ASPEKTIYMPÄRISTÖT ASPECTJ JA SPRING AOP

Miia Sorsa

10.6.2008

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

Tiivistelmä

Aspektiohjelmointi on melko uusi ohjelmointiparadigma jonka tarkoituksena on parantaa olio-ohjelmoinnin paradigman puutteita poikkileikkaavien ominaisuuksien modularisoimisessa. Se pyrkii muun muassa parantamaan järjestelmien modulaarisuutta ja uudelleenkäytettävyyttä tarjoamalla uudenlaisia ohjelmistokomponentteja. Paradigman perusajatukset syntyivät rinnakkain AspectJ – ohjelmointikielen kehitystyön kanssa. Myöhemmin on syntynyt myös useita erilaisia aspektiparadigman toteutuksia, kuten Spring Frameworkin AOP-moduuli. Tutkielmassa selvitetään kirjallisuuden ja testattujen esimerkkien avulla AspectJ:n ja Spring AOP:n tapoja toteuttaa aspektiohjelmoinnin perusperiaatteita.

ACM luokat (ACM Computing Classification System, version 1998): D.3.2, D.2.6, D.2.10, D.1.m

Avainsanat: aspektiohjelmointi, AspectJ, Spring AOP

Sisällysluettelo

1. JOHDANTO 1

2. TAUSTAA 3

2.1 Olioparadigma 3

2.2 Aspektiparadigma 5

2.3 Vaatimusten jäsentäminen 7

3. ASPEKTIOHJELMOINTI 11

3.1 Aspektiohjelmoinnin vaiheet 12

3.2 Aspektiohjelmoinnin määritelmä 14

3.3 Keskeiset käsitteet 16

3.3.1 Liitoskohta 16

3.3.1.1 Liitoskohtamalli 17

3.3.2 Liitoskohtamäärittely 18

3.3.3 Kehote 19

3.3.4 Aspekti 19

3.3.5 Tyyppien väliset deklaraatiot 20

3.3.6 Staattinen ja dynaaminen punominen 21

3.4 Suhde olio-ohjelmointiin 23

3.5 Aspektiohjelmointikielet 24

4. ASPECTJ 26

4.1 Liitoskohta 28

4.2 Liitoskohtamäärittely 29

4.2.1 Liitoskohtamäärittelyn nimeäjä 30

4.3 Kehote 37

4.3.1 Liitoskohdan introspektio 42

4.4 Tyyppien väliset deklaraatiot 42

4.5 Aspekti 44

4.6 Esimerkkiaspekti 46

4.7 Poikkileikkaaminen annotaatioiden avulla 48

5. SPRING AOP 50

5.1 Miten Spring AOP toteuttaa AOP:n keskeisiä käsitteitä **52**
5.2 Skeemapohjainen AOP **54**
5.2.1 AOP- skeema 54
5.2.2 Papumäärittelyt 56
5.2.3 Aspekti 57
5.2.4 Liitoskohtamääritys 58
5.2.5 Kehote 59
5.2.6 Tyyppien väliset deklaraatiot 63
5.2.7 Esimerkkiaspekti 64
5.3 Poikkileikkaaminen annotaatioiden avulla **66**
6. YHTEENVETO **68**
VIITELUETTELO **70**
LIITE 1. ESIMERKEISSÄ POIKKILEIKATTAVA JAVA-OHJELMA **76**
LIITE 2. ASPECTJ –ESIMERKKIEN LÄHDEKOODIT **79**
LIITE 3. SPRING AOP –ESIMERKKIEN LÄHDEKOODIT **84**

1. JOHDANTO

Tietojenkäsittelytiede on käynyt olemassaolonsa aikana läpi evoluution muun muassa ohjelmointikielien sekä tietokonejärjestelmien saralla. Ohjelmointikieli voidaan määritellä keinotekoiseksi kieleksi, jota on mahdollista käyttää koneen toiminnan kontrolloimiseen. Kielet määritellään syntaksiin ja semantiikkaan liittyvien sääntöjen avulla ja niiden tavoitteena on, paitsi toteuttaa algoritmeja, myös kuvata sitä, miten informaatiota tulisi organisoida ja muokata. Ensimmäisistä konekielisistä ohjelmointikielistä on siirrytty muun muassa proseduraalisen, rakenteisen, funktionaalisen ja logiikkaohjelmoinnin kautta yhä korkeamman tason ohjelmointikieliin, joiden kautta on ollut mahdollista mallintaa maailmaa ohjelmoijalle yhä luonnollisemmalla tavalla.

Yleinen trendi ohjelmointikielien kehityksessä on ollut tarve ratkoa ohjelmointiongelmia yhä korkeammalla abstraktiotasolla. Ensimmäiset ohjelmointikielet olivat erittäin sidottuja tietokoneen fyysiseen kokoonpanoon. Uusien ohjelmointikielien myötä niihin on lisätty ominaisuuksia, joiden avulla ohjelmoijan on mahdollista esittää käsitteitä ja käskyjä yhä korkeammalla tasolla. Tämän jatkuvan kehityksen kautta yhä monimutkaisempien ohjelmointiongelmien tehokas ratkaiseminen ja toteuttaminen on mahdollista. Samoin yhä laajempien ja monimutkaisempien ohjelmistojen rakentaminen on yksinkertaisempaa ja tehokkaampaa.

Yksi uusimmista virtauksista ohjelmointikielien saralla on aspektiohjelmointi. Sen tavoitteena on parantaa ohjelmien modulaarisuutta ja uudelleenkäytettävyyttä tarjoamalla mekanismeja poikkileikkaavien ominaisuuksien modularisoimiseksi. Tässä tutkielmassa tarkastellaan aspektiohjelmoinnin perusperiaatteita ja -käsitteitä sekä kahden aspektiohjelmointiympäristön, AspectJ:n ja Spring AOP:n perusmekanismeja aspektiohjelmoinnin peruskäsiteiden pohjalta esimerkkien kautta. Luvussa 2 tarkastellaan ensin niitä ohjelmointikielien ja ohjelmistotuotannon taustoja, jotka ovat johtaneet aspektiparadigman syntymiseen. Luvussa 3 tarkastellaan aspektiparadigman keskeisiä ajatuksia ja käsitteitä. Luvussa 4 ja 5 tutustutaan kahden eri aspektiohjelmointiympäristön, AspectJ:n ja Spring AOP:n perusmeka-

nismeihin sekä vertaillaan niiden eroavaisuuksia. Vertailussa tullaan käyttämään lähdekirjallisuutta, lyhyitä eri aspektiohjelmoinnin peruskäsitteitä toteuttavien ympäristöjen mekanismien esimerkkejä sekä useamman toiminnallisuuden toteuttavaa aspektiohjelmaa molemmilla aspektiohjelmointiympäristöillä toteutettuna. Esimerkit sekä molemmat kokonaiset aspektiohjelmat ovat testattuja ja niiden lähdekoodit löytyvät liitteistä 1, 2 ja 3. Luvussa 6 on yhteenveto sekä pohdintaa käsitellyistä aiheista.

2. TAUSTAA

Nykyaikana tyypillisissä nopeasti muuttuvissa ympäristöissä mukautumiskyvystä on tullut tärkeä ominaisuus tietokonejärjestelmille, ohjelmointikielille sekä ohjelmistotuotannon menetelmille. Kasvavat vaatimukset ovatkin johtaneet erilaisten tekniikoiden, metodologioiden ja lähestymistapojen kehitystyöhön. Niiden tarkoituksena on tarjota parempia työkaluja ohjelmistojen tuottamiseen, jotta paremmin mukautettavissa olevien ohjelmistojen luominen olisi paitsi mahdollista, myös yksinkertaista.

2.1 Olioparadigma

Tällä hetkellä valta-asemassa ohjelmoinnin ja ohjelmistotuotannon saralla on *olio-ohjelmoinnin paradigma (Object-Oriented Paradigm, OOP).* Koskimies (1998, s.2) esittää olio-ohjelmointia käsittelevässä kirjassaan, että olioperustaisuus vastaa hyvin ihmisen tapaa hahmottaa maailmaa. Järjestelmän vaatimukset jaetaan objekteihin, jotka toteutetaan ohjelmointikielellä. Nämä objektit abstraktoivat yhdessä tiedon ja datan yhteen käsitteelliseen ja fyysiseen entiteettiin, olioon. Paradigman on tehnyt erityisen suosituksi tarve systematisoida ohjelmistojen rakentamista, lisätä niiden uudelleenkäytettävyyttä ja helpottaa niiden ylläpitoa. Sen ympärille on syntynyt rikas ohjelmointikulttuuri, joka kielineen, ohjelmointiympäristöineen, kirjastoineen, analyysi- ja suunnittelumenetelmineen, suunnittelumalleineen, sovelluskehyksineen ja CASE-välineineen tukee käytännön ohjelmistokehitystä monipuolisemmin kuin mikään muu paradigma.

Khatchadourianin (2006, s.3-4) mukaan olio-ohjelmoinnissa on kuitenkin puutteensa. Sen perinteinen tyyli on toteuttaa monimutkaisia järjestelmiä jakamalla järjestelmän funktionaalisia vaatimuksia pienemmiksi, vähemmän monimutkaisiksi, erikoistuneiksi yksiköiksi, jotka toteutetaan kukin erillään, kun yksikkö on tarpeeksi yksinkertainen. Nämä yksiköt kootaan toteutuksen jälkeen takaisin yhteen, jolloin syntyy kokonainen järjestelmä. Tuloksena on puurakenne, jonka juurena on alkuperäinen, monimutkainen ongelma. Puun solmut lähtevät puun juuresta alifunktioina, jotka auttavat alkuperäisen ongelman ratkaisemisessa.

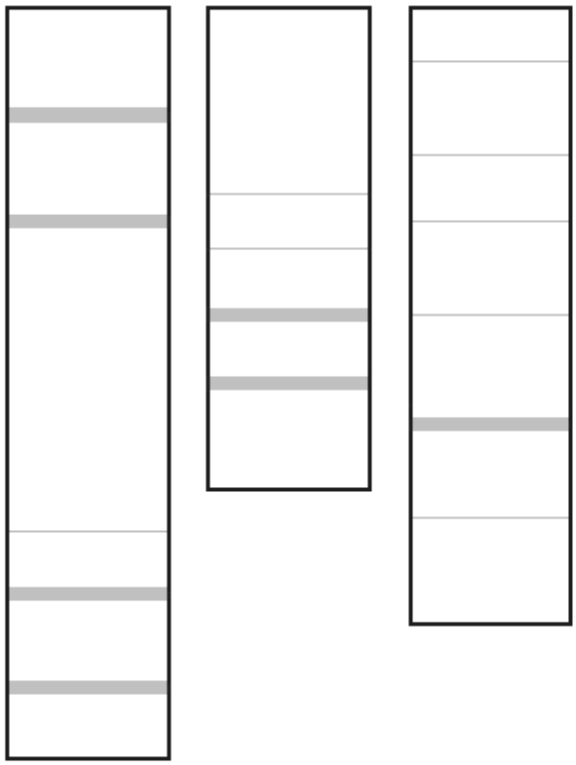
Ratkaisu toimii hyvin perinteisissä ohjelmointikielissä, jolloin ohjelmiston suunnittelumalli voidaan jäljittää suoraan implementaatioon. Ensimmäiset ongelmat nousevat kuitenkin esiin, kun yksittäinen toiminnallisuus hajoaa eri puolille puun solmuja. Jos puun solmut kuvaavat eri osia järjestelmän toiminnasta ja tietty toiminnallisuus, kuten autentikointi tai lokin kirjoittaminen koskettaa usean solmun toimintaa, ei ole mahdollista toteuttaa toiminnallisuutta yhdessä solmussa, kuten paradigman periaatteet edellyttävät.

Viega ja Voas (2000, s.19) esittelevät hieman käytännönläheisemmän esimerkin olioparadigman puutteellisista abstraktiomenetelmistä. Perinteinen Javan virheidenkäsittelytapa sisältää monia ongelmia. Tyypillisin ongelma on se, että virheidenkäsittely ei ole modulaarista, vaan ohjelmoijat käsittelevät usein poikkeustilanteita improvisoidusti ja ennalta valmistelemattomasti aina kun tarve siihen tulee vastaan. Virheidenkäsittelyn ohjelmakoodia uudelleenkäytetään harvoin, samoin kuin virheidenkäsittelijöissä käytetään vähän mitään abstraktointimenetelmiä. Virheidenkäsittelyn suorittaminen geneerisesti johtaa kuitenkin helposti epäluotettavaan ja tehottomaan koodiin. Ohjelmoijat tyypillisesti käsittelevät ennalta arvaamattomia virhetilanteita catch-all -käsittelijöiden avulla. Tässä menetelmässä puutteena on kuitenkin se, että nousseesta poikkeustilanteesta tyylikäs palautuminen on yleensä mahdotonta. Tyypillisesti ratkaisuna onkin vain kirjata poikkeus lokiin ja sulkea ohjelma. Ideaalitilanteessa olisi parempi yhdistää geneeriset virhetilanteet erikoistuneeseen toimintoon niin, että kukin toiminto on ohjelmoitu tilanteen mukaan kullekin poikkeuksen tyypille. Jotta tämän tekeminen olisi mahdollista Javan perinteisen virheidenkäsittelyn menetelmillä, joutuu ohjelmoija muuttamaan käsin joka ikistä virheidenkäsittelytilannetta. Tämä johtaa kuitenkin suureen määrään samanlaista koodia.

Esiteltyyn Javan virheidenkäsittelyn ongelmaan liittyy Viegan ja Voasin mukaan myös toinen puute, ohjelmakoodin ylläpito. Esimerkiksi esiin nousseiden poikkeusten kirjaaminen lokiin ei välttämättä ole itsestäänselvyys ohjelman ylläpitäjälle, jolloin tämä saattaa lisätä uusia virheidenkäsittelijöitä, mutta jättää lokin kirjaamisen toiminnallisuuden niistä kokonaan pois. Tällöin järjestelmän lähdekoodin kompleksisuus kasvaa ja ylläpidettävyys huononee entisestään.

2.2 Aspektiparadigma

Tyypillisiä oireita puutteelliselle ohjelmointimetodologialle ovat *ohjelmakoodin sekoittuminen (code tangling)* sekä *ohjelmakoodin pirstaloituminen (code scattering)* (Laddad, 2002a). Ohjelmakoodin sekoittumisella tarkoitetaan tilannetta, jossa ohjelman moduuli on sidoksissa useampaan kuin yhteen vaatimukseen. Ohjelmakoodin sekoittuminen johtaa ohjelmakoodin pirstaloitumiseen. Yhden moduulin toiminnallisuuksien ollessa yhteydessä useampaan kuin yhteen vaatimukseen pirstaloituu vaatimuksen toteuttava ohjelmakoodi pitkin poikin muita vaatimuksia toteuttavaa ohjelmakoodia. Kuvassa 2.1 on esitetty toiminnon pirstaloituminen eri puolille ohjelmakoodia, eikä yhteen modulaariseen yksikköön kuten olioparadigman periaatteet edellyttävät.



Kuva 2.1. Ohjelmakoodin pirstaloituminen (Pawlak & al., 2005, s.9)

Kuva 2.1 esittää hyvin esimerkiksi aikaisemmin esitettyä Javan virheidenkäsittelyn ongelmaa. Valkoiset laatikot kuvassa ovat yksittäisten toiminnallisuuksien toteutuksia eli modu-

laarisia yksiköitä, olioita. Harmaat palkit valkoisten laatikoiden sisällä kuvaavat vuorostaan virheidenkäsittelyn suorittavaa ohjelmakoodia. Kuten kuvasta näkyy, se on pirstaloitunut eri puolille muita ohjelmamoduuleita. Olioparadigman periaatteilla virheidenkäsittelyn siirtäminen omaan modulaariseen yksikköönsä on, ellei mahdotonta, ainakin erittäin hankalaa.

Koodin sekoittuminen ja pirstaloituminen vaikuttavat ohjelmistoihin useilla eri tavoilla. Laddad (2002a) esittää, että näitä ovat muun muassa huono jäljitettävyys vaatimuksista ohjelmakoodiin, huonontunut tuottavuus, uudelleenkäytettävyysmahdollisuuksien väheneminen, ohjelmakoodin laadun huononeminen sekä ohjelman jatkokehityksen vaikeutuminen.

Aspektiparadigmassa näitä yllä kuvattuja pirstaloituvia ja sekoittuvia toimintoja kutsutaan *poikkileikkaaviksi ominaisuuksiksi (crosscutting concerns).* Lafferty ja Cahill (2003) määrittelevät, että poikkileikkaavat ominaisuudet ovat ominaisuuksia tai mielenkiinnon kohteita, jotka normaalisti rikkovat olio-ohjelmoinnin mallinnustapaa, koska toimintojen sijoittuminen niiden tukemiseksi eli ole linjassa oliomallin kompositiomenetelmien kanssa. Jopa konseptuaalisesti yksinkertaiset poikkileikkaavat ominaisuudet, kuten virheiden jäljitys tai synkronisointi voi johtaa sekoittumiseen, kun poikkileikkaavaan ominaisuuteen liittyvät ohjelmakoodin pätkät lomittuvat muihin ominaisuuksiin liittyvään ohjelmakoodiin.

Toinen tyypillinen ongelma on *arkkitehdin dilemma (architect's dilemma)* (Khatchadourian, 2006). Usein on vaikea arvioida sitä missä vaiheessa ohjelman suunnittelu riittää sekä tulisiko, ja missä määrin, järjestelmän ehkä tulevaisuudessa muuttuviin ja lisättäviin vaatimuksiin varautua. Arkkitehdin dilemma on yleinen varsinkin suurten ohjelmistojen kehityksessä. Laddad (2002a) esittää, että ihanneohjelmassa mukautuminen muutoksiin ja täydentävien toiminnallisuuksien lisääminen jälkikäteen on helppoa. Varautuminen liian vähäisessä määrin tulevaisuuden mahdollisesti muuttuviin vaatimuksiin voi aiheuttaa jatkokehityksessä valtavasti ylimääräistä työtä; monia ohjelman osia saatetaan joutua muuttamaan tai toteuttamaan kokonaan uudestaan. Tämän vastakohtana on myös ylisuunnittelu. Varautumalla liiallisessa määrin vähemmän todennäköisiin vaatimuksiin on lopputuloksena usein sekava ja ”turvonnut” lähdekoodi. Tarve paremmalle tuelle järjestelmän muokkaamiselle jälkikäteen onkin yhä suurempi.

Post-object programming (POP) –menetelmät, jotka pyrkivät parantamaan olio-ohjelmoinnin ilmaisukykyä, ovat tuottelias kenttä ajankohtaiselle tutkimukselle (Elrad & al., 2001, s.30). POP-teknologioita ovat esimerkiksi toimialakohtaiset (domain) kielet, geneerinen ja generatiivinen ohjelmointi, rajoitekielet, reflektio ja metaohjelmointi sekä asynkroninen viestinvälitys. Yksi tärkeä POP-teknologia on myös *aspektiohjelmointi (Aspect-Oriented Programming, AOP)*

Clarken ja Baniassadin (2005, s.3) mukaan AOP:n tarkoituksena on vapauttaa ohjelmistokehittäjät *hallitsevan dekomponoinnin hegemoniasta eli määräävästä asemasta*. Aksit (2004, s.74) esittää, että suurin osa mallinnustekniikoista suosii tiettyä dekomponointiskeemaa. Tämä voi aiheuttaa kaksi erilaista ongelmaa: ensinnäkin, ennalta määrätyn dekomponointiskeeman vuoksi tietyt laatuun liittyvät tekijät saattavat saada liikaa painoarvoa. Toiseksi, muuttuvan bisneskontekstin takia voi olla vaikeaa (uudelleen)organisoida dekomponointi niin, että uudet vaatimukset olisi mahdollista saavuttaa. Clarke ja Baniassad (2005, s.3) esittävät, että esimerkiksi olio-ohjelmoinnissa hallitseva dekomponointi on luokkien ja metodien modulaarisuus. Hegemonia vuorostaan viittaa siihen, miten sidottuna olio-ohjelmointiin ohjelmistokehittäjät pakotetaan tekemään suunnitteluratkaisuja, jotka johtavat pirstoutumiseen ja sekoittumiseen.

2.3 Vaatimusten jäsentäminen

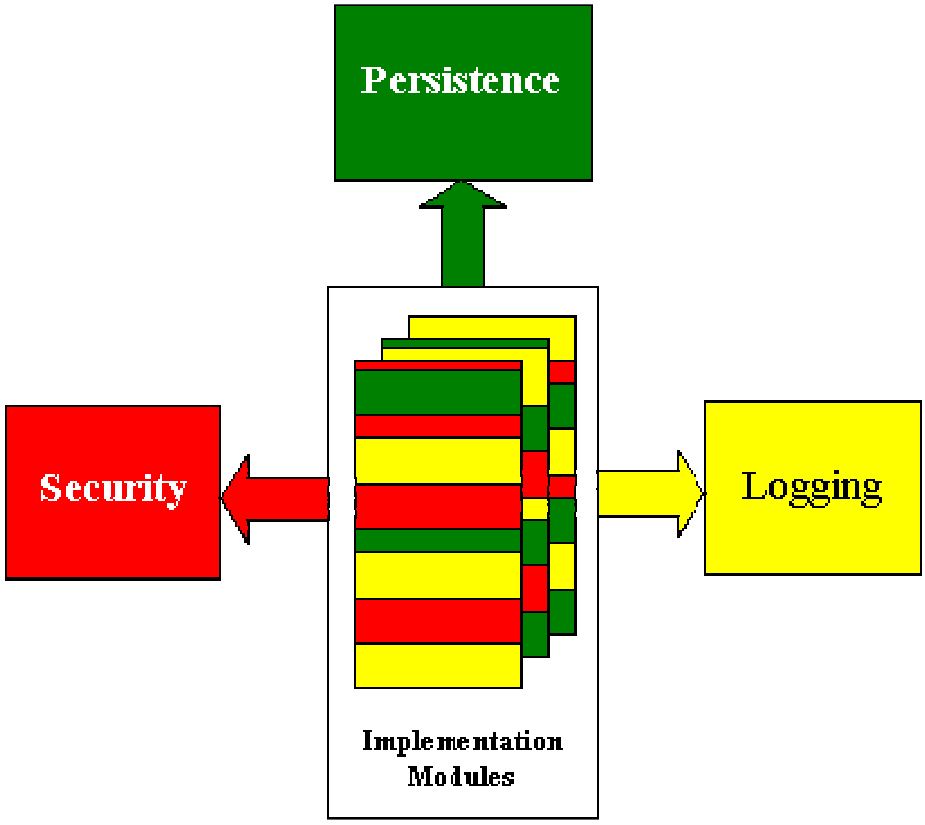
Ohjelmistojen suunnittelu ja ohjelmointityö ovat hyvin monimutkaisia kognitiivisia tehtäviä. Niiden suorittamiseen vaaditaan useaa erityyppistä tietoa, tiedonlähteitä ja monenlaisia ongelmanratkaisutaitoja. Monimutkaisuuden on osoitettu olevan olennainen osa suunnittelutyötä, osaksi puutteellisen ongelman muodostamisen vuoksi, osaksi siksi, että yksilön kognitiiviset resurssit eivät riitä kaikkien yksittäisen ongelman rajoitteiden yhtäaikaiseen käsittelemiseen.

Mili & al. (2004, s.1) määrittelevät, että *vaatimusten jäsentäminen (separation of concerns, SOC)* on yleinen ongelmanratkaisuidiomi, jonka avulla voimme pilkkoa monimutkaisen ongelman löyhästi yhteen kytketyiksi, helpommin ratkaistavissa oleviksi aliongelmiksi.

