivaiheen aikaisten ohjelmistokomponenttien yhteis-

toimintana tarjoaa perustellun lähtökohdan myös suorituskelpoisten ohjelmistokomponenttien rajapintojen dokumentoimiselle, sillä molemmissa tapauksissa on kyse ohjelmistokomponenttien ulkoisen toiminnan kuvaamisesta. Lisäksi edellä mainittu Jaaksin ym. (1999) kuvaustapa sisältää esi- ja jälkiehdot, jotka liittyvät keskeisesti myös sopimuksenmukaisten rajapintojen kuvaamiseen (ks. Meyer, 1992 tai Szyperski, 1997).

Sekvenssikaavioita käytetään varsin usein oliokeskeisen ohjelmistonkehityksen yhteydessä järjestelmän dynaamisten piirteiden mallintamiseen, kun taas luokkakaavioiden (class diagrams) avulla saadaan mallinnettua järjestelmän staattiset ominaisuudet (ks. Jaaksi ym. 1999, tai Jacobson, Booch ja Rumbaugh, 1999). Järjestelmän dynaamisten piirteiden mallintamiseen voidaan käyttää sekvenssikaavioiden sijaan myös yhteistoimintakaavioita (collaboration diagrams, ks. Jacobson ym., 1999), mutta ne eivät kuitenkaan tarjoa yhtä paljon mahdollisuuksia ohjelmistokomponentin toiminnan kuvaamiseen kuin sekvenssikaaviot. Jaaksin (1996a) mukaan OMT++-menetelmän yhteydessä käytettävien herätekaavioiden (event trace diagrams) avulla voidaan esittää olioiden välittämien viestien lisäksi ehto- ja toistorakenteita, kommentteja sekä olioiden tiloja ja toimintoja. Herätekaavioita kutsutaan UML:n vakiintumisen myötä monessa yhteydessä sekvenssikaavioksi (ks. Booch ym., 1998 tai Jaaksi ym., 1999), vaikka niiden käyttötarkoitus tai esitystapa ei ole muuttunut lainkaan. Tämä tarkoittaa sitä, että kaikki edellä mainitut herätekaavioiden avulla mallinnettavat asiat voidaan esittää myös sekvenssikaavioita käyttäen.

Sekvenssikaaviot soveltuvat hyvin ohjelmistokomponentin suorittamien ulkoisten - ja synkronisten takaisinkutsujen kuvaamiseen, sillä näiden mahdolliset suoritusajankohdat ovat aina ennalta tiedossa. Ulkoisten kutsujen yhteydessä on tärkeää kuvata ne tilat, joissa ohjelmistokomponentti tekee ulkoisia kutsuja, sekä näiden kutsujen suoritusjärjestys (ks. Büchi ja Weck, 1997). Sen sijaan asynkronisten takaisinkutsujen kuvaaminen sekvenssikaavioiden avulla ei ole mielekästä, sillä sekvenssikaavioita käytetään juuri ohjelmiston dynaamisten ts.

ajallisten piirteiden mallintamiseen, eikä asynkronisten takaisinkutsujen suoritusajankohtia tiedetä ennakolta. Ohjelmistokomponentin tekemät asynkroniset takaisinkutsut eivät välttämättä liity mihinkään tiettyyn ohjelmistokomponentin toteuttamaan palveluun, joten ne tulee kuvata dokumentointimallin yleisessä osassa.

Kaikkien ohjelmistokomponentin toteuttamien ja suorittamien palvelupyyntöjen kutsumuodot on myös syytä liittää osaksi dokumentointimallia. Yleensä operaation kutsumuoto sisältää palvelun eli operaation nimen, mahdollisten parametrien nimet ja tietotyypit, sekä mahdollisen paluuarvon nimen ja tietotyypin (ks. esimerkiksi Stroustrup, 1997). Kutsun kohde on myös syytä mainita. Sekvenssikaavioiden avulla kuvattavien ulkoisten kutsujen yhteydessä kutsujen kohteet ovat havaittavissa suoraan itse kaaviosta. Takaisinkutsut kohdistuvat aina ohjelmistokomponentin asiakkaaseen. Myös nämä voidaan eksplisiittisesti kuvata sekvenssikaavion avulla. Ohjelmistokomponentin toteuttaman operaation kutsumuoto voidaan esittää dokumentointimallin yksittäisen palvelun kuvaavassa osassa.

KUVASSA 8 on esitetty kootusti se, kuinka ohjelmistokomponentin toiminta voidaan kuvata dokumentointimallin avulla. KUVAN 8 sisältö voidaan tulkita seuraavalla tavalla. Dokumentointimalli on jaettu kahteen osaan; yleiseen ja yksittäisen palvelun kuvaavaan osaan. Dokumentoitavat asiat on esitetty lihavoidulla tekstillä, ja näistä kaikista on annettu myös esimerkki. Dokumentointimallissa ei siis kiinnitetä tarkasti kuvattavien asioiden esitystapaa. Tätä voidaan perustella sillä, että dokumentointimallia on tarkoitus voida soveltaa yleisesti erityyppisille suorituskelpoisille ohjelmistokomponenteille, joiden dokumentoinnissa voidaan joutua noudattamaan erilaisia esitystapoja. Erityyppisillä suorituskelpoisilla ohjelmistokomponenteilla tarkoitetaan esimerkiksi ActiveX- tai JavaBeans-ohjelmistokomponentteja. KUVASSA 8 esitettyä ohjelmistokomponentin toiminnan kuvaavaa dokumentointimallin osaa täydennetään jatkossa siten, että se kattaa myös ohjelmistokomponentin laadun kuvauksen.

YLEINEN OSA

Ohjelmistokomponentin nimi ja versio: `Pankkitili` 0.1

Herätteet: `Päivitä()`-pyyntö välitetään `abstraktiTarkkailija`-rajapinnan toteuttaville rekisteröityneille vastaanottajille ohjelmisto-komponentin saldon muuttuessa.

Asynkroniset takaisinkutsut: `void huomauta(String huomautus)`

YKSITTÄISEN PALVELUN KUVAAVA OSA

Toteutettavan rajapinnan nimi: `abstraktiPankkitili`

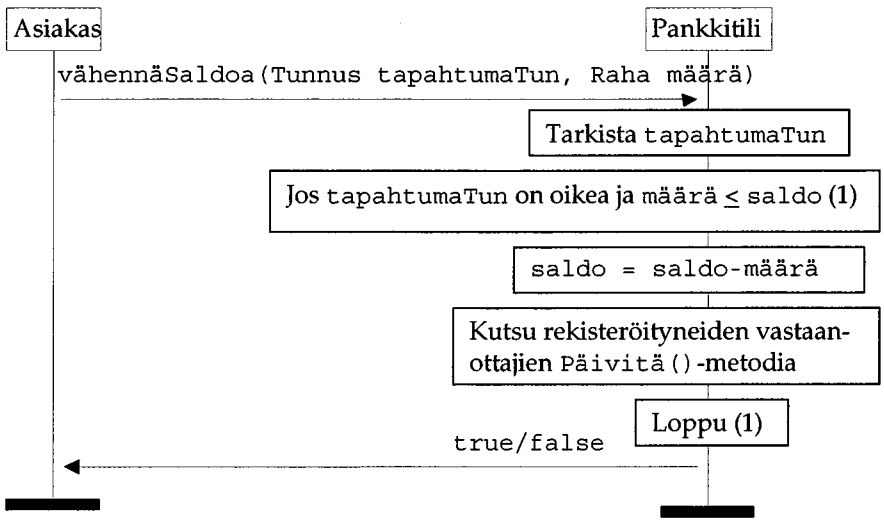
Operaation nimi:

`boolean vähennäSaldoa(Tunnus tapahtumaTun, Raha määrä)`

Esiehdot: Asiakas on hankkinut käyttöönsä tarvittavan `tapahtumaTun`-tunnuksen.

Invariantit: `saldo`-kentän arvoa lukuunottamatta ohjelmisto-komponentin tila säilyy muuttumattomana.

Sekvenssikaavio:



Jälkiehdot: `saldo = saldo – määrä`

Poikkeukset: `tapahtumaTun`-tunnuksen tarkistaminen ei onnistunut: Poikkeus palautetaan asiakkaalle.

KUVA 8. Ohjelmistokomponentin toiminnan kuvaava dokumentointimallin osa

KUVASSA 8 on esitetty itse dokumentointimallin lisäksi se, kuinka kuvitellun `Pankkitili`-ohjelmistokomponentin yleinen sekä yhden operaation toiminta voidaan dokumentointimalliin perustuen esimerkinomaisesti kuvata. Seuraavaksi selvitetään, mitä KUVASSA 8 esitetyillä `Pankkitili`-ohjelmistokomponentin toimintaa kuvaavilla esimerkeillä tarkoitetaan. Esimerkit eivät muodosta kattavaa kuvausta `Pankkitili`-ohjelmistokomponentin toiminnasta, vaan niiden avulla on esitetty vain se, kuinka dokumentoitavat asiat on mahdollista kuvata.

`Pankkitili`-ohjelmistokomponentin versionumeroksi on KUVASSA 8 annettu yksinkertaisesti 0.1. Yksittäisen palvelun kuvaavassa osassa esimerkkinä käytetty operaatio sisältyy `abstraktiPankkitili`-rajapintaan, joka on yksi ohjelmistokomponentin toteuttamista rajapinnoista. Mikäli jokin ohjelmistokomponentin operaatioista kuuluu osana tämän proseduraaliseen rajapintaan (ks. Szyperski, 1997), ei dokumentaatiossa voida kyseisen operaation kuvauksen yhteydessä esittää toteutettavan rajapinnan nimeä, sillä proseduraalisia rajapintoja ei ole tapana nimetä erikseen. Tällöin ohjelmistokomponentin dokumentaatiossa on syytä mahdollisten sekaannusten välttämiseksi jollakin tapaa mainita se, että operaatio ei kuulu mihinkään erikseen nimettyyyn oliorajapintaan (ks. Szyperski, 1997).

`Pankkitili`-ohjelmistokomponentti lähettää saldonsa muuttumisen seurauksena herätteet `abstraktiTarkkailija`-rajapinnan toteuttaville vastaanottajille, jotka voivat olla olioita tai toisia ohjelmistokomponentteja. Herätteiden lähettäminen tarkoittaa tässä tapauksessa `Päivitä()`-operaation kutsumista kaikkien rekisteröityneiden vastaanottajien osalta, ja nämä voivat herätteen saadessaan esimerkiksi kysyä tarkan saldotiedon `Pankkitili`-ohjelmistokomponentilta. Toinen mahdollisuus on se, että herätteen mukana välitetään parametrina tieto nykyisestä saldosta, jolloin herätteiden vastaanottajien ei tarvitse kysyä sitä `Pankkitili`-ohjelmistokomponentilta. Ajatus ohjel-

mistokomponentin tilan muuttumisen seurauksena lähetettävistä herätteistä on peräisin Gamman ym. (1995) esittämästä Tarkkailija-suunnittelumallista, ja edellä mainitut lähetettävien herätteiden variaatiot ovat esimerkkejä siitä, kuinka suunnittelumalleja voidaan soveltaa eri tavoin käytännön ohjelmistonkehitystyössä. Tarkkailija-suunnittelumallin mukaan oliot voivat rekisteröityä dynaamisesti tietyn kohdeolion lähettämien herätteiden vastaanottajiksi (ks. Gamma ym., 1995), ja tämän seurauksena `Pankkitili`-ohjelmistokomponentin lähettämien herätteiden vastaanottajia ei voida sen dokumentaatiossa tarkasti esittää.

`Pankkitili`-ohjelmistokomponentin tekemistä asynkronisista takaisinkutsuista on annettu esimerkkinä operaatio `huomauta(String huomautus)`, jota `Pankkitili`-ohjelmistokomponentti kutsuu silloin, kun sen tulee huomauttaa asiakastaan jonkin sen tarjoaman palvelun suoritukseen liittyvästä asiasta. Kaikkien `Pankkitili`-ohjelmistokomponentin asiakkaiden tulee tarjota toteutus kyseiselle operaatiolle huolimatta siitä, tuleeko `Pankkitili`-ohjelmistokomponentille koskaan tarvetta kutsua tätä. Esimerkkinä tällaisesta tarpeesta on tilanne, jossa `Pankkitili`-ohjelmistokomponentti ilmoittaa asiakkaalleen tilin olevan tilapäisesti käyttökiellossa. Tällöin kyseinen asiakas voi esimerkiksi välittää `huomautus`-viestin eteenpäin jollekin käyttöliittymäkerrokseen (view layer, ks. esimerkiksi Jaaksi ym., 1999) kuuluvalle oliolle tai ohjelmistokomponentille, joka voi puolestaan näyttää tämän vaikkapa laskua maksavalle henkilölle. Dokumentointimallissa voidaan kuvata myös asynkronisten takaisinkutsujen syyt, vaikka KUVASSA 8 niin ei ole tehtykään, sillä tämän voidaan ajatella parantavan ohjelmistokomponentin toiminnasta saatavaa kuvaa. Toisaalta asynkronisten takaisinkutsujen syiden poisjättämistä voidaan perustella sillä, että ohjelmistokomponentin asiakas joutuu joka tapauksessa varautumaan siihen, että ohjelmistokomponentti saattaa tehdä asynkronisia takaisinkutsuja.

Operaation kutsumisen esiehtona on se, että Pankkitili-ohjelmistokomponentin asiakas on hankkinut itselleen tarvittavan tapahtumaTun-tunnuksen, jota Pankkitili-ohjelmistokomponentti tarvitsee tapahtuman vahvistamiseen. Esiehtojen ei tarvitse välttämättä olla suoranaisia vaatimuksia sille, että operaatiota on mahdollista kutsua, vaan ne voidaan ymmärtää laajemmassa merkityksessä seikkoina, jotka on syytä huomioida ennen operaation kutsumista. Jälkimmäistä ei voidaan kuitenkaan ymmärtää esiehtojen vahvistamisena (ks. Szyperski, 1997), sillä asiakkaan on täytettävä kirjatut esiehdot ennen operaation kutsumista, vaikka operaation kutsuminen olisikin teknisesti mahdollista ilman niiden täyttämistä.

Sekvenssikaavio kuvaa operaation onnistuneen suorituksen. KUVAN 8 esimerkissä asiakkaana voi toimia vaikkapa luokasta Lasku muodostettu olio. Asiakas kutsuu Pankkitili-ohjelmistokomponentin toteuttamaa operaatiota, jonka kutsumuoto on boolean vähennäSaldoa(Tunnus tapahtumaTun, Raha määrä). Operaation kutsumisen seurauksena Pankkitili-ohjelmistokomponentti tarkistaa ensin tapahtumaTun-tunnuksen, ja vähentää tämän jälkeen muuttujan saldo arvoa parametrina välitetyn muuttujan määrä sisältämän arvon verran, mikäli tapahtumaTun on oikea ja määrä $\leq$ saldo. Lisäksi edellä mainittujen ehtojen täyttymisen seurauksena Pankkitili-ohjelmistokomponentti kutsuu abstraktiTarkkailija-rajapinnan toteuttavien rekisteröityneiden vastaanottajien Päivitä()-operaatiota. Sama asia on kuvattu dokumentointimallin yleisessä osassa esimerkkinä ohjelmistokomponentin lähettämistä herätteistä. Sekvenssikaaviossa esitetty Loppu (1) –merkintätapa tarkoittaa sitä, että ehtolause päättyy siihen. Tämän jälkeen operaatio palauttaa tiedon siitä, oliko operaation suoritus onnistunut vai ei. Vaikka KUVAN 8 esimerkissä ei olekaan esitetty Pankkitili-ohjelmistokomponentin tekemiä ulkoisia kutsuja, voidaan sekvenssikaavioiden avulla kuvata myös Büchin ja Weckin (1997) esityksen mukaisesti tilat, joissa ohjelmistokomponentti tekee ulkoisia kutsuja sekä näiden kutsujen suoritusjärjestys. Samoin voidaan

kuvata ohjelmistokomponentin tekemät synkroniset eli jonkin sen operaation kutsumisesta aiheutuvat takaisinkutsut.

Jälkiehdot ovat voimassa vain, jos operaation suoritus on tapahtunut onnistuneesti. KUVAN 8 esimerkissä onnistunut suoritus tarkoittaa sitä, että operaatio palauttaa arvon `true`. Tällöin `Pankkitili`-ohjelmistokomponentin asiakas voi luottaa siihen, että dokumentaatiossa esitetty jälkiehto `saldo = saldo - määrä` pitää paikkansa. Sekvenssikaavion `true/false` –paluuarvo tarkoittaa sitä, että operaatio palauttaa `boolean`-tyyppisen muuttujan, jonka arvo on `true` tai `false` riippuen siitä, onko operaation suoritus onnistunut vai ei. Invariantit eli operaation suorituksen ajan voimassa olevat ehdot eivät sen sijaan riipu siitä, onko operaation suoritus onnistunut vai ei.

KUVAN 8 operaatio aiheuttaa poikkeuksen, mikäli `tapahtumaTun`-tunnuksen tarkistaminen ei onnistu. Tällöin `Pankkitili`-ohjelmistokomponentin aiheuttaman ja palauttaman poikkeuksen kirjaa virhelokiin jokin sellaisen olio tai ohjelmistokomponentti, joka huolehtii kaikkien järjestelmän aiheuttamien poikkeuksien lokiin kirjaamisesta. `Pankkitili`-ohjelmistokomponentti ei itse huolehdi poikkeusten käsittelystä. Mikäli jokin operaatio aiheuttaa poikkeuksen, voidaan implisiittisenä oletuksena pitää sitä, että operaation suoritus keskeytyy. Tätä ei ole syytä mainita ohjelmistokomponentin dokumentaation yhteydessä, eikä poikkeuksen sattuessa operaation jälkiehdot ole voimassa.

