hyväksymispäivä arvosana

arvostelija

Aspektilähtöinen sovelluskehitys, metaohjelmointimekanismit ja reflektiiviset sovelluspalvelimet Javassa

Tuomas Nurmela

Helsinki 3. marraskuuta 2002
58302304 Reflective Middleware
Seminaari
HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Aspektilähtöinen sovelluskehitys, metaohjelmointimekansimt ja reflektiiviset sovelluspalvelimet Javassa

Tuomas Nurmela

58302304 Reflective Middleware

Seminaari

Tietojenkäsittelytieteen laitos

Helsingin Yliopisto

3.11.2002, 24 sivua + liitteet

Tiivistelmä

Aspektilähtöinen sovelluskehitys, metaohjelmointimekanismit ja reflektiiviset järjestelmät ovat kukin itsenäisiä tutkimusalueita, joiden yhdistäminen voi johtaa käytännönläheisiä ongelmia ratkovien, autonomisten sovelluspalvelinten tulemiseen. Sekä Javan tutkimusyhteisö että avoimen lähdekoodin yhteisö ovat selvittäneet näiden alueiden sovellettavuutta niin Java kieleen kuin J2EE-sovelluspalvelimiin. Tästä huolimatta CORBA-projektit sekä näiden saavutukset ovat herättäneet akateemisissa piireissä aina enemmän mielenkiintoa. Paperi tarkastelee kysymyksiä Javan kannalta.

Paperi keskittyy aluksi tarkastelemaan aspektilähtöistä sovelluskehitystä ja metaohjelmointimekanismeja pääosin itsenäisinä kokonaisuuksina esitellen katsauksia useisiin toteutuksiin. Näkökulma on aspektien osalta ohjelmistokehityksessä, metaohjelmointimekanismien osalta puolestaan tarkastellaan mekanismien toimintaperiaatteita. Tämän jälkeen paperi siirtyy käsittelemään sovelluspalvelimia, erityisesti keskittyen ensin näiden aspektilähtöisen ohjelmoinnin tukeen ja seikkoihin, jotka tulee huomioida aspektilähtöisen sovelluspalvelimieen toteutuksessa. Lopuksi esitellään reflektiivinen sovelluspalvelin, joka tuottaa vikasietoisuutta sähköisen kaupankäynnin kontekstissa. Kahdessa viimeisessä sektiossa esitetään viitteet vastaaviin tutkimuksiin ja standardeihin Java-yhteisön ulkopuolella. sekä johtopäätökset ja viitekehys paperin sisällöstä.

Avainsanat: Aspektilähtöinen ohjelmointi, metaohjelmointimekanismit, reflektiiviset järjestelmät

SISÄLTÖ

1 Johdanto

Komponenttilähtöinen sovelluskehitys (component-based software engineering, CBSE) sisältää tällä hetkellä useita haasteita. Näistä keskeisiä ovat *olio-ohjelmoinnin* (object-oriented programming, OOP) modularisoinnin rajoittuminen, sovelluspalvelinten komponenttiarkkitehtuurien rajoitteet sekä sovelluspalvelinten rajoitteet reflektiivisyyden tuessa.

OOP:ssa modularisoinnin rajoittuminen johtuu siitä, että olion on kapseloitava kaikki näkökulmat, joista sitä voidaan lähestyä. Tämä edellyttää mm. ei-toiminnallisten piirteiden (kuten lokiinkirjoituksen tai debuggauksen) koodin sekoittamisen itse ydinongelmaa käsittelevään koodin. Yhtälailla OOP ei kykene tarjoamaan vaihtoehtoisia näkökulmia itse liiketoimintalogiikkaan vaan on, proseduraalisen ohjelmoinnin kaltaisesti, pohjimmiltaan vain tapa modularisoida sovelluslogiikka kokonaisuudessaan. *Aspektilähtöinen sovelluskehitys ja ohjelmointi* (aspect-oriented software development and programming, AOSD ja AOP) ovat OOP:n ja CBSE:n rinnalla toimivia lähestymistapoja, mahdollistaen monipuolisemman *aihealueiden erottamisen* (seperation of concerns [HL95]), täydentäen OOP:a ja CBSE:a, joskin yhtälailla soveltuen proseduraalisten kielien tukimekanismiksi. AOSD koostuu tällä hetkellä useista eri koulukunnista, joiden puitteissa sen rajauksiin ja sisältöön on (ehkä hieman itseironisestikin) useita eri näkökulmia. Kuitenkaan yksikään AOSD:n lähestymistapa ei edes väitä olevansa täydellinen ratkaisu kaikkiin ongelmiin, vaan ne tuovat mukanaan omia haasteita, mukaanlukien mm. aspektikoodin uudelleenkäytön.

Sovelluspalvelimet ovat CBSE:n myötä tarjonneet *yleispalveluita* (Common Services Middleware) tietoturvasta tietokantakäsittelyyn, automatisoiden samalla esim. säikeistyksen ja tietoliikenneyhteyksien käsittelyn sekä standardoiden sovellusten koonnin ja jakelun. Tämä kokonaisuus on pyrkinyt tarjoamaan sovelluskehittäjille paremmat puitteet toteuttaa liikentoimintalogiikan uudelleenkäyttämistä kuitenkin edellyttäen, että sovellukset mukautuvat tietyn komponenttimallin järjestelmäarkkitehtuuriin ja tämän tarjoamiin ohjelmointirajapintoihin. AOP:n sisällyttäminen sovelluspalvelimiin mahdollistaa kuitenkin yleispalveluiden tuottamisen riippumatta mukautumisesta komponenttimalliin sekä ohjelmoijien itsensä toteuttamien palveluiden sisällyttämisen tuntumattomasti sovellukseen. Yhtälailla sovelluspalvelintoimittajat voivat itsekin refaktoroida tuotteensa toteutusta paremman modularisoinnin saavuttamiseksi.

Sovelluspalvelinten reflektiivisyys, kyky *kuvata ja tulkita* (reify) ja mukautua poikkeamiin edellyttää vaivatonta yleispalveluiden käyttöä. AOSD mahdollistaa tiettyjen yleispalveluiden ulottamisen tuntumattomasti sovellustasolle itsenäisten, reflektiivisten sovelluspalvelimien toimesta. Reflektiivisyys kuitenkin edellyttää myös mekanismeja, joilla sekä kerätä tietoa ympäristöstä että tulkita tätä kuvausta vasten, jotta järjestelmä voi oikeassa tilanteessa käyttää oikeita yleispalveluita ja mahdollisesti ladata uusia yleispalveluita ympäristön muuttuvien tarpeiden mukaan. Lisäksi reflektiivisten järjestelmien

tutkijoiden vaateet autonomisille, itseään hallinnoiville järjestelmille ovat erittäin kattavat, muistuttaen toisinaan ”paint-it-on-the-wall” –ilmiöstä (tilanteesta, jolloin kattavat vaateet halutaan kokonaisuudessaan, mutta sen järkevästä osiin jakamisesta ja näiden priorisoinnista ei ole yhtenäistä näkemystä). Näiden toteutuminen edellyttää - AOSD:n muunkin käytön kanssa - kykyä käyttöympäristökohtaisille toteutuksille: tuotteissa käytettynä edellytetään tehokkuutta, sulautetut järjestelmät edellyttävät pientä lisämuistinkäyttöä, suljettu lähdekoodi puolestaan edellyttää ajonaikaisia mekanismeja aspektin käyttöönottamiseksi. Joskin aspektin esikääntäminen sovellukseen ennen käännöstä vastaa osaan näistä haasteista, tarvitaan *metaohjelmointimekanismeja* (metaprogramming mechanism, runtime meta-object protocols), jotta aspektit kyetään ottamaan käyttöön ajonaikaisesti. AOSD:n periaatteiden mukaisesti nämä on kyettävä tuottamaan sovellukselle, sovelluspalvelimelle ja Javan tapauksessa virtuaalikoneellekin tuntumattomasti. Tämän lisäksi ympäristöön mukautuvat, avoimet reflektiiviset järjestelmät voivat edellyttää kyvykkyyttä vaihtaa metaohjelmointimekanismi toiseen, mikäli ympäristön mukautuminen esim. edellyttää pienintä mahdollista lisärasitetta muun toiminnan yllättävän CPU-kuorman vuoksi, joskin tällaiset vaatimukset kuvaavasti lähestyvät yhtä versiota reflektiivisen tornin ongelmasta (eli halutaan vaihtaa mekanismi, joka vaihtaa mekanismin ad infinitum), minkä vuoksi nämä vaateet eivät ole keskeisimmässä asemassa.

Tutkielma keskittyy ylläolevaan ongelmakenttään Javan näkökulmasta. Paperin rakenne on seuraava: kohdassa 2 tarkastellaan Javan tunnetuinta aspektilähtöistä ohjelmointia mahdollistava sovelluskehystä, Aspect J:tä. Tämän lisäksi esitellään kolme muuta koulukuntaa. AspectJ on kuitenkin keskeisin osuus, koska se toimii vertailukohtana kohdan 3 metaohjelmointimekanismeihin pohjautuviin dynaamisiin aspektilähtöisiin ratkaisuihin, joissa painotus on itse mekanismien tarkastelussa aspektikielen ilmaisuvoiman sijaan. Kohdassa 4 tarkastellaan JAC java sovelluspalvelimen ja JBoss J2EE sovelluspalvelimen aspektitukea. Kohdassa 5 esitellään JBossiin päälle rakennettu vikasietoisuusmekanismia JAGR, jonka tavoitteena on tuottaa autonominen elvytyksen hallinta sähköisen kaupankäynnin sovelluksille. JAGR huomioi kohdealueensa erityispiirteet, jonka puitteissa mekanismi poikkeaa huomattavasti perinteisistä tavoista toteuttaa vikasietoisuutta sovelluspalvelimiin, kuten, replikointiin tukeutuvat ryhmäkommunikointimekanismit [Bir93] tai toistoelpyvät sovellukset [BG02]. Kohdassa 6 määritetään kytköksiä muuhun tutkikmus- ja standardointityöhön. Viimeinen luku sisältää johtopäätöksiä ja viitekehyksen käsiteltyihin asioihin.

2 Aspektilähtöinen sovelluskehitys Javassa

Aspektit mahdollistavat *ei-toiminnallisen* (non-functional) koodin modularisoinnin. Tällä viitataan liiketoimintalogiikan ulkopuolisiin alueisiin, *tukiaspekteihin* (auxilary aspects), jotka jakauvat sekä sovelluskehittäjille kohdistettuihin alueisiin että perinteisesti sovelluspalvelimien sovelluskehyksen tarjoamiin yleispalveluihin. Lisäksi aspektit mahdollistavat sovelluslogiikan *ydinaspektien* (core aspects)

erottelun muusta sovelluslogiikasta. Ydinaspektit kuvaavat jotakin sovelluksen tehtävälle keskeistä toiminnetta, jonka toiminta halutaan erottaa ylläpidon selkeyttämiseksi.

Sovelluskehittäjille tukiaspektit tarjoaa uudenlaisen lähestymistavan alueisiin kuten debuggaus (esim. tarkistetaan onko metodikutsun muuttujien arvot sallituissa rajoissa tai paluuarvot oikeaa muotoa), virhekäsittely (esim. poikkeuskäsittelyssä tapahtuvan lokinkirjoituksen, jolloin kyetään vaivattomammin tuottamaan yhtenäinen muoto virhelokille), sovellustason eheysehtojen käsittely (esim. tarkistaa että osastokoodia käsittelevät muuttujat pitävät yllä muuttujalle ominaista muotoa). Näiden modularisointi mahdollistaa esim. koodin paremman laatutarkastelun sekä helpomman ylläpidon. Sovelluskehittäjien on mahdollista myös helpommin erikoistua erikoisalueisiin, kuten reaaliaikaisuus tai rinnakkaisuuden hallinta, mikäli tätä joudutaan toteuttamaan sovellustasolla.