Idiomin taustalla on ajatus, että näiden aliongelmien ratkaisut voidaan yhdistää kohtuullisen helposti, jolloin saadaan ratkaisu alkuperäiseen, monimutkaiseen ongelmaan. Ossherin ja Tarrin (2000, s.2) mukaan SOC on osa ohjelmistokehityksen ydintä. Yleisimmässä muodossaan se viittaa kykyyn tunnistaa, kapseloida ja manipuloida vain niitä osia järjestelmästä, jotka ovat merkityksellisiä tietylle konseptille, päämäärälle tai tarkoitukselle.

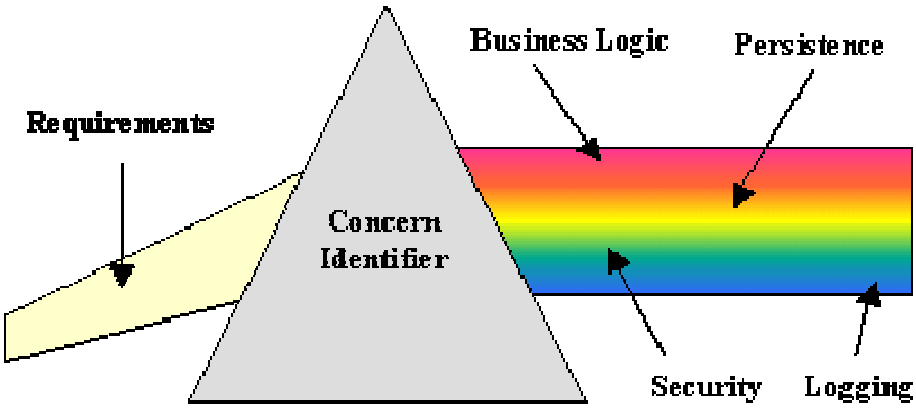
SOC-termin esitteli luultavasti ensimmäisen kerran Edger W. Dijkstra 1970-luvun alkupuoliskolla artikkelissaan ”On the role of scientific thought”. Sutton Jr:n ja Rouvellonin (2004, s.2) mukaan termi on myöhemmin saanut laajaa huomiota moderneissa ohjelmointikielissä moduulien, pakettien, luokkien ja rajapintojen sekä niitä tukevien abstraktio-, kapselointi ja tiedonpiilottamismenetelmien muodossa. Mili & al (2004, s.1) esittävätkin, että ohjelmointikielien historia voidaan nähdä jatkuvana modularisaation rajojen etsintänä, joista on yksinkertaisinta kartoittaa tie takaisin alkuperäisten vaatimusten luonnollisiin rajoihin. Olio-ohjelmoinnin ominaisuuksia kuvaavien olioiden erottelun lisäksi esimerkiksi proseduraaliset ohjelmointikielet erottelevat erilaiset toiminnallisuudet proseduureihin. MVC-malli vuorostaan erottelee sisällön esityksestä ja tiedon käsittelyn sisällöstä.

Laddadin (2002a) mukaan tyypillisesti järjestelmät koostuvat erilaisista vaatimuksista, kuten liiketoimintalogiikka, suorituskyky, tiedon pysyvyys, lokin kirjoitus, virheiden etsintä, autentikointi, turvallisuus, monisäieturva, virhetarkastus ja niin edelleen. Ohjelmistojen vaatimuksiin liittyy tämän lisäksi myös sovelluskehitysprosessiin liittyviä seikkoja, kuten ymmärrettävyys, ylläpidettävyys, jäljitettävyys sekä järjestelmän evoluution helppous.



Kuva 2.2. Ohjelmiston modulaarisuus (Laddad, 2002a)

Vaatimusten jäsentämisen periaatteiden mukaisesti ohjelmistot voidaan nähdä kokonaisuutena, joka muodostuu erilaisten vaatimusten implementaatioista. Kuvassa 2.2 esitetään järjestelmä erilaisten ominaisuuksien yhdistelmänä, jotka on implementoitu erillisissä moduuleissa. Kuvassa 2.3 vuorostaan esitetään vaatimusten jäsentämisen prisma-analogia. Analogiaa on käytetty esimerkkinä siitä, miten erilaiset vaatimukset voidaan erottaa toisistaan AOP:n avulla samaan tapaan kuin prisma erottaa valonsäteen värien kirjoksi.



Kuva 2.3. Prisma-analogia (Black & Harman, 2006, s.1)

Kuvassa vaatimus-valonsäde (requirements) kulkeutuu vaatimusten erottelijaprisman (concern identifier) läpi, jolloin erityyppiset vaatimukset erotellaan toisistaan (liiketoimintalogiikka, tiedon pysyvyys, turvallisuus, lokin kirjoitus) (Laddad, 2002a). Black ja Harman (2006, s.3) esittävät, että prisma-analogia voidaan pitää aspektiparadigman teoreettisena lähtökohtana. Siinä missä prisman erottelemat värit ovat koherenssissa toistensa kanssa, ei yhtäkään niistä ole mahdollista johtaa toisista. Samoin toimivat aspektit aspektiparadigmassa: esimerkiksi turvallisuusaspekti (security) ja lokiaspekti (logging) ovat koherenttejä toistensa kanssa, mutta niitä ei ole mahdollista johtaa toinen toisistaan.

Tarr ja Ossher (2001, s.778) esittävät, että *edistyneen vaatimusten jäsentämisen (advanced separation of concerns, ASOC)* tarkoituksena on tarttua ongelmiin, jotka nousevat esiin yhä monimutkaisempien ja laajempien ohjelmistojen tuotannossa. Tyypillisimmät ongelmat liittyvät vaikeuksiin ymmärtää lisääntyvää monimutkaisuutta, ohjelmakoodin huonoon luettavuuteen ja uudelleenkäytettävyyteen sekä ohjelman evoluution vaatimaan resurssimäärään. He esittävätkin, että tilanteeseen sopiva vaatimusten jäsentäminen voi vähentää järjestelmän kompleksisuutta ja parantaa ymmärrettävyyttä, edistää jäljitettävyyttä sekä järjestelmän osien sisällä että läpi koko järjestelmän, helpottaa uudelleenkäytettävyyttä, ulkopuolelta tapahtuvaa adaptaatiota ja kustomointia, sekä evoluutiota sekä yksinkertaistaa komponenttien integraatiota.

3. ASPEKTIOHJELMOINTI

Elradin et al. (2001) mukaan aspektiohjelmoinnin paradigma perustuu ajatukseen, että tietokonejärjestelmien tuottaminen on helpompaa, kun järjestelmän erilaiset mielenkiinnon kohteet tai *vaatimukset (concerns)* sekä kuvaukset niiden välisistä suhteista määritellään erillään, jonka jälkeen annetaan taustalla olevan AOP-ympäristön yhdistää niistä yhtenäinen ohjelma. Vaatimukset ovat yksittäisiä järjestelmän toiminnallisuuksia ja ne voivat vaihdella korkean tason käsitteistä, kuten turvallisuus tai palvelun laatu matalamman tason käsitteisiin, kuten välimuistin käyttö tai puskurointi. Ne voivat olla funktionaalisia, kuten järjestelmän käyttöliittymän ominaisuus tai liiketoimintasäännöt tai ei-toiminnallisia, kuten synkronointi tai transaktioiden hallinta. Aspektiohjelmoinnin tavoitteena ei siis olekaan ratkaista uusia ongelmia, vaan luoda uusia tapoja ongelmien ratkaisemiseksi helpommin ja tehokkaammin vähentämällä ongelmien kompleksisuutta.

Laddad (2003, s.90) esittää, että aspektiohjelmointi on ohjelmointikielien evoluution askel, joka parantaa implementaation ymmärrettävyyttä ja yksinkertaistaa uusien vaatimusten lisäämistä sekä vanhojen muuttamista. Sen systemaattinen lähestymistapa tarjoaa vaatimusten ja suunnittelutavoitteiden suoran jäljittämisen implementaatioon.

Aspektiparadigmalle löytyy myös kriitikoita. Muun muassa Steinmann (2006) huomauttaa, että aspektiohjelmoinnin periaate on paradoksi. Tämän mukaan aspektiohjelmoinnin tavoite ei voi olla ohjelmistojen modulaarisuuden parantaminen, koska jo paradigman perusperiaatteiden tarkoituksena on rikkoa järjestelmän modulaarisuus. Steinmann esittääkin, että aspektiohjelmoinnin tulisi luopua tavoitteestaan ”modularisoida ei-modularisoitava” ja sen sijaan keskittyä sulauttamaan pääkonseptinsa ja hillitön toimintatapansa muihin ohjelmointimalleihin, joissa modulaarisuus ja rakenteisuus eivät ole ongelma.

Aspektiparadigma onkin vielä melko nuori, joten siitä löytyy paljon puutteita. Esimerkiksi tutkittaessa aspektiohjelmointia ohjelmistotuotantokäytössä, havaittiin siinä uudenlaisia ongelmia, jotka liittyvät ohjelmistojen kompleksisuuteen, oikeellisuuteen ja testattavuuteen (Bradley, 2003, s.38). Ison osa havaituista ongelmista todettiin johtuneen teknisistä ongel-

mista, jotka liittyivät aspektiohjelmointikielten implementaatioon. Aspektiparadigma kärsii tällä hetkellä eniten riittämättömistä ohjelmistokehityksen menetelmistä ja apuvälineistä. Sille ei ole kehitetty esimerkiksi yhtenäistä mallinnuskieltä, vaikkakin useita erilaisia on esitetty. Näitä ovat muun muassa koko ohjelmistokehitysprosessin määrittely ja suunnitteluvaiheet kattava Theme (Clarke & Baniassad, 2005), vaatimusten mallintamiseen keskittyvä Cosmos (Sutton Jr. & Rouvellon, 2002) sekä Suzukin ja Yamamoton ohjelmistojen suunnitteluvaiheeseen keskittyvä malli (1999).

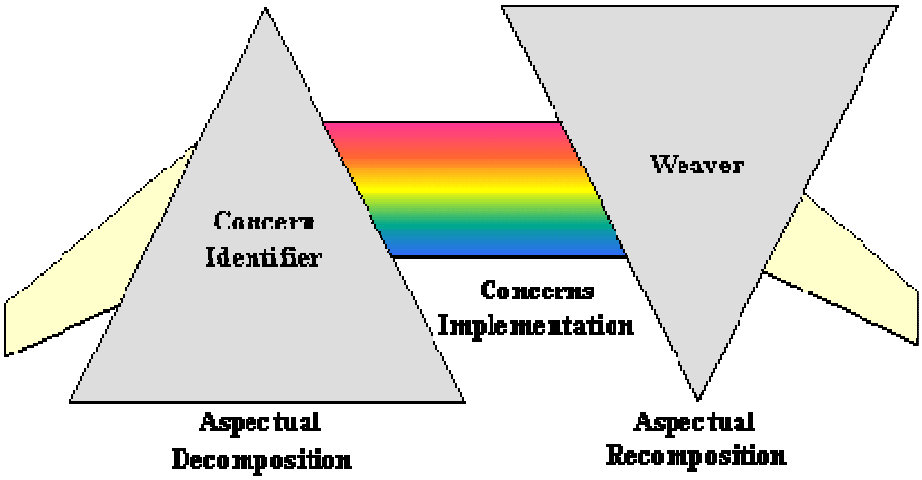
3.1 Aspektiohjelmoinnin vaiheet

AOP pyrkii erittelemään ominaisuuksia kategorisoimalla ne kahteen erilliseen tyyppiin, *ydinominaisuuksiin (core concerns)* sekä poikkileikkaaviin ominaisuuksiin (Khatchadourian, 2006, s.3-4). Ydinominaisuudet ovat järjestelmän toimintaan liittyviä ominaisuuksia, jotka ovat ohjelmamoduulin keskeisellä toiminta- tai vastuualueella. Niitä kutsutaan usein myös *liiketoiminnallisiksi ominaisuuksiksi (business concerns*) tai *komponenteiksi (components)*. Toiseen kategoriaan kuuluvat poikkileikkaavat ominaisuudet, jotka ovat järjestelmän toiminnallisuuksia, jotka liittyvät useisiin komponentteihin. Ne ovat toissijaisia ominaisuuksia, joiden toiminta-alue voi koskettaa useita ydinominaisuuksia. Tyypillisiä poikkileikkaavia ominaisuuksia ovat virheidenkäsittely, lokin kirjoittaminen tai autentikointi. Sato (2005, s.2) esittääkin, että tämän perinteisestä poikkeavan aspektiohjelmoinnin paradigman ominaisuuksien erottelumenetelmän takia paradigma toteuttaa periaatteissaan edistynyttä vaatimusten jäsentämistä tai *poikkileikkaavien ominaisuuksien jäsentämistä (separation of crosscutting concerns)*.

Jacobson ja Ng (2005, s.3) toteavat, että kyvyttömyys eritellä poikkileikkaavat ominaisuudet suunnittelu- ja implementaatiovaiheessa johtaa todennäköisesti ohjelmaan, jonka ymmärtäminen ja ylläpitäminen on vaikeaa. Se estää rinnakkaista ohjelmiston kehitystyötä, tekee ohjelman laajentamisesta vaikeampaa sekä aiheuttaa monia niistä muista tyypillisistä ongelmista, jotka vaivaavat niin monia projekteja nykyaikana. Jacobson ja Ng esittävätkin, että onnistunut ratkaisu ongelmaan vaatii kahta asiaa: ohjelmistotuotannon apuvälinettä,

jonka avulla tällaisten vaatimusten eritteleminen muista vaatimuksista on mahdollista ohjelmakoodiin saakka, sekä yksinkertaista ja tehokasta komponointimekanismia, jotta kunkin vaatimuksen suunnittelu ja implementaatio luo lopputuloksena halutunlaisen järjestelmän.

Laddad (2002a) jakaa aspektiohjelmoinnin vaiheet kolmeen selvästi toisistaan erotettavaan osaan: aspektien dekomponointi (aspectual decomposition), vaatimusten toteuttaminen (concerns implementation) ja aspektien komponointi (aspectual recomposition). Vaiheet on esitetty kuvassa 3.1.

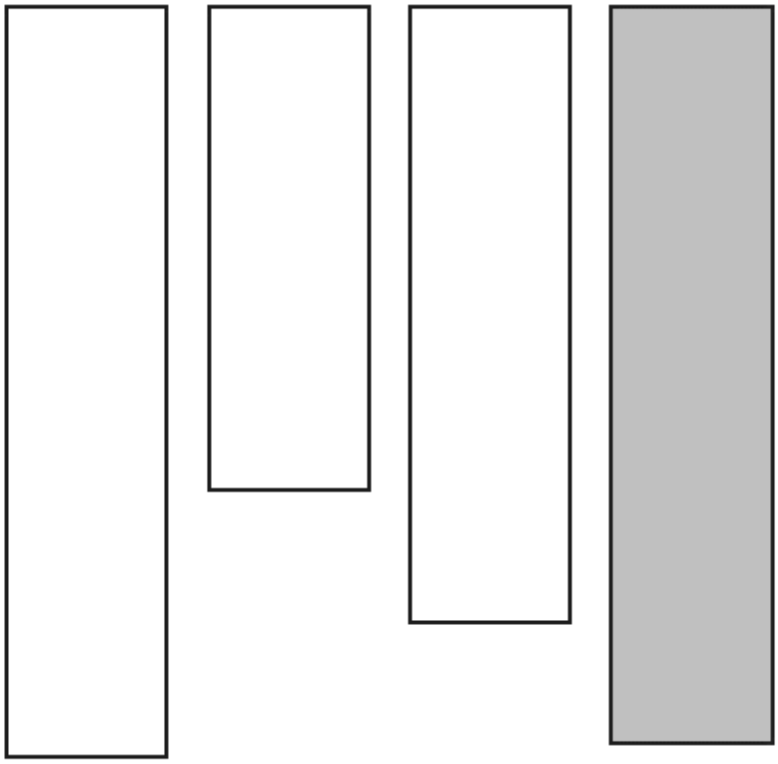


Kuva 3.1. AOP:n vaiheet (Laddad, 2002a)

Aspektien dekomponointiin käytetään kuvassa 2.3 esitettyä prisma-analogiaa, jotta liiketoiminnalliset vaatimukset saadaan erotettua poikkileikkaavista ominaisuuksista. Tämän jälkeen jokainen eroteltu vaatimus toteutetaan erillään toisistaan. Aspektien komponointivaiheessa aspektien integroija luo komponointisäännöt luomalla modularisoituja yksiköitä eli aspekteja. Uudelleenkomponointiprosessi eli punominen tai integroiminen käyttää tätä informaatiota komponoidakseen lopullisen järjestelmän.

3.2 Aspektiohjelmoinnin määritelmä

Aspektiohjelmoinnin pyrkimys on parantaa ohjelmistojen luotettavuutta pääasiassa tarjoamalla menetelmiä poikkileikkaavien ominaisuuksien modularisoimiseksi sekä helpottamaan ylläpitoa ja uudelleenkäytettävyyttä. Kuvassa 3.2 havainnollistetaan aspektiohjelmoinnin tapaa parantaa ohjelman modulaarisuutta.



Kuva 3.2. Poikkileikkaamisen edut (Pawlak & al.,2005, s.9)

Perinteisillä menetelmillä poikkileikkaavien ominaisuuksien implementaatio hajaantuu pitkin poikin lähdekoodia, jolloin tapahtuu ohjelmakoodin sekoittumista ja pirstaloitumista. Aikaisemmin kuvassa 2.1 on esitetty poikkileikkaavan ominaisuuden implementaatio olio-ohjelmointikielellä. Siinä poikkileikkaavan ominaisuuden toteutus pirstaloitui pitkin poikin ohjelman muita moduuleita. Kuvassa 3.2 on esitetty vastaavan poikkileikkaavan ominaisuuden toteutusta kun siihen on käytetty aspektiohjelmointia. Kolmen ydinominaisuuden toteuttavan moduulin rinnalle on tullut neljäs moduuli, joka sisältää kokonaisuudessaan

poikkileikkaavan ominaisuuden toteutuksen. Näin ohjelmakoodi ei takkuunnu tai pirstaloidu ja sen ymmärrettävyys ja ylläpidettävyys paranee.

Khatchadourianin (2006, s.3) mukaan aspektiohjelmointi on ohjelmointimetodologia, joka rakentuu olemassa olevien metodologioiden, kuten OOP:n ja proseduraalinen ohjelmoinnin päälle kasvattaen niitä käsitteillä ja rakenteilla, jotta poikkileikkaavien ominaisuuksien modularisoiminen olisi mahdollista. Filman ja Friedman (2000) vuorostaan esittävät, että AOP:n tärkeimmät ominaisuudet ovat kvantifiointi (quantification) ja tietämättömyys (obliviousness). Näiden kahden käsitteen ja niiden välisen suhteen kautta on mahdollista kuvata aspektiohjelmointikieliä: miksi jotkut ohjelmointikielet saattavat vaikuttaa aspektiohjelmointikieliltä, mutta eivät sitä ole sekä minkä takia tietyistä ohjelmointikielistä olisi yksinkertaista luoda aspektiohjelmointikieliä. Tietämättömyydellä tarkoitetaan sitä, miten poikkileikattava ohjelma on tietämätön siitä, miten aspektit muokkaavat sen toimintaa sekä missä vaiheessa ohjelman muokkaaminen tapahtuu. Kvantifioinnilla vuorostaan viitataan siihen, miten aspektit voivat mielivaltaisesti vaikuttaa eri kohtiin ohjelmassa. Filman ja Friedman (2000) määrittelevätkin, että AOP:n voi ymmärtää tavoitteena luoda määriteltävissä olevia väittämiä ohjelman toiminnasta niin, että nämä määrittelyt pätevät ohjelmassa, jonka on tehnyt niistä tietämätön ohjelmoija.

Kiczales & al (1997, s.8) määrittelevät, että AOP:n tavoitteena on tukea ohjelmoijaa erottelemalla siististi ja virheettömästi komponentit ja aspektit toisistaan tarjoamalla mekanismeja joiden avulla on mahdollista abstraktoida ja muodostaa niitä kokonaisjärjestelmän tuottamiseksi. Elradin & al (2001, s.31) mukaan pohjimmiltaan AOP on ohjelmointitekniikka. Kuten kaikkien ohjelmointitekniikoiden, myös AOP:n tulee ottaa huomioon se, mitä ohjelmoija voi sanoa sekä se, miten tietokonejärjestelmä tulee toteuttamaan toimivan järjestelmän. Näin ollen aspektipohjaisten järjestelmien ei ole tarkoitus ainoastaan tarjota tapaa esittää poikkileikkaavia ominaisuuksia, vaan myös taata, että nämä mekanismit ovat käsitteellisesti yksinkertaisia ja niiden toteutukset tehokkaita.

3.3 Keskeiset käsitteet

Aspektiohjelmoinnin paradigma esittelee useita uusia käsitteitä, kuten liitoskohta, liitoskohtamääritys, aspekti tai punominen. Nämä uudet käsitteet eivät kuitenkaan korvaa entisiä, kuten oliot, luokat ja proseduurit, vaan ne voidaan nähdä entisiä täydentävinä. Kohdissa 3.1 ja 3.2 on käsitelty jo useita aspektiohjelmoinnin paradigman keskeisistä käsitteistä. Tässä luvussa tarkastellaan niitä vielä aikaisemmin mainitsemattomia käsitteitä, jotka kaikkien AOP-kielien tulee jollain lailla toteuttaa ja jotka luovat käytännössä ne mekanismit, joiden avulla aspektiohjelmoinnin paradigman tavoitteet, kuten poikkileikkaavien ominaisuuksien modulaarinen toteuttaminen, on mahdollista saavuttaa.

Aspektiohjelmoinnin käsitteet eivät siis syrjäytä sen pohjalla olevien metodologioiden käsitteitä, vaan täydentävät niitä. Tyypillisillä aspektiohjelmointikielillä on mahdollista määritellä vain säännöt, joiden mukaan ohjelmaa tullaan poikkileikkaamaan, jolloin ydinvaatimusten toteuttaminen täytyy suorittaa aspektiohjelmointikielen pohjalla olevalla ohjelmointikielellä. Aspektiohjelmointikielien rakennetta käsitellään tarkemmin kohdassa 3.5.

3.3.1 Liitoskohta

Aspektit eivät voi poikkileikata olioita sattumanvaraisesti, vaan ne voivat lisätä ylimääräistä tai korvaavaa toiminnallisuutta ainoastaan tarkoin määriteltyihin kohtiin ohjelman suorituksessa (Viega & Voas, 2000, s.20). Näitä tarkoin määriteltyjä kohtia poikkileikattavassa järjestelmässä kutsutaan *liitoskohdiksi (join point)*. Kukin aspektiohjelmointikieli määrittää oman mallinsa liitoskohdille. Tätä kutsutaan nimellä *liitoskohtamalli (join point model).*

Rajanin (2007) mukaan liitoskohdalle löytyy kaksi erilaista määritelmää. AspectJ:n ohjelmointiopas määrittelee sen uudeksi käsitteeksi, jolla kuvataan tarkoin määriteltyä kohtaa ohjelman suorituksessa. Toisaalla taas liitoskohta kuvataan implisiittiseksi, kielellisesti määritellyksi kohdaksi ohjelman suorituksessa. Menetelmää joukon liitoskohtia yhteen ka-

saamiseksi voidaan kutsua termillä *kvantifiointi (quantification)*. Käsite kvantifioitavissa olevista liitoskohdista onkin keskeinen aspektiohjelmoinnissa.