Dokumentointimalli kattaa kaikki kappaleessa 3 esitetyt, yleisesti kirjallisuudessa kuvatut, ohjelmistokomponentin ulkoiseen toimintaan liittyvät seikat. Tämän perusteella dokumentointimallissa esitetyn dokumentoitavien asioiden joukon voidaan arvioida olevan kohtuullisen kattava kuvaus ohjelmistokomponentin ulkoisesta toiminnasta. Seuraavaksi tarkastellaan sitä, kuinka ohjelmistokomponentin laatu voidaan dokumentointimallissa kuvata.

5.2.2 Ohjelmistokomponentin laadun kuvaus

Ohjelmistokomponentin laadun kuvaamisessa rajaudutaan tarkastelemaan ohjelmistokomponentin vaatimien resurssien, kuten ajan ja muistin käytön (ks. Szyperski, 1997), esittämistä dokumentointimalliin perustuvassa ohjelmistokomponentin dokumentaatiossa. Edellä mainitut ovat jokaisen ohjelmistokomponentin vaatimia resursseja, joten vaatimus niiden esittämisestä ohjelmistokomponentin dokumentaatiossa on perusteltua.

Szyperskin (1997) mukaan sopimuksenmukaisten rajapintojen tulee ihannetapauksessa kattaa ohjelmistokomponentin kaikkien olennaisten toiminnallisten ja laadullisten piirteiden kuvaukset. Vaikka jälkimmäisten kuvaamiseen ei vielä tällä hetkellä olekaan mitään yleisesti noudatettua tapaa, toteaa Szyperski (1997), että joitakin esimerkkejä laadullisten piirteiden kuvaamisesta on jo olemassa. Yksi tällainen on C++-ohjelmointikielen STL-kirjasto (Standard Template Library, Musser ja Saini, 1996), jossa kiinnitetään abstraktien funktioiden hyväksyttävien (legal) toteutusten suoritusajat. Tämä tarkoittaa sitä, että tietyn abstraktin funktion hyväksyttävät toteutukset eivät saa ylittää määriteltyä suoritusaikaa. Kiinnitettyä suoritusaikaa ei ole määritelty sekunteina, sillä se on toteutuksen lisäksi riippuvainen käytettävissä olevasta laitteistoalustasta (platform). Sen sijaan hyväksyttävien toteutusten aikavaativuus (time complexity) on määritelty, kuten on usein tapana tehdä myös algoritmien analysoinnin yhteydessä (ks. Cormen, Leiserson ja Rivest, 1990 tai Penttonen, 1997). Cormen ym. (1990) toteavat, että algoritmin analysoinnilla tarkoitetaan algoritmin vaatimien resurssien, kuten esimerkiksi suoritusajan, muistin ja tietoliikenneyhteyden kaistanleveyden (communication bandwidth), ennustamista.

Cormen ym. (1990) ovat esittäneet, kuinka väliinsijoitus-algoritmin (insertion sort algorithm) suoritusajan aikavaativuus voidaan määritellä. Samaa esimerkkiä ovat käyttäneet monet muutkin kirjoittajat, kuten esimerkiksi Purdom ja Brown (1985) sekä Banachowski, Kreczmar ja Rytter (1991), joten sen voidaan

ajatella sopivan hyvin käytettäväksi myös tässä tutkimuksessa esimerkkinä operaation suoritusajan aikavaativuuden määrittelystä. Väliinsijoitus-algoritmin toimintaperiaate voidaan selvittää pelikorttien avulla; monet ihmiset asettavat pakasta ottamansa pelikortit tiettyyn järjestykseen kädessään siten, että he asettavat pakasta ottamansa kortin heti oikeaan paikkaan vertaamalla sitä muihin kädessä oleviin kortteihin (ks. Cormen ym., 1990). Väliinsijoituksen voidaan ajatella toimivan saman periaatteen mukaisesti. Esimerkiksi kokonaislukutaulukon järjestäminen tapahtuu käymällä taulukko läpi vasemmalta oikealle ja vertaamalla kutakin taulukossa olevaa kokonaislukua sen vasemmalla puolella oleviin kokonaislukuihin oikealta vasemmalle. Kokonaisluvun paikka määräytyy joko vertailuehdon toteutumisen tai vertailun kohteena olevien kokonaislukujen loppumisen perusteella. Jälkimmäinen toteutuu silloin, kun mikään vertailun kohteena oleva kokonaisluku ei täytä vertailun ehtoa. Tällöin vertailtava kokonaisluku sijoittuu taulukossa ensimmäiseksi. Mikäli kokonaislukutaulukko järjestetään esimerkiksi nousevaan suuruusjärjestykseen, on vertailun ehtona se, että vertailun kohteena oleva kokonaisluku on pienempi kuin vertailtava kokonaisluku. Pienimmän kokonaisluvun tapauksessa mikään taulukossa oleva kokonaisluku ei täytä vertailun ehtoa, joten kokonaisluku sijoittuu taulukossa ensimmäiseksi.

KUVASSA 9 on esitetty väliinsijoitus-algoritmin Java-ohjelmointikielinen toteutus, joka perustuu Cormenin ym. (1990, s. 3) esittämään väliinsijoitus-algoritmin pseudokoodiin. Syynä pseudokoodin muuntamiseen lähdekoodiksi on se, että ohjelmistokomponentin toteuttamien operaatioiden aikavaativuuksien määrittämiseksi joudutaan tarkastelemaan näiden operaatioiden toteutuksia, jotka on esitetty pseudokoodin sijasta lähdekoodina. KUVASSA 9 kuvatun lähdekoodin toteutus järjestää taulukon `A` sisältämät kokonaisluvut nousevaan suuruusjärjestykseen. Taulukon `A` sisältämien kokonaislukujen määrä saadaan selville taulukon `length`-kentän avulla (ks. Gosling, Joy ja Steele, 1996). Taulukon indeksointi alkaa nollasta, joten `for`-silmukassa käydään läpi taulukon kaikki muut paitsi ensimmäinen elementti. Tämä perustuu

siihen, että taulukon ensimmäistä elementtiä ei voida verrata mihinkään, koska sen vasemmalla puolella ei ole elementtiä, johon sitä voisi verrata. Sen sijaan `for`-silmukan ensimmäisellä suorituskerralla taulukon toista elementtiä verrataan taulukon ensimmäiseen elementtiin, joka toimii tällöin vertailun kohteena. KUVAN 9 vasemmassa reunassa olevat numerot ovat lähdekoodin ohjelmarivien numeroita, eivätkä kuulu itse lähdekoodiin.

```
1 for (int j = 1; j < A.length; j++) {
2     int x = A[j];
3     int i = j-1;
4     while (i >= 0 && A[i] > x) {
5         A[i+1] = A[i];
6         i = i-1; }
7     A[i+1] = x; }
```

KUVA 9. Väliinsijoitus-algoritmin Java-ohjelmointikielinen toteutus

Cormenin ym. (1990) mukaan yleisesti voidaan todeta, että algoritmin suoritusaika kasvaa sen syötteen koon (size of input) kasvaessa, joten ohjelman suoritusaika kuvataan tavallisesti sen syötteen koon funktiona. Esimerkiksi kolme kokonaislukua sisältävän taulukon lajittelu on nopeampaa kuin taulukon, jossa on sata kokonaislukua. Toinen väliinsijoitus-algortimin suoritusaikaan vaikuttava tekijä on se, kuinka lähellä oikeaa järjestystä taulukko on ennen lajittelun suorittamista (ks. Cormen ym., 1990). Mikäli taulukko on jo valmiiksi lähellä haluttua järjestystä, pienenee algoritmin suoritusaika, sillä taulukon elementtien vertailuun ei tarvitse käyttää paljoa aikaa.

KUVASSA 9 esitetyn lähdekoodin toteutuksen suoritusajan aikavaativuuden määrittelemiseksi täytyy kunkin lausekkeen osalta selvittää, kuinka monta kertaa se suoritetaan sekä mikä sen suoritusaika on (ks. Cormen ym., 1990). Nämä tiedot on esitetty TAULUKOSSA 1 ohjelmariveittäin kuvattuna. Ohjelmarivit vastaavat tässä tapauksessa suoritettavia lausekkeita ts. jokainen lauseke on esitetty omalla ohjelmarivillään (ks. KUVA 9).

TAULUKKO 1. Väliinsijoitus-algoritmin Java-ohjelmointikielisen toteutuksen lähdekoodin lausekkeiden suoritusajat ja –kerrat ohjelmariveittäin kuvattuna

ohjelmarivi	suoritusaika	suorituskerrat
1	s_1	n
2	s_2	$n-1$
3	s_3	$n-1$
4	s_4	$\sum_{j=1}^{n-1} t_j$
5	s_5	$\sum_{j=1}^{n-1} (t_j-1)$
6	s_6	$\sum_{j=1}^{n-1} (t_j-1)$
7	s_7	$n-1$

Kunkin lausekkeen suoritusajan voidaan olettaa olevan vakio (ks. Cormen ym., 1990) eli KUVAN 9 ohjelmarivin i sisältämän lausekkeen suoritusaika s_i on vakio. TAULUKON 1 mukaan ensimmäinen `for`-lauseke suoritetaan n kertaa. Tämä perustuu siihen, että kyseisessä silmukassa käydään läpi taulukon, jonka koko on n, kaikki muut paitsi ensimmäinen elementti, jolloin suorituskertoja tulee $n - 1$. Lisäksi `for`-lauseke suoritetaan vielä yhden kerran, kun taulukon kaikki elementit on käyty läpi. Tällöin ehto on epätosi, ja `for`-silmukkaan ei enää mennä. Ohjelmarivien 2, 3 ja 7 sisältämien lausekkeiden suorituskertojen lukumäärä on kunkin osalta $n - 1$, sillä ne suoritetaan normaalisti `for`-silmukan osana. Sen sijaan ohjelmariveillä 4, 5 ja 6 olevien lausekkeiden suorituskertojen

lukumäärien selvittäminen ei ole niin suoraviivaista, sillä ne ovat osa `for`-silmukan sisällä olevaa `while`-silmukkaa. Ohjelmarivillä 4 olevan `while`-lausekkeen suorituskertojen lukumäärä riippuu muuttujan `j` arvosta. TAULUKOSSA 1 esiintyvä t_j tarkoittaakin `while`-lausekkeen suorituskertojen lukumäärää tietyllä muuttujan `j` arvolla.

Cormenin ym. (1990) mukaan algoritmin suoritusaika on jokaisen lausekkeen suoritusaikojen summa; lauseke, jonka suoritusaika on s_i, ja joka suoritetaan n kertaa, muodostaa $s_i n$ suuruisen osuuden algoritmin suoritusaikaan. KUVASSA 9 esitetyn väliinsijoitus-algoritmin Java-ohjelmointikielisen toteutuksen suoritusajan laskeminen ei poikkea Cormenin ym. (1990) kuvaamasta lisäyslajittelu-algoritmin suoritusajan laskemisesta, joka tapahtuu laskemalla yhteen lausekkeiden suoritusaikojen ja -kertojen (ks. TAULUKKO 1) tulot:

$$T(n) = s_1 n + s_2(n-1) + s_3(n-1) + s_4 \sum_{j=1}^{n-1} t_j + s_5 \sum_{j=1}^{n-1} (t_j - 1) + s_6 \sum_{j=1}^{n-1} (t_j - 1) + s_7(n-1) .$$

$T(n)$ tarkoittaa algoritmin syötteen koon funktiona laskettavaa algoritmin suoritusaikaa. Cormen ym. (1990) toteavat, että algoritmin suoritusaika voi vaihdella, vaikka syötteen koko pysyisikin samana. Tämä johtuu heidän mukaansa siitä, että syöte voi olla sisällöltään erilainen, vaikka olisikin saman kokoinen. Esimerkiksi väliinsijoitus-algoritmin parhaan tapauksen suoritusaika (best-case running time) saavutetaan silloin, kun taulukko on jo valmiiksi oikeassa järjestyksessä (ks. Cormen ym., 1990). Tällöin `while`-lausekkeen ehto on aina epätosi, koska vertailun kohteena oleva luku on aina pienempi tai yhtäsuuri kuin vertailtava luku. Toisin sanoen KUVAN 9 `while`-lausekkeessa `A[i] <= x`, kun muuttuja `i` saa alkuarvonsa `j-1` (ks. Cormen ym., 1990). Tämän seurauksena $t_j = 1$ ja `while`-silmukkaan ei mennä lainkaan. Väliinsijoitus-algoritmin parhaan tapauksen suoritusaika voidaan laskea seuraavasti (ks. Cormen ym., 1990):

$$T(n) = s_1 n + s_2(n-1) + s_3(n-1) + s_4(n-1) + s_7(n-1)$$

$$= (s_1 + s_2 + s_3 + s_4 + s_7)n - (s_2 + s_3 + s_4 + s_7) .$$

Samalla tavalla voidaan laskea myös väliinsijoitus-algoritmin Java-ohjelmointikielisen toteutuksen parhaan tapauksen suoritusaika. Cormen ym. (1990) toteavat, että edellä lasketun väliinsijoitus-algoritmin parhaan tapauksen suoritusajan ilmaisemiseen voidaan käyttää myös esitystapaa $an + b$, jossa a ja b ovat lausekkeiden suoritusajoista s_i riippuvia vakioita; väliinsijoitus-algoritmin parhaan tapauksen suoritusajan voidaan siis todeta olevan *lineaarinen* (linear) kertoimen n funktio. Tämä tarkoittaa sitä, että oikeassa järjestetyksessä olevien taulukoiden väliinsijoitus-algoritmin suoritusaika kasvaa samassa suhteessa kuin taulukon koko.

Mikäli taulukko on järjestetty käänteiseen järjestykseen ennen väliinsijoituksen suorittamista, saavutetaan taulukon lajittelussa pahimman tapauksen suoritusaika (worst-case running time, ks. Cormen ym., 1990). KUVASSA 9 esitetyn Java-ohjelmointikielisen väliinsijoitus-algoritmin toteutuksen tapauksessa tämä tarkoittaa sitä, että taulukon kokonaisluvut on järjestetty laskevaan suuruusjärjestykseen. Tällöin kaikkia `for`-silmukassa läpikäytäviä taulukon kokonaislukuja joudutaan vertaamaan kaikkiin taulukossa sen vasemmalla puolella oleviin kokonaislukuihin. Toisin sanoen t_j = `j` (ks. Cormen ym., 1990), sillä `while`-silmukkaan mennään `j-1` kertaa, jonka lisäksi `while`-lauseke suoritetaan vielä yhden kerran, jolloin `i < 0` eli ehto on epätosi. Tämän seurauksena vertailtava kokonaisluku sijoittuu taulukossa aina ensimmäiseksi. Koska t_j = `j`, voidaan TAULUKOSSA 1 ohjelmariveillä 4, 5 ja 6 olevat lausekkeiden suorituskerrat esittää myös seuraavalla tavalla:

$$\sum_{j=1}^{n-1} t_j = \sum_{j=1}^{n-1} j = (n(n+1)/2) - n \text{ ja } \sum_{j=1}^{n-1} (t_j - 1) = \sum_{j=1}^{n-1} (j-1) = n(n-1)/2 .$$

Tämä perustuu siihen, että väliinsijoitus-algoritmin pahimman suoritusajan tapauksessa TAULUKOSSA 1 ohjelmariveillä 4, 5 ja 6 olevien lausekkeiden

suorituskertojen summalauseet voidaan kukin esittää aritmeettisena sarjana (arithmetic series), kuten yllä on tehty (ks. Cormen ym., 1990). TAULUKOSSA 1 ohjelmariveillä 5 ja 6 olevien lausekkeiden suoritusaikojen summalauseet ovat samat, joten yllä esitetyistä aritmeettisista sarjoista jälkimmäinen kuvaa näiden molempien aritmeettisen sarjan. Näiden tietojen perusteella väliinsijoitus-algoritmin pahimman tapauksen suoritusaika voidaan laskea seuraavasti (ks. Cormen ym., 1990):

$$\begin{aligned} T(n) &= s_1 n + s_2(n-1) + s_3(n-1) + s_4(n(n+1)/2 - 1) + s_5(n(n-1)/2) \\ &\quad + s_6(n(n-1)/2) + s_7(n-1) \\ &= (s_4/2 + s_5/2 + s_6/2)n^2 + (s_1 + s_2 + s_3 + s_4/2 - s_5/2 - s_6/2 + s_7)n \\ &\quad - (s_2 + s_3 + s_4 + s_7)\,. \end{aligned}$$

Samalla tavalla voidaan laskea myös väliinsijoitus-algoritmin Java-ohjelmointikielisen toteutuksen pahimman tapauksen suoritusaika. Cormenin ym. (1990) mukaan edellä esitetty väliinsijoitus-algoritmin pahimman tapauksen suoritusaika voidaan ilmaista myös muodossa $an^2 + bn + c$, jossa a, b ja c ovat lausekkeiden suoritusajoista s_i riippuvia vakioita; väliinsijoitus-algoritmin pahimman tapauksen suoritusajan voidaan siis todeta olevan *neliöllinen* (quadratic) kertoimen n funktio. Tämä tarkoittaa sitä, että käänteisessä järjestetyksessä olevien taulukoiden tapauksessa väliinsijoitus-algoritmin suoritusaika kasvaa varsin nopeasti taulukon koon kasvaessa.

Edellä esitettiin väliinsijoitus-algoritmin sekä parhaimman että pahimman tapauksen suoritusajan laskeminen. Yleensä ei kuitenkaan olla kiinnostuneita tarkasta lopputuloksesta, vaan ainoastaan sen kasvukertoimesta (order of growth), jolla on muiden kuin pienten syötteiden yhteydessä ratkaiseva merkitys algoritmin suoritusajan kannalta; tällöin tarkastellaan algoritmin asymptoottista tehokkuutta (ks. Cormen ym., 1990) Myös Penttonen (1997) toteaa, että on tärkeämpää tietää kasvun tyyppi kuin sen kerroin. Kasvun tyypillä

Penttonen (1997) tarkoittaa samaa kuin Cormen ym. (1990) asymptoottisella tehokkuudella.

