Aspektiohjelmointi ja hauraat liitoskohtamääritykset

Timo Tapanainen

Helsinki 25.9.2007

Pro gradu -tutkielma

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta/Osasto – Fakultet/Sektion – Faculty/Section	Laitos – Institution – Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä – Författare – Author
Timo Tapanainen

Työn nimi – Arbetets titel – Title
Aspektiohjelmointi ja hauraat liitoskohtamääritykset

Oppiaine – Läroämne – Subject
Tietojenkäsittelytiede

Työn laji – Arbetets art – Level	Aika – Datum – Month and year	Sivumäärä – Sidoantal – Number of pages
Pro gradu -tutkielma	25.9.2007	73

Tiivistelmä – Referat – Abstract

Läpileikkaava näkökulma on tietokoneohjelman toteutukseen liittyvä vaatimus, jota ei voida toteuttaa käytetyllä ohjelmointikielellä omaan ohjelmayksikköön, vaan sen toteutus hajaantuu useisiin ohjelmayksiköihin. Aspektiohjelmointi on uusi ohjelmointiparadigma, jolla läpileikkaava näkökulma voidaan toteuttaa omaan ohjelmayksikköön, aspektiin. Aspekti kapseloi näkökulman toteutuksen neuvon ja liitoskohtamäärityksen avulla. Neuvo sisältää näkökulman toteuttavan ohjelmakoodin ja liitoskohtamääritys valitsee ne ohjelman liitoskohdat, joihin ohjelmakoodi liitetään.

Nykyisillä aspektikielillä voidaan valita liitoskohtia pääasiassa niiden syntaktisten ominaisuuksien, kuten nimen ja sijainnin, perusteella. Syntaksiin sidoksissa olevat liitoskohtamääritykset ovat hauraita, sillä ohjelmaan tehdyt muutokset voivat rikkoa syntaksista riippuvia liitoskohtamäärityksiä, vaikka itse liitoskohtamäärityksiin ei tehtäisi muutoksia. Tätä ongelmaa kutsutaan hauraan liitoskohtamäärityksen ongelmaksi. Ongelma on merkittävä, koska hauraat liitoskohtamääritykset vaikeuttavat ohjelman kehitettävyyttä ja ylläpidettävyyttä.

Tässä tutkielmassa perehdytään hauraan liitoskohtamäärityksen ongelmaan ja siihen esitettyihin ratkaisuihin. Tutkielmassa näytetään, että ongelmaan ei ole tällä hetkellä kunnollista ratkaisua.

ACM Computing Classification System (CCS):
D.3.2 [Language Classifications]

Avainsanat – Nyckelord – Keywords
aspektiohjelmointi, AspectJ, hauraan liitoskohtamäärityksen ongelma

Säilytyspaikka – Förvaringställe – Where deposited
Kumpulan tiedekirjasto, sarjanumero C-

Muita tietoja – Övriga uppgifter – Additional information

Sisältö

1 Johdanto

Näkökulmien erottelu –periaate (separation of concern) on yksi ohjelmistotekniikan tärkeimmistä periaatteista [TeA00]. Se perustuu ajatukseen, että ongelma on helpompi ratkaista, kun se jaetaan pienempiin ja itsenäisesti ratkaistaviin osaongelmiin, joiden ratkaisut yhdistämällä saadaan alkuperäisen ongelman ratkaisu [Gyb01].

Tietokoneohjelman kehittäminen on ongelma, jossa kehitettävän ohjelman vaatimuksille etsitään ja toteutetaan ohjelmallinen ratkaisu. Näkökulmien erottelu -periaatteen mukaan tietokoneohjelmasta saadaan sekä laadukkaampi että helpommin kehitettävä, jos ohjelmaan liittyvän *näkökulman* (concern), kuten vaatimuksen, ominaisuuden tai käsitteen, ratkaisu voidaan laatia suunnittelussa ja toteutuksessa muita näkökulmia huomioimatta [TeA00]. Tämä vähentää sekä ohjelman kehittämiseen liittyvää monimutkaisuutta että parantaa ohjelman ymmärrettävyyttä, ylläpidettävyyttä ja uudelleenkäytettävyyttä, koska näkökulman toteutus on yhdessä paikassa muista näkökulmista erotettuna.

Näkökulmien erottelu –periaatteen soveltaminen edellyttää, että suunnittelussa erotetut näkökulmat voidaan erottaa myös toteutustasolla [LoH95]. Suunnittelu osittaa ohjelman näkökulmien perusteella pieniksi osiksi, ja ohjelmointikieli tarjoaa sekä moduulit että mekanismit, joilla suunnitellut osat toteutetaan ja yhdistetään kokonaiseksi ohjelmaksi [KLM97]. Suunnittelu ja ohjelmointikieli toimivat hyvin yhteen, jos ohjelmointikieli tarjoaa moduulit ja mekanismit, jotka tukevat suunnittelun ositusta.

Kaikkia suunnittelussa erotettuja näkökulmia ei voida toteuttaa nykyisillä ohjelmointikielillä omaan moduuliin, vaan näkökulman toteutus *hajaantuu* (scattering) useisiin moduuleihin, joissa se *sotkeutuu* (tangling) moduulin alkuperäisen näkökulman toteutuksen kanssa [ShP05]. Näitä hajaantuvia ja sotkeutuvia näkökulmia nimitetään *läpileikkaaviksi näkökulmiksi* (crosscutting concern). Yleensä läpileikkaavia näkökulmia ovat esimerkiksi tietoturva, pysyvyys, samanaikaisuuden hallinta ja lokikirjaus.

Aspektiohjelmointi (aspect-oriented programming) on uusi ohjelmointiparadigma, jonka tarjoamalla uudella ositusyksiköllä, *aspektilla* (aspect), ohjelman läpileikkaavat näkökulmat voidaan kapseloida [KLM97]. Aspektiohjelmointia käytetään muiden ohjelmointiparadigmojen rinnalla. Ohjelma toteutetaan ensin läpileikkaavia näkökulmia lukuun ottamatta perinteisellä ohjelmointikielellä, minkä jälkeen läpileikkaavat näkökulmat toteutetaan aspekteihin aspektikielellä. Ohjelma koostuu siten perinteisellä ohjelmointikielellä toteutetusta osasta eli *perusohjelmasta* (base program) [Lam99] ja aspekteista.

Aspekti kapseloi läpileikkaavan näkökulman *neuvon* (advice) ja *liitoskohtamäärityksen* (pointcut) avulla [KLM97]. Neuvo sisältää läpileikkaavan näkökulman toteutuksen, ja liitoskohtamääritys poimii ne perusohjelman kohdat, joihin neuvon sisältämä toiminnallisuus liitetään. Liitoskohtamääritys voi poimia ainoastaan aspektikielen sallimia kohtia, *liitoskohtia* (join point). Aspektikielen tukemia liitoskohtia voivat olla esimerkiksi metodikutsut ja metodien suoritukset.

Perusohjelman kehittäjän ei tarvitse valmistella perusohjelman ohjelmakoodia siihen vaikuttavia aspekteja varten, koska liitoskohtamäärityksillä ja neuvoilla voidaan lisätä toiminnallisuutta perusohjelmaan sitä muuttamatta. Aspektien toiminnalliset lisäykset eivät myöskään näy perusohjelman ohjelmakoodissa. Perusohjelman kehittäjä ei ole siten tietoinen aspekteista, minkä vuoksi tätä aspektiohjelmoinnille luonteenomaista piirrettä kutsutaan *tietämättömyydeksi* (obliviousness) [FiF01]. Tietämättömyys on yksi aspektiohjelmoinnin merkittävimmistä eduista, sillä näin perusohjelman näkökulmat ovat aidosti erillään aspektien kapseloimista näkökulmista.

Liitoskohtamääritykset poimivat liitoskohtia pääasiassa liitoskohtiin liittyvien nimien perusteella. Esimerkiksi metodikutsu poimitaan kutsutun metodin allekirjoituksen perusteella. Jotta liitoskohtamäärityksessä ei tarvitsisi nimetä jokaista poimittavaa liitoskohtaa, liitoskohdat voidaan poimia *allekirjoitushahmolla* (signature pattern), joka poimii kaikki hahmon mukaiset liitoskohdat. Tarkastellaan esimerkkinä liitoskohtamääritystä, jonka tulisi poimia kaikki henkilöä mallintavan luokan metodit, jotka muuttavat henkilön tilaa. Kuvitellaan, että kaikki henkilön tilaa muuttavat metodit alkavat nimellä ”aseta”, kuten `asetaNimi`, `asetaIkä` ja `asetaHetu`. Näiden poimimiseen riittää tällöin allekirjoitushahmoon perustuva liitoskohtamääritys, joka poimii kaikki aseta-alkuiset metodit.

Perusohjelman rakenteiden nimiin viittaavat liitoskohtamääritykset ovat tiukasti sidoksissa perusohjelmaan, minkä vuoksi perusohjelman muutokset voivat edellyttää muutoksia myös liitoskohtamäärityksiin. Tarkastellaan esimerkiksi edellisen kappaleen henkilöä mallintavaa luokkaa ja sen tilaa muuttavia metodeja poimivaa liitoskohtamääritystä. Luokkaan lisätään liitoskohtamäärityksen laatimisen jälkeen henkilön nimen vaihtava `vaihdaNimi`-metodi ja `asetaTila(Henkilö h)`-metodi, joka kopioi henkilön tilan parametrina annettuun henkilöön. Ensin mainittu metodi muuttaa olion tilaa, mutta ”aseta”-alkuisia metodeja poimiva liitoskohtamääritys ei poimi sitä. Liitoskohtamääritys poimii taas toisena mainitun metodin, mutta tämä metodi ei muuta olion tilaa. Liitoskohtamääritykseen on siten tehtävä muutoksia: ensimmäisenä mainittu metodi on lisättävä poimittavien metodien joukkoon ja toisena mainittu metodi on poistettava sieltä.

Liitoskohtamääritysten ja perusohjelman välinen tiukka sidos tekee liitoskohdista hauraita perusohjelman muutoksille: perusohjelman muutokset voivat rikkoa

liitoskohtamääritysten merkityksen, vaikka itse liitoskohtamääritystä ei muuteta [KoS04]. Tätä ongelmaa nimitetään *hauraan liitoskohtamäärityksen ongelmaksi* (fragile pointcut problem). Ongelmaa vaikeuttaa se, että perusohjelman kehittäjä ei tunne perusohjelmaan vaikuttavia aspekteja, jolloin hän voi tietämättään muutoksilla rikkoa liitoskohtamäärityksiä.

Hauraan liitoskohtamäärityksen ongelma on paitsi periaatteellisesti myös käytännön tasolla merkittävä, koska ongelma koskettaa kaikkia laajasti käytettyjä aspektikieliä. Näihin lukeutuvat muun muassa AspectJ [KHH01], JBoss AOP [JBo07] ja Spring-kehykseen sisältyvä Spring AOP [Spr07].

Tässä tutkielmassa perehdytään hauraan liitoskohtamäärityksen ongelmaan ja esitetään siihen erilaisia ratkaisuvaihtoehtoja. Luvussa 2 esitellään näkökulmien erottelu -periaate, joka on keskeisessä asemassa aspektiperustaisen ohjelmoinnin ymmärtämisessä. Luvussa 3 perehdytään aspektiohjelmointiin ja sen käsitteistöön. Neljännessä luvussa esitellään AspectJ-aspektikieli [KHH01], joka on tällä hetkellä kehittynein, koetelluin ja eniten käytetty aspektikieli [YBK04]. Luvussa 5 perehdytään hauraan liitoskohtamäärityksen ongelmaan ja luvussa 6 tarkastellaan ongelman eri ratkaisuvaihtoehtoja. Luku 7 on yhteenveto.

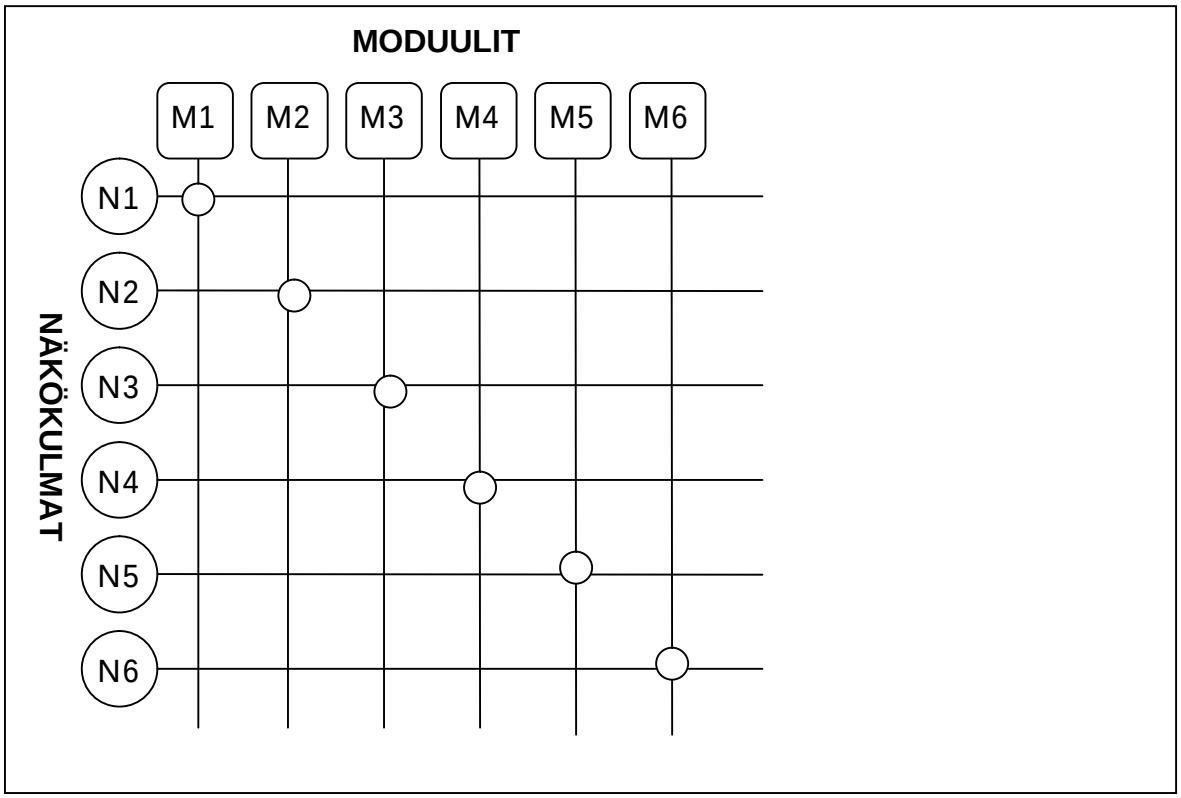
2 Näkökulmien erottelu –periaate

Näkökulmien erottelu -periaate on yleinen ongelmanratkaisuperiaate [Dij82]. Sen mukaan ongelman ratkaisemiseen liittyy erilaisia näkökulmia. Ongelman ratkaiseminen on periaatteen mukaan helpompaa, jos ongelma voidaan ratkaista yhdestä näkökulmasta kerrallaan muita näkökulmia huomioimatta. Periaatteen soveltaminen edellyttää, että ongelma voidaan osittaa näkökulmien perusteella itsenäisesti ratkaistaviksi osaongelmiksi, joiden ratkaisut yhdistämällä saadaan varsinaisen ongelman ratkaisu. Periaatteen isänä pidetty E. Dijkstra, pitää näkökulmien erottelu -periaatetta ainoana tehokkaana tapana vähentää ongelmaan liittyvää monimutkaisuutta.

Näkökulmien erottelu –periaatetta voidaan soveltaa tietokoneohjelman kehittämisessä. Tietokoneohjelman kehittäminen on ongelma, jossa ohjelman vaatimuksille etsitään ja toteutetaan ohjelmallinen ratkaisu [TeA00]. Tietokoneohjelman kehittämiseen liittyy erilaisia näkökulmia, jotka vaihtelevat kehittäjän roolin, ajan ja tehtävän mukaan [OsT01]. Esimerkiksi ohjelmistoarkkitehdille tärkeitä näkökulmia voivat olla ohjelman pysyvyys, tietoturva, poikkeuskäsittely ja hajautus. Ohjelmistosuunnittelijalle tärkeä näkökulma voi olla jokin toiminnallinen vaatimus, ominaisuus tai käsittelysääntö. Näkökulma on siten jokin ongelma-alueen kokonaisuus, kuten vaatimus, ominaisuus, käsite tai käsittelysääntö [Gyb03], johon haetaan ohjelmallista ratkaisua.

Näkökulmien erottelu -periaatteen hyödyt saavutetaan tietokoneohjelman kehittämisessä, jos ohjelman näkökulmien kehittäminen voidaan jakaa itsenäisesti suunniteltaviin ja toteutettaviin moduuleihin, jotka yhdistämällä syntyy valmis ohjelma. Periaatetta noudattaen toteutetussa ohjelmassa jokainen näkökulma on toteutettu omaan moduuliin, joka ei sisällä muiden näkökulmien toteutusta.

Kuvassa 2.1 on esitetty näkökulmien erottelu –periaatteen mukaisesti toteutettu ohjelma. Ohjelma sisältää näkökulmat N1-N6 ja moduulit M1-M6, jotka on esitetty kuvassa vastaavasti pysty- ja vaaka-akselilla. Ohjelman näkökulmat on toteutettu moduuleihin siten, että näkökulma N1 on toteutettu moduuliin M1, näkökulma N2 on toteutettu moduuliin M2 ja niin edelleen. Näin ohjelman jokainen näkökulma on toteutettu vain yhteen moduuliin, joka ei sisällä muiden näkökulmien toteutusta. Näkökulma ja sen toteuttava moduuli on esitetty kuvassa näkökulman vaaka-akselin ja moduulin pystyakselin leikkauskohdassa olevalla ympyrällä. Kuvasta nähdään, että jokainen moduuli sisältää vain yhden näkökulman toteutuksen.



Kuva 2.1: Näkökulmien erottelu –periaatteen mukaisesti ositettu ohjelma.

2.1 Näkökulmien erottelu parantaa ohjelman laatua

Näkökulmien erottelu –periaatetta noudattamalla helpotetaan ohjelman kehittämistä ja parannetaan kehitettävän ohjelman laatua, kuten ymmärrettävyyttä, ylläpidettävyyttä ja uudelleenkäytettävyyttä [TeA00]. Nämä hyödyt syntyvät ohjelman näkökulmiin perustuvasta modularisoinnista.

Ohjelma on helpompi kehittää, koska ohjelman kehittäjä voi suunnitella ja toteuttaa ratkaisun yhteen näkökulmaan muita näkökulmia huomioimatta. Esimerkiksi ohjelmistosuunnittelija voi suunnitella ja toteuttaa toiminnallisen vaatimuksen toteuttavan käyttöliittymän, palvelun ja käsitteet teknisiä näkökulmia, kuten pysyvyyttä, tietoturvaa ja poikkeuskäsittelyä huomioimatta. Samoin ohjelmistoarkkitehti voi laatia ratkaisun pysyvyyteen hajautusta, tietoturvaa, poikkeuskäsittelyä ja toiminnallisia näkökulmia huomioimatta.

Näkökulmien ja moduulien välinen yksikäsitteinen vastaavuus edistää erityisesti ohjelman ymmärrettävyyttä ja sen kautta myös ohjelman ylläpidettävyyttä. Ohjelman toiminnan ymmärtäminen edellyttää ongelma-alueen käsitteistön tuntemusta, ja käsitteiden toteutus on osattava paikantaa ohjelmakoodista [Raw02]. Yleensä ongelma-alueen käsitteistö tunnetaan, mutta käsitteiden toteutuksen sijaintia ei tunneta. Tämä on yleinen tilanne ohjelman ylläpitäjillä, joilla suurin osa muutospyyntöjen toteutukseen kuluvasta ajasta menee käsitteiden toteutuksen paikantamiseen [BeR00]. Näkökulmien erottelu –periaate edistää käsitteiden paikantamista, koska näkökulman toteutus sijaitsee vain yhdessä

moduulissa. Koska moduuli ei sisällä muiden näkökulmien toteutusta, sen toimintaa on helpompi ymmärtää ja ylläpitää.

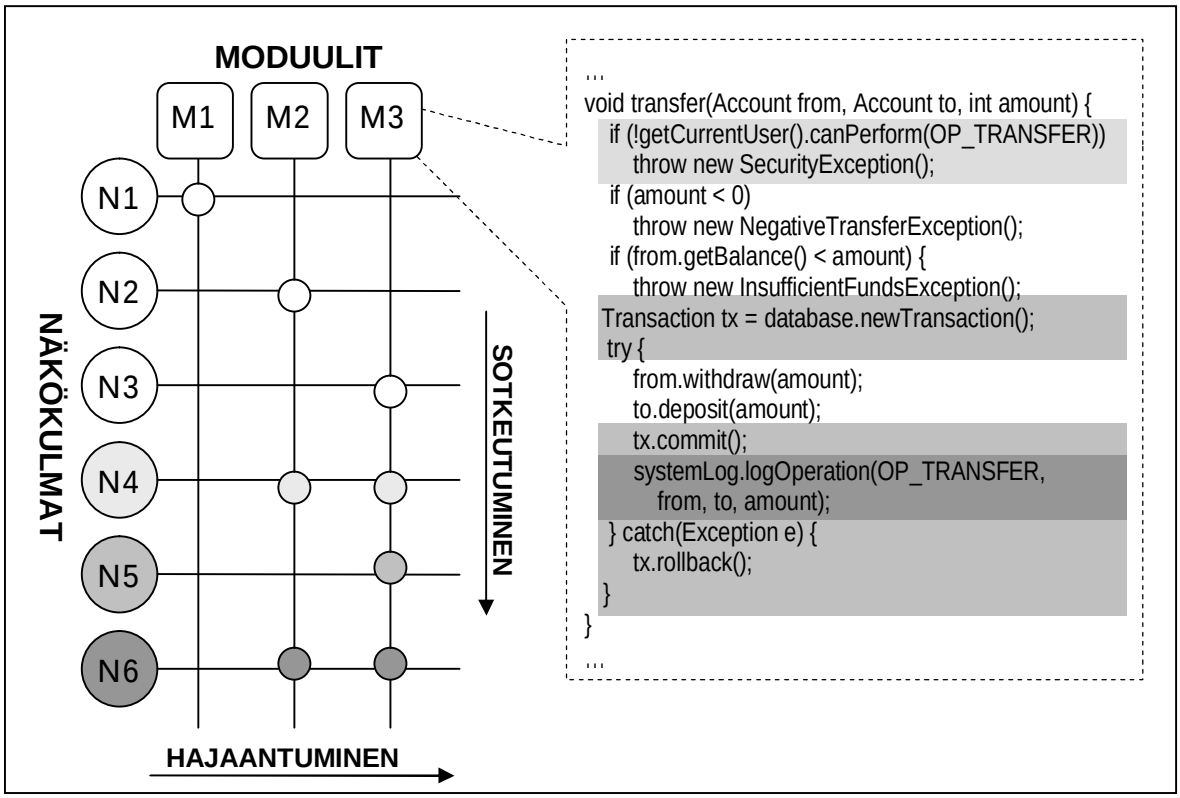
Näkökulmien erottelu –periaatetta noudattavat moduulit ovat myös helpommin uudelleenkäytettäviä, koska moduulit eivät sisällä muiden näkökulmien tuomia riippuvuuksia. Esimerkiksi liiketoimintakäsitteiden toteutukset ovat helpommin uudelleenkäytettäviä, jos ne eivät sisällä tietoturvaan ja lokikirjoitukseen liittyvää toteutusta. Yhteen moduuliin toteutettu lokikirjoitusnäkökulma on myös helpommin uudelleenkäytettävä, koska sen toteutus ei sijaitse eri moduuleissa.

2.2 Läpileikkaava näkökulma ei modularisoidu

Nykyiset ohjelmointikielet rajoittavat näkökulmien erottelu -periaatteen soveltamista, sillä niillä ei voida toteuttaa kaikkia näkökulmia modulaarisesti. Tällaisten näkökulmien toteutus hajaantuu useisiin moduuleihin, joissa se sotkeutuu moduulin alkuperäisen näkökulman toteutuksen kanssa [ShP05]. Näitä hajaantuvia ja sotkeutuvia näkökulmia nimitetään läpileikkaaviksi näkökulmiksi. Esimerkiksi tietoturva, pysyvyys, samanaikaisuuden hallinta ja lokikirjaus ovat yleensä läpileikkaavia näkökulmia. On syytä huomata, että näkökulman läpileikkaavuus on sovelluskohtaista. Esimerkiksi pysyvyys voidaan toteuttaa joissakin sovelluksissa modulaarisesti, jolloin se ei ole läpileikkaava näkökulma.

Kuvassa 2.2 on esitetty ohjelma, jonka toteutuksessa esiintyy sekä näkökulmien hajaantumista että sotkeutumista. Kuvan vasemmassa laidassa on matriisi, jossa ohjelman moduulit M1-M3 ovat vaaka-akselilla ja näkökulmat N1-N6 pystyakselilla. Näkökulma tai sen osa on toteutettu moduuliin, kun näkökulman vaaka-akselin ja moduulin pystyakselin leikkauskohdassa on ympyrä. Kuvasta nähdään, että näkökulmat N1, N2 ja N3 noudattavat näkökulmien erottelu -periaatetta, sillä niiden toteutus sijaitsee vain yhdessä moduulissa. Näkökulmien N4-N6 toteutus taas hajaantuu useampaan moduuliin (näkökulmien vaakarivillä on useita ympyröitä). Sotkeutuminen on taas nähtävissä moduulien pystyakseleilta. Moduuli sisältää useampien näkökulmien toteutuskoodia, jos pystyakselilla on useita ympyröitä.

Kuvan 2.2 oikeassa laidassa on esimerkki näkökulmien sotkeutumisesta M3-moduulissa. Moduulin yhteen metodiin on sotkeutunut neljä eri näkökulmaa: moduulin alkuperäinen liiketoimintanäkökulma N3, tietoturvanäkökulma N4, transaktion hallinta -näkökulma N5 ja jäljitettävyysnäkökulma N6. Näkökulman toteutukseen kuuluva ohjelmakoodi on merkitty metodiin näkökulman omalla värillä. Värittömät lauseet kuuluvat liiketoimintanäkökulmaan. Sotkeutumista voisi esiintyä moduulissa laajemmaltikin, mutta tässä on kuvattu sotkeutuminen vain yhden metodin osalta.



Kuva 2.2: Esimerkki näkökulmien hajaantumisesta ja sotkeutumisesta.

Hajaantuminen ja sotkeutuminen heikentävät ohjelman kehitettävyyttä, ymmärrettävyyttä, ylläpidettävyyttä ja uudelleenkäytettävyyttä. Vaikutuksia voidaan tarkastella näkökulmaan liittyvän hajaantumisen ja moduuliin liittyvän sotkeutumisen näkökulmista erikseen.

Hajaantumiseen liittyvät ongelmat alkavat jo hajaantuvaa näkökulmaa toteutettaessa, sillä näkökulma on muistettava toteuttaa kaikkiin näkökulman edellyttämiin paikkoihin. Paikkoja voi olla useita ja ne voivat edellyttää lisättävän ohjelmakoodin mukauttamista. Koska toteutuksen tekee ihminen, virheiden mahdollisuus on aina olemassa. Toteutus voi unohtua joistakin paikoista, tai ohjelmakoodia ei muisteta mukauttaa paikan edellyttämään muotoon. Esimerkiksi lokikirjausnäkökulman toteuttaminen edellyttää yleensä lokikirjauskoodin lisäämistä useaan paikkaan, jolloin koodia ei ehkä muisteta lisätä kaikkiin paikkoihin. Lisäksi lokikirjauskoodin mukauttaminen voi unohtua, kun koodia lisätään leikkaa ja liimaa -tyylillä.

Näkökulman hajaantunut toteutus vaikeuttaa näkökulman ymmärtämistä, ylläpidettävyyttä ja uudelleenkäytettävyyttä. Näkökulman hajaantuneesta toteutuksesta on vaikea hahmottaa näkökulman toimintaa. Tämä johtuu siitä, että toteutuksen osat on ensin paikannettava moduuleista, minkä jälkeen niistä on yritettävä muodostaa kokonaiskuva. Kun näkökulman toteutusta on vaikea ymmärtää, sen ylläpito on vaikeaa. Erityisesti muutosvaikutusten arviointi on työlästä, sillä se on periaatteessa tehtävä jokaisen muutettavan kohdan osalta

erikseen. Hajaantuneen näkökulman toteutuksen uudelleenkäyttö on käytännössä mahdotonta hajaantuneen toteutuksen vuoksi.