Krishnan (2005, s.2) jakaa liitoskohdat staattisiin ja dynaamisiin. *Staattiset liitoskohdat (static join points)* ovat kohtia ohjelmassa, jonne on mahdollista lisätä uusia jäseniä. *Dynaamiset liitoskohdat (dynamic join points)* taas ovat kohtia ohjelmavuossa. Krishnan jakaa dynaamiset liitoskohdat edelleen *suoritukseen liittyviin liitoskohtiin (execution join points)*, *kutsuihin liittyviin liitoskohtiin (call join points)* sekä *kenttien käsittelyyn liittyviin liitoskohtiin (field access join points)*. Tyypillisiä suoritukseen liittyviä liitoskohtia ovat metodin tai konstruktorin suorittaminen tai objektin alustaminen. Kutsuihin liittyvät liitoskohdat taas ovat usein metodin tai konstruktorin kutsuja ja kenttien käsittelyyn liittyvät liitoskohdat kentän arvon muuttamista tai siihen viittaamista.

Oikeiden liitoskohtien valinta on yksi keskeisimpiä asioita aspektiohjelmoinnissa. Hyvin valittujen liitoskohtien kautta on mahdollista saavuttaa haluttu taso ohjelmakoodin ilmaisukyvylle, uudelleenkäytettävyydelle, muokattavuudelle ja vakaudelle.

3.3.1.1 Liitoskohtamalli

Kukin aspektiohjelmointikieli määrittää oman mallinsa sille, millaisia kohtia ohjelman suorituksessa voidaan poikkileikata. Tätä mallia liitoskohtamäärittelyille kutsutaan nimellä liitoskohtamalli (Kiczales & al, 2001, s.60). Tyypillisiä liitoskohtamäärittelyitä ovat metodin tai funktion kutsut, muuttujan käsittely tai luokan alustaminen. Teoriassa olisi esimerkiksi mahdollista määritellä Javalle aspektilaajennus, jonka liitoskohtamalli sallii poikkileikkaamisen ainoastaan virhetilanteiden catch-osioiden suoritukseen.

AOP-paradigman kehittäjien mukaan liitoskohtamalli on kriittinen elementti minkä tahansa aspektiohjelmointikielen mekanismien suunnittelussa. Malli kuvaa yleisen kehyksen sille, miten ohjelman aspektikoodin ja ei-aspektikoodin toiminta voidaan sujuvasti koordinoida.

3.3.2 Liitoskohtamäärittely

Aspektit laajentavat ohjelman olemassa olevaa toiminnallisuutta kehotteiden ja *liitoskohtamäärittelyiden (pointcut)* avulla. Liitoskohtamäärittely identifioi ohjelman suorituksen kohdat pohjalla olevassa ohjelmakoodissa eli kokoaa yhteen toisiinsa liittyvät liitoskohdat. Lisäksi se voi mahdollisesti sisältää arvoja, jotka liittyvät yhteen kokoamiensa liitoskohtien ajonaikaiseen kontekstiin. Liitoskohtamäärittelyt voivat olla nimettyjä tai anonyymejä. Nimetyt liitoskohtamäärittelyt mahdollistavat aspektien uudelleenkäytettävyyden.

Liitoskohtamäärittelyitä voidaan pitää liitoskohtien abstraktioina. Masuhara & al. (2006, s.131-132) määrittelevät tälle kolme eri syytä:

1. Liitoskohtamäärittelyt voivat antaa nimen joukolle liitoskohtia.
2. Liitoskohtien, esimerkiksi tyyppiin tai parametreihin liittyvästä, erilaisuudesta huolimatta ne voidaan koota yhteen liitoskohtamäärittelyyn.
3. Liitoskohtamäärittelyt voivat erottaa konkreettisia määrittelyjä niihin liittyvien liitoskohtien joukosta kehotteiden sisällä eli kehotteen sisältä on mahdollista parametrisoida liitoskohtia.

Liitoskohdat voidaan jakaa kolmeen eri tyyppiin: *nimeen perustuvat (name-based), ominaisuuksiin perustuvat (property-based)* ja *cflow* (kutsuttu myös nimellä *control flow)*. Nimeen perustuvat liitoskohtamäärittelyt määritellään käyttämällä luokan metodien nimiä. Ominaisuuksiin perustuvat liitoskohdat vuorostaan määritellään liitoskohtien ominaisuuksien perusteella käyttämällä jokerimerkkejä haluttujen ominaisuuksien kuvaamiseen. Ominaisuudet voivat liittyä muun muassa metodin parametrinä ottamaan tyyppiin tai metodin palauttamaan arvoon. Cflow-liitoskohdat ovat vuorostaan staattisia pisteitä ohjelman ohjelmavuossa. (Krishnan, 2005, s.2-3)

3.3.3 Kehote

Kiczales & al (2001, s.335) määrittelee *kehotteen (advice)* metodinkaltaiseksi mekanismiksi, jonka avulla määritellään, että tietty ohjelmakoodin pätkä tulee ajaa jokaisessa liitoskohtamäärittelyn määrittelemässä liitoskohdassa. Kehote siis määrittelee itse lisättävän toiminnallisuuden ohjelmakoodin. Kehotteet koostuvat tyypillisesti paitsi lisättävästä ohjelmakoodista, myös liitoskohtamäärittelyistä, joissa kootaan yhteen ne pisteet ohjelman suorituksessa, joihin haluttu ohjelmakoodi tullaan lisäämään.

Kehotteita on kolmea päätyyppiä (Krishnan, 2001 s.3-4):

1. *before*-kehote suoritetaan juuri ennen liitoskohdan suorittamista
2. *after*-kehote suoritetaan heti liitoskohdan, esimerkiksi metodin, suorittamisen jälkeen
3. *around*-kehote suoritetaan liitoskohdan ympärillä. Toisin kuin before- ja after-kehotteet, around-kehote voi kontrolloida sitä, tullaanko liitoskohdan ohjelmakoodi suorittamaan.

Kehotteen ohjelmakoodi voidaan siis myös määritellä suoritettavaksi liitoskohdan ohjelmakoodin sijasta. Tätä ominaisuutta kutsutaan nimellä *kääriminen (wrapping)*. Kukin AOP-kielistä määrittää omat sääntönsä käärimiselle. Joissakin aspektiohjelmointikielissä liitoskohdan sisältämän ohjelmakoodin suoritus voidaan sivuuttaa kokonaan, toiset taas sallivat ainoastaan suoritettavan ohjelmakoodin muokkaamiseen.

3.3.4 Aspekti

Aspekti (aspect) on olio-ohjelmoinnin oliota vastaava modulaarinen yksikkö, jonka tarkoitus on koota yhteen tiettyyn yksittäiseen vaatimukseen liittyvä toiminnallisuus. Viega ja Voas (2000, s.20) määrittelevät, että käsitteenä aspektit ovat kuitenkin erilaisia kuin objektit. Aspektit voivat tarkkailla toisia objekteja ja reagoida niiden toimintaan. Ne ovat tavallaan vastakohta periytymiselle: siinä missä perittäessä luokat voivat valita mitä toiminnalli-

suuksia ne itseensä liittävät, aspektit valitsevat toiminnot, jotka toisiin objekteihin tullaan liittämään.

Clarke ja Baniassad (2005, s.4) määrittelevät aspektin ominaisuudeksi, jonka toiminnan käynnistää toinen ominaisuus sekä tämän ominaisuuden erilaiset tilat. Jos ominaisuutta ei olisi eroteltu aspektiin, sen toiminnallisuus tulisi käynnistää suoraan siihen liittyvän ominaisuuden ohjelmakoodista, jolloin näiden kahden erillisen ominaisuuden toteutukset sekoittuisivat yhteen. Lisäksi, koska ominaisuuden toiminnan käynnistäminen jouduttaisiin suorittamaan useista eri kohdista ohjelmakoodia, hajoaisivat ominaisuuden käynnistysmekanismit erilleen.

Pitkälti olio-ohjelmoinnin luokkien tapaisesti, aspektit ovat tyypitettyjä entiteettejä, jotka sisältävät toiminnallisuuksia. Ne kuitenkin poikkeavat luokista siinä, että niiden tarkoitus on vangita poikkileikkaava ominaisuus. Aspektit voivat myös sisältää uusia ohjelmointielementtejä, joita perinteiset luokat eivät voi (Viega & Voas, 2000, s.20). Perinteisesti aspektit koostuvat kehotteista, liitoskohtamäärittelyistä, tyyppien välisistä deklaraatioista sekä perinteisestä ohjelmakoodista.

3.3.5 Tyyppien väliset deklaraatiot

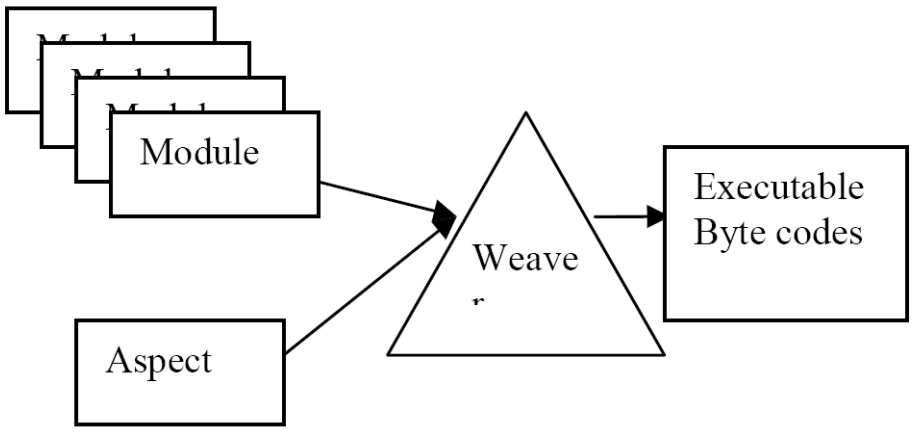
Krishnan (2005, s.4) mukaan ohjelman staattisia rakenteita, kuten yksityisiä muuttujia tai suhteita muihin luokkiin voidaan muuttaa *tyyppien välisten deklaraatioiden (inter-type declarations)* avulla. Ne muuttavat olemassa olevan luokan rakennetta lisäämällä yksityisiä muuttujia ja metodeja käännöksen aikana.

Siinä missä kehotteet on deklaraatio siitä, että aspekti tulee suorittamaan tietyn toiminnallisuuden ohjelman kontrollivuon valituissa liitoskohdissa, tyyppien väliset deklaraatiot ovat määrityksiä, että aspekti ottaa kokonaan vastuulleen toisten tyyppien tietyt ominaisuudet eli tyyppien välisten deklaraatioiden kohteet (Colyer & al., 2004a). Ne ovat käyttökelpoisia tilanteissa, joissa tarvitaan yhtä tai useampaa tyyppiä, jotta tietyn kyvyn (kuten tulostamisen) saavuttaminen olisi mahdollista johonkin yhteiseen tarkoitukseen.

Ennen AspectJ:n versiota 1.1 tyyppien väliset deklaraatiot tunnettiin nimellä *esittelyt (introductions)* ja termi on vieläkin osittain käytössä. Ominaisuuden tarkoitus on ylläpitää ja parantaa modulaarisuutta, eikä vain tarjota uutta tapaa lisätä kenttiä ja metodeja olemassa oleviin luokkiin (Colyer & al., 2004a).

3.3.6 Staattinen ja dynaaminen punominen

Kukin ohjelmointikieli määrittelee oman tapansa tuottaa ohjelmakoodista lopullinen ajettava ohjelma. Aspektiohjelmoinnissa tämä suoritetaan *punomisen (weaving)* kautta. Kuvassa 3.3 esitetään punomismekanismin perusidea.



Kuva 3.3. Punominen (Krishnan, 2005, s.4)

Kuten kuvassa 3.3 esitetään, punomisen kautta yhdistetään erillään toteutettujen perinteisten ja poikkileikkaavien ominaisuuksien implementaatiot yhdeksi ajettavaksi ohjelmaksi. Kukin aspektiohjelmointikieli määrittelee omat sääntönsä punomiselle. Ohjelmointikielien mukana tulee yleensä *punoja (weaver)*, jonka tehtävä on suorittaa punomistyö. Jos esimerkiksi AspectJ -ohjelmaa ei ajettaisi punojan kautta ennen varsinaista suoritusta, eivät aspektien toteutukset integroituisi osaksi ohjelmaa. Tällöin lopullinen järjestelmä olisi kyllä suoritettava Java-ohjelma, mutta siitä puuttuisivat poikkileikkaavien ominaisuuksien implementaatiot kokonaan.

Punominen voidaan suorittaa kahdella eri tavalla, staattisesti ja dynaamisesti (Forgáč & Kollár, 2007). *Staattinen punominen (static weaving)* suoritetaan ohjelman kääntämisen aikana. Tällöin liitoskohtamäärittelyn kehotteen sisältämä koodi sijoitetaan kuhunkin siihen liittyvään liitoskohtaan. Lopputuloksena on erittäin optimoitu punottu ohjelmakoodi, jonka suoritusnopeus on verrattavissa perinteisillä menetelmillä tehtyyn ohjelmakoodiin. Joissain tapauksissa staattinen punominen saattaa jopa parantaa ohjelman suorituskykyä. Staattisessa punomisessa on myös huonot puolensa. Kehotteiden koodin upottaminen osaksi alkuperäistä ohjelmakoodia vaikeuttaa aspektikoodin erottamista poikkileikattavasta koodista, jolloin aspektien muokkaaminen tai vaihtaminen jälkikäteen voi olla aikaa vievää tai jopa mahdotonta.

Dynaaminen punominen (dynamic weaving) vuorostaan sallii ohjelman ajonaikaisen muokkaamisen (Forgáč & Kollár, 2007). Krishnanin (2005, s.5) mukaan dynaaminen punominen voidaan suorittaa kahdella eri tavalla, *ohjelman lataamisen aikana (load time weaving)* tai *ohjelman ajon aikana (run time weaving)*. Ohjelman lataamisen aikana suoritettavassa punomisessa käännetyt aspektit ja luokat yhdistetään nimensä mukaisesti silloin kun ohjelmaa ladataan. Esimerkiksi Java-pohjaisissa aspektikielissä tämä yhdistäminen suoritettaisiin, kun luokkia ladataan Javan virtuaalikoneeseen. Ajonaikaisessa punomisessa aspektit ja luokat yhdistetään vuorostaan ajon aikana, aina kun ohjelman suoritus saavuttaa jonkin aspektien määrittelemän liitoskohdan. Dynaaminen punomisen avulla on mahdollista vapaasti lisätä ja poistaa aspekteja ohjelmasta esimerkiksi ohjelman ajoympäristön tai käyttäjän tarpeiden mukaisesti. Sekä paikallisten että globaalien muutosten tekeminen on helppoa ja aspektien poistaminen ohjelman suorituksesta yksinkertaista. Dynaaminen punominen vaikuttaa kuitenkin huomattavasti ohjelman suorituskykyyn. Esimerkiksi ajonaikaisessa punonnassa ohjelman suoritus hidastuu huomattavasti, kun taustalla joudutaan jatkuvasti tarkistamaan onko joku liitoskohdista on saavutettu. Forgáč ja Kollár (2007) huomauttavat lisäksi, että dynaaminen AOP saattaa olla turvaton vaihtoehto, koska se voi sallia myös vahingollisten kehotteiden punomisen.

3.4 Suhde olio-ohjelmointiin

Aspektiohjelmoinnilla on läheinen suhde olio-ohjelmointiin. Molempien paradigmojen tavoitteena on tukea ohjelmointiongelmien pilkkomista pienempiin, helpommin hallittaviin ongelmiin tarjoamalla muun muassa abstraktointi- ja modularisointimenetelmiä. Aspektit ovat tavallaan kuin olio-ohjelmoinnin olioita. Niiden tarkoitus on kapseloida ominaisuus yhteen paikkaan, jolloin ohjelman luettavuus ja ylläpidettävyys paranee. Poiketen olioista niiden on kuitenkin tarkoitus kapseloida järjestelmän poikkileikkaavia ominaisuuksia. Tämän saavuttamiseksi ne voivat sisältää uudenlaisia ohjelmointikomponentteja, joita oliot eivät voi sisältää, kuten kehotteita ja liitoskohtamäärittelyitä.

Huolimatta tästä läheisestä suhteesta ei aspektiparadigma ole kuitenkaan sidottu olioparadigmaan (Khatchadourian, 2006, s.4). Näin on mahdollista luoda aspektilaajennoksia myös ei-oliopohjaisille ohjelmointikielille. Khatchadourian esittääkin, että aspektiohjelmointi laajentaa sen pohjalla olevaa metodologiaa, olipa se sitten funktionaalinen, proseduraalinen tai oliolähtöinen. Se toimii yhdessä tämän alla olevan metodologian kanssa tarjoten siihen enemmän abstraktiota kapseloimalla poikkileikkaavat ominaisuudet aspekteiksi.

Tietyt aspektiohjelmoinnin piirteet muistuttavat olioparadigman peruspiirteitä. Yksi näistä on moniperiytyminen. Perimisen mekanismia OOP:ssä voidaan pitää menetelmänä lisätä ylimääräistä toiminnallisuutta luokkaan. Luokka voi laajentaa sen vanhempien toimintoja, jos se niin päättää. Moniperinnässä luokka voi periä useiden eri luokkien toimintoja, jopa sellaisten luokkien, joiden toiminnallisuus ja kuvaama semantiikka on perivän luokan toimialan ulkopuolella. Näin luokka voi lisätä itseensä toiminnallisuuksia jotka poikkileikkaavat koko järjestelmän (Khatchadourian, 2006, s.4). Aspektiohjelmoinnin periaatteilla luodun ohjelman lopputuloksena on järjestelmä, joka käyttää hyväkseen löyhästi yhteen nivottuja, modularisoituja poikkileikkaavien ja yleisten ominaisuuksien implementaatioita. Olio-ohjelmoinnin paradigma vuorostaan luo järjestelmän joka koostuu löyhästi yhteen nivotuista, modularisoiduista yleisten ominaisuuksien implementaatioista.

Esimerkiksi Flavors, New Flavors, CommonLoops ja CLOS kaikki tukevat moniperiytymistä, deklaratiivistä metodin yhdistämistä sekä avoimia luokkia (Kiczales & al., 2001, s.349). C++ vuorostaan tukee moniperiytymistä. Vaikkakin AspectJ sisältää elementtejä niistä, se myös tarjoaa tehokkaamman ja modulaarisemman tuen poikkileikkaamiselle kuin mitä näillä menetelmillä on mahdollista saavuttaa.

3.5 Aspektiohjelmointikielet

Clarke ja Baniassad (2005, s.3) määrittelevät, että aspektiohjelmointikielet eli AOP-kielet tukevat poikkileikkaavien ominaisuuksien, tai aspektien, toteuttamisen yhdessä paikassa sekä niiden toiminnan automaattisen siirtämisen useisiin kohtiin ohjelman suorituksessa. Näiden mekanismien kautta ohjelmoijan on mahdollista tuottaa toiminnallisuutta, joka peittää pohjalla olevan luokkamallin.

Kukin aspektiohjelmointikieli määrittelee omat tapansa aspektiohjelmoinnin peruskäsitteiden toteutukselle (Laddad, 2006). Aspektipohjaiset kielet ovat tyypillisesti muiden ohjelmointikielten laajennuksia, annotaatiopohjaisia tai XML-pohjaisia. Vaikkakin kukin kieli sisältää omat rakenteensa, ovat erot kielien välillä hyvin pieniä.

AOP-kielet koostuvat tyypillisesti kahdesta osasta (Laddad, 2002a):

1. *Vaatimusten implementaatio.* Yksittäisen vaatimuksen siirtäminen ohjelmakoodiin niin, että kääntäjä (compiler) voi muuntaa sen suoritettavaksi bittikoodiksi. Koska vaatimusten toteuttaminen suoritetaan proseduurien kautta, voidaan siihen käyttää perinteisiä ohjelmointikieliä kuten C, C++ tai Java.
2. *Punomissääntöjen määritteleminen.* Säännöt joiden perusteella yhdistetään eri vaatimusten toteutuksista lopullinen järjestelmä. Tähän tarkoitukseen tarvitaan kieli, esimerkiksi aspektiohjelmointikieli, jonka avulla määritellään millä tavalla eri osat toisiinsa yhdistetään.

Aspektiohjelmointikielet ovat siis tyypillisesti toisten ohjelmointikielien laajennuksia. Pohjalla olevalla ohjelmointikielellä toteutetaan ydinominaisuuksien ja poikkileikkaavien ominaisuuksien toiminnallisuudet. Ohjelmointikielen aspektilaajennusosalla vuorostaan määritellään säännöt, miten eri osat toimivat yhdessä, eli muun muassa miten poikkileikkaavien ominaisuuksien toteutukset tullaan lisäämään ydinominaisuuksien joukkoon.

Aspektiohjelmointikieliä on kehitetty jo runsaasti erilaisia. Osa niistä mahdollistaa staattisen poikkileikkaamisen, osa dynaamisen ja osa molemmat. Näitä ovat esimerkiksi JBoss AOP Framework, AspectWerkz, PROSE, Handi-Wrap, AspectS tai SMove. Suuri määrä olemassa olevista, muun muassa proseduraalisista ja oliokielistä, ovat jo saaneet omat aspektilaajennuksensa. Näitä ovat muun muassa Aspect.NET, JAC, AspectC++, PHPaspect, tai AspectCocoa. Jopa XML on saanut oman aspektilaajennuksensa, AspectXML:n.

Tässä tutkielmassa tutustutaan kahteen Java-ohjelmointikieleen perustuvaan AOP -ympäristöön, AspectJ ja Spring AOP. AspectJ on aspektiohjelmointikieli, joka kehitettiin rinnakkain aspektiparadigman kehitystyön kanssa. Sitä voidaankin pitää ensimmäisenä aspektiohjelmointikielenä. Toinen käsiteltävä aspektiohjelmointiympäristö, Spring AOP, vuorostaan on Spring Frameworkin moduuli, joka toteuttaa aspektiohjelmoinnin periaatteet. Luvuissa 4 ja 5 tullaan käsittelemään tarkemmin molempia ohjelmointiympäristöjä, sekä vertailemaan niiden eroja. Vertailussa käytetään lähdekirjallisuutta sekä testattuja esimerkkejä. Esimerkeissä ohjelmointiympäristöjen erilaisuutta pyritään kuvaamaan luomalla samanlaisten poikkileikkaavien ominaisuuksien toteutukset molemmilla käsitellyillä ohjelmointiympäristöillä. Molempien ohjelmointiympäristöjen esimerkkeinä käytettyjen sovelluksien lähdekoodit löytyvät liitteistä 1, 2 ja 3.

4. ASPECTJ

AOP:n ja AspectJ:n historia on yhtenäinen. AspectJ kehitettiin Palo Alto Research Center:ssä (PARC) Gregor Kiczalesin ja hänen tiiminsä toimesta. Vuonna 1997 Kiczales esitti ECOOP-konferensissa dokumenttinsa (Kiczales et al., 1997), jota voidaan pitää AOP:n lähtökohtana (Pawlak & al., 2005, s.23). Dokumentissa esiteltiin kohtia olio-ohjelmoinnin paradigmasta, joita Kiczales tiiminsä kanssa piti ongelmallisina, sekä esitti näihin oman alustavan ratkaisunsa. Dokumentissa esiteltiin myös ensimmäisen kerran monet aspektiohjelmoinnin keskeisistä termeistä, kuten poikkileikkaaminen ja punominen.