Sovelluspalvelimille tukiaspektit mahdollistavat sovellusriippumattoman tuen yleispalveluille kuten käyttöoikeushallinta (esim. tarkistetaan käyttäjän roolin oikeudet kyseisen metodin edellyttämiä), persistenttiys (esim. tallentamalla tämän piiriin määritettyjen olioiden muuttujat ennen ja jälkeen tähän kohdistuvia metodikutsuja massamuistiin) tai sovellusprofilointi (esim. kuvaamalla komponenttien metodikutsujen suhteita ja täten muodostamalla verkon sovelluksen metodikutsujen liittymistä toisiinsa). Nämä palvelut voidaan katsoa sijaitsevan toisella *aspektikerroksella* (layered aspect) [NEF01] kuin sovelluskehittäjien tukiaspektit. Yleispalveluiden aspektitoteutukset mahdollistavat sovelluspalvelimen komponenttimallista poikkeavien sovellustoteutusten (esim. POJO) kuorruttamisen lisäpiirteillä, tuottaen kilpailevan sovellusarkkitehtuurivaihtoehdon, kuten luvussa 4 ilmenee.

Ydinaspektit eroavat tukiaspekteista soveltamalla aspekteja jonkin sovelluslogiikan osan kapselointiin. Tsä esimerkkinä toimii käyttöjärjestelmän virtuaalimuistin sivunpuutoskäsittely [CKF+01], jossa käsittely itsessään kyetään kapseloimaan sen sijaan että se sotkeutuu virtuaalitiedostojärjestelmän tai kunkin tiedostojärjestelmän koodin sekaan.

Käytännön näkökulmasta aspekteilla tarkoitetaan sovelluksen suorituspolussa sijaitsevien tiettyjen metodi- tai funktiokutsujen väliin tai tilalle sijoitettavaa suoritettavaa lisäkoodia, joka on erotettuna fyysisesti sovelluksen luokkatiedostoista tai sovellusmoduleista. Aspektin määrityskielen ilmaisuvoima on keskeisessä asemassa määrittäessä mitä toiminteita aspektikieli voi tuottaa. Aspektikieli voi olla joko

- ohjelmointikielen laajennus, jolloin sovelluskehittäjien on helpompi totuttautua sen käyttöön
- metakieli, joka määrittää miten aspektikohtaisia kieliä rakennetaan. Tällöin ilmaisuvoima pyritään takaamaan yllättävien tulevaisuuden tarpeiden varalta
- yksi kaiken kattava aspektikieli, joka pyrkii määrittämään aspektien kaikki käyttömahdollisuudet
- joukko konfiguraatiotiedoston määritysoptioita, jossa ilmaistaan luokkien ominaisuuksia tai käyttäjävaateita, jonka puitteissa aspekteja otetaan käyttöön, soveltuen erityisesti sovelluspalvelinten yleispalveluihin

Tällä hetkellä keskeisiä avoimia kysymyksiä ovat mm. Javan aspektikielen standardointi ja aspektikielten kuvaaminen laajennetulla UML notaatiolla [Kic03][Jak03]. Sovelluskehitysvälineiden tuki on oletettavasti kattavampi jahka java-yhteisö (tai markkinat) on päässyt yhteisymmärrykseen näistä kysymyksistä, joskin esim. yritykset, jotka ajavat tietyn aspektikielen tai työkalun hyväksyntää ovat työkaluissaan jo toteuttaneet tarvittavat tukipiirteet.

AOSD vaalii *minimaalista kytkentää* (minimal coupling) sovelluskoodiin, mahdollistaen aspektikoodin erilisen modularisoinnin. Staattisten metaolioprotokollakielten (MOP) kirjoittaminen sovelluskoodin joukkoon tai sinne lisättävät metatagit eivät ole AOP:a, koska nämä eivät ole sovelluksen lähdekoodille tuntumattomia menettelyitä. Tämä tuntumattomuus ylettyy myös arkkitehtuuriselle tasolle, erottaen aspektit *suunnittelumalleista* (design patterns) [GHJ+95], jotka voivat lähestyä jotakin aspektien ongelma-aluetta kuten rinnakkaisuuden käsittely sovelluksen luokkakaavioon pohjautuvilla muutoksilla. Tämä johtaa toiseen AOP:n periaatteista, joka on aspektin määrittämisen ja käyttöönottamisen ajankohdan erottaminen: aspekti voidaan vaihtoehtoisia määrittää rajapintamäärityksen, luokan suunnittelun, toteutuksen tai toteutuksen jälkeen, mutta sen käyttöönotto on riippuvainen *kutomisen* (weaving) ajankohdasta. Kutomisella tarkoitetaan aspektien lisätoiminnallisuuden sijoittamista sovelluskoodin suorituspolun väliin. Näiden periaatteiden lisäksi on useita lisäperiaatteita ja jaoitteluita, jotka vaihtelevat kirjoittajan ja koulukunnan mukaan [HL95] [KLM+97][FF99].

Kutominen voi javassa tapahtua käännösaikaisesti, latausaikaisesti tai suoritusaikaisesti. Kaksi jälkimmäistä, dynaamista tapaa edellyttävät luvussa 3 tarkasteltavien metaohjelmointimekanismien käyttöä. Staattinen, käännöksenaikainen toteutus omaa tiettyjä etuja dynaamisiin mekanismeihin nähden: sovellukseen lisätty koodi tarkistetaan käännösvaiheessa, joten ajonaikaisilta virheiltä vältytään ja optimoija voi optimoida aspektikoodin tavanomaiseen tapaan. Staattinen kudonta tukee alkuperäistä ajatus kääntäjästä erillisestä ”tyhmästä” jäsentimestä ja koodigeneraattorista, jonka olennainen tehtävä on ainoastaan tulkata aspektikielen syntaksia, muodostaa verkko, jonka perusteella voi määrittää mihin osiin sovelluskoodia aspektikoodi on sijoitettava ja toteutaa tämä kooditransformaatio, hieman analogisesti esikääntäjiin nähden.

Javalle on ilmaantunut viimeaikoina useita sovelluskehyksiä AOSD:n tukemiseksi, mm. Nannings [Nan03], AspectWerkz [Asp03] sekä jAdvice [Lee03] [Lee03b]. Sektion seuraavat osat kuitenkin pitäytyvät käsittelemään lähestymistapoja, jotka kuvaavat itse AOSD:a erilaisista näkökulmista, tuoden jotain uutta myös sovelluskehitystyöhön pelkän ohjelmointinäkökulman sijaan. Lyhyet esittelyt kattavat AspectJ:n, vanhimman Java AOP-toteutuksen, komponenttilähtöistä uudelleenkäyttöä painottavan JasCon, subjektilähtöisen sovelluskehityksen jälkeläisen Hyper/J:n sekä adaptiivisen sovelluskehityksen jälkeläisen, DJ sovelluskehyksen.

2.1 AspectJ

Xerox PARC:ssa kehitetyn AspectJ:n taustalta löytyy *metaobjektiprotokollien* (meta-object protocol, MOP) [Kic91] parista tunnettuja hahmoja: mm. Gregor Kiczales, kehitysryhmän vetäjä, oli osana määrittämässä LISPin CLOS MOP-laajennusta ja toteuttamassa tämän referenssi-implementaatiota. Aspektit ovatkin kehitelmä yhdistäen MOP:n ja *avoimen implementaation* (open implementation) [Kic96] piirteitä ajatukseen vaihtoehtoisesta tavasta modularisoida mm. näiden lähestymistapojen koodi [Kic03].

AspectJ on Javan laajennuksiin mukautuva aspektikieli, joka määrittää y*hdyspistemallin* (Join Point Model) mukaisen aspektien määrityksen. Malli on ilmaisuvoimaltaan tällä hetkellä joustavin ja kattavin. Malli perustuu yhdyspisteisiin (Join Point), jotka määrittävät mihin yksittäiseen sovelluksen metodiin, luokkaan, kenttään tai metodin suorituspolun kohtaan aspektikoodi vaikuttaa. AspectJ käyttää *leikkauspisteitä* (Point Cut), jotka määrittävät aspektin vaikutuksen yhdessä tai useammassa yhdyspisteessä. *Vihjeet* (Advice) puolestaan sisältävät leikkauspisteessä suoritettavan aspektikoodin sekä määrittävät, toteutetaanko aspektikoodi ennen, jälkeen vai leikkauspisteen metodien tilalla. Esimerkkinä tästä on liitteessä 1 HelloWorld-toteutus, jossa metodin suoritusta ennen ja jälkeen halutaan tehdä ylimääräisiä metodikutsuja sekä korvata toisen metodin suoritus [Lad02].

AspectJ sisältää näiden perusmääritteiden lisäksi useita monimutkaisempia aspektirakenteita, joita ovat esimerkiksi [Lad02]:

- suorituspolun ymmärtäminen cflow-rakenteella, jonka puitteissa voidaan määrittää yhdyspiste metodikutsuun, mikäli tämän kutsu on seurausta jonkin toisen metodin kutsusta. Näin ollen voidaan ymmärtää perintäsuhteita ja esim. välttää kaksinkertainen aspektin suorittaminen, joka voisi tuottaa virheellisen aspektin toiminnan.
- yhdyspisteiden kesken määritettävät loogiset operaatiot, joiden totuusarvo ratkaisee aspektin kutomisen
- yhdyspisteen määrityksessä regexp:llä, lisäten huomattavasti ilmaisuvoimaa. Toteutukseltaan AspectJ:n kutoja ei tue tätä aivan täydellisesti.

Muita piirteitä ovat mm. reflektion käyttäminen aspektiin, jolloin aspektin tietoja voidaan esim. tulostaa debug-koodissa sekä *oletusarvoinen poikkeuskäsittely* (soft exception handling), joka mahdollistaa poikkeuskäsittelijöiden määrittämisen leikkauspisteen metodeille, mikäli mitään poikkeuskäsittelyä ei ole määritettynä.

AspectJ:n aspektien modularisointia ei ole erityisemmin tuettu, uudelleenkäytettävyys perustuu täysin perintään ja onkin tässä suhteessa jäljessä myöhemmin julkaistuihin järjestelmiin nähden. Uudelleenkäytön edistämiseksi on määritetty menettelyjä [HU01], mutta nämä ovat vaivaloisia.

2.2 JasCo

JasCo:n [SVJ+02] lähtökohta pohjautuu ajatukseen ns. aspektikomponenteista [LLM99], jotka tukevat uudelleenkäyttöä. JasCo:n tuoma lisäpanos on AspectJ:n tasoisen ilmaisuvoiman omaava javapohjainen aspektikieli. Lisäksi kutomiseen on tuotu ylimääräinen välikerros, koska JasCon suunnittelijoiden mukaan ei ole realistista olettaa, että sovelluksen lähdekoodi olisi käytettävissä: kolmennen osapuolen kirjastot ovat tyypillisesti bytecodeksi käännettyjä luokkia.

JasCo tukee JavaBeanien komponenttimallia. JasCon omat komponentit, AspectBeanit, paketoivat sekä itse aspektin toteuttavan modularisoitavan koodin sekä *koukut* (Hook), jotka yhdistävät aspektin suorituskohdan (AspectJ:n vihje) sekä epäsuorasti leikkauspisteen. Leikkauspiste määritetään suhteessa erilliseen yhdistäjään (Connector). Yhdistäjä puolestaan määrittää sekä sovelluksen leikkauspisteen että AspectBeanin ja siitä suoritettavan koukun, näin yhdistäen sovelluksen leikkauspisteen suoritettavaan aspektikoodiin.