Cormen ym. (1990) esittävät, että esimerkiksi väliinsijoitus-algoritmin pahimman tapauksen suoritusajan aikavaativuus voidaan ilmaista käyttäen muotoa $\Theta(n^2)$, jota sovelletaan algoritmin pahimman tapauksen suoritusajan aikavaativuuden asymptoottisesti tarkan rajan (asymptotically tight bound) esittämiseksi. Cormen ym. (1990) antavat tälle notaatiolle tarkan määritelmän. Sen mukaan annetulle funktiolle $g(n)$ merkinnällä $\Theta(g(n))$ tarkoitetaan seuraavaa funktioiden joukkoa: $\Theta(g(n)) = \{f(n)$: on olemassa positiiviset vakiot c_1, c_2 ja n_0 siten, että $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ kaikilla arvoilla $n \geq n_0\}$. Tämän määritelmän perusteella voidaan formaalisti osoittaa, että esimerkiksi $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ laskemalla positiiviset vakiot c_1, c_2 ja n_0 siten, että $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ kaikilla arvoilla $n \geq n_0$ (ks. Cormen ym., 1990). Ohjelmistokomponentin toteuttaman operaation suoritusajan aikavaativuuden voidaankin formaalisti osoittaa olevan esimerkiksi $\Theta(n^2)$.

Algoritmille voidaan antaa myös pahimman tapauksen suoritusajan aikavaativuuden asymptoottinen ylä- ja alaraja (asymptotic upper and lower bound), joiden vastaavat merkintätavat ovat $O(g(n))$ ja $\Omega(g(n))$, ja joille on olemassa asymptoottisesti tarkan rajan tapaan tarkat määritelmät (ks. Cormen ym., 1990 tai Penttonen, 1997). Tässä tutkimuksessa ei kuitenkaan ole tarpeen määrittää ohjelmistokomponentin toteuttaman operaation suoritusajan aikavaativuuden asymptoottisia rajoja, sillä näiden tunteminen ei ole ohjelmistokomponentin uudelleenkäytön kannalta olennaista. Edellä esitettyjä notaatioita voidaan kuitenkin käyttää ohjelmistokomponentin toteuttaman operaation suoritusajan aikavaativuuden ilmaisemiseen. Cormenin ym. (1990) mukaan $f(n) = \Theta(g(n))$ jos ja vain jos $f(n) = O(g(n))$ ja $f(n) = \Omega(g(n))$. Tämän seurauksena notaation $\Theta(g(n))$ voi ajatella soveltuvan parhaiten ohjelmistokomponentin toteuttaman operaation suoritusajan aikavaativuuden esittämiseksi, koska tällöin myös muiden notaatioiden määritelmät ovat voimassa. Cormen ym. (1990)

kuitenkin käyttävät notaatiota $\Omega(g(n))$ yleisesti algoritmin suoritusajan aikavaativuuden ilmaisemiseksi. He perustelevat tätä sillä, että vaikka algoritmin parhaan tapauksen suoritusajan aikavaativuuden esittämiseen on perusteltua käyttää asymptoottisen alarajan ilmaisevaa Ω-notaatiota, ei silti ole väärin todeta algoritmin pahimman tapauksen suoritusajan aikavaativuuden olevan esimerkiksi $\Omega(n^2)$. Tässä tutkimuksessa käytetään jatkossa Cormenin ym. (1990) tavoin Ω-notaatiota yleisesti ohjelmistokomponentin toteuttaman operaation suoritusajan aikavaativuuden esittämiseksi. Horowitzin ja Sahnin (1978) mukaan operaation, jonka suoritusaika on vakio, aikavaativuus voidaan esittää käyttäen muotoa O(1), mutta sama asia voidaan ilmaista myös Cormenin ym. (1990) tavoin käyttäen muotoa $\Omega(1)$.

Cormenin ym. (1990) mukaan algoritmin pahimman tapauksen suoritusajan laskeminen on suositeltavaa kolmesta syystä:

- Algoritmin pahimman tapauksen suoritusaika on syötteen koosta tai sisällöstä riippumatta algoritmin suoritusajan yläraja, jonka tietäminen takaa algoritmin suoritusajan olevan aina korkeintaan tämä.
- Joidenkin algoritmien tapauksessa pahimman tapauksen suoritusaika on melko usein myös algoritmin todellinen suoritusaika.
- Keskimääräisen tapauksen (average case) suoritusaika on usein karkeasti arvioiden sama kuin pahimman tapauksen suoritusaika, sillä sekin on usein neliöllinen kertoimen n funktio.

Ohjelmistokomponentin toteuttaman operaation pahimman tapauksen suoritusajan voidaan ajatella soveltuvan hyvin esitettäväksi sopimuksenmukaisten rajapintojen ajatukseen (ks. Meyer, 1992 tai Szyperski, 1997) perustuvan ohjelmistokomponentin dokumentaation yhteydessä, sillä ohjelmistokomponentin toteuttaman operaation suoritusajan voidaan taata olevan enintään tämä. Operaation toteutus voi tällöin kuitenkin vapaasti ylittää velvollisuutensa eli suoriu-

tua operaatiosta nopeammin kuin mikä sen pahimman tapauksen suoritusaika on (ks. Szyperski, 1997).

Penttonen (1997) sen sijaan toteaa, että toisinaan algortimin pahimman tapauksen suoritusajan toteutuminen on hyvin harvinaista. Esimerkkinä hän käyttää pikalajittelu-algoritmia (quicksort algorithm), jonka pahimman tapauksen suoritusajan aikavaativuus on $\Omega(n^2)$, vaikka se toimiikin yleensä hyvin nopeasti. Tämän seurauksena Penttonen (1997) esittää, että tällaisissa tapauksissa keskimääräisen tapauksen suoritusajan aikavaativuus antaa realistisemman kuvan tehtävän vaikeudesta. Se ei kuitenkaan ole riittävä ilmaistavaksi sopimuksenmukaisten rajapintojen ajatukseen perustuvassa ohjelmistokomponentin dokumentaatiossa, sillä ohjelmistokomponentin asiakas ei voi luottaa siihen, että operaation suoritus todella on aikavaatimuksen mukainen. Operaation keskimääräisen suoritusajan aikavaativuus voidaan liittää osaksi ohjelmistokomponentin rajapintojen dokumentaatiota operaation pahimman tapauksen suoritusajan aikavaativuuden ohella. Tätä voidaan perustella sillä, että se täydentää ohjelmistokomponentin toiminnallisuudesta saatavaa kuvaa niissä tapauksissa, joissa pahimman tapauksen todennäköisyys on pieni.

Ohjelmistokomponentin toteuttaman operaation vaatimien aikaresurssien määrittämistä saattaa vaikeuttaa erilaiset monisäikeisen ympäristön (multithreaded environment) mukanaan tuomat ongelmat. Säikeellä tarkoitetaan yksittäistä, tiettyä järjestystä noudattavaa kontrollivuota (single sequential flow of control) ohjelmassa (ks. Campione ja Walrath, 1998). Vaikka säikeiden todetaan yleisesti toimivan rinnakkaisesti, ei tämä käytännössä useinkaan pidä paikkaansa, sillä yhden prosessorin järjestelmissä säikeiden suoritus tapahtuu yksi kerrallaan siten, että säikeet vain näyttävät toimivan rinnakkaisesti (ks. Campione ja Walrath, 1998). Säikeet voivatkin välillä joutua odottamaan omaa suoritusvuoroaan, minkä seurauksena ohjelmistokomponentin toteuttaman operaation vaatimien aikaresurssien määrittäminen vaikeutuu. Tässä tutkimuksessa ei kuitenkaan kiinnitetä huomiota monisäikeisen ympäristön aiheuttamiin ongel-

miin aikaresurssien määrittämisen suhteen, sillä kaikki ohjelmistokomponentit eivät sovellu käytettäväksi tällaisessa ympäristössä, ja dokumentointimallia on tarkoitus voida soveltaa mahdollisimman monelle erityyppiselle ohjelmistokomponentille.

Ohjelmistokomponentin toteuttaman operaation vaatimien muistiresurssien määrittämiseen ei ole olemassa mitään kovin helppoa tapaa eikä siitä ole algoritmien analysoinnin yhteydessä juurikaan kirjoitettu. Esimerkiksi Cormen ym. (1990) toteavat vain, että algoritmin vaatimia muistiresursseja ei voida laskea algoritmin suoritusajan tapaan, sillä lauseke, joka käyttää m muistiyksikköä, ja joka suoritetaan n kertaa, ei välttämättä kuluta mn yksikköä muistia. Tämän huomion perusteella algoritmin ja myös ohjelmistokomponentin toteuttaman operaation vaatimien muistiresurssien määrittämisen voidaan ajatella olevan suoritusajan aikavaativuuden määrittämistä vaikeampi tehtävä. Vaikka ohjelmistokomponentin toteuttamien operaatioiden vaatimat resurssit saataisiin jollakin tapaa tarkasti määriteltyäkin, täytyy huomioida se, että dokumentointimalliin perustuvan ohjelmistokomponentin dokumentaation tuottaminen ei saa olla liian työläs tai vaikea toimenpide. Dokumentointimallin tulee olla sellainen, että se tarjoaa hyvän tuen uudelleenkäytettävien ohjelmistokomponenttien rajapintojen dokumentoimiselle, jolloin sen käytettävyys tulee ottaa huomioon sisällön ohella.

Dynaaminen muistinvaraus saattaa tuottaa lisää ongelmia ohjelmistokomponentin vaatimien muistiresurssien määrittämisen suhteen. Esimerkiksi Java-ohjelmointikieli tukee automaattista roskankeruuta (garbage collection, ks. Campione ja Walrath, 1998), joka tarkoittaa sitä, että Javan ajonaikainen ympäristö (Java runtime environment), joka tunnetaan myös Java-alustana (Java Platform, ks. Flanagan, 1999), huolehtii käytöstä pois jääneiden olioiden tuhoamisesta. Käytöstä pois jääneillä olioilla Campione ja Walrath (1998) tarkoittavat sellaisia olioita, joihin ei ole olemassa viitteitä (ks. Campione ja Walrath, 1998). Automaattisen roskankeruun seurauksena olioiden tuhoamisajankohtaa ei tie-

detä tarkasti, sillä Campionen ja Walrathin (1998) mukaan automaattinen roskankeruu käynnistyy jaksoittaisesti (periodically) eli asynkronisesti. Ohjelmoija voi tosin käynnistää roskankeruun itsekin kutsumalla tarvittavia Java-alustan sisältämän `System`-luokan operaatioita (ks. Gosling ym., 1996). Roskankeruu on kuitenkin aikaa vievä tapahtuma, eikä sitä kannata käynnistää aina, kun jokin olio jää käytöstä pois. Campione ja Walrath (1998) toteavat, että roskankeruu tulisi käynnistää sellaisena ajankohtana, jolloin sillä ei ole vaikutusta ohjelman suorituskyvyn suhteen. Java-ohjelmointikielestä poiketen esimerkiksi C++-ohjelmointikieltä käyttävän ohjelmoijan tulee aina itse huolehtia olioiden vaatiman tilan vapauttamisesta (ks. esimerkiksi Stroustrup, 1997), jolloin olioiden tuhoamisajankohta tiedetään jo ohjelman kehittämisvaiheessa.

Ohjelmistokomponentin toteuttaman operaation vaatimien muistiresurssien määrittäminen eroaa aikaresurssien määrittämisestä siinä, että edellisen osalta täytyy huomioida kumulatiivinen resurssienkulutus. Jokin operaatio saattaa esimerkiksi varata dynaamista muistia, mutta ei vapauta sitä lainkaan. Tällöin saman tai jonkin toisen operaation kutsumisen seurauksena suoritettava dynaaminen muistinvaraus lisää entisestään ohjelmistokomponentin vaatimien muistiresurssien määrää. Java-ohjelmointikielellä toteutettujen ohjelmien tapauksessa automaattinen roskankeruu saattaa itsessään aiheuttaa hetkittäisen muistiresurssien kumulatiivisen kasvun, mikäli ohjelmoija ei itse huolehdi roskankeruun käynnistämisestä aina, kun jokin olio jää käytöstä pois.

Ohjelmistokomponentin vaatimien kumulatiivisten muistiresurssien määrittäminen on vaikeaa, sillä ennakolta ei voida tietää, kuinka asiakkaat käyttävät sen tarjoamia palveluita. Kuitenkin ohjelmistokomponentin uudelleenkäytön kannalta tärkeänä tietona voidaan pitää sitä, kuinka paljon ohjelmistokomponentti voi enimmillään käyttää muistiresursseja. Tämän tiedon esittäminen osana dokumentointimallia noudattaa myös ajatusta sopimuksenmukaisista rajapinnoista (ks. Meyer, 1994 tai Szyperski, 1997), sillä tällöin voidaan taata se, että ohjelmistokomponentti ei missään tapauksessa kuluta enempää muistire-

sursseja kuin mitä sen dokumentaatiossa on mainittu. Tietoa ohjelmistokomponentin vaatimien muistiresurssien enimmäismäärästä ei kuitenkaan kannata liittää osaksi dokumentointimallia siitä syystä, että vaatimus sen määrittämisestä alentaa merkittävästi dokumentointimallin käytettävyyttä. Tällä tarkoitetaan sitä, että muistiresurssien enimmäismäärän selvittäminen on niin hankala toimenpide, ettei sen suorittamista voida perustellusti edellyttää dokumentointimallissa. Kyseessä voidaan ajatella olevan samantapainen *inhimillinen tekijä* (human factor), johon vedoten Büchi ja Weck (1997) hylkäävät ajatuksen formaalin esitystavan soveltamisesta ohjelmistokomponenttien toiminnan kuvaamisen yhteydessä. Tässä tutkimuksessa tyydytäänkin jatkossa dokumentoimaan ohjelmistokomponentin vaatimien muistiresurssien vähimmäismäärä, jolla tarkoitetaan pelkästään ohjelmistokomponentin käyttöönoton aiheuttamaa tilan tarvetta. Ratkaisua voidaan perustella edellä mainitun lisäksi sillä, että ohjelmistokomponentin vaatimien muistiresurssien vähimmäismäärä ei vaihtele enimmäismäärän tavoin ajan suhteen, eikä siihen vaikuta myöskään esimerkiksi Java-ohjelmointikielen roskankeruu-mekanismi. Edelleen muistiresurssien vähimmäismäärän dokumentoimisen voidaan ajatella olevan hyvä ratkaisu siinä mielessä, että se ei ole liian työläs tai vaativa toimenpide, jonka seurauksena dokumentointimallin käytettävyyden voidaan todeta perustellusti olevan parempi.

Toisaalta yksittäisten operaatioiden vaatimien muistiresurssien kuvaukset antavat paremman kuvan ohjelmistokomponentin toiminnallisuudesta kuin pelkän ohjelmistokomponentin käyttöönoton vaatimien muistiresurssien määrän kuvaaminen. Näiden molempien selvittäminen on edellytyksenä ohjelmistokomponentin vaatimien muistiresurssien kumulatiivisen määrän esittämiselle, joten jälkimmäisen dokumentoiminen on perusteltua. Dokumentointimallissa voidaan kuvata myös yksittäisten operaatioiden vaatimat muistiresurssit, mikäli niiden määrittäminen onnistuu kohtuullisella vaivalla.

Ohjelmistokomponentin vaatimien muistiresurssien vähimmäismäärän selvittämiseen on olemassa erilaisia tapoja, joista yksi on ohjelmistokomponentin att-

ribuuttien muistivarauksien yhteenlaskeminen. Perustietotyyppien, joita ovat esimerkiksi C++- tai Java-ohjelmointikielen `int`- ja `char`-tietotyypit, osalta tämä ei tuota suuria vaikeuksia, mutta itse määriteltyjen tai käytettävissä olevien abstraktien tietotyyppien (abstract data type, ks. esimerkiksi Budd, 1997 tai Stroustrup, 1997) tapauksessa tilanne on toinen. Erityisesti jälkimmäisten muistivarauksien tutkiminen saattaa olla todella hankalaa tai jopa mahdotonta, mikäli tietotyypin sisäinen rakenne ei ole tiedossa. Esimerkiksi Java-alustan sisältämän `String`-luokan olion vaatimien muistiresurssien selvittäminen on mahdotonta, mikäli kyseisen luokan toteutuksesta tiedetään vain sen tarjoamat palvelut. Java–alustan uusin versio (Java 1.2 tai Java 2 Platform) tarjoaa 1520 erilaista luokkaa ohjelmoijan käyttöön (ks. Flanagan, 1999), joten sen palveluita käyttävän ohjelmistokomponentin vaatimien muistiresurssien vähimmäismäärän selvittäminen laskemalla yhteen ohjelmistokomponentin kaikkien attribuuttien vaatimat muistiresurssit voidaan pitää liian työläänä tehtävänä. Lisäksi on otettava huomioon se, että ohjelmistokomponentin vaatimien muistiresurssien vähimmäismäärä saattaa vaihdella esimerkiksi sen mukaan, miten ohjelmistokomponenttia on mukautettu ennen sen liittämistä osaksi muuta ohjelmistoa. Szyperski (1997) toteaa, että ohjelmistokomponentin loogisesti muodostavista olioista luodaan ilmentymiä tarpeen mukaan. Tämän seurauksena ohjelmistokomponentin käyttöönoton vaatimien muistiresurssien määrän ei voida olettaa aina olevan sama. Tämä ei kuitenkaan vaihtele ajan suhteen, kuten aikaisemmin todettiin, vaan ohjelmistokomponentin mukauttamisen seurauksena.