AspectJ:n ensimmäinen versio julkaistiin vuonna 1998. Vuonna 2001 ilmestyi ensimmäinen AspectJ:n merkittävämpi julkaisu, versio 1.0. Tällä hetkellä uusin AspectJ:n version on 5.0, kuudes versio on työn alla. Vuoden 2002 lopulla AspectJ -projekti siirtyi pois PARC:lta osaksi Eclipse -yhteisöä (Pawlak & al., 2005, s.23-24). Nykyään Eclipse tarjoaa useita erilaisia apuvälineitä AspectJ -kehitykseen muun muassa AspectJ Development Tools (AJDT) -lisäosan kautta. AJDT tarjoaa esimerkiksi erilaisia visualisointityökaluja, muun muassa aspektien rungon kuvauksille, ristiinviittauksille sekä AspectJ:n annotaatioille (Chapman, 2006). Eclipsen lisäksi AspectJ tarjoaa hyvän tuen myös muille suosituille IDE:ille, kuten Sun Microsystemsin Fortelle tai Borlandin JBuilderille.

Ohjelmointikielen vahvuuksia ovat sen lisäominaisuudet, joiden vuoksi AspectJ on noussut käytetyimmäksi aspektiohjelmointikieleksi. Sen mukana tulee aspektien punoja kääntäjän muodossa, aspekteja ymmärtävä debugger, dokumenttien generoija sekä itsenäinen aspektiselain, jonka avulla on mahdollista visualisoida se, miten aspekti leikkaa järjestelmän tai sen osan. AspectJ:n kääntäjällä ajetut ohjelmat voidaan suorittaa millä tahansa Javan virtuaalikoneella, eikä niille ole muita ajonaikaisia erityisvaatimuksia kuin, että ohjelman luokkapolusta tulee löytyä AspectJ:n ajonaikainen kirjasto, aspectjrt.jar (Colyer, 2005).

AspectJ:n mukana tuleva punoja *ajc* suorittaa punomisen kääntämisen aikana, vaikkakin version 1.2 jälkeen myös ohjelman lataamisen aikainen punominen on ollut mahdollista.

Sen ensimmäinen punoja tuotti Java-koodia, joka oli mahdollista kääntää Javan javac-kääntäjällä suoritettavaksi ohjelmaksi. Menetelmässä havaittujen ongelmien takia kuitenkin uudemmat AspectJ:n punojat osaavat luoda suoraan class -tiedostoja tai jar -paketteja lähdekoodista.

Kääntäjä käyttää implementaatiomenetelmää, joka perustuu vastuun jakamiseen (Kiczales & al., 2001, s.343). Kaikki järjestelmän osat, joihin kehotteet eivät vaikuta käännetään samoin kuin tavallinen Java-kääntäjä sen suorittaisi. Aspektit, sekä ne osat ohjelmasta, joihin kehotteet vaikuttavat, AspectJ:n kääntäjä muuntaa kolmella eri tavalla:

1. Jokaisen kehotteen runko käännetään tavalliseksi metodiksi.
2. Osat ohjelmasta, joihin kehote vaikuttaa, muunnetaan niin, että niihin lisätään staattisen liitoskohdat vastaavien dynaamisten liitoskohtien kohtiin.
3. Kaikki ylimääräinen dynaamisen liitoskodan tehtävän suorittava ohjelmakoodi lisätään luotuihin staattisiin liitoskohtiin.

Laddad (2003c) esittää AspectJ:lle useita tehokkaita käyttötapoja. Sitä voidaan käyttää esimerkiksi, paitsi aikaisemmin mainittuun lokin kirjoittamiseen, myös Design by contract (DBC) –menetelmän ja erilaisten suunnittelumallien sekä välimuistin käytön tehostamiseen.

Seuraavissa kohdissa tutustutaan AspectJ:n perusmekanismeihin sekä teorian että esimerkkien kautta. Liitoskohtamäärittelyiden sekä kehotteiden koodiesimerkkejä on kerätty liitteen 2 aspektiin ExampleAspect. Lyhyiden, ohjelmointikielen perusmekanismeihin keskittyvien esimerkkien lisäksi kohdassa 4.6 luodaan aspektiohjelma, joka lisää neljä eri ominaisuutta liitteen 1 Java-luokkaan. Aspektiohjelman lähdekoodi sekä testaukseen käytetyn suoritettavan Java-ohjelman lähdekoodit löytyvät liitteestä 2.

4.1 Liitoskohta

AspectJ:n liitoskohtamalli mahdollistaa yksitoista eri kohtaa ohjelmassa, joihin on mahdollista poikkileikata. Nämä liitoskohdat voidaan luokitella seuraaviin kategorioihin (Pawlak & al, 2005, s.42):

1. metodeihin liittyvät liitoskohdat
2. kenttiin liittyvät liitoskohdat
3. poikkeuksiin liittyvät liitoskohdat
4. konstruktoriin liittyvät liitoskohdat
5. ohjelman staattisiin osiin liittyvät liitoskohdat
6. kehotteisiin liittyvät liitoskohdat

Taulukossa 4.1 esitellään kaikki AspectJ:n liitoskohdat.

Taulukko 4.1. AspectJ:n liitoskohdat tyypin mukaan (Palo Alto Research Center, 2003, ch.3)

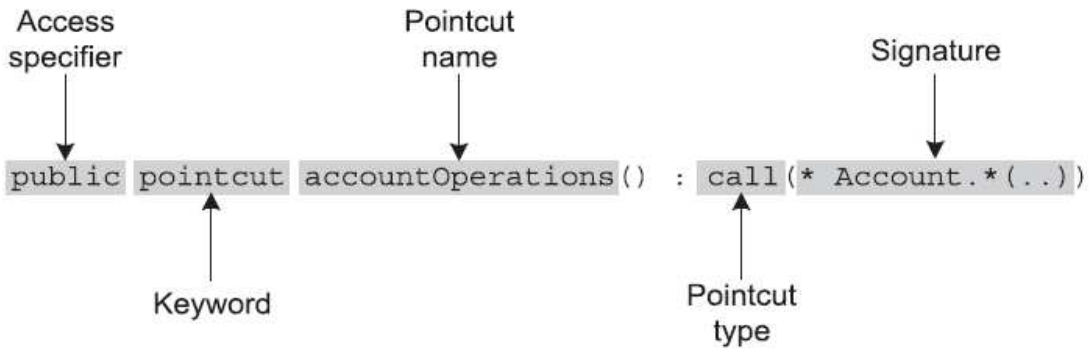
Tyyppi	Kuvaus
metodin kutsu	Kun metodia kutsutaan pois lukien ei-staattisten metodien super-kutsut.
metodin suoritus	Kun metodin sisältämä koodi suoritetaan.
konstruktorin kutsu	Kun objekti rakennetaan ja sen konstruktoria kutsutaan pois lukien super- ja this -konstruktorikutsut.
konstruktorin suoritus	Kun konstruktorin sisältämä koodi suoritetaan super- tai this -konstruktorikutsun jälkeen.
staattisen alustajan suoritus	Kun luokan staattinen alustaja suoritetaan.
olion esialustaminen	Ennen kuin olion alustaminen tietyssä luokassa suoritetaan.
olion alustaminen	Kun olion alustaminen tietyssä luokassa suoritetaan.
kentän arvon hakeminen	Kun ei-vakiokenttään viitataan.
kentän arvon asettaminen	Kun kentän arvo asetetaan.
poikkeuksenkäsittelijän suorittaminen	Kun poikkeuksenkäsittelijä suoritetaan.
kehotteen suorittaminen	Kun kehotteen sisältämä ohjelmakoodi suoritetaan.

Metodeille AspectJ tarjoaa kaksi erilaista liitoskohtaa, metodin kutsu ja metodin suorittaminen. Molemmissa tapauksissa halutut metodit tulee nimetä lausekkeiden (expressions) avulla. Lausekkeet muodostetaan AspectJ:n määrittelemien *jokerimerkkien (wildcards)* avulla, joita tullaan käsittelemään tarkemmin kohdassa 4.2.

Pawlakin et al. (2005, s.36) mukaan metodin suoritus ja kutsu -liitoskohdat eroavat muun muassa kontekstissa, jossa niitä käsitellään. Kutsun tapauksessa liitoskohta tapahtuu kutsuvan objektin kontekstissa, kun taas suorituksessa kontekstina on kutsuttu objekti. Kenttiin liittyvät arvon asettaminen ja haku -liitoskohdat ovat käteviä työkaluja, kun halutaan luoda aspekteja, jotka muokkaavat toisten olioiden sisäisiä arvoja. Viimeinen taulukossa 4.1 määritellyistä liitoskohdista voi määritellä toisissa kehotteissa olevia kohtia, joten sen on mahdollista muokata toisten aspektien toimintaa. Kehotteen suoritus -liitoskohdan käytössä tulee kuitenkin olla varovainen, koska niiden huolimaton käyttö saattaa johtaa päättymättömiin silmukoihin ohjelman suorituksessa.

4.2 Liitoskohtamäärittely

Kohdassa 3.3.2 määriteltiin että, liitoskohtamäärittely kokoaa yhteen joukon liitoskohtia sekä mahdollisesti niihin liittyvän kontekstin arvoja. AspectJ:ssä halutut liitoskohdat kuvataan kolmen eri osan avulla: liitoskohtamäärittelyn nimeäjä, mallikieli ja binäärioperaattorit. Kuvassa 4.1 on esitetty liitoskohtamäärittelyn rakenne.



Kuva 4.1. Liitoskohtamäärittelyn rakenne (Laddad, 2004, s.65)

Kuvan 4.1 mukaisesti liitoskohtamäärittely koostuu viidestä erillisestä osasta, näkyvyysmäärittelystä (access specifier), avainsanasta (pointcut), liitoskohtamäärittelyn nimestä (pointcut name), liitoskohtamäärittelyn nimeäjästä (pointcut type) sekä tarkenteesta (signature). Leikkauspisteiden näkyvyyteen pätevät Javan näkyvyysmäärittelyt public, protected ja private. Se voidaan jättää myös pois, jolloin käytetään Javan oletusnäkyvyysmäärettä.

Näkyvyysmäärittelyn jälkeen liitoskohtamäärittelyssä käytetään avainsanaa *pointcut*, jonka avulla määritellään, että seuraava kehotteen osio on liitoskohtamäärittely, eikä tavallinen Javan metodi. Avainsanan jälkeen määritellään liitoskohtamäärittelyn nimi. AspectJ:n liitoskohtamäärittelyt voivat olla joko anonyymejä tai nimettyjä. Kuten anonyymit luokat myös anonyymit liitoskohdat määritellään niiden käyttövaiheessa, esimerkiksi osana kehotetta tai toista liitoskohtaa määritellessä. Anonyymissä liitoskohtamäärittelyssä kuvataan ainoastaan liitoskohtamäärittelyn nimeäjä sekä signatuuri. Tällöin liitoskohtamäärittelyn nimi sekä näkyvyys jää määrittelemättä, joten siihen ei ole mahdollista viitata toisista kohdista ohjelmaa, eikä se näin ollen ole uudelleenkäytettävä. Nimetyt liitoskohtamäärittelyt vuorostaan ovat uudelleenkäytettäviä, koska niihin viittaaminen on mahdollista.

Seuraavassa kohdassa keskitytään liitoskohtamäärittelyn kahteen viimeiseen osaan. Niissä kuvataan liitoskohtamäärittelyn nimeäjä sekä tarkenne, joiden tehtävä yhdessä on koota yhteen halutut liitoskohdat.

4.2.1 Liitoskohtamäärittelyn nimeäjä

Liitoskohtamäärittelyn nimeäjät (pointcut designator) identifioivat liitoskohtamäärittelyitä. Kohdassa 4.1 esiteltiin AspectJ:n mahdollistamat liitoskohdat. AspectJ:n syntaksissa ne kuvataan ohjelmakoodissa taulukon 4.2 mukaisesti:

Taulukko 4.2. AspectJ:n liitoskohdat tyypin perusteella (Palo Alto Research Center, 2003, ch.A)

Liitoskohtamäärittelyn nimeäjän tyyppi	**Kuvaus**
Metodit ja konstruktorit	
call(*Signature)*	Minkä tahansa metodin tai konstruktorin kutsu, joka kutsukohdassa on yhteensopiva *Signature*-arvon kanssa.
execution(*Signature)*	Minkä tahansa metodin tai konstruktorin suoritus, joka suorituskohdassa on yhteensopiva *Signature*-arvon kanssa.
Kentät	
get(*Signature)*	Mikä tahansa viittaus kenttään, joka on yhteensopiva *Signature*-arvon kanssa.
set(*Signature)*	Mikä tahansa arvon asettaminen kenttään, joka on yhteensopiva *Signature*-arvon kanssa.
Poikkeuksenkäsittelijät	
handler(*TypePattern)*	Mikä tahansa Throwable-tyypin poikkeuksenkäsittelijä, joka on yhteensopiva *TypePattern*-arvon kanssa.
Kehote	
adviceexecution()	Minkä tahansa kehotteen suorittaminen.
Alustukset	
staticinitialization(*TypePattern)*	Minkä tahansa staattisen alustuksen suorittaminen, jonka tyyppi on yhteensopiva *TypePattern*-arvon kanssa.
initialization(*Signature)*	Minkä tahansa objektin alustaminen, kun ensimmäisen kutsuttavan konstruktorin tyyppi on yhteneväinen *Signature*-arvon kanssa.
preinitialization(*Signature)*	Minkä tahansa objektin esialustaminen, kun ensimmäisen kutsuttavan konstruktorin tyyppi on yhteneväinen *Signature*-arvon kanssa.
Leksikaaliset	
within(*TypePattern*)	Mikä tahansa liitoskohta ohjelmakoodissa, joka on määritelty *TypePattern*-arvon kanssa yhteensopivan tyypin sisällä.
withincode(*Signature*)	Mikä tahansa liitoskohta ohjelmakoodissa, joka on määritelty *Signature*-arvon kanssa yhteensopiva metodin tai konstruktorin sisällä.
Instanceof-tarkistukset ja kontekstin paljastaminen	
this(*Type* or *Id*)	Mikä tahansa liitoskohta kyseisellä hetkellä suoritettavan objektin sisällä, kun objekti on *Type* tai *Id*-arvon määrittelemän tyypin ilmentymä.
target(*Type* or *Id*)	Mikä tahansa liitoskohta kyseisellä hetkellä kohteena olevan suoritettavan objektin sisällä, kun objekti on *Type* tai *Id*-arvon määrittelemän tyypin ilmentymä.

Taulukko 4.2. AspectJ:n liitoskohdat tyypin perusteella (Palo Alto Research Center, 2003, ch.A) (jatk.)

Liitoskohtamäärittelyn nimeäjän tyyppi	**Kuvaus**
args(*Type* or *Id, …)*	Mikä tahansa liitoskohta, jossa argumentit ovat *Type* tai *Id*-arvojen tyyppisiä.
Kontrollivuo	
cflow(*Pointcut*)	Mikä tahansa liitoskohtamäärittely *Pointcut*-arvon määrittelemän liitoskohtamäärittelyn kontrollivuossa, mukaan lukien itse liitoskohtamäärittely.
cflowbelow(*Pointcut*)	Mikä tahansa liitoskohtamäärittely *Pointcut*-arvon määrittelemän liitoskohtamäärittelyn kontrollivuossa, pois lukien itse liitoskohtamäärittely.
Ehdolliset	
if(*expression*)	Mikä tahansa liitoskohta, kun *expression*-arvo on tosi.

Liitoskohtamäärittelyiden tyyppimäärittelyillä ei ole kuitenkaan vielä mahdollista saavuttaa riittävän tarkkaa tasoa liitoskohtien kuvaamiseksi. Kuten taulukossa 4.2 voi huomata, sisältävät erilaiset liitoskohtamäärittelyn tyyppimäärittelyt lisäarvoja, kuten *TypePattern, Signature* tai *Expression*. Niiden avulla tarkennetaan liitoskohtamäärittelyt tarpeeksi yksiselitteisesti, jolloin halutut liitoskohdat on mahdollista kuvata tarpeeksi tarkasti ilman, että mukaan kuvataan ylimääräisiä liitoskohtia tai haluttuja jää pois. Suurin osa liitoskohdista käyttää tarkenteena signatuuria, joka kootaan tietyn syntaksin mukaan.

Signatuuria, samoin kuin muiden liitoskohdan nimeäjän tarkenteita varten AspectJ tarjoaa mallikielen, joka sallii sellaisten ilmaisujen määrittelyn, jotka implisiittisesti merkitsevät tiettyjä metodeja tai luokkia. Jokerimerkkejä on mahdollista käyttää seuraavien luokan tai järjestelmän ominaisuuksien kuvaamiseen (Pawlak & al., 2005, s.31-32):

1. metodin ja luokan nimet
2. metodin allekirjoitus
3. paketin nimi
4. luokan perivät luokat

AspectJ:n jokerimerkkien mekanismi sisältää vain kolme eri merkkiä, mutta yhdistettynä liitoskohtamäärittelyiden nimeäjiin ja binäärioperaattoreihin on mahdollista saavuttaa monipuolinen kuvaus liitoskohdille. AspectJ:n jokerimerkit on lueteltu taulukossa 4.3.

Taulukko 4.3. AspectJ:n jokerimerkit (Laddad, 2004, s.67)

Kuvaus	**AspectJ:n syntaksi**
Mikä tahansa määrä kirjaimia, lukuun ottamatta pistettä	*
Mikä tahansa määrä kirjaimia, sekä mikä tahansa määrä alipaketteja	..
Mikä tahansa annetun tyypin aliluokka tai alirajapinta	+

Signatuurissa määritellään liitoskohdan näkyvyys, mahdollinen palautusarvo sekä jokerimerkkien avulla kuvattu nimi. Jatkossa luvun kaikissa esimerkeissä tullaan viittaamaan liitteessä 1 olevaan MyObject Java-luokkaan. Esimerkissä käytettyjä kehotteita ja liitoskohtamäärittelyjä on koottu liitteen 2 ExampleAspect-aspektiin.

Kuvassa 4.2 on yksinkertainen signatuuri. Ensimmäisenä on määritelty liitoskohdan näkyvyys (public). Näkyvyyden määrittelemiseen voidaan käyttää Javan näkyvyysmääritteitä tai se voidaan jättää kokonaan pois, jolloin kuvataan mitä tahansa näkyvyyttä. Tämän jälkeen määritellään liitoskohdan palauttama arvo.

```
public * *(..)
```

Kuva 4.2 Signatuuri

Kuvassa 4.2 on määritelty, että liitoskohta voi palauttaa minkä tahansa arvon tai jättää arvon palauttamatta kokonaan (ensimmäinen tähtimerkki). Tähtimerkin tilalla on mahdollista käyttää mitä tahansa Javan alkeistyyppiä tai oliotyyppiä. Void-sanalla vuorostaan määritellään, että liitoskohta ei palauta mitään arvoa. Tämän jälkeen kuvataan jokerimerkkien avulla liitoskohdan nimi.

Liitoskohdan nimen kuvaamiseen voidaan käyttää kaikkia AspectJ:n jokerimerkkejä. Tähtimerkkiä (*) voidaan käyttää luokan tai metodin nimen kuvaamiseen sekä metodin palau-

tusarvon sekä näkyvyysmäärittelyn kuvaamiseen (public, protected tai private). Sitä voidaan käyttää myös jokerimerkkinä, kun halutaan kuvata vain osia nimestä (Pawlak & al., 2005, s.31) .

```
void cs.joensuu.fi.msorsa.MyObject.*(String)
```

Kuva 4.3. Tähtimerkki -jokerimerkki

Kuvan 4.3 esimerkki kuvaa kaikkia paketin cs.joensuu.fi.msorsa luokan MyObject metodeja, joiden näkyvyysmäärittely on mikä tahansa, palautusarvo on *void* ja parametri on String -arvoinen. Esimerkissä tähtimerkki siis kuvaa kaikkia luokan metodeja. Tähtimerkin korvaaminen esimerkiksi setValue –arvolla määrittelee, että liitoskohta sijaitsee luokan MyObject metodissa setValue.

```
void cs.joensuu.fi.msorsa.MyObj*.set*(String)
```

Kuva 4.4. Tähtimerkki –jokerimerkki

Kuvassa 4.4 on vuorostaan korvattu osia luokan ja metodin nimestä tähtimerkillä. Näin on mahdollista määritellä, että liitoskohta sijaitsee missä tahansa metodissa, joka sijaitsee luokassa, jonka nimi alkaa MyObj –sanalla ja metodissa, joka alkaa set –sanalla. Tähtimerkkiä voidaan käyttää myös nimen edessä. Esimerkiksi kuvan 4.4 set* -merkinnän korvaaminen *set* -merkinnällä tarkoittaa, että metodin nimen tulee sisältää set –sana.

Kahta pistettä (..) voidaan käyttää metodin parametrien kuvaamiseen tai paketin kuvaamiseen (Pawlak & al., 2005, s.31).

```
void cs..MyObject.*(..)
```

Kuva 4.5. Kaksi pistettä -jokerimerkki

Kuvassa 4.5 merkityt liitoskohtamäärittelyt ovat muuten samanlaisia kuin kuvan 4.3 esimerkissä, mutta merkityt metodit voivat ottaa minkä tahansa määrän minkä tahansa tyyppisiä parametrejä ja niiden voivat olla missä tahansa cs -paketin hierarkiassa.

Plusmerkkistä (+) jokerimerkkiä voidaan käyttää luokan alityyppien kuvaamiseen. Paitsi että se kokoaa kaikki luokan perivät luokat, voidaan sitä käyttää myös rajapinnan yhteydessä (Pawlak & al., 2005, s.32).

```
* cs.joensuu.fi.msorsa.MyObject+.*(..)
```

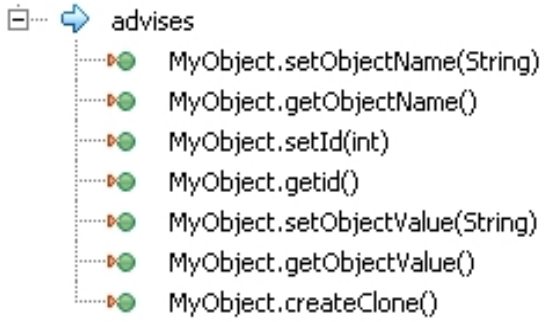
Kuva 4.6. Plus -jokerimerkki

Kuvassa 4.6 on merkitty kaikki cs.joensuu.fi.msorsa -paketissa olevan MyObject-luokan sekä sen perivien luokkien metodit, joiden parametrit, näkyvyysmäärittelyt ja palautusarvot ovat mitä tahansa. Esimerkissä viimeisen tähtimerkin korvaaminen esimerkiksi setValue-nimellä merkitsee kaikki MyObject-luokan ja sen perivien luokkien setValue-metodit. Kuvassa 4.7 on luotu jokerimerkkien avulla kokonainen liitoskohtamäärittely.

```
pointcut allOperations()
      : execution(* cs.joensuu.fi.msorsa.MyObject+.*(..));
```

Kuva 4.7. Liitoskohtamäärittely

Kuvassa 4.7 on luotu allOperations -niminen liitoskohtamäärittely, joka merkitsee kaikkien metodin suoritukset cs.joensuu.fi.msorsa.MyObject-luokassa. Kun liitoskohtamäärittelyn avulla poikkileikataan liitteessä 1 olevaan Java-luokkaan, poikkileikkaa liitoskohtamäärittely AspectJ:n mallinnustyökaluilla kuvattuna kuvan 4.8 mukaisiin ohjelmakohtiin.



Kuva 4.8 Poikkileikatut ohjelmakohdat

Signatuurissa käytetyn mallikielen lisäksi AspectJ tarjoaa kolme binäärioperaattoria. Niiden avulla on mahdollista yhdistää signatuureja yhden liitoskohtamäärittelyn sisällä sekä luoda uusia liitoskohtamäärittelyjä useasta nimetystä liitoskohtamäärittelystä. AspectJ:n käyttämät binäärioperaattorit on lueteltu taulukossa 4.4.