Sotkeutuminen vaikutukset moduulille ovat samat kuin hajaantumisen vaikutukset näkökulmalle: huonontunut kehitettävyys, ymmärrettävyys, ylläpidettävyys ja uudelleenkäytettävyys. Moduulin kehittäminen vaatii, että kehittäjä on tietoinen erilaisista näkökulmista ja että hän ottaa näkökulmat huomioon toteutusta tehdessä. Esimerkiksi liiketoimintalogiikkaa toteutettaessa on usein huomioitava poikkeuskäsittely ja lokikirjoituspolitiikka. Kehittäjä ei ole välttämättä tietoinen erilaisista näkökulmista tai hän on voinut ymmärtää niiden käytön väärin.

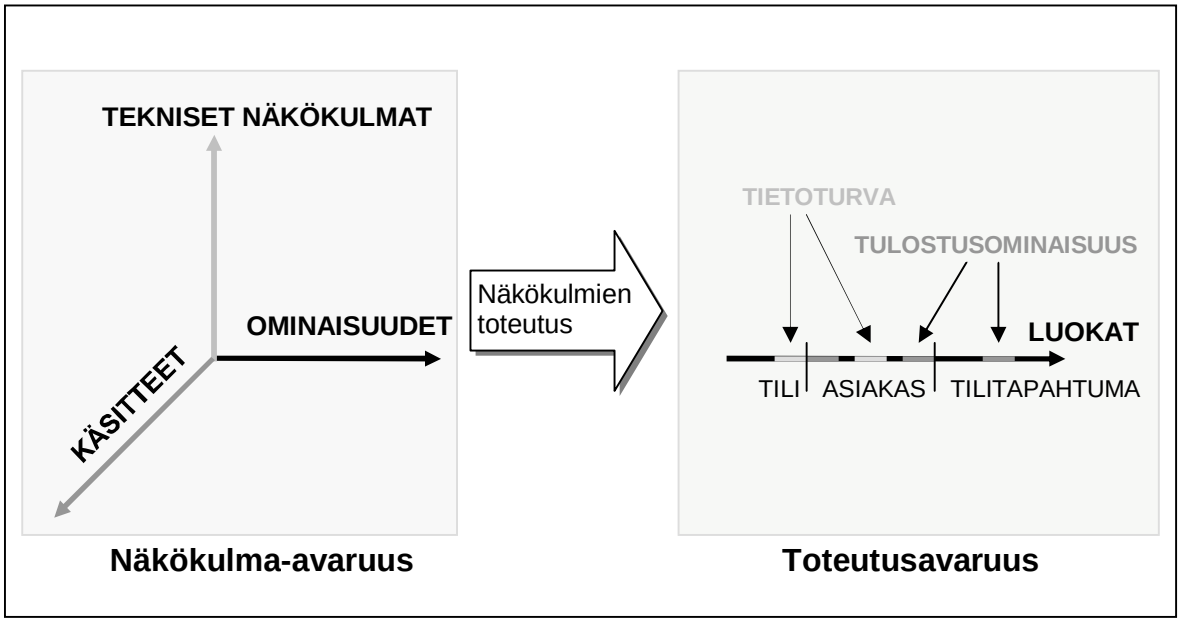
Yksi keskeisistä esteistä ohjelmien uudelleenkäytölle ja uusiin käyttötarkoituksiin mukauttamiselle on näkökulmien erottelun puute [OHT00]. Kaivatun ominaisuuden toteuttava moduuli voi sisältää muita näkökulmia, jotka ovat yksilöllisiä alkuperäiselle käyttöyhteydelle, mutta joita ei tarvita uudessa käyttötarkoituksessa. Esimerkiksi komponentti voi sisältää alkuperäiselle käyttöyhteydelle yksilöllisen lokikirjauspolitiikan, joka ei sovellu enää uuteen käyttötarkoitukseen. Ongelma voidaan välttää, jos yleiset osat erotellaan käyttöyhteydelle yksityiskohtaisista osista.

2.3 Ensisijaisen ositustavan tyrannia ohjelmointikielten ongelmana

Tietokoneohjelman toteutuksen ositus voi nykyisillä ohjelmointikielillä perustua vain yhden tyyppisiin näkökulmiin kerrallaan, kuten esimerkiksi tietoon, käsitteiden luokitteluun, tiedon peittämiseen [Par72] tai toimintoihin [TOH99]. Olio-ohjelmien ensisijainen ositus perustuu usein ongelma-alueen käsitteiden luokitteluun. Esimerkiksi pankkisovelluksen liiketoimintalogiikan ositus voi perustua ensisijaisesti käsitteiden luokitteluun, mikä tuottaa esimerkiksi tili-, asiakas- ja tilitapahtuma-luokat. Tämä käsitteisiin perustuvat ositus estää kuitenkin käsitteille yhteisten näkökulmien, kuten tietoturvan tai jäljitettävyyden, osittamisen omiin luokkiin. Näiden toteutus hajaantuu väistämättä liiketoimintakäsitteitä edustaviin luokkiin. Ensisijainen ositustapa on siis vallitseva, minkä vuoksi ongelmaa kutsutaan *ensisijaisen ositustavan tyranniaksi* (tyranny of the dominant decomposition) [EAK01].

Ensisijaisen ositustavan tyrannian taustalla on yksinkertainen syy. Moniulotteiset näkökulmat on toteutettava yksiulotteisen osituksen tarjoavilla ohjelmointikielillä [Lad03, s. 10]. Kuva 2.3 havainnollistaa tätä ongelmaa. Näkökulma-avaruus sisältää useita toisilleen kohtisuoria näkökulmatyyppejä, kuten käsittelysääntöjä, käsitteitä, ominaisuuksia ja teknisiä näkökulmia. Kohtisuoruudella tarkoitetaan sitä, että eri näkökulmatyyppien näkökulmat voivat olla risteäviä, eli esimerkiksi yhteen ominaisuuteen voi liittyä useita käsitteitä ja käsittelysääntöjä. Näkökulmat toteutetaan kuitenkin ohjelmointikielellä, jolla ositus voi perustua vain yhteen näkökulmatyyppiin kerrallaan. Kuvan esimerkissä on kyseessä olio-

ohjelmointikieli, jolla toteutettujen ohjelmien ositus tehdään luokilla. Kuvan tapauksessa ensisijainen ositus on tehty ohjelman käsitteiden perusteella. Ensisijainen ositus on tuottanut tili-, asiakas- ja tilitapahtuma-luokat, jotka kapseloivat vain kyseiseen käsitteeseen liittyvän ohjelmakoodin. Muut näkökulmatyypit ovat kuitenkin ensisijaista osituksen käsitteitä läpileikkaavia, eli niiden toteutus hajaantuu käsitteitä mallintavien luokkien sekaan.



Kuva 2.3: Yksiulotteisella ohjelmointikielellä ei voida modularisoida ensijaista ositusta läpileikkaavia näkökulmia (kuva mukautettu lähteestä [Lad03, s. 10])

3 Aspektiohjelmointi

Aspektiohjelmointi on uusi ohjelmointiparadigma, jonka avulla läpileikkaavat näkökulmat voidaan toteuttaa modulaarisesti [PRS05, MeR03]. Aspektiohjelmointi ei korvaa aikaisempia ohjelmointiparadigmoja, kuten olio-ohjelmointia tai proseduraalista ohjelmointia, vaan se täydentää näitä uuden tyyppisellä moduulilla, aspektilla, ja uudella moduulien yhdistämistavalla, *kudonnalla* (weaving).

Aspektiohjelmointi muuttaa ohjelman kehitystyötä vain läpileikkaavan toiminnallisuuden osalta, sillä muu ohjelma kehitetään aivan kuten ennenkin. Ohjelma kehitetään normaalisti jollain ohjelmointikielellä eli *peruskielellä* (base language) [MeW99]. Peruskielellä toteutetaan kaikki se ohjelman toiminnallisuus, joka osittuu hyvin peruskielen tarjoamiin moduuleihin. Periaatteessa kaikki ne näkökulmat, joita ei voida osittaa peruskielellä, toteutetaan aspekteina aspektiohjelmointia tukevalla *aspektikielellä* (aspect language). Lopullinen ohjelma koostuu sekä peruskielellä toteutetusta osasta eli *perusohjelmasta* (base program) että aspektikielellä toteutetuista aspekteista [Lam99].

Aspektikielellä toteutetut aspektit eivät vielä sinällään vaikuta perusohjelmaan, vaan ne on kudottava perusohjelman kanssa vaikutuksen aikaansaamiseksi. Kudonta yhdistää aspektit ja perusohjelman ohjelmaksi, joka sisältää sekä aspektien että perusohjelman toiminnallisuuden.

Seuraavissa aliluvuissa perehdytään aspektiohjelmoinnin eri osa-alueisiin. Luvussa 3.1 esitellään aspektiohjelmoinnin ohjelmointimalli ja käsitteet. Luvussa 3.2 esitellään aspektikielen määrittelyyn kuuluvia asioita ja sitä seuraavassa luvussa 3.3 kuvataan aspektikielen toteutus eli kudonta ja siihen liittyvä käsitteistö ja mekanismit.

3.1 Aspektiohjelmoinnin malli

Aspektiohjelmoinnin malli nojaa liitoskohta-käsitteeseen. Liitoskohdat ovat perusohjelmassa sijaitsevia kohtia, joihin aspektit voivat liittää toiminnallisuutta. Aspekti kuvaa liitoskohtien avulla ne perusohjelman kohdat, joihin aspektin sisältämä toiminnallisuus lisätään.

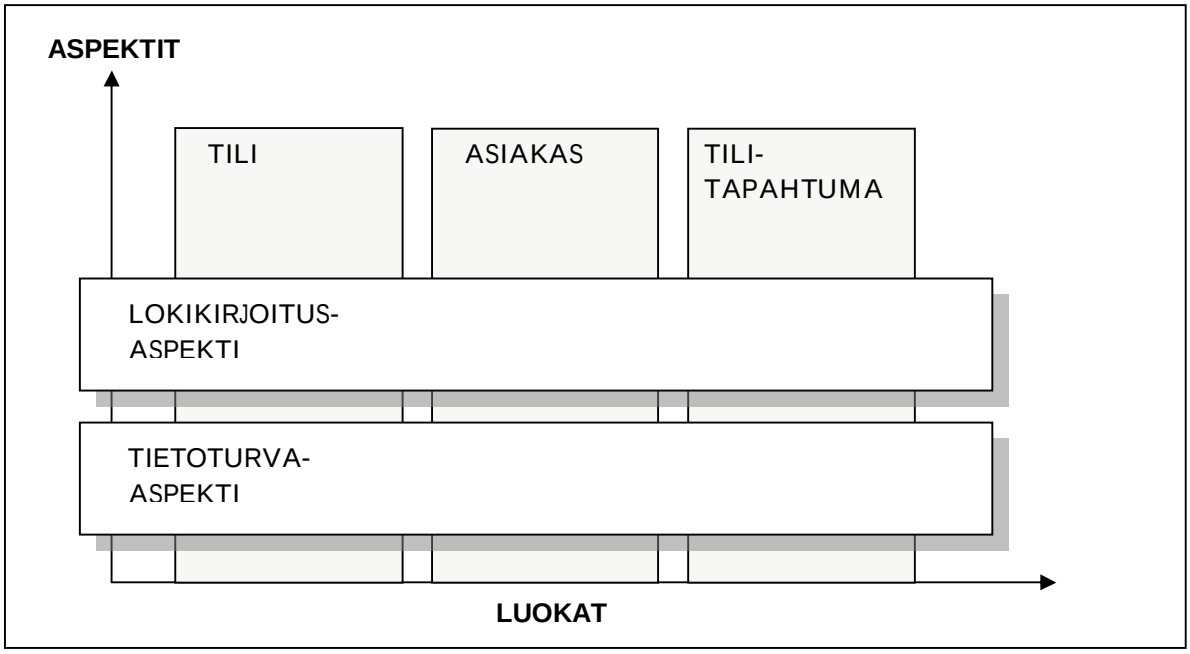
3.1.1 Liitoskohdat aspektien ja perusohjelman välisenä rajapintana

Aspektit vaikuttavat perusohjelman liitoskohtien kautta [ChL03]. Liitoskohdat ovat perusohjelman rakenteessa tai suoritusvuossa sijaitsevia kohtia. Rakenteellisia liitoskohtia kutsutaan staattisiksi liitoskohdiksi, joita voivat olla esimerkiksi luokka, rajapinta, metodi ja muuttuja. Ohjelman suoritusvuossa sijaitsevia liitoskohtia kutsutaan dynaamisiksi liitoskohdiksi, joihin lukeutuvat muun muassa metodikutsu, metodin suoritus, muuttujan luku ja poikkeuksen käsittely.

Liitoskohdat voivat myös paljastaa liitoskohtaan liittyvää staattista tai dynaamista tietoa. Metodikutsu-liitoskohta voi paljastaa staattisena tietona esimerkiksi metodikutsun sisältävän luokan nimen tai kutsutun metodin nimen. Metodikutsu-liitoskohdasta voidaan paljastaa lisäksi ohjelman suoritusaikana esiintyvää dynaamista tietoa, kuten metodikutsun parametrien arvot, kutsuva olio ja kutsuttava olio.

3.1.2 Aspektin rakenne

Aspekti on ositusyksikkö, jolla voidaan toteuttaa modulaarisesti sellaiset näkökulmat, jotka muuten toteutettuna olisivat perusohjelmaa läpileikkaavia. Aspekti tuo näin ohjelman osittamiseen toisen ulottuvuuden [PRS05]. Tällä ulottuvuudella voidaan osittaa ohjelman ensisijaista ositusta läpileikkaava toiminnallisuus. Ensimmäisen ulottuvuuden ositusyksiköitä ovat peruskielen moduulit, joita voivat olla esimerkiksi luokat tai proseduurit. Ensisijainen ositus on tehty kuvassa 3.1 käsitteitä luokittelemalla. Sen jälkeen luokkia läpileikkaavat näkökulmat (lokikirjoitus ja tietoturva) on toteutettu aspekteina.



Kuva 3.1: Luokkiin perustuvaa ositusta läpileikkaavat näkökulmat on toteutettu modulaarisesti aspekteilla.

Aspekti kapseloi läpileikkaavan näkökulman neuvon ja liitoskohtamäärityksen avulla [KLM97]. Neuvo sisältää läpileikkaavan näkökulman toteutuksen ja liitoskohtamääritys kertoo, minne neuvon toiminnallisuus liitetään perusohjelmassa. Neuvo ja liitoskohtamääritys ovat aspektin jäseniä.

Neuvo sisältää vaikutuksen perusohjelman rakenteeseen tai perusohjelman suoritusaikaiseen toimintaan. Rakenteellinen vaikutus voi esimerkiksi lisätä luokkaan uusia operaatioita tai attribuutteja. Suoritusaikainen vaikutus voi lisätä

ohjelmaan uutta toiminnallisuutta tai muuttaa suoritusaikaisten tapahtumien toimintaa. Neuvo kuvaa vaikutuksen lisäksi sen, missä vaiheessa neuvo suoritetaan suhteessa vaikutuskohtaan. Neuvo voi vaikuttaa vaikutuskohtaa ennen, sen jälkeen tai vaikutuskohdan sijasta. Viimeksi mainitun sanotaan olevan vaikutuskohdan ympärillä. Käytettävissä olevat vaihtoehdot vaihtelevat aspektikielittäin.

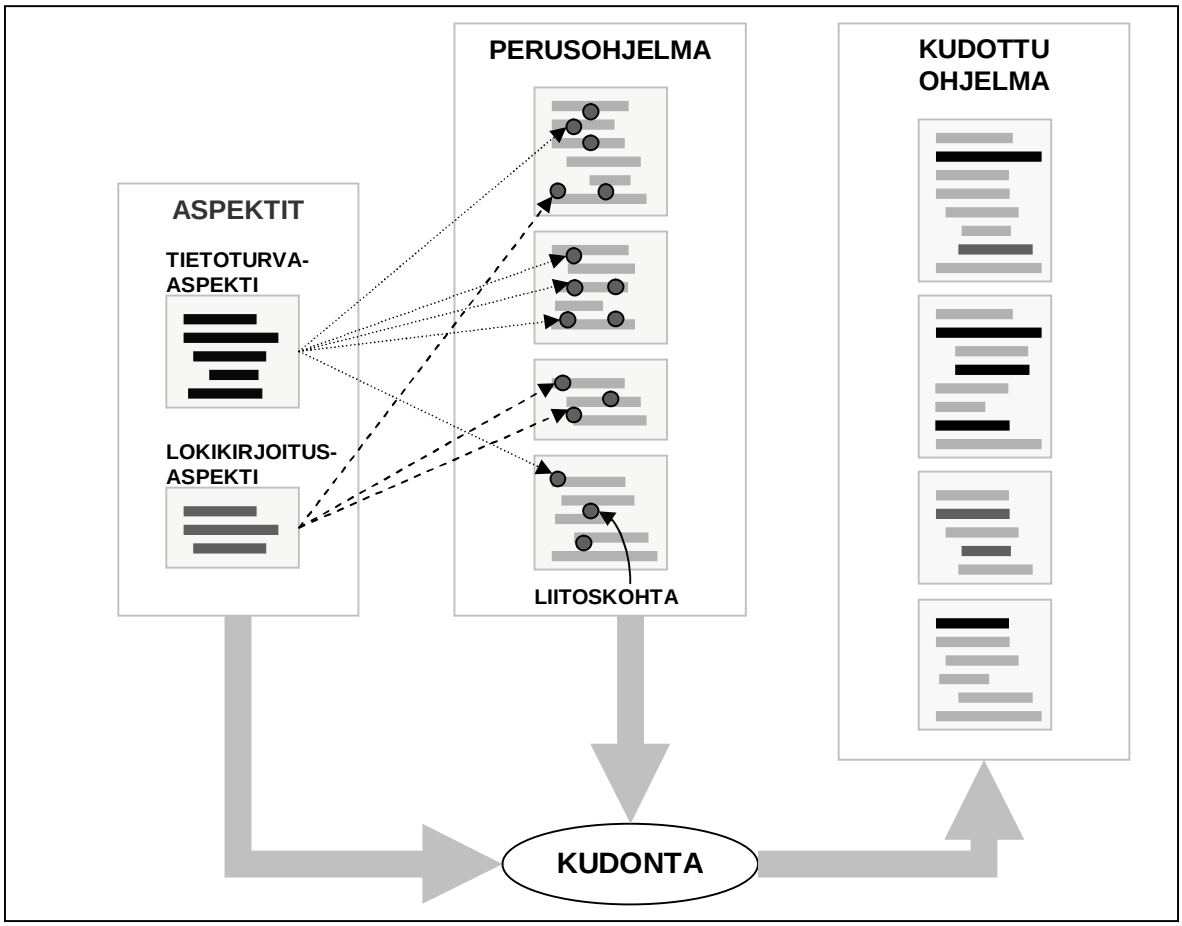
Neuvon vaikutuskohteet määrittelee neuvoon liittetty liitoskohtamääritys. Liitoskohtamääritys on rakenne, joka kuvaa neuvon vaikutuskohdat perusohjelmassa liitoskohtia poimimalla. Liitoskohtien poiminta tehdään liitoskohtamäärityksen sisältämällä väitelauseella, joka poimii kaikki ne liitoskohdat, joista väitelause antaa arvon tosi. Tällainen väitelause voi olla esimerkiksi ”*liitoskohta X on metodikutsu, kutsutun metodin nimi alkaa arvolla set ja metodikutsun parametrina annetaan int-tyyppinen arvo*”.

Liitoskohtamääritykset voidaan nimetä. Tällöin liitoskohtamääritys koostuu nimestä ja siihen liitetystä väitelauseesta. Esimerkiksi edellä kuvattu väitelause voitaisiin yhdistää *setKutsu*-nimeen, jolloin *setKutsu*-nimi edustaa kaikkia set-nimellä alkavien metodien kutsuja, joille välitetään parametrina int-tyyppinen arvo.

Liitoskohtamääritykset voidaan jakaa staattisiin ja dynaamisiin liitoskohtamäärityksiin. Staattiset liitoskohtamääritykset poimivat liitoskohtia sellaisten ominaisuuksien perusteella, jotka voidaan päätellä staattisesti ohjelmaa suorittamatta. Metodikutsuja metodin nimen perusteella poimiva liitoskohtamääritys on staattinen, sillä metodikutsut voidaan poimia ohjelmaa suorittamatta. Dynaamisten liitoskohtamäärityksen poimintaehto on riippuvainen suoritusaikaisesta tiedosta, joten poimitut liitoskohdat tiedetään vasta ohjelman suoritusaikana. Esimerkiksi metodikutsuja poimiva liitoskohtamääritys on dynaaminen, jos metodikutsut poimitaan niiden välittämien parametrien arvojen perusteella.

3.1.3 Kudonta yhdistää aspektit perusohjelmaan

Aspektit eivät itsestään vaikuta perusohjelmaan, vaan niiden vaikutus saadaan aikaan kutomalla. *Kutoja* (weaver) on väline tai mekanismi, joka tekee kudonnan. Kudonnassa aspektin neuvo liitetään perusohjelmaan liitoskohtamäärityksen poimimiin kohtiin. Liittäminen voidaan toteuttaa eri tavoilla, mutta lopputuloksen tulee aina olla sama: neuvon vaikutus esiintyy kaikissa liitoskohtamäärityksen poimimissa liitoskohdissa. Kuvassa 3.2 on esitetty kudonta yleisellä tasolla. Vasemmalla olevat aspektit poimivat perusohjelman liitoskohtia. Kun aspektit ja perusohjelma kudotaan, syntyy kudottu ohjelma, joka sisältää aspektien vaikutuksen.



Kuva 3.2: Kudonta yhdistää aspektit ja perusohjelman ohjelmaksi, joka sisältää molempien toiminnallisuuden.

3.2 Aspektikieli

Aspektikieli määrittelee rakenteet, joilla aspektiohjelmoinnin käsitteet toteutetaan. Aspektikielen on määriteltävä, miten liitoskohtamääritykset, neuvot ja aspektit toteutetaan. Edellä mainittujen rakenteiden toteutustapa ja terminologia vaihtelee aspektikielittäin.

Yleiskäyttöiset aspektikielet voidaan jakaa karkeasti kieliperustaisiin ja kehysperustaisiin aspektikieliin [FVS06]. Kieliperustainen aspektikieli tarjoaa aspektien ohjelmointiin uuden ohjelmointikielen tai laajentaa olemassa olevaa ohjelmointikieltä aspektiohjelmoinnin mahdollistavilla rakenteilla. AspectJ [KHH01] on tällainen aspektikieli, sillä se laajentaa Java-ohjelmointikieltä uusilla avainsanoilla ja rakenteilla. AspectJ:n aspekti on Java-luokkaa muistuttava rakenne, jonka esittelyssä käytetään luokan *class*-avainsanan sijasta *aspect*-avainsanaa. Liitoskohtamääritys ja neuvo toteutetaan aspektin sisään, ja myös niiden esittelyt annetaan uusilla avainsanoilla.

Kehysperustaiset aspektikielet eivät sisällä uutta ohjelmointikieltä, vaan aspektit ohjelmoidaan peruskielen rakenteilla, joille annetaan erityismerkitys lisämerkinnöillä. Spring AOP [Spr07] on ohjelmistokehykseen perustuva aspektikieli, jonka

peruskielenä on Java. Spring AOP:n aspekti toteutetaan Java-luokkana, joka merkitään aspektiksi joko luokkaan liitetyllä Java-annotaatiolla tai erillisellä XML-tiedostolla. Samoin menetellään liitoskohtamäärityksen ja neuvon kanssa, jotka toteutetaan luokan sisään metodeina.

3.2.1 Liitoskohtamalli

Aspektikielen tärkein osa on *liitoskohtamalli* (join point model) [KHH01]. Liitoskohtamalli kertoo aspektikielen ilmaisuvoimasta, sillä se määrittelee neljä asiaa: mitkä ovat aspektikielen sallimat liitoskohdat, mitä tietoja liitoskohta paljastaa, miten liitoskohtamääritykset voivat poimia liitoskohtia ja miten neuvo voi vaikuttaa liitoskohtamäärityksen poimimiin liitoskohtiin.

Aspektikieli määrittää sallitut liitoskohdat peruskielen syntaksista ja semantiikasta. Esimerkiksi laajan käyttäjäkunnan omaavat aspektikielet [AOS07] tarjoavat seuraavat liitoskohdat:

- *Spring 2.0 AOP:* metodin suoritus [Spr07]
- *JBoss AOP*: metodin ja konstruktorin suoritus ja kutsu, muuttujaan viittaus ja kirjoitus [BrH05]
- *AspectJ*: metodin ja konstruktorin suoritus ja kutsu, muuttujaan viittaus ja kirjoitus, neuvon suoritus, olion alustus ja esialustus, luokan staattinen alustus, poikkeuksen käsittely [BrH05]

Aspektikieli määrittelee myös sen, mitä tietoja liitoskohdasta paljastetaan aspekteille. Paljastettavat tiedot riippuvat peruskielestä ja sen tukemasta ohjelmointiparadigmasta. Esimerkiksi olio-ohjelman metodikutsu-liitoskohdasta voidaan paljastaa kutsun parametrit, kutsuva olio ja kutsuttava olio. Proseduraalinen ohjelmointi ei taas tunne olioita, joten proseduraalisten kielten metodikutsuista ei voida paljastaa kutsuvaa ja kutsuttua oliota.

Aspektikielen määrittelemät liitoskohdat ovat tiukasti sidoksissa tiettyyn peruskieleen. Tämän vuoksi aspektikieltä käytetään yleensä vain tietyn peruskielen kanssa. Esimerkiksi AspectJ-aspektikieltä käytetään Java-ohjelmointikielen kanssa, AspectC#:ia käytetään C#:in kanssa ja AspectC:tä C-ohjelmointikielen kanssa.

Liitoskohtamääritykset laaditaan aspektikielen määrittämällä *liitoskohtamäärityskielellä* (pointcut language). Aspektikielen sallimat liitoskohdat ja niiden poimimiseen käytetty liitoskohtamäärityskieli ovat tärkeimmät aspektikielen ilmaisuvoimaan vaikuttavat tekijät. Läpileikkaavaa näkökulmaa on vaikea tai mahdotonta modularisoida, jos aspektikieli ei tarjoa tarvittavia liitoskohtia tai jos liitoskohtamäärityskielellä ei voida poimia haluttuja liitoskohtia.

Liitoskohtamäärityksen väitelause toteutetaan liitoskohtamäärityskielellä. Liitoskohtamäärityskieli määrittää sen, mihin liitoskohdan ominaisuuksiin väite voi perustua. Liitoskohtamäärityskieli voi sallia liitoskohdan poimimisen esimerkiksi seuraavien ominaisuuksien perusteella: liitoskohdan leksikaalinen sijainti (esim. luokka, metodi), liitoskohtaan liittyvä nimi (esim. metodin nimi, muuttujan nimi) tai liitoskohtaan liittyvä allekirjoitus (esim. näkyvyysmääre).

Aspektikieli määrittelee myös neuvoon liittyvät asioita kuten sen, miten neuvo voi vaikuttaa liitoskohtaan (ennen, jälkeen tai ympärillä), miten liitoskohdat paljastamat tiedot ovat neuvon käytettävissä.

3.3 Aspektikielen toteutus: kudonta

Aspektikieli määrittelee sillä ohjelmoitujen aspektien semantiikan, eli suoritusaikaisen merkityksen. Kutoja on ohjelma tai mekanismi, joka saa aspektikielen määrittelyn mukaisen vaikutuksen aikaan ohjelmaa suoritettaessa. Aspektikielen määrittelylle voi olla useita erilaisia toteutuksia. Esimerkiksi AspectJ-aspektikielelle löytyy kaksi erilaista kutojaa: abc-kääntäjä [ACH05] ja ajc-kääntäjä [TAP98b]. Yleensä aspektikielelle on kuitenkin toteutettu vain yksi kutoja.

Kutoja voidaan toteuttaa kahdella eri tavalla: tulkilla tai kääntäjällä [Lad03, s. 25]. Tulkintaan perustuva kutoja osaa suorittaa peruskielellä ja aspektikielellä laadittua koodia. Tulkki tunnistaa aspektit ja suorittaa aspektien neuvot, kun ohjelman suoritus saavuttaa liitoskohdan, jonka aspektin liitoskohtamääritys poimii. Tulkki voi myös kutoa neuvot koodiin sen latausvaiheessa. Tulkkiin perustuva kutoja on luonnollisinta toteuttaa peruskielen tulkkia laajentamalla. Tulkin laajentamiseen liittyy kuitenkin monia ongelmia. Yksi keskeisistä ongelmista on aspektikielten yhtenäisyyden puute. Vaikka tulkkia laajennetaan yhden aspektikielen tuella, se ei osaa suorittaa muita aspektikieliä. Tulkkiin perustuvat kutojat ovat ongelmien ja haasteiden vuoksi harvinaisia; vaikkakin sellaisia on olemassa [JRO07]. Suurin osa aspektikielten toteutuksista perustuu kääntämiseen.