AspectBeanit käännetään erilliselle kääntäjätyökalulla tavallisiksi JavaBeaneiksi. Yhdistäjien käyttöönotto puolestaan edellyttää leikkauspisteiden *instrumentointia* (instrumentation), jolla tässä yhteydessä tarkoitetaan leikkauspisteiden kohteiden (metodien) muuttamista niin, että JasCo on tietoinen metodin käytöstä ja pääsee vaikuttamaan käsittelyyn. Tämä tehdään staattisesti kääntämällä bytecode Jasmin assembler-kieleksi erillisellä BCEL-kirjastolla [BCE02], instrumentoimalla jokainen metodi kutsumaan erillistä yhdistäjien rekisteriä (Connector Register, CR) ja kääntämällä sovellus takaisin bytecodeksi Jasmin assemblerilla [Mey97]. Yhdistäjät ovat täten erotettuna sekä sovelluksesta että AspectBean –komponenteista.

Ajonaikana yhdistäjän lataaminen tuottaa rekisteröinnin erillisessä prosessissa toimivaan CR:iin. Jokaisessa intrumentointipisteessä (metodikutsussa) tehdään instrumentointikoodin myötä kutsu CR:iin, jossa tarkistetaan onko sovelluksen metodille määritetty yhdistäjiä. Mikäli on, näiden koukun mukaiset metodit suoritetaan ennen kontrollin palautusta. Täten, mikäli koko AspectJ:n ilmaisuvoima halutaan käyttöön, on huomattava määrä koodista instrumentoiva, tehden toteutuksesta erittäin raskaan. Toisaakta malli mahdollistaa uudelleenkäytön lisäksi dynaamisen yhdistäjien rekisteröinnin ja AspectBeanien vaihtamisen sovelluksen ajonaikaisesti, DynamicTAO CORBA-sovelluspalvelimen kaltaisesti [KCB+02]. Liitteessä 2 on esimerkinomaisesti sovellettu käyttöoikeuksien hallintatamekanismia liitteen 1 HelloWorld-luokkaan.

Vaikkakin JasCo pyrkii mukauttamaan kudonnan huomioimaan tosimaailman ongelmat, on tilanteita, jolloin kudontaa ei voida tai sitä ei tulisi suorittaa. Näitä ovat mm. tietyt lähdekoodin palvelun laatutakeet (esim. muistinkäytön maksimimäärätae, metodien suoritusaika), digitaalisen allekirjoitetun sisältävät tiedostot sekä salatut tiedostot.

2.3 Hyper/J

Hyper/J:n [TOH+99] [OT01] taustalta löytyy subjektilähtöinen ohjelmointi (subject-oriented programming, SOP) [HO93], jota 1990-luvun puolivälissä ponnekkaasti tuettiin erityisesti IBM:n tutkijoiden toimesta. SOP:n puitteissa aspektilähtöisyys tarkoittaa kykyä suunnitella itse sovellus niin että myös sen sovelluslogiikkaa voidaan vaivatta modularisoida eri näkökulmien mukaan. SOP:n mukaan tämä ei ole mahdollista proseduraalisessa ohjelmoinnissa tai olio-ohjelmoinnissa, koska molemmat ovat modularisointitekniikoita, jotka aina nostavat esille jonkin näkökulman, mahdollistamatta moduulin sisäisten näkökulmien erottelua, luoden näin ollen ns. *dominoivan jaoittelun* (tyranny of dominant decomposition). Kuitenkin näkökulmat edellyttävät erottelua: naivina esimerkkinä kirja-luokan keskeisimmät ominaisuudet ja sidosluokat ovat markkinointiyksikölle tekijä, kustantaja ja omakustannehinta siinä missä tuotannosta vastaava yksikkö on kiinnostunut kansimateriaalista, sivumääristä, sivunpaksuudesta, painosta jne. Täten luokat tulisi kyetä kokoamaan itse asiassa pienemmistä kokonaisuuksista (muutamat kielet tukevatkin ns. mixin-luokkia, jotka eivät ole kokonaisia luokkia vaan luokan osia, jotka kootaan kokonaisen luokan muodostamiseksi). Kukin näistä yhdistettäisiin sovellukseen ns. *vastaavuussääntöjen* (correspondense rules) avulla, jotka määrittäisivät suhteet luokkien, metodien ja kenttien tasolla, riippuen granulariteetista sekä *yhdistelmäsääntöjen* (combination rules) avulla, jotka kuvaavat eksiplisiittisesti kahden luokan väliset suhteet. SOP:n perinteiseksi ongelmaksi tulee kuitenkin mahdolliset ristiriitatilanteet yhdistelyssä sekä näiden ratkaiseminen.

Poiketen AspectJ:stä Hyper/J on työkalu. Hyper/J käyttää käsitettä *aihealueen näkökulmat* (dimensions of concern) erottaakseen *aihepiiririippuvaisen* (domain-dependent), luokkakohtaisen, piirrekohtaiset (esim. palkanlaskenta, henkilöstö), työvaihekohtaisen (esim. testaus) sekä ei-funktionaalisen (esim. tietoturva, suorituskyky) aspektin itse luokan toteutuksesta. Hyper/J toteuttaa tämän muodostamalla kustakin luokan aspektista oman kokonaisuuden, jota kutsutaan *hyperviipaleeksi* (hyperslice). Hyperviipaleet ryhmittyvät aspektikohtaisesti joukoksi kokonaisia tai osittaisia luokkakuvauksia sovelluksen ohjelmointikielellä, siitä huolimatta ettei Java tue osittaisia luokkakuvauksia. Rajoitteen sivuttamiseksi hyperviipaleet kootaan niiden suhteet kuvaavan *koontisäännön* (combination rule) kanssa *hypermoduliin* (hypermodule), joka muodostaa Hyper/J sovelluskehyksen ja ydinsovelluksen kanssa kääntyvän kokonaisuuden Javalla. *Hyperavaruus* (hyperspace) on suunnitteluvaihetta tukeva moniulotteinen malli, jossa kunkin näkökulman (aspektin) edustajat muodostavat oman akselin, X-akselin määrittäessä luokat, joilla näkökulmaa esiintyy.

Ivar Jakobsenin näkemyksen [Jak03] mukaan Hyper/J mahdollistaa lopultakin käyttötapauslähtöisen sovelluskehityksen [JCJ+94]: Riippumattomat sovelluksen käyttötapaukset muodostavat sovelluksen perustan tavallisen käytännön mukaisesti. Mikäli käyttötapausta on *laajennettava* (extend), luodaan

laajennuksesta erillinen hyperviipale. Liitteessä 3 on esimerkinomaisesti muodostettu yleiskuvaus puhelun signaalinkäsittelystä ja siihen tulevista laajennuksista.

2.4 DJ

DJ:n [OL01] taustalta löytyy adaptiivinen ohjelmointi (adaptive programming, AD) [LS93], joka painottaa kykyä tuottaa vaivattomasti muutoksia olioiden välisiin metodikutsuketjuihin ymmärtämällä vaikutuksen kokonaisuuteen. Tarve pohjautuu ns. *Demeterin lakiin* (Law of Demeter), joka ehdottaa että luokat tulisi toteuttaa mahdollisimman itsenäisiksi yksiköiksi. Täten ne kutsuisivat vain omia metodejaan, luomiaan objekteja tai suoraan riippuvaisia komponentteja, eivätkä käyttäisi muuta kuin parametreina tullutta tietoa, jotta toiminta ei rikkoutuisi muiden luokkien muutoksen yhteydessä [HT00].

Ajatusmallin mukaisesti adaptiivinen ohjelmointi käyttää polkuja (kärjistetysti yksisuuntainen verkko, jonka solmut koostuvat luokista ja kaaret kuvaavat luokkien välisiä (metodikutsujen) suhteita polun kulkiessa kutsun suorittavasta lähdeoliosta kohteeseen), joiden perusteella määritetään muutostarpeet ja perinteisessä mielessä sovelletaan staattista koodigenerointia. Verrattuna AspectJ:n lähtökohta on toinen, sillä tavoitteena on mahdollistaa Demeterin lain kaltaisen lähestymistavan tukemalla luokkayhteyksiin perustuvaa muutosta ja suunnittelua, jonka myötä ylläpito on helpompaa. Sovelluskehys mahdollistaa turvallisen *polkusiirtymän* (path travelsal) tiettyyn olion erillisellä sovelluskehyksen metodilla. Tämän jälkeen polkusiirtymän tehnyt *vieraileva luokka* (visitor) voi turvallisesti kutsua metodia toisella kehyksen tarjoamalla metodilla. DJ erikoisuus aiempiin adaptiivisen ohjelmoinnin toteutuksiin nähden on Reflection API:n käyttö (kts. 3.4), jonka puitteissa se ajonaikaisesti määrittää luokkien suhteet ja metodit (pakottamalla kaikkien luokkapolun luokkien latauksen).

3 Metaohjelmointimekanismit Javassa

Metaohjelmointimekanismit kattavat kaikki lähestymistavat, joiden myötä voidaan ajonaikaisesti sovellukselle tuntumattomasti suorittaa sovellussuorituksesta riippuvaa koodia. Tällaisiin mahdollisuuden tarjoavat perinteisesti mm. käyttöjärjestelmissä esiintyvät rajapinnat (esim. /proc-pseudotiedostojärjestelmä tai Linux Netfilter Hookit) tai systeemikutsujen uudelleentoteutus jaettujen kirjastojen latausjärjestystä hyödyntämällä. Olio-ohjelmoinnin tapauksessa periyttämisellä tehty metodin *kääriminen* (wrapping) on myös mahdollisesti luettavissa yhdeksi mekanismiksi, joskaan ei ole sellaisenaan tuntumattomasti toteuttettavissa.

Metaohjelmointimekanismit eroavat erityisesti ylimääräisen suorituskykyrasitteen sekä mahdollisten toimintaa rajaavien tekijöiden puitteissa (esim. voidaan ainoastaan lukea muuttuja-arvoja tai manipulaatio kohdistuu ainoastaan esim. etämetodikutsuihin). Täten mekanismia valittaessa oleellista on tuntea vaatimukset, jotka sen täytyy täyttää.

Perinteisten tapojen rinnalla on muistettava, että niin Javan virtuaalikone (Java Virtual Machine, JVM) kuin dotNET arkkitehtuurin virtuaalikone (Common Language Runtime, CLR) mahdollistavat ylimääräisen abstraktiokerroksen hyväksikäytön. Täten esim. Javan bytecode-manipulaatio on mahdollista, kuten JasCon yhteydessä todettiin, joskin JasCo tukeutui staattiseen kudontaan. Javan ympäristössä koodin siirrettävyys on periaatteellinen edellytys eli metaohjelmointimekanismi ei voi vaatia esimerkiksi Java virtuaalikoneen erityistoteutusta ellei tämä ole standardi kaikissa.

Java on tukenut metaohjelmointimekanismeja standarditasolla tarjoamalla JDK 1.1:ssä Reflection API:n, JDK 1.2:ssa JVM Debugger Interfacen, JDK 1.3:ssa Dynamic Proxy luokat ja JDK 1.4:ssä Java Debugger arkkitehtuurin. Kuitenkin näiden ajonaikaisten mekanismien lisäksi JVM:n luokkalataajat mahdollistavat myös latausaikaisten luokkien manipuloinnin esimerkiksi muiden metaohjelmointimekanismien tukemiseksi.