Taulukko 4.4. AspectJ:n binäärioperaattorit (Laddad, 2004, 67-68)

Operaattori	**AspectJ:n syntaksi**
NOT	!
AND	&&
OR	||

Binäärioperaattoreista AND tarkoittaa, että kehote suoritetaan, jos kaikki operaattorin yhdistämistä määrittelyistä ovat totta. OR vuorostaan tarkoittaa sitä, että kehote suoritetaan, jos vähintään yksi operaattorin yhdistämistä määrittelyistä on totta. NOT-operaattorin tapauksessa kehote suoritetaan vain silloin, jos määrittely saa arvon epätosi. Yhdistämällä operaattoreita samaan määrittelylauseeseen on mahdollista luoda monipuolisia liitoskohtien kuvauksia.

```
pointcut setOperations() : execution(* cs..MyObject.set*(..));

pointcut getOperations() :
            execution(* cs.joensuu.fi.msorsa.MyObject.get*(..));

pointcut fieldOperations() : setOperations() || getOperations();
```

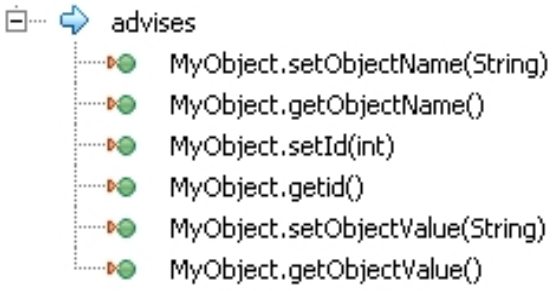
Kuva 4.9. Liitoskohtamäärittely ja binäärioperaattorit

Kuvassa 4.9 luodaan ensin kaksi nimettyä liitoskohtamäärittelyä. Liitoskohtamäärittely setOperations kokoaa yhteen kaikki MyObject-luokan metodinkutsut, joissa metodin nimi alkaa sanalla set. Liitoskohtamäärittely getOperations tekee saman get-alkuisille metodeille. Viimeisessä liitosmäärittelyssä fieldOperations kootaan yhteen kaikki MyObject-luokan kenttiin kohdistuvat set- ja get -operaatiot yhdistämällä OR-binäärioperaattorilla sekä setOperations- että getOperations-liitoskohtamäärittelyt. Erillisten nimettyjen liitoskohtamäärittelyjen tekeminen ei ole kuitenkaan välttämätöntä, vaan ne voidaan yhdistää myös kuvan 4.10 mukaisesti.

```
pointcut fieldOperations2() :
      execution(* cs..MyObject.set*(..)) ||
      execution(* cs.joensuu.fi.msorsa.MyObject.get*(..));
```

Kuva 4.10 Liitoskohtamäärittely ja binäärioperaattori

Sekä kuvan 4.9 liitoskohtamäärittely fieldOperations että kuvan 4.10 liitoskohtamäärittely ovat identtisiä. Kun kumman tahansa liitoskohtamäärittelyn avulla poikkileikataan liitteessä 1 olevaan Java-luokkaan, poikkileikkaa liitoskohtamäärittely AspectJ:n mallinnustyökaluilla kuvattuna kuvan 4.11 mukaisiin ohjelmakohtiin.



Kuva 4.11 Poikkileikatut ohjelmakohdat

4.3 Kehote

AspectJ:ssä kehote on metodinkaltainen mekanismi, jota käytetään ilmaisemaan, että tietty ohjelmakoodin pätkä tulisi suorittaa jokaisen liitoskohtamäärittelyn kokoamassa jokaisessa

liitoskohdassa. Kehotteet koostuvat paitsi liitoskohtamäärittelyistä, myös tavallisista Java-koodin pätkistä, joissa lisättäväksi haluttu ohjelmakoodi esitetään.

AspectJ sisältää viisi erilaista kehotetta (Pawlak & al., 2005, s.43). Yleisimmät kehotteet ovat *before*, *after* ja *around* sekä niiden lisäksi after-kehotteen erikoistapaukset, *after returning* ja *after throwing*.

Halutun tehtävän suorittamiseen on suositeltu käytettäväksi aina vähiten voimakasta kehotetyyppiä, koska niin on mahdollista saavuttaa optimaalisempi suoristuskyky sekä vähentää virhealttiutta. Mitä heikompi kehote on, sitä vähemmän se vaikuttaa ohjelman suoritusnopeuteen. Samalla kehotteen toteutus on yksinkertaisempi lähdekoodin tasolla.

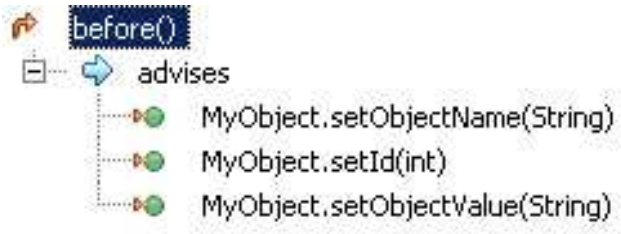
```
pointcut setOperations() : execution(* cs..MyObject.set*(..));

before() : execution(* cs..MyObject.set*(..)) {
      tulosta("Kenttä asetettu.");
}

before() : setOperations() {
      tulosta("Kenttä asetettu.");
}
```

Kuva 4.12. Before -kehote

Before -kehotteet suoritetaan ennen kuin joku liitoskohtamäärittelyssä identifioiduista liitoskohdista suoritetaan. Kuvassa 4.12 ylimpänä on luotu kahdella eri tavalla sama before-kehote. Kehotteen rakenteessa määritellään, että ennen minkä tahansa liitteen 1 MyObject-luokan set-nimellä alkavan metodin suoritusta tulostetaan oletustulostimelle teksti ”Kenttä asetettu.”. Kuvassa 4.12 on ylimpänä luotu setOperations-niminen liitoskohtamäärittely, joka kokoaa yhteen kaikki halutut liitoskohdat. Tämän jälkeen on luotu kehote, joka sisältää anonyymin liitoskohdan. Kuten kuvassa 4.12 voi huomata, ei kehotteen anonyymi liitoskohtamäärittely sisällä pointcut-avainsanaa tai nimimääritettä. Jos taas kehotteen sisältävän aspektin sisällä on luotu jokin nimetty liitoskohtamäärittely, voidaan siihen viitata kehotteesta, kuten kuvan 4.12 alemmassa kehotteessa on tehty. Tällöin before-avainsanan jälkeen viitataan haluttuun liitoskohtamäärittelyyn sen nimen perusteella. Koska sekä nimetty että

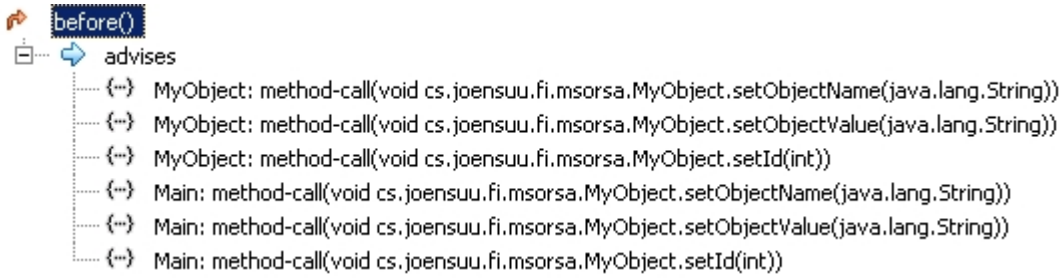
anonyymi liitoskohdan määrittely on samanlainen, ovat kuvan 4.12 molemmat kehotteet identtisiä.

Kuvan 4.12 esimerkin kautta on myös helppo kuvata call- ja execution -kehotteiden eroa. Jos esimerkin kehotetta käyttää liitteen 1 Java-luokan poikkileikkaamiseen, ovat poikkileikattavat liitoskohdat kuvan 4.13 mukaiset. Liitteen 1 MyObject-luokkaa on käytetty liitteen 2 Main-ohjelman suorituksessa.



Kuva 4.13 Execution-liitoskohtamäärittely

Kun kuvan 4.12 liitoskohtamäärittelyn execution -tyypin korvaa call-tyypillä, mutta kehotteen muut osat pysyvät samanlaisina, muuttuvat kootut liitoskohdat huomattavasti. Kuvassa 4.14 on mallinnettu AJDT:n työkaluilla call-liitoskohtamäärittelyn poikkileikkaaminen samassa ohjelmassa. Kuten kuvassa näkyy, liitoskohdat ovat siirtyneet kutsuttavasta MyObject-luokasta kutsuvaan Main-luokkaan.



Kuva 4.14 Call -liitoskohtamäärittely

After-kehote suoritetaan vuorostaan liitoskohdan suorittamisen jälkeen (Pawlak & al., 2005, s.44-45). After -kehotteen luomiseen käytetään samaa syntaksia kuin before-kehotteeseen. Ainoana poikkeuksena on before-avainsanan korvaaminen after -avainsanalla. Af-

ter-kehote suoritetaan aina liitoskohdan suorittamisen jälkeen huolimatta siitä, että liitoskohdan suorittaminen on saattanut päättyä virhetilanteeseen.

After returning -kehote suoritetaan liitoskohdan normaalin suorittamisen jälkeen. Kuvassa 4.15 on luotu after returning -kehote. Kehotteessa määritellään, että aina kun suoritetaan luokan MyObject metodi createClone, tulostetaan oletustulostimelle teksti ”Objektin nimi on (liitoskohdan palauttaman MyObject -olion nimi), mutta me kutsumme sitä nimellä klooniobjekti”.

```
after() returning (MyObject o) :
      execution(MyObject cs..MyObject.createClone(..)) {

      tulosta("Objektin nimi on " + o.getObjectName()
      + " mutta me kutsumme sitä nimellä klooniobjekti");
}
```

Kuva 4.15. After returning -kehote

Kuten kuvassa 4.15 olevassa esimerkissä näkyy, on kehotteessa viitattu liitoskohdan palauttamaan arvoon o-objektin kautta. Palautettua arvoa on mahdollista muokata missä määrin tahansa ennen sen palauttamista liitoskohtaa kutsuneelle luokalle. After throwing -kehote vuorostaan suoritetaan vain siinä tapauksessa, kun jokin liitoskohtamäärittelyn identifioimista liitoskohdista päättää suorituksensa synnyttämällä poikkeuksen. Kuvassa 4.16 on luotu yksinkertainen after throwing -kehote.

```
pointcut setOperations() : execution(* cs..MyObject.set*(..));

after() throwing (SetOperationException se) : setOperations() {

      tulosta("Asetettu id -arvo oli virheellinen.");
}
```

Kuva 4.16. After throwing kehote

Kuvan 4.16 kehotteessa määritellään, että aina kun jokin setOperations-liitoskohtamäärittelyn liitoskohdista päättyy setOperationException -virhetilanteen nousemiseen, tulee oletustulostimelle tulostaa teksti ”Asetettu id-arvo on virheellinen”. Aivan kuten after re-

turning -kehotteessa, myös after throwing -kehotteessa nousseeseen poikkeukseen pystytään viittaamaan määritellyn virhetilanteen nimen perusteella, tässä tapauksessa se -muuttujan.

Around -kehote suoritetaan sekä ennen liitoskohdan suorittamista että sen jälkeen. Around -kehote koostuu kolmesta osasta. Before -kohta suoritetaan ennen liitoskohdan suorittamista. Proceed -avainsanalla siirrytään itse liitoskohdan suorittamiseen ja suorituksen jälkeen suoritetaan kehotteen after -osio. Proceed -avainsanan käyttö on valinnaista. Jos kehotteen sisällä sitä ei määritellä, jätetään liitoskohdan sisältämä ohjelmakoodi kokonaan suorittamatta (Pawlak & al., 2005, s.43-46). Tämän mekanismin kautta around-kehote onkin ainoa, jolla on erikoisoikeus päättää tullaanko liitoskohdan ohjelmakoodi lainkaan suorittamaan.

```
MyObject around() : execution(* cs..MyObject.createClone(..)) {

	tulosta("CreateClone-metodia kutsuttu");
	MyObject o = proceed();
	tulosta("Objektin alkuperäinen nimi on " + o.getObjectName()
		+ " mutta muutetaan se Object2:ksi");
	o.setObjectName("Object2");

	return o;
}
```

Kuva 4.17. Around -kehote

Kuvassa 4.17 on luotu around -kehote. Kehotteessa ilmoitetaan ensin, että se palauttaa MyObject -tyyppisen muuttujan. Kehotteen before-osiossa, ennen saavutetun liitoskohdan suorittamista, oletustulostimelle kirjoitetaan teksti ”CreateClone-metodia kutsuttu.”. Tämän jälkeen käytetään proceed -avainsanaa liitoskohdan suorittamiseksi, jolloin poikkileikattava metodi palauttaa tietyn arvon. Liitoskohdan suorittamisen jälkeen, after-osiossa, liitoskohdan palauttamat arvo luetaan o-olioon. Tämän jälkeen oletustulostimelle tulostetaan teksti ”Objektin alkuperäinen nimi on (liitoskohdan palauttaman olion nimi), mutta muutetaan se Object2:ksi”, jonka jälkeen o-olion nimi muutetaan muotoon Object2. Lopuksi kehote palauttaa liitoskohtaa kutsuneelle luokalle muokatun MyObject-olion return-avainsanan kautta. Avainsanan puuttuminen kehotteesta vuorostaan johtaa tilanteeseen, jossa liitoskohdan palauttaman arvon siirtäminen kutsuvalle luokalle ohitetaan kokonaan. Kuvan 4.17 esimer-

kissä liitoskohtaa kutsunut luokka saa palautusarvona createClone-metodissa luodun MyObject –olion, mutta sen nimi on vaihtunut.

4.3.1 Liitoskohdan introspektio

Kehotteet vaativat usein pääsyä liitoskohdan sisältämiin arvoihin. Tätä ominaisuutta kutsutaan termillä *liitoskohdan introspektio (join point introspection)*. Termi introspektio viittaa tietyn sisäisen ilmiön tutkimiseen sekä tiedon etsimiseen siitä (Pawlak & al., 2005, s.33). Esimerkiksi tietyn operaation kirjaamiseksi lokiin kehote tarvitsee tietoa poikkileikattavasta metodista sekä sen argumenteista. Tätä tietoa kutsutaan termillä *konteksti (context)*. AspectJ tarjoaa liitoskohdat target(), this() ja args() kontekstin keräämiseen (Laddad, 2002b).

4.4 Tyyppien väliset deklaraatiot

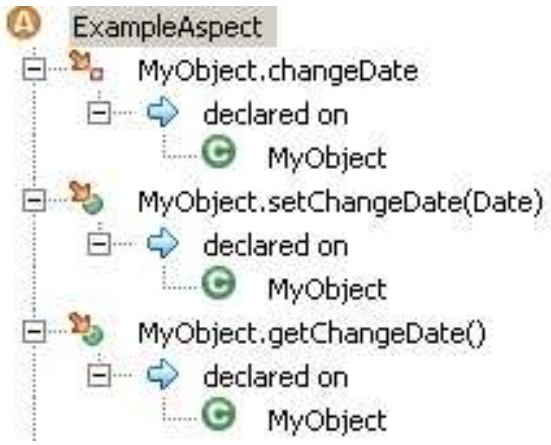
Aspektit voivat asettaa uusia jäseniä (kenttiä, metodeja tai konstruktoreita) toisille luokille (Palo Alto Research Center, 2003, ch.1). Niitä kutsutaan *tyyppien välisiksi deklaraatioiksi (inter-type declarations)*, koska ne leikkaavat läpi luokkien ja niiden hierarkioiden. Aspektit voivat myös määritellä, että toiset tyypit implementoivat uusia rajapintoja tai laajentavat muita luokkia.

Toisin kuin kehotteet, jotka toimivat pääasiassa dynaamisesti, deklaraatiot toimivat staattisesti kääntämisen aikana (Palo Alto Research Center, 2003, ch.3). Staattisia poikkileikkaamisen menetelmiä ovat myös käännöksen aikaisten virheiden ja varoitusten deklaraatiot (error and warning declaration) sekä poikkeusten pehmentäminen (exception softening) (Laddad, 2004, s.95).

```
private Date MyObject.changeDate;

public void MyObject.setChangeDate(Date newDate) {

      changeDate = newDate;
}

public Date MyObject.getChangeDate() {

      return changeDate;
}
```

Kuva 4.18. Muuttujan ja metodin esittely

Kuvassa 4.18 on tehty muuttujan ja metodin esittely aspektin sisällä. Muuttujan esittelyssä ilmoitetaan, että kukin MyObject -olio sisältää java.util.Date –arvoisen muuttujan Uusien muuttujien määrittelyssä tulee käyttää poikkileikattavan luokan nimeä muuttujan nimen määrittelyssä (MyObject), muuten muuttujan määrittely suoritetaan samoin kuin minkä tahansa Java-muuttujan määrittely. Sen näkyvyysmäärittely on private, joka tarkoittaa, että muuttuja on yksityinen aspektille. Private -näkyvyysmäärittely ei aiheuta ristiriitaa, vaikka MyAspect -luokassa olisi jo ennestään changeDate -niminen kenttä. Jos muuttujan kuitenkin määrittelee aspektin sisällä julkiseksi (public), syntyy ristiriita, jos poikkileikattava luokka sisältää samannimisen muuttujan jo ennestään. Esimerkissä toisena on kaksi uuden metodin esittelyä. Metodin esittelyssä tulee olla taas poikkileikattavan luokan nimi, jonka jälkeen esitellään metodin varsinainen nimi. Uuden metodin esittely suoritetaan muulta osin kuin Javassa. Kuvan esimerkissä luodaan changeDate –muuttujalle get- ja set –metodit. Tämän jälkeen metodeja on mahdollista käyttää aspektin sisällä. Kuvassa 4.19 on mallinnettu AspectJ:n työkalujen avulla kuvan 4.18 mukaista tyyppien välistä deklaraatiota.



Kuva 4.19 Tyyppien välinen deklaraatio

Kuvassa 4.20 on käytetty luokkaan lisättyä metodia around-kehotteen sisältä.

```
MyObject around() : execution(* cs..MyObject.createClone(..)) {

      MyObject o = proceed();
      o.setChangeDate(new Date());

      return o;
}
```

Kuva 4.20. Lisätyn metodin käyttäminen aspektin sisällä

Määriteltäessä poikkileikattava luokka implementoimaan uusia rajapintoja sekä laajentamaan toisia luokkia käytetään declare parents -avainsanaa.

4.5 Aspekti

Aspekti on olio-ohjelmoinnin luokkaa vastaava modulaarinen yksikkö (Laddad, 2002b). Aspektit koostuvat liitoskohtia kokoavista liitoskohtamäärittelyistä, niihin liittyvistä kehotteista sekä mahdollisesti ylimääräisistä Java-metodeista, jotka ovat aspektin sisäisten kehotteiden yhteisesti käytettävissä. Aspektit ovat AspectJ:n modulaarisia yksiköitä aivan kuten luokat Javassa. Aspekti yhdistää liitoskohtamäärittelyt ja kehotteet. Ne muistuttavat luokkia: aspekti voi sisältää metodeja ja kenttiä, laajentaa toisia luokkia tai aspekteja sekä im-

plementoida rajapintoja. Aspektit poikkeavat luokista kuitenkin siinä, että niistä ei ole mahdollista luoda uusia olioita *new* -avainsanan avulla. Aspektin syntaksi on kuvassa 4.21.

```
[ privileged ] [ Modifiers ] aspect id [extends Type]
                        [implements TypeList] [PerClause] { Body }
```

Kuva 4.21. Aspektin syntaksi (Colyer & al., 2004b)

Privileged -avainsanan avulla voidaan lisätä aspektin oikeuksia (Colyer & al., 2004b). Koska itse aspekteilla on suoraan oikeus poikkileikata myös luokkien yksityisiin metodeihin ja kenttiin, ei sen käyttö vaikuta aspektin oikeuksiin suoranaisesti. Avainsanan avulla voidaan kuitenkin lisätä aspektin sisältämien Java-koodipätkien (aspektin sisäisten metodien ja kehotteiden runkojen) oikeuksia niin, että ne voivat tarvittaessa käsitellä myös poikkileikattavan luokan yksityisiä kenttiä ja metodeja. Modifiers -kohdassa määritellään aspektin näkyvyys. Aivan kuten luokat, myös aspektit voidaan määritellä arvoilla kuten public, abstract, final tai static. Tämän jälkeen tulee avainsana *aspect*, jonka jälkeen määritellään aspektin yksilöivä nimi. Tarvittaessa aspektit voidaan määritellä laajentamaan toisia aspekteja tai implementoimaan rajapintoja. PerClause -kohdassa voidaan määritellä malli aspektin alustamiselle. Tyypillisesti aspektit ovat singleton-suunnittelumallin mukaisia ja se on myös aspektin oletusmalli, joten kohdan määrittely on harvoin tarpeellista. Aspektin otsikon määrittelyn jälkeen lisätään aspektin runko eli kehotteet, liitoskohtamäärittelyt sekä mahdolliset Java-metodit. Kuvassa 4.22 on luotu esimerkkiaspekti, joka sisältää liitoskohtamäärittelyn, sitä käyttävän kehotteen sekä yksityisen Java-metodin.

```
public aspect MyAspect {

pointcut setOperations() : execution(* cs..MyObject.set*(..));

before() : setOperations() {
            tulosta("Kenttä asetettu.");
}

private void tulosta(String line) {

      System.out.println(line);
}
}
```

Kuva 4.22 Aspekti

4.6 Esimerkkiaspekti

Kuvassa 4.23 luodaan testiaspekti, jonka tarkoitus on poikkileikata liitteessä 1 olevaa Java-luokkaa, jota suoritetaan liitteessä 2 olevan Main -ohjelman kautta. Poikkileikkaamisen tarkoitus on lisätä seuraavat ominaisuudet ohjelmaan:

- Ominaisuus 1. Kaikki luokan MyObject metodin createClone kutsut tulee kirjata oletustulostimelle.
- Ominaisuus 2. Ennen kuin luokan MyObject metodi createClone palauttaa klooniobjektin, tulee objektin nimi muuttaa muotoon ”Klooniobjekti”.
- Ominaisuus 3. Luokkaan MyObject tulee lisätä uusi java.util.Date-arvoinen kenttä sekä kentälle get- ja set-metodit.
- Ominaisuus 4. Luokan MyObject get- ja set-alkuisten metodien onnistunut suorittaminen tulee kirjata oletustulostimelle.

```
public aspect Aspect {

	pointcut setOperations() : execution(* cs..MyObject.set*(..));

	pointcut getOperations() :
		execution(* cs.joensuu.fi.msorsa.MyObject.get*(..));

	pointcut fieldOperations() : setOperations() || getOperations();

	pointcut createCloneCalls() :
		call(MyObject cs..MyObject.createClone(..));

	pointcut createCloneExecutions() :
		execution(MyObject cs..MyObject.createClone(..));

	//Ominaisuus 1
	before() : createCloneCalls() {
		System.out.println("Kutsutaan kloonausmetodia.");
	}

	//Ominaisuus 2
	MyObject around() : createCloneExecutions() {

		tulosta("CreateClone-metodia kutsuttu.");
		MyObject o = proceed();
		tulosta("Aspect: Objektin nimi ennen muutosta: "
			+ o.getObjectName());
		o.setObjectName("Klooniobjekti");

		return o;
	}
	//Ominaisuus 3
	private Date MyObject.date;

	public void MyObject.setDate(Date newDate) {
		date = newDate;
	}

	public Date MyObject.getDate() {

		return date;
	}
	//Ominaisuus 4
	after() : fieldOperations() {

		System.out.println
			("Get- tai set-metodi suoritettu onnistuneesti.");
	}

	private void tulosta(String line) {

		System.out.println(line);
	}
}
```

Kuva 4.23 Esimerkkiaspekti

Kuvassa 4.23 on ensimmäiseksi määritelty liitoskohdat, joihin testiaspektin tulee poikkileikata. Tämän jälkeen määritellyt neljä lisättävää toiminnallisuutta on määritelty omissa kehotteissaan luotuja liitoskohtamäärityksiä käyttäen. Kuten kuvassa voi nähdä, on kukin vaadittu ominaisuus mahdollista toteuttaa poikkileikkaavasti AspectJ:n avulla. Kuvan 4.23 lähdekoodi testattuna kokonaisuutena löytyy liitteestä 2, sekä poikkileikattava luokka liitteestä 1.