Tulkin toimintaa voidaan simuloida ohjelmistokehyksillä, jotka toteuttavat aspektien tunnistamisen ja aspektien ja perusohjelman yhdistämisen. Nämä eivät vaadi erillistä ja eksplisiittistä kudontavaihetta, vaan kehys kutoo aspektit perusohjelman moduuleihin tarvittaessa. Kehyksiin perustuvia aspektikielten toteutuksia ovat esimerkiksi Spring AOP ja JBoss AOP. Spring AOP perustuu edustajaolioihin (proxy), jotka suorittavat aspektikoodia edustetun olion metodin suoritusta ennen, suorituksen jälkeen tai suorituksen sijasta. Yleisesti edustajaolioihin perustuvat ratkaisut tarjoavat vain metodin suoritus -liitoskohdan ja näin on myös Spring AOP:n kohdalla.

Kääntämiseen perustuva kutoja yhdistää perusohjelman ja aspektit peruskieliseksi ohjelmaksi, joka voidaan suorittaa peruskielen toteuttavalla tulkilla. Tulkin

näkökulmasta kudottu ohjelma ei eroa muista peruskielellä toteutetuista ohjelmista. Perusohjelman ja aspektien yhdistävä kudonta voidaan tehdä ohjelman käännösaikana, käännöksen jälkeen, ohjelman latausaikana tai juuri ohjelman suoritusta ennen. Kaikissa näissä kudonta noudattaa samaa tapaa. Kutoja käsittelee perusohjelman moduulin etsimällä siitä kaikki liitoskohdat, jotka kutoja evaluoi kutojalle näkyvien aspektien liitoskohtamäärityksiä vasten. Jos liitoskohtamääritys poimii evaluoitavan liitoskohdan, liitetään liitoskohtaan neuvon ohjelmakoodi. Liittäminen muuttaa siis perusohjelman moduulin ohjelmakoodia.

Kudontamekanismit voidaan luokitella dynaamisiin ja staattisiin kudontamekanismeihin. Dynaamisilla kudontamekanismeilla voidaan lisätä tai poistaa aspekteja suoritettavasta ohjelmasta ilman sen pysäyttämistä, kun taas staattisilla kudontamekanismeillä tätä ei voida tehdä. Kääntämiseen perustuva kudonta on yleensä staattista. Tämä johtuu siitä, että perusohjelman suoritusta ennen tehtyjä ohjelmakoodin muutoksia on hankalaa tai mahdotonta poistaa ohjelman suoritusaikana.

4 AspectJ

AspectJ on yleiskäyttöinen aspektikieli Java-ohjelmointikielelle. AspectJ laajentaa Java-kieltä aspektiohjelmointiin tarkoitetuilla avainsanoilla ja rakenteilla. Se määrittelee myös liitoskohdat, joihin aspekteilla voidaan vaikuttaa.

Aspektiohjelmointi ja AspectJ liittyvät toisiinsa läheisesti [PRS05, CCH04]. Aspektiohjelmointi syntyi 90-luvun lopulla Palo Alton tutkimuskeskuksessa (PARC). Aspektiohjelmointi-termi esitettiin ensimmäisen kerran Gregor Kiczalezin tutkimusryhmän vuonna 1997 julkaisemassa artikkelissa ”Aspect-Oriented Programming” [KLM97]. Tutkimusryhmä kehitti myös ensimmäisen aspektikielen AspectJ:n, jonka tarkoitus oli havainnollistaa aspektiohjelmoinnin käsitteitä. Vaikka jotkut AspectJ:tä ennen kehitetyt teknologiat voidaan lukea aspektikieliksi, AspectJ oli ensimmäinen liitoskohtamalliin perustuva aspektikieli.

AspectJ:n ensimmäinen julkinen versio julkaistiin maaliskuussa 1998. Vuonna 2003 PARC siirsi AspectJ:n kehitystyön avoimelle Eclipse-yhteisölle, joka on jatkanut AspectJ:n kehittämistä. Vuonna 2005 AspectWerkz-aspektikieli yhdistettiin AspectJ:hin ja saman vuoden joulukuussa julkaistiin yhteistyön tulos AspectJ 5.

Tällä hetkellä AspectJ on kehittynein, koetelluin ja eniten käytetty aspektikieli [YBK04]. Se on toiminut mallina muille myöhemmin kehitetyille aspektikielille, kuten Spring AOP:lle ja JBoss AOP:lle, jotka perustuvat AspectJ:n liitoskohtamalliin. AspectJ:n vaikutus on ollut niin merkittävä, että sitä pidetään nykyään aspektiohjelmoinnin de facto –standardina [Alm03].

AspectJ-kieli tarjoaa erilaiset rakenteet Java-ohjelman suoritusvuossa ja staattisessa rakenteessa esiintyvän läpileikkaavuuden modularisointiin. Ensin mainittua nimitetään AspectJ-terminologian mukaan *dynaamiseksi läpileikkaavuudeksi* (dynamic crosscutting) ja viimeksi mainittua *staattiseksi läpileikkaavuudeksi* (static crosscutting). Tässä tutkielmassa keskitytään pääasiassa rakenteisiin, joilla modularisoidaan dynaaminen läpileikkaavuus.

Dynaaminen läpileikkaavuus modularisoidaan liitoskohtien, liitoskohtamääritysten ja neuvojen avulla. AspectJ:n liitoskohdat ovat Java-ohjelman suoritusvuossa sijaitsevia kohtia. Liitoskohtamääritys kuvaa dynaamisen läpileikkaavuuden liitoskohtia poimimalla, ja siihen liitetty neuvo kuvaa läpileikkaavan toiminnallisuuden. Neuvon toiminnallisuus suoritetaan joka kerta, kun ohjelman suoritus saavuttaa liitoskohtamäärityksen poimiman liitoskohdan.

Staattinen läpileikkaavuus modularisoidaan *tyyppien välisten esittelyiden* (inter-type declarations) avulla. Tyyppien väliset esittelyt voivat esitellä tyypeille uusia

metodeja, muuttujia ja konstruktoreja. Tyyppien väliset esittelyt voivat myös asettaa tyypin laajentamaan toista luokkaa tai toteuttamaan rajapinnan.

Liitoskohtamääritykset, neuvot ja tyyppien väliset esittelyt toteutetaan aspektiin. Aspekti vastaa Java-kielen luokkaa muutamin poikkeuksin. Aspekti voi sisältää myös Java-luokan sisältämiä jäseniä, kuten muuttujia ja metodeja.

Seuraavissa luvuissa esitetään AspectJ:n keskeiset rakenteet ja mekanismit. Luvussa 4.1 kuvataan AspectJ:n tukemat liitoskohdat ja sitä seuraavassa luvussa 4.2 esitetään liitoskohtamääritys. Luvussa 4.3 perehdytään neuvon rakenteeseen. Luvussa 4.4 on esitelty aspektin rakenne, ja luvussa 4.5 kuvataan AspectJ:n kudonta.

4.1 AspectJ:n liitoskohdat

AspectJ:n liitoskohdat ovat Java-ohjelman suoritusvuossa sijaitsevia tapahtumia [APG]. Liitoskohdat voidaan jakaa suoritusaikaisen tapahtuman luonteen perusteella yhteentoista eri liitoskohtatyyppiin. Liitoskohtatyypit ovat seuraavat:

- metodikutsu
- metodin suoritus
- konstruktorikutsu
- konstruktorin suoritus
- luokan staattisen alustuslohkon suoritus
- olion esialustus
- olion alustus
- muuttujaan viittaus
- muuttujaan kirjoitus
- poikkeuskäsittelijän suoritus
- neuvon suoritus

Liitoskohdat määrittävät dynaamisen läpileikkaavuuden ilmaisuvoiman. Esimerkiksi Java-ohjelman silmukan suorituksessa ja metodin toisella rivillä esiintyvää dynaamista läpileikkaavuutta ei voida ilmaista AspectJ:llä. AspectJ:n suunnittelussa on lähdetty kuitenkin siitä, että se ei tue liitoskohtia, joiden varaan rakennetut liitoskohtamääritykset rikkoutuisivat helposti ohjelmaa muutettaessa [Lad03, s. 43].

Liitoskohdan suoritukseen voi myös liittyä kolmenlaista tietoa: suorittava olio, kohdeolion ja argumentit. Liitoskohdan suorittamiseen liittyvä olio on liitoskohdan suoritusaikana *this*-viitteeseen sidottu olio. Kohdeolio on olio, jolle kontrolli siirretään liitoskohdan suorittamisen johdosta, ja argumentit ovat kontrollin

siirtämisessä välitetyt tiedot. Paljastettavat tiedot riippuvat liitoskohtatyypistä ja suorituskontekstista. Staattisessa kontekstissa (staattinen metodi tai luokan staattinen alustuslohko) esiintyvä liitoskohta ei paljasta suoritettavaa oliota. Kohdeoliota ei taas paljasteta liitoskohdista, joiden kohteena on staattinen konteksti (staattinen metodi tai muuttuja).

4.2 Liitoskohtamääritys

AspectJ:n liitoskohtamääritys on rakenne, joka poimii liitoskohtia ja paljastaa liitoskohtaan liittyvää tietoa. Liitoskohtamäärityksiä on kahdenlaisia: nimettyjä ja nimettömiä.

Nimetyn liitoskohtamäärityksen esittely koostuu näkyvyysmääreestä, *pointcut*-avainsanasta, liitoskohdan nimestä, liitoskohdasta paljastetuista parametreista ja väitelauseesta tässä järjestyksessä. Näkyvyysmääre ja parametrit ovat vapaaehtoisia. Liitoskohtamääritys voidaan määritellä myös abstraktiksi, jolloin väitelausetta ei anneta.

Kuvassa 4.2 on esitetty nimetyn liitoskohtamäärityksen rakenne. Hakasuluissa esitetyt kohdat eivät ole pakollisia, ja tummennetut kohdat ovat AspectJ:n avainsanoja.

```
[määreet] pointcut liitoskohtamäärityksen_nimi ([parametrit]) : väitelause;
```

Kuva 4.2: Kuvassa on esitetty nimetyn liitoskohtamäärityksen rakenne.

Nimettömät liitoskohtamääritykset muistuttavat nimettömiä luokkia, sillä niiden määrittely annetaan niiden käyttöpaikassa. Aivan kuten nimettömillä luokilla ei ole nimeä, nimettömillä liitoskohtamäärityksilläkään ei ole nimeä. Niinpä niitä ei voida käyttää uudelleen muualla. Nimetön liitoskohtamääritys koostuu ainoastaan väitelauseesta.

Väitelause on liitoskohtamäärityksen tärkein osa, sillä se poimii halutut liitoskohdat. Väitelause on liitoskohtien ominaisuuksista laadittu väite, joka edustaa niitä kaikkia liitoskohtia, joilla on väitelauseen ominaisuus. Väitelauseet ovat luonnollisella kielellä esitettynä esimerkiksi muotoa ”liitoskohta on set-alkuisen metodin metodikutsu”. Tämä väitelause poimii kaikki ne liitoskohdat, joilla väitelause on totta.

Väitelause rakennetaan AspectJ:n määrittelemistä *liitoskohtamäärityksen primitiiveistä* (primitive pointcut). Yksi AspectJ:n primitiiveistä on call-primitiivi, joka poimii metodikutsu- tai konstruktorikutsu-liitoskohtia. Esimerkiksi yhdestä primitiivistä koostuva väitelause *call(String getValue())* poimii kaikki metodikutsut getValue-nimiseen metodiin, joka palauttaa String-tyyppisen arvon.

Primitiivien yhdistämisessä voidaan käyttää sekä loogista ja-konnektiivia (&&) että tai-konnektiivia (||). Primitiivin vaikutus voidaan myös kääntää ei–konnektiivilla (!). Kuvassa 4.3 on esitetty *aksessori*-niminen liitoskohtamääritys, jonka väitelauseessa on yhdistetty kaksi call-primitiiviä tai-konnektiivilla. Liitoskohtamäärityksen tulkinta on: ”aksessori-liitoskohtamääritys poimii kaikki metodikutsut metodeihin, jotka ovat joko set-alkuisia paluuarvottomia metodeja tai paluuarvon omaavia get-alkuisia metodeja”.

```
public pointcut aksessori() : call(void set*(..)) || call(* get*());
```

Kuva 4.3: Kuvan liitoskohtamääritys poimii kaikki set- tai get-alkuisten metodien metodikutsut.

AspectJ tarjoaa useita primitiivejä, jotka poimivat liitoskohtia erilaisten ominaisuuksien perusteella. Tällaisia ominaisuuksia ovat esimerkiksi liitoskohdan allekirjoitus, tyyppi, sijainti ja tila. Useimmat primitiivit poimivat liitoskohtia kahden ominaisuuden perusteella samanaikaisesti, ja pääsääntöisesti toinen ominaisuus on liitoskohdan allekirjoitus. Primitiivit poimivat halutulla allekirjoituksella varustetut liitoskohdat primitiiville annetulla allekirjoitushahmolla, joka tunnistaa haluttujen liitoskohtien allekirjoitukset. Seuraavassa luvussa on esitellään liitoskohtien allekirjoitukset ja näiden tunnistamiseen käytetyt allekirjoitushahmot.

4.2.1 Liitoskohdan allekirjoitus ja allekirjoitushahmot

Kaikilla Java-kielen rakenteilla, kuten luokilla, rajapinnoilla, metodeilla, konstruktoreilla ja muuttujilla, on allekirjoitus. AspectJ:n määrittelemät liitoskohdat liittyvät johonkin Java-kielen rakenteeseen, josta liitoskohta saa allekirjoituksensa [ASP]. Seuraavassa on kuvattu liitoskohtien allekirjoitukset:

- metodikutsu-liitoskohdan allekirjoitus koostuu metodin määreistä, paluuarvon tyypistä, metodikutsussa käytetystä staattisesta tyypistä, metodin nimestä, parametrien tyypeistä ja metodin heittämistä poikkeuksista.
- metodin suoritus -liitoskohdan allekirjoitus koostuu metodin määreistä, paluuarvon tyypistä, metodin esittelevästä tyypistä, metodin nimestä, parametrien tyypeistä ja metodin heittämistä poikkeuksista.
- olion alustus, olion esialustus, konstruktorikutsu ja konstruktorin suoritus -liitoskohtien allekirjoitus koostuu esittelevästä tyypistä, parametrien tyypeistä ja heitettyjen poikkeusten tyypeistä
- muuttujaan viittaus ja kirjoitus -liitoskohtien allekirjoitus koostuu muuttujan esittelevästä tyypistä, muuttujan tyypistä ja muuttujan nimestä.
- staattisen alustuslohkon suoritus -liitoskohdan allekirjoitus koostuu lohkon esittelevästä tyypistä.

- neuvon suoritus -liitoskohdan allekirjoitus koostuu neuvon esittelevästä aspektista, neuvon parametrien tyypeistä, neuvon paluuarvon tyypistä ja neuvon heittämien poikkeuksien tyypeistä
- poikkeuskäsittelijän suoritus -liitoskohdan allekirjoitus koostuu käsiteltävän poikkeuksen tyypistä

Allekirjoitushahmoja käytetään primitiiveissä tietyn allekirjoituksen omaavien liitoskohtien poimimiseen. Allekirjoitushahmoja on neljä: metodihahmo (method pattern), konstruktorihahmo (constructor pattern), muuttujahahmo (field pattern) ja tyyppihahmo (type pattern).

Tyyppihahmo tunnistaa tyyppejä, jotka voivat olla luokkia, rajapintoja, aspekteja ja primitiivityyppejä [Lad03, s. 68]. Yksittäinen tyyppi voidaan tunnistaa hahmolla, jossa käytetään tyypin täydellistä nimeä. Esimerkiksi tyyppihahmo *java.lang.String* poimii vain java.lang.String-tyypin. Jos tyyppihahmolla ja myös muilla allekirjoitushahmoilla halutaan tunnistaa useita allekirjoituksia kerralla, voidaan allekirjoitushahmossa käyttää seuraavia erikoismerkkejä:

- *-merkki edustaa mitä tahansa merkkijonoa (myös tyhjää merkkijonoa), joka ei sisällä pakkauksen erotinmerkkinä käytettyä pistettä. Pelkän *-merkin sisältävä tyyppihahmo tunnistaa kaikki tyypit, missä tahansa pakkauksessa.

- +-merkkiä käytetään tyyppihahmon loppuliitteenä, jolloin yhdistetty tyyppihahmo tunnistaa kaikkia tyyppihahmon poimimat tyypit ja niistä perityt tyypit.

- ..-merkkijono edustaa mitä tahansa merkkijonoa, joka alkaa ja päättyy pisteeseen.

Tyyppihahmoja voidaan yhdistää myös ja-, tai- ja ei-konnektiiveilla, jotka esitetään tyyppihahmossa vastaavasti merkkijonoilla &&, || ja !. Taulukossa 4.4 on esitetty erilaisia tyyppihahmoja ja niiden tunnistamia tyyppejä.

Tyyppihahmo	Hahmon tunnistamat tyypit
java.lang.Date	java.lang.Date
java.*.Date	Kaikki Date-tyypit, jotka sijaitsevat java-pakkauksen alipakkauksessa. Java standardikirjastossa tämä tarkoittaa tyyppejä java.sql.Date ja java.util.Date
*..*Date	Missä tahansa pakkauksessa sijaitsevat tyypit, joiden nimi päättyy Date-

	loppuliitteeseen.
java.rmi.Remote+	java.rmi.Remote-tyyppi ja siitä perityt tyypit.
java.rmi.Remote+ && !java.rmi.Remote	Vain java.rmi.Remote-rajapinnasta perityt tyypit.

Taulukko 4.4: Esimerkkejä tyyppihahmoista.

Metodihahmolla tunnistetaan metodeihin liittyviä liitoskohtien allekirjoituksia. Metodihahmot muistuttavat metodin allekirjoitusta, sillä myös siinä voidaan antaa metodin määreet, paluuarvon tyyppi, metodin nimi, parametrien tyypit ja metodin heittämät poikkeukset. Esimerkiksi *public Object next() throws NoSuchElementException* on metodihahmo, joka tunnistaa kaikki julkiset next-nimiset metodit, jotka palauttavat *Object*-tyyppisen arvon ja heittävät *NoSuchElementException*-tyyppisen poikkeuksen.

Metodihahmo eroaa metodin allekirjoituksesta kuitenkin muutamalla tavalla. Metodihahmon pakollisia elementtejä ovat vain metodin paluuarvon tyyppi ja metodin nimi, muut määrittelemättömät elementit eivät rajaa hahmon tunnistamia allekirjoituksia. Esimerkiksi *Object next()* -metodihahmo tunnistaa kaikki *Object*-tyyppisen olion palauttavat next-nimiset metodit, välittämättä metodin määreistä ja heittämistä poikkeuksista. Edellä esitetty metodihahmo tunnistaa kuitenkin vain sellaiset metodit, joilla ei ole parametreja. Jos parametreista ei välitetä, voidaan parametrien tilalla antaa ”..”-merkkijono. Se edustaa tyhjää tai useampia parametreja. Metodihahmo *void next(.., int)* tunnistaa *next*-nimiset metodit, joiden viimeisenä parametrina on int-tyyppi. Hahmo tunnistaa myös tapaukset, joissa int-tyyppi on ainoa parametri.

Metodihahmossa voidaan antaa myös metodin esittelevä tyyppi, jolloin metodihahmo huomioi vain näissä tyypeissä esitellyt metodit. Esittelevä tyyppi annetaan metodin nimen etuliitteenä, metodin nimestä pisteellä erotettuna. Kaikki metodihahmon tyypit, kuten paluuarvon tyyppi, metodin esittelevä tyyppi, parametrien tyypit ja poikkeusten tyypit, voidaan käyttää tyyppihahmojen syntaksia. Tyyppihahmoissa voidaan käyttää kaikkia seuraavia erikoismerkkejä: *, .. ja +. Metodin nimessä tai sen tilalla voidaan käyttää *-merkkiä. Metodin nimen tilalla voidaan käyttää *-merkkiä, joka edustaa mitä tahansa metodin nimeä. *-merkki voi olla myös osa metodin nimeä. Esimerkiksi *void java..*.set*(MyType+, .., double) throws *BoundsException* -metodihahmo tunnistaa seuraavanlaiset metodit: metodilla ei ole paluuarvoa, metodi on esitelty missä tahansa java-pakkauksessa tai sen alipakkauksessa sijaitsevassa tyypissä, metodin nimi on set-alkuinen, metodin ensimmäinen parametri on *MyType*-tyyppinen tai sen perivää tyyppiä, viimeinen parametri on *double*-tyyppinen ja metodi heittää poikkeuksen, jonka tyyppinimi päättyy *BoundsException*-loppuliiteeseen.

Taulukossa 4.5 on esitetty erilaisia metodihahmoja ja kuvattu hahmon tunnistamat metodit.

Metodihahmo	Hahmon tunnistamat metodit
public static void main(String[])	Kaikki julkiset, staattiset ja arvoa palauttamattomat main-nimiset metodit, joilla on yksi String[]-tyyppinen parametri.
!public * *Bean+.set*(..)	Kaikki ei-julkiset, Bean-loppuisessa tyypissä tai sen perivässä luokassa esitetyt set-alkuiset metodit.
* *(..) throws (IOException+ && !RemoteException)	Kaikki metodit, jotka heittävät IOException-tyyppisen poikkeuksen lukuun ottamatta RemoteException-tyyppisiä poikkeuksia.

Taulukko 4.5: Metodihahmojen esimerkkejä.

Konstruktorihahmon rakenne on lähes identtinen metodihahmon kanssa. Se poikkeaa metodihahmosta kahdella tavalla: konstruktorihahmossa ei anneta paluuarvoa ja metodin nimen tilalla käytetään *new*-avainsanaa. Taulukossa 4.6 on esitetty kaksi konstruktorihahmoesimerkkiä.

Konstruktorihahmo	Hahmon tunnistamat konstruktorit
new(..)	Kaikki konstruktorit.
public String.new()	String-luokan julkinen parametriton konstruktori.

Taulukko 4.6: Konstruktorihahmojen esimerkkejä.

Muuttujahahmo tunnistaa muuttujien allekirjoituksia. Se koostuu muuttujan määreistä, muuttujan tyyppiä edustavasta tyyppihahmosta ja muuttujan nimestä. Muuttujan nimi voidaan antaa sammalla tavalla kuin metodihahmossa annetaan metodin nimi: muuttujan nimen edessä voidaan antaa muuttujan esittelevä tyyppi tyyppihahmona ja muuttujan nimessä voidaan käyttää *-merkkiä. Taulukossa 4.7 on esitetty muuttujahahmoja ja niiden tunnistamia muuttujia.

Muuttujahahmo	Hahmon tunnistamat muuttujat
private float _value	Kaikki peitetyt float-tyyppiset muuttujat, joiden nimi on _float.
public * com.mypkg..*.*	Kaikki julkiset muuttujat, jotka on esitelty com.mypkg-pakkauksesssa tai sen alipakkauksessa sijaitsevassa

	tyypissä.
* MyClass.*	Kaikki MyClass-tyypin muuttujat.

Taulukko 4.7: Muuttujahahmoesimerkkejä.

4.2.2 AspectJ:n primitiivit

AspectJ määrittelee 20 erilaista primitiiviä, jotka poimivat liitoskohtia erilaisten ominaisuuksien perusteella. Primitiivit voidaan jakaa neljään eri kategoriaan poiminnassa käytetyn ominaisuuden perusteella. Ensimmäinen kategoria koostuu primitiiveistä, jotka poimivat liitoskohtia niiden tyypin perusteella. Näitä primitiivejä on yksitoista: jokaiselle liitoskohtatyypille sitä poimiva primitiivi. Toinen kategoria sisältää viisi primitiiviä, jotka poimivat liitoskohtia niiden sijainnin perusteella. Kolmannen kategorian muodostavat primitiivit, jotka poimivat liitoskohtia niihin liittyvän tilan perusteella. Neljänteen kategoriaan kuuluu yksi primitiivi, jolla käyttäjä voi määrittää liitoskohdan poimintaehdon. Seuraavissa kappaleissa kuvataan primitiivikategoriat yksityiskohtaisemmin aloittaen ensimmäisestä kategoriasta.

AspectJ tarjoaa jokaisen liitoskohtatyypin poimimiseen oman primitiivin. Primitiiveille annetaan parametrina - yhtä primitiiviä lukuunottamatta - allekirjoitushahmo, jolla voidaan rajoittaa primitiivin poimimien liitoskohtien joukkoa vielä allekirjoituksen osalta. Näillä primitiiveillä voidaan siten poimia tietyn tyyppisiä ja tietyn allekirjoituksen omaavia liitoskohtia. Alla on kuvattu liitoskohdan tyypin perusteella poimivat primitiivit:

- *call(metodihahmo)* ja *call(konstruktorihahmo)*: nämä primitiivit poimivat kaikki metodi- tai konstruktorikutsu-liitoskohdat, joiden allekirjoitus vastaa metodihahmoa tai konstruktorihahmoa. Esimerkiksi *call(public * String.*(..))* poimii kaikki String-luokan julkisten metodien metodikutsut

- *execution(metodihahmo)* ja *execution(konstruktorihahmo)*: nämä primitiivit poimivat kaikki metodin ja konstruktorin suoritus-liitoskohdat, joiden allekirjoitus vastaa metodihahmoa tai konstruktorihahmoa

- *initialization(konstruktorihahmo):* poimii olion alustus-liitoskohdat, joiden allekirjoitus vastaa konstruktorihahmoa

- *preinitialization(konstruktorihahmo)*: poimii olion esialustus-liitoskohdat, joiden allekirjoitus vastaa konstruktorihahmoa

- *staticinitialization(tyyppihahmo)*: poimii staattisen alustuslohkon suoritus-liitoskohdat, jotka sijaitsevat tyyppihahmon tunnistamisessa tyypeissä. Esimerkiksi *staticinitialization(com.mypkg..*)* poimii kaikki com.mypkg-

pakkauksesssa tai sen alipakkauksessa sijaitsevien tyyppien staattisten alustuslohkojen suorituksen.

- *get(muuttujahahmo)*: poimii muuttujan viittaus-liitoskohdat, joiden allekirjoitus vastaa muuttujahahmoa.

- *set(muuttujahahmo)*: poimii muuttajan kirjoitus-liitoskohdat, joiden allekirjoitus vastaa muuttujahahmoa. Esimerkiksi set(public * *) poimii kaikki julkisen muuttajan asettamiset

- *handler(tyyppihahmo)*: poimii poikkeuskäsittelijän suoritus -liitoskohdat, jotka käsittelevät tyypiltään tyyppihahmon tunnistamia poikkeuksia. Väitelause handler(IOException+) poimii kaikki poikkeuskäsittelijät, jotka käsittelevät IOException-tyyppisiä tai siitä perittyjä poikkeuksia.

- *adviceexecution()*: poimii kaikki neuvon suoritus -liitoskohdat.

Liitoskohdat voidaan poimia myös niiden sijainnin perusteella. Liitoskohdalla on kaksi sijaintia: sijainti suoritusvuossa ja sen synnyttävän rakenteen sijainti ohjelmatekstissä. Ensin mainittua sijaintia kutsutaan dynaamiseksi ja viimeksi mainittua staattiseksi. AspectJ tarjoaa kaikkiaan viisi primitiiviä sijaintiin perustuvaa poimintaa varten. Staattisen sijainnin perusteella poimivia primitiivejä on kolme:

- *within(tyyppihahmo)*: poimii kaikki liitoskohdat, jotka sijaitsevat tyyppihahmon tunnistamissa tyypeissä. Primitiivillä voidaan poimia vain tietyissä pakkauksissa ja tyypeissä sijaitsevat liitoskohdat. Esimerkiksi *within(com..*)* poimii kaikki *com*-pakkauksesssa ja sen alipakkauksissa sijaitsevissa tyypeissä esiintyvät kaikki liitoskohdat, kuten kaikki metodikutsut, kaikki viittaukset muuttujaan, kaikki suoritettavat poikkeuskäsittelijät ja niin edelleen.