Toinen tapa ohjelmistokomponentin vaatimien muistiresurssien vähimmäismäärän selvittämiseen on tämän kokeellinen määrittäminen. Tällä tarkoitetaan sitä, että esimerkiksi jonkin käyttöjärjestelmän prosessien vaatimien muistiresurssien määrittämiseen soveltuvan ohjelmiston avulla selvitetään, kuinka paljon muistia pelkkä ohjelmistokomponentin käyttöönotto vaatii. Tämän menetelmän etuna on sen yksinkertaisuus, vaikka siihenkin saattaa liittyä joitakin ongelmia. Yksi tällainen on esimerkiksi se, kuinka ohjelmistokomponentin vaa-

timan tilan tarve voidaan erottaa sen käynnistävän prosessin varaamista muistiresursseista. Kokeellisen muistiresurssien määrittämisen haittapuolena voidaan pitää sitä, että sen avulla saatavat tulokset ovat riippuvaisia käytettävästä laiteympäristöstä. Tosin on huomattava, että myöskään ohjelmistokomponentin attribuuttien vaatiman tilan määrä ei välttämättä ole aina sama. Esimerkiksi Hirvonen (1995) toteaa, että perustietotyypeille varattavan tilan suuruus vaihtelee laiteympäristöittäin. Dokumentointimallissa onkin syytä mainita ohjelmistokomponentin vaatimien muistiresurssien vähimmäismäärän yhteydessä myös laiteympäristö tai käyttöjärjestelmä, jossa saadut tulokset pitävät paikkansa. Muistiresurssien kokeellisen määrittämisen tapauksessa voidaan itse muistiresurssien vähimmäismäärän lisäksi mainita kääntäjä, jota ohjelmistokomponentin kehittämisessä on käytetty, sekä ohjelmisto, jonka avulla muistiresurssien vähimmäismäärä on selvitetty. Koska ohjelmistokomponentin vaatima tila saattaa vaihdella sen mukaan, miten sitä on mukautettu, voidaan mukaan liittää tarvittaessa myös tiedot siitä, mitä toimenpiteitä ohjelmistokomponentille on tehty ennen sen käyttöönottoa. Kaupalliseen levitykseen suunnattujen ohjelmistokomponenttien tapauksessa lienee näiden dokumentaatiossa syytä esittää vaadittavat muistiresurssien vähimmäismäärät sekä mukauttamista kuvaavia tietoja lukuunottamatta edellä mainitut tiedot useamman kuin yhden laiteympäristön osalta.

Dokumentointimalli kattaa yleisimmät ohjelmistokomponentin toiminnan laatuun liittyvät seikat eli vaadittavat aika – ja muistiresurssit. Edellinen voidaan kuvata kaikkien ohjelmistokomponentin toteuttamien operaatioiden osalta, mutta jälkimmäisen osalta tyydytään esittämään ohjelmistokomponentin käyttöönoton vaatimien muistiresurssien määrä. Aika – ja muistiresurssien määrittäminen ei tarjoa kattavaa kuvausta ohjelmistokomponentin toiminnan laadusta, mutta dokumentointimallissa onkin tarkoitus esittää vain yleisimmät ohjelmistokomponentin toiminnan laatua kuvaavat seikat.

KUVASSA 10 on esitetty kootusti, kuinka ohjelmistokomponentin laatu voidaan dokumentointimallin avulla kuvata. KUVA 10 täydentää KUVASSA 8 esitettyä ohjelmistokomponentin toiminnan kuvaavaa dokumentointimallin osaa.

YLEINEN OSA

Ohjelmistokomponentin käyttöönoton vaatimien muistiresurssien määrä:

JVM mukaanlukien 5230 KB

Kääntäjä: Java SDK 1.2.2 –kehitysympäristöön kuuluva javac

Käyttöjärjestelmä: SunOS 5.7

Muistiresurssien määrä selvitetty käyttöjärjestelmän komennon `ps` avulla.

YKSITTÄISEN PALVELUN KUVAAVA OSA

Operaation suoritusajan aikavaativuus: $\Omega(n)$

KUVA 10. Ohjelmistokomponentin laadun kuvaava dokumentointimallin osa

KUVAN 10 esitystapa voidaan tulkita samaan tapaan kuin KUVAN 8 esitystapa; dokumentoitavat asiat on esitetty lihavoituna. Esimerkkinä on käytetty Java-ohjelmointikielellä toteutettua ohjelmistokomponenttia, joita ovat JavaBeans- sekä Enterprise JavaBeans -ohjelmistokomponentit. KUVASSA 10 ohjelmistokomponentin laadun kuvaaminen jaetaan toiminnan kuvaamisen tapaan kahteen osaan; yleiseen ja yksittäisen palvelun kuvaavaan osaan. Edellisessä esitetään ohjelmistokomponentin vaatimien muistiresurssien vähimmäismäärä, jolla tarkoitetaan pelkän ohjelmistokomponentin käyttöönoton vaatimien muistiresurssien määrää. Jälkimmäisessä kuvataan puolestaan yksittäisen operaation suoritusajan aikavaativuus.

Esimerkkinä käytetyn ohjelmistokomponentin käyttöönoton todetaan vaativan yhdessä Java-virtuaalikoneen (JVM, Java Virtual Machine, ks. Lindholm ja Yellin, 1999) kanssa 5230 KB muistia. Koska vaadittavien muistiresurssien määrä on

selvitetty kokeellisesti, on esimerkissä mainittu myös kääntäjä, käyttöjärjestelmä, sekä muistiresurssien määrittämiseen käytetty komento. Ohjelmistokomponentin kehittämisessä on käytetty Java SDK –kehitysympäristön (Software Development Kit) version 1.2.2 sisältämää javac-kääntäjää. Käyttöjärjestelmä, jossa ohjelmistokomponentti on otettu käyttöön, on SunOS 5.7. Tämä sisältää myös ohjelmistokomponentin käyttöönoton sekä Java-virtuaalikoneen vaatimien muistiresurssien yhteismäärän selvittämiseen käytetyn `ps`-komennon (ks. esimerkiksi Kaare, 1988), jonka yhteydessä annettavien optioiden avulla voidaan vaikuttaa siihen, mitä tietoja prosesseista näytetään.

5.3 Yhteenveto

Tässä kappaleessa esitettiin ohjelmistokomponentin rajapintojen dokumentoimiseen soveltuva malli, joka perustuu enimmäkseen havaintoihin ohjelmistokomponentin uudelleenkäytön yhteydessä tarvittavista tiedoista, mutta myös uudelleenkäytön oppaaseen ja hyvän ohjelmistodokumentaation kriteereihin liittyviin asioihin. Dokumentointimallin rakenteen lisäksi kuvattiin sen sisältö.

Ohjelmistokomponentin rajapintojen dokumentointimalli jakaantuu yleiseen ja yksittäisen palvelun kuvaavaan osaan, joista jälkimmäinen kattaa yhden ohjelmistokomponentin toteuttaman operaation kuvauksen. Yleinen osa sisältää seuraavat tiedot:

- ohjelmistokomponentin nimi ja versio,
- ohjelmistokomponentin käyttöönoton vaatimien muistiresurssien määrä,
- herätteet, ja
- asynkroniset takaisinkutsut.

Ohjelmistokomponentin nimi ja versio ovat mukana selvyyden vuoksi. Herätteet ja asynkroniset takaisinkutsut kuvaavat ohjelmistokomponentin yleisen toiminnan. Herätteiden voidaan ajatella täydentävän ohjelmistokomponentin

toiminnan määrittelyä, sillä ne voidaan kuvata myös yksittäisten operaatioiden tasolla. Ohjelmistokomponentin tekemiä asynkronisia takaisinkutsuja ei voida liittää yksittäisten operaatioiden kuvausten yhteyteen, joten ne on esitettävä dokumentointimallin yleisessä osassa. Ohjelmistokomponentin käyttöönoton vaatimien muistiresurssien määrää voidaan pitää vaadittavien muistiresurssien vähimmäismääränä. Yksittäisen palvelun kuvaava osa pitää sisällään seuraavat tiedot:

- toteutettavan rajapinnan nimi,
- operaation nimi,
- operaation suoritusajan aikavaativuus,
- esiehdot,
- invariantit,
- sekvenssikaavio,
- jälkiehdot, ja
- poikkeukset.

Operaation suoritusajan aikavaativuutta lukuunottamatta muut yksittäisen palvelun kuvaavan osan tiedot liittyvät ohjelmistokomponentin toiminnan kuvaamiseen. Toteutettavan rajapinnan nimi on syytä mainita, sillä eri rajapinnat saattavat sisältää samannimisiä palveluita. Proseduraalisia rajapintoja ei ole tosin tapana nimetä erikseen, mutta ne eivät voine sisältää samannimisiä palveluita. Operaation nimen kohdalla kannattanee esittää operaation kutsumuoto mahdollisine parametreineen ja paluuarvoineen. Operaation suorituksen esiehdot voidaan ymmärtää seikkoina, jotka on syytä huomioida ennen operaation kutsumista. Niiden ei siis tarvitse välttämättä olla suoranaisia vaatimuksia sille, että operaatiota on ylipäätään mahdollista kutsua. Jälkiehdot ovat voimassa vain, mikäli operaation suoritus on onnistunut. Invariantit ovat sen sijaan voimassa aina. Sekvenssikaavion avulla voidaan kuvata operaation normaali suoritus mukaanlukien ohjelmistokomponentin tekemät ulkoiset – ja synkroniset takaisinkutsut. Operaation suorituksen aikana tapahtuneet poikkeukset

voidaan kuvata tarkemmin omassa kohdassaan. Tämän tarkemman kuvauksen yhteydessä on syytä esittää ainakin poikkeuksista aiheutuvat toimenpiteet.

Dokumentointimalli ei tarjoa merkittävää tukea ohjelmistokomponentin uudelleenkäyttöprosessin huomioivan dokumentaation tuottamiselle, eikä myöskään varsinaisia keinoja dokumentaation oikeellisuuden varmistamiselle. Dokumentointimalliin perustuvan ohjelmistokomponentin dokumentaation voidaan arvioida olevan kohtuullisen kattava kuvaus ohjelmistokomponentin toiminnallisuudesta, vaikkakin erityisesti toiminnan laadun kuvaamiseen voidaan liittää muitakin asioita kuin vaadittavat aika- ja muistiresurssit. Tämä on kuitenkin ristiriidassa dokumentointimallin yleisluontoisuuden kanssa, sillä kaikki ohjelmistokomponentit eivät esimerkiksi käytä tietoliikenneyhteyksiä. Vaatimus tällaisten resurssien määrittämisestä onkin jätetty dokumentointimallista pois. Dokumentointimallin käytettävyyttä voidaan arvioida dokumentaation tuottajan kannalta, kun taas dokumentointimalliin perustuvan dokumentaation käytettävyyttä voidaan arvioida ohjelmistokomponentin uudelleenkäyttäjän kannalta. Edellisen osalta tärkeää on havaita se, että dokumentointimalliin perustuvan ohjelmistokomponentin dokumentaation tuottaminen ei saa olla liian vaativa tehtävä. Jälkimmäisen kannalta ei voida arvioida dokumentointimallin käytettävyyttä, koska tämä ei ole suoranaisesti tekemisissä itse dokumentointimallin kanssa. Kuitenkin, mikäli esimerkiksi tietyn rajapinnan sisältämien palveluiden kuvaukset esitetään ohjelmistokomponentin dokumentaatiossa loogisesti yhteenkuuluvana joukkona, voidaan tämän ajatella parantavan kyseisen ohjelmistokomponentin dokumentaation käytettävyyttä. Dokumentointimalliin perustuvan ohjelmistokomponentin dokumentaation muokkaamisen tai täydentämisen voidaan ajatella olevan ainakin joissakin tapauksissa kohtalaisen helppoa, sillä esimerkiksi yksittäisten operaatioiden kuvaukset ovat toisistaan suhteellisen riippumattomia.

6 DOKUMENTOINTIMALLIN SOVELTAMINEN

Tässä kappaleessa sovelletaan edellisessä kappaleessa esitettyä ohjelmistokomponentin rajapintojen dokumentointimallia esimerkkinä käytettävän JavaBeans-ohjelmistokomponentin toiminnallisuuden dokumentoimiseksi. Tätä ennen kuvataan lyhyesti, mitä JavaBeans-ohjelmistokomponenttimallilla ja –ohjelmistokomponentilla tarkoitetaan, sekä esitellään dokumentointimallin soveltamisen yhteydessä esimerkkinä käytettävä JavaBeans-ohjelmistokomponentti. Lopuksi arvioidaan dokumentointimallin toimivuutta esimerkkinä käytetyn JavaBeans-ohjelmistokomponentin kannalta.

6.1 JavaBeans-määritys

JavaBeans-määritys (JavaBeans specification, Sun Microsystems, 1997) määrittelee, mitä JavaBeans-ohjelmistokomponenttimallilla ja –ohjelmistokomponentilla tarkoitetaan. Seuraavaksi esitetään näiden molempien määritelmät, ja kuvataan lyhyesti JavaBeans-ohjelmistokomponentin ominaisuuksia. Tämän jälkeen esitellään esimerkkinä käytettävä JavaBeans-ohjelmistokomponentti.

6.1.1 Ohjelmistokomponenttimalli ja ohjelmistokomponentti

JavaBeans on standardi, joka määrittelee Java-ohjelmointikielelle räätälöidyn, matalan tason ohjelmistokomponenttimallin (ks. Vanhelsuwé, 1997, s. 40). Ohjelmistokomponenttimallilla tarkoitetaan puolestaan sääntöjen ja rajapintojen muodostamaa joukkoa, joka määrää tähän kuuluvien ohjelmistokomponenttien vuorovaikutuksen (ks. Szyperski, 1997). Matalan tason ohjelmistokomponenttimalli tarkoittaa sitä, että JavaBeans-standardi ei määrittele esimerkiksi CORBA-arkkitehtuurin (ks. esimerkiksi Vinoski, 1997) tapaista sovelluskehystä (ks. Sun

Microsystems, 1997 tai Vanhelsuwé, 1997), joka tarjoaa laajan joukon korkean tason palveluita ohjelmistonkehittäjien käyttöön.

JavaBeans-ohjelmistokomponentti on uudelleenkäytettävä, laitteistoriippumaton ohjelmistokomponentti, jota on mahdollista käsitellä visuaalisesti kehitysvälineiden avulla (Sun Microsystems, 1997). Vanhelsuwé (1997) toteaa, että edellä mainitun määritelmän mukaan JavaBeans-ohjelmistokomponentti 'on tietoinen' sitä käsittelevistä kehitysvälineistä, ja on yhteensopiva näiden kanssa. Tämän tutkimuksen kannalta tärkeää on se, että JavaBeans-ohjelmistokomponentit ovat Szyperskin (1997) määritelmän mukaisia, suorituskelpoisia ohjelmistokomponentteja. Niiden dokumentoimiseen voidaan täten soveltaa edellisessä kappaleessa esitettyä ohjelmistokomponentin rajapintojen dokumentointimallia.

Tyypilliset JavaBeans-ohjelmistokomponenttien tukemat toiminnalliset piirteet ovat (ks. Sun Microsystems, 1997):

- Itsehavannointi (introspection): kehitysväline voi analysoida, kuinka JavaBeans-ohjelmistokomponentti toimii.
- Mukauttaminen (customization): käyttäjä voi kehitysvälineen avulla mukauttaa JavaBeans-ohjelmistokomponentin ulkomuotoa ja käyttäytymistä.
- Herätteet (events): yksinkertainen kommunikointimekanismi, jota voidaan käyttää JavaBeans-ohjelmistokomponenttien yhdistämiseen.
- Ominaisuudet (properties): näillä tarkoitetaan JavaBeans-ohjelmistokomponentin nimettyjä attribuutteja, jotka voidaan asettaa tai lukea kutsumalla asianmukaisia operaatioita. Näiden mukauttaminen onnistuu kehitysvälineen avulla, mutta niitä voidaan käsitellä myös ohjelmallisesti.
- Pysyvyys (persistence): kehitysvälineen avulla mukautetun JavaBeans-ohjelmistokomponentin tila voidaan tallettaa ja ladata myöhemmin uudelleen.

JavaBeans-ohjelmistokomponentit voivat olla sekä herätteiden lähettäjiä että niiden vastaanottajia (ks. Sun Microsystems, 1997 tai Szyperski, 1997). Yksit-

täinen JavaBeans-ohjelmistokomponentti voidaankin nähdä eri tilanteissa eri roolissa. Nimetyillä attribuuteilla tarkoitetaan sitä, että niihin päästään käsiksi tietyllä tavalla nimettyjen metodiparien kautta (pairs of `getter` and `setter` methods, ks. Szyperski, 1997). Tosin molemmat metodit tarvitaan vain, jos ominaisuuden tulee olla sekä luettavissa että asetettavissa.

Pelkästään luettavan ominaisuuden lukumetodista käytetään aina muotoa `public <propertyType> get<propertyName>()`, kun taas pelkästään asetettavan ominaisuuden kirjoitusmetodi noudattaa aina muotoa `public void set<propertyName>(<propertyType> nimi)` (ks. Sun Microsystems, 1997). Esimerkkinä sekä luettavasta että kirjoitettavasta JavaBeans-ohjelmistokomponentin ominaisuudesta voidaan käyttää vaikkapa pankkitilin rahamäärän kertovaa attribuuttia, jonka nimi on `saldo` ja tyyppi `Raha`. Kyseisen attribuutin luku- ja kirjoitusmetodit täytyy kirjoittaa seuraavasti: `public Raha getSaldo()` ja `public void setSaldo(Raha määrä)`. Attribuutti ei täytä ominaisuuksille asetettuja ehtoja, mikäli JavaBeans-ohjelmistokomponentti ei tarjoa edellä mainitun tapaisia metodeja kyseisen attribuutin asettamiseksi ja lukemiseksi. Ominaisuuksien tavoin JavaBeans-ohjelmistokomponentin voidaan katsoa toimivan tietyn herätteen lähteenä, mikäli se tarjoaa tietyllä tavalla nimetyt metodit, joiden avulla oliot tai toiset ohjelmistokomponentit voivat rekisteröityä kyseisen herätteen vastaanottajiksi sekä peruuttaa aiemmin suorittamansa rekisteröitymisen (ks. Sun Microsystems, 1997).