4.7 Poikkileikkaaminen annotaatioiden avulla

Javan versiosta 5 lähtien alusta on sisältänyt yleiskäyttöisten annotaatioiden (tai metadatan) apuvälineet, joiden avulla on mahdollista määritellä ja käyttää omia annotaatiotyyppejä (Sun Microsystems Inc, 2008). Annotaatiot määritellään oman syntaksin mukaan. Javan annotaatiot eivät vaikuta suoraan ohjelman semantiikkaan, mutta ne vaikuttavat siihen, miten ohjelmaa käsitellään erilaisissa työkaluissa ja kirjastoissa, joka saattaa kuitenkin loppujen lopuksi päätyä vaikuttamaan ohjelman semantiikkaan. Niitä voidaan lukea lähdekoodista, class -tiedostoista tai reflektiivisesti ajon aikana. Annotaatioita voidaan käyttää paketeissa tai tyyppien, kuten luokkien, rajapintojen, enumeraatioiden sekä muiden annotaatioiden määrittelyissä, konstruktoreissa, metodeissa, kentissä, parametreissä ja muuttujissa. Annotaatiot määritellään ohjelman lähdekoodissa @-symbolin avulla. Kehotteiden, liitoskohtamäärityksien ja aspektien määrittelemiseen käytetään omia annotaatioita, kuten @Aspect, @Pointcut tai @Around. Liitoskohtamäärityksien parametrien määrittelyyn käytetään AspectJ:n omaa kielioppia. Muu ohjelmakoodi kirjoitetaan Java-ohjelmointikielellä.

AspectJ 5 tukee annotaatioita aspekteihin, metodeihin, kenttiin, konstruktoreihin, kehotteisiin, sekä uusien jäsenten määrittelyihin aspektien sisällä. Lisäksi metodien ja kehotteiden parametreihin voidaan lisätä annotaatioita. Ne eivät kuitenkaan ole sallittuja liitoskohtamäärittelyihin tai declare –määrittelyn yhteydessä. (The AspectJ Team, 2005, ch.2) Tätä AspectJ:n sisältämää tapaa luoda aspekteja annotaatioiden avulla kutsutaan nimellä *@AspectJ annotaatiot (@AspectJ annotations).* Menetelmän avulla on mahdollista luoda aspekteja käyttämällä perinteistä Javan syntaksia. Luotuun ohjelmakoodiin lisätään @AspectJ annotaatioita, joita AspectJ:n punoja osaa tulkata tarpeen mukaan. Annotaatio -

ominaisuudet lisättiin osaksi AspectJ:tä vuonna 2005, kun kieli liitettiin yhteen AspectWerkz -aspektiohjelmointikielen kanssa (Colyer, 2005). Vaikkakin annotaatiot ovat yksi tärkeä aspektiohjelmointimenetelmä, ei tässä tutkielmassa tulla tarkastelemaan niitä lähemmin.

5. SPRING AOP

Spring Framework sai alkunsa Rod Johnsonin ”Expert One-on-One J2EE Design and Development” –kirjassa (Johnson, 2007) esittämistä ideoista, jonka jälkeen se on saavuttanut eräänlaisen de facto –standardin J2EE –sovellusten toteuttamisessa. Johnsonin (2005) mukaan se on yleiskäyttöinen Java/J2EE-sovelluskehys, jonka keskeisin idea on tarjota rajapinnan kautta *Inversion of Control (IoC)* –suunnittelumallin ja aspektiohjelmoinnin yksinkertainen toteutus. IoC:n perusajatus on se, että ohjelmoijan ei tarvitse luoda tarvitsemiaan olioita, vaan kuvata miten ne tulisi luoda. Johnson (2007) määritteleekin, että Spring Frameworkin tavoite on tehdä J2EE:n käyttö helpommaksi, sekä edistää hyviä ohjelmointikäytäntöjä.

Spring Frameworkissä ohjelman erilaisia komponentteja tai palveluja ei tarvitse yhdistää suoraan lähdekooditasolla, vaan niiden väliset suhteet kuvataan konfiguraatiotiedostoissa. Säiliö (Spring Frameworkin tapauksessa IOC-säiliö) luo kaikki oliot, yhdistää ne toisiinsa asettamalla tarpeelliset arvot ja päättää mitä metodeja suoritetaan. Tämä toteutetaan kehyksen sovelluskontekstin avulla, josta olio voi poimia tarvitsemansa palvelut. Sovelluskonteksti on tyypillisesti kuvattu vähintään yhtenä XML-tiedostona. Kehys sisältää myös integraatioita muihin apukirjastoihin ja sovelluskehyksiin ja sen avulla on mahdollista toteuttaa myös muita yleisesti tunnettuja suunnittelumalleja. Spring Frameworkin keskeisiä käsitteitä ovat *POJO (Plain Old Java Object)* ja *papu (bean).* POJO on perinteisestä Java-luokasta luotu olio. Pavut ovat olioiden edustuksia konfiguraatiotiedostoissa. Niissä kuvataan olioiden ominaisuudet, sekä niiden yhteydet muihin komponentteihin.

Aspektimenetelmät muodostavat tärkeän osan Spring Frameworkistä. Spring AOP on yksi kehyksen moduuleista, jonka tavoitteena on tarjota tehokkaita ja helppokäyttöisiä aspektilähtöisiä ratkaisuja yleisimpien poikkileikkaavien ominaisuuksien modularisoimiseksi. Monet Spring Frameworkin käyttäjät käyttävät aspektipohjaisia menetelmiä jopa tietämättään, koska suuri osa kehyksen muista osista nojaa vahvasti sen AOP-moduuliin. Spring AOP on toteutettu puhtaasti Javalla ja se tukee ainoastaan metodeihin kohdistuvaa poikki-

leikkaamista (Velde & al., 2008, s.6-7). Spring AOP koki täydellisen muodonmuutoksen kehyksen siirtyessä versiosta 1.2 versioon 2.0. Vanha versio tarjosi matalamman tason ratkaisun aspektipohjaisten periaatteiden tukemiseen, joka perustui aspektien, liitoskohtamäärityksien ja kehotteiden ohjelmalliseen lisäämiseen. Version 2.0 mukana tuli kaksi kokonaan uutta tapaa poikkileikata ohjelmia, skeemapohjainen poikkileikkaaminen sekä tuki AspectJ:n annotaatioille. Samalla sovelluskehyksen ensimmäinen aspektimenetelmä siirtyi taka-alalle, eikä sitä suositella enää käytettävän laisinkaan. Spring AOP:n tavoitteena ei ole olla täydellinen aspektiohjelmointiympäristö, kuten AspectJ:n, vaan tarkoituksena on, että sitä käytetään silloin, kun sovelluskehykseksi on valittu muutenkin Spring Framework. Se onkin enemmän sovelluskehyksen tarjoama lisätyökalu, ei itsenäinen aspektiympäristö. Tästä syystä Spring AOP ei tarjoakaan kaikkia niitä ominaisuuksia, jotka AspectJ:stä löytyy. Sen sijaan sen tavoitteena on keskittyä yleisimpiin tapauksiin, joissa aspekteja tarvitaan ja tarjota niihin yksinkertainen ja elegantti ratkaisu.

AOP Alliance on tiettyjen aspektiohjelmointiympäristöjen kehittäjien keskinäinen liittouma. He esittävät, että AOP tarjoaa paremman ratkaisun moniin ongelmiin kuin monet olemassa olevista tekniikoista. Sen tavoitteena on helpottaa ja standardoida AOP:n käyttöä sekä tarjota yhteensopivuutta erilaisten Java/J2EE-implementaatioiden välille. Tämä tavoite pyritään saavuttamaan luomalla standardit rajapinnat Java-pohjaisille ohjelmistoympäristöille. Spring AOP toteuttaa kaikki AOP Alliancen API:n määrittelemät rajapinnat.

Spring AOP on tehty puhtaasti Javalla, joten sen kääntämiseen tai suorittamiseen ei tarvita muuta kuin tavallinen Java-kääntäjä (Johnson & al., 2008). Se suorittaa aspektien punomisen ajonaikaisesti (dynaamisesti). Aspektiparadigman peruskäsitteiden lisäksi Spring AOP sisältää kaksi uutta termiä, joiden on hyvä olla tuttuja sitä käyttävälle. *Kohde (target)* tarkoittaa objektia, joka poikkileikataan yhdestä tai useammasta aspektista. Kohdetta kutsutaan myös nimellä *kehotettava objekti (advised object)*. *AOP Proxy* on Spring AOP:n luoma objekti. Sen tarkoitus on toteuttaa aspektin toiminnot, kuten kehotteen metodin suorittaminen. Spring Frameworkissä AOP Proxy on joko JDK:n dynaaminen proxy tai CGLIB proxy.

Seuraavassa kohdassa tutustutaan Spring AOP:n tapoihin toteuttaa AOP:n keskeiset käsitteet niiltä osin kuin ne ovat yhteisiä molemmille tässä tutkielmassa käsiteltäville menetelmille AspectJ ja Spring AOP. Tämän jälkeen keskitytään tarkemmin Spring AOP:n skeemapohjaiseen poikkileikkaamiseen. Poikkileikkaamista käsitellään samanlaisten esimerkkien kautta kuin AspectJ:tä luvussa 4. Lopuksi luodaan vastaavat poikkileikkaavat ominaisuudet kuin kohdassa 4.6 AspectJ:llä, sekä vertaillaan niiden toteutuksia sekä mahdollisia puutteita.

5.1 Miten Spring AOP toteuttaa AOP:n keskeisiä käsitteitä

Spring AOP:n liitoskohtamalli on huomattavasti AspectJ:tä suppeampi. Muun muassa liitoskohtia kuten call, get, set, initialization, adviceexecution tai withincode ei tueta, vaan niiden lisääminen Spring AOP:n ohjelmakoodiin aiheuttaa IllegalArgumentException-poikkeuksen nousemisen (Johnson & al., 2008).

Taulukossa 5.1 on kuvattu Spring AOP:n mahdollistamat liitoskohtamäärittelyt. Taulukon yhdeksän ensimmäistä liitoskohtamäärittelyä ovat AspectJ:n mukaisia ja niiden tarkentamiseen käytetään myös AspectJ:n mallikieltä. Spring AOP mahdollistaa myös uuden liitoskohtamäärittelyn tyypin. Bean-liitoskohtamäärittelyn avulla on mahdollista määritellä liitoskohtia Spring-pavun id:n tai nimen perusteella. Toisin kuin AspectJ, Spring AOP ei mahdollista kahden liitoskohtamäärittelyn yhdistämistä uudeksi liitoskohtamäärittelyksi binäärioperaattorien avulla. Sen sijaan binäärioperaattorien käyttö liitoskohtamäärittelyn signatuurissa on mahdollista.

Taulukko 5.1. Spring AOP:n liitoskohtamäärittelyjen tyypit (Johnson & al., 2008)

Liitoskohdan tyyppi	Liitoskohdan sijainti
execution	Metodin suorittamiseen liittyvät liitoskohdat.
within	Metodin suorittaminen määrittelyn tyypin sisällä.
this	Metodin suorittaminen kohdassa, jossa pavun referenssi on määritellyn tyypin instanssi.
target	Metodin suorittaminen kohdassa, jossa kohdeolio on määritellyn tyyppinen.
args	Metodin suorittaminen, jonka argumentit ovat määritellyn tyyppisiä.
@target	Metodin suorittaminen luokassa, jossa on suoritettavan objektin luokka sisältää määritellyn tyyppisen annotaation.
@args	Metodin suorittaminen kohdassa, jossa ajonaikana annettava argumentti sisältää määritellyn tyyppisen annotaation.
@within	Metodin suorittaminen sellaisen tyypin sisällä, joka sisältää määritellyn tyyppisen annotaation.
@annotation	Metodin suorittaminen kohdassa, jossa liitoskohdan subjekti sisältää määritellyn annotaation.
bean(idOrNameOfBean)	Liitoskohta määritellyn nimen tai id:n mukaisen Spring pavun sisällä.

Kuten AspectJ:ssä, myös Spring AOP:ssä esitellyistä liitoskohdista execution on ylivoimaisesti käytetyin, joten jatkossa tullaan keskittymään pääasiassa vain siihen. Kuvassa 5.1 esitellään formaatti execution -lausekkeelle..

```
execution(modifiers-pattern? ret-type-pattern
      declaring-type-pattern? name-pattern(param-pattern)
      throws-pattern?)
```

Kuva 5.1 Execution -lausekkeen formaatti (Johnson & al., 2008)

Palautettavan attribuutin tyypin (ret-type-pattern), nimen (name-pattern) ja parametrin (param-pattern) kaavoja lukuun ottamatta kaikki muut ovat vapaavalintaisia. Lausekkeen muodostukseen käytetään samaa AspectJ:n syntaksia, joka on esitelty kohdassa 4.2.

Kehotteista Spring AOP:n sekä skeemapohjainen- että annotaatiomenetelmä toteuttavat päätyypit before, after ja around, sekä after-kehotteen erikoistapaukset after returning ja after throwing. Liitoskohtien, liitoskohtamäärityksien ja kehotteiden käyttämistä skeemapohjaisessa AOP:ssä tullaan käsittelemään lähemmin kohdassa 5.2.

5.2 Skeemapohjainen AOP

Spring AOP mahdollistaa aspektiohjelmoinnin paitsi AspectJ:n annotaatioiden avulla, myös XML:ään perustuvan formaatin avulla. Skeemapohjainen aspektiohjelmointi perustuu aspektien määrittelemiseen käyttämällä ”aop” -nimiavaruuden tageja. Liitoskohtamäärityksien lausekkeiden ja kehotteiden rakenteen määrittelemiseen käytetään samaa formaattia kuin AspectJ:ssä (Johnson & al., 2008). Seuraavissa kohdissa tutustutaan Spring AOP:n tapaan toteuttaa samat AOP:n perusmekanismit, joihin AspectJ:n osalta tutustuttiin jo luvussa 4. Esimerkeissä käytetyt Spring AOP-konfiguraatiotiedostot sekä niihin liittyvät Java-luokkien lähdekoodit on koottu liitteeseen 3. Esimerkeissä poikkileikattava luokka on sama, jota käytettiin AspectJ:n esimerkeissä ja se löytyy liitteestä 1. Lisäksi kohdassa 5.2.7 pyritään luomaan kokonainen aspektiohjelma, joka tuottaa samanlaiset ominaisuudet liitteen 1 Java-luokkaan, kuin kohdan 4.7 AspectJ -aspekti. Tämän aspektiohjelman konfiguraatiotiedosto, siihen liittyvän Java -luokan lähdekoodi, sen testaamisen käytetyn suoritettavan Java-ohjelman lähdekoodit on myös koottu liitteeseen 3.

5.2.1 AOP- skeema

Spring Frameworkin käyttäminen vaatii aina omassa XML-tiedostossaan määritellyn sovelluskontekstin. Tiedostossa kuvataan eri komponenttien ja palveluiden väliset suhteet sekä koko sovelluksen konfiguraatio siltä osin kuin se on mahdollista. Kuvan 5.2 esimerkissä on esitelty sovelluskonteksitiedoston ”exampleAspectContext.xml”.

Kuvassa 5.2 esitettyjen rivien tulee olla Spring-sovelluskontekstitiedostossa ensimmäisenä. Tämä esimerkeissä käytetty konfiguraatiotiedosto on kokonaisuudessaan liitteessä 3. Ku-

vassa 5.2 esitetyt vahvennetut rivit ovat pakollisia, jotta AOP -nimiavaruuden tagien käyttäminen olisi mahdollista. Ensimmäinen vahvennetuista riveistä määrittelee nimiavaruuden Springin sovelluskontekstitiedostossa, eli osoittavat mitä XML-skeemaa tullaan käyttämään. XML-skeeman tarkoituksena on kuvata se, mitä elementtejä tiedostossa saa käyttää, mitä arvoja niihin saa asettaa sekä missä järjestyksessä elementit tulee määritellä. Kaksi alempana olevaa vahvennettua riviä kartoittavat aop -nimiavaruuden itse skeemaan (Velde & al., 2008, s.7).

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-
      context.xsd">
</beans>
```

Kuva 5.2. Sovelluskontekstitiedosto exampleAspectContext.xml

Sovelluskontekstitiedoston nimeämiselle ei ole rajoitteita. Sen tulee kuitenkin olla toteutettu validilla XML-kielellä ja se tulee ladata ohjelmaan sovelluskehystä käyttävän sovelluksen alussa. Kuvassa 5.3 on esitelty yksi tapa ladata luotu sovelluskonteksti ohjelmaan.

```
ApplicationContext context = new ClassPathXmlApplicationContext(
                    "exampleAspectContext.xml");
```

Kuva 5.3 Sovelluskontekstin lataaminen

Kuvan 5.3 esimerkki etsii ohjelman polusta tiedostoa nimeltä exampleAspectContext.xml. Jos tiedosto löytyy, Spring Framework lataa tiedoston suoritettavan ohjelman kontekstiksi. Kuvan esimerkki on otettu tutkielman esimerkkeihin käytetystä Java-ohjelmasta, jonka lähdekoodi on kokonaisuudessaan liitteissä 1 ja 3. Sovelluskontekstissa määriteltyjen tietojen perusteella Spring-säiliö luo suorituksessa tarvittavat oliot sekä määrittelee niiden väliset

suhteet. Samalla luodaan konfiguraatiotiedostossa määritellyt aspektit, jolloin ne punoutuvat osaksi ohjelman toimintaa.

Spring -konfiguraatiotiedostoissa kaikki aspektiohjelmointielementit tulee sisällyttää <*aop:config*> -elementin sisään. Elementtejä voi olla useampia yhdessä sovelluskontekstin konfiguraatiossa (Johnson & al., 2008).

5.2.2 Papumäärittelyt

Ennen liitoskohtamäärittelyjen, kehotteiden ja aspektien lisäämistä Spring-sovelluskontekstitiedostoon, tulee sovelluksessa käytetyille luokille tehdä papumäärittelyt. Kuvassa 5.4 on luotu papumäärittely esimerkeissä käytetystä MyObject-luokasta, jonka lähdekoodi löytyy liitteestä 1.

```
<bean id="MyObject" class="cs.joensuu.fi.msorsa.MyObject">
      <property name="objectName" value="java.lang.String"/>
      <property name="id" value="0"/>
      <property name="objectValue" value="java.lang.String"/>
</bean>
```

Kuva 5.4 Papumäärittely

Papumäärittelyt sisällytetään aina <*bean*> -tagien sisään. Kuvassa 5.4 ensimmäisellä rivillä on määritelty, että pavun id on ”MyObject” sekä pavun toteutuksen sisältävä luokka on ”cs.joensuu.fi.msorsa.MyObject”. Tämän jälkeen määritellään luokan ominaisuudet, jotka esitelllään <*property*> -tagien sisällä. Kuvassa 5.4 MyObject-pavulle on määritelty kolme ominaisuutta, String –arvoiset objectName ja ObjectValue sekä int-muuttuja id. Papumäärittelyn jälkeen Spring Framework tietää, että luokkaa tullaan käyttämään sovelluksessa.

5.2.3 Aspekti

Kuten kohdassa 5.2.1 mainittiin, kaikki aop -nimiavaruuden tagit tulee sisällyttää <aop:config> -tagin sisään. Aspektien luominen Spring-sovelluskontekstissa on yksinkertaista. Niiden luomiseen käytetään <*aop:aspect*> -tagia. Kuvassa 5.5 on esimerkkiaspekti, jonka sisälle luodaan seuraavissa kohdissa käytetyt esimerkit.

```
<aop:config>
      <aop:aspect id="ExampleAspect" ref="ExampleAspectImpl">
      </aop:aspect>
</aop:config>
```

Kuva 5.5 Yksinkertainen esimerkkiaspekti

Kukin <aop:aspect> -elementin tulee sisältää vähintään kaksi osaa, id ja ref -attribuutit. Id -attribuutti yksilöi aspektin antamalla sille nimen, jonka kautta siihen on mahdollista viitata. Ref -attribuutin osoittama papu vuorostaan toimii referenssinä luokkaan, jossa kehotteiden lisäämät ominaisuudet toteutetaan. Tätä kutsutaan nimellä *aspektia tukeva papu (aspect-backing bean)* (Velde & al., 2008, s.8). Kuvassa 5.5 aspektin nimi on ExampleAspect ja viitattavana oleva papu on nimeltään ExampleAspectImpl. Tämän lisäksi konfiguraatiotiedostoon tulee luoda ExampleAspectImpl-niminen papumääritys samaan tapaan kuin kohdassa 5.2.2 on tehty.

```
<bean id="ExampleAspectImpl"
      class="cs.joensuu.fi.msorsa.ExampleAspect"/>
```

Kuva 5.6 Yksinkertainen esimerkkiaspekti

Kuvassa 5.6 määritellään ExampleAspectImpl-niminen papu, joka viittaa luokkaan cs.joensuu.fi.msorsa.ExampleAspect. Kaikki aspektin sisällä olevien kehotteiden metodiviittaukset tulee olla toteutettuna tässä luokassa. Esimerkeissä käytetyn ExampleAspect-luokan lähdekoodi on liitteessä 3.

5.2.4 Liitoskohtamääritys

Spring AOP:n liitoskohtamäärityksien signatuuri on samanlainen kuin AspectJ:ssä. Liitoskohtamäärityksien merkitsemiseen käytetään <*aop:pointcut*> -tagia, jonka sisään sijoitetaan liitoskohdat yhteen kokoava lauseke. Liitoskohtamääritys voidaan sijoittaa suoraan <aop:config> -elementin sisään, jolloin sen käyttäminen on mahdollista useista eri aspekteista (Johnson & al., 2008). Kuvassa 5.7 on kuvaa 4.7 vastaava liitoskohtamäärittely, joka kokoaa kaikki luokan MyObject sekä sen perivän luokan metodin suoritukset.

```
<aop:config>
<aop:pointcut id="allOperations"
      expression="execution(* cs.joensuu.fi.msorsa.MyObject+.*(..))"/>
</aop:config>
```

Kuva 5.7 Nimetty liitoskohtamääritys

Liitoskohtamäärityksien tulee aina sisältää id-attribuutti. Sen tehtävä on yksilöidä liitoskohtamääritys, jolloin siihen viittaaminen on mahdollista. Liitoskohtamäärityksen lausekkeen (expression -attribuutin) muodostamiseen käytetään samoja sääntöjä kuin AspectJ:n signatuurissa. Ainoana poikkeuksena tästä on useamman lausekkeen yhdistäminen. Kuvassa 5.8 on yhdistetty kaksi eri signatuuria liitoskohtamäärittelyyn binäärioperaattorien avulla.