- *withincode(metodihahmo)* ja *withincode(konstruktorihahmo)*: poimivat metodihahmon tai konstruktorihahmon tunnistamissa metodeissa tai konstruktoreissa sijaitsevat liitoskohdat.

Dynaamisen sijainnin perusteella poimivia primitiivejä on kaksi. Nämä primitiivit poimivat kaikki liitoskohdat, jotka sijaitsevat primitiiville määritettyjen liitoskohtien suoritusvuossa. Liitoskohta X sijaitsee liitoskohdan Y suoritusvuossa, jos X tapahtuu Y:n suorituksen aikana. Näillä primitiiveillä voidaan poimia esimerkiksi kaikki muuttujan kirjoitus -liitoskohdat, jotka tapahtuvat annettujen metodien suoritusaikana. Dynaamisen sijainnin perusteella poimivia primitiivejä ovat:

- *cflow(liitoskohtamääritys)*: poimii kaikki liitoskohdat (annetun liitoskohtamäärityksen liitoskohdat mukaan lukien), jotka esiintyvät annetun liitoskohtamäärityksen poimimien liitoskohtien suoritusvuossa.

- *cflowbelow(liitoskohtamääritys):* poimii kaikki liitoskohdat, jotka esiintyvät annetun liitoskohtamäärityksen poimimien liitoskohtien suoritusvuossa. Tätä primitiiviä käytetään usein rekursiivisten kutsujen poissulkemiseen. Ajatellaan esimerkiksi *transaktionaalinenOperaatio*-nimistä liitoskohtamääritystä, joka poimii kaikki metodit (metodin suoritus -liitoskohdat), joissa on aloitettava transaktio. Kuvitellaan, että tällaisia operaatioita ovat *tallennaHenkilö(Henkilö)* ja *tallennaHenkilöt(Henkilö[])*. Viimeksi mainittu operaatio käyttää ensin mainittua operaatiota henkilöiden tallentamiseen. *TransaktionaalinenOperaatio*-liitoskohtamääritys ei kuitenkaan toimi halutulla tavalla, sillä transaktio aloitetaan useita kertoja kun *tallennaHenkilö*-operaatio kutsuu yksittäisen henkilön tallennusoperaatiota. Transaktio tulisi aloittaa, kun *tallennaHenkilöt*-operaatio suoritetaan, mutta ei enää tallennaHenkilö-operaatiossa. Tämä tilanne voidaan korjata käyttäen cflowbelow-primitiiviä. Väitelause *transaktionaalinenOperaatio() && !cflowbelow(transaktionaalinenOperaatio())* poimii kaikki transaktionaaliset operaatiot lukuunottamatta niitä, jotka ovat jo transaktionaalisen operaation suoritusvuossa.

Liitoskohdalla on suoritusaikainen tila ja se voidaan poimia tämän tilan perusteella. Liitoskohdan tilan muodostavat liitoskohdan suoritusaikana this-viitteeseen sidottu olio, liitoskohdan kohteena oleva olio ja kohteelle välitettävät argumentit. AspectJ tarjoaa kolme tilan perusteella poimivaa primitiiviä:

- *this(tyyppi)*: poimii kaikki liitoskohdat, joiden suoritushetkellä *this*-viittaus osoittaa annetun tyyppiseen olioon. Esimerkiksi *this(MyClass)* poimii kaikki liitoskohdat, joiden this-viite osoittaa *MyClass*-tyyppiseen olioon.

- *target(tyyppi)*: poimii kaikki liitoskohdat, joiden kohdeolio on annetun tyyppinen.

- *args(tyyppi, ...)*: poimii kaikki liitoskohdat, jotka välittävät annetut argumentit. Metodikutsu- ja konstruktorikutsu-liitoskohdissa argumentteja ovat metodi- ja kontruktorikutsussa välitetyt parametrit.

AspectJ tarjoaa *if(ehtolauseke)*-primitiivin, jonka avulla käyttäjä voi määritellä liitoskohtien poimintaehdon. Esimerkiksi *if(debuggausPäällä)* poimii kaikki liitoskohdat, jos totuusarvoinen *debuggausPäällä*-muuttuja on tosi.

4.2.3 Annotaatioihin perustuva poiminta

Java 5 –versiosta lähtien Java-ohjelman jäseneen liittyvä metatieto on voitu ilmaista annotaatiolla [TAP04]. Annotaatioita voidaan liittää muun muassa pakkauksiin, luokkiin, rajapintoihin, konstruktoreihin, metodeihin ja muuttujiin. Annotaatiot annetaan ohjelmakoodissa ennen jäsenen esittelyä käyttäen @-merkkiä. Esimerkiksi kuvassa 4.9 esitelty *lisääTilaus*-metodi on annotoitu transaktionaaliseksi. Annotaatiot eivät vielä itsestään sisällä merkitystä, vaan merkityksen antaa annotaatioiden käsittelijä.

```
@Transaktionaalinen
public void lisääTilaus(Asiakas asiakas, Tilaus tilaus) {
    ...
}
```

Kuva 4.9: @Transaktionaalinen-annotaatiolla merkitty *lisääTilaus*-metodi.

AspectJ 5 –versiossa lähtien allekirjoitushahmojen tunnistamia jäseniä on voitu rajata jäsenen annotaatioiden perusteella [TAP04]. Tyyppi-, metodi-, konstruktori- ja muuttujahahmoja on laajennettu siten, että ne voivat tunnistaa jäseniä myös jäseniin liitettyjen annotaatioiden perusteella. Taulukossa 4.10 on esitetty muutamia allekirjoitushahmoja, jotka tunnistavat jäseniä niiden annotaatioiden perusteella.

Liitoskohtamäärityksen väitelause	Väitelauseen poimimat liitoskohdat
within(@Salainen *)	Kaikki liitoskohdat, jotka sijaitsevat @Salainen-annotaatiolla merkityissä tyypeissä.
execution(@Transaktionaalinen * *(..))	Jokainen @Transaktionaalinen-annotaatiolla merkityn metodin suoritus.
set(!@Muuttumaton * *)	Jokainen kirjoitusoperaatio muuttujaan, jota ei ole merkitty @Muuttumaton-annotaatiolla.

Taulukko 4.10: Esimerkkejä väitelauseista, jotka poimivat liitoskohtia annotaatioiden perusteella.

4.3 Neuvo

Neuvo sisältää ohjelmakoodin, joka suoritetaan neuvoon liitetyn liitoskohtamäärityksen poimimissa liitoskohdissa. Neuvo ja liitoskohtamääritys kapseloivat yhdessä läpileikkaavan toiminnallisuuden.

Neuvo koostuu kolmesta osasta: neuvon esittelystä, liitoskohtamäärityksen kuvauksesta ja ohjelmakoodia sisältävästä neuvon rungosta. Neuvon esittely

kuvaa sen, milloin neuvon ohjelmakoodi suoritetaan suhteessa liitoskohtamäärityksen poimimiin liitoskohtiin. Esittelyssä voidaan esimerkiksi määritellä, että neuvo suoritetaan ennen liitoskohtien suoritusta. Neuvon esittely kuvaa myös liitoskohtamäärityksen paljastamat tiedot, jotka ovat käytettävissä neuvon rungossa ja poikkeukset, joita neuvon rungon ohjelmakoodi voi heittää. Neuvon esittely erotetaan kaksoispisteellä liitoskohtamäärityksen esittelystä. Liitoskohtamäärityksen jälkeisten aaltosulkujen välissä on neuvon vartalo, joka sisältää varsinaisen suoritettavan ohjelmakoodin.

AspectJ:n neuvo voidaan suorittaa ennen liitoskohtia, niiden jälkeen tai liitoskohtien ympärillä. Edellä mainitut suoritusajat osoitetaan neuvon esittelyssä *before*-, *after*- tai *around*-avainsanalla. Jälkeen suoritettavaa neuvoa voidaan vielä täsmentää *returning*- tai *throwing*-avainsanoilla. Neuvon esittely voi olla siis jokin seuraavista:

- *before(parametrit): Esineuvo* suoritetaan ennen liitoskohdan suoritusta
- *after(parametrit)*: *Jälkineuvo* suoritetaan liitoskohdan suorituksen jälkeen riippumatta siitä, päättyikö liitoskohdan suoritus onnistuneesti tai poikkeukseen.
- *after(parametrit) throwing [(poikkeustyyppi)]: Poikkeusneuvo* suoritetaan liitoskohdan jälkeen vain, jos liitoskohdan suoritus päättyi poikkeukseen. Esimerkiksi neuvo *after() throwing : call(public * *(..)) {...}* suoritetaan jokaisen julkisen metodin metodikutsun jälkeen, jos metodin suoritus päättyi poikkeuksen. Neuvossa voidaan myös suorittaa vain tietyn tyyppisissä poikkeustilanteissa. Neuvo *after() throwing(RemoteException e) : call(public * *(..)) {...}* suoritetaan vain, kun julkisen metodin suoritus päättyi RemoteException-tyyppiseen poikkeukseen.
- *after(parametrit) returning [(paluuarvon tyyppi)]: Paluuneuvo* suoritetaan vain, kun liitoskohdan suoritus päättyi ilman poikkeusta. Neuvossa voidaan myös suorittaa vain tietyn tyyppisen paluuarvon palauttavien liitoskohtien jälkeen.
- *paluutyyppi around(parametrit): Ympärineuvo* suoritetaan ennen liitoskohtia ja liitoskohtien jälkeen. Ympärineuvo poikkeaa muista neuvoista siinä, että se voi estää liitoskohdan suorituksen. Ympärineuvo voi myös muuttaa liitoskohdan suoritusympäristöä ennen sen suorittamista, ja se voi myös muuttaa suoritetun liitoskohdan paluuarvoa.

Kuvassa 4.11 on esitetty esineuvo ja jälkineuvo, jotka on liitetty *logattavaOperaatio*-nimiseen liitoskohtamääritykseen. Esineuvo tulostaa viestin ennen liitoskohtamäärityksen poimimien julkisten metodien suoritusta, ja jälkineuvo tulostaa viestin metodin suorituksen jälkeen. Neuvot liittävät viestin vakiotekstin perään

suoritettavan metodin allekirjoituksen. Metodin allekirjoitus saadaan *thisJoinPoint*-muuttujan kautta. ThisJoinPoint-muuttuja on neuvon sisällä käytettävissä oleva erikoismuuttuja, joka tarjoaa reflektiivisen pääsyn liitoskohdan tietoihin.

```
pointcut logattavaOperaatio() : execution(public * *(..));

before() : logattavaOperaatio() {
    System.out.println("Suoritetaan metodia "
        + thisJoinPoint.getSignature());
}

after() : logattavaOperaatio() {
    System.out.println("Metodi "
        + thisJoinPoint.getSignature()
        + " suoritettiin onnistuneesti");
}
```

Kuva 4.11: *LogattavaOperaatio* on nimetty liitoskohtamääritys, johon sen alla määritelty esineuvo ja jälkineuvo viittaavat. Esineuvo ja jälkineuvo suoritetaan *logattavaOperaatio*-liitoskohtamäärityksen poimimia liitoskohtia ennen ja jälkeen.

4.4 Aspekti

Aspekti on Java-kielen luokkaa muistuttava rakenne, joka kapseloi neuvot, liitoskohtamääritykset ja staattisen läpileikkaavuuden modularisointiin käytetyt rakenteet. Aspekti toteutetaan luokan tapaan omaan tiedostoon.

Aspektin esittely muistuttaa Java-kielen luokan esittelyä, sillä erolla, että *class*-avainsanan tilalla käytetään *aspect*-avainsanaa. Aspektin esittelyssä voidaan antaa näkyvyysmääre ja aspekti voidaan määritellä abstraktiksi tai lopulliseksi (*final*-määre). Aspekti voidaan määritellä myös etuoikeutetuksi *priviledged*-avainsanalla, jolloin aspektin neuvo voi viitata muiden luokkien yksityisiin jäseniin.

Aspekti voi sisältää neuvojen, liitoskohtamääritysten ja muiden AspectJ:n tarjoamien rakenteiden lisäksi myös Java-kielen rakenteita, kuten metodeja ja muuttujia. Aspektilla on myös oltava parametriton julkinen konstruktori, jolla aspektista luodaan ilmentymiä. Ilmentymiä ei luoda eksplisiittisesti, vaan ilmentymien luonnin hoitaa AspectJ:n ajoympäristö.

Kuvassa 4.12 on esitetty *PoikkeustenJäljittäjä*-niminen aspekti, joka tekee lokikirjauksen jokaisesta ohjelman metodin tai konstruktorin heittämästä poikkeuksesta. Aspekti kirjoittaa lokiin sekä poikkeuksen että poikkeuksen heittäneen metodin tai konstruktorin tiedot. Aspekti sisältää kolme jäsentä: *loki*-nimisen muuttujan, *jäljitettävätMetodit*-nimisen liitoskohtamäärityksen ja poikkeusneuvon. Aspekti käyttää lokitukseen Javan lokikirjoitusrajapintaa, jossa lokikirjaukset tehdään *Logger*-tyyppisen olion kautta. *Logger*-olio luodaan rivillä 6, ja sen nimeksi annetaan nimi ”poikkeusjäljittäjä”. *Loki*-muuttujan alla, rivillä 8, esitellään

jäljitettävätMetodit-liitoskohtamääritys, joka poimii ohjelman jokaisen metodin ja konstruktorin suorituksen. Liitoskohtamääritykseen on liitetty poikkeusneuvo (rivi 12), joka suoritetaan aina, kun metodin tai konstruktorin suoritus päättyy poikkeukseen. Poikkeusneuvon toteutus hakee liitoskohdan allekirjoituksen (rivit 13-14) ja kirjoittaa varoitusviestin (rivit 15-18), joka sisältää poikkeuksen heittäneen tyypin nimen (rivi 16), metodin tai konstruktorin nimen (rivi 17) ja viestin, joka koostuu vakiotekstistä ja poikkeuksen kutsupinosta.

```
 1 import java.util.logging.*;
 2 import org.aspectj.lang.*;
 3
 4 public aspect PoikkeustenJäljittäjä {
 5
 6     private Logger loki = Logger.getLogger("poikkeusjäljittäjä");
 7
 8     pointcut jäljitettävätMetodit() :
 9        execution(* *.*(..)) || execution(*.new(..));
10
11     after() throwing(Throwable t) : jäljitettävätMetodit() {
12         Signature liitoskohdanAllekirjoitus =
13             thisJoinPoint.getSignature();
14         loki.logp(Level.WARNING, liitoskohdanAllekirjoitus
15                 .getDeclaringTypeName(),
16                 liitoskohdanAllekirjoitus.getName(),
17                 "Metodi heitti poikkeuksen: " + t);
18     }
19 }
```

Kuva 4.12: PoikkeustenJäljittäjä-aspekti kirjoittaa lokiin poikkeuksen ja sen heittäneen metodin tai konstruktorin tiedot.

4.5 Kudonta à la AspectJ

AspectJ:n aspektien vaikutus syntyy vasta kudonnan jälkeen, missä aspektit yhdistetään ohjelman muiden luokkien kanssa. Kudonta tuottaa ohjelman, joka sisältää sekä aspektien että luokkien vaikutuksen.

Aspektit käännetään Java-spesifikaation mukaiseksi tavukoodiksi ennen kudontaa [HiH04]. AspectJ:n aspekti käännetään Java-luokaksi class-tiedostoon. Aspektin sisältämä neuvo käännetään standardiksi Java-metodiksi, jonka runko sisältää neuvon rungon ohjelmakoodin. Käännetyn metodin tavukoodi varustetaan lisämääreillä, jotka merkitsevät metodin neuvoksi ja liittävät neuvon liitoskohtamääritykseen. Liitoskohtamääritykset eivät käänny luokan jäseniksi, vaan ne merkitään käännettyyn luokkaan lisätietoina.

Kutoja yhdistää aspektit ja luokat ohjelmamuunnoksilla. Kutoja kartoittaa kudonnan alkuvaiheessa kaikki kudonnassa mukana olevat aspektit. Tämän jälkeen kutoja käsittelee kudonnassa mukana olevat luokat yksi kerrallaan. Se etsii luokan tavukoodista kohtia, jotka mahdollisesti synnyttävät liitoskohdan suoritusaikana. Tällaista kohtaa nimitetään liitoskohdan *staattiseksi varjoksi* (static shadow). Kutoja käsittelee jokaisen staattisen varjon samalla tavalla. Se etsii

neuvojen joukosta ne, joiden liitoskohtamääritys mahdollisesti poimii staattisen varjon. Jos tällainen neuvo löytyy, kutoja lisää luokan tavukoodiin metodikutsun, joka kutsuu metodiksi käännettyä neuvoa. Mikäli neuvon suoritukseen liittyy dynaaminen ehto, kutoja lisää dynaamisen ehdon tarkistavan ohjelmakoodin staattiseen varjoon.

AspectJ:n kudonta voidaan tehdä kolmessa eri vaiheessa: käännösaikana, kääntämisen jälkeen tai luokkien latauksen aikana. Ohjelman kehittäjä voi kutoa kaikki aspektit yhdessä kudonnassa tai hän voi kutoa aspekteja eri kudonta-aikoina. Käännösaikainen kudonta tehdään aspektien ja luokkien käännösaikana. Kun kääntäjä on kääntänyt aspektit ja luokat, se tekee myös niiden välisen kudonnan. Käännösaikaisesta kudonnasta on huomattava se, että kudonta tehdään vain niiden aspektien ja luokkien kesken, jotka ovat käännöksessä mukana. Kääntämisen jälkeinen kudonta tehdään nimensä mukaisesti käännöksen jälkeen. Kääntämisen jälkeinen kudonta on kätevä, jos aspekteja halutaan lisätä jo käännettyihin ohjelmiin ja kirjastoihin. Se ei vaadi luokkien lähdekoodia, toisin kuin käännösaikainen kudonta. Kudonta voidaan tehdä myös silloin, kun luokat ladataan virtuaalikoneeseen. Kudonnan tekee virtuaalikoneeseen rekisteröity agentti, joka muuttaa virtuaalikoneeseen ladattujen luokkien tavukoodia ennen niiden käyttöä.

5 Hauraan liitoskohtamäärityksen ongelma

Aspektiohjelmointi parantaa ohjelman modulaarisuutta, sillä läpileikkaavien näkökulmien toteutus saadaan kapseloitua aspekteihin, pois perusohjelman moduuleista. Parantunut modulaarisuus johtaa ohjelman helpompaan kehitettävyyteen [TGB03]. Näkökulmia on helpompi kehittää, koska muutos näkökulmaan heijastuu todennäköisemmin vain yhteen, näkökulman toteuttavaan moduuliin.

Liitoskohtamääritysten tämän hetkinen ongelma on kuitenkin siinä, että liitoskohtamääritykset ovat tiukasti sidoksissa perusohjelman moduulien syntaksiin ja rakenteeseen. Tarkastellaan esimerkiksi AspectJ:llä toteutettua aspektia, joka toteuttaa vaatimuksen *”kun olion tilaa muuttavan metodin suoritus päättyy, ilmoita kuuntelijoille”*. Vaatimuksen kohta *”olion tilaa muuttavan metodin suoritus”* toteutetaan liitoskohtamäärityksenä ja tämän jälkeen kuuntelijoille tehtävä ilmoitus toteutetaan jälkineuvona. Koska nykyisillä liitoskohtamäärityksillä ei voida suoraan ilmaista edellä esitettyä semantiikka, liitoskohtamäärityksen on johdettava merkitys muista liitoskohtien ominaisuuksista, yleensä liitoskohdan syntaktisista ja rakenteellisista ominaisuuksista. Edellä esitetty liitoskohtamääritykseen liittyvä vaatimus toteutetaan yleensä liitoskohtamäärityksenä, joka poimii metodit nimeämiskäytännön perusteella [CCH04]. Tässä tapauksessa liitoskohtamääritys voisi poimia *set*-alkuisia metodeja. Taustalla on ajatus, että olion tilaa muuttavat metodit ovat *set*-alkuisia. Tällöin tavoiteltu merkitys *”olion tilaa muuttavan metodin suoritus”* toteutetaan liitoskohtamäärityksellä, jonka merkitys on *”set-alkuisen metodin suoritus”*. Näin liitoskohtamääritys ilmaisee semantiikan syntaksin avulla.

Moduulien syntaksista ja rakenteesta oletuksia tekevät liitoskohtamääritykset rikkoutuvat helposti, kun moduulin syntaksia ja rakennetta muutetaan. Liitoskohtamääritykset voivat muutosten johdosta tehdä *vääriä poimintoja* (unintended join point capture) [KMB06b], eli ne poimivat ei-toivottuja liitoskohtia. Liitoskohtamääritykset voivat myös jättää poimimatta sellaisia liitoskohtia, jotka olisi poimittava. Näitä kutsutaan *puuttuviksi poiminnoiksi* (accidental join point miss). Jos esimerkiksi edellisessä kappaleessa esitetyn olion luokkaan lisätään olion tilaa muuttamaton *settings*-niminen metodi, poimii liitoskohtamääritys myös tämän metodin suorituksen. Edellä mainittu poiminta on väärä poiminta, ja sen seurauksena aspekti ilmoittaisi kuuntelijoille olion tilan muutoksesta, vaikka olion tila ei ole muuttunut. Luokkaan voitaisiin lisätä myös *change*-alkuinen metodi, joka muuttaa olion tilaa. Tämä on puuttuva poiminta, sillä liitoskohtamääritys ei poimi tämän metodin suoritusta, jolloin tilan muutos jäisi ilmoittamatta.

Liitoskohtamääritystä nimitetään hauraaksi, kun turvalliselta vaikuttava muutos perusohjelmaan voi muuttaa liitoskohtamäärityksen merkityksen virheelliseksi, vaikka itse liitoskohtamääritystä ei muuteta [KMB06b]. Liitoskohtamäärityksen merkitys muuttuu virheelliseksi, kun se muutoksen vuoksi poimii ei-toivottuja

liitoskohtia ja jättää poimimatta halutut liitoskohdat. Haurauteen liittyvästä ongelmasta käytetään nimitystä hauraan liitoskohtamäärityksen ongelma.

Hauraat liitoskohtamääritykset vaikeuttava perusohjelman kehittämistä, sillä muutoksien vaikutuksia arvioitaessa on otettava huomioon myös ohjelman aspektien liitoskohtamääritykset. Modulaarinen ohjelmointi ja moduulit helpottavat ohjelman kehittämistä ja ylläpitoa. Moduulin palveluita voidaan käyttää vain sen tarjoaman rajapinnan kautta. Näin moduulin toteutusta voidaan muuttaa, kunhan moduuli noudattaa rajapinnan sopimusta. Kun moduulin sisäiseen toteutukseen tehdään muutoksia, riittää muutosvaikutusten arviointiin *modulaarinen päättely* (modular reasoning) [KiM05], jossa otetaan huomioon moduulin rajapinta, sisäinen toteutus ja muiden moduulien tarjoamat rajapinnat, joita moduulin toteutus käyttää. Modulaarinen päättely ei ole enää mahdollista, kun ohjelmassa on mukana hauraita liitoskohtamäärityksiä. Muutosvaikutuksien arvioinnissa on otettava huomioon myös liitoskohtamääritykset.

Aspektiohjelmoinnin vaikutus ohjelman kehitettävyydelle on kaksitahoinen, sillä se sekä parantaa että heikentää ohjelman kehitettävyyttä [TBG03]. Läpileikkaavan näkökulman kehitettävyyteen saadaan selkeä parannus, sillä ylläpidettävänä on yksi aspekti useaan moduuliin hajaantuneen toteutuksen sijaan. Aspektiohjelmointi parantaa myös moduulien kehitettävyyttä, sillä ne eivät enää sisällä muiden näkökulmien toteutusta. Hauraat liitoskohtamääritykset heikentävät taas perusohjelman moduulien kehitettävyyttä, sillä moduulin sisäiset muutokset voivat rikkoa liitoskohtamäärityksiä. Liitoskohtamääritykset rikkoutuvat semanttisella tasolla, joten kääntäjät eivät voi tarkistaa niiden eheyttä. Näin ollen moduulin ylläpitäjän on huomioitava potentiaalisesti kaikki liitoskohtamääritykset jokaista muutosta tehdessään, mikä vaikeuttaa kehittämistä merkittävästi.

Knieselin ja Rhon mukaan hauraan liitoskohtamäärityksen ongelmaa esiintyy ensimmäisen sukupolven aspektikielillä, joihin he laskevat AspectJ:n, HyperJ:n [TaO01], Composition Filtersin [AWB93], DemeterJ:n [LiO97] ja myöhemmät teknologiat, jotka ovat omaksuneet tai kehittyneet edellä mainituista aspektikielistä [KnR05]. Ensimmäisen sukupolven kielillä laaditut aspektit ovat riippuvaisia perusohjelman toteutuksen yksityiskohdista, kuten syntaksista ja rakenteesta. Tämä tiukka sidos ja sen luonne tekee aspekteista hauraita.

Hauraan liitoskohtamäärityksen ongelma on osoitettu selvästi AspectJ:n liitoskohtamalliin perustuvien aspektikielten osalta [CPA06, CJR06, KoS04]. Vaikka tässä tutkielmassa tarkastellaan hauraan liitoskohtamäärityksen ongelmaa ja sen syitä vain AspectJ:n osalta, ongelma koskettaa myös muita AspectJ:n liitoskohtamallin omaksuneita aspektikieliä, kuten Spring AOP:tä ja JBoss AOP:tä [BrR07]. AspectJ, Spring AOP ja JBoss AOP ovat eniten käytettyjä aspektikieliä ja ne ovat tarpeeksi koeteltuja kaupalliseen käyttöön [Kir05]. Siksi hauraan liitoskohtamäärityksen ongelma on paitsi periaatteellisesti myös käytännön tasolla merkittävä.

5.1 Ongelmaesimerkki: grafiikkaeditori

Tarkastellaan hauraan liitoskohtamäärityksen ongelmaa AspectJ:llä ja Java-kielellä toteutetun esimerkkisovelluksen kautta. Esimerkki perustuu yleisesti aspektiohjelmoinnin ja erityisesti AspectJ:n esittelemiseen käytettyyn grafiikkaeditoriin [KiM05, KHH01]. Sovelluksen ohjelmakoodi on esitetty kuvassa 5.1. Kuvassa ei ole esitetty kaikkien sovellukseen kuuluvien luokkien ohjelmakoodia tilan puutteen vuoksi. Samasta syystä myös joitain luokkien jäseniä kuten konstruktorit on jätetty pois.

Sovellus koostuu erilaisista graafisista muodoista, kuten pisteistä ja janoista. Nämä käsitteet on mallinnettu vastaavasti *Piste*- ja *Jana*-luokkina, jotka periytyvät yhteisestä *Muoto*-rajapinnasta. Sovellus sisältää myös *Näyttö*-luokan (ei kuvassa), joka toteuttaa piirtoalustan, jossa graafiset muodot esitetään. *Näyttö*-luokka sisältää myös staattisen *päivitä*-metodin, joka piirtää muodot uudelleen.

Muodon tila koostuu muodon piirtämiseen tarvittavista tiedoista. Esimerkiksi *Jana*-luokka sisältää janan päätepisteet *Piste*-olioina mallinnettuna. Muodot sisältävät myös metodeja tilan kyselyyn ja päivittämiseen. Kun muodon tilan muutos vaikuttaa sen esitykseen näytöllä, näyttöä on päivitettävä kutsumalla *Näyttö*-luokan *päivitä*-metodia. Jos näytön päivitys hoidetaan ilman aspektiohjelmointia, näytön on päivitysvastuu loogisinta antaa eri muotojen hoidettavaksi, sillä muoto itse tietää parhaiten, milloin sen tilan muutos vaatii näytön päivitystä. Näin ollen esimerkiksi *Piste*-luokan metodien *asetaX*- ja *asetaY*-metodien lopussa olisi *Näyttö.päivitä()*-metodikutsu.

```
public interface Muoto {}
```

```
public class Piste implements Muoto {
    int x, y;

    public int annaX() {
        return x;
    }

    public int annaY() {
        return y;
    }

    public void asetaX(int x) {
        this.x = x;
    }

    public void asetaY(int y) {
        this.y = y;
    }
}
```

```
public class Jana implements Muoto {

    private Piste p1, p2;

    public Piste annaP1() {
        return p1;
    }

    public Piste annaP2() {
        return p2;
    }

    public void asetaP1(Piste p) {
        p1.x = p.x;
        p1.y = p.y;
    }

    public void asetaP2(Piste p) {
        p2.x = p.x;
        p2.y = p.y;
    }
}
```

Kuva 5.1: Kuvassa on esitetty grafiikkaeditorin *Muoto*-rajapinnan, *Piste*- ja *Jana*-luokan ohjelmakoodi.