6.1.2 Esimerkkinä käytettävä JavaBeans-ohjelmistokomponentti

Vanhelsuwé (1997) esittää useita esimerkkejä JavaBeans-ohjelmistokomponenteista, joista yksi sisältää tarvittavan toiminnallisuuden hakemistopuun sisällön listaamiseksi. Kyseistä JavaBeans-ohjelmistokomponenttia käytetään myös tässä tutkimuksessa esimerkkinä. Vaikka se on toteutukseltaan suhteellisen yksinkertainen eikä tue monia JavaBeans-ohjelmistokomponenttien tyypil-

lisesti tukemia piirteitä (ks. Sun Microsystems, 1997), voidaan dokumentointimallia sen toiminnallisuuden dokumentoimiseksi kuitenkin soveltaa. Esimerkkinä käytettävästä JavaBeans-ohjelmistokomponentista käytetään jatkossa Vanhelsuwén (1997) tavoin nimitystä `DirLister`, sillä se on myös varsinaisen ohjelmistokomponentin toteutuksen tarjoavan luokan nimi. Nykyisen JavaBeans-määrityksen mukaan JavaBeans-ohjelmistokomponentti vastaa yksittäistä Java-ohjelmointikielen luokasta muodostettua oliota (Java object), mutta jatkossa määritykseen on tarkoitus lisätä tuki JavaBeans-ohjelmistokomponenteille, joiden toteutus perustuu joukkoon yhdessä toimivia olioita (ks. Sun Microsystems, 1997). Java-ohjelmistokomponentti on tosin tapana pakata erityiseen JAR-tiedostoon (Java ARchive, ks. Campione ja Walrath, 1998), joka voi sisältää itse JavaBeans-ohjelmistokomponentin lisäksi mm. tämän käyttämiä luokkia sekä erilaisia resurssitiedostoja, jotka voivat sisältää esimerkiksi kuvia tai ääntä (ks. Sun Microsystems, 1997). JAR-tiedostot ovat suorituskelpoisia (ks. Campione ja Walrath, 1998), joten JavaBeans-ohjelmistokomponentin jakeleminen ja käyttöönotto on helppoa, vaikka se hyödyntäisikin toisten olioiden tarjoamia palveluita tai erilaisia resurssitiedostoja toimintansa yhteydessä. Flanagan (1999) toteaa, että JAR-tiedostoja käytetään yleisesti JavaBeans-ohjelmistokomponenttien jakeluun.

`DirLister`-ohjelmistokomponentin toiminta voidaan kuvata seuraavasti (ks. Vanhelsuwé, 1997). `DirLister` sisältää toiminnallisuuden sille välitetyn hakemistonimen perusteella tapahtuvaan hakemiston ja tämän alihakemistojen sisältämien tiedostojen läpikäymiseen ja näiden täydellisten polkunimien lähettämiseen herätteinä rekisteröityneille vastaanottajille. Tällaista kaikki elementit huomioivaa iteratiivista logiikkaa (for all iterative logic) käytetään usein esimerkiksi tiedostojen etsimisen, jostakin tiedostosta etsittävän sanan, tai tiedostojen jollakin tavalla muuttamisen yhteydessä. `DirLister` tarjoaa uudelleenkäytettävän ratkaisun tällaisen toiminnan aikaansaamiseksi.

DirLister-ohjelmistokomponentin toteutuksen tarjoavan luokan lähdekoodi on kuvattu LIITTEESSÄ 1, joka sisältää myös FilenameActionEvent-luokan lähdekoodin. Jälkimmäinen edustaa DirLister-ohjelmistokomponentin lähettämien herätteiden parametrina välitettävien olioiden tyyppiä. Se perii ActionEvent-luokan, ja laajentaa tätä lisäämällä tiedoston nimen sisältävän attribuutin sekä tarvittavan saantimetodin (ks. LIITE 1). Koska FilenameActionEvent-luokka perii ActionEvent-luokan, ja on siten myös tyyppiä ActionEvent, voidaan siitä muodostettuja oliota lähettää ActionListener-rajapinnan toteuttaville rekisteröityneille vastaanottajille samaan tapaan, kuin ActionEvent-luokasta luotuja olioita. Vanhelsuwé (1997) toteaa, että aina kannattaa pyrkiä hyödyntämään jo olemassa olevia rajapintoja, sillä tällöin asiakkaiden ei tarvitse toteuttaa uusia rajapintoja. Mikäli FilenameActionEvent-luokka ei perisi ActionEvent-luokkaa, jouduttaisiin määrittelemään uusi rajapinta, jonka toteuttaville asiakkaille voitaisiin välittää FilenameActionEvent-luokasta luotuja olioita. Ajan myötä aina uusien rajapintojen määrittteleminen johtaa kuitenkin siihen, että sovelluksista tulee entistä laajempia ja vaikeaselkoisempia (ks. Vanhelsuwé, 1997).

Vaikka DirLister-ohjelmistokomponentin voidaankin ajatella olevan taustalla toimiva, käyttöliittymätön ohjelmistokomponentti, jonka palveluita oliot ja toiset ohjelmistokomponentit voivat käyttää, on Vanhelsuwé (1997) antanut sille myös graafisen käyttöliittymän (ks. KUVA 11), jonka seurauksena sen käyttöä voidaan helposti kokeilla. DirLister-ohjelmistokomponentin käyttöliittymä on varsin yksinkertainen; se sisältää vain kentän, johon käyttäjä voi kirjoittaa haluamansa hakemiston nimen, sekä tekstin, joka pyytää käyttäjää valitsemaan sopivan hakemiston, jonka nimi tulee kirjoittaa edellä mainittuun kenttään. KUVASSA 11 kyseiseen kenttään on kirjoitettu esimerkinomaisesti 'utilities/gui'. LIITTEESSÄ 2 on esitetty DirLister-ohjelmistokomponentin tuottama listaus, joka sisältää kaikkien edellä mainitussa hakemistossa olevien tiedostojen täydelliset polkunimet.

KUVA 11. `DirLister`-ohjelmistokomponentin käyttöliittymä

`DirLister`-ohjelmistokomponentin toiminta käynnistyy painettaessa return-näppäintä kentässä, johon tulee kirjoittaa hakemiston nimi (ks. KUVA 11). Tällöin se käy läpi sille välitetyn merkkijonon osoittaman hakemiston alihakemistoineen, ja lähettää herätteen jokaisen hakemistorakenteesta löydetyn tiedoston osalta. Herätteen parametrina välitettävä `FilenameActionEvent`-tyyppinen olio sisältää tiedoston täydellisen polkunimen, joten kukin vastaanottaja saa sen halutessaan selville kutsumalla kyseisen olion `getFilename()`-operaatiota (ks. LIITE 1).

6.2 Dokumentointimallin soveltaminen ja arviointi

Dokumentointimallin soveltamisessa rajoitutaan kokeilemaan sen käyttöä edellä esitellyn `DirLister`-ohjelmistokomponentin rajapintojen dokumentoimiseksi. Tämän jälkeen dokumentointimallia arvioidaan `DirLister`-ohjelmistokomponentin dokumentoinnin kannalta. Yleisesti ohjelmistokomponentin toiminnallisuuden kuvaavan dokumentaation tuottamisen näkökulmasta dokumentointimallia arvioitiin sen kuvaamisen yhteydessä kappaleessa 5.

6.2.1 Soveltaminen

Dokumentointimalliin perustuvan `DirLister`-ohjelmistokomponentin dokumentaatio on esitetty kokonaisuudessaan LIITTEESSÄ 3, joka sisältää tämän yleisen toiminnallisuuden sekä kaikkien julkisten operaatioiden kuvaukset. Tässä yhteydessä kyseistä dokumentaatiota ei käydä yksityiskohtaisesti läpi, sillä yksittäisten operaatioiden kuvaukset sisältävät monia asioita, joiden doku-

mentoinnissa käytetty tapa ei juurikaan poikkea eri operaatioiden kesken. Seuraavaksi selvennetään LIITTEESSÄ 3 esitettyä `DirLister`-ohjelmistokomponentin dokumentaatiota vain niiden asioiden osalta, joiden sisältö ei välttämättä selviä kappaleessa 5 kuvatun dokumentointimallin yhteydessä esitettyjen tietojen perusteella.

`DirLister`-ohjelmistokomponentin dokumentaation yleisessä osassa esitetyt tiedot eivät vaadi erityisiä perusteluita. Ohjelmistokomponentin nimi on sama kuin sen toteutuksen tarjoavan luokan nimi. Versionumeroa ohjelmistokomponentilla ei ole lainkaan, sillä Vanhelsuwé (1997) ei käytä versiointia esimerkkiensä yhteydessä. `DirLister`-ohjelmistokomponentin käyttöönoton vaatimien muistiresurssien määrä on selvitetty kokeellisesti, ja tämän vuoksi dokumentaatiossa on mainittu tiedot kääntäjästä, käyttöjärjestelmästä sekä siitä, kuinka muistiresurssien määrä on selvitetty. `DirLister` lähettää herätteitä vain löytämiensä tiedostojen osalta. Tämä on esitetty dokumentaation yleisen toiminnallisuuden kuvauksen lisäksi myös operaation `run()` kuvauksen yhteydessä. Asynkronisia takaisinkutsuja `DirLister` ei tee lainkaan.

`DirLister`-ohjelmistokomponentin käyttäjän kannalta tärkeitä operaatioita ovat vain `ActionListener`-rajapinnan `actionPerformed(ActionEvent actionEvent)`, `addActionListener(ActionListener listener)` ja `removeActionListener(ActionListener listener)`, sekä proseduraalisen rajapinnan `startListing(String path)` ja konstruktori `DirLister()`, sillä `DirLister`-ohjelmistokomponentin käyttö onnistuu näiden avulla. Muut julkiset operaatiot eli `Runnable`-rajapinnan `run()` ja proseduraalisen rajapinnan luokkametodi `setFrameCursor(Component component, Cursor pointerType)` ovat sen sijaan lähinnä `DirLister`-ohjelmistokomponentin käyttöön tarkoitettuja operaatioita. Niiden kuvaukset on kuitenkin esitetty LIITTEESSÄ 3, koska ne ovat julkisia operaatiota, ja siten `DirLister`-ohjelmistokomponentin asiakkaiden kutsuttavissa. Sen sijaan ope-

raatio `scanDir(String dir)` ei ole julkinen, joten sen kuvausta ei ole liitetty `DirLister`-ohjelmistokomponentin dokumentaatioon.

Seuraavat ovat yleisiä LIITTEESSÄ 3 esitettyihin operaatioiden kuvauksiin liittyviä huomioita. Proseduraalisen rajapinnan sisältämien operaatioiden toteutettavan rajapinnan nimen kohdalla on pelkkä viiva, koska proseduraalisia rajapintoja ei ole tapana nimetä erikseen. `DirLister`-ohjelmistokomponentin toteutuksessa ei ole hyödynnetty poikkeusten käsittelyä (exception handling, ks. Campione ja Walrath, 1998) lainkaan, joten dokumentaatiossa ei ole mainintaa poikkeuksista. Monille operaatioille ei myöskään ole olemassa mielekkäitä esiehtoja eikä invariantteja. Mikäli operaation paluuarvona on `void`, on tämä kuvattu sekvenssikaaviossa pelkän paluunuolen avulla. Muussa tapauksessa kyseisen nuolen yhteydessä on esitetty myös operaation paluuarvo tai tämän sisältävän muuttujan nimi. Yleisin operaation suoritusajan aikavaativuus on $\Omega(1)$, joka tarkoittaa sitä, että operaation suoritus vie vakioajan. Tällöin operaatio ei sisällä silmukoita tai rekursiivisia kutsuja.

Seuraavaksi kuvataan `DirLister`-ohjelmistokomponentin proseduraalisen rajapinnan julkiset operaatiot. Operaatio `DirLister()` on `DirLister`-luokan konstruktori. Sen tehtävänä on hoitaa tyypilliset luokan alustustoimenpiteet. Usein JavaBeans-ohjelmistokomponentin käyttöönotto ei kuitenkaan tapahdu sen konstruktorin avulla. Szyperski (1997) toteaa, että JAR-tiedosto voi sisältää JavaBeans-ohjelmistokomponentin sarjallistetun prototyypin (serialized prototype), jonka avulla tämä voidaan välittää alustetussa oletustilassa (initialized default form) asiakkaiden käytettäväksi. Tällöin JavaBeans-ohjelmistokomponentti voidaan ottaa käyttöön rekonstruoimalla sen sarjallistettu tila (deserialize), jolloin saadaan aikaiseksi kopio (Szyperski, 1997).

Operaation `startListing(String path)` tehtävänä on tarkistaa käyttäjän antaman merkkijonon oikeellisuus, ja tarvittaessa käynnistää herätteet lähettävä säie, joka suorittaa myös hakemistorakenteen läpikäymisen. Mielekkäänä ope-

raation kutsumisen esiehtona voidaan pitää sitä, että käyttäjä on antanut hakemiston polkunimen. Toisaalta `DirLister`-ohjelmistokomponentin palveluita voivat käyttää oliot tai toiset ohjelmistokomponentit, jolloin kyseinen esiehto ei pidä paikkaansa.

Operaatio `setFrameCursor(Component component, Cursor pointerType)` muuttaa käyttöliittymän kursorin tyypiksi sille parametrina välitetyn muuttuja `pointerType` tyypin, mikäli parametri `component` on tyyppiä `Frame`. Operaation suoritusajan aikavaativuus on $\Omega(n)$, koska operaatio sisältää `while`-silmukan, joka käy läpi parametrin `component` perintähierarkiaa selvittääkseen, onko tämä perinyt luokan `Frame`.

Seuraavaksi käydään läpi `DirLister`-ohjelmistokomponentin toteuttamien `ActionListener`- ja `Runnable`-rajapintojen sisältämien operaatioiden kuvaukset. Operaatiota `actionPerformed(ActionEvent actionEvent)` kutsutaan silloin, kun käyttäjä on antanut hakemiston nimen. Tällöin parametrina välitettävä olio ei ole tyyppiä `FilenameActionEvent`, joten ensimmäinen ehtolause ei pidä paikkaansa. Sen sijaan toinen ehtolause pitää paikkansa eli `actionEvent`-olion lähteenä on `TextField`. Tämä tarkoittaa sitä, että jokin `TextField`-luokan olio on lähettänyt `actionPerformed`-herätteen (ks. Campione ja Walrath, 1998). Tässä tapauksessa kutsutaan `startListing(String path)`-operaatiota, jonka parametriksi annetaan käyttöliittymän tekstikentässä oleva merkkijono. Lisäksi operaatio `run()` kutsuu operaatiota `actionPerformed(ActionEvent actionEvent)` jokaisen hakemistorakenteesta löydetyn tiedoston osalta. Tällöin parametrina välitettä olio on tyyppiä `FilenameActionEvent`, jonka seurauksena `DirLister` tekee ulkoisen kutsun eli kysyy parametrina välitettävältä oliolta sen sisältämän tiedoston nimen, ja tulostaa tämän.

Operaatioden `addActionListener(ActionListener listener)` ja `removeActionListener(ActionListener listener)` avulla `DirLister`-ohjelmistokomponentin asiakkaan on mahdollista rekisteröityä tämän lähettämien `actionPerformed`-herätteiden vastaanottajaksi tai poistaa itsensä rekisteröityneiden vastaanottajien joukosta. Kumpikaan toimenpide ei vaikuta muihin listalla oleviin herätteiden vastaanottajiin millään tavalla, joten tämä voidaan esittää invarianttina molempien operaatioiden kuvauksen yhteydessä.

Kun rajapinnan `runnable` toteuttavaa oliota käytetään säikeen luomiseen, aiheuttaa säikeen käynnistäminen kyseisen olion `run()`-operaation kutsumisen tuossa erikseen suoritettavassa säikeessä (Campione ja Walrath, 1998). Näin tapahtuu myös silloin, kun `DirLister`-ohjelmistokomponentin operaatiossa `startListing(String path)` käynnistetään herätteet lähettävä säie. Tässä erikseen suoritettavassa säikeessä käydään läpi käyttäjän antama hakemisto ja tämän alihakemistot, sekä lähetetään heräte jokaisen hakemistorakenteesta löydetyn tiedoston osalta. Oikeastaan hakemistorakenteen läpikäymisen suorittaa operaatio `scanDir(String dir)`, mutta koska tämä ei ole julkinen operaatio, on sen sisältämä toiminnallisuus esitetty sitä kutsuvan operaation `run()` kuvauksen yhteydessä. Operaation `run()` suoritusajan aikavaativuus on $\Omega(n^2)$, sillä sen kutsuma operaatio `scanDir(String dir)` sisältää `for`-silmukan, joka kutsuu operaatiota `scanDir(String dir)` rekursiivisesti. Sen sijaan operaation `run()` sisältämä `while`-silmukka ei vaikuta suoritusajan aikavaativuuteen korottavasti, sillä sen suoritus tapahtuu vasta operaation `scanDir(String dir)` palautumisen jälkeen, eikä sillä näin ollen ole merkitystä suoritusajan kasvun tyypin kannalta.