```
<aop:config>
<aop:pointcut id="fieldOperations" expression=
      "execution(* cs..MyObject.set*(..)) or
      execution(* cs.joensuu.fi.msorsa.MyObject.get*(..))"/>
</aop:config>
```

Kuva 5.8 Liitoskohtamääritys ja binäärioperaattori

Koska XML-tiedostoissa on hankala käyttää AspectJ:n binäärioperaattoreita sellaisenaan, voi ne korvata sanoilla *and*, *or* ja *not*, kuten kuvassa 5.8 on tehty. Spring AOP sallii binäärioperaattorien käytön ainoastaan liitoskohtamäärittelyn expression-attribuutin arvossa. Näin ollen kuvassa 4.9 esitelty kahden nimetyn liitoskohdan yhdistäminen uudeksi liitoskohdaksi ei ole mahdollista Spring AOP:ssä. Liitoskohtamäärittely voidaan sijoittaa myös

<aop:aspect> -elementin sisään, jolloin siihen viittaaminen on mahdollista ainoastaan aspektin sisältä. (Johnson & al., 2008)

5.2.5 Kehote

Kuten kohdassa 5.1 esitettiin, Spring AOP mahdollistaa kaikki viisi erilaista kehotetyyppiä. Taulukossa 5.2 on esitelty kunkin kehotetyypin aop-nimiavaruuden tagi.

Taulukko 5.2. Kehotteen tagit (Velde & al., 2008, s.9-11)

Kehotteen tyyppi	**Tagi**
Before	<aop:before>
After returning	<aop:after-returning>
After throwing	<aop:after-throwing>
After	<aop:after>
Around	<aop:around>

Kehote-elementit tulee sijoittaa aina aspektielementin sisään. Kuva 5.9 sisältää yksinkertaisen before-kehotteen. Kuvan esimerkki käyttää anonyymiä liitoskohtamäärittelyä, jolloin liitoskohtamäärittelyn tyyppi ja tarkenne asetetaan pointcut-attribuutin arvoksi.

```
<aop:before pointcut=
      "execution(* cs..MyObject.set*(..))"
      method="ilmoitaSetSuoritus"/>
```

Kuva 5.9 Before-kehote

Kuvan 5.9 esimerkissä method-attribuutti viittaa ennen saavutettua liitoskohtaa suoritettavaan metodiin. Esimerkin tapauksessa aina määriteltyyn liitoskohtaan saavuttaessa suoritetaan aspektia tukevan pavun ”ExampleAspect” metodi ”ilmoitaSetSuoritus”. Tällöin luokkaan ExampleAspect tulee sisältää samanniminen julkinen metodi.

```
public class ExampleAspect {

      public void ilmoitaSetSuoritus() {

            tulosta("Kenttä asetettu.");
      }
}
```

Kuva 5.10 Before-kehotteen lisäämä toiminnallisuus

Kuten kuvassa 5.10 on esitetty, aina kuvan 5.9 liitoskohtaan saavuttaessa tullaan suorittamaan kuvan 5.10 ilmoitaSetSuoritus –metodi. Kuvassa 5.9 liitoskohtamäärittely esitellään suoraan kehotteen sisällä. Kehotteesta on myös mahdollista viitata aspektielementin tai aop:config -elementin sisällä määriteltyihin liitoskohtamäärittelyihin, kuten kuvassa 5.11.

```
<aop:aspect id="ExampleAspect" ref="ExampleAspectImpl">

      <aop:pointcut id="setOperations" expression=
            "execution(* cs..MyObject.set*(..))"/>

      <aop:before pointcut-ref="setOperations"
            method="ilmoitaSetSuoritus"/>
</aop:aspect>
```

Kuva 5.11 Liitoskohtamäärittelyyn viittaaminen kehotteesta

Kehotteen ulkopuolella määriteltävän liitoskohtamäärittelyn käyttämiseen kehotteessa käytetään pointcut-ref -attribuuttia, jonka arvoksi asetetaan käytettävän liitoskohtamäärittelyn id. Muuten kehotteen rakenne on samanlainen, kuin anonyymiä liitoskohtamäärittelyä käytettäessä. Kuvissa 5.10 ja 5.11 suoritettu poikkileikkaaminen on identtinen kuvan 4.12 AspectJ:llä toteutetun before –kehotteen kanssa. Kuvassa 4.13 on esitetty AspectJ:n mallinnustyökaluilla se, mihin kohtiin MyObject –luokasta kuvan 4.12 esitetty kehote poikkileikkaa. Koska kuvan 4.12 esimerkin kootut liitoskohdat, kehotteen sisältämä toiminnallisuus sekä poikkileikattava luokka ovat identtisiä kuvien 5.10 ja 5.11 Spring AOP –kehotteen kanssa, pätee kuvan 4.13 mallinnus myös esiteltyihin Spring AOP:n before –kehotteisiin.

After returning -kehotteet luodaan Spring AOP:ssä pitkälti samalla tavalla kuin before- ja after –kehotteet. Kuvassa 5.12 on luotu kuvan 4.15 kanssa identtinen kehote.

```
<aop:aspect id="ExampleAspect" ref="ExampleAspectImpl">

      <aop:pointcut id="createClone" expression=
            "execution(* cs..MyObject.createClone(..))"/>

      <aop:after-returning pointcut-ref="createClone"
            method="ilmoitaKlooninLempinimi" returning="object"/>

</aop:aspect>
```

Kuva 5.12 After returning -kehote

Kehotteessa viitataan aspektia tukevan pavun metodiin ilmoitaKlooninLempinimi. Returning-argumentti kehotteessa ilmoittaa, että createClone-liitoskohdan palauttama arvo tullaan antamaan viitatulle metodille argumenttina. Näin ilmoitaKlooninLempinimi-metodilla tulee olla object-niminen muuttuja metodin argumenttina. Kuvassa 5.13 on esitelty ilmoitaKlooninLempinimi-metodi, sekä liitoskohdasta palautettuun muuttujaan viittaaminen.

```
public void ilmoitaKlooninLempinimi(Object object) {

      tulosta("Objektin nimi on " + ((MyObject)object).getObjectName()
            + " mutta me kutsumme sitä nimellä klooniobjekti");
}
```

Kuva 5.13 After returning -kehotteen lisäämä toiminnallisuus

After throwing -kehote määritellään sovelluskontekstitiedostossa pitkälti samoin kuin after returning -kehote. Ainoana poikkeuksena on returning -attribuutin korvaaminen throwing-attribuutilla, kuten esimerkissä 5.14 on tehty.

```
<aop:aspect id="ExampleAspect" ref="ExampleAspectImpl">

      <aop:pointcut id="setOperations" expression=
            "execution(* cs..MyObject.set*(..))"/>

      <aop:after-throwing pointcut-ref="setOperations"
            method="tulostaSetOperationException" throwing="soe"/>

</aop:aspect>
```

Kuva 5.14 After throwing -kehote

Kuvassa 5.14 on luotu kuvassa 4.16 esitetyn AspectJ kehotteen kanssa identtinen kehote. Throwing-avainsanassa määritelty soe-niminen muuttuja viittaa liitoskohdan nostamaan poikkeukseen, joka annetaan argumenttina kehotteen toiminnallisuuden toteuttavalle tulostaSetOperationException-metodille. Kuvassa 5.15 on esitetty viitatun ExampleAspect –luokan tulostaSetOperationException-metodi. Vaikkakin esimerkin sisällä noussutta poikkeusta ei käsitellä, on siihen mahdollista viitata metodin sisällä soe-argumenttina saadun muuttujan kautta.

```
public void tulostaSetOperationException(SetOperationException soe) {

            tulosta("Asetettu id -arvo oli virheellinen.");
}
```

Kuva 5.15 After throwing –kehoteen lisäämä toiminnallisuus

Kuvassa 5.16 luodaan kuvan 4.17 kanssa identtinen around-kehote. Kehotteen määrittäminen sovelluksen konfiguraatiotiedostossa tapahtuu kuten before- ja after-kehotteissa.

```
<aop:aspect id="ExampleAspect" ref="ExampleAspectImpl">

      <aop:pointcut id="createClone" expression=
            "execution(* cs..MyObject.createClone(..))"/>

      <aop:around pointcut-ref="createClone"
            method="vaihdaKlooninNimi"/>
</aop:aspect>
```

Kuva 5.16 Around-kehote

Kuvassa 5.16 määritellään, että aina createClone-liitoskohtaan saavuttaessa tullaan suorittamaan vaihdaKlooninNimi-metodi. Around-kehotteen kohdalla suoritettavalta metodilta vaaditaan kuitenkin muita kehotteita enemmän toiminnallisuuksia.

```
public Object vaihdaKlooninNimi(ProceedingJoinPoint pjp)
                                        throws Throwable {

            tulosta("CreateClone-metodia kutsuttu");
            Object o = pjp.proceed();
            tulosta("Objektin alkuperäinen nimi on "
                  + ((MyObject) o).getObjectName()
                  + " mutta muutetaan se Object2:ksi");
            ((MyObject) o).setObjectName("Object2");

            return o;

}
```

Kuva 5.17 Around-kehotteen lisäämä toiminnallisuus

Kuvassa 5.17 on esitetty kuvan 5.16 around-kehotteen toiminnot suorittava Java-metodi. Metodi ottaa argumenttina ProceedingJoinPoint-rajapinnan toteuttavan olion, jonka kautta on mahdollista jatkaa saavutetun liitoskohdan suoritusta proceed-metodin kautta. Kuvassa 5.17 suoritettavat toiminnot ovat identtiset kuvan 4.17 AspectJ-kehotteen kanssa. Ainoana erona on proceed-käskyn kutsuminen argumenttina saadun muuttujan kanssa. Aivan kuten AspectJ:ssäkin, proceed-kutsun pois jättäminen ohittaa saavutetun liitoskohdan suorituksen kokonaan.

5.2.6 Tyyppien väliset deklaraatiot

Spring AOP:n toteutus tyyppien välisille deklaraatioille on huomattavasti AspectJ:tä suppeampi (Johnson & al, 2008). Siinä missä AspectJ tarjoaa menetelmät muuttujien ja metodien lisäämiselle, luokkien periytymishierarkian muuttamiselle, uusien rajapintojen määrittelemiselle sekä implementaation lisäämiselle toiseen luokkaan, mahdollistaa Spring AOP ainoastaan uuden rajapinnan määrittelemisen ja implementoinnin lisäämisen toiseen luokkaan. Tähän käytetään aop-nimiavaruuden tagia <*aop:declare-parents*>.

5.2.7 Esimerkkiaspekti

Kuvissa 5.18, 5.19 ja 5.20 on pyritty luomaan kuvassa 4.23 esitetty aspekti. Aspektin tavoitteena oli poikkileikata liitteen 1 MyObject-ohjelmaa liitteen 3 Main-ohjelman kautta suoritettuna. Kuten kohdan 4.6 esimerkkiaspektissa, myös tässä aspektiohjelmassa lisättäviä ominaisuuksia olivat createClone-metodikutsujen kirjaaminen, createClone-metodissa luodun olion nimen muuttaminen ennen olion palauttamista kutsuvalle luokalle, uuden kentän ja kahden metodin lisääminen sekä get- ja set-alkuisten metodien onnistuneen suorittamisen kirjaaminen. Esimerkkien lähdekoodit löytyvät kokonaisuudessaan liitteestä 3 sekä poikkileikattava ohjelma liitteestä 1.

```
ApplicationContext context =
      new ClassPathXmlApplicationContext("aspectContext.xml");
```

Kuva 5.18 Kontekstin lataaminen Main-luokassa

```
public class Aspect {

      public Object createCloneExecutions(ProceedingJoinPoint pjp)
                                                  throws Throwable {

            tulosta("CreateClone-metodia kutsuttu.");
            Object o = pjp.proceed();
            tulosta("Aspect: Objektin nimi ennen muutosta: "
                  + ((MyObject)o).getObjectName());

            ((MyObject)o).setObjectName("Klooniobjekti");

            return o;

      }

      public void fieldOperations() {

            tulosta
                  ("Get- tai set-metodi suoritettu onnistuneesti.");
      }

}
```

Kuva 5.19 Aspect -luokka

```
<aop:config>

  <aop:aspect id="aspect" ref="Aspect">

      <aop:pointcut id="fieldOperations"
            expression="execution(* cs..MyObject.set*(..)) or
            execution(* cs.joensuu.fi.msorsa.MyObject.get*(..))"/>

      <!--AspectJ:n createCloneCall-liitoskohtamäärittelyä ei ole
      mahdollinen toteuttaa-->

      <aop:pointcut id="createCloneExecutions"
            expression="execution(* cs..MyObject.createClone(..))"/>

      <!--Ominaisuus 1. ei mahdollista toteuttaa-->

      <!--Ominaisuus 2.-->
      <aop:around pointcut-ref="createCloneExecutions"
            method="createCloneExecutions"/>

      <!--Ominaisuus 3. ei mahdollista toteuttaa-->

      <!--Ominaisuus 4. -->
      <aop:after pointcut-ref="fieldOperations"
            method="fieldOperations"/>

  </aop:aspect>
</aop:config>

<bean id="Aspect" class="cs.joensuu.fi.msorsa.Aspect"/>

<bean id="MyObject" class="cs.joensuu.fi.msorsa.MyObject">
      <property name="objectName" value="java.lang.String"/>
      <property name="id" value="0"/>
      <property name="objectValue" value="java.lang.String"/>
</bean>
```

Kuva 5.20 aspectContext.xml

Kuvassa 5.18 on esitelty Main-ohjelmassa suoritettu esimerkkiaspektin kontekstin lataaminen. Kuvassa 5.20 on luotu haluttuja ominaisuuksia toteuttavat kehotteet konfiguraatiotiedostossa, sekä ohjelman suorittamiseksi tarvittavien Java-olioiden papumäärittelyt. Kuvassa 5.19 on esitetty kontekstitiedostossa määriteltyä Aspect-aspektia tukevan pavun toteutus, Aspect-luokka, joka sisältää kaikki aspektin kehotteiden tarvitsemat toiminnallisuudet. Lähdekoodit kokonaisuudessaan löytyvät liitteestä 3.

Kuten kuvan 5.20 kontekstitiedostosta voi huomata, ei kaikkia aspektille asetettuja ominaisuuksia ole mahdollista toteuttaa Spring AOP:llä. Sen rajallisen toteutuksen takia ei poikkileikattavaan objektiin ole mahdollista lisätä uusia muuttujia ja metodeja tai rekisteröidä metodikutsuja. Mutta, kuten luvun alussa mainittiin, ei Spring AOP:n tavoite olekaan olla täydellinen aspektiohjelmointiympäristö. Vertaillessa Spring AOP:n ja AspectJ:n testiaspektien toteutuksia, voidaan kuitenkin jo yksinkertaisista perusmekanismeistä huomata menetelmien erot. Spring AOP:n suppeampi liitoskohtamalli saattaa olla helpompi käyttää, mutta sillä ei ole mahdollista saavuttaa niin laajaa kirjoa ohjelman eri kohdista kuin AspectJ:llä.

5.3 Poikkileikkaaminen annotaatioiden avulla

Spring Framework osaa käyttää myös AspectJ:n annotaatioita poikkileikkaamiseen. Niiden käyttäminen ei kuitenkaan tarkoita, että AspectJ:n punojan käyttäminen olisi tarpeellista, vaan Spring AOP:n mekanismit osaavat huomioida annotaatiot lähdekoodissa ja konfiguroida kehyksen toimintaa näiden annotaatioiden mukaisesti. Annotaatiot lisätään lähdekoodin joukkoon samoin kuin AspectJ:ssä. Jotta aspektien toteuttaminen ohjelmallisesti olisi mahdollista, tulee Spring kuitenkin konfiguroida huomioimaan @AspectJ -tyyli. Tämä tapahtuu lisäämällä Springin konfiguraatiotiedostoon kuvassa 5.21 esitetty rivi.

```
<aop:aspectj-autoproxy/>
```

Kuva 5.21 Annotaatioiden käyttöönotto Spring AOP:ssä

Kuvan 5.21 syntaksi kertoo Spring Frameworkille, että sen tulee etsiä annotaatioita lähdekoodin joukosta (Velde & al., 2008, s.13). Koska annotaatioiden käyttäminen nojaa vahvasti AspectJ:hin, vaatii sen käyttäminen AspectJ:n jar -paketteja, jotka tulee löytyä ohjelman luokkapolusta. Vaadittavat paketit ovat aspetcjweaver.jar ja aspectjrt.jar. Lisäksi luodusta aspektista tulee tehdä normaali papumäärittely ohjelmakontekstiin, joka osoittaa luokkaan joka sisältää @Aspect -annotaation (kuva 5.22) (Johnson & al., 2008).