Näytön päivitys on luonnollista toteuttaa aspektilla, sillä kyseessä on muotoja läpileikkaava näkökulma. Toteutukseen riittää aspekti, joka sisältää yhden liitoskohtamäärityksen ja jälkineuvon. Liitoskohtamäärityksen tehtävänä on poimia liitoskohdat, jotka edustavat olion tilan muutoksia. Näytön päivitys on tehtävä poimittujen liitoskohtien jälkeen. Tämä logiikka on liitoskohtamääritykseen liitetyssä jälkineuvossa, joka kutsuu näytön päivitystä aina, kun olion tila muuttuu.

Yllä mainitun liitoskohtamäärityksen toteutustapa ei ole itsestään selvä, sillä yhtä tilan muutosta edustaa useampi liitoskohta, jotka voidaan poimia erilaisten ominaisuuksien perusteella. Oli liitoskohtamäärityksen toteutus mikä tahansa, sen tulisi edustaa kaikkia tilan muutoksia ja tukea perusohjelman kehitettävyyttä. Liitoskohtamäärityksen tulisi ilman muutoksia poimia oikeat liitoskohdat, vaikka perusohjelmaa muutetaan. Seuraavissa kappaleissa esitetään liitoskohtamäärityksen vaihtoehtoisia toteutustapoja ja niihin liittyviä ongelmia.

Pikaisesti ajateltuna parhaiten vaatimuksia vastaa liitoskohtamääritys, joka poimii kaikki muoto-olioiden tilaa muuttavat kirjoitusoperaatiot. Tämä onnistuu AspectJ:n *set*-primitiivillä. Kuvassa 5.2 on tällä ajatuksella toteutettu aspekti. *NäytönPäivittäjä*-aspekti sisältää *tilaMuuttunut*-nimisen liitoskohtamäärityksen, joka poimii jokaisen kirjoitusoperaation ei-staattisen muuttujaan, joka on esitelty jossain *Muoto*-rajapinnan toteuttavassa tyypissä.

```
public aspect NäytönPäivittäjä {
```

```
    pointcut tilaMuuttunut() : set(!static * Muoto+.*);

    after() : tilaMuuttunut() {
        Näyttö.päivitä();
    }
}
```

Kuva 5.2: *NäytönPäivittäjä*-aspekti päivittää näytön aina, kun *Muoto*-tyyppisen olion oliomuuttujaan kirjoitetaan.

Kuvan 5.2 aspektissa on kuitenkin yksi virhe. Se päivittää näyttöä, kun Jana-oliot ovat epäkonsistentissa tilassa. Virhe tapahtuu *asetaP1*- ja *asetaP2*-metodien suorituksen aikana, jolloin aspekti päivittää näyttöä kahdesti: ensimmäisen kerran x-koordinaatin asettamisen jälkeen ja toisen kerran y-koordinaatin asettamisen jälkeen. Jana-olio on epäkonsistensissa tilassa näytön ensimmäisen päivityksen aikana, koska y-koordinaattia ei ole vielä asetettu. Liitoskohtamääritys ei voi siis perustua pelkästään muuttujan kirjoitusoperaatioiden poimintaan.

Poiminta täytyy kohdistaa kirjoitusoperaatioiden sijaan olion tilaa muuttaviin metodeihin, eli mutaattoreihin, sillä mutaattorien on jätettävä olio eheään tilaan. Kun jälkineuvo päivittää näytön muodon mutaattorin suorituksen jälkeen, muoto on aina eheässä tilassa. Esitetyn kaltaista liitoskohtamääritystä ei voida kuitenkaan suoraan toteuttaa AspectJ:llä, sillä metodin suoritus –liitoskohtia ei voida poimia sen perusteella, muuttaako metodin suoritus olion tilaa vai ei. Metodin suoritukset on poimittava jonkin muun metodin ominaisuuden perusteella.

Metodien allekirjoitus on käytännössä ainoa käyttökelpoinen ominaisuus, jonka perusteella muotojen mutaattorit on järkevää poimia. Esimerkiksi metodin sijainnin tai tilan perusteella ei voida paljoakaan sanoa metodin tilaa päivittävästä luonteesta.

Liitoskohtamääritys voi poimia mutaattorit joko luetteloinnin tai nimeämiskäytännön perusteella [CJR06]. Luettelointiin perustuvassa liitoskohtamäärityksessä luetellaan kaikkien niiden mutaattoreiden nimet, jotka halutaan poimia. Kuvassa 5.3 on esitetty uusi versio *NäytönPäivittäjä*-aspektista, joka poimii mutaattoreita luetteloimalla. Luetteloinnilla on kaksi heikkoutta. Ensinnäkin mutaattoreiden luetteloinnin kehittäminen ja ylläpito on työlästä, jos muotoja ja niiden mutaattoreita on useita. Vielä merkittävämpi heikkous on liitoskohtamäärityksen hauraus. Liitoskohtamääritys rikkoutuu esimerkiksi, kun perusohjelman mutaattori uudelleennimetään tai kun uusi mutaattori lisätään perusohjelmaan.

```
public aspect NäytönPäivittäjäV2 {

    pointcut tilaMuuttunut() :
        execution(void Jana.asetaP1(Piste)) ||
        execution(void Jana.asetaP2(Piste)) ||
        execution(void Piste.asetaX(int)) ||
        execution(void Piste.asetaY(int));

    after() : tilaMuuttunut() {
```

```
        Näyttö.päivitä();
    }
}
```

Kuva 5.3: NäytönPäivittäjä-aspektin toinen versio, joka poimii mutaattorit luetteloimalla.

Nimeämiskäytäntöön perustuva liitoskohtamääritys poimii liitoskohdat, joiden allekirjoitus noudattaa tiettyä nimeämiskäytäntöä. Nimeämiskäytäntöön perustuva liitoskohtamääritys on kestävämpi kuin pelkkään luettelointiin perustuva liitoskohtamääritys, koska se ei rikkoudu, kun perusohjelmaan tehdään nimeämiskäytäntöä noudattava muutos.

Esimerkkisovelluksen mutaattoreilla on yhteinen nimeämiskäytäntö, sillä jokaisen mutaattorin nimi alkaa ”aseta”-etuliitteellä. Mutaattoreiden poimimiseen riittää siis liitoskohtamääritys, joka poimii kaikkien aseta-alkuisten metodien suorituksen. Näin liitoskohtamääritys perustuu nimeämiskäytäntöön: se odottaa kaikkien mutaattoreiden alkavan aseta-etuliitteellä. Kuvassa 5.4 on esitetty *NäytönPäivittäjä*-aspektin kolmas versio, jonka liitoskohtamääritys poimii kaikki *Muoto*-rajapinnan toteuttavien luokkien sisältämät aseta-alkuiset metodit.

```
public aspect NäytönPäivittäjäV3 {

    pointcut tilaMuuttunut() :
        execution(void Muoto+.aseta*(..));

    after() : tilaMuuttunut() {
        Näyttö.päivitä();
    }
}
```

Kuva 5.4: Kolmas versio *NäytönPäivittäjä*-aspektista, joka poiminta perustuu nimeämiskäytäntöön.

Mutaattoreiden poimintaan käytetään pääsääntöisesti nimeämiskäytäntöön perustuvia liitoskohtamäärityksiä. Näin tehdään esimerkiksi AspectJ:n ohjelmointioppaassa [TAP98a], AspectJ:tä käsittelevissä kirjoissa [CCH, s. 43] [Lad03, s. 319] ja lukuisissa tieteellisissä artikkeleissa [KoS04, Les06, HaK02], mikä viittaa vahvasti siihen, että nimeämiskäytäntöön perustuva liitoskohtamääritys on paras tapa poimia mutaattoreita.

Nimeämiskäytäntöön perustuvat liitoskohtamääritykset eivät poista hauraan liitoskohtamäärityksen ongelmaa. Liitoskohtamääritykset rikkoutuvat heti, kun nimeämiskäytäntöä ei noudateta. Nimeämiskäytäntö jää helposti noudattamatta, sillä sitä ei voida valvoa automaattisella mekanismilla. Nimeämiskäytännön vastaiset toteutukset jäävät siten helposti huomaamatta. Kun tähän vielä lisätään se, että nimeämiskäytännön noudattaminen on ihmisen huolellisuudesta kiinni, virheiltä ei aina voida välttyä.

Nimeämiskäytäntö on epäkäytännöllinen myös muista syistä. Ensinnäkin nimeämiskäytännön noudattaminen voi johtaa epäluonteviin nimiin. Perusohjelman

kehittäjän on nimettävä ohjelman rakenteet nimeämiskäytännön mukaisesti, mikä voi johtaa nimiin, jotka eivät kuvaa rakenteen varsinaista käyttötarkoitusta tai toimintaa. Toiseksi nimeämiskäytäntöön perustuvat liitoskohtamääritykset siirtävät vastuun liitoskohtamääritysten eheyden säilyttämisestä perusohjelman kehittäjälle. Tämä on vastoin aspektiohjelmoinnin periaatteita, jonka mukaan perusohjelman kehittäjän ei tarvitse olla tietoinen aspekteista.

5.2 Liitoskohtamääritysten haurauden syy

Keskeinen syy hauraan liitoskohtamäärityksen ongelmaan on aspektikielten rajallisessa ilmaisuvoimassa [CJR06]. Aspektikielen ilmaisuvoima on kiinni sen käyttämästä liitoskohtamallista, joka määrittää sallitut liitoskohdat ja niiden poimimiseen tarkoitetun liitoskohtamäärityskielen. Näin liitoskohtamalli kertoo sen, mitä liitoskohtia voidaan poimia ja miten liitoskohtia voidaan poimia.

Liitoskohtamäärityksessä on turvauduttava korvaaviin liitoskohtiin ja poimintaehtoihin, jos aspektikielen liitoskohtamalli ei tue haluttuja liitoskohtia tai niiden poimintaehtoja. Esimerkiksi grafiikkaeditorin tapauksessa AspectJ:n liitoskohtamalli sisältää tarvittavat liitoskohdat (metodin suoritus –liitoskohdat), mutta niitä ei voinut poimia muodon tilaa muuttavan luonteen perusteella, vaan poiminta oli tehtävä metodin allekirjoituksen perusteella. Aspektikieli tarjoaa usein tarvittavat liitoskohdat, mutta ne on poimittava korvaavien ominaisuuksien perusteella. Yleensä poiminnassa on turvauduttava liitoskohdan allekirjoitukseen tai sijaintiin.

Allekirjoituksiin ja sijaintiin perustuvat liitoskohdat ovat tiukasti sidoksissa perusohjelman ohjelmakoodiin. Perusohjelman muutokset vaikuttavat helposti perusohjelman syntaksiin ja rakenteeseen, mikä voi rikkoa yllä mainittuihin ominaisuuksiin perustuvat liitoskohtamääritykset. Liitoskohtamääritysten tulisi perustua korkeamman tason ominaisuuksiin, jotta vältyttäisiin nykyiseltä tiukalta sidokselta.

6 Ratkaisuja AspectJ:n liitoskohtamäärityksen haurauteen

Tässä luvussa tarkastellaan niitä hauraan liitoskohtamäärityksen ongelmaan esitettyjä ratkaisuja, joita voidaan soveltaa AspectJ:ssä. Osa ratkaisuista laajentaa AspectJ:n uusilla ominaisuuksilla ja osaa voidaan soveltaa AspectJ:tä muuttamatta. Ratkaisut on jaettu tutkielmassa kolmeen ryhmään: AspectJ:n ilmaisuvoimaa parantaviin ratkaisuihin, sopimukseen perustuviin ratkaisuihin ja liitoskohtamääritysten muutosanalyysiin perustuviin ratkaisuihin.

AspectJ:n ilmaisuvoimaa parantavat ratkaisut perustuvat joko AspectJ:n liitoskohtamäärityskielen laajennuksiin tai uuteen liitoskohtamäärityskieleen, joka korvaa osittain AspectJ:n liitoskohtamäärityskielen. Ilmaisuvoimaisempien liitoskohtamäärityskielten tavoitteena ovat liitoskohtamääritykset, jotka eivät rikkoudu niin helposti perusohjelmaa kehitettäessä [KMB06b]. Luvussa 6.1 esitetään AspectJ:n liitoskohtamäärityskielelle esitettyjä laajennuksia, jotka on toteutettu abc-kääntäjällä. Luvussa 6.2 esitetään yleisesti metalogiikkakielet, jotka ilmaisuvoimaltaan Turing-täydellisinä logiikkakielinä soveltuvat hyvin liitoskohtamäärityskieliksi. Luvussa 6.2.1 perehdytään LogicAJ-metalogiikkakieleen, joka laajentaa AspectJ:tä uudella metalogiikkakieleen perustuvalla liitoskohtamäärityskielellä.

Sopimukseen perustuvissa ratkaisuissa laaditaan aspektien ja perusohjelman välille eksplisiittinen tai implisiittinen rajapinta tai rajapintoja, jotka velvoittavat sekä aspektien että perusohjelman kehittäjää. Aspektit voivat vaikuttaa perusohjelmaan vain perusohjelman kehittäjän toteuttamien ja ylläpitämien rajapintojen kautta. Sopimukseen perustuvat ratkaisut ratkaisevat hauraan liitoskohtamäärityksen ongelman tarjoamalla aspektien liitoskohtamäärityksille vakaan rajapinnan sen sijaan, että liitoskohtamääritykset olisivat riippuvaisia perusohjelman toteutuksen epävakaista yksityiskohdista.

Sopimukseen perustuvia ratkaisuja ovat luvussa 6.3 esitetyt XPI:t ja luvussa 6.4 esitetty liitoskohtien annotointi. XPI on aspektien ja perusohjelman välinen eksplisiittinen rajapinta, jossa määritellyt liitoskohtamääritykset paljastavat perusohjelman tapahtumia aspektien neuvottavaksi. Perusohjelman kehittäjän on huolehdittava siitä, että XPI:n liitoskohtamääritykset säilyvät eheänä perusohjelman muutoksien jälkeen. Liitoskohtien annotointi on toinen sopimukseen perustuva ratkaisu. Siinä perusohjelman kehittäjä merkitsee liitoskohdat lisätiedoilla, annotaatiolla, joiden perusteella aspekti voi poimia oikeat liitoskohdat. Annotaatiot muodostavat aspektien ja perusohjelman välisen implisiittisen rajapinnan: aspektit riippuvat annotaatioista ja perusohjelman kehittäjä huolehtii niiden merkitsemisestä.

Liitoskohtamäärityksen muutosanalyysiin perustuvat ratkaisut on esitetty luvussa 6.5. Liitoskohtamääritysten muutosanalyysi kertoo, miten liitoskohtien poiminta muuttui liitoskohtamäärityksissä perusohjelman muutosten johdosta. Aspektioh-

jelmoijat voivat tarkistaa muutosanalyysin avulla, että poiminnassa tapahtuneet muutokset eivät sisällä virheitä. Muutosanalyysiin on käytettävissä kaksi eri työkalua, jotka on kuvattu luvun 6.5 aliluvuissa.

6.1 AspectJ:n liitoskohtamäärityskielen laajennukset

Abc-kääntäjä on vaihtoehto ajc-kääntäjälle, AspectJ:n referenssitoteutukselle [ACH05]. Abc-kääntäjä toteuttaa AspectJ-kielen samassa laajuudessa kuin ajc:n 1.2.1-versio. Näin ollen se ei sisällä AspectJ 5 –version esittelemiä ominaisuuksia, kuten Java 5 –tukea. Ajc-kääntäjän heikkous on kuitenkin siinä, että sitä ei ole suunniteltu helposti laajennettavaksi, minkä vuoksi sen liitoskohtamäärityskielen laajentaminen on hankalaa. Näin ei ole abc:n tapauksessa, sillä sen tavoitteena on olla helposti laajennettava alusta AspectJ-kielen laajennusten ja optimointien toteutukseen. Siten ei ole yllättävää, että abc-kääntäjään on toteutettu sekä abc:n kehittäjien että tutkijoiden toimesta useita laajennuksia, jotka laajentavat AspectJ-kieltä muun muassa uudella liitoskohdalla ja useilla liitoskohtamääritys-primitiiveillä.

Abc-kääntäjän LoopsAJ-laajennus laajentaa AspectJ:tä uudella silmukka-liitoskohdalla ja sen poiminnan mahdollistavalla *loop*-liitoskohtamääritys-primitiivillä [HaG06]. LoopAJ:llä voidaan paljastaa tietoja silmukasta ja tietoraken-teesta, jota silmukka iteroi. Silmukan tiedot paljastetaan AspectJ:n standardilla *args*-primitiivillä. Esimerkiksi liitoskohtamääritys ”*loop() && args(min, max, askel, taulukko)*” poimii taulukkoa iteroivan silmukan. Args-primitiivi sitoo silmukan alkuarvon *min*-muuttujaan, lopetusarvon *max*-muuttujaan, indeksiä kasvattavan arvon askel-muuttujaan ja iteroitavan taulukon taulukko-muuttujaan.

Abc-kääntäjään on toteutettu laajennuksina myös useita hyödyllisiä primitiivejä. Eaj-laajennus sisältää *throw*- ja *cast*-primitiivin, joista ensimmäisellä voidaan poimia poikkeuksien heitot ja toisella eksplisiittiset tyyppimuunnokset [ACH05]. Throw-primitiivi täydentää hyvin standardi AspectJ-kieltä, jolla on mahdollista poimia vain poikkeuskäsittelijöiden suoritukset *handler*-primitiivillä. *Cast*-primitiivillä voidaan poimia kaikki implisiittiset ja eksplisiittiset tyyppimuunnok-set. Tällä voitaisiin esimerkiksi tarkistaa, että tyyppimuunnoksissa int-tyypistä short-tyyppiin ei menetetä tarkkuutta (int-arvoa ei voida esittää short-tyypillä).

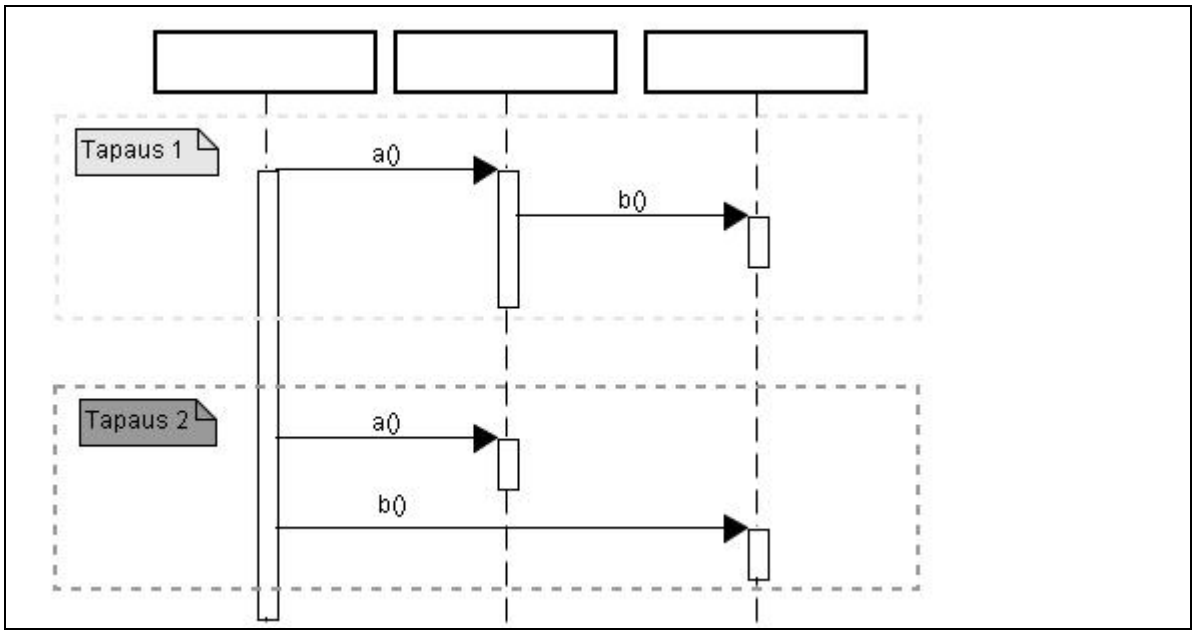
ArrayPT-laajennus parantaa AspectJ:n muuttujien käsittelyä taulukoiden osalta [ChC06]. Standardi AspectJ:llä ei voida selvittää taulukon kirjoitusoperaatiossa tai lukuoperaatiossa käytettyä indeksiä ja kirjoitettavaa tai luettua arvoa. ArrayPT laajentaa AspectJ:tä siten, että edellä mainitut tiedot voidaan paljastaa neuvon käsiteltäväksi.

6.1.1 Tracematch-laajennus

Tracematch-laajennuksella voidaan poimia tapahtumia (liitoskohtia) niitä edeltäneiden tapahtumien eli suoritushistorian perusteella [AAC05]. Pääosa AspectJ:n standardiprimitiiveistä poimii tapahtuman (liitoskohdan) vain sen suoritushetken tietojen tai ominaisuuksien, kuten suoritukseen liittyvien olioiden, argumenttien tai tapahtumaan liittyvän allekirjoituksen, perusteella. Ainoastaan suoritusvuoprimitiivit *cflow* ja *cflowbelow* poimivat tapahtuman sitä edeltäneen tapahtuman perusteella ja tällöinkin vain, jos tapahtuma sijaitsee sitä edeltäneen tapahtuman suoritusvuossa.

Tarkastellaan cflow-primitiivin toimintaa kuvan 6.1 sekvenssikaavion avulla. Tavoitteena on poimia b:n metodikutsut, joita edeltää a:n metodikutsu. Kuvan kaksi eri tapausta täyttävät poimintakriteerit: ensimmäisessä tapauksessa kutsutaan *a*-metodia, jonka suorituksen aikana kutsutaan *b*:tä, ja toisessa tapauksessa *a*:n metodikutsun jälkeen kutsutaan *b*:tä. Cflow-primitiivillä voidaan poimia näistä kuitenkin vain ensimmäinen, sillä siinä *b*:n metodikutsu on *a*:n suoritusvuossa. Cflow-primitiivin poimintaa voi ajatella siten, että se poimii juuri suoritettavan liitoskohdan, jos edeltävä liitoskohta on ohjelman suorituspinossa.

AspectJ:n cflow- ja cflowbelow-primitiiveillä ei voida poimia luvun 5.1 ongelmaesimerkin muodon tilaa muuttavia metodia. Näillä kahdella primitiivillä voidaan poimia muodon metodeissa tehdyt kirjoitusoperaatiot. Sen sijaan niillä ei voi poimia muodon tilaa muuttavia metodeita eli metodeja, joiden suorituksen aikana tehdään kirjoitusoperaatio. Tämä ei ole mahdollista, koska metodit eivät ole kirjoitusoperaatioiden suoritusvuossa.



Kuva 6.1: Ensimmäisessä tapauksessa b-metodi suoritetaan a:n suoritusvuossa. Toisessa tapauksessa a:n suoritus on jo päättynyt, joten b:tä ei suoriteta a:n suoritusvuossa.

Tracematch-rakenteella voidaan poimia tapahtumia niitä edeltävien tapahtumien perusteella. Rakenne ei edellytä, että poimittavien tapahtumien on sijaittava edeltäneiden tapahtumien suoritusvuossa. Tracematch-rakenteessa määritellään yksi tai useampi tapahtuma, tapahtumien kaava ja ohjelmakoodi, joka suoritetaan, kun ohjelman suoritushistoriaan kertyneiden tapahtumien määrä ja järjestys vastaavat tapahtumien kaavaa.

Kuvassa 6.2 on esitetty esimerkki tracematch-rakenteen käytöstä. Tracematch-rakenne esitellään AspectJ:n aspektin sisällä tracematch-määreellä (rivi 3). Määrettä seuraavien sulkujen välissä annetaan tapahtumista paljastetut parametrit, joita esimerkin tapauksessa ei ole. Riveillä 4-10 annetaan rakenteen vartalo, joka sisältää tapahtumien määrittelyt (rivit 4-6), tapahtumien kaavan (rivi 8) ja suoritettavan ohjelmakoodin (rivi 9).

```
 1 public aspect TracematchEsimerkki {
 2
 3     tracematch() {
 4         sym a_aloitus before : call(void a());
 5         sym a_lopetus after : call(void a());
 6         sym b_lopetus after : call(void b());
 7
 8         (a_aloitus | a_lopetus) b_lopetus {
 9             System.out.println("suoritettu a()b() tai a(b())!");
10         }
11     }
12 }
```

Kuva 6.2: Esimerkki tracematch-rakenteen käytöstä.

Tapahtumat määritellään *sym*-määreellä, jota seuraa tapahtuman nimi ja tapahtumaa edustavia liitoskohtia poimiva lause. Lauseen muoto vastaa AspectJ:n neuvon

esittelyä ilman vartaloa. Kuvassa 6.2 riveillä 4-6 on esitelty kolme tapahtumaa: *a_aloitus*, *a_lopetus* ja *b_lopetus*. Rivillä 4 esitelty *a_aloitus*-niminen tapahtuma edustaa *call(void a())*-primitiivin poimimien liitoskohtien aloitustapahtumaa. Lopetustapahtumat annetaan jälki-, poikkeus- tai paluuneuvolla. Rivillä 5 esitelty *a_lopetus* tapahtuma edustaa *a()*-metodikutsujen lopetusta.

Tapahtumien kaava kuvaa tapahtumien suoritushistorian, joka laukaisee kaavan vartalossa annetun ohjelmakoodin suorituksen. Kuvan 6.2 rivillä 8 on määritelty tapahtumien kaava, joka tulostaa viestin (rivi 9), kun suoritushistoriassa on *a_aloitus*- tai *a_lopetus*-tapahtuma, jota seuraa *b_lopetus*-tapahtuma. Tämä kaava poimii kuvan 6.1 molemmat tapaukset. Viesti tulostetaan siis, kun suoritushistoria sisältää joko tapahtumat *a_aloitus* ja *b_lopetus* tai *a_lopetus* ja *b_lopetus*. Kaava *a_aloitus b_lopetus* vastaa AspectJ:n suoritusvuoprimitiivien toimintaa, sillä *b_lopetus*-tapahtuma on *a_aloitus*-tapahtuman suoritusvuossa: *b_lopetus* on syntynyt *a_aloitus*-tapahtuman jälkeen, mutta kuitenkin ennen *a_lopetus*-tapahtumaa. Kaava *a_lopetus b_lopetus* poimii taas a:n metodikutsun jälkeisen b:n metodikutsun, joka ei ole a:n suoritusvuossa. Tämä kaava poimisi siis kuvan 6.1 toisen tapauksen, mitä cflow-primitiivillä ei voitu poimia.

Tapahtumien kaavassa voidaan ilmaista toistoa, valintaa ja peräkkäisyyttä. Kaava *a_lopetus*+ edustaa yhtä tai useampaa *a_lopetus*-tapahtumaa ja *a_lopetus[10]* edustaa kymmentä peräkkäistä *a_lopetus*-tapahtumaa.

Seuraavassa esitetään, miten luvun 5.1 grafiikkaeditorin näyttöä päivittävä aspekti voidaan toteuttaa tracematch-rakenteella. Rakenteella poimitaan muodon mutaattorit eli muodon tilaa muuttavat metodit. Kuvassa 6.3 on esitetty muodon mutaattorit poimiva tracematch-rakenne. *NäytönPäivittäjä*-aspekti sisältää kaksi liitoskohtamääritystä: *muodonMetodinSuoritus* ja *muodonIMetodinSuoritus*. Ensimmäinen poimii muodon metodin suoritukset ja toinen vain muodon ensimmäisen metodin suorituksen. Rivit 7-8 tulee lukea siten, että poimi muodon metodin suoritukset, mutta ei sellaisia, jotka sijaitsevat jonkin muodon metodin suoritusvuossa.

```
 1 public aspect NäytönPäivittäjä {
 2
 3     pointcut muodonMetodinSuoritus() :
 4         execution(* Muoto+.*(..));
 5
 6     pointcut muodonIMetodinSuoritus() :
 7         muodonMetodinSuoritus() &&
 8         !cflowbelow(muodonMetodinSuoritus());
 9
10     tracematch () {
11
12         sym metodinAloitus before : muodonIMetodinSuoritus();
13
14         sym muodonTilanMuutos after :
15             set(!static * Muoto+.*);
16
17         sym metodinLopetus after : muodonIMetodinSuoritus();
```

```
18
19          metodinAloitus muodonTilanMuutos+ metodinLopetus {
20              Näyttö.päivitä();
21          }
22      }
23  }
```

Kuva 6.3: *NäytönPäivitys*-aspektin toteutus tracematch-rakenteella.