6.2.2 Arviointi

Esimerkkinä käytetyn `DirLister`-ohjelmistokomponentin dokumentoimisessa ei ollut suuria vaikeuksia. Yhtenä syynä tähän lienee se, että dokumentointimal-

lissa ei kiinnitetä tiukasti kuvattavien asioiden esitystapaa, vaan annetaan ohjelmistonkehittäjälle mahdollisuus dokumentoida kuvattavat asiat parhaaksi katsomallaan tavalla. Dokumentoitavien asioiden esittämisen suhteen noudatettiin kuitenkin samoja periaatteita kuin kappaleessa 5 dokumentointimallin esittämisen yhteydessä. Näiden periaatteiden voidaankin ajatella korvaavan formaalin esitystavan käyttämisen dokumentointimallissa kuvattavien asioiden osalta.

LIITTEESSÄ 3 esitetty dokumentaatio tarjoaa yhden kuvan `DirLister`-ohjelmistokomponentin toiminnallisuudesta. Koska dokumentointimallissa ei kiinnitetä asioiden esitystapaa, on sen käyttäjällä suuri merkitys lopputuloksena syntyvän dokumentaation kannalta. Näin ollen eri ohjelmistonkehittäjien tuottamien ohjelmistokomponentin dokumentaatioiden sisällöt luultavasti poikkeavat toisistaan jonkin verran.

Esitystavan valinnan lisäksi ohjelmistonkehittäjän valittavana on esimerkiksi se, kuinka tarkasti ohjelmistokomponentin toiminnallisuus kuvataan. `DirLister`-ohjelmistokomponentin toiminnallisuus on kuvattu varsin tarkasti, vaikka toisaalta esimerkiksi tämän attribuuttien ja sisäisten operaatioiden esittämistä pyrittiin välttämään, sillä ne eivät ole ohjelmistokomponentin uudelleenkäyttäjän kannalta tärkeitä tietoja. Sopivan tarkkustasoa mietittäessä kannattaakin pohtia, mitkä asiat ovat merkittäviä ohjelmistokomponentin uudelleenkäyttäjän kannalta, sekä mitä asioita tämän ei ole syytä tietää. Sopivan tarkkuustason määrittäminen on tärkeää, sillä ohjelmistokomponentin onnistunut uudelleenkäyttö riippuu suuressa määrin tästä.

Edellä esitettyä seikkaperäisempää arviota dokumentointimallin soveltamisesta on vaikea antaa, eikä yhden ohjelmistokomponentin dokumentoiminen toisaalta edes tarjoa kovin hyvää perustaa dokumentointimallin arvioimiselle. Esimerkkinä käytetyn `DirLister`-ohjelmistokomponentin dokumentoiminen voidaankin ymmärtää lähinnä dokumentointimallin soveltamisen kuvaamisena, ja

dokumentointimallin arvioiminen kyseisen tehtävän myötä syntyneiden ajatusten esittämisenä.

6.3 Yhteenveto

Tässä kappaleessa esitettiin, kuinka ohjelmistokomponentin rajapintojen dokumentointimallia voidaan soveltaa JavaBeans-ohjelmistokomponentin dokumentoimiseksi. JavaBeans-ohjelmistokomponenttimallin ja –ohjelmistokomponentin määritelmien lisäksi esiteltiin esimerkkinä käytettävä JavaBeans-ohjelmistokomponentti, joka sisältää toiminnallisuuden sille parametrina välitettävän hakemiston ja tämän alihakemistojen sisältämien tiedostojen polkunimien listaamiseksi. Dokumentointimallia sovellettiin edellä mainitun JavaBeans-ohjelmistokomponentin dokumentoimiseksi, sekä arvioitiin sen soveltuvuutta kyseiseen tehtävään.

JavaBeans-määrityksessä on esitetty JavaBeans-ohjelmistokomponenttimallin sekä –ohjelmistokomponentin määritelmät. Edellisellä tarkoitetaan standardia, joka määrittelee Java-ohjelmointikielelle räätälöidyn, matalan tason ohjelmistokomponenttimallin. Ohjelmistokomponenttimallilla tarkoitetaan puolestaan sääntöjen ja rajapintojen muodostamaa joukkoa, joka määrää tähän kuuluvien ohjelmistokomponenttien vuorovaikutuksen. Jälkimmäisellä tarkoitetaan uudelleenkäytettävää, laitteistoriippumatonta ohjelmistokomponenttia, jota on mahdollista käsitellä visuaalisesti kehitysvälineiden avulla. Tyypilliset JavaBeans-ohjelmistokomponenttien tukemat toiminnalliset piirteet ovat itsehavainnointi, mukauttaminen, herätteet, ominaisuudet, sekä pysyvyys.

Esimerkkinä käytetyn JavaBeans-ohjelmistokomponentin dokumentoiminen ei tuottanut ongelmia. Yhtenä syynä tähän voidaan pitää sitä, että dokumentointimallissa ei kiinnitetä tiukasti asioiden esitystapaa. Tämän seurauksena käyttäjällä voidaan ajatella olevan suuri merkitys lopputuloksena aikaan saatavan

dokumentaation kannalta. Dokumentointimalliin perustuvan dokumentaation tuottamisen yhteydessä on syytä kiinnittää erityistä huomiota siihen, kuinka tarkasti esitettävät asiat kuvataan.

7 YHTEENVETO

Tutkimuksen tavoitteena oli selvittää, kuinka uudelleenkäytettävien ohjelmistokomponenttien rajapinnat voidaan dokumentoida. Tutkimus oli luonteeltaan käsitteellisteoreettinen, ja sen kantavana ajatuksena oli se, että rajapintojen kuvausten voidaan ajatella olevan ohjelmistokomponentin ja tämän asiakkaiden välisiä sopimuksia. Tämä oli myös ohjelmistokomponentin rajapintojen dokumentointimallin kehittämisen lähtökohta. Kuvattavien asioiden selvittämisen lisäksi tutkimuksessa tarkasteltiin ohjelmistokomponentin uudelleenkäyttöprosessia, yleisesti ohjelmistojen dokumentointia, sekä hyvän ohjelmistodokumentaation kriteereitä. Näiden merkitystä pohdittiin mahdollisuuksien mukaan ohjelmistokomponentin rajapintojen dokumentointimallin sekä tähän perustuvan dokumentaation kannalta. Dokumentointimallia sovellettiin esimerkkinä käytetyn JavaBeans-ohjelmistokomponentin rajapintojen dokumentoimiseksi.

Tutkimuksen keskeiset käsitteet olivat ohjelmistokomponentti, rajapinta, ja uudelleenkäyttö. Suorituskelpoisten ohjelmistokomponenttien uudelleenkäytön yhteydessä tuli huomata, että yleisesti uudelleenkäytön tehtävänä mainittu ohjelmistokomponentin muuttaminen tuli ymmärtää ohjemistokomponentin mukauttamisena. Sopimuksenmukaisten rajapintojen tuli sisältää riittävästi tietoa ohjelmistokomponentin toiminnallisuudesta, joka voitiin ymmärtää ohjelmistokomponentin toimintana ja toiminnan laatuna. Toiminnan kuvaamisen osalta tuli huomioida ohjelmistokomponentin itsenäisen toiminnan lisäksi ohjelmistokomponenttien välinen vuorovaikutus. Ohjelmistokomponentin laatuun liittyi monia piirteitä, joista yleiset ohjelmistokomponentin vaatimat resurssit tuli kuvata osana sopimuksenmukaisia rajapintoja.

Perinteinen ohjelmistodokumentaatio ei sisältänyt ohjelmistokomponenttien uudelleenkäytön kannalta tarpeellisia tietoja, minkä vuoksi ohjelmistokompo-

nenttien dokumentoinnin yhdeksi osa-alueeksi on ehdotettu uudelleenkäytön dokumentaatiota. Tämä kannatti ymmärtää eksplisiittisenä ohjelmistokomponentin uudelleenkäytön oppaana. Hyvän ohjelmistodokumentaation kriteerit olivat oikeellisuus, kattavuus, käytettävyys, ja laajennettavuus.

Dokumentointimalli jakaantui ohjelmistokomponentin yleisen ja yksittäisen palvelun toiminnallisuuden kuvaukset sisältäviin osiin. Edellisessä esitettäviä asioita olivat ohjelmistokomponentin nimi ja versio, ohjelmistokomponentin käyttöönoton vaatimien muistiresurssien määrä, herätteet, ja asynkroniset takaisinkutsut. Jälkimmäisessä esitettäviä asioita olivat puolestaan toteutettavan rajapinnan nimi, operaation nimi, operaation suoritusajan aikavaativuus, esiehdot, invariantit, sekvenssikaavio, jälkiehdot, sekä poikkeukset. Dokumentointimalli ei tarjonnut merkittävää tukea ohjelmistokomponentin uudelleenkäyttöprosessin huomioivan dokumentaation tuottamiselle, eikä myöskään varsinaisia keinoja dokumentaation oikeellisuuden varmistamiselle. Dokumentointimalliin perustuvan ohjelmistokomponentin dokumentaation arvioitiin olevan kohtuullisen kattava kuvaus tämän toiminnallisuudesta, vaikka dokumentointimallin yleiskäyttöisyyden tavoitteen vuoksi tietyt asiat päätettiin jättää siitä pois. Dokumentointimallin soveltamisessa esimerkkinä käytetyn JavaBeans-ohjelmistokomponentin dokumentoimiseksi ei ollut suuria vaikeuksia. Tämä johtui ainakin siitä, että dokumentointimallissa ei kiinnitetty tiukasti esitettävien asioiden esitystapaa.

Tutkimuksen tärkeimpänä tuloksena oli malli uudelleenkäytettävien ohjelmistokomponenttien rajapintojen dokumentoimiseksi. Muita selkeästi eriteltäviä tuloksia tutkimuksessa ei dokumentointimallin soveltamisen yhteydessä tehtyjen huomioiden kirjaamista lukuunottamatta ollut. Tutkimus käsitteli suorituskelpoisia ohjelmistokomponentteja yleisesti eikä ottanut kantaa yksittäisiin ohjelmistokomponenttityyppeihin. Tämän seurauksena tulokset ovat yleistettävissä suorituskelpoisille ohjelmistokomponenteille. Mahdollinen jatkotutkimuksen aihe on dokumentointimallin käytön empiirinen tutkiminen.

LÄHTEET

Arthur J. D., Stevens K. T., Document Quality Indicators: A Framework for Assessing Documentation Adequacy, Software Maintenance: Research and Practice, Vol. 4, No. 3, 1992, 129 – 142

Banachowkski L., Kreczmar A., Rytter W., Analysis of Algorithms and Data Structures, Addison-Wesley, Reading, MA, 1991

Blokdijk A., Blokdijk P., Planning and Design of Information Systems, Academic Press, London, 1987

Boehm B. W., A Spiral Model of Software Development and Enhancement, IEEE Computer, Vol. 21, No. 5, 1988, 61 – 72

Booch G., Software Components with Ada: Structures, Tools, and Subsystems, Benjamin-Cummings, Redwood City, CA, 1987

Booch G., Rumbaugh J., Jacobson I., The Unified Modeling Language User Guide, Addison-Wesley, Reading, MA, 1998

Braun C. L., NATO Standard for the Development of Reusable Software Components, Vol. 1 (of 3 documents), 1994

Brockman R. J., The Why, Where, and How of Minimalism, In Proceedings of the Conference on SIGDOC '90, October 31 - November 2, 1990, Little Rock, AR USA 1990, ACM Press, New York, 1990, 111 – 119

Budd T., An Introduction to Object-Oriented Programming, Second Edition, Addison-Wesley, Reading, MA, 1997

Büchi M., Weck W., A Plea for Grey-Box Components, In Leavens G. T., Sitaraman M. (eds.) Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, September 26, 1997, 39 – 49

Campione M., Walrath K., The Java™ Tutorial Second Edition: Object-Oriented Programming for the Internet, Addison-Wesley, Reading, MA, 1998

Carroll J. M., The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill, MIT Press, Cambridge, MA, 1990

Carroll J. M., Rosson M. B., Paradox of the Active User, In Carroll J. M. (ed.) Interfacing Thought: Cognitive Aspects of Human-Computer Interaction, MIT Press, Cambridge, MA, 1987, 80 – 111

Chappell D., The Next Wave: Component Software Enters the Mainstream, Rational Software Corporation, 1997, [Viitattu 14.4.2000], Saatavilla www-muodossa:
<URL:http://www.rational.com/sitewide/support/whitepapers/dynamic.jtmpl?doc_key=354>

Collofello J. S., Bortman S., An Analysis of the Technical Information Necessary to Perform Effective Software Maintenance, In Proceedings of the Fifth Annual Phoenix Conference on Computers and Communications, March 1986, IEEE Computer Society Press, Washington, D.C., 1986, 420 – 424

Coplien J. O., Advanced C++ Programming Styles and Idioms, Addison-Wesley, Reading, MA, 1992

Cormen T. H., Leiserson C. E., Rivest R. L., Introduction to Algorithms, Second Edition, The MIT Press, Cambridge, MA, 1990

Cox B. J., Planning the Software Industrial Revolution, IEEE Software, Vol. 7, No. 6, 1990, 25 – 35

Dijkstra E. W., A Discipline of Programming, Prentice-Hall, Englewood Cliffs, NJ, 1976

D'Souza D. F., Wills A. C., Objects, Components, and Frameworks with UML : The Catalysis Approach, Addison-Wesley, Reading, MA, 1998

Fischer B., Snelting G., Reuse by Contract, In Leavens G. T., Sitaraman M. (eds.) Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, September 26, 1997, 91 – 100

Flanagan D., Java™ in a Nutshell, Third Edition, O'Reilly, Sebastopol, 1999

Forsell M., Päivärinta T., A Proactively Explicated Genre System for Documenting Reusable Software Components, Submitted to HICSS-34 (Hawaii International Conference on System Sciences), January 3 - 6, 2001

Franch X., The Convenience for a Notation to Express Non-Functional Characteristics of Software Components, In Leavens G. T., Sitaraman M. (eds.) Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, September 26, 1997, 101 – 110

Gamma E., Helm R., Johnson R., Vlissides J., Design Patterns: Elements of Reusable Object-Oriented Software, Fifth Edition, Addison-Wesley, Reading, MA, 1995

Gosling J., Joy B., Steele G., The Java™ Language Specification, Addison-Wesley, Reading, MA, 1996

Haikala I., Märijärvi J., Ohjelmistotuotanto, Fourth Edition, Suomen ATK-kustannus, Helsinki, 1997

Helm R., Holland I. M., Gangopadhyay D., Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, In Meyrowitz N. (ed.) Proceedings, OOPSLA/ECOOP 90, Ottawa, Canada, October 1990, ACM Sigplan Notices, Vol. 25, No. 10, 1990, 169 – 180

Hirvonen P., Ohjelmoinnin alkeet C-kieltä käyttäen, 3. uusittu painos, Jyväskylän yliopistopaino, Jyväskylä, 1995

Hondt K., Lucas C., Steyaert P., Reuse Contracts as Component Interface Descriptions, In Weck W., Bosch J., Szyperski C. (eds.) Proceedings of the Second International Workshop on Component-Oriented Programming (WCOP'97), Turku Centre for Computer Science, TUCS General Publication No. 5, September 1997

Hooper J. W., Chester R. O., Software Reuse: Guidelines and Methods, Plenum Press, New York, 1991

Horowitz E., Sahni S., Fundamentals of Computer Algorithms, Computer Science Press, USA, 1978

Horowitz E., Williamson R. C., SODOS: a Software Documentation Support Environment – It's Definition, IEEE Transactions on Software Engineering, SE-12, 1986a, 849 – 859

Horowitz E., Williamson R. C., SODOS: a Software Documentation Support Environment – It's Use, IEEE Transactions on Software Engineering, SE-12, 1986b, 1076 – 1087

Jaaksi A., Modelling the Behaviour of Object Systems with Event Traces, In Ege R., Singh M., Meyer B. (eds.) TOOLS 20, Technology of Object-Oriented Languages & Systems, Prentice Hall, 1996a

Jaaksi A., Type-safe Callbacks with Abstract Partners, In Dilip P., Yuan S. (eds.) Proceedings of the Object-Oriented Information Systems (OOIS '96) Conference, Springer, London, 1996b, 93 – 105

Jaaksi A., From Objects to Components, Presentation at the Component Based Software Engineering, April 30th 1998, Merito Forum, 1998, [Viitattu 24.2.2000], Saatavilla www-muodossa:
<URL:http://www.cs.uta.fi/~aj/omtpp/presentations/swcomponents_1 >

Jaaksi A., Aalto J-M., Aalto A., Vättö K., Tried & True Object Development : Industry-Proven Approaches with UML, Cambridge University Press, Cambridge, 1999

Jaaksi A., Laitkorpi M., Extending the Object-Oriented Software Development Process with Component-Oriented Design, Journal of Object-Oriented Programming, Vol. 12, No. 1, 1999, 41 – 50

Jacobson I., Object-Oriented Software Engineering, Addison-Wesley, Reading, MA, 1992

Jacobson I., Booch G., Rumbaugh J., The Unified Software Development Process, Addison-Wesley, Reading, MA, 1999

Kaare C., The UNIX Command Reference Guide, John Wiley & Sons, New York, 1988

Kain B. J., Software Components as Application Building Blocks, QUOIN, 1998, [Viitattu 5.4.2000], Saatavilla www-muodossa: <URL:http://www.quoininc.com/quoininc/ComponentsABB.html>

Karlsson E.-A., Software Reuse: A Holistic Approach, John Wiley & Sons, Chichester, 1995

Koskimies K., Pieni oliokirja, Gummerus Kirjapaino Oy, Jyväskylä, 1997

Krasner G. E., Pope S. T., A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80, Journal of Object-Oriented Programming, Vol. 1, No. 3, 26 – 49

Kruchten P. B., The 4+1 View Model of Architecture, IEEE Software, Vol. 12, No. 6, 1995, 42 – 50

Krueger C. W., Software Reuse, ACM Computing Surveys, Vol. 24, No. 2, 1992, 131 – 183