```
<bean id=”MyObject” class=”cs.joensuu.fi.msorsa.MyObject”>
      <!-- normaaleja pavun konfigurointiominaisuuksia-->
</bean>
```

Kuva 5.22. Aspektin papumäärittely ohjelmakontekstissa

Aspektit voivat sisältää liitoskohtamäärittelyiden, kehotteiden ja koodinesittelyjen lisäksi normaaleja Javan metodeja ja attribuutteja (Johnson & al., 2008).

6. YHTEENVETO

Aspektiohjelmointi on melko uusi ohjelmointiparadigma, jonka pyrkimyksenä on muun muassa parantaa ohjelmistojen modulaarisuutta sekä edistää komponenttien uudelleenkäytettävyyttä tarjoamalla uudenlaisia ohjelmistokomponentteja. Se rakentuu pitkälti olio-ohjelmoinnin periaatteiden päälle täydentäen sitä käsitteillä kuten aspekti, liitoskohta tai punoja. Paradigman periaatteita toteuttavia ohjelmointiympäristöjä löytyy useita erilaisia. Tässä tutkielmassa on tutustuttu paitsi aspektiohjelmoinnin taustoihin ja periaatteisiin, myös kahden aspektiohjelmointiympäristön, AspectJ:n ja Spring AOP:n perusmekanismeihin sekä vertailtu niiden eroavaisuuksia.

Aspektiparadigman tavoitteet ja perusajatukset ovat hyvin selkeitä. Monelle olioparadigmaa tuntevalle esitetyt ongelmat ovat varmasti tuttuja. Tästä huolimatta aspektiohjelmointikielet saattavat olla vaikea oppia. Esimerkiksi AspectJ:n syntaksi saattaa alussa vaikuttaa monimutkaiselta. Aspektiparadigman opettelu vaatii myös uudenlaista ajattelutapaa. Loikka esimerkiksi olio-ohjelmoinnin ajattelutavasta aspektiparadigmaan saattaa tuntua suurelta. Kun on nähnyt vaivan opiskella AOP-paradigman ja siihen perustuvan ohjelmointikielen periaatteet, ei AOP tunnukaan enää niin vaikealta. AspectJ esimerkiksi on mahdollista nähdä pienenä lisäosana Java-ohjelmointikielelle. Jos Java on ennestään tuttu, ei uusien ohjelmointirakenteiden opiskelu ole suuri työ. Spring AOP:n skeemapohjainen poikkileikkaaminen vuorostaan on yksinkertainen, jos tuntee sovelluskehyksen periaatteet.

Vertailtaessa AspectJ:n ja Spring AOP:n aspektitoteutusta ei ole mahdollista nostaa toista etusijalle. AspectJ on monipuolinen aspektiohjelmointikieli, johon löytyy jo pieni joukko apuvälineitä. Sen oppiminen saattaa kuitenkin tuntua työläältä tiettyjen monimutkaisten piirteiden takia. Myös ohjelmistojen kehitysprosessin joudutaan integroimaan vähintään punoja, suuremmissa ohjelmistoissa myös useita muita apuvälineistä. Kuitenkin ohjelmistoissa, joissa on tarvetta laaja-alaiseen ja monipuoliseen poikkileikkaamiseen, AspectJ voi olla oikein käytettynä tehokas apuväline.

Spring AOP:n XML-skeemaan perustuva menetelmä on vuorostaan huomattavasti keveämpi aspektitoteutus. Sen oppiminen on helppoa, jos Spring Frameworkin toimintaperiaate on ennestään tuttu. Jos ohjelmistokehitysprosessissa on ennestään sovelluskehys käytössä, ei AOP -moduulin integroiminen osaksi prosessia ole suurikaan työ. Toteutus ei kuitenkaan ole kattava. Sen poikkileikkaamismahdollisuudet ovat huomattavasti AspectJ:tä suppeammat ja siitä puuttuvat kaikki sovelluskehitysprosessin apuvälineet. Jos poikkileikkaaminen onnistuu Spring -papujen kautta, eikä tarve aspekteille ole kovinkaan monipuolinen voi Spring AOP tarjota tarpeelliset apuvälineet.

AOP-paradigman melko nuoren iän takia siinä on kuitenkin vielä paljon puutteita. Aspektien käytös saattaa joskus olla arvaamatonta. Ne voivat vahingoittaa järjestelmän luotettavuutta ja rikkoa ominaisuuksia, jotka olivat toimivia ennen aspektien lisäämistä. Paradigma kärsiikin tällä hetkellä eniten riittämättömistä ohjelmistokehityksen apuvälineistä. Sen tueksi tulisi kehittää toimivia menetelmiä ja työkaluja, joiden avulla muun muassa poikkileikkaamisen suunnittelu ja mallintaminen on mahdollista. Paradigmalle ei esimerkiksi ole kehitetty vielä yhtenäistä mallinnuskieltä, vaikka useita erilaisia, pääasiassa UML:ään pohjautuvia onkin esitetty. Myöskään aspektien hyödyllisyyttä käytännön ohjelmistokehityksessä ei ole vielä tutkittu riittävästi. Onko esimerkiksi aspektien avulla mahdollista saavuttaa hyötyä pienissä ohjelmissa? Suurissa ohjelmistoissa ne pääsevät varmasti oikeuksiinsa, mutta nykyisillä aspektiapuvälineillä voi aspektien hyödyntäminen aiheuttaa paljon ylimääräistä vaivaa. Onkin vaikea arvioida onko saavutettu hyöty riittävä. Miten aspektien toiminta voidaan suunnitella ja mallintaa ohjelmistokehitysprosessin alkuvaiheessa? Miten niiden toiminta tulisi testata prosessin loppuvaiheessa? Miten ne käyttäytyvät ohjelmiston evoluution aikana? Tutkimustuloksia kaivataankin siitä, miten aspektit toimivat osana erikokoisia ohjelmistoja pitkällä aikavälillä.

Tietyistä puutteista huolimatta aspektiparadigmassa on paljon potentiaalia. Tällä hetkellä sen siirtyminen vakiintuneeksi osaksi ohjelmistokehitystä vaatii kuitenkin paljon työtä monella eri saralla.

Viiteluettelo

Aksit, M. (2004) The 7 C's for Creating Living Software: A Research Perspective for Quality-Oriented Software Engineering, *The Turkish Journal of Electrical Engineering & Computer Sciences*, **12**(2), 61-95

Black, S., Harman, M. (2006) *Aspect-Oriented Software Development: Towards A Philosophical Basis*. Technical Report TR-06-01, Department of Computer Science, King's College London, http://www.dcs.kcl.ac.uk/technical-reports/papers/TR-06-01.pdf (13.5.2008)

Bradley J.T (2003) *An Examination of Aspect-Oriented Programming in Industry.* Technical Report CS-03-108, Department of Computer Science, Colorado State University http://www.cs.colostate.edu/~rta/publications/CS-03-108.pdf (5.6.2008)

Chapman, M. (2006) *Making AspectJ development easier with AJDT.* InfoQ, http://www.infoq.com/articles/aspectj-with-ajdt (15.5.2008)

Clarke, S., Baniassad, E. (2005) *Aspect-Oriented Analysis and Design: The Theme Approach*, Addison-Wesley, Stoughton, Massachusetts

Colyer, A., Clement, A., Harley, G., Webster, M. (2004a) *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools, Chapter 8.* Addison-Wesley Pofessional, http://www.ubookcase.com/book/Addison.Wesley/Eclipse.AspectJ/0321245873/ch08.html (29.5.2008)

Colyer, A., Clement, A., Harley, G., Webster, M. (2004b) *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools, Chapter 9.* Addison-Wesley Professional,

http://www.ubookcase.com/book/Addison.Wesley/Eclipse.AspectJ/0321245873/ch09.html (29.5.2008)

Colyer, A. (2005) *AOP@Work: Introducing AspectJ5*. IBM DeveloperWorks, http://ibm.com/developerworks/java/library/j-aopwork8/ (13.5.2008)

Elrad, T., Filman, R.E., Bader, A. (2001) Aspect-Oriented Programming. *Communications of the ACM,* October 2001/**44**(10), 28- 32

Filman, R.E., Friedman D.P., (2000) *Aspect-Oriented Programming is Quantification and Obliviousness*, RIACS Technical Report

Forgáč, M., Kollár, J. (2007) Static and Dynamic Approaches to Weaving. *5th Slovakian-Hungarian Joint Symposium on Applied Machine Intelligence and Informatics January 25-26 2007,* 201-210

Jacobson, I., Ng, P-W. (2005) *Aspect-Oriented Software Development With Use Cases*, Addison-Wesley, Crawfordsville, Indiana

Johnson, R. (2005) J2EE Development Frameworks, *Computer*, **38**(1), Jan 2005, 107-110

Johnson, R. (2007) *Introduction to the Spring Framework 2.5*. http://www.theserverside.com/tt/articles/article.tss?l=IntrotoSpring25 (13.5.2008)

Johnson, R., Hoeller, J., Arendsen, A., Sampaleanu, C., Harrop, R., Risberg, T., Davison, D., Kopylenko, D., Pollack, M., Templier, T., Vervaet, E., Tung, P., Hale, B., Colyer, A., Lewis, J., Leau, C., Evans, R. (2008) *The Spring Framework - Reference Documentation, Chapter 6*. springframework.org, http://static.springframework.org/spring/docs/2.5.x/reference/aop.html (13.5.2008)

Khatchadourian, R. (2006) *Aspects of AOP: An Exploration of the Aspect-Oriented Paradigm.* Ohio State University, http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=E92546F8E386ABAFE743C94350C2F744?doi=10.1.1.60.1270&rep=rep1&type=pdf

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Videira Lopes, C., Loingtier, J-M., Irwin, J. (1997) Aspect-Oriented Programming, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, June 1997, 220-242

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G. (2001a) An Overview of AspectJ. *Lecture Notes in Computer Science, ECOOP 2001 — Object-Oriented Programming*, Springer Berlin / Heidelberg, 327-354

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G., (2001b) Getting Started With AspectJ, *Communications of the ACM*, October 2001, **44**(10), 59-65

Koskimies, K. (1998) *Pieni oliokirja.* Suomen ATK-Kustannus Oy, Jyväskylä

Krishnan, S. (2005) *Aspect Oriented Programming: Weaving Aspects with AspectJ and AspectWerkz*. http://www.poly.asu.edu/technology/dcst/Projects/04-05/May05/Krishnan,%20Srinivas.pdf (13.5.2008)

Laddad, R. (2002a) *I Want My AOP, Part 1: Separate software concerns with aspect-oriented programming*. JavaWorld.com 18.1.2002, http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html (13.5.2008)

Laddad, R. (2002b) *I Want My AOP. Part 2: Learn AspectJ to better understand aspect-oriented programming*, JavaWorld.com 1.3.2002, http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2.html (13.5.2008)

Laddad, R. (2002c) *I Want My AOP. Part 3: Use AspectJ to modularize crosscutting concerns in real-world problems*, JavaWorld.com 12.4.2002, http://www.javaworld.com/javaworld/jw-04-2002/jw-0412-aspect3.html (13.5.2008)

Laddad, R. (2003) Aspect-Oriented Programming Will Improve Quality, *IEEE Software*, 20(6), 90-92

Laddad, R. (2004) *AspectJ in Action: Practical Aspect-Oriented Programming, Chapter 3: AspectJ Syntax Basics.* Manning Publications, Greenwich, USA

Laddad, R. (2006) *AOP@Work: AOP myths and realities*, IBM DeveloperWorks, http://www.ibm.com/developerworks/java/library/j-aopwork15/ (13.5.2008)

Lafferty, D., Cahill, V. (2003) Language Independent Aspect-Oriented Programming. *Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Anaheim, California, USA , 26-30

Masuhara, H., Endoh, Y., Yonezawa, A. (2006) A Fine-Grained Join Point Model For More Reusable Aspects. *Lecture Notes In Computer Science*, Springer Berlin / Heidelberg, 131-147

Mili, H., Elkharraz, A., Mcheick, H. (2004) *Understanding Separation of Concerns.* Laboratory for research on Technology for E-Commerce (LATECE) Publications, http://www.latece.uqam.ca/publications/mili-kharraz-mcheick.pdf (13.5.2008)

Ossher, H., Tarr, P. (2000) *Multi-Dimensional Separation of Concerns and The Hyperspace Approach.* IBM Research, http://researchweb.watson.ibm.com/hyperspace/Papers/sac2000.pdf (13.5.2008)

Palo Alto Research Center (2003) *The AspectJ Programming Guide*. http://www.eclipse.org/aspectj/doc/released/progguide/ (14.5.2008)

Pawlak, R., Retaillé, J-P., Seinturier, L. (2005) *Foundations of AOP for J2EE Development,* Springer-Verlag, New York

Rajan, H.A. (2007) Case For Explicit Join Point Models For Aspect-Oriented Intermediate Languages. *Proceedings of the 1st workshop on Virtual machines and intermediate languages for emerging modularization mechanisms,* ACM New York, NY, USA, Article No. 4

Sato, Y. (2005) *A Study of Dynamic Weaving for Aspect-Oriented Programming.* http://www.csg.is.titech.ac.jp/paper/yoshiki-phd2006.pdf (13.5.2008)

Steinmann, F. (2006) The Paradoxical Success of Aspect-Oriented Programming, *ACM SIGPLAN Notices,* Proceedings of the 2006 OOPSLA Conference, ACM, New York, 481-497

Sun Microsystems Inc (2008) *Java Programming Language – Annotations*. WWW-sivusto, http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html (13.5.2008)

Sutton Jr., S.M., Rouvellon, I. (2002) Modeling of software concerns in Cosmos. *Proceedings of the 1st international conference on Aspect-oriented software development,* Enschede, Netherlands, 127-133

Sutton Jr., S.M., Rouvellon, I. (2004) *Concern Modeling for Aspect-Oriented Software Development.* IBM Research Publications, http://www.research.ibm.com/AEM/pubs/Cosmos--Chapter_21.pdf (13.5.2008)

Suzuki J., Yamamoto Y. (1999) Extending UML with Aspects: Aspect Support in the Design Phase. *Proceedings of the 3 rd AOP Workshop held in conjunction with ECOOP '99,* http://trese.cs.utwente.nl/aop-ecoop99/papers/suzuki.pdf (5.6.2008)

Tarr, P., Ossher, H. (2001) Workshop on Advanced Separation of Concerns in Software Development. *Proceedings of the 23rd International Conference on Software Engineering***,** Toronto, Ontario, Canada, 778 – 779

The AspectJ Team (2005) *The AspectJ 5 Development Kit Developer's Notebook*. http://eclipse.org/aspectj/doc/released/adk15notebook/ (13.5.2008)

Van de Velde, T., Snyder, B., Dupuis, C., Li, S., Horton, A., Balani, N. (2008) *Beginning Spring Framework 2,* Wiley Publishing Inc, Indianapolis, Indiana

Viega, J., Voas, J. (2000) Can Aspect-Oriented Programming Lead To More Reliable Software?, *IEEE Software,* **17**(6), 19-21

Liite 1. Esimerkeissä poikkileikattava Java-ohjelma

MyObject.java

```
package cs.joensuu.fi.msorsa;


/**
 * MyObject-luokka, joka sisältää kolme muuttujaa sekä niille arvon ase-
tus- ja hakumetodit.
 * @author msorsa
 *
 */

public class MyObject {


/**
 * Objektin sisäiset muuttujat.
 */

private String objectName;
//Id-muuttuja on rajattu niin, että sen arvo saa olla väliltä 0-
//100. Sitä kontrolloidaan arvon asetusmetodin yhteydessä.
private int id;
private String objectValue;


  /**
  * Konstruktori
  */

  public MyObject() {

      super();
  }


  /**
   * Arvon asetusmetodi objectName -muuttujalle.
   * @param newObjectName objektin nimi
   */

  public void setObjectName(String newObjectName) {

      this.objectName = newObjectName;
  }
```

```
/**
 * Arvon hakumetodi objectName -kentälle.
 * @return objektin nimi
 */

public String getObjectName() {

    return this.objectName;
}


/**
 * Arvon asetusmetodi id -kentälle. Tarkistaa onko id väliltä 0-100
 * ja heittää poikkeuksen jos id ei ole sallittu.
 * Poikkeuksen tapauksessa id-kenttä saa arvon -1.
 * @param newId uusi id
 * @throws SetOperationExcpetion jos annettu id ei ole sallittu
 */

public void setId(int newId) throws SetOperationException {

    if(newId > 100 || newId < 0) {
          this.id = -1;
          throw new SetOperationException();
    }

    else {
          this.id = newId;
    }
}


/**
 * Arvon hakumetodi objektin id-numerolle
 * @return objektin id
 */

public int getid() {

    return this.id;
}


/**
 * Arvon asetusmetodi objektin arvo -kentälle.
 * @param newObjectValue uusi objektin arvo
 */

public void setObjectValue(String newObjectValue) {

    this.objectValue = newObjectValue;
}
```

```
  /**
   * Arvon hakumetodi objektin arvolle.
   * @return objektin arvo
   */

  public String getObjectValue() {

      return this.objectValue;
  }


  /**
   *Luo luokasta tehdyn olion kanssa identtisen olion.
   */

  public MyObject createClone() {

      MyObject clone = new MyObject();
      clone.setObjectName(this.objectName);
      clone.setObjectValue(this.objectValue);

      try {
            clone.setId(this.id);
      }catch(SetOperationException soe) {}

      return clone;

  }
}
```

SetOperationException.java

```
package cs.joensuu.fi.msorsa;


/**
 * Oma poikkeustyyppi joka heitetään aina jos id-kenttään asetettu arvo
 * ei ole oikeanlainen.
 * @author msorsa
 */

public class SetOperationException extends Exception {

  public SetOperationException() {

      super();
  }

}
```

Liite 2. AspectJ –esimerkkien lähdekoodit

ExampleAspect.aj

```
package cs.joensuu.fi.msorsa;

import java.util.Date;


/**
 * AspectJ-esimerkeissä käytetyt liitoskohdat ja kehotteet.
 * @author msorsa
 */

public aspect ExampleAspect {

  /**
   * Tyyppien väliset deklaraatiot
   */

  private Date MyObject.changeDate;

  public void MyObject.setChangeDate(Date newDate) {

      changeDate = newDate;
  }

  public Date MyObject.getChangeDate() {

      return changeDate;
  }


  /**
   * Liitoskohtamääritykset
   */

  pointcut allOperations() :
      execution(* cs.joensuu.fi.msorsa.MyObject+.*(..));

  pointcut setOperations() : execution(* cs..MyObject.set*(..));

  pointcut getOperations() :
      execution(* cs.joensuu.fi.msorsa.MyObject.get*(..));

  pointcut fieldOperations() : setOperations() || getOperations();

  pointcut fieldOperations2() :
      execution(* cs..MyObject.set*(..)) ||
      execution(* cs.joensuu.fi.msorsa.MyObject.get*(..));
```

```
    /**
     * Kehotteet
     */

    before() : execution(* cs..MyObject.set*(..)) {

        tulosta("Kenttä asetettu.");
    }

    before() : setOperations() {

        tulosta("Kenttä asetettu.");
    }

    after() returning (MyObject o) :
        execution(MyObject cs..MyObject.createClone(..)) {

        tulosta("Objektin nimi on " + o.getObjectName()
        + " mutta me kutsumme sitä nimellä klooniobjekti");
    }

    after() throwing (SetOperationException se) : setOperations() {

        tulosta("Asetettu id -arvo oli virheellinen.");
    }

    MyObject around() : execution(* cs..MyObject.createClone(..)) {

        tulosta("CreateClone-metodia kutsuttu");
        MyObject o = proceed();
        tulosta("Objektin alkuperäinen nimi on "
              + o.getObjectName() + " mutta muutetaan se Object2:ksi");
        o.setObjectName("Object2");
        o.setChangeDate(new Date());

        return o;
    }


    /**
     * Aspektin sisäinen apumetodi oletustulostimelle tulostamiseen.
     */

    private void tulosta(String line) {

        System.out.println(line);
    }
}
```

Aspect.aj

```
package cs.joensuu.fi.msorsa;

import java.util.Date;


/**
 * Testiaspekti, jossa toteutetaan ohjelmointikielen testausta varten
 * määritellyt poikkileikattavaan ohjelmaan lisättävät ominaisuudet.
 * @author msorsa
 *
 */

public aspect Aspect {


  /**
   * Kehotteet.
   */
  pointcut setOperations() : execution(* cs..MyObject.set*(..));

  pointcut getOperations() :
      execution(* cs.joensuu.fi.msorsa.MyObject.get*(..));

  pointcut fieldOperations() : setOperations() || getOperations();

  pointcut createCloneCalls() :
      call(MyObject cs..MyObject.createClone(..));

  pointcut createCloneExecutions() :
      execution(MyObject cs..MyObject.createClone(..));


  /**
   * Ominaisuus 1. Kaikki luokan MyObject metodin createClone kutsut
   * tulee kirjata oletustulostimelle.
   */
  before() : createCloneCalls() {

      System.out.println("Kutsutaan kloonausmetodia.");
  }


  /**
   * Ominaisuus 2. Ennen kuin luokan MyObject metodi createClone
   * palauttaa klooniobjektin, tulee objektin nimi muuttaa muotoon
   * ”Klooniobjekti”.
   */

  MyObject around() : createCloneExecutions() {

      tulosta("CreateClone-metodia kutsuttu.");
      MyObject o = proceed();
      tulosta("Aspect: Objektin nimi ennen muutosta: "
            + o.getObjectName());
      o.setObjectName("Klooniobjekti");
```

```
        return o;
    }


    /**
     * Ominaisuus 3. Luokkaan MyObject tulee lisätä uusi java.util.Date
     * –arvoinen kenttä sekä kentälle get- ja set –metodit.
     */

    private Date MyObject.date;

    public void MyObject.setDate(Date newDate) {

        date = newDate;
    }

    public Date MyObject.getDate() {

        return date;
    }


    /**
     * Ominaisuus 4. Luokan MyObject get- ja set –alkuisten metodien
     * onnistunut suorittaminen tulee kirjata oletustulostimelle.
     */

    after() : fieldOperations() {

        System.out.println
              ("Get- tai set-metodi suoritettu onnistuneesti.");
    }


    private void tulosta(String line) {

        System.out.println(line);
    }
}
```

Main.java

```
package cs.joensuu.fi.msorsa;


/**
 * Main-luokka. Käytetään MyObject-luokan testaamiseen.
 * @author sorsam
 */

public class Main {

  /**
   * Main-metodi.
   */
  public static void main(String[] args) {

      //luodaan uusi objekti ja asetetaan siihen arvot
      MyObject object1 = new MyObject();
      object1.setObjectName("Object 1");
      object1.setObjectValue("Ensimmäinen myObject -olio.");

      try {
            object1.setId(223);

      }catch(SetOperationException soe) {}

      //haetaan id-kentän arvo
      int value = object1.getid();

      //luodaan objektista klooni
      MyObject object2 = object1.createClone();

      //haetaan objetin arvoja
      object1.getObjectName();
      object1.getid();
      object1.getObjectValue();

      System.out.println("Klooniobjektin nimi on "
            + object2.getObjectName() + ".");

  }

}
```

Liite 3. Spring AOP –esimerkkien lähdekoodit

exampleAspectContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">



<!—luvussa 4 käytetyt esimerkit yhdistettynä yhteen konfiguraatioon -->

<!-- papumäärittelyt -->

<bean id="MyObject" class="cs.joensuu.fi.msorsa.MyObject">
      <property name="objectName" value="java.lang.String"/>
      <property name="id" value="0"/>
      <property name="objectValue" value="java.lang.String"/>
</bean>

<bean id="ExampleAspectImpl" class="cs.joensuu.fi.msorsa.ExampleAspect"/>


<!-- aspektimäärittelyt -->

<aop:config>

  <!-- esimerkkiaspekti -->

  <aop:aspect id="ExampleAspect" ref="ExampleAspectImpl">


      <!-- liitoskohtamäärittelyt -->

      <aop:pointcut id="allOperations" expression=
            "execution(* cs.joensuu.fi.msorsa.MyObject+.*(..))"/>
      <aop:pointcut id="fieldOperations" expression=
            "execution(* cs..MyObject.set*(..)) or
            execution(* cs.joensuu.fi.msorsa.MyObject.get*(..))"/>
      <aop:pointcut id="setOperations" expression=
            "execution(* cs..MyObject.set*(..))"/>
      <aop:pointcut id="createClone" expression=
            "execution(* cs..MyObject.createClone(..))"/>
```

```
      <!-- kehotteet -->

      <aop:before pointcut="execution(* cs..MyObject.set*(..))"
            method="ilmoitaSetSuoritus"/>
      <aop:before pointcut-ref="setOperations"
            method="ilmoitaSetSuoritus"/>
      <aop:after-returning pointcut-ref="createClone"
            method="ilmoitaKlooninLempinimi" returning="object"/>
      <aop:after-throwing pointcut-ref="setOperations"
            method="tulostaSetOperationException" throwing="soe"/>
      <aop:around pointcut-ref="createClone" method="vaihdaKlooninNimi"/>


  </aop:aspect>
</aop:config>
</beans>
```

ExampleAspect.java

```
package cs.joensuu.fi.msorsa;

import org.aspectj.lang.ProceedingJoinPoint;


/**
 * ExampleAspect -kontekstitiedostossa määriteltyjen ExampleAspect –
 * aspektin
 * kehotteiden toiminnot toteuttava luokka.
 * @author msorsa
 *
 */

public class ExampleAspect {


  public void ilmoitaSetSuoritus() {

      tulosta("Kenttä asetettu.");
  }


  public void ilmoitaKlooninLempinimi(Object object) {

      tulosta("Objektin nimi on "
            + ((MyObject)object).getObjectName()
            + " mutta me kutsumme sitä nimellä klooniobjekti");
  }


  public void tulostaSetOperationException(SetOperationException soe)
                                                                    {

      tulosta("Asetettu id -arvo oli virheellinen.");
  }
```

```
public Object vaihdaKlooninNimi(ProceedingJoinPoint pjp)
                                          throws Throwable {

      tulosta("CreateClone-metodia kutsuttu");
      Object o = pjp.proceed();
      tulosta("Objektin alkuperäinen nimi on "
            + ((MyObject) o).getObjectName()
            + " mutta muutetaan se Object2:ksi");
      ((MyObject) o).setObjectName("Object2");

      return o;

  }


  private void tulosta(String line) {

      System.out.println(line);
  }
}
```

Main.java

```
package cs.joensuu.fi.msorsa;

import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.ApplicationContext;
import
org.springframework.context.support.ClassPathXmlApplicationContext;


/**
 * Main-luokka. Käytetään sekä ExampleAspect- että Aspect-aspektien
 * testaamiseen.
 * @author sorsam
 */

public class Main {

  /**
   * Main-metodi
   */

  public static void main(String[] args) {

      /**
       * Käytetään tätä kontekstia kun suoritetaan ExamleAspect.aj.
       * ApplicationContext context =
       * new ClassPathXmlApplicationContext
       * ("exampleAspectContext.xml");
       */

      /**
       * Käytetään tätä konteksia kun suoritetaan testiaspekti
       * Aspect.aj.
       */
```

```
      ApplicationContext context =
            new ClassPathXmlApplicationContext("aspectContext.xml");

      BeanFactory factory = context;

      //luodaan uusi objekti ja asetetaan siihen arvot
      MyObject object1 = (MyObject)factory.getBean("MyObject");
      object1.setObjectName("Object 1");

      object1.setObjectValue("Ensimmäinen myObject -olio.");

      try {
            object1.setId(223);
      }catch(SetOperationException soe) {}

      //haetaan id-kentän arvo
      int value = object1.getid();

      //kutsutaaan createClone-metodia
      MyObject object2 = object1.createClone();

      //testataan getOperations-kehotetta
      object1.getObjectName();
      object1.getid();
      object1.getObjectValue();

      System.out.println
            ("Klooniobjektin nimi on " + object2.getObjectName());


  }

}
```

Aspect.java

```
package cs.joensuu.fi.msorsa;

import org.aspectj.lang.ProceedingJoinPoint;


/**
 * Testiaspekti, jossa toteutetaan ohjelmointikielen testausta varten
 * määritellyt poikkileikattavaan ohjelmaan lisättävät ominaisuudet.
 * @author msorsa
 *
 */

public class Aspect {

  public Aspect() {

      super();
  }
```

```
 /**
   * Ominaisuus 2.
   */

  public Object createCloneExecutions(ProceedingJoinPoint pjp)
                                                        throws Throwable {

      tulosta("CreateClone-metodia kutsuttu.");
      Object o = pjp.proceed();
      tulosta("Aspect: Objektin nimi ennen muutosta: "
            + ((MyObject)o).getObjectName());
      ((MyObject)o).setObjectName("Klooniobjekti");

      return o;

  }


  /**
   * Ominaisuus 4.
   */

  public void fieldOperations() {

      tulosta("Get- tai set-metodi suoritettu onnistuneesti.");
  }


  /**
   * Luokan yksityinen apumetodi
   */

  private void tulosta(String line) {

      System.out.println(line);
  }

}
```

AspectContext.java

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xmlns:context="http://www.springframework.org/schema/context"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop.xsd
      http://www.springframework.org/schema/context
      http://www.springframework.org/schema/context/spring-context.xsd">


<!-- esimerkkiaspektin kontekstitiedosto -->
```

```
<!-- aspektimäärittelyt -->
<aop:config>

  <!--esimerkkiaspekti-->

  <aop:aspect id="aspect" ref="Aspect">


      <!--liitoskohtamäärittelyt-->

      <aop:pointcut id="fieldOperations" expression=
            "execution(* cs..MyObject.set*(..)) or
            execution(* cs.joensuu.fi.msorsa.MyObject.get*(..))"/>

      <!--AspectJ:n createCloneCall-liitoskohtamäärittelyä ei ole
      mahdollinen toteuttaa-->

      <aop:pointcut id="createCloneExecutions" expression=
            "execution(* cs..MyObject.createClone(..))"/>


      <!-- kehotemäärittelyt -->

      <!--Ominaisuus 1. ei mahdollista toteuttaa-->

      <!--Ominaisuus 2.-->
      <aop:around pointcut-ref="createCloneExecutions"
            method="createCloneExecutions"/>

      <!--Ominaisuus 3. ei mahdollista toteuttaa-->

      <!--Ominaisuus 4. -->
      <aop:after pointcut-ref="fieldOperations"
            method="fieldOperations"/>

  </aop:aspect>
</aop:config>

<!-- papumäärittelyt -->

<bean id="Aspect" class="cs.joensuu.fi.msorsa.Aspect"/>

<bean id="MyObject" class="cs.joensuu.fi.msorsa.MyObject">
      <property name="objectName" value="java.lang.String"/>
      <property name="id" value="0"/>
      <property name="objectValue" value="java.lang.String"/>
</bean>

</beans>
```