Kuvan 6.3 tracematch-rakenne päivittää näytön muodon mutaattorin suorituksen jälkeen. Rakenteessa määritellään kolme tapahtumaa: *metodinAloitus*-symboli edustaa muodon ensimmäisen metodin suorituksen aloitustapahtumaa (rivi 12), *muodonTilanMuutos*-symboli edustaa muodon tilaa muuttavan kirjoitusoperaation lopetustapahtumaa (rivit 14-15) ja *metodinLopetus*-symboli edustaa muodon ensimmäisen metodin suorituksen lopetustapahtumaa. Rivillä 19 esitetty tapahtumien kaava, jonka mukaan näyttö on päivitettävä, kun muodon ensimmäisen metodin suorituksen alun ja lopun välissä on tapahtunut yksi tai useampi muodon tilan muutos.

6.1.2 AspectJ:n laajennusten käyttökelpoisuus

Edellä esitetyt AspectJ-kielen laajennukset parantavat kielen ilmaisuvoimaa. Uusilla liitoskohdilla ja primitiiveillä päästään lähemmäksi liitoskohtamäärityksiä, jotka vastaavat tarkemmin haluttua merkitystä ja tavoitetta. Sen sijaan, että poimimme mutaattorit nimeämiskäytännön perusteella, voimme poimia metodit, jotka todellakin muuttavat olion tilaa.

Laajennukset ovat hyvä alku ilmaisuvoiman kehittämisessä, mutta vielä on paljon tehtävää. Esimerkiksi näytön päivittämisestä vastaavaa aspektia voidaan kehittää vielä pidemmälle poimimalla ainoastaan sellaiset muodon metodit, joiden suorituksen aikana muutetaan muodon graafiseen esitykseen vaikuttavia muuttujia. Tällaista ei voida vielä ilmaista, vaikka AspectJ:n laajennukset olisivat käytössä.

Epäselvää on, voidaanko laajennusten toiminnallisuus toteuttaa standardi-AspectJ:llä muun muassa reflektion avulla. Laajennusten merkitys ei vähene, vaikka ne voitaisiin toteuttaa standardi-AspectJ:llä. Reflektioon perustuva toteutus ei ole tyyppiturvallinen, ja se vaatii yleensä paljon ohjelmakoodia, jota on vaikea ymmärtää. Suoritusaikaiseen reflektioon perustuvat poiminta on myös hidasta, sillä se tehdään aina suoritusaikana jokaiselle liitoskohdalle. Jos poiminta perustuu staattiseen tietoon, voidaan liitoskohdat poimia jo kudonnassa ennen ohjelman suoritusta. Reflektioon perustuva ratkaisu ei ole myöskään aspektiohjelmoinnin hengen mukaista, missä on siirrytty pois eksplisiittisestä metaohjelmoinnista korkeamman tason rakenteisiin, kuten liitoskohtiin, liitoskohtamäärityksiin ja neuvoihin [Kiz04]. Reflektio olisi paluu takaisin eksplisiittiseen metaohjelmointiin.

Laajennusten käytössä on kuitenkin muutamia esteitä. Laajennukset tarjoavaa abc-kääntäjää ei ole tällä hetkellä integroitu kehitysvälineisiin, joten sitä on hieman hankala käyttää. Abc-kääntäjä on myös ominaisuuksiltaan rajallisempi kuin ajc-kääntäjä. Abc-kääntäjästä puuttuu esimerkiksi Java 5 ja AspectJ 5 –versioiden tuki ja latausaikaisen kudonnan tuki.

6.2 Metalogiikkakieli liitoskohtamäärityskielenä

Hauraan liitoskohtamäärityksen ongelman ratkaisuksi on esitetty metalogiikkakieliin perustuvia liitoskohtamäärityskieliä [KnR05, OMB05, GyB03]. Rhon ja Knieselin mukaan metalogiikkakielet parantavat aspektien uudelleenkäytettävyyttä, turvallisuutta, ilmaisuvoimaa ja selkeyttä [KnR05]. Metalogiikkakielet helpottavat hauraan liitoskohtamäärityksen ongelmaa pääasiassa ilmaisuvoimaa parantamalla.

Metalogiikkakielet käyttävä liitoskohtien poimintaan ominaisuuksiltaan täydellistä logiikkakieltä. Logiikkakielet soveltuvat liitoskohtamäärityskieleksi, koska niissä yhdistyy deklaratiivinen ohjelmointi Turing-täydellisyyden kanssa. [Gyb02]. Deklaratiiviset liitoskohtamäärityskielet soveltuvat hyvin liitoskohtien poimintaan, sillä ne kuvaavat vain poimittavien liitoskohtien ominaisuudet. Ei-deklaratiivisessa tavassa olisi toteuttava liitoskohtia poimiva algoritmi, jonka tarkoitusta on vaikeampi ymmärtää deklaratiiviseen kuvaukseen verrattuna. Liitoskohtamäärityskielet ovat tästä syystä yleensä deklaratiivisia ja näin on myös AspectJ:n tapauksessa. Logiikkakielen ja AspectJ:n malliin perustuvien deklaratiivisten liitoskohtamäärityskielten ero on kuitenkin ilmaisuvoimassa: logiikkakieli on Turing-täydellinen, mitä taas AspectJ:n liitoskohtamäärityskieleen perustuvat kielet eivät ole.

Metalogiikkakielen ilmaisuvoiman taustalla ovat metalogiikkamuuttujat, joita voidaan käyttää aspektin rakenteissa, kuten liitoskohtamäärityksissä ja neuvoissa [KnR05]. Metalogiikkamuuttujat ovat sekä logiikkamuuttujia että metamuuttujia. Logiikkamuuttuja on muuttuja, johon voidaan sitoa arvoja ainoastaan evaluoimalla liitoskohtamääritysprimitiivejä, jotka ottavat muuttujan parametrina. Metalogiikkakielen primitiivit toimivat kaksoisroolissa: ne poimivat liitoskohtia ja sitovat poimitun liitoskohdan primitiiville annettuun logiikkamuuttujaan. Logiikkamuuttujat viittaavat siten perusohjelman suoritusaikaisiin tapahtumiin tai peruskielen syntaktisiin rakenteisiin, kuten luokkiin, pakkauksiin, muuttujiin ja metodeihin. Tämän vuoksi niitä kutsutaan metalogiikkamuuttujiksi.

Metalogiikkamuuttujien etu AspectJ:n kaltaisiin liitoskohtamäärityskieliin verrattuna on siinä, että samannimiset metalogiikkamuuttujat viittaavat samaan liitoskohtaan, jolloin kyseistä liitoskohtaa voidaan käyttää liitoskohtamäärityksessä ja neuvossa eri yhteyksissä. AspectJ:llä ei voida toteuttaa esimerkiksi liitoskohtamääritystä, joka poimii kaikki metodikutsut, joissa metodia kutsutaan metodin

esittelevän tyypin ohjelmakoodista. Liitoskohtamäärityksen pitäisi siis poimia tyypin sisäiset metodikutsut. Liitoskohtamääritys voitaisiin periaatteessa toteuttaa AspectJ:llä käyttäen call-primitiiviä ja withincode-primitiiviä. Call-primitiivi poimii metodikutsut, joten se tuntee kutsuttavan metodin esittelevän tyypin X. Withincode-primitiivillä valitaan metodikutsuista vain ne, jotka on tehty tyypin X metodeista. Call- ja withincode-primitiivien olisi siis jaettava keskenään tietoa tyypistä X, mutta tätä ei voida tehdä AspectJ:n liitoskohtamäärityskielellä.

Edellä mainittu liitoskohtamääritys voidaan kuitenkin toteuttaa metalogiikkamuuttujien avulla (kuva 6.4). Kuvassa call-primitiivi ja withincode-primitiivi viittaavat samaan metalogiikkamuuttujaan. Kuvan call-primitiivi evaluoi metodikutsu-liitoskohdan ja sitoo kutsuttavan metodin tyypin *?tyyppi*-nimiseen metalogiikkamuuttujaan. Sama metodikutsu-liitoskohta evaluoidaan vielä withincode-primitiivissä, joka poimii metodikutsun, jos se tehdään *?tyyppi*-metalogiikkamuuttujan viittaaman tyypin metodista.

```
pointcut selfcall() :
    call(* ?tyyppi.*(..)) && withincode(* ?tyyppi.*(..));
```

Kuva 6.4: Tyypin sisäiset metodikutsut poimiva liitoskohtamääritys, jossa käytetään metalogiikkamuuttujia.

Metalogiikkakieliä on kehitetty useita ja eri ohjelmointikielille: LogicAJ tukee Javaa, CARMA-kieltä [GyB03] käytetään Smalltalkin kanssa, ALPHA-kieltä [OMB05] käytetään kokeellisen oliokielen kanssa. ALPHA-kieli on näistä ilmaisuvoimaisin, mutta kokeellisen luonteensa ja peruskielensä vuoksi se ei ole hyödyllinen käytännössä.

Seuraavassa esitetään esimerkkien avulla muutamia AspectJ:n liitoskohtamäärityskielen puutteita ja miten metalogiikkakielellä voidaan ratkaista nämä ongelmat. Metalogiikkakielenä käytetään LogicAJ:tä [RhK04].

6.2.1 LogicAJ

LogicAJ on AspectJ:n laajennus [RhK04, Log07]. Se laajentaa AspectJ:tä pääasiassa metalogiikkamuuttujilla, joita voidaan käyttää liitoskohtamäärityksissä, neuvoissa ja tyyppien välisissä esittelyissä. Metalogiikkamuuttujat ilmaistaan ohjelmakoodissa kysymysmerkillä, jota seuraa muuttujan nimi, esimerkiksi *?luokka* on luokkaniminen metalogiikkamuuttuja. LogicAJ sisältää myös listamuuttujan, joka edustaa mielivaltaista määrää elementtejä, kuten esimerkiksi metodin parametreja. Listamuuttujat esitetään ohjelmakoodissa metalogiikkamuuttujan nimeä edeltävällä kahdella kysymysmerkillä, esimerkiksi *??argumentit*.

LogicAJ:n metalogiikkamuuttujat ovat hienojakoisia, sillä niillä voidaan viitata kaikkiin Java-kielen rakenteisiin, kuten pakkauksiin, tyyppeihin, muuttujiin,

metodeihin, metodien parametreihin, metodien vartaloihin ja määreisiin. Metalogiikkamuuttujia voidaan käyttää liitoskohtamäärittelyissä, neuvoissa ja tyyppien välisissä esittelyissä periaatteessa jokaisen Java-kielen rakenne-elementin tilalla.

LogicAJ:n liitoskohtamäärityskieli tarjoaa samat primitiivit kuin AspectJ sillä poikkeuksella, että LogicAJ:n primitiiveissä voidaan käyttää metalogiikkamuuttujia. LogicAJ tarjoaa myös uusia primitiivejä, joilla voidaan sitoa rakenteita hienojakoisemmin metalogiikkamuuttujiin.

Seuraavassa esitellään LogicAJ:n ja yleisesti metalogiikkakielten käyttökelpoisuutta esimerkin avulla. Lähteeseen [RhK05] pohjautuvassa esimerkissä laaditaan aspekti, joka korvaa olioviitteet olioita jäljittelevien olioiden (mock-olioiden) viitteillä.

6.2.2 LogicAJ esimerkki: aspekti korvaa oliot mock-olioilla

Testauksessa käytetään yleisesti jäljittelyolioita, joilla korvataan testattavan luokan riippuvuudet. Jäljittelyolio toteuttaa saman rajapinnan kuin korvattu olio, mutta sen toteutus palvelee vain testausta. Jäljittelyoliot toimivat kontrolloidulla ja tunnetulla tavalla, jolloin yksikkötestauksessa voidaan keskittyä vain yksikkötestattavan luokan toiminnallisuuden testaukseen.

Tässä esimerkissä toteutetaan aspekti, joka korvaa oliot niitä jäljittelevillä olioilla. Korvaaminen tehdään olioiden luontivaiheessa: jos luotavan olion luokalle löytyy sitä jäljittelevä luokka, luodaan olion sijasta sitä jäljittelevä olio. Jäljittelevä luokka päätellään seuraavasti: luokalla C on sitä jäljittelevä luokka, jos löytyy C:n perivä luokka, jonka nimi on C:n nimi yhdistettynä ”Mock”-loppuliitteellä. Esimerkiksi *Henkilö*-luokkaa jäljittelevä luokka on *HenkilöMock*-luokka, mikäli se on *Henkilö*-luokan aliluokka.

Kuvassa 6.5 on esitetty aspekti, joka korvaa olioviitteet mock-olioilla. Rivillä 3 esitetty liitoskohtamääritys *korvattavaOlionLuonti* poimii kaikki konstruktorikutsut, joissa konstruktorin esittelevälle tyypille löytyy sitä jäljittelevä luokka. Liitoskohtamääritys paljastaa jäljittelevän luokan ja konstruktorikutsun argumentit vastaavasti *?mock*- ja *??argumentit*-metalogiikkamuuttujilla (rivi 3).

Liitoskohtamäärityksen väitelause koostuu riveistä 4-8. Väitelause poimii kaikki konstruktorikutsut ja konstruktorin esittelevä tyyppi sidotaan *?luokka*-muuttujaan (rivi 4). Konstruktorikutsun parametrit sidotaan *??argumentit*-listamuuttujaan (rivi 5). Väitelause katenoi ?luokka-muuttujan viittaaman luokan nimeen ”Mock”-loppuliitteen ja sitoo tuloksen *?mock*-muuttujaan (rivi 6). Väitelause tarkistaa jäljittelyluokan olemassaolon (rivi 7) ja sen, että jäljittelyluokka periytyy *?luokka*-muuttujaan sidotusta tyypistä.

Rivillä 10 esitetty ympärineuvo neuvoo *korvattavaOlionLuonti*-liitoskohtamääritystä. Ympärineuvo saa jäljittelevän luokan ja konstruktorikutsun argumentit liitoskohtamääritykseltä parametreina. Ympärineuvona se korvaa alkuperäisen konstruktorikutsun jäljittelyluokan konstruktorikutsulla, jolle välitetään samat argumentit kuin alkuperäiselle konstruktorikutsulle (rivi 13).

```
 1 public aspect MockAspekti {
 2
 3     pointcut korvattavaOlionLuonti(?mock, ??argumentit) :
 4         call(?luokka.new(..)) &&
 5         args(??argumentit) &&
 6         concat(?luokka, "Mock", ?mock) &&
 7         class(?mock) &&
 8         subtype(?mock,?luokka);
 9
10     Object around(?mock, ??argumentit) :
11             korvattavaOlionLuonti(?mock, ??argumentit) {
12
13         return new ?mock(??argumentit);
14     }
15 }
```

Kuva 6.5: MockAspekti korvaa ohjelman olioviitteet jäljittelyolioilla, jos olion tyypille löytyy siitä peritty jäljittelyluokka.

Kuvan 6.5 MockAspekti voidaan toteuttaa myös AspectJ:llä. AspectJ:n versio on esitetty kuvassa 6.6. Se eroaa LogicAJ:n toteutuksesta liitoskohdan poimintalogiikan osalta. LogicAJ:ssä poimintalogiikka on toteutettu liitoskohtamääritykseen primitiiveillä (kuva 6.5, rivit 4-8), kun taas AspectJ:ssä poimintalogiikka on toteutettu neuvossa suoritusaikaisen reflektion avulla. Kuvan 6.6 konstruktorikutsu-liitoskohtamäärity (rivi 6) poimii kaikki konstruktorikutsut, joten siihen liitetyn ympärineuvon (rivit 8-39) vastuulle jää jäljittelyluokan etsintä ja sen instantiointi. Seuraavassa on selitetty ympärineuvon toiminta:

1. Haetaan kutsutun konstruktorin allekirjoitus (rivit 9-10)

2. Haetaan konstruktorin esittelevä tyyppi (rivit 12-13)

3. Yhdistetään konstruktorin esittelevän tyypin nimen perään ”Mock”-liite, jolloin saadaan jäljittelyluokan nimi (rivit 15-16)

4. Ladataan jäljittelyluokka (rivit 19-25). Jos jäljittelyluokka löytyi, jatketaan kohdasta 5. Jos jäljittelyluokkaa ei löydetä, suoritetaan alkuperäinen konstruktori ja palautetaan olio neuvosta.

5. Haetaan alkuperäisen konstruktorikutsun parametrit ja parametrien tyypit (rivit26-28)

6. Haetaan jäljittelyluokasta konstruktori, jonka parametrit ovat yhteensopivat alkuperäisen konstruktorikutsun parametrien kanssa (rivit 31-32)

7. Suoritetaan konstruktori alkuperäisen konstruktorikutsun parametreilla ja palautetaan luotu jäljittelyolio (rivi 34)

```
 1 import java.lang.reflect.Constructor;
 2 import org.aspectj.lang.reflect.ConstructorSignature;
 3
 4 public aspect MockAspekti {
 5
 6     pointcut konstruktorikutsu() : call(*.new(..));
 7
 8     Object around() : konstruktorikutsu() {
 9         ConstructorSignature konstruktorinAllekirjoitus =
10             (ConstructorSignature) thisJoinPoint.getSignature();
11         // haetaan konstruktorin esittelevä luokka
12         Class konstruktorinLuokka =
13             konstruktorinAllekirjoitus.getDeclaringType();
14         // muodostetaan mock-luokan nimi
15         String mockLuokanNimi =
16             konstruktorinLuokka.getName() + "Mock";
17         Class mockLuokka;
18         // ladataan mock-luokka
19         try {
20             mockLuokka = Class.forName(mockLuokanNimi);
21         } catch (ClassNotFoundException e) {
22             // koska mock-luokkaa ei löydy,
23             // suoritetaan alkuperäinen konstruktori
24             return proceed();
25         }
26         Object[] parametrit = thisJoinPoint.getArgs();
27         Class[] parametrityypit =
28             konstruktorinAllekirjoitus.getParameterTypes();
29         try {
30             // etsitään mock-luokasta sopiva konstruktori
31             Constructor mockKonstruktori =
32                 mockLuokka.getConstructor(parametrityypit);
33             // luodaan mock-instanssi ja palautetaan se
34             return mockKonstruktori.newInstance(parametrit);
35         } catch(Exception e) {
36             throw new RuntimeException("mock-olion luonti " +
37                     "päättyi poikkeukseen \n" + e);
38         }
39     }
40 }
```

Kuva 6.6: AspectJ:llä toteutettu MockAspekti, joka vastaa toiminnallisuudeltaan kuvan 6.5 MockAspektia.

6.2.3 Metalogiikkakielten yhteenveto

Metalogiikkakieliä on ehdotettu ratkaisuksi hauraan liitoskohtamäärityksen ongelmaan. Ne tarjoavat liitoskohtamäärityskieleksi ilmaisuvoimaltaan Turing-täydellisen logiikkakielen. Liitoskohtamäärityksessä voidaan ilmaista logiikkakielten ansiosta monimutkaisempia poimintaehtoja kuin esimerkiksi AspectJ:n liitoskohtamäärityskielellä toteutetuissa liitoskohtamäärityksissä.

AspectJ:llä toteutetuissa aspekteissa joudutaan usein turvautumaan suoritusaikaiseen reflektioon, koska AspectJ:n liitoskohtamäärityskielen ilmaisuvoima ei ole

riittävä [KnR04]. Reflektioon perustuva toteutukset sisältävät muutamia heikkouksia metalogiikkamuuttujien avulla toteutettuihin aspekteihin verrattuna. Reflektiolla toteutetut aspektit vaativat enemmän ohjelmakoodia, mitä on myös vaikea ymmärtää.

Reflektioon perustuvat ratkaisut ovat myös tehottomia, sillä poimittavien liitoskohtien päättely tehdään pääosin suoritusaikana, vaikka poiminta perustuu täysin liitoskohdan staattiseen tietoon. Aikaisemmin esitetyn esimerkin (kuva 6.5) tapauksessa konstruktorin luokan, sitä vastaavan mock-luokan, luokkien periytymissuhteen ja mock-luokan konstruktorin päättely voidaan tehdä staattisesti ohjelmaa suorittamatta. Näin LogicAJ itse asiassa tekeekin, sillä LogicAJ:n kutoja tekee nämä päättelyt jo kudonnan aikana, jolloin päättelyä ei tarvitse tehdä enää suoritusaikana. LogicAJ:n tapauksessa ainoastaan konstruktorin kutsu tehdään suoritusaikana, kun taas AspectJ:n tapauksessa ainoastaan konstruktorien poiminta tehdään staattisesti ja kaikki muu päättely tehdään ohjelman suoritusaikana. LogicAJ:n aspekti on siten suoritusaikana tehokkaampi.

Vaikka logiikkakielet tarjoavat kehittyneen liitoskohtamäärityskielen, ne eivät ole nykyisiä aspektikieliä merkittävästi ilmaisuvoimaisempia. Tämä johtuu siitä, että metalogiikkakielet tarjoavat karkeasti samat liitoskohdat ja niiden poimintaan tarkoitetut primitiivit kuin muutkin aspektikielet. Logiikkakielillä voidaan muodostaa monimutkaisia liitoskohtamäärityksiä, mutta koska poimittavat liitoskohdat ja niiden poimintaehdot eivät muutu, ei ilmaisuvoimaankaan ole odotettavissa suurta parannusta. Tämä tulee esille kuvassa 6.6: AspectJ:llä voitiin kapseloida sama toiminnallisuus kuin LogicAJ:llä, joskin hieman pidemmällä toteutuksella.

6.3 XPI

XPI (crosscut programming interface) on aspektien ja perusohjelman välinen rajapinta, joka peittää perusohjelman ohjelmakoodin yksityiskohdat aspekteilta [GSS06]. XPI:t vastaavat monella tapaa moduulien välisiä rajapintoja. XPI toimii perusohjelman julkisena rajapintana, joka paljastaa liitoskohtamääritysten muodossa joitain perusohjelman tapahtumia. Aspektit voivat vaikuttaa perusohjelmaan vain näitä XPI:iden liitoskohtamäärityksiä neuvomalla. Tämä poikkeaa perinteisestä aspektiohjelmointityylistä, jossa aspektit vaikuttavat suoraan perusohjelman yksityiseen toteutukseen.

XPI-rajapinta koostuu rajapinnan nimestä ja yhdestä tai useammasta liitoskohtamäärityksestä, joiden esittelyissä annetaan liitoskohtamäärityksen nimi ja sen paljastamat parametrit. Nämä liitoskohtamääritykset paljastavat joitain keskeisiä tapahtumia perusohjelman toiminnasta. Jokaisen liitoskohtamäärityksen yhteydessä on myös dokumentoitu liitoskohtamäärityksen toteuttajan ja käyttäjän velvoitteet. Toteuttajan velvoitteena kuvataan ne perusohjelman tapahtumat, jotka

liitoskohtamäärityksen on poimittava. Käyttäjän velvoitteet voivat asettaa rajoituksia liitoskohtamäärityksen tapahtumasta paljastamien parametrien käytölle.

XPI-rajapinnan toteuttaminen on perusohjelman kehittäjien vastuulla. Rajapinnan toteutuksessa annetaan jokaisen liitoskohtamäärityksen toteutus eli liitoskohtia poimivat väitelauseet. Rajapinnan liitoskohtamääritykset ovat käytetyistä aspektikielistä johtuen hauraita, joten rajapinnan yksityisen toteutuksen on huolehdittava niiden eheyden säilyttämisestä. Perusohjelman kehittäjien on tunnettava XPI:t ja niiden liitoskohtamääritykset ja noudatettava rajapinnan sopimusta. Kehittäjän on huolehdittava siitä, että toteutus vastaa liitoskohtamäärityksen perusohjelmasta tekemiä oletuksia, jolloin väitelause poimii vain ja ainoastaan oikeat liitoskohdat. Esimerkiksi jos liitoskohtamääritys poimii liitoskohtia nimeämiskäytännön perusteella, perusohjelman kehittäjän on huolehdittava että ohjelman toteutus noudattaa tätä nimeämiskäytäntöä. Toisaalta liitoskohtamäärityksen väitelause on toteutuksen yksityiskohta, jota voidaan myös muuttaa ilman, että se vaikuttaisi liitoskohtamääritystä käyttäviin aspekteihin.

XPI:n kaltaisia ratkaisuja on ehdotettu myös muiden tahojen toimesta. Ratkaisut eroavat siinä, mihin aspektien ja perusohjelman välisen rajapinnan muodostavat liitoskohtamääritykset toteutetaan. Gudmundson ja Kiczales ehdottavat ratkaisuksi moduulin rajapinnassa määriteltäviä nimettyjä liitoskohtamäärityksiä, jotka moduuli tarjoaa aspektien neuvottavaksi [GuK01]. Moduulin kehittäjä huolehtii liitoskohtamääritysten ylläpidosta ja eheydestä. Gudmundson ja Kiczales ehdottavat myös käytäntöä, jolla moduulit tarjoavat liitoskohtamääritykset: luokkaan liittyvä liitoskohtamääritys toteutetaan kyseisessä luokassa, pakkaukseen liittyvät liitoskohtamääritykset toteutetaan pakkauksen *Pointcuts*-luokassa ja globaali liitoskohtamääritys toteutetaan *pointcuts*-pakkauksen luokassa. Aldrich on taas esittänyt ratkaisuksi uutta moduulisysteemiä [Ald04]. Moduulisysteemi sisältää uuden tyyppisen moduulin, jonka avulla perusohjelman kehittäjät voivat ilmaista toteutuksen yksityiset ja julkiset osat. Aldrichin ratkaisu eroaa XPI:stä ja Gudmundsonin ja Kiczalesin ratkaisusta siinä, että moduulisysteemin kääntäjä voi tarkistaa, että yksityiseen toteutukseen ei viitata. Ongkingco et al. ovat toteuttaneet Aldrichin moduulisysteemin AspectJ-kieleen abc-kääntäjän laajennuksena [OAT06].

Seuraavassa esitetään aikaisemmin esitellyn grafiikkaeditorin toteutus, jossa näytön päivityksen hoitava aspekti erotetaan muusta grafiikkaeditorista XPI-rajapinnalla. Esimerkki perustuu lähteeseen, jossa on demonstroitu XPI:n käyttöä AspectJ:llä ja juuri samaisella grafiikkaeditorilla [GSS06]. Lähteen esimerkki on käännetty suomeksi ja luokkien nimet on muutettu vastaamaan luvussa viisi esitetyn grafiikkaeditorin luokkia.

6.3.1 Grafiikkaeditorin toteutus XPI:llä

Grafiikkaeditoriin toteutetaan yksi XPI, joka paljastaa muotojen tilan muutokset rajapintaa käyttäville aspekteille. Rajapinnan toteutus on esitetty kuvassa 6.7. Rajapinta paljastaa muodoissa tapahtuneet tilan muutokset kahdella julkisella liitoskohtamäärityksellä: *liitoskohta*- ja *ylätasonliitoskohta*-liitoskohtamäärityksillä. Ensin mainittu liitoskohtamääritys paljastaa jokaisen muodon tilan muutokset ja toiseksi mainittu vain muutokset koosteisen muodon tilassa, mutta ei sen komponenteissa. Esimerkiksi *Jana*-olio koostuu kahdesta *Piste*-oliosta. Ylätasonliitoskohta poimii *Jana*-olion tilan muutoksen, mutta ei yksittäisen *Piste*-olion tilan muutosta. Molemmat liitoskohtamääritykset tarjoavat muuttuneen muodon liitoskohtamäärityksen parametrina.

```
public aspect XMuodonTilanMuutos {

    /*
     * liitoskohta()-liitoskohtamääritys paljastaa kaikkien muotojen
     * tilan muutokset. Tilanmuutokset on toteutettava
     * mutaattoreilla, joiden nimet ovat liitoskohtamäärityksen
     * nimeämiskäytännön mukaisia. Kaikkien tällaisten metodien
     * oletetaan muuttavan muodon tilaa.
     * ylätasonLiitoskohta()-liitoskohtamääritys paljastaa ainoastaan
     * useista muodoista koostuvan muodon tilan ”ylätason” muutokset.
     * Esimerkiksi pisteistä koostuvat janan tilan ylätason
     * muutoksiin eivät kuulu pisteitten tilassa tapahtuvat
     * muutokset.
     *
     * Liitoskohtamäärityksiä neuvovat aspektit eivät voi aiheuttaa
     * suorasti tai epäsuorasti parametrina paljastetun muodon tilaa.
     */

    public pointcut liitoskohta(Muoto m):
        target(m)
        && (call(void Muoto+.aseta*(..))
        || call(void Muoto+.siirry(..))
        || call(Muoto+.new(..)));

    public pointcut ylätasonLiitoskohta(Muoto m):
        liitoskohta(m) && !cflowbelow(liitoskohta(Muoto));

    protected pointcut mutaattorit():
        withincode (void Muoto+.set*(..))
        || withincode(void Muoto+.moveBy(..))
        || withincode (Muoto+.new(..));
}
```

Kuva 6.7: *XMuodonTilanMuutos*-rajapinta paljastaa muotojen tilan muutokset kahden julkisen liitoskohtamäärityksen kautta.