3.1 Luokkalataajat

Java *luokkalataajat* (class loader, CL) [BK02] [Joh03] [Pet02] [Sos03] muodostavat Javassa vanhempi/lapsi –suhteen CL:n ladatessa toinen CL. Lapsi-CL tarkistaa aina vanhemmaltaan, josko tällä olisi luokka ladattuna ennenkuin se itse lataa luokan Vanhemmat eivät puolestaan näe lastensa lataamia luokkia. Hierarkia juurena toimii JVM:n boostrap CL, joka lataa java.lang.Object-luokan.

J2EE-standardi kuitenkin rajoittaa luokkien näkyvyyttä, koska mikäli esim. sovelluspalvelin käyttäisi samaa kirjastoa lokiinkirjoitukseen kuin sovellus, mutta kirjastojen versiot poikkeavat toisistaan, voisi tämä johtaa yllättäviin suorituksenaikaisiin poikkeuksiin. Sovelluspalvelintoimittajat voivat suojautua tältä toteuttamalla esim. ensimmäisen CL:n ja sovelluspalvelimen CL:n väliin väli-CL:n, joka sallii lataukset ainoastaan JDK:n luokkiin (varmistaen ettei perimmäiseltä luokkalataajalta kysytä muuta) sekä eristää sovelluspalvelimen CL sovelluksen CL:stä vastaavalla väli-CL:llä, joka sallii kutsut vain sovelluspalvelimen ja JDK:n luokkiin. Sovelluspalvelimen CL on erotettava sovelluksen CL:stä myös dynaamisen sovelluksen latauksen mahdollistamiseksi. Orbix käyttää tällaisesta suojaavasta CL-toteutusta nimitystä palomuuriluokkataus [BK02]. Kuitenkin sovellusten kesken, mikäli sovelluskehittäjä itse huolehtii esim. kolmannen osapuolen kirjastojen versioinnista, olisi suotavaa kyetä määrittämään tällaisia suhteita ajoaikaisen muistinkäytön pienentämiseksi, mikäli sovelluksen koonti määrittää pakettien suhteet. Tuki yhteiskäytölle on kuitenkin sovelluspalvelinriippuvaista.

Puhdas luokkalataajalähtöinen aspektien kudonta edellyttää bytecode-manipulaatiota, jotta aspektikoodi saadaan kudottua käännettyyn luokkaan, kuten JasCon yhteydessä esitettiin. JasCo myös osoitti sen, että dynaamisen AOP:n toteuttamiseksi bytecode-manipulaatiolla, ei leikkauspisteisiin sinällään kudota aspektin koodia vaan ainoastaan instrumentointi. Vaihtoehtoisesti staattisen kudonnan tasoisen ratkaisun saa kutomalla aspektin suoraan luokkaan, mutta tämän puitteissa ei saavuteta mitään verrattuna kääntämisen yhteydessä tapahtuvaan kutomiseen.

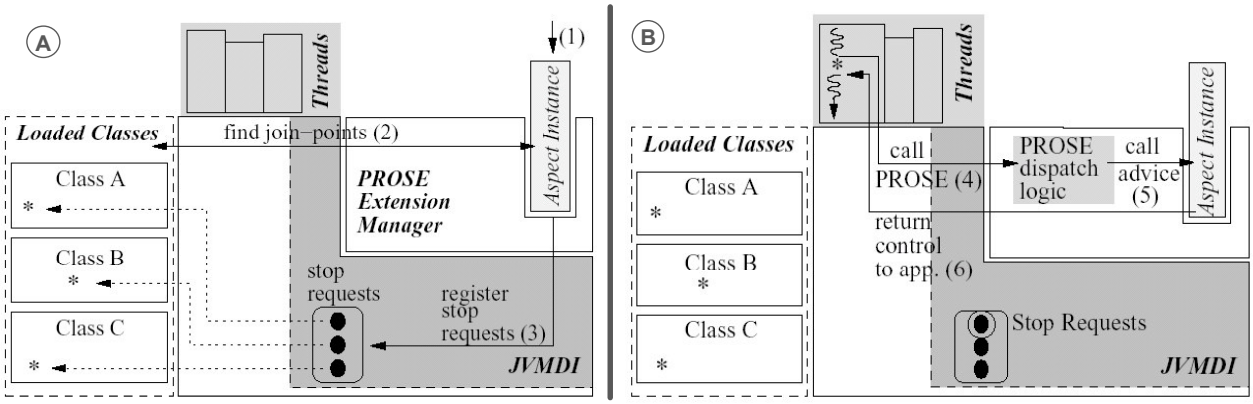
Bytecode-manipulaatio voidaan toteuttaa mm. sovelluspalvelintoimittajakohtaisilla ei-siirrettävillä mekanismeilla (esim. Weblogic esikääntäjä [Lee03b]) tai avoimilla bytecode-manipulaattorikirjastoilla, kuten aiemmin mainitulla BCEL:llä Jasmin assembleriksi käyttämälä erityisen suosittua Javassist-kirjastoa [Chi03]. Javassist (Java Programming Assistant) on MOP-tutkijayhteisössä aktiivisesti työskennelleen professori Shigeru Chiban luokkakirjasto, joka mahdollistaa bytecode-manupulaation kahdella eritasoisen abstraktion toteuttavalla API:lla: Bytecode API tarjoaa suoran bytecodella tehtävän menipulaation, Source API puolestaan mahdollistaa luokkatiedostojen editoinnin ilmaisultaan puhtaasti Javan mukaisella kielellä, edellyttämättä bytecode-osaamista. Täten Javalla kirjoitettu koodi voidaan lisätä luokkaan latauksen yhteydessä, Javassitin huolehtiessa tämän muuttamisesta bytecodeksi ja bytecoden sijoittamisesta sovelluskoodin sekaan joko instrumentoinnin tai kudonnan toteuttamiseksi.

Kirjastojen suhteellinen helppokäyttöisyys onkin ollut yksi pääsyistä Java AOP-sovelluskehysten kasvavaan määrään.

3.2 Java Virtual Machine Debugger Interface (JVMDI)

Java Debugger Interface (JVMDI) [Sun99]on matalan tason rajapinta debuggeriin, joka tuli JDK 1.2:ssa tukemaan debugger-työkalujen luontia tuottamalla palvelurajapinnat, joita JVM tukee. JVMDI:n käyttö edellyttää että sovellus on käynnistetty debugger-tilassa, jolloin JVM:llä on mahdollista JVMDI-rajapinnan kautta käsitellä sovelluksen toimintaa. Debugger-tilassa sovelluksen ajaminen tuottaa keskimäärin alle kymmenen prosentin lisäkuorman palvelimelle.

JVMDI:tä hyödyntävän Prosen [PGA01] alkuperäinen tavoitte on mahdollistaa hot fixien toteuttaminen sekä adaptaatio mobiiliympäristöissä. Prototyypin ajatus on osoittaa, että virtuaalikone tarvitsisi JVMDI:n kaltaisen aspekteja tukevan rajapinnan, JVMAI:n (Java Virtual Machine Aspect Interface), jonka puitteissa aspekteja voidaan kutoa sovellus- ja aspektikieliriippumattomasti.



Kuva 1: Prose arkkitehtuuri ja toiminta: ajonaikainen instrumentointi (A) ja kudonta (B) [PGA01]

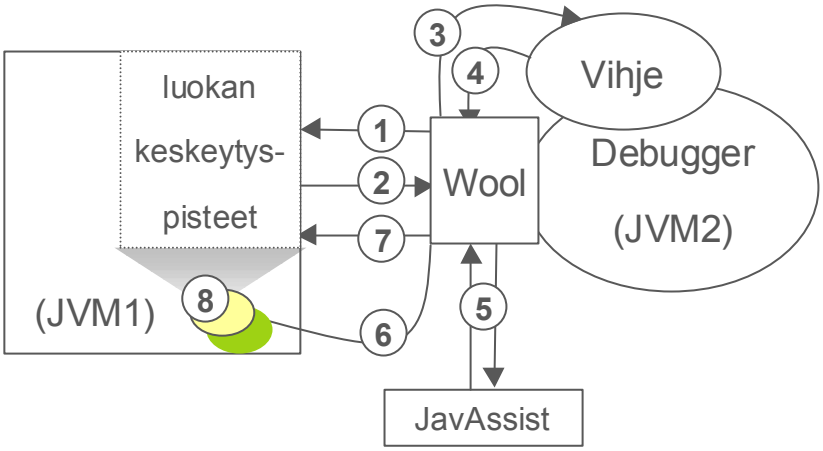
Prosen käsittely jakautuu kahteen vaiheeseen, luokan ajonaikaiseen instrumentointiin ja kudontaan. Instrumentoinnissa Prosen erillinen laajennushallinnoijan (Extension Manager, EM), joka on loogisella tasolla JVM:n yhteyteen liitettävä lisämekanismi, tutkii läpi kaikki aspektimääritykset näiden käynnistyksen yhteydessä (1), etsien yhdyspisteet (2). JVMAI ajatuksen mukaisesti tämä tapahtuisi virtuaalikoneen tukemana, mutta tässä tapauksessa EM tuottaa toiminteen. Jokaista yhdyspistettä kohden EM rekisteröi JVMDI:n kautta keskeytyspisteen (3). Suorituksen aikainen kudonta alkaa keskeytyspisteessä JVMDI:n siirtäessä suorituksenhallinta Proselle (4), joka huolehtii sen vihjeen suorittamisesta, jolle kyseinen yhdyspiste oli rekisteröitynä (5). Vihjeen suorituksen jälkeen Prose palauttaa suorituksenhallinnan sovellukselle (6).

Prosen toistuva keskeytyspisteiden käyttö on hyvin raskasta, erityisesti lyhyiden metodikäsittelyjen puitteissa. Prosen onkin vain prototyyppitoteutus.

3.3 Java Platform Debugger Architecture (JPDA)

Java Platform Debugger Architecture (JPDA) [Sun02] on JDK 1.4:n myötä uudelleenkoostettu kokonaisarkkitehtuuri, joka kokoaa joukon debbugger-ohjelmointirajapintoja yhden kerrosarkkitehtuurin alaisuuteen sekä tarkemmin määrittää kunkin roolin.

Wool [CST03] on JavAssistin luoneen Shigeru Chiban luokkalataajaa sekä JPDA:n uutta HotSwap-ominaisuutta käyttävä luokkakirjasto. Toteutus ei määritä AOP-kieltä tai toteutusta vaan toimii tällaisen rakennusosana. Koska Wool on sovelluksesta erillisessä prosessi pyörivä debugger, ei sitä tarvitse ladata sovelluksen käynnistyksen aikana, mutta kuten Prose, sovellus on käynnistettävä debugger-moodissa.



Kuva 2: Wool arkkitehtuuri ja toiminta

Verrattuna Proseen, Woolin merkittävä lisäarvo on vain kerran toteutettavissa keskeytyskäsittelyssä per kudottava aspekti, koska keskeytyskäsittely on hidasta. Kuva 2 käsittää tämän toiminnan kuvauksen: Woolin käynnistyessä määritettyihin yhdyspisteisiin rekisteröidään JPDA:n JDI API:n kautta keskeytyspisteet (1). Keskeytyspisteessä JPDA siirtää suorituksen Woolille yhdyspisteen käsittävän

luokan sarjallistetun objektikontekstin kanssa (2). Wool määrittää objektikontekstista yhdyspisteen kontekstin ja siirtää tämän vihjeelle suoritettavaksi (3). Vihje suoritetaan ja ohjaus siirtyy takaisin Woolillle (4). Wool käyttää JavAssistia ja kutoo luokkaan vihjeen sisällön (5). Vihjeen sisältämä luokka uudelleenladataan JPDA:n HotSwap-mekanismilla (6) ja keskeytyspiste poistetaan (7). Seuraavalla kerralla käsittely sisältää jo kudotun aspektin, joten mitään keskytyskäsittelyä tai prosessin vuoronnusta ei tarvita (8).