Lindholm T., Yellin F., The Java™ Virtual Machine Specification, Second Edition, Addison-Wesley, Reading, MA, 1999

McIlroy M. D., Mass Produced Software Components, In Naur P., Randell B., (eds.) Proceedings of NATO Conference on Software Engineering, Garmisch, Germany, October 1968, NATO Science Committee, Brussels, 1969

Meyer B., Applying 'Design by Contract', IEEE Computer, Vol. 25, No. 10, 1992, 40 – 51

Meyer B., Reusable Software: The Base Object-Oriented Component Libraries, Prentice-Hall, Englewood Cliffs, NJ, 1994

Murine G., Using Software Quality Metrics as a Tool for Independent Verification and Validation, In Proceedings of the Fifth Annual Phoenix Conference on Computers and Communications, March 1986, IEEE Computer Society Press, Washington, D.C., 1986, 433 – 437

Musser D. R., Saini A., STL Tutorial and Reference Guide, Addison-Wesley, Reading, MA, 1996

Mylopoulos J., Chung L., Nixon B. A., Representing and Using Nonfunctional Requirements: A Process-Oriented Approach, IEEE Transactions on Software Engineering, Vol. 18, No. 6, 1992, 483 – 497

Olafsson A., Doug B., On the Need for Required Interfaces of Components, In Mühlhäuser M. (ed.) Special Issues in Object-Oriented Programming – ECOOP96 Workshop Reader, dpunkt Verlag, Heidelberg, 1997, 159 – 165

Orfali R., Harkey D., Edwards J., The Essential Distributed Objects Survival Guide, John Wiley & Sons, New York, 1995

Parnas D. L., On the Criteria to Be Used in Decomposing Systems into Modules, Communications of the ACM, Vol. 15, No. 12, 1972, 1053 – 1058

Penttonen M., Johdatus algoritmien suunnitteluun ja analysointiin, Hakapaino Oy, Helsinki, 1997

Pomberger G., Software Engineering and Modula-2, Prentice-Hall, Englewood Cliffs, NJ, 1984

Pressman R. S., Software Engineering: A Practitioner's Approach, Third Edition, McGraw-Hill, London, 1994

Purdom P. W. Jr., Brown C. A., The Analysis of Algorithms, Holt, Rinehart and Winston, New York, 1985

Sametinger J., Reuse Documentation and Documentation Reuse, In Mitchell R., Nerson J.-M., Meyer B. (eds.) Proceedings of TOOLS 19: Technology of Object-Oriented Languages and Systems, Paris, France, 1996, Prentice-Hall, Englewood Cliffs, NJ, 1996

Sametinger J., Software Engineering with Reusable Components, Springer-Verlag, Town, 1997

Sneed H. M., Merey A., Automated Software Quality Assurance, IEEE Transactions on Software Engineering, SE-11, 1985, 909 – 916

Sommerville I., Software Engineering, Fifth Edition, Addison-Wesley, Reading, MA, 1995

Sprague R. H., Electronic Document Management: Challenges and Opportunities for Information Systems Managers, MIS Quarterly, Vol. 19, No. No. 1, 1995, 29 – 49

Staunstrup J., Interface Consistency, In Leavens G. T., Sitaraman M. (eds.) Proceedings of the First Workshop on the Foundations of Component-Based Systems, Zurich, Switzerland, September 26, 1997, 101 – 110

Steyaert P., Lucas C., Mens K., D'Hondt T., Reuse Contracts: Managing the Evolution of Reusable Assets, In Proceedings of OOPSLA'96, Conference on Object-Oriented Programming Systems, Languages and Applications, San Jose, CA, USA, October 6-10 1996, ACM SIGPLAN Notices, Vol. 31, No. 10, 1996, 268 – 285

Stroustrup B., The C++ Programming Language, Third Edition, Addison-Wesley, Reading, MA, 1997

Sun Microsystems, Hamilton G. (ed.), JavaBeans, Version 1.01, Sun Microsystems, 1997, [Viitattu 7.6.2000], Saatavilla www-muodossa: <URL:http://java.sun.com/beans/docs/spec.html>

Szyperski C., Component Software - Beyond Object-Oriented Programming, Addison-Wesley, Reading, MA, 1997

Thomas A., Enterprise JavaBeans™ Technology : Server Component Model for the Java™ Platform, Patricia Seybold Group, 1998, [Viitattu 2.3.2000], Saatavilla www-muodossa:
<URL: http://java.sun.com/products/ejb/white_paper.html >

Tracz W., Confessions of a Used Program Salesman – Institutionalizing Software Reuse, Addison-Wesley, Reading, MA, 1995

US Air Force, Acquisition Management: Software Maintainability – Evaluation Guide, AFOTEC Pamphlet 800-2, Vol. 3, 1987

US Army, Software Quality Engineering Handbook, Computer Systems Command, Ft. Belvoir VA, 1984

Vanhelsuwé L., Mastering JavaBeans, Sybex, London, 1997

Vinoski S., CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments, IEEE Communications, Vol. 35, No. 2, 1997, 46 – 55

Wegner P., Why Interaction Is More Powerful Than Algorithms, Communications of the ACM, Vol. 40, No. 5, 1997, 80 – 91

Wilkin L., Wulff W., Document Means More Than Manual: Document Design Outside the Computer Industry, Proceedings of the conference on SIGDOC '90, October 31 - November 2, 1990, Little Rock, AR USA, ACM Press, New York, 1990, 79 – 86

LIITTEET

LIITE 1. Luokkien `FilenameActionEvent` ja `DirLister` toteutukset

```
import java.awt.event.*;

public class FilenameActionEvent extends ActionEvent {

  protected String filename;

  //-------------------------------------------------------
  // FilenameActionEvent Constructor
  //-------------------------------------------------------
  public FilenameActionEvent(Object source,
                             String filename) {

    super(source, ACTION_PERFORMED, "Filename Action");
    this.filename = filename;

  }

  //-------------------------------------------------------
  // Filename accessor
  //-------------------------------------------------------
  public String getFilename() {

    return filename;

  }

} // End of Class FilenameActionEvent
```

```
import java.io.*;
import java.awt.*;
import java.awt.event.*;

import java.util.*;
import utilities.gui.*;

public class DirLister extends Panel
  implements ActionListener, Runnable {

  protected String filePath;
  protected Vector fileList;
  protected TextField pathTextField;
  protected ActionListener fNameListenerChain;
  protected static int treeDepth;

  //-------------------------------------------------------
  // DirLister Constructor
  //-------------------------------------------------------
  public DirLister() {

    Label prompt = new Label("Select Directory:");
    pathTextField = new TextField("", 40);
    pathTextField.addActionListener(this);
    add(prompt);
    add(pathTextField);
    fNameListenerChain = null;

  }

  //-------------------------------------------------------
  // DirLister Thread body
  //-------------------------------------------------------
  public void run() {

    fileList = new Vector();
    System.out.println("Scanning directory: "+ filePath);
    scanDir(filePath);

    // transmit all filenames to client bean
    Enumeration fileIterator = fileList.elements();

    while ( fileIterator.hasMoreElements() ) {

     String filename = (String) fileIterator.nextElement();
     fNameListenerChain.actionPerformed(
                 new FilenameActionEvent(this, filename) );
    }
```

```
    // finally, terminate list of filenames with a null
    // String
    fNameListenerChain.actionPerformed(
                        new FilenameActionEvent(this, null) );

    // re-enable GUI to allow user input.
    setFrameCursor(this,
       Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR) );

    pathTextField.setEnabled(true);
  }

  //-----------------------------------------------------
  // Recursive directory scanner
  //-----------------------------------------------------
  protected void scanDir (String dir) {

    String dirEntries[]; // list of dir entries (file|dir)
    String filename;
    File path, fpath;

    treeDepth++; // going down...
    path = new File(dir);

    // create list of files in this dir
    dirEntries = path.list();

    for (int i=0; i < dirEntries.length; i++) {

      filename = dir + File.separator + dirEntries[i];
      fpath = new File(filename);

      if ( fpath.isDirectory() ) {

        // recursively descend dir tree
        scanDir(fpath.getPath());
                                  //
      }
      else {
        fileList.addElement(fpath.getPath());
      }

    }

    treeDepth--; // and going up...
  }
```

```
//-------------------------------------------------------
// ActionListener interface method
//-------------------------------------------------------
public void actionPerformed(ActionEvent actionEvent) {

  Component source;

  // for Self-Test only:
  if ( actionEvent instanceof FilenameActionEvent ) {
    String filename = ((FilenameActionEvent)
                              actionEvent).getFilename();
    System.out.println("File: " + filename );
  }
  else {
    source = (Component) actionEvent.getSource();

    if ( source instanceof TextField ) {
      filePath = pathTextField.getText();
      startListing(filePath);
    }
  }

}

//-------------------------------------------------------
// Start the file listing process as a separate Thread.
//-------------------------------------------------------
public boolean startListing(String path) {

  File tentative;
  tentative = new File(path);

  if ( tentative.exists()) {

    if (tentative.isDirectory()) {

      // disable GUI to block user input until thread is
      // done.
      setFrameCursor(this,
        Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR) );

      pathTextField.setEnabled(false);

      // start up filename event firing thread.
      new Thread(this).start();

      return true;
    }
```

```
      else {
        DialogKit.ok( "'"+ path +"' Is not a directory.");
      }

    }
    else {
      DialogKit.ok( "'"+ path +"' Does not exist.");
    }

    return false;

  }

  //-----------------------------------------------------------
  // ActionListener management methods
  //-----------------------------------------------------------
  public void addActionListener(ActionListener listener) {

    fNameListenerChain = AWTEventMulticaster.add(
                              fNameListenerChain, listener);

    }

  public void removeActionListener(ActionListener listener)
  {

    fNameListenerChain = AWTEventMulticaster.remove(
                              fNameListenerChain, listener);

  }

  //-----------------------------------------------------------
  // Set this Component's parent Frame pointer to some
  // Cursor type.
  //-----------------------------------------------------------
  public static void setFrameCursor(Component component,
                                    Cursor pointerType) {

    Component parent;
    parent = component;

    while ( parent != null ) {

      if ( parent instanceof Frame ) {
        parent.setCursor(pointerType);
        break;
      }

      parent = parent.getParent();
```

```
    }
  }

  //-------------------------------------------------------
  // Self-test
  //-------------------------------------------------------
  public static void main (String[] args) {

    Frame window = new Frame("DirLister Self-Test");
    DirLister dirLister = new DirLister();

    // let DirLister dump own dir list
    dirLister.addActionListener(dirLister);

    window.add("Center", dirLister);
    window.pack();
    window.setVisible(true);

  }

} // End of Class DirLister
```

LIITE 2. `DirLister`-ohjelmistokomponentin tuottama listaus

```
Scanning directory: utilities/gui
File: utilities/gui/AngleDial.java
File: utilities/gui/CardDeck.java
File: utilities/gui/ChoiceKit.java
File: utilities/gui/Dial.java
File: utilities/gui/DialogKit.java
File: utilities/gui/GUIKit.java
File: utilities/gui/NDPDialDescriptor.java
File: utilities/gui/NumericDial.java
File: utilities/gui/NumericDialPanel.java
File: utilities/gui/SineParameters.java
File: utilities/gui/TabStrip.java
File: utilities/gui/AngleDial.class
File: utilities/gui/CardDeck.class
File: utilities/gui/ChoiceKit.class
File: utilities/gui/Dial.class
File: utilities/gui/DialogKit.class
File: utilities/gui/GUIKit.class
File: utilities/gui/NDPDialDescriptor.class
File: utilities/gui/NumericDial.class
```

```
File: utilities/gui/NumericDialPanel.class
File: utilities/gui/SineParameters.class
File: utilities/gui/TabStrip.class
File: null
```

LIITE 3. `DirLister`-ohjelmistokomponentin dokumentaatio

YLEISEN TOIMINNALLISUUDEN KUVAUS

Ohjelmistokomponentin nimi ja versio: `DirLister`

Ohjelmistokomponentin käyttöönoton vaatimien muistiresurssien määrä:

JVM mukaanlukien 10980 KB

Kääntäjä: Java SDK 1.2.2 –kehitysympäristöön kuuluva javac

Käyttöjärjestelmä: SunOS 5.7

Muistiresurssien määrä selvitetty käyttöjärjestelmän komennon `ps` avulla

Herätteet: `DirLister` lähettää `actionPerformed(FilenameActionEvent filenameActionEvent)`-herätteen kaikille `ActionListener`-rajapinnan toteuttaville rekisteröityneille vastaanottajille kaikkien sille parametrina välitetyn hakemiston ja tämän alihakemistojen sisältämien tiedostojen osalta.

Asynkroniset takaisinkutsut: –

YKSITTÄISTEN OPERAATIOIDEN KUVAUKSET

Toteutettavan rajapinnan nimi: –

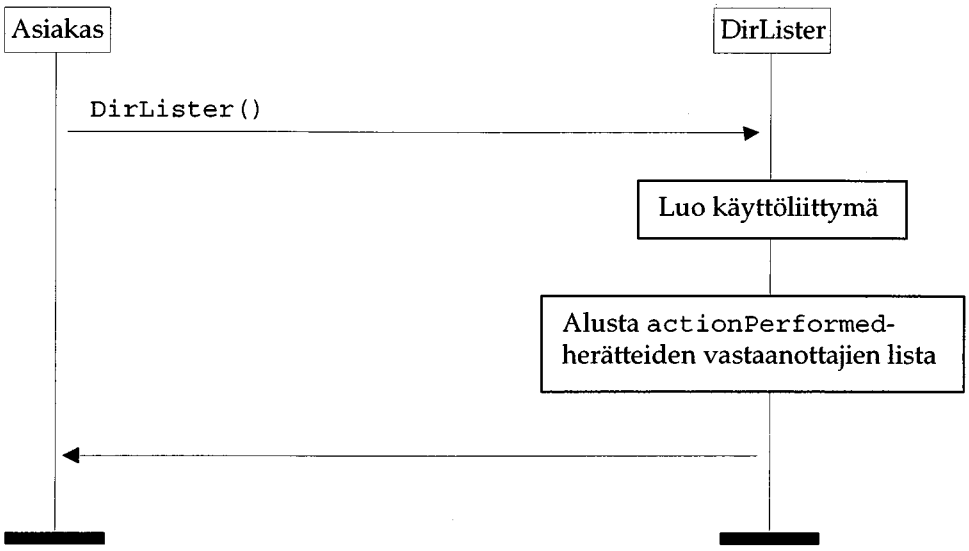
Operaation nimi: `DirLister()`

Operaation suoritusajan aikavaativuus: Ω(1)

Esiehdot: –

Invariantit: –

Sekvenssikaavio:



Jälkiehdot: Käyttöliittymä on luotu, vaikka sitä ei konstruktorin toimesta näytetäkään. Lisäksi `actionPerformed`-herätteiden vastaanottajien lista on alustettu.

Poikkeukset: –

Toteutettavan rajapinnan nimi: –

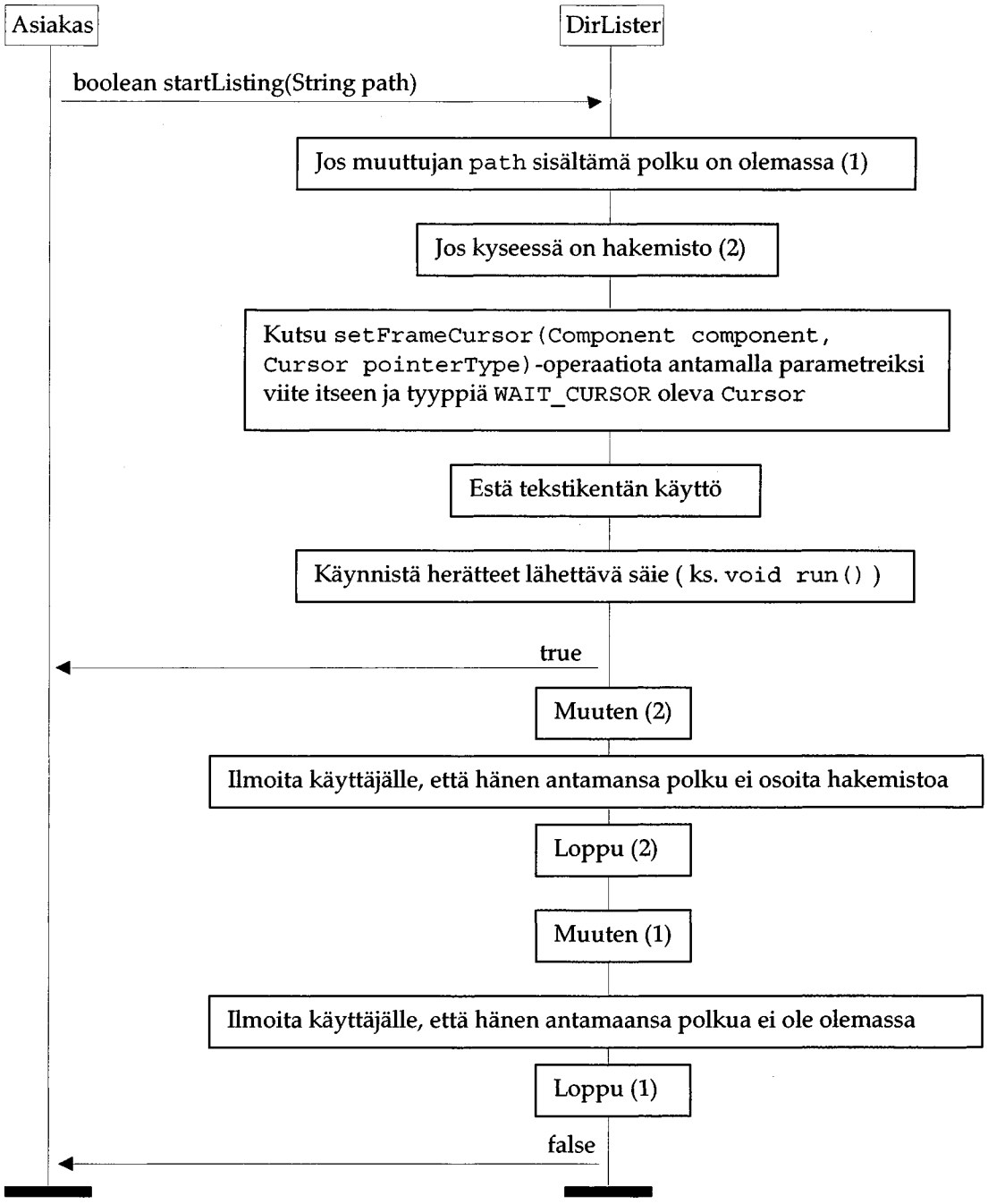
Operaation nimi: `boolean startListing(String path)`

Operaation suoritusajan aikavaativuus: Ω (1)

Esiehdot: Mikäli operaatiota ei kutsuta ulkopuolelta, on käyttäjän täytynyt kirjoittaa hakemiston polkunimi sille varattuun kenttään.

Invariantit: –

Sekvenssikaavio:



Jälkiehdot: Käyttöliittymän käyttö on estetty ja herätteet lähettävä säie on käynnistetty. Mikäli käyttäjän antama hakemistopolku on virheellinen, on käyttäjää informoitu asiasta.

Poikkeukset: –

Toteutettavan rajapinnan nimi: –

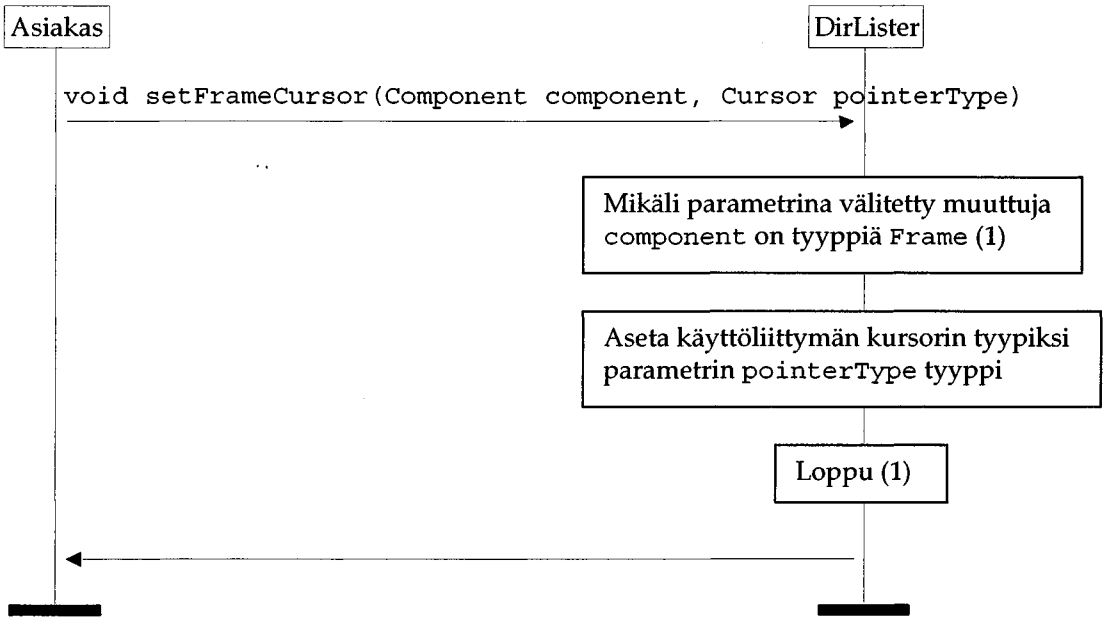
Operaation nimi: `static void setFrameCursor(Component component, Cursor pointerType)`

Operaation suoritusajan aikavaativuus: Ω (n)

Esiehdot: –

Invariantit: –

Sekvenssikaavio:



Jälkiehdot: Mikäli parametrina välitetyn muuttujan `component` tyyppinä on `Frame`, on käyttöliittymän kursorin tyypiksi asetettu parametrin `pointerType` tyyppi.

Poikkeukset: –

Toteutettavan rajapinnan nimi: `ActionListener`

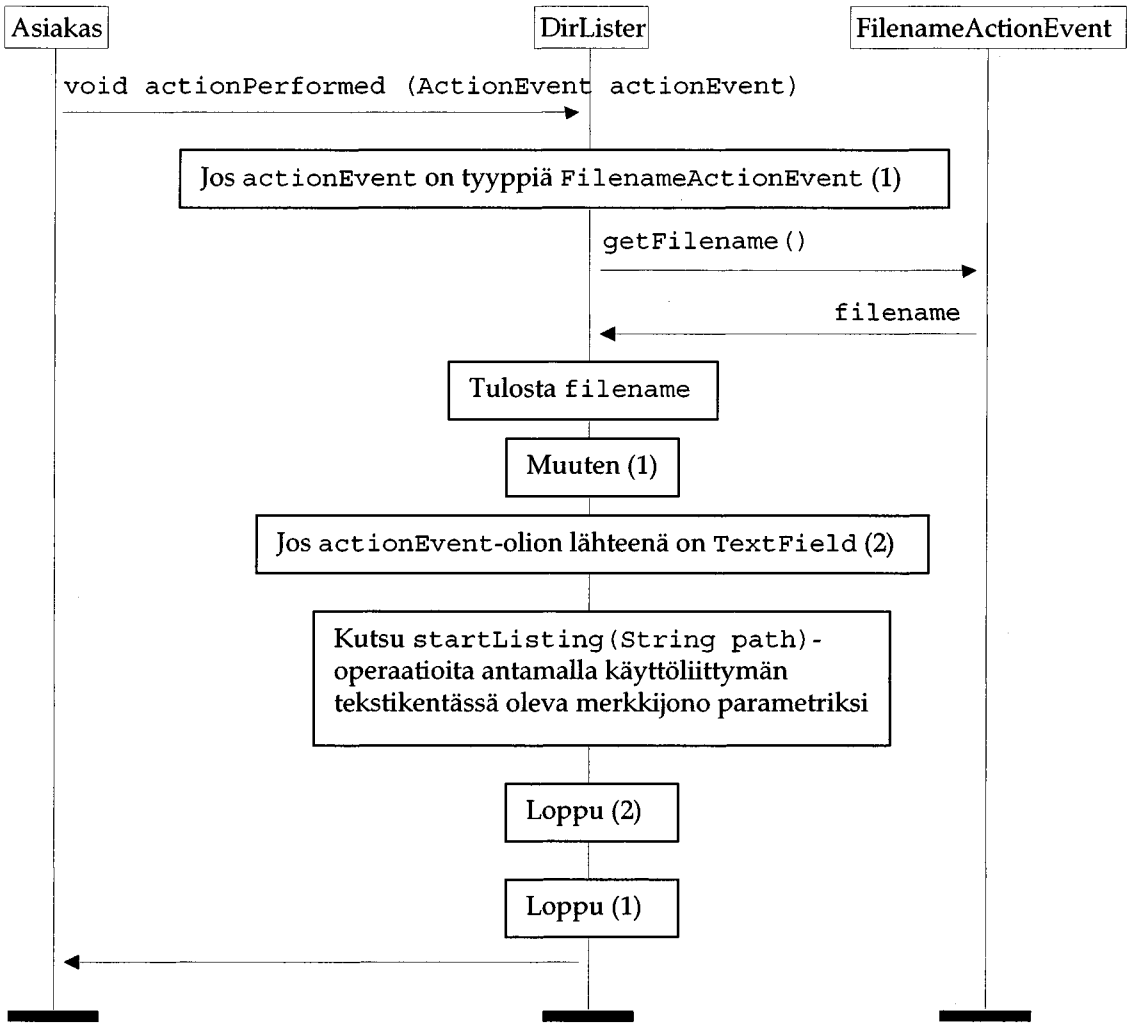
Operaation nimi: `void actionPerformed(ActionEvent actionEvent)`

Operaation suoritusajan aikavaativuus: Ω (1)

Esiehdot: –

Invariantit: –

Sekvenssikaavio:



Jälkiehdot: Muuttujan `filename` sisältämä tiedoston täydellinen polkunimi on tulostettu tai operaatiota `startListing(String path)` on kutsuttu antamalla käyttäjän kirjoittama merkkijono parametriksi.

Poikkeukset: –

Toteutettavan rajapinnan nimi: `ActionListener`

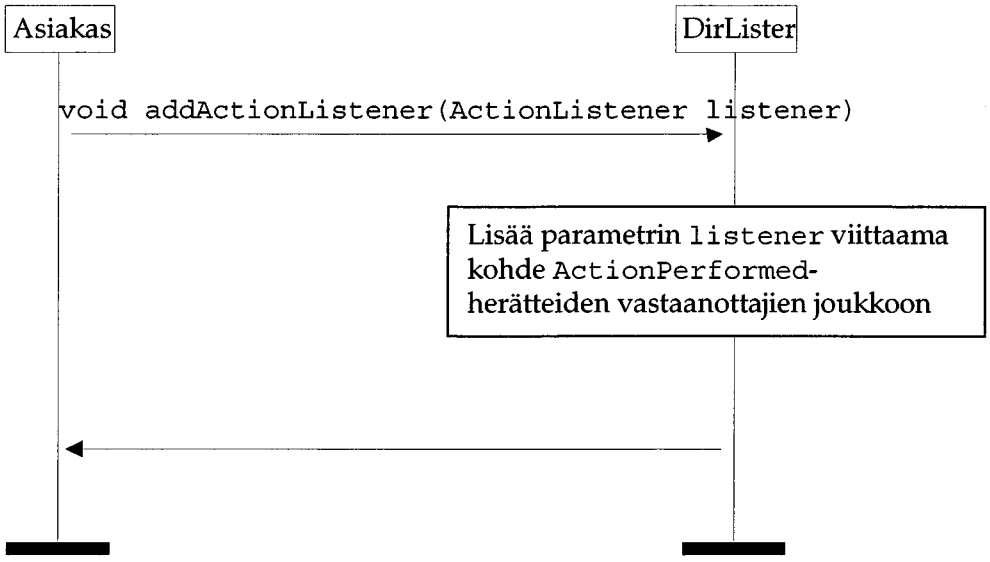
Operaation nimi: `void addActionListener(ActionListener listener)`

Operaation suoritusajan aikavaativuus: $\Omega(1)$

Esiehdot: –

Invariantit: Parametrin `listener` viittaaman kohteen lisääminen ei vaikuta listassa oleviin herätteiden vastaanottajiin millään tavalla.

Sekvenssikaavio:



Jälkiehdot: Parametrin `listener` viittaama kohde on lisätty `actionPerformed`-herätteiden vastaanottajaksi.

Poikkeukset: –

Toteutettavan rajapinnan nimi: `ActionListener`

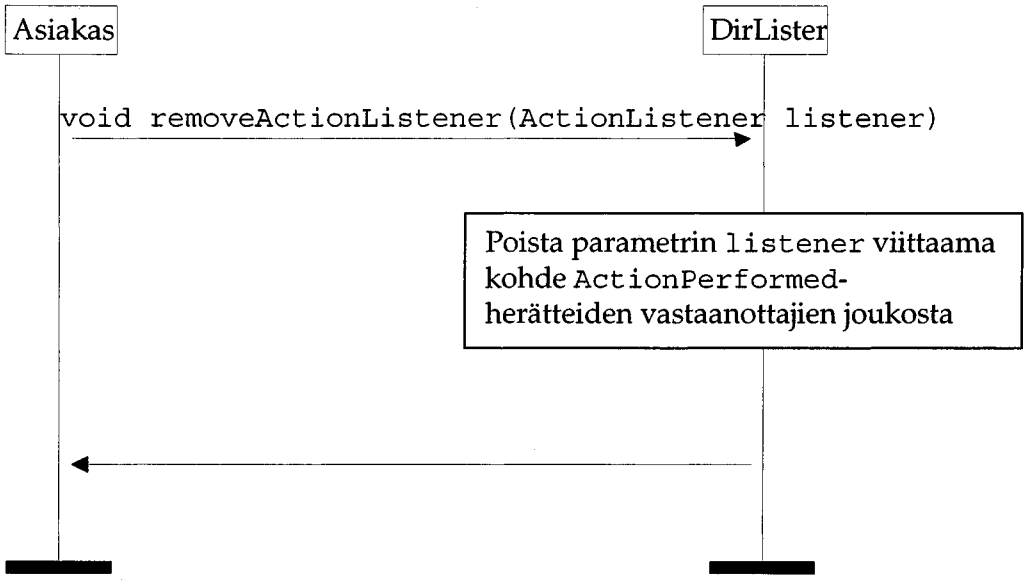
Operaation nimi: `void removeActionListener(ActionListener listener)`

Operaation suoritusajan aikavaativuus: $\Omega(1)$

Esiehdot: –

Invariantit: Parametrin `listener` viittaaman kohteen poistaminen ei vaikuta listassa oleviin herätteiden vastaanottajiin millään tavalla.

Sekvenssikaavio:



Jälkiehdot: Parametrin `listener` viittaama kohde on poistettu `actionPerformed`-herätteiden vastaanottajien joukosta.

Poikkeukset: –

Toteutettavan rajapinnan nimi: `Runnable`

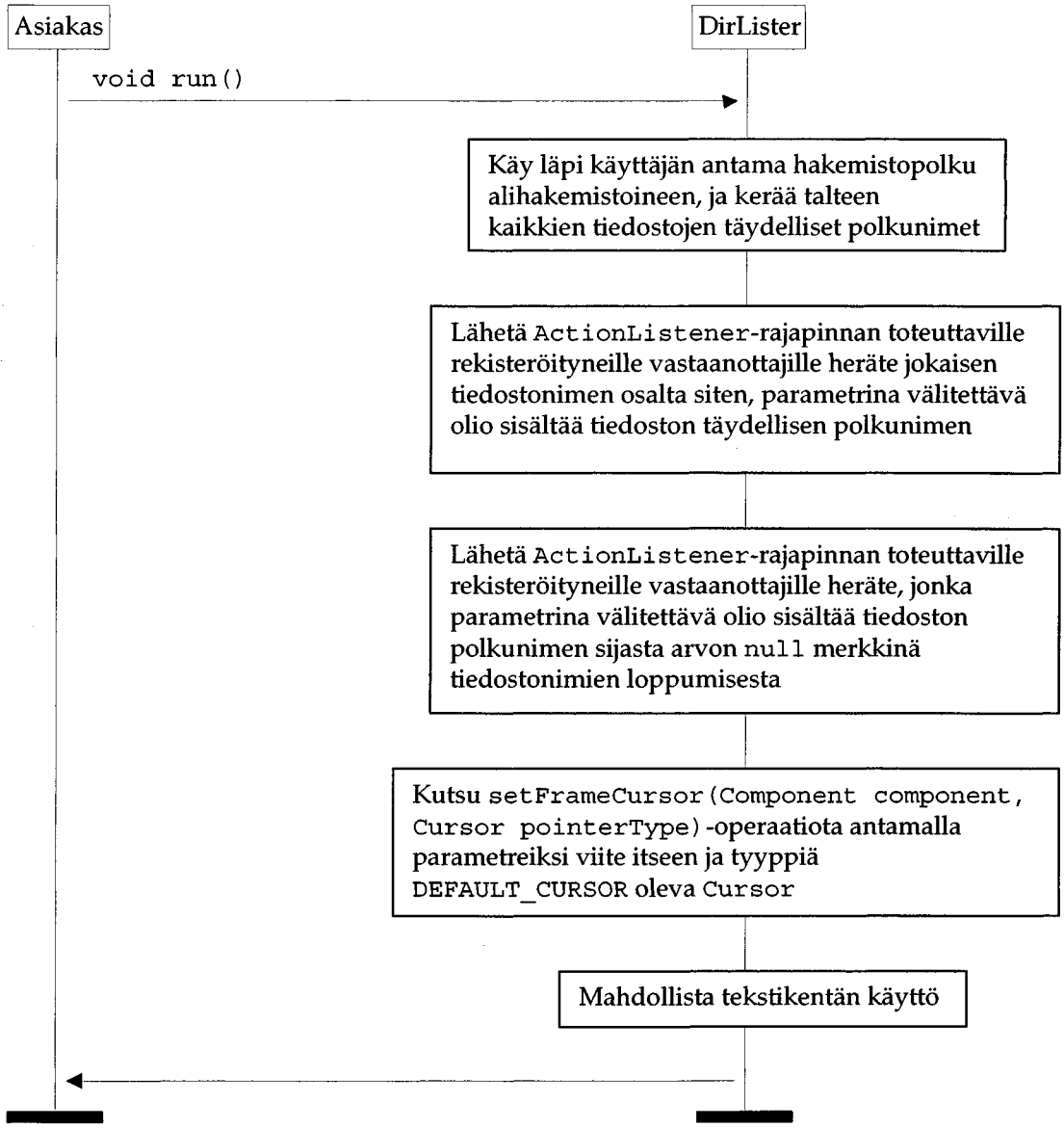
Operaation nimi: `void run()`

Operaation suoritusajan aikavaativuus: $\Omega(n^2)$

Esiehdot: Käyttäjän antama hakemistopolku on olemassa, ja kyseessä on hakemisto.

Invariantit: –

Sekvenssikaavio:



Jälkiehdot: Rajapinnan `ActionListener` toteuttaville rekisteröityneille vastaanottajille on lähetetty herätteiden muodossa tiedot kaikista käyttäjän antaman hakemiston ja tämän alihakemistojen sisältämistä tiedostoista täydellisine polkunimineen. Käyttöliittymän käyttö on mahdollistettu.

Poikkeukset: –