Rajapinnan kommenteissa on kuvattu myös rajapinnan toteuttajan ja käyttäjän velvoitteet. Perusohjelman kehittäjän on huolehdittava siitä, että liitoskohta-liitoskohtamääritys poimii vain ja ainoastaan muotojen tilan muutokset. Se onnistuu, kun jokainen muodon tilan muutos tehdään muodon mutaattorissa,

jonka nimi noudattaa liitoskohtamäärityksen nimeämiskäytäntöä. Liitoskohtamääritysten käyttäjä ei taas saa muuttaa liitoskohtamääritysten paljastamien muotojen tilaa suoraan tai epäsuorasti.

Kuvassa 6.8 on esitetty näytön päivittämisen hoitava aspekti, joka käyttää *XMuodonTilanMuutos*-rajapintaa. Aspekti ei ole enää riippuvainen muotojen toteutuksen yksityiskohdista, vaan rajapinnan julkisesta liitoskohtamäärityksestä, joka peittää poiminnan toteutuksen aspektilta. Aspekti ei itse päättele tilan muutoksia, vaan *XMuodonTilanMuutos*-rajapinta tarjoaa tilan muutokset asiakkaiden neuvottaviksi *ylätasonLiitoskohta*-liitoskohtamäärityksen kautta.

```
public aspect NäytönPäivittäjäV4 {

    after() : XMuodonTilanMuutos.ylätasonLiitoskohta(Muoto) {
        Näyttö.päivitä();
    }
}
```

Kuva 6.8: Kuvan aspekti päivittää näytön, kun muodon tila muuttuu.

XPI-sopimuksen noudattamista voidaan myös osittain valvoa. Kuvan 6.9 aspekti valvoo rajapinnan toteutuksen ja käytön sopimuksenmukaisuutta. *MuodonTilanMuutosSopimus*-aspektin ensimmäinen jäsen käyttää AspectJ:n tarjoamaa käännösvirheen esittely –mekanismia (rivi 11). Käännösvirheen esittely –mekanismi koostuu avainsanoista *declare error*, jota seuraa liitoskohtamääritys (rivi 12) ja käännösvirheen yhteydessä näytettävä virheteksti (rivit 13-15) kaksoispisteillä eroteltuina. Käännösvirhe annetaan kaikista niistä liitoskohdista, jotka annettu liitoskohtamääritys poimii. Kuvan tapauksessa käännösvirhe syntyy, kun muodon tilaa muuttava kirjoitusoperaatio sijaitsee jossain muualla, kuin *XMuodonTilanMuutos*-rajapinnan määrittelemässä mutaattorissa.

Rajapinnan käytön valvonta on toteutettu esineuvona riveillä 22-27. Esineuvo tulostaa virheilmoituksen (rivit 25-28), jos neuvon suorituksen aikana (rivi 23) muutetaan muodon tilaa (rivi 24).

```
 1 /*
 2  * Tarkistaa XMuodonTilanMuutos-sopimuksen noudattamisen.
 3  */
 4 public aspect MuodonTilanMuutosSopimus {
 5
 6     /*
 7      * TOTEUTTAJAN VELVOITTEET:
 8      * muodon tilaa muutetaan ainoastaan mutaatoreista,
 9      * jotka noudattavat mutaattoreiden nimeämiskäytäntöä
10      */
11     declare error :
12         (!XMuodonTilanMuutos.mutaattorit() && set(* Muoto+.*)) :
13             "Rajapinnan sopimusrikkomus: Muodon tilaa voidaan"
14                 + " muuttaa ainoastaan nimeämiskäytännön "
15                 + " mukaisesta mutaattorista!";
16
17     /*
```

```
18          * KÄYTTÄJÄN VELVOITTEET:
19          * rajapintaa neuvovat aspektit eivät voi muuttaa muotojen
20          * tilaa
21          */
22         before():
23             cflow(adviceexecution())
24             && XMuodonTilanMuutos.liitoskohta(Muoto) {
25                 System.err.println("Rajapinnan sopimusrikkomus:"
26                     + " XMuodonTilanMuutos-rajapintaa neuvovat "
27                     + " aspektit eivät voi muuttaa muotojen "
28                     + " tilaa!");
29         }
30     }
```

Kuva 6.9: Kuvassa on *XMuodonTilanMuutos*-rajapinnan sopimuksen noudattamista valvova aspekti.

6.3.2 XPI:n yhteenveto

XPI-rajapintoja voidaan käyttää nykyisten aspektikielten kanssa, sillä niiden käyttö ei vaadi uutta aspektikieltä tai nykyisten kielten laajentamista. Lähestymistapa ei myöskään rajoita aspektikielen ominaisuuksien käyttöä, vaan ainoastaan niiden käyttötapaa.

XPI:t modularisoivat paremmin perusohjelman ja sitä neuvovat aspektit. XPI:t vastaavat monelta osin moduulien välisiä rajapintoja, joten ei ole yllättävää että myös niistä saatavat hyödyt ovat samankaltaiset. XPI:t nostavat ohjelman abstraktiotasoa, parantavat perusohjelman kehitettävyyttä, helpottavat muutosvaikutusten arviointia ja mahdollistavat aspektien ja perusohjelman samanaikaisen kehittämisen.

XPI:t tekevät perusohjelman merkittävät tapahtumat eksplisiittisiksi rajapintojen kautta, jolloin ohjelman rakenne heijastaa paremmin keskeisiä abstraktioita. Tämä helpottaa esimerkiksi ohjelman hahmottamista ja parantavat kehittäjien välistä kommunikointia. Grafiikkaeditorin keskeinen ja ohjelman kannalta tärkeä käsite oli muodon tilan muutos, minkä editoriin toteutettu XPI teki eksplisiittiseksi.

XPI-rajapinnat parantavat perusohjelman kehitettävyyttä, koska aspektit eivät voi olla enää riippuvaisia perusohjelman toteutuksen yksityiskohdista. Perusohjelman yksityiseen osaan voidaan tehdä vapaasti muutoksia, kunhan muutokset säilyttävät XPI-rajapintojen sopimukset. Rajapinnat suojaavat aspekteja perusohjelman yksityisen toteutuksen muutosvaikutuksilta.

Rajapinnat helpottavat myös perusohjelman muutosvaikutusten arviointia. Perinteisessä ohjelmoidut aspektit ovat suoraan riippuvaisia perusohjelman toteutuksesta, minkä vuoksi perusohjelman muutoksilla voi olla vaikutuksia aspekteihin, jotka on tarkistettava virheiden varalta. XPI-lähestymistavassa tätä ongelmaa ei ole, sillä muutosvaikutukset rajoittuvat rajapinnan toteutukseen.

XPI:t mahdollistavat perusohjelman ja aspektien samanaikaisen kehittämisen. Perinteisessä aspektiohjelmointimallissa perusohjelma kehitetään ensin, johon liitetään tämän jälkeen aspektit. Tämä järjestys on välttämätön, koska aspektit riippuvat perusohjelman yksityiskohdista, jotka tunnetaan vasta, kun perusohjelma on kehitetty. XPI lähestymistavassa aspektien ja perusohjelman kehittäminen voidaan tehdä samanaikaisesti XPI-rajapintojen ansiosta.

XPI-lähestymistapa ei varsinaisesti poista hauraan liitoskohtamäärityksen ongelmaa, vaan ongelma siirtyy aspektien kehittäjiltä perusohjelmaan kehittäjille. Kehittäjän on oltava tietoinen kaikista XPI-rajapinnoista, ymmärrettävä niiden toiminta ja ylläpidettävä niitä perusohjelman kehittämisen aikana. Kehittäjä ei voi enää keskittyä yhden näkökulman kehittämiseen, sillä hänen on otettava huomioon myös ohjelmaan vaikuttavat aspektit. XPI-lähestymistapa ei siten ole paras mahdollinen ratkaisu näkökulmien erotteluun.

XPI-rajapinnan sopimusta ei voida läheskään aina valvoa automaattisesti, sillä AspectJ:n käännösvirheiden esittely –mekanismi ei ole ominaisuuksiltaan kovinkaan monipuolinen. Se ei esimerkiksi toimi tilanteissa, joissa sopimusrikkomukset paljastuvat vasta ohjelmaa suoritettaessa. Mekanismia voidaan käyttää vain, kun virheet voidaan havaita ohjelmaa suorittamatta.

6.4 Liitoskohtien annotointi

Annotointiin perustuvat liitoskohtamääritykset poimivat liitoskohtia niihin liitettyjen lisätietojen eli annotaatioiden perusteella. Annotaatiot kuvaavat liitoskohdan luonnetta tai tarpeita ohjelmakoodia korkeammalla tasolla. Esimerkiksi transaktiossa suoritettava metodi voidaan merkitä lisätiedolla transaktionaaliseksi. Annotaatioiden merkintä ja ylläpito on perusohjelmana kehittäjien vastuulla.

Tutkitaan annotaatioiden käyttökelpoisuutta luvun 5 grafiikkaeditorin avulla. Grafiikkaeditorin näytön päivitys hoidetaan aspektilla, joka päivittää näytön aina kun muodon tila muuttuu. Annotaatioihin perustuvassa ratkaisussa muotojen tilaa muuttavat metodit merkitään tätä luonnetta kuvaavalla annotaatiolla, esimerkiksi annotaatiolla @TilaaMuuttava. Kuvassa 6.10 grafiikkaeditorin Pisteluokan tilaa muuttavat metodit on annotoitu @TilaaMuuttava-annotaatioilla. Näitä metodeita ei poimita enää nimeämiskäytännön perusteella, vaan liitoskohtamääritys poimii @TilaaMuuttava-annotaatiolla merkityt metodit. Tällä tavalla toteutettu liitoskohtamääritys on esitetty kuvassa 6.11.

```
public class Piste implements Muoto {
    int x, y;

    public int annaX() {
        return x;
    }

    public int annaY() {
        return y;
    }

    @TilaaMuuttava
    public void asetaX(int x) {
        this.x = x;
    }

    @TilaaMuuttava
    public void asetaY(int y) {
        this.y = y;
    }
}
```

Kuva 6.10: Piste-luokan toteutus, jossa pisteen tilaa muuttavat metodit on merkitty @TilaaMuuttava-annotaatiolla.

```
public aspect NäytönPäivittäjä {

    pointcut tilaMuuttunut() :
        execution(@TilaaMuuttava * *(..));

    after() : tilaMuuttunut() {
        Näyttö.päivitä();
    }
}
```

Kuva 6.11: Annotaatioihin perustuva NäytönPäivittäjä-aspektin toteutus. Aspektin liitoskohtamääritys poimii kaikkien @TilaaMuuttava-annotaatiolla merkittyjen metodien suorituksen.

Bonérin mukaan annotaatioita käyttäessä tulisi varoa ”neuvo minua tässä”-tyyliä, jossa annotaatio kertoo suoraan toiminnallisuuden, jota annotoitu elementti on vailla [Bon06]. Esimerkiksi metodien @Transaktionaalinen-annotaatio viittaa siihen, että näistä metodeista on tehtävä transaktionaalisia. Tällaisia annotaatioita on vaikea ja epäluontevaa käyttää hyväksi muissa aspekteissa, koska nämä annotaatiot eivät kerro elementin luonteesta. Tämän vuoksi Bonér ehdottaa, että annotaatiot kuvaisivat vain annotoidun elementin luonnetta, kuten roolia, toimintaa, vastuuta tai ominaisuuksia. Näin näitä annotaatioita voidaan käyttää hyväksi myös muissa aspekteissa. Kuvassa 6.10 käytetty @TilaaMuuttava-annotaatio on siten parempi kuin esimerkiksi @PäivitäNäyttö, sillä muutkin aspektit voivat poimia tilan muutoksia.

Annotaatioiden käyttö ratkaisee hauraan liitoskohtamäärityksen ongelman vain osittain [KMB06b]. Liitoskohtamääritykset ovat nyt kestävämpiä, koska ne poimivat liitoskohtia korkean tason semanttisten ominaisuuksien perusteella sen

sijaan, että ne yrittäisivät päätellä merkitystä ohjelman syntaksin ja rakenteen perusteella. Hauraan liitoskohtamäärityksen ongelma on kuitenkin vaihtunut annotaatioiden ylläpito-ongelmaksi. Liitoskohtamääritysten kestävyys on nyt kiinni perusohjelman annotoinnin tarkkuudesta. Perusohjelman puuttuvat tai virheelliset annotaatiot johtavat rikkoutuneisiin liitoskohtamäärityksiin.

Annotaatioiden ylläpito -ongelman ratkaisuksi on ehdotettu aspektilähtöistä ratkaisua. Aspektit voivat annotoida perusohjelman jäseniä, jolloin annotaatioiden ylläpito voidaan siirtää pois perusohjelman kehittäjiltä aspektin kehittäjälle. Aspekteilla tehtävä annotointi ei kuitenkaan ratkaise ongelmaa, koska annotoitavat kohdat joudutaan poimimaan perinteisesti.

6.5 Liitoskohtamääritysten muutosanalyysi

Liitoskohtamääritysten muutosanalyysi (pointcut delta analysis) lähestyy hauraan liitoskohtamäärityksen ongelmaa analysoimalla liitoskohtien poiminnassa tapahtuneet muutokset [KoS04]. Analyysi kuvaa sekä liitoskohtien *uudet poiminnat* että liitoskohtien *poistuneet poiminnat*. Uusia poimintoja ovat ohjelman muutosten seurauksena poimitut liitoskohdat, joita ei ole poimittu ennen muutosta. Poistuneita poimintoja ovat taas liitoskohtamääritysten ennen muutosta poimimat liitoskohdat, joita ei enää muutoksen jälkeen poimita.

Liitoskohtamääritysten muutosanalyysi soveltuu hauraiden liitoskohtamääritysten ylläpitoon, sillä ohjelmoija voi tarkistaa muutosanalyysin avulla liitoskohtamääritysten poiminnan oikeellisuuden. Muutosanalyysin avulla voidaan tarkistaa, että liitoskohtien poiminnassa tapahtuneet muutokset eivät sisällä liitoskohtien vääriä ja puuttuvia poimintoja. Vääriä poimintoja voi olla uusien poimintojen joukossa ja puuttuvia poimintoja voi olla poistuneiden poimintojen joukossa. Muutosanalyysi helpottaa ylläpitoa, sillä liitoskohtamääritysten oikeellisuus voidaan tarkistaa vain muutosten osalta.

Muutosanalyysi ei varsinaisesti ratkaise hauraan liitoskohtamäärityksen ongelmaa, vaan sillä voidaan lieventää ongelman vaikutuksia. Muutosanalyysi on kuitenkin hyödyllinen nykyisten ongelmasta kärsivien aspektikielien kanssa käytettynä, kunnes hauraan liitoskohtamäärityksen ongelmaan saadaan lopullinen ratkaisu.

Muutosanalyysi perustuu ohjelmasta otettaviin tilannekuviin, joihin tallennetaan tieto liitoskohtamääritysten sillä hetkellä poimimista liitoskohdista. Ohjelmasta otetaan tilannekuva ennen siihen tehtäviä muutoksia ja muutosten jälkeen. Näitä tilannekuvia vertaamalla saadaan kuva siitä, miten liitoskohtien poiminta on muuttunut perusohjelman muutoksien vaikutuksesta.

Muutosanalyysi sisältää kaksi heikkoutta: sen avulla ei voida selvittää kaikkia puuttuvia poimintoja ja muutosanalyysit ovat epätarkkoja dynaamisten liitoskohtamääritysten poimimien liitoskohtien osalta. Muutosanalyysin avulla voidaan havaita vain sellaisten liitoskohtien puuttuvat poiminnat, jotka on poimittu ennen muutosta, mutta joita ei poimita enää muutoksen jälkeen. Muutosanalyysi ei havaitse sellaisten liitoskohtien puuttuvia poimintoja, joita ei ole poimittu ennen ohjelmamuutostakaan. Tämä johtuu siitä, että muutosanalyysin näkökulmasta poimittujen liitoskohtien joukko ei ole muuttunut.

Muutosanalyysi on tarkka vain kun analysoitavana ovat staattiset liitoskohtamääritykset, joiden poimimat liitoskohdat voidaan päätellä staattisesti, ohjelmaa suorittamatta. Dynaamisten liitoskohtamääritysten poimimat liitoskohdat tunnetaan staattisesti, mutta poiminta tiedetään varmaksi vasta suoritusaikaisen tiedon perusteella. Muutosanalyysi approksimoi dynaamisten liitoskohtamääritysten poiminnan siten, että se katsoo jokaisen liitoskohdan poimituksi, vaikka näin ei suoritusaikana välttämättä tapahtuisi. Muutosanalyysi ei ole siis tarkka dynaamisten liitoskohtamääritysten osalta, mikä on yksi muutosanalyysin heikkous.

Muutosanalyysiin löytyy ainakin kaksi välinettä: PCDiff [KoS04] ja Eclipsen AJDT-lisäosaan sisältyvä muutosanalyysiväline. Seuraavissa kahdessa luvussa esitetään nämä muutosanalyysivälineet PCDiffistä aloittaen.

6.5.1 PCDiff

PCDiff-työväline tarjoaa näkymän liitoskohtamääritysten poiminnassa tapahtuneisiin muutoksiin [KoS04]. PCDiff kertoo, miten liitoskohtamääritysten poimimien liitoskohtien joukko muuttui perusohjelman muutoksien seurauksena. Työväline on kehittäjiensä mukaan tarkoitettu juuri hauraiden liitoskohtamääritysten ylläpidon apuvälineeksi.

PCDiff-työväline on Eclipse-kehitysympäristöön toteutettu lisäosa (plugin), joka on riippuvainen AJDT-lisäosasta. AJDT (The AspectJ Development Tools) on Eclipsen lisäosa, joka tarjoaa AspectJ:n kehitysvälineiden tuen Eclipse-alustalla. Näiden riippuvuuksien vuoksi PCDiffillä voidaan analysoida vain AspectJ:llä toteutettuja ohjelmia ja vain Eclipse-ympäristössä.

PCDiffin toiminta perustuu perusohjelmasta otettuihin tilannekuviin (snapshot). Tilannekuva sisältää kuvauksen liitoskohtien joukosta, jonka sen hetkiset liitoskohtamääritykset poimivat. PCDiff ei itse päättele poimittuja liitoskohtia, vaan se saa tiedot poimituista liitoskohdista AJDT-lisäosalta ja ajc-kääntäjältä. Ohjelman eri versioista otettuja tilannekuvia vertaamalla saadaan tietoa liitoskohtien poiminnassa tapahtuneista muutoksista. PCDiff tarjoaa kuvauksen tilannekuvien eroista Eclipsen tehtävänäkymässä (tasks view), pcdiff-näkymässä (pointcut diff

view) ja ohjelmakoodin muokkausnäkymässä. Kaikki näkymät näyttävät muutoksien seurauksena sekä kadonneet liitoskohdat että liitoskohtamääritysten poimimat uudet liitoskohdat.

Seuraavassa näytetään, miten PCDiff näyttää liitoskohtamääritysten poiminnassa tapahtuneet muutokset. Esimerkki pohjautuu luvun 5.1 esimerkkisovellukseen, joka koostuu erilaisista muodoista, kuten pisteestä ja janasta, sekä aspektista, joka päivittää näyttöä muotojen tilan muutoksien jälkeen. Aspektin poiminta perustui nimeämiskäytäntöön. Se poimii kaikki aseta-alkuisten metodien suoritukset, jotka on esitelty *Muoto*-rajapinnan toteuttavissa luokissa. Kuvassa 6.12 on esitelty grafiikkaeditorin *Piste*-luokasta kaksi eri versiota. Versio 1 on identtinen luvun 5.1 *Piste*-luokan kanssa. Versio 2 on *Piste*-luokan uusi versio. Siitä on poistettu *asetaX*- ja *asetaY*-metodit, jotka on korvattu uusilla *asetaSijainti*- ja *siirry*-metodeilla.

Versio 1

```
public class Piste
        implements Muoto {

    int x, y;

    public int annaX() {
        return x;
    }

    public int annaY() {
        return y;
    }

    public void asetaX(int x) {
        this.x = x;
    }

    public void asetaY(int y) {
        this.y = y;
    }
}
```

Versio 2

```
public class PisteV2
        implements Muoto {

    int x, y;

    public int annaX() {
        return x;
    }

    public int annaY() {
        return y;
    }

    public void asetaSijainti(
            int x, int y) {
        this.x = x;
        this.y = y;
    }

    public void siirry(int dx,
            int dy) {
        this.x += dx;
        this.y += dy;
    }
}
```

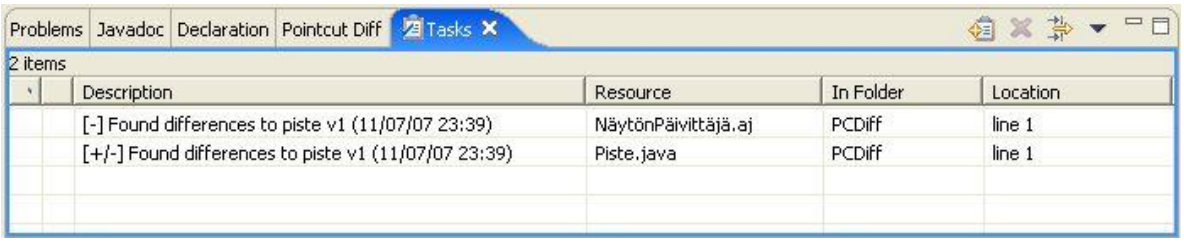
Kuva 6.12: Grafiikkaeditorin *Piste*-luokan kaksi eri versiota.

Otetaan PCDiffillä tilannekuvaus esimerkkisovelluksesta, joka sisältää *Piste*-luokasta version 1 mukaisen toteutuksen. Annetaan tälle tilannekuvaukselle *pisteV1*-nimi. *Piste*-luokka muutetaan tämän jälkeen version 2 mukaiseksi. Lopuksi version 2 sisältävästä sovelluksesta otetaan tilannekuvaus, jolle annetaan nimeksi *pisteV2*.

Tutkitaan tilannekuvausten eroja ensin Eclipsen tehtävänäkymässä (kuva 6.13). Tehtävänäkymä näyttää miten tilannekuvaukset *pisteV1* ja *pisteV2* eroavat toisistaan. Tehtävänäkymä näyttää eri tiedostoissa tapahtuneet muutokset. Ensimmäi-

nen tehtävärivi liittyy *NäytönPäivittäjä*-aspektiin, jonka kuvauksen alun miinusmerkki kertoo liitoskohtamäärityksen poiminnan muutoksesta. Seuraava rivi kuvaa *Piste*-luokassa tapahtuneita muutoksia. Kuvauksen plus- ja miinusmerkki kertovat, että luokasta on poimittu uusia liitoskohtia ja joitain ennen poimittuja liitoskohtia ei enää poimita.

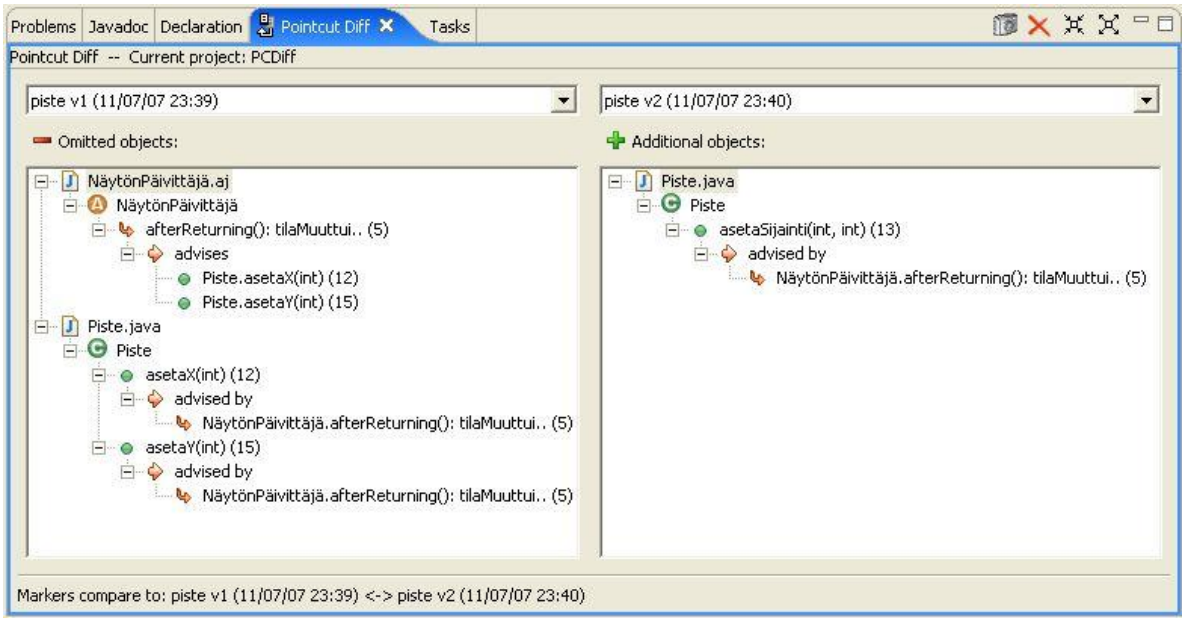
Tehtävänäkymän rivi toimii linkkinä ohjelmakoodin muokkaus –näkymään, jossa muutoksia voi tarkastella tarkemmin. Tehtävänäkymä tukee liitoskohtamääritysten validointia, sillä tehtävät voidaan kuitata suoritetuiksi, kun tiedosto on katselmoitu.



Kuva 6.13: Tehtävänäkymän sisältö, kun tilannekuvauksia piste v1 ja piste v2 verrataan toisiinsa.

Pcdiff-näkymä (kuva 6.14) antaa yksityiskohtaisemman kuvan muutoksen vaikutuksista. Näkymä on jaettu kahteen osaan, joista vasemman puoleinen sisältää *pisteV1* tiedot ja oikean puoleinen *pisteV2* tiedot. Vasemmalla puolella näytetään, mitä liitoskohtamäärityksen *Piste*-luokan versiosta 1 poimimista liitoskohdista ei poimita enää versiossa 2. Näkymä vastaa muutosta, sillä *Piste*-luokan *asetaX*- ja *asetaY*-metodeja ei enää poimita versiossa 2, koska ne on poistettu. Näkymän oikealla puolella taas näytetään, mitä uusia liitoskohtia liitoskohtamääritys poimii versiosta 2. Näkymä on ajan tasalla, sillä *asetaSijainti*-metodi lisättiin versiossa 2, ja sitä ei näin ollen ole poimittu versiossa 1.

Siirry-metodi demonstroi yhtä muutosanalyysin heikkouksista: liitoskohdan puuttuvaa poimintaa ei havaita, jos liitoskohtaa ei ole poimittu aikaisemmin. *Siirry*-metodia ei ole voitu poimia *Piste*-luokan ensimmäisestä versiosta, sillä metodia ei ole vielä silloin ollut olemassa. Näin ensimmäisestä versiosta otettu tilannekuva ei sisällä *siirry*-metodin poimintaa. *Piste*-luokan toiseen versioon lisätty *siirry*-metodi on aspektin odottaman nimeämiskäytännön vastainen, joten sitä ei poimita. *Siirry*-metodi aiheuttaa kuitenkin pisteen tilan muutoksen, joten se olisi poimittava. *Piste*-luokan toisesta versiosta otettu tilannekuva ei siten sisällä *siirry*-metodin poimintaa. *Siirry*-metodin puuttuvaa poimintaa ei siten huomata, koska tilannekuvissa ei ole tapahtunut muutosta *siirry*-metodin osalta.



Kuva 6.14: Pcdiff-näkymän sisältö, kun tilannekuvauksia piste v1 ja piste v2 verrataan toisiinsa.

6.5.2 AJDT:n jäljitysväline

AJDT sisältää PCDiffin kaltaisen jäljitysvälineen, joka jäljittää liitoskohtamääritysten poiminnassa tapahtuneita muutoksia. Jäljitysväline on käytettävissä AJDT:n versiosta 1.2.1 lähtien. Jäljitysväline toimii samalla tavalla kuin PCDiff: ohjelman eri versioista otetaan tilannekuvia, joita verrataan tähän kehitetyllä näkymällä.