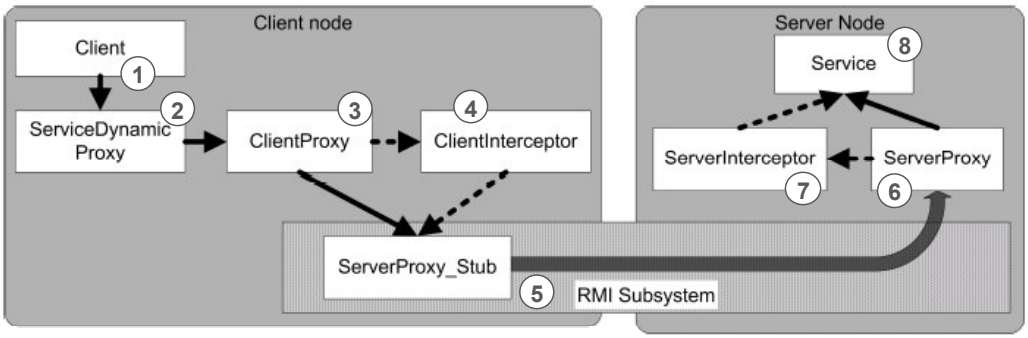
Sovellusten näkökulmasta Woolia ei voi käyttää sellaisenaan, koska on mahdollista että luokkalataajat eivät esim. kykene osoittamaan keskeytyksiä oikeaan sovellukseen vaan samoja kirjastoja useissa sovelluksissa käyttävä järjestelmä kutoo aspektit kaikkiin kirjaston metodeja käyttäviin sovelluksiin, mahdollisesti myös sovelluspalvelimeen.

3.4 Reflection API

Java Reflection API [Sun98] on JDK versiosta 1.1 asti tuottanut mahdollisuuden tarkastella ladattujen luokkien metodeja, kenttiä ja tarvittaessa suorittaa metodeja, mahdollistaen ajonaikaisen instrospektion. Mahdollisesti pitkän ikänsä vuoksi menettelyn toimintaan on tarttunut joukko uskomuksia, kuten hitaus [Sos03] ja monimutkaisuus, joista sinällään voi jatkossa olla vaikea päästä eroon vaikka JVM implementaatiot kehittyisivätkin tuottamaan sen tehokkaasti [Joh03: s. 132-138]. Keskeistä on, että reflektoinnin kautta suoritettava metodikutsu itsessäänkin sisältää käsittelyä, joka voi kestää moninkertaisesti itse reflektoituun metodikutsuun nähden. Reflection API on esimerkiksi erityisen tehokas dynaamisen metadatan tuottamisessa, jota voidaan hyödyntää mm. metaohjelmointimekanismien tai reflektiivisyyden tukena. Lisäksi JDK 1.3:sta lähtien Reflection API on sisältänyt tuen *älykkäille proxeille* (dynamic proxies, smart proxies).

3.4.1 Älykkäät Proxyt

Älykkäät proxyt ovat pääasiassa RMI:n yhteydessä käytettävä metaohjelmointimekanismi. Tavallisesti RMI-etäkutsut suoritetaan palvelukohtaisen *tyngän* (Stub) välityksellä, joka keskustelee *RMI-rekisterissä* (RMI registry) haetussa osoiteessa sijaitsevan *palvelinrungon* (ServerProxy) välityksellä itse palvelun palvelimelle. Tynkä ja palvelinrunko generoidaan staattisesti ennen käännöstä. Tätä mekanismia ei voi muokata RMI-järjestelmää muuttamalla tai muuttamalla kutakin palvelinrunkoa ja tynkää, sillä nämä eivät mahdollista dynaamista metodikutsun muokkausta. Sen sijaan käyttämällä Dynamic Proxy luokkia, kyetään tynkä, välikäsittelijät ja palvelinrunko generoimaan ajonaikaisesti. Kuva 3 sisältää kuvauksen älykkäiden proxyjen ja välikäsittelijöiden vaikutuksesta RMI-kutsuun.



Kuva 3: ÄlykkäätProxyt ja välikäsittelijöt laajennetussa RMI-alijärjestelmässä [SMS02]

Kuvassa 3 muuttumaton asiakassovellus tekee etäkutsun (1). Tässä yhteydessä RMI-järjestelmä lataa tynkänä generoidun älykkään proxyn (2), joka sisältää palvelimen rajapinnat, sekä kutsun *asiakasproxyyn* (client proxy) (3), joka on sarjallistettava objekti, joka lähtetään asiakkaalle älykkään proxyn yhteydessä. Asiakasproxy vastaa itse kommunikoinnista palvelintyngän kanssa, kutsuen tarvittaessa erillisiä asiakaspuolen *välikäsittelijöitä* (Interceptors) (4) ennen varsinaista etämetodikutsua. Itse *etämetodikutsun käsittely* (marshalling) eli parametreina välitettävien objektien sarjallistus ja siirto, kutsutunnisteiden luonti ja käsittely jne. tapahtuvat RMI-alijärjestelmän toimesta tavalliseen tapaan, sovellukselle tuntumattomasti. Palvelimen puolella voidaan palvelinproxyn (6) kutsusta kutsua suorittaa välikäsittelyä (7) ennen varsinaisen etäpalvelimen (8) kutsua.

Mekanismin suurin ongelma on rajapintojen edellytys, joita vasten älykkäitä proxeja tuotetaan. Täten, mikäli sovellus ei ole tekemässä rajapinnan mukaista metodikutsua yhdyspisteessä, on metodi muunnettava toisella kudontamenettelyllä tekemään sellainen, jotta välikäsittelijät kykenevät suorittamaan aspektien metodit.

4 Java sovelluspalvelinten AOP-tuki

Yleispalveluiden tuki aspekteina on sovelluspalvelimissa toteuttavissa joko sovelluspalvelimen omalla AOP-sovelluskehyksellä tai sovelluskehyksellä, joka tarjoaa mahdollisuuden liittää sovelluspalvelimeen esim. aspektikohtaisia sovelluskehyksiä, ollen täten laajennettava ja tarpeiden mukaisesti räätälöivissä. Mikäli sovelluspalvelintoimittaja itse vastaa kuitenkin AOP-sovelluskehyksestä, käytännön asiat kuten vikaselvitys saattavat olla kuitenkin huomattavasti vaivattomampia.

Seuraavat kaksi sektiota käsittelevät kahden sovelluspalvelimen lähestymistä aspektilähtöisiin yleispalveluihin Ensimmäinen sektio käsittelee JBossin AOP-sovelluskehystä painottaen sen deklaratiivisen ohjelmoinnin lähtökohtia. Toinen sektio käsittelee JAC sovelluspalvelimen suunnittelun esilletuomia haasteita tuotettaessa rinnakkaisia aspekteja sovelluksille, SOP:n ristiriidan käsittelyn kaltaisesti.

4.1 JBoss 4.0 Developer Release 2

JBoss:n AOP-toteutus [Bur03] [Bur03b] [Bur03c] on vielä hyvin kehitysvaiheessa. AOP-sovelluskehyksen aiempi toteutus, DR1:n luokkalataaja ei noudattanut luokkakirjastojen latausjärjestystä [BeS03], täten rajoittaen käytön vain JBoss sovelluspalvelimeen. Lähtökohtaisesti DR2.n AOP-sovelluskehys on kuitenkin irroitettavissa itse sovelluspalvelimesta.

Sovelluskehyksen tavoitteena on sekä mahdollistaa kehittäjille omien AOP-toteutusten vaivattomampi käyttöönotto että tarjota valmiita yleispalveluita kattavia aspekteja. Taustalla on ollut sovelluspalvelimen omien, modularisoitujen yleispalvelutoteutusten käyttöönotto tavallisten java-sovellusten toimesta deklaratiivisesti. AOP-sovelluskehyksen toiminta perustuu älykkäisiin proxyihin, mahdollistaen näiden ja välikäsittelijäketjujen ajonaikaisen määrittämisen sekä ajonaikaisen rajapintojen *esittelyn* (introductions). Aspektien kudonta mahdollistetaan luokkien latausaikaisella instrumentoinnilla, edellyttäen täten että AOP-sovelluskehyksen oma luokkalataaja määritetään oletusarvoiseksi luokkalataajaksi sovelluspalvelimen käynnistyksessä, uottaen rajoitteita käytettäessä sovelluskehystä toisten sovelluspalelimien kanssa. Instrumentointi toteutuetaan käytännössä Javassistilla. Luokkalataaja instrumentoi kaikki luokat, joihin kohdistuu aspektikäsittelyä sovelluksen latauksen ajankohtana tai jotka ovat eksplisiittisesti merkittty instrumentoitaviksi. Instrumentoinnin arvioidaan hidastavan luokkalatausta noin 10-15%. Itse aspektit voidaan määrittää ja tarvittaessa muuttaa ajonaikaisesti edellyttäen että luokka on instrumentoitu latauksessa.

AOP sovelluskehys tarjoaa valmiina palveluina transaktioille, käyttäjähallinnalle, klusterituetun ja klusteroimattoman etäkutsun, sekä transaktionaalisuutta tukevan lukko- ja välivarastointipalvelun, joista jälkimmäinen voi tukea myös hajautettua välivarastointia klusteroidussa ympäristössä. Deklaratiivisesti näiden käyttö kuitenkin edellyttää metadatamääritykset, jotka ovat tehtynä sovelluskehyksen toimesta tällä hetkellä transaktioille ja käyttöoikeuksien hallinnalle. Transaktiomääritys vastaa J2EE:n *säiliönhallinnoimien transaktioiden* (container-managed persistence, CMP) määritystä, joskin aspektien myötä tämä on otettavissa käyttöön tavallisessa Java-koodissa. Määrittämättömien aspektien osalta sovelluskehittäjän on määritettävä metadata ja tätä vastaava suoritettava java-koodi sovelluskehyksen puitteissa.

Sovelluspalvelin mahdollistaa lisäksi *metadataketjutuksen* (metadata chaining) määrittääkseen kudontaan hierarkian. Näin voidaan määrittää esim. oletusarvoja aspektien käytölle. Oletusarvot puolestaan voidaan korvata sovelluskehitysprojektien toimesta tarkemman tason määrityksillä. Järjestys on hienoimmasta karkeimpaan: kutsu→säie→objekti→luokka→ JVM→ sovelluspalvelinklusteri.

Liite 4 sisältää esimerkin [Bur03] transaktioinaalisuuden konfiguroinnista JBossissa.

4.2 Java Aspect Components (JAC)

JAC:n [PSD+01] taustalla ei ole J2EE määritystä toteuttavaa sovelluspalvelinta, vaan alkuperäinen lähtökohtakin on ollut toteuttaa yleispalvelut aspekteina. Lisäksi sovelluspalvelimeen on tarkoitus kyetä yhdistämään muita AOP-sovelluskehyksiä. JAC:n aspektitoteutusriippumaton lähestyminen on hyvin luonnollisella tavalla johtanut aspektien koontiongelman käsittelyyn SOP:n kaltaisesti. Koontiongelma jaetaan kahteen osa-alueeseen riippuen onko kyseessä vain staattinen kudonta vai dynaaminen kudonta. Staattiseen kudontaan pätee seuraavat ongelmat:

- Soveltuvuus: mikäli sovellus jo hyödyntää yleispalvelua, joka vastaa aspektia, ei aspektia tulisi kutoa.
- Ristiriidan ja vaihtoehdon ratkaisu: mikäli yhdyspisteeseen on kudottava useita aspekteja, näiden keskinäiset ristiriidat on kyettävä ratkaisemaan sopimuspohjaisesti. Yhtälailla vaihtoehtotilanteissa sopimusten on määritettävä mitä aspektia käytetään.

Dynaaminen kudonta puolestaan edellyttää lisäksi seuraavien seikkojen huomiointia:

- Kontekstisidonnaisuus: on oleeellista ymmärtää perintärakenne niiden aspektien tapauksessa, joiden suorittama koodi ei ole idempotenttia. Toisaalta suorituskykymielessä tämä tulisi ulottaa kaikkiin aspekteihin. Täten aspekteissa tarvitaan AspectJ:n cflow:n kaltainen rakenne, jotta metodikutsu ei aiheuta toistuvia aspektimetodin suorituksia. Lähtökohtaisesti tämän voi toteuttaa joko reflektiolla tai manuaalisesti konfiguroinnilla.
- Aspektin ohitus: aspektien metodikutsujen aiheuttaman lisäkäsittelyn vuoksi voisi olla hyvä, mikäli aspektia ei tarvitsisi joka kerta suorittaa, vaan esimerkiksi persistenssi suoritettaisiin vain joka viides metodikutsukerta.
- Aspektin valinta: reflektiivinen sovelluspalvelin voisi itse valita kudonnan aikaina vaihtoehdoista soveltuvan tilannekohtaisesti esim. perustuen palveluvaateisiin.

Näiden asioiden huomiointi ei ole ilmeistä esim. JBossin toteutuksessa, jossa mm. kentille ja metodeille pätevät yhtäaikaa useat leikkauspisteet tai yleispalvelun ja aspektin yhtäaikaiskäyttöä ei dokumentaatiosta löytynyt paperin valmistelun aikana.

5 Reflektiiviset Java sovelluspalvelimet

Sovelluspalvelinten reflektiivisyydellä (tai adaptiivisuudella) tarkoitetaan kykyä kuvata omaa ja ympäristön tilaa ja rakennetta, tulkita omaa toimintaa ja ympäristön muutoksia kuvausta vasten ja tuottaa toiminnan mukautumista sovelluksen edellyttämien vaateiden mukaisesti. Koska ympäristön kuvauksen tulkinta edellyttää datan keräystä sovelluksen ulkopuolisista lähteistä, on sovellus instrumentoitava [KHL+99], tarkoittaen tässä yhteydessä sitä, että on tuotettava sensoreita (prosesseja, jotka pystyvät kommunikoimaan esim. infrastruktuurin eri osien kanssa), jotka tuottavat tätä dataa järjestelmälle. Itse muutostapahtuman käsittely on joko käännöksen yhteydessä määritetty, jolloin kyseessä on suljettu

adaptiivinen järjestelmä tai sen käsittelyä voi muuttua ajonaikaisesti, jolloin kyseessä on avoin adaptiivinen järjestelmä. Muutostapahtuma laukaistaan joko reaktiivisesti, vaateiden rikkoutumisen jälkeen tai proaktiivisesti, ennen järjestelmän vaateiden rikkoutumista. *Tunkeutumisenhavaitsemisjärjestelmien* (Intrusion Detection Systems, IDS) puitteissa reaktiiviset järjestelmät mukautuvat tiettyyn eksplisiittiseen sääntöjoukkoon, jotka määrittävät milloin adaptaatiota tehdään, kun taas proaktiiviset järjestelmät tyypillisesti perustuvat *poikkemien analysointiin* (anomaly-based adaptation). Tällöin järjestelmä rakentaa kuvan *perustilasta* (baseline), toteuttaen adaptaation aina kun perustilasta poiketaan, oppien ajan myötä erottamaan tavalliseen käyttöön liittyvät *normaalit poikkeamat* (false positives) varsinaisista adaptaatiota edellyttävistä poikkeamista. Muutostapahtuman muutos on joko eksplisiitinen, vaateet määrittävässä sopimuksessa kuvattu muutos tai implisiittinen, tulkinnan perusteella järjestelmän määrittämä muutos, jolloin puhutaan *itseadaptoituvista* (self-adaptive) järjestelmistä. Itseadaptoituvat järjestelmät pyrkivät siis yleiseen, ympäristöriippumattomaan adaptiivisuuteen siinä missä muut adaptiiviset järjestemät ovat ympäristökohtaisia.

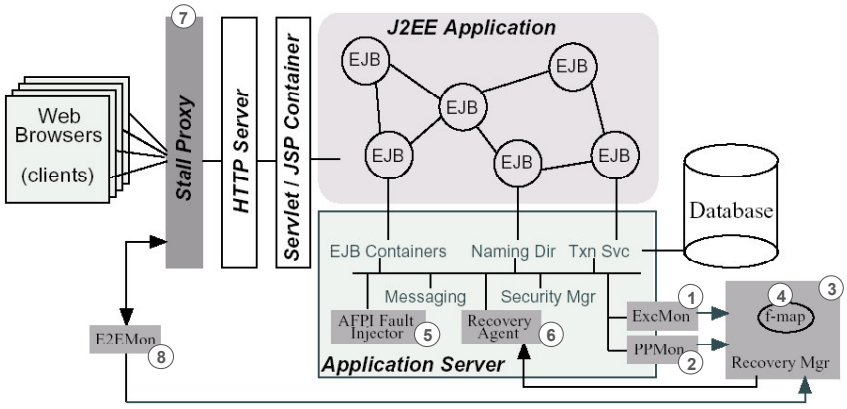
5.1 JBoss with Application Generic Recovery (JAGR)

JAGR [CKZ+03] on osa Standfordin yliopiston *elpymislähtöisen tietojenkäsittelyn* (Recovery-oriented computing, ROC) hanketta [PBB+02], jonka taustajoukoista löytyvät mm. David Patterson ja John Hennessey. Lähtökohtana on tutkia mekanismeja, joilla minimoida virheestä elpymiseen kuluva aika sen sijaan, että tavanomaisten vikasietoisten menettelyiden puitteissa pyritään kasvattamaan vikaantumisväliä. ROC ottaa mallia mm. rakennusarkkitehtuurista, jossa masiivisiin rakennelmiin (esim. sillat, pilvenpiirtäjät) sovelletaan moninkertaista, kerrostettua vikasietoisuutta jo suunnittelussa, jotta toteutuksen ja käytön ennakoimattomat tilanteet eivät kykene rikkomaan rakennelmaa. ROC on yksi osa IBM:n lanseeraamaa *autonomisen tietojenkäsittelyn* (autonomic computing, AC) visiota [KC03], jonka puitteissa pyritään helpottamaan merkittävästi monimutkaistuneiden järjestelmien hallinnointia, niin sovelluspalvelinten kuin tietokantojenkin osalta. AC jakaa autonomian (itseadaptaation) itseoptimointiin (resurssien hallinta), itsekonfigurointiin (perusasetusten automaattinen konfigurointi ja ajonaikainen adaptaation konfigurointi), itsetoipumiseen (vikasietoisuus) ja itsesuojeluun (tietoturva).

JAGR integroi JBoss-sovelluspalvelimeen sovelluksen elvytyksestä vastaavan alijärjestelmän. Tavanomaiseen JBoss tapaan JAGR latautuu omana dynaamisena palveluna, jonka puitteissa sen rekisteröinnin voi lisätä ja poistaa JBossin ytimeltä sovelluspalvelimen ajonaikaisesti.

Poiketen muista adaptiivisista järjestelmistä, JAGR on jo suunnittelulähtökohdistaan tarkoitettu puhtaasti Internetin sähköiseen kaupankäyntiin, mikä näkyy sen adaptaatiomekanismeissa. Näiden toimintaa tarkastellaan seuraavaksi sekä tuotantoa edeltävien toimenpiteiden että tuotannonaikaisten virhetilanteiden osalta.

Järjestelmäarkkitehtuuri sisältää erillisessä prosessissa toimivan *elvytyksenhallinnoijan* (recovery manager) (3-4), kaksi JBossiin lisättävää monitoria (1-2), erillisen käyttäjäkokemusta simuloivan monitorin (8) sekä lisätoiminteita itse JBoss palvelimeen (5-7).



Kuva 4: JAGR arkkitehtuuri [CKZ+03]

Ennen tuotantoa JBoss luo kuvauksen sovellusriippumattomasti Reflection API:lla, muodostaen näiden perusteella listan poikkeuksista per metodi per luokka. Lisäksi listaan lisätään kaikkiin metodeihin mahdollisiksi poikkeuksiksi yleisiä JVM poikkeuksia. Luokkien virhesuhteet tutkitaan tuottamalla poikkeuksia JBossiin integroidulla *vikasyöttäjällä* (fault-injector) (5). Vikasyöttäjän [CDC+03] toiminta perustuu JBossin muutokseen, jossa vikasyöttäjä voi määrittää metodikutsun paluuarvoksi jonkin listassa olevan metodille kohdistuvan poikkeuksen, jonka JBoss tuottaa metodikutsulle paluuarvojen sijaan. Täten erillisen kuormageneraattorin tuottaessa kyselyitä, voi virhesyöttäjä syöttää virheet yksi kerrallaan, monitoreiden tuottaessa dataa elvytyksenhallinnoijalle virheen vaikutuksista. Vaihe kestää pientenkin sovellusten osalta tunteja. JAGR sisältää kome monitoria:

- *Poikkeuskäsittelijämonitori* (ExcMon) (1): käyttäen JBossin poikkeuskäsittelyaspektia, ennen sovelluksen poikkeuskäsittelyä poikkeus lähetetään ExcMon:lle. Tämä ei sinällään kykene raportoimaan useita virtuaalikoneen virheitä, mutta se saa kiinni sekä sovellustason virheet että useat resurssi- ja verkkovirheilmoitukset.
- *PinPoint-monitori* (PPMon) (2): PPMon merkitsee asiakaspyynnöt ja jäljittää näiden etenemistä läpi järjestelmän, tuottaen tarkempaa tietoa sovelluksen komponenttien välisistä metodikutsuista edellyttämättä sovelluskohtaista ohjelmointia. Merkintä ja jäljitys perustuu luokkien instrumentointiin JBoss AOP-sovelluskehyksen luokkaltaajan toimesta ja metodikutsujen väliin sijoitettavaan, sovellusprofilointia tuottavaan aspektikoodiin. Mikäli asiakaskutsu ei etene polun mukaisesti, PPMon raportoi elvytyksenhallinnoijalle. Testeissä PetStore sovelluksella mekanismi lisää 2-40 ms viivettä kuhunkin asiakaspyyntöön, tuottaen lisäksi testialustalle n. 17% lisäkuorman. PPMon pohjautuu projektin aiempaan versioon vastaavan kaltaisesta järjestelmästä, joka toimi itsenäisesti [CKF+02].

- *Käyttjäkokemusmonitori* (E2E-mon) (6): E2Emon on erillisessä prosessissa pyörivä monitori, joka tekee HTTP-kyselyjä tiettyihin sovelluksen sivuihin, kulkien esim. tietyn sovelluksen reitin. E2Emon kykenee erottamaan tietyt palvelimen verkkokorttivirheet, HTTP-virheet sekä yksinkertaiset HTML-virheet (esim. tyhjä sivu).

Raportoitujen vikojen perusteella elvytyksenhallinoija kuvaa virheenetenemisen verkoiksi (fault-propagation maps, f-maps), jotka kuvastavat riippuvuuksia virhetyyppikohtaisesti. Suunnatussa verkossa solmut ovat tietokantaskeemoja, komponentteja, JSP-sivuja tai servlettejä, kaaret ovat näiden välisten virheitä. Kokonaisuudessaan virheverkot muodostavat tietyn sovelluksen kaikkien virhetilanteiden etenemisen.

Tuotantotilanteessa virhemonitorit raportoivat ja ilmoittavat virheitä aivan kuten tuotantoa ennen. Elvytyksenhallinnoija hyödyntää tilastollista analysointimekanismia, jonka perusteella se tekee päätöksiä virheilmoituksien kohteista ja näiden liittymstä toisiin f-mapin solmuista. Virhetilannetulkintaan edellytetään tyypillisesti virheilmoitus useilta monitoreilta, jotta reagointi normaaleihin poikkeamiin vältetään.

Virhetilannekäsittelyssä JBossiin lisätty www-palvelinta edeltävä, virheen peittämisestä vastaava jonokäsittelijä (7) aktivoidaan. Tämän myötä uudet pyynnöt odottavat jonossa kahdeksan sekunttia ennenkuin ne poistetaan tai siirretään sovelluspalvelimen käsiteltäväksi, mikäli sovellus on kyetty elvyttämään tänä aikana. Ajan pituus perustuu Internet-surffaajien käyttäjätutkimuksiin, kuvastaen odotusaikaa, jonka tyypillinen surffaaja jaksaa odottaa ennenkuin klikkaa uudelleenlatausta tai siirtyy eteenpäin.

Jonokäsittelijän toimiessa elvytyshallinnoija mikroboottaa tarvittavat sovelluspalvelimen komponentit. Mekanismista ei löytynyt paperin aikana kuvausta. Lopputuloksena voidaan menettää käyttäjäkonteksteja, esim. jonkun ostoskori voi tyhjentyä. Tätä ei kuitenkaan JAGR:n tekijöiden mukaan keskeistä, mikäli huomioidaan käyttäjämassa esim. Amazon.com:n kaltaisissa portaaleissa sekä kustannukset, jotka tulevat siitä, että koko sovelluspalvelin jää jonkinlaiseen jumitilanteeseen, joita on tyypillisesti vaikea havaita tai joudutaan boottaamaan kokonaisuudessaan. JAGR täten nopeuttaa elpymistä sovellusvirheistä, joskin yhdistettämällä tämän esim. toistoelpyvään JVM:een [Wha01], olisi mahdollista kuvitella prototyyppi kokonaisjärjestelmästä, jonka elpymisnopeus on huomattavasti tavanomaista nopeampaa riippumatta virhetyypistä.

JAGR:n lähestyminen moniin asioihin on kovin käytännönläheinen, mutta sen suunnittelu sivuuttaa toistaiseksi useita virhetilanteita, joihin elvytyksenhallinnoijan tulisi reagoida mikroboottauksesta poikkeavalla tavalla. Näitä ovat mm.:

- LDAPin tai tietokannan jumiutumistilanteisiin elvytyksenhallinnoijan olisi bootattava todellinen vian lähde eikä tehtävä sovelluksen mikroboottia

- Jonokäsittelyn ohella elvytyksenhallinnoijan tulisi kyetä kuvaamaan tila myös esim. sovelluspalvelimen edessä olevalle kuormanjakokytkimelle, mikäli käytössä on
- Sovelluspalvelimen klusterointi voi mahdollistaa tuntumattoman vikasietoisuuden käyttäjille, joten sovelluspalvelimen jumiutuessa voisi olla joskus oikea päätös kaataa sovelluspalvelin kokonaan. Tällöin jonokäsittelijän ei tulisi ottaa myöskään lainkaan kantaa käsittelyyn.
- OS-tason virheet tulevat nyt satunnaisesti poikkeuskäsittelyn kautta järjestelmälle. Mahdollisesti integraatio erillisen sensorin kanssa muuttaisi käsittelyn systemaattiseksi.

Näiden lisäki virheverkot olisi syytä kyetä siirtämään sovelluspalvelimesta toiseen.

Osittain nämä seikat heijastavat JACin yhteydessä (4.1) esitettyjä aspektien ristiriitojen ratkaisua edellyttämiä asioita. Toisaalta JAGR:n lähteissä on osa näistä tuotu esille. Kehityskohteena oleva todennäköisyyspohjaisten vikojen havainnointi (eli kahden vian ilmetessä toistuvasti ilman että näiden välille olisi havaittu seuraussuhdetta kyteään rekisteöimään mahdolliseksi viaksi) tukee puutteiden osittaista ratkaisua. JAGR on tarkoitus sisällyttää seuraavaan JBossin pääversion julkaisuun.

6 Liittymät muuhun tutkimus- ja standardointityöhön

6.1 Metaohjelmointimekanismit

CORBA:n [OMG02] standardin mukaisten metaohjelmointimekanismin lisäksi useat CORBA-palvelintoimittajat ovat rakentaneet oman älykkään proxyn toteutuksen. Tämän lisäksi vikasietoisen CORBAn (Fault tolerant CORBA) pilottitoteutuksen tuottanut ryhma on käsittellyt vaihtoehtoisia lähestymistapoja sovellukselle ja ORBille tuntumattoman toisinnon toteuttamiseksi [NMS99]. Tätä täydentää toisen ryhmän tuottama kooste CORBA:n metaohjelmointimekanismeista ORB-tasolla [WPS01], joka sisältää myös suorituskykymittaukset näille mekanismeille. Virtuaalikoneen puute muodostaa CORBAn osalta keskeisimmän haasteen, mikäli toteutus aiotaan tehdä standardoimattomalla tavalla ja tuntumattomasti niin sovelluksen kuin sovelluspalvelimen osalta.

COM [Box97] arkkitehtuurille metaohjelmointimekanismit tulivat keskeiseen asemaan MTS-sovelluspalvelimen säiealtaiden ja kuormanjakotekniikoiden [Ewa00] myötä. Nämä pohjautuvat älykkäisiin proxyihin. COM metaohjelmointimekanismit ovat kuitenkin olleet aimminkin keskeisessä asemassa Microsoftin tuoteperheidenkin rakennuksessa [HS99], joskin tässä yhteydessä määritetyt idiomaattiset ratkaisut ovat sidottuja IPC-mekanismeja käyttävään ”in-process” COMiin, eivätkä mahdollisesti ole suoraan siirrettävissä esim. DCOMiin. dotNET-arkkitehtuurin virtuaalikoneeneen [Box02] metaohjelmointimekanismeihin liittyviä artikkeleita ei löytynyt paperin valmistelun aikana. dotNET itsessään sisältää Remoting sovelluskehyksen, joka kattaa sekä älykkäät proxyt (RealProxy – luokka) sekä välikäsittelijät (TransparentProxy-luokka) [Mic03].

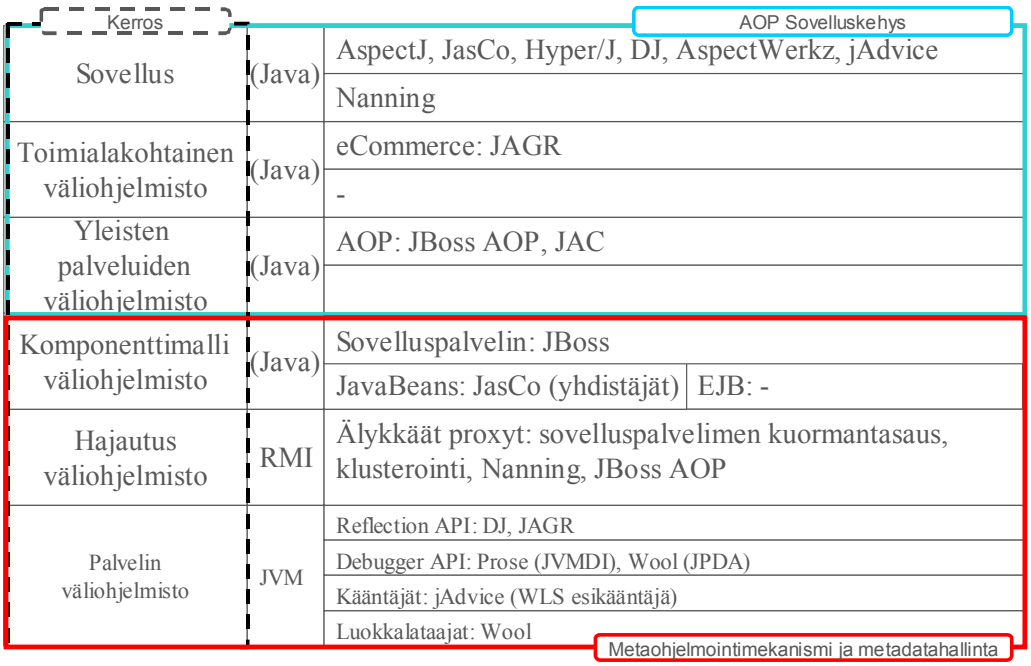
6.2 Reflektiiviset sovelluspalvelimet

Selviytyvät järjestelmät (survivable systems) [EFL+97] on yksi SEI:n ja CERTin aloittamista yhteisprojekteista 1990-luvun loppupuolella, jossa lähtökohtana on mahdollistaa sovellusten suojautuminen tietoturvan ja jatkuvuushallinnan näkökulmasta. QuO-hanke [QuO03] on laajentanut reflektiivisen QuO-sovelluspalvelimen ytimen toimintaa APOD-projektissa [QuO03b], joka keskittyy tuottamaan adaptaation tietoturvahyökkäyksiä vastaan integroimalla varusohjelmia osaksi joukkoa vaihtoehtoisia sovelluspalvelimen sensoreita. Tässä yhteydessä vikasietoisuudetta lähtestytään myös toisenlaisesta näkökulmasta, sillä sovelluspalvelimen on pyrittävä estämään hyökkäys proaktiivisen IDS:n käytön perusteella, jotta hyökkääjän oletusarvoisesti jo onnistunut järjestelmään tunkeutuminen saadaan minimoitua kestoltaan mahdollisimman lyhyeksi [APJ+03].

7 Johtopäätökset

Java-yhteisö on tuottanut merkittäviä toteutuksia niin aspektilähtöisen sovelluskehityksen, metaohjelmointimekanismien, aspektilähtöisten yleispalveluiden kuin reflektiivisten sovelluspalvelinten alueella. Kehityksen taustalta löytyy merkittäviä yrityksiä, yliopistoja ja tutkimuskeskuksia, mutta varsinaista keskitettyä koordinointia hankkeille ei näyttäisi olevan; paperin valmistelun aikana ei löytynyt näille Javan alueille dedikoituja tutkimuskonferensseja, joissa aluetta käytäisiin läpi vaan pääasiassa tietoa eri järjestelmistä oli saatavilla eri yhteisöjen sivuilta, haastatteluista jne. Alueet ovat kuitenkin alati muutoksessa, erityisestikun huomioidaan että virtuaalikoneisiin on tulossa profilointia tukevia rajapintoja [Sun03] ja metadatan käsittelyn standardointikin [Sun03b] on etenemässä hyvää vauhtia.

Viitekehystä kokonaisuudelle ei löytynyt, mutta tukeutumalla kerrostettuun komponenttimalliin [SKK+00], voidaan paperin aiheet jaoitella kuvan 5 mukaisesti:



Kuva 5: Viitekehys esityksen sovelluskehyksistä, mekanismeista ja sovelluspalvelimista

Poiketen alkuperäisestä mallista, kerroksiin on lisätty komponenttimalliväliohjelmisto, joka hyödyntää komponenttimallin koonnin edellyttämiä kuvauksia sekä sovelluspalvelimen konfiguraatiotiedosta, tuottaen pääasiassa staattista metadataa muille mekanismeille, Reflectio API:n dynaamisen metadatan ohella. Yhtälailla poiketen on JAGR sijoitettuna toimialakohtaiseksi väliohjelmistoksi, vaikka tämä tyypillisesti kattaa tietylle toimialalle räätälöidyt palvelut, eivät yleispalveluita kuten elpyvyys. Korostuksen taustalla on järjestelmän suunnittelun kohdistus sähköiseen kaupankäyntiin.

Lähteet

Viitteet tarkistettiin 17.10.2003

[APJ+03] Michael Atighetchi et al: "Building Auto-Adaptive Distributed Applications: The QuO-APOD Experience", The 3rd International Workshop on Distributed Auto-adaptive and Reconfigurable Systems, in conjunction with the 23rd International Conference on Distributed Computing Systems, Providence, Rhode Island, USA. May 19-22, 2003 http://www.dist-systems.bbn.com/papers/2003/DARES/index.shtml

[Asp03] Aspectwerkz homepage, http://aspectwerkz.codehaus.org/

[Bac01] J.R.Bachrach: "The Java Syntactic Extender", http://www.ai.mit.edu/~jrb/jse

[BCE03] ByteCode Engineering Library homepage, http://jakarta.apache.org/bcel/

[BeS03] BeSee homepage, http://besee.sourceforge.net/

[Bha03] Nitin Bharti: "Tech talk with Gregor Kickzales on AOP", TheServerSide, 2003 http://www.theserverside.com/events/videos/GregorKiczalesText/interview.jsp

[Bir93] Kenneth Birman, "The Process Group Approach to Reliable Distributed Computing", CACM December, 1993

[BK02] David Bosschaert, Arne Koschiel: "Advanced Class Loading in J2EE", The ServerSide, 2002, http://www.theserverside.com/resources/article.jsp?l=AdvancedClassLoading

[Box97] Don Box, *Essential COM,* Addison-Wessley, 1997

[Box02] Don Box, *Essential .NET: Common Language Runtime* Addison-Wessley, 2002

[Bur03] Bill Burke: "Aspect-Oriented Programming and JBoss", OnJava, http://www.onjava.com/lpt/a/3878

[Bur03b] Bill Burke: "Interview: JBoss 4.0 AOP Framework", TheServerSide, 2003 http://www.theserverside.com/events/videos/BillBurke/dsl/q15.html

[Bur03c] Bill Burke: "JBoss Aspect-Oriented Programming", JBoss.org, 2003 http://www.jboss.org/index.html?module=html&op=userdisplay&id=developers/projects/jboss/aop

[CCF03] G. Candea et al: "Improving availability with recursive Micro-boots: a soft-state system case study", to appear, Performance Evaluation Journal, Summer 2003 http://citeseer.nj.nec.com/candea03improving.html

[Cha98] K. Czarnecki, *Generative Programming: Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models,* Väitöskirja, Technische Universität Ilmenau, 1998

[CKF+02] Mike Chen et al: "Pinpoint: Problem Determination in Large Dynamic Internet Services", DSN 2002, http://www.cs.berkeley.edu/~mikechen/publications/ipds2002/ipds2002_pinpoint.pdf

[CKZ+03] George Candea et al: "JAGR: An autonomous Self-Recovering Application Server", Proceedings of the 5th International Workshop on Active Middleware Services, Seattle, WA, June 2003 http://citeseer.nj.nec.com/candea03jagr.html

[Chi03] Shigeru Chiga, Javassist, http://www.csg.is.titech.ac.jp/~chiba/javassist/

[CKF+01] Yvonne Coady et al. “Structuring Operating Systems Aspects”, CACM October 2001

[CST03] Shigeru et al: “Using HotSwap for Implementing Dynamic AOP Systems”, July 13th, 2003, http://www.csg.is.titech.ac.jp/paper/chiba-ecoopws03.pdf

[Dah03] M. Dahm, The Bytecode Engineering Library, http://jakarta.apache.org/bcel/

[LLM99] K. Lieberherr et al: “Programming with Aspectual Components”, Tekninen raportti, **NU**-CCS-99-01, Northeastern University, March 1999 http://www.ccs.neu.edu/home/lorenz/papers/reports/NU-CCS-99-01.html

[LS93] K. Lieberherr, I. Silva-Lepe, “Adaptive Object-Oriented Programming Using Graph-based customisation”, CACM May 1994

[EFL+97] R.J. Ellison: “Survivable Network Systems: An emerging discipline”, Tekninen raportti, CMU/SEI-97-TR-013, November 1997 http://www.sei.cmu.edu/pub/documents/97.reports/pdf/97tr013.pdf

[Emm00] Wolfgang Emmrich: “Software Engineering and Middleware: A roadmap”, Proceedings of the conference on The future of Software engineering (ICSE 2000, http://citeseer.nj.nec.com/emmerich00software.html

[Ewa00] Timothy Ewald: “Use AppCenter Server or COM and MTS for Load Balancing your Component Servers”, Microsoft Systems Journal, January 2000

[BG02] R. Barga, J. Grey, “Phoenix project: fault-tolerant applications”, SIGMOD Record 31:2 (2002), 94-100

[GHJ+95] Eric Gamma et al, *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wessley, 1995

[HL95] Walter L. Hursch, Cristina Videira Lopes: “Seperation of Concerns”, Tekninen raportti NU-CCS-95-03, Northeastern University, Febtruary 1995 ftp://ftp.ccs.neu.edu/pub/people/lieber/crista/techrep95/index.html

[HO93] W. Harrison, H.Ossher: “Subject-Oriented Programming (a critique of pure objects)”, OOPSLA’93, 1993

[HT00] Andrew Hunt, David Thomas, *The Pragmatic Programmer: From Journeyman to Master,* Addison-Wessley, 2000

[HU01] R. Hanenberg, R.Unland: “Using and reusing aspects in AspectJ”, OOPSLA, October 2001, http://citeseer.nj.nec.com/hanenberg01using.html

[HS99] Galen C. Hunt, Michael L. Scott: “Intercepting and Instrumenting COM Applications”, COOTS’99, May 3-7, 1999, http://research.microsoft.com/~galenh/Publications/HuntCoots99.pdf

[JCJ+94] Ivat Jakobsen et al, *Object-Oriented Software Engineering: A Use-Case driven approach,* Addison-Wessley, 1994

[Jak03] Ivar Jakobsen: “Use Cases and Aspects - Working Seamlessly Together”, Journal of Object Technology, Vol 2, No 4, August-July 2003, http://www.jot.fm/issues/issue_2003_07/column1.pdf

[Jon03] Rod Johnson, *Expert One-on-One: J2EE Design and Development,* Wrox Press, 2003

[KC03] J. Kephart, D. M. Chess: "The Vision of Autonomic Computing", IEEE Computer 35, 2003

[KCB+02] Fabio Kon, Fabio Costa, Gordon Blair, Roy H. Cambell: "The Case for Reflective Middleware", CACM October 2002

[KHL+99] Michael J. Katchabaw et al: "Making Distributed applications manageable through instrumentation", The Journal of Systems and Software 45, 1999, http://citeseer.nj.nec.com/40013.html

[Kic91] Gregor Kiczales, *Art of Meta-object Protocols,* MIT Press, 1991

[Kic96] Gregor Kiczales: "Beyond the Black-Box: Open Implementation", IEEE Software January 1996, http://www2.parc.com/csl/groups/sda/projects/oi/ieee-software/

[Lad02] Ramnivas Laddad: "I want my AOP!", artikkelisarja, JavaWorld, http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect_p.html

[Lee03] Bob Lee, Jadvice Home Page, http://crazybob.org/downloads/jadvise-javadoc/

[Lee03b] Bob Lee: "Intercepting Method Invocations on POJOs using jAdvice", February 2003, http://www.ociweb.com/jnb/jnbFeb2003.html

[Mic03] Microsoft: ".NET Framework Development Guide", http://msdn.microsoft.com/library/

[Mey97] Jasmin homepage, http://mrl.nyu.edu/~meyer/jasmin/

[Nan03] Nannings homepage, http://nanning.snipsnap.org/space/start

[NEF01] Paniti Netinant, Tzilla Elrad, Mohammed E. Fayad: "A Layered Approach to Building Open Aspect-Oriented Systems", CACM October 2001

[NMS99] Priya Narasimhan, Louise E. Moser, P.M. Melliar-Smith: "Using Interceptors to Enhance CORBA", IEEE Computer, July 1999

[OL01] D. Orleans, K. Lieberherr: "DJ: Dynamic adaptive programming in Java", Tech. Rep. College of Computer Science, Northeastern University, Boston, MA, March 2001.

[OMG01] OMG, *Extensible Transport Framework for Real-Time CORBA*, document orbos/2000-09-12

[OMG02] OMG, *CORBA 3.0.2 Specification,* 02.12.2002 www.omg.org/technology/documents/formal/corba_iiop.htm

[OT01] Harold Ossher, Perri Tarr: "Using Multidimensional Seperation of Concerns to (Re)shape Evolving Software", CACM October 2001

[Pet02] Brett Peterson: "Understanding J2EE Application Server Class Loading Architecture", TheServerSide, May2002, http://www.theserverside.com/resources/article.jsp?l=ClassLoading

[PBB+02] David Patterson et al: "Recovery Oriented Computing (ROC): Motivation, Definition, Techniques, Case Studies", Technical Report CSD-02-1175, UCB, 2002 http://citeseer.nj.nec.com/patterson02recovery.html

[PGA01] Andrei Popovici et al: "Dynamic Weaving for Aspect-Oriented Programming", Proceedings of the 1st International Conference on Aspect-Oriented Software Development, 2001 http://citeseer.nj.nec.com/popovici02dynamic.html

[PSD+01] R. Pawlak, et al, “JAC: A Flexible Solution for Aspect-Oriented Programming in Java,” in Metalevel Architectures and Separation of Crosscutting Concerns (Reflection 2001), LNCS 2192, pp. 1–24, Springer, 2001.

[QuO03] QuO program homepages, http://quo.bbn.com/

[QuO03b] QuO program APOD project homepages, http://apod.bbn.com/

[SKK+00] Douglas Schmidt et al: “Developing Next-Generation Distributed Applications with QoS-enabled middleware”, 2000, http://www.arl.wustl.edu/Publications/2000-04/cm00fgk.pdf

[SMS02] Nuno Santos et al: ”A Framework for Smart Proxies an Interceptors in Java”, University of Coimbra, Portugal, 2002, http://citeseer.nj.nec.com/542994.html

[Sos03] Dennis M. Sosnovski: ”Java-programming dynamics”, artikkelisarja, developerWorks, IBM, http://www-106.ibm.com/developerworks/java/library/j-dyn0429/

[Sul01] Gregory T. Sullivan: ”Aspect Oriented Programming Using Reflection and Metaobject protocols”, CACM October, 2001

[Sun98] Sun: “Reflection API”, SDK 1.1 dokumentaatio, 1997
http://java.sun.com/products/archive/jdk/1.1/index.html

[Sun99] Sun: “Java Virtual Machine Debugger Interface”, SDK 1.2 dokumentaatio, 1999
http://java.sun.com/products/jdk/1.2/docs/guide/jvmdi/

[Sun99b] Sun: “Java Dynamic Proxy Classes”, SDK 1.3 dokumentaatio, 1999
http://java.sun.com/j2se/1.3/docs/guide/reflection/proxy.html

[Sun02] Sun: ”Java Platform Debugger Architecture”, SDK 1.4 dokumentaatio, 2002
http://java.sun.com/j2se/1.4.1/docs/guide/jpda/index.html

[Sun03] Sun: “JVM Profiling Interface”