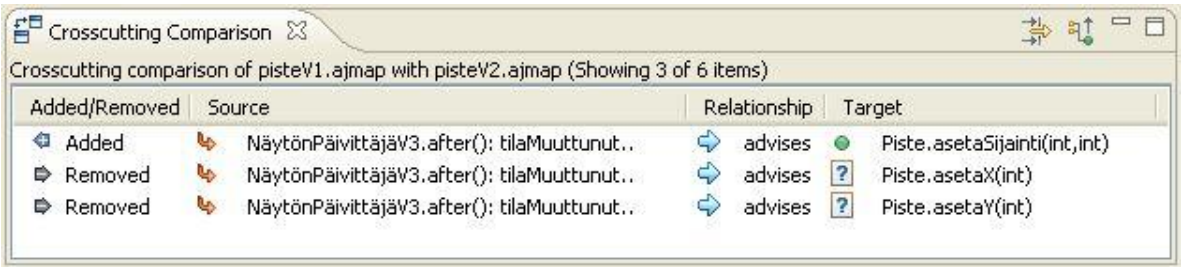
Seuraavassa kuvataan AJDT 1.4 versioon sisältyvän jäljitysvälineen toiminta. Esimerkissä näytetään, miten jäljitysväline näyttää liitoskohtamäärityksessä tapahtuneet muutokset, kun grafiikkaeditorin *Piste*-luokalle tehdään PCDiffiä käsittelevässä luvussa kuvassa 6.12 esitetyt muutokset.

Grafiikkaeditorisovelluksesta otetaan tilannekuva ennen *Piste*-luokaan tehtäviä muutoksia. Tämä tilannekuvan nimeksi annetaan *pisteV1*, jolle AJDT lisää automaattisesti ”ajmap”-päätteen. Tilannekuvan tallentavan tiedoston lopullinen nimi on siten *pisteV1.ajmap*. Seuraava tilannekuva otetaan uudestaan perusohjelmaan tehtyjen muutosten jälkeen ja se tallennetaan *pisteV2.ajmap*-tiedostoon.

Tilannekuvien eroja tarkastellaan jäljitysvälineen tarjoamalla läpileikkauksen vertailu –näkymällä (crosscutting comparison view). Käyttäjä valitsee molemmat tiedostot Eclipsen resurssinäkymästä ja valitsee kontekstimenusta läpileikkauksien vertailun. Vertailun tulos näytetään läpileikkauksen vertailu -näkymässä (kuva 6.15).

Läpileikkauksen vertailu –näkymä näyttää selvästi *NäytönPäivittäjäV3*-aspektin poiminnassa tapahtuneet muutokset. Ensimmäisen ja toisen tilannekuvan välillä on tapahtunut kolme muutosta: aspekti neuvoo uutta *Piste*-luokan *asetaSijainti-*

metodia (ensimmäinen rivi) ja aspekti ei enää neuvo *Piste*-luokan *asetaX*- ja *asetaY*-metodeja (kaksi viimeistä riviä).



Kuva 6.15: Läpileikkauksen vertailu –näkymän sisältö kun *pisteV1.ajmap* ja *pisteV2.ajmap* tilannekuvia verrataan toisiinsa.

6.5.3 PCDiff vs. AJDT:n jäljitysväline

PCDiff ja AJDT:n jäljitystyökalu tarjoavat samankaltaisen toiminnallisuuden. Näistä kahdesta AJDT:n jäljitystyökalu on intuitiivisempi käyttää. PCDiff on taas tällä hetkellä ominaisuuksiltaan monipuolisempi.

PCDiffin viimeisin versio 1.0.0 on julkaistu kesällä 2004. Se on yhteensopiva Eclipse 3.0 -version kanssa. AJDT sisältää jäljitystyökalun AJDT:n versiosta 1.2.1 lähtien, joten jäljitystyökalu on saatavissa Eclipsen eri versioihin versiosta 3.0 lähtien.

AJDT:n jäljitystyökalu on näistä suositeltavampi valinta. Se tukee Eclipsen eri versioita ja sen kehitystyö on aktiivista. Jos käytössä on jokin Eclipsen 3-versio lukuun ottamatta 3.0-versiota, on AJDT:n jäljitystyökalu ainoa vaihtoehto.

6.6 Ratkaisujen yhteenveto

Edellä esitetyiden ratkaisujen selkein ero on niiden suhtautuminen tietämättömyyteen. AspectJ:n laajennukset ja metalogiikkaohjelmointi säilyttävät perusohjelman kehittäjän tietämättömyyden: ne eivät edellytä, että perusohjelman kehittäjä on tietoinen aspekteista. Ne tukevat siis tietämätöntä ohjelmointitapaa. Sopimukseen perustuvat ratkaisuissa taas luovutaan tietämättömyydestä. Ratkaisut siirtävät aspektien ja perusohjelman välisen rajapinnan ylläpitovastuun perusohjelman kehittäjälle, joka tulee näin tietoiseksi aspekteista. Liitoskohtamääritysten muutosanalyysi ei ota varsinaisesti kantaa tietämättömyyteen. Perusohjelman kehittäjä pysyy tietämättömänä, jos aspektiohjelmoija korjaa liitoskohtamääritykset muutosanalyysin avulla aina perusohjelman muutosten jälkeen. Perusohjelman kehittäjä tulee taas tietoiseksi aspekteista, jos hänen on varmistettava liitoskohtamääritysten eheys muutosanalyysillä.

Tietämättömyys on kiistanalainen aihe aspektiohjelmoinnin tutkimusyhteisössä. Toiset pitävät tietämättömyyttä aspektiohjelmoinnille luonteenomaisena [FiF01, ChL03], kun taas toiset olisivat valmiita luopumaan tietämättömyydestä [GSS06]. Filman ja Friedman ovat tietämättömyyden puolestapuhujia ja he liittävät tietämättömyyden määrän aspektikielen laatuun: mitä tietämättömämpi perusohjelman kehittäjä, sitä parempi aspektikieli. Griswold et al. ovat puolestaan sitä mieltä, että tietämättömyyden tavoittelu johtaa hauraan liitoskohtamäärityksen ongelmaan.

Tietämättömyys on kaikesta huolimatta hyödyllinen ja tavoittelemisen arvoinen ominaisuus, vaikka nykyisillä liitoskohtamäärityskielillä tavoiteltu tietämättömyys voi johtaa hauraan liitoskohtamäärityksen ongelmaan. Tietämättömyyden ansiosta perusohjelman kehittäjä voi keskittyä yhden ongelman ratkaisemiseen kerrallaan. Tämän lisäksi hänen ei tarvitse opetella aspektikieltä, jonka käsitteistö ja ohjelmointimalli eroavat huomattavasti perinteisistä ohjelmointikielistä. Näin käsitteistön ymmärtämiseen ja kielen opetteluun voi kulua paljon aikaa.

Hauraan liitoskohtamäärityksen ongelmaan voisi löytyä ratkaisu myös aspektiohjelmointia tukevista refaktorointivälineistä. Liitoskohtamääritysten ylläpito voitaisiin liittää refaktorointivälineisiin, jotka pitäisivät liitoskohtamääritykset eheinä perusohjelman muutoksista huolimatta. Aspektiohjelmointia tukevia refaktorointivälineitä ei ole vielä saatavilla, mikä johtuu osaksi siitä, että sellaisen toteuttaminen on erittäin vaikeaa. Ensinnäkin ne ovat käyttökelpoisia vain, kun käyttäjä lisää, poistaa tai muuttaa ohjelmakoodia refaktorointivälineiden avulla. Toiseksi refaktorointivälineet voivat vain säilyttää liitoskohtamäärityksen poimimien liitoskohtien joukon muutoksesta toiseen, koska ne eivät voi ohjelmallisesti

päätellä liitoskohtamääritys todellista tarkoitusta. Ne eivät osaa siten sanoa, pitäisikö uusi liitoskohta poimia vai ei.

Hauraan liitoskohtamäärityksen ongelmaan etsitään ratkaisua pääasiassa ilmaisuvoimaisemmista aspektikielistä [OMB05]. Metalogiikkakielet ja AspectJ:n laajennukset ovat tällaisia ilmaisuvoiman kasvattamiseen tähtääviä yrityksiä. Erilaisten laajennusten ongelma on kuitenkin siinä, että ne tekevät aspektikielestä monimutkaisempia oppia ja käyttää. AspectJ sisältää jo nykyään 20 erilaista primitiiviä, joiden opetteluun kuluu runsaasti aikaa.

Aspektikielten ilmaisuvoiman kasvattamisessa on ollut näkyvissä myös suuntaus, jossa palataan takaisin explisiittiseen metaohjelmointiin. Metaohjelmoinnissa käsittellään ja muokataan suoraan muita ohjelmia ja ohjelman rakenteita, minkä ansiosta metaohjelmilla voidaan vaikuttaa ohjelmaan ilman rajoituksia [Kiz04]. Metaohjelmointi on siten selvästi ilmaisuvoimaisempi kuin aspektiohjelmointi. Kiczalesin mukaan aspektiohjelmoinnissa haluttiin pois explisiittisestä metaohjelmoinnista, koska rajoittamattomalla metaohjelmoinnilla johtivat helposti vaikeasti luettaviin ja ylläpidettäviin ratkaisuihin, mitkä haluttiin poistaa rajoitetummalla aspektiohjelmoinnilla. Aspektiohjelmoinnin ilmaisuvoiman kehittämisessä tulisi siis välttää paluu metaohjelmointiin, vaikkakin se voisi tarjoa ratkaisun liitoskohtamääritysten haurauteen.

7 Yhteenveto

Tässä tutkielmassa esiteltiin aspektiohjelmointi ja AspectJ-aspektikieli, jota pidetään nykyään aspektiohjelmoinnin de facto –standardina. AspectJ oli ensimmäinen aspektikieli, joka perustui liitoskohtamalliin. Sen jälkeen kehitetyt aspektikielet, kuten Spring AOP ja JBoss AOP, ovat myös omaksuneet AspectJ:n liitoskohtamallin.

Liitoskohtamalli on aspektikielen tärkein osa, sillä se määrittää aspektikielen ilmaisuvoiman. Liitoskohtamallin määrittelemät liitoskohdat vaikuttavat siihen, mitä kohtia ohjelmasta voidaan poimia, ja liitoskohtamäärityskieli asettaa rajat sille, millä perusteella nuo kohdat voidaan poimia. Tutkielmassa on näytetty, miten AspectJ:n nykyinen liitoskohtamalli ei ole riittävän ilmaisuvoimainen. AspectJ:n liitoskohtamäärityksillä ei voida aina poimia liitoskohtia haluttujen ominaisuuksien perusteella, jolloin poimintaehto on ilmaistava muilla ominaisuuksilla. Yleensä liitoskohtamäärityksissä turvaudutaan liitoskohtien allekirjoituksiin, jotka poimitaan nimeämiskäytännön perusteella. Aspektiohjelmoinnissa tavoitellaan tietämättömyyttä, joten nimeämiskäytännöt eivät velvoita perusohjelman kehittäjää, jotka eivät ole niistä tietoisia.

Tutkielman luvussa 5 näytettiin, miten nimeämiskäytäntöön perustuvat liitoskohtamääritykset rikkoutuvat, kun aspekteista tietämätön perusohjelman kehittäjä tekee nimeämiskäytännön vastaisia muutoksia. Näitä liitoskohtamäärityksiä kutsutaan hauraiksi, sillä ne rikkoutuvat, vaikka itse liitoskohtamäärityksiin ei tehdä muutoksia. Hauraat liitoskohtamääritykset vaikeuttavat perusohjelman kehitettävyyttä ja ylläpidettävyyttä, sillä muutokset voivat liitoskohtamäärityksiä, joiden eheys olisi tarkistettava aina muutoksien jälkeen.

Luvussa 6 esitettiin ratkaisuja hauraan liitoskohtamäärityksen ongelmaan. Ratkaisut valittiin siten, että niitä voidaan soveltaa AspectJ:ssä. Ratkaisut jaettiin kolmeen ryhmään: aspektikielen ilmaisuvoimaa kasvattaviin ratkaisuihin, sopimukseen pohjautuviin ratkaisuihin ja liitoskohtamäärityksen muutosanalyysin perustuviin ratkaisuihin.

Aspektikielen ilmaisuvoiman kasvattaminen on luonteva ratkaisu hauraan liitoskohtamäärityksen ongelmaan. Liitoskohtamääritykset ovat kestävämpiä, koska poimintaehto voidaan perustaa liitoskohdan allekirjoitusta vakaampiin ominaisuuksiin. Tutkielmassa esitettiin kaksi AspectJ:n ilmaisuvoimaa parantavaa ratkaisua: abc-kääntäjän laajennukset ja LogicAJ-metalogiikkakieli. Abc-kääntäjään toteutetut AspectJ-kielen laajennukset kasvattavat ilmaisuvoimaa tarjoamalla uusia liitoskohtia ja niiden poimimiseen tarkoitettuja primitiivejä. Abc-kääntäjä sisältää kuitenkin muutamia heikkouksia, joita ovat muun muassa olematon työvälinetuki ja puuttuva Java 5 ja AspectJ 5 –versioiden tuki. LogicAJ-

metalogiikkakieli korvaa AspectJ:n liitoskohtamääritys-kielen ilmaisuvoimaisemmalla kielellä, joka perustuu meta- ja logiikkaohjelmointiin. Metalogiikkakielellä laadittu poimintaehto voidaan kutoa tehokkaasti ennen ohjelman suoritusta, kun taas AspectJ:llä on vastaavissa tapauksissa turvauduttava hitaaseen ja vaikeasti ymmärrettävään suoritusaikaiseen reflektioon.

Sopimukseen perustuvissa ratkaisuissa aspektien ja perusohjelman väliin laaditaan rajapinta, joka peittää perusohjelman yksityisen ja epävakaan toteutuksen. Aspektit voivat neuvoa vain rajapintaa, jonka ylläpito on perusohjelman kehittäjän vastuulla. Rajapinta tarjoavat vakaan alustan aspekteille, joiden liitoskohtamääritykset eivät enää rikkoudu perusohjelman muutosten vuoksi. Tutkielmassa on esitetty annotaatioihin ja XPI-rajapintoihin perustuvat ratkaisut. Näiden ratkaisuja voidaan soveltaa nykyisissä aspektikielissä, mikä on näiden ratkaisujen etu. Molempien ratkaisuiden heikkoutena on se, että ne tekevät perusohjelman kehittäjän tietoiseksi aspekteista.

Liitoskohtamääritysten muutosanalyysi tarjoaa välineen, jolla voidaan jäljittää liitoskohtamääritysten poiminnassa tapahtuneet muutokset. Muutosanalyysi ei siten poista hauraan liitoskohtamäärityksen ongelmaa, mutta se auttaa hallitsemaan sitä. Muutosanalyysin avulla aspektiohjelmoija tai perusohjelman kehittäjä voi tarkistaa, että liitoskohtamääritykset eivät ole rikkoutuneet perusohjelman muutosten seurauksena. Muutosanalyysiä voidaan käyttää nykyisten aspektikielten kanssa, ja sitä on mahdollistaa tietämättömän aspektiohjelmoinnin. Muutosanalyysillä ei voida kuitenkaan havaita kaikkia liitoskohtamääritysten virheitä, mikä on yksi muutosanalyysin heikkous.

Tällä hetkellä hauraan liitoskohtamäärityksen ongelmaan ei ole ratkaisua. Aspektikielten ilmaisuvoima kasvaa kuitenkin koko ajan, mikä voi tarjota aikanaan ratkaisun ongelmaan ratkaisun tai vähentää ongelman merkitystä. Ilmaisuvoimaisempia aspektikieliä odotellessa ongelmaa voidaan hallita sopimuksiin ja muutosanalyysiin perustuvilla ratkaisuilla.

Lähteet

AAC05 Allan, C., Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. ja Tibble, J., Adding trace matching with free variables to AspectJ. *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, San Diego, CA, USA, lokakuu 2005, sivut 345-364.

ACH05 Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhoták, J., Lhoták, O., de Moor, O., Sereni, D., Sittampalam, G. ja Tibble, J., abc: an extensible AspectJ compiler. *Proceedings of the 4th international Conference on Aspect-Oriented Software Development*, Chicago, Illinois, maaliskuu 2005, sivut 87-98.

AId04 Aldrich, J., Open Modules: a proposal for modular reasoning in aspect-oriented programming. Carnegie Mellon Technical Report CMU-ISRI-04-108, maaliskuu 2004.

AIm03 Almaer, D., Tech talk with Gregor Kiczales on AOP. http://www.theserverside.com/tt/talks/videos/GregorKiczalesText/interview.html, heinäkuu 2003, [18.8.2007].

AOS07 aosd.net, Tools for developers. http://aosd.net/wiki/index.php?title=Tools_for_Developers, 2007, [18.8.2007]

AWB93 Aksit, M., Wakita, K., Bosch, J., Bergmans, L., ja Yonezawa, A. Abstracting Object Interactions Using Composition Filters. *Proceedings of the Workshop on Object-Based Distributed Programming*, Kaiserslautern, Germany, heinäkuu 1993, sivut 152-184.

BeR00 Bennett, K. H. ja Rajlich, V. T., Software maintenance and evolution: a roadmap. *Proceedings of the Conference on the Future of Software Engineering*, Limerick, Ireland, kesäkuu 2000, sivut 73-87.

Bon06 Boner, J., Domain-Driven Pointcut Design. http://jonasboner.com/2006/04/24/domain-driven-pointcut-design/, 2006, [19.8.2007].

BrH05 Brichau, J. ja Haupt, M., Survey of aspect-oriented languages and execution models. Technical Report AOSD-Europe-VUB-01, AOSD-Europe, toukokuu 2005.

BrR07 Breuel, C. ja Reverbel, F., Join point selectors. *Proceedings of the 5th Workshop on Engineering Properties of Languages and Aspect Technologies*, Vancouver, British Columbia, Canada, maaliskuu 2007,

artikkeli numero 3.

CaP06 Cazzola, W. ja Pini, S., Join Point Patterns: a high-level join point selection mechanism. *Model Driven Engineering Languages and Systems Workshops*, Genova, Italy, lokakuu 2006, sivut 17-26.

CCH04 Colyer, A., Clement, A., Harley, G. ja Webster, M., Eclipse AspectJ: aspect-oriented programming with AspectJ and the Eclipse AspectJ Development Tools. Addison-Wesley Professional, Boston, MA, 2004.

ChC06 Chen, K. ja Chien, C.H., Extending the Field Access Pointcuts of AspectJ to Arrays. *International Workshop on Software Engineering, Databases, and Knowledge Discovery, in International Computer Symposium*, Taipei, Taiwan, joulukuu 2006, artikkeli numero 16.

ChL03 Chavez, C. ja Lucena, C. A theory of aspects for aspect-oriented development. *Proceedings 17th Brazilian Symposium on Software Engineering*, Manaus/Amazonas, Brazil, lokakuu 2003, sivut 130-145.

CJR06 Cazzola, W., J´ez´equel, J.-M. ja Rashid, A., Semantic join point models: motivations, notions and requirements. *Proceedings of the Software Engineering Properties of Languages and Aspect Technologies Workshop (SPLAT 06)*, Bonn, Germany, maaliskuu 2006.

CPA06 Cazzola, W., Pini, S. ja Ancona, M., Design-Based Pointcuts Robustness Against Software Evolution. *Proceedings of the 3rd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE 06), in 20th European Conference on Object-Oriented Programming (ECOOP 06)*, Nantes, France, heinäkuu 2006, sivut 35–45.

Dij82 Dijkstra, E. W., On the role of scientific thought. Selected Writings on Computing: A Personal Perspective, Springer-Verlag, 1982, sivut 60-66.

EAK01 Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., ja Ossher, H., Discussing aspects of AOP. *Communications of the ACM*, 44, 10 (2001), sivut 33-38.

FiF01 Filman, R. E. ja Friedman, D. P., Aspect-oriented programming is quantification and obliviousness. RIACS Technical Report 01.12, toukokuu 2001.

FVS06 De Fraine, B., Vanderperren, W. ja Suvee, D., Motivations for framework-based AOP. *Proceedings of Open and Dynamic Aspect Languages Workshop, in AOSD conference*, Bonn, Germany, maaliskuu 2006, artikkeli numero 2.

Gyb01 Gybels, K., Aspect-oriented programming using a logic meta programming language to express cross-cutting through a dynamic joinpoint structure. Bachelors thesis, Programming Technology Lab, Vrije Universiteit Brussel, elokuu 2001.

Gyb02 Gybels, K., Using a logic language to express cross-cutting through dynamic joinpoints. In Proceedings of the Second German Workshop on Aspect-Oriented Software Development, Technical Report IAI-TR-2002-1. Universität Bonn, 2002.

GyB03 Gybels, K. ja Brichau, J., Arranging language features for more robust pattern-based crosscuts. In Proceedings of the 2nd international Conference on Aspect-Oriented Software Development, Boston, MA, maaliskuu 2003, sivut 60-69.

GSS06 Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y. ja Rajan, H., Modular software design with crosscutting interfaces. *IEEE Software*, 23, 1 (2006), sivut 51-60.

GuK01 Gudmundson, S. ja Kiczales, G., Addressing practical software development issues in AspectJ with a pointcut interface. *Proceedings of Workshop on Advanced Separation of Concerns in Object-Oriented Systems, in ECOOP 2001,* Budabest, Hungary, kesäkuu 2001.

HaG06 Harbulot, B. ja Gurd, J. R., A join point for loops in AspectJ. *Proceedings of the 5th international Conference on Aspect-Oriented Software Development*, Bonn, Germany, maaliskuu 2006, sivut 63-74.

HaK02 Hannemann, J. ja Kiczales, G., Design pattern implementation in Java and AspectJ. *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Seattle, Washington, USA, marraskuu 2002, sivut 161-173.

HiH04 Hilsdale, E. ja Hugunin, J., Advice weaving in AspectJ. *Proceedings of the 3rd international Conference on Aspect-Oriented Software Development*, Lancaster, UK, maaliskuu 2004, sivut 26-35.

JBo07 JBoss.org, JBoss AOP. http://labs.jboss.com/jbossaop/, 2007,

[19.8.2007].

KHH01 Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. ja Griswold, W. G., An overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming*, Budabest, Hungary, kesäkuu 2001, sivut 327-353.

Kir05 Kirsten, M, Aop tools comparison, part 1: Language mechanisms. http://www.ibm.com/developerworks/library/j-aopwork1/, 2005, [21.8.2007].

Kiz04 Kiczales, G., It's Not Metaprogramming. Software Development Magazine, 12, 11 (2004), sivut 60-61.

KiM05 Kiczales, G. ja Mezini, M., Aspect-oriented programming and modular reasoning. *Proceedings of the 27th international Conference on Software Engineering*, St. Louis, MO, USA, toukokuu 2005, sivut 49-58.

KLM97 Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. ja Irwin, J., Aspect-oriented programming. *Proceedings of the 11th European Conference on Object-Oriented Programming*, Jyväskylä, Finland, kesäkuu 1997, sivut 220– 242.

KMB06b Kellens, A., Mens, K., Brichau, J. ja Gybels, K., Managing the evolution of aspect-oriented software with model-based pointcuts. *Proceeding of the 20th European Conference on Object-Oriented Programming*, Nantes, France, heinäkuu 2006, sivut 501-525.

KnR05 Kniesel, G., ja Rho, T., Generic aspect languages - needs, options and challenges. *Proceedings of the 2ème Journée Francophone sur le Développement de Logiciels Par Aspects*, Lille, France, syyskuu 2005.

KoS04 Koppen, C., ja Stoerzer, M., PCDiff: Attacking the fragile pointcut problem. *Proceedings of the 1st European Interactive Workshop on Aspects in Software,* Berlin, Germany, syyskuu 2004.

Lad03 Laddad, R., AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications Co, 2003.

Lam99 Lamping, J., The role of the base in aspect-oriented programming. *Proceedings of the Workshop on Object-Oriented Technology*, Lisbon, Portugal, kesäkuu 1999, sivut 289-291.

Les06 Lesiecki, N., Applying AspectJ to J2EE application development.

IEEE Software, 23, 1 (2006), sivut 24-32.

LiO97 Lieberherr, K. J. ja Orleans, D., Preventive program maintenance in Demeter/Java. *Proceedings of the 19th international Conference on Software Engineering,* Boston, Massachusetts, United States, maaliskuu 1997, sivut 604-605.

Log07 LogicAJ, LogicAJ (Logic Aspects for Java), http://roots.iai.uni-bonn.de/research/logicaj/, 2007, [21.8.2007].

LoH95 Lopes, C. V. ja Hursch, W. L., Separation of concerns. Northeastern University, College of Computer Science Technical Report NU-CCS-95-03, helmikuu 1995.

MeR03 Mehner, K. ja Rashid, A., Towards a generic model for AOP (GEMA). Technical Report CSEG/1/03, Computing Department, Lancaster University, UK, tammikuu 2003.

MeW99 Mehner, K. ja Wagner, A., An assessment of aspect language design. In Position Paper Young Researcher Workshop, GCSE '99., 1999.

OAT06 Ongkingco, N., Avgustinov, P., Tibble, J., Hendren, L., de Moor, O. ja Sittampalam, G., Adding Open Modules to AspectJ. *Proceedings of the 5th international Conference on Aspect-Oriented Software Development*, Bonn, Germany, maaliskuu 2006, sivut 39-50.

OHT00 Ossher, H., Harrison, W. ja Tarr, P., Software engineering tools and environments: a roadmap. *Proceedings of the Conference on the Future of Software Engineering*, Limerick, Ireland, kesäkuu 2000, sivut 261-277.

OMB05 Ostermann, K., Mezini, M. ja Bockisch, C., Expressive pointcuts for increased modularity. *Proceedings of the 19th European Conference on Object-Oriented Programming*, Glasgow, UK, heinäkuu 2005, sivut 214--240.

OsT01 Ossher, H. ja Tarr, P., Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, 44, 10 (2001), sivut 43-50.

Par72 Parnas, D. L., On the criteria to be used in decomposing systems into modules. *Communications of the ACM,* 15, 12 (1972), sivut 1053-1058.

PRS05 Pawlak R., Retaillé J.-P ja Seinturier, L., Foundations of AOP for J2EE Development, APress, 2005.

RaW02 Rajlich, V. ja Wilde, N., The role of concepts in program comprehension. *Proceedings of the 10th International Workshop on Program Comprehension*, Paris, France, kesäkuu 2002, sivut 285--288.

RhK04 Rho, T. ja Kniesel, G., Uniform genericity for aspect languages. Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn, joulukuu 2004.

ShP05 Shaker, P. ja Peters, D. K., An introduction to aspect-oriented software development. *Proceedings of the Newfoundland Electrical and Computer Engineering Conference*, IEEE Newfoundland and Labrador Section, St. John's, Canada, marraskuu 2005.

Spr07 Spring Framework, Aspect Oriented Programming with Spring. http://www.springframework.org/docs/reference/aop.html, 2007, [19.8.2007].

TaO01 Tarr, P. ja Ossher, H., Hyper/J: Multi-Dimensional Separation of Concerns for Java. *Proceedings of the 23rd International Conference on Software Engineering*, Toronto, Ontario, Canada, toukokuu 2001, sivut 729-730.

TAP98a The AspectJ Project, The AspectJ™ programming guide. http://www.eclipse.org/aspectj/doc/released/progguide/index.html, 1998, [18.8.2007].

TAP98b The AspectJ Project, The AspectJ development environment guide. http://www.eclipse.org/aspectj/doc/released/devguide/index.html, 1998, [18.8.2007].

TAP04 The AspectJ Project, The AspectJ™ 5 Development Kit Developer's Notebook. http://www.eclipse.org/aspectj/doc/next/adk15notebook/index.html, 2004, [18.8.2007].

TBG03 Tourwé, T., Brichau, J. ja Gybels, K., On the existence of the AOSD-evolution paradox. *Proceedings of the Aspect Oriented Software Desing 2003 Workshop on Software Engineering Properties of Languages for Aspect Technologies*, Boston, USA, maaliskuu 2003.

TeA00 Tekinerdogan, B. ja Aksit, M., Separation and composition of concerns through synthesis-based design, *ACM OOPSLA'2000 workshop on Advanced Separation of Concerns*, Minneapolis, USA,

Iokakuu 2000.

TOH99 Tarr, P., Osher, H., Harrison, W. ja Sutton, S. M. Jr., N Degrees of separation: multi-dimensional separation of concerns, *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, USA, toukokuu 1999, sivut 107-119.

YBK04 Yo, Y., Bachand, A., Kienzle, J., Comparing different AOP approaches, Technical Report SOCS-TR-2004.7, marraskuu 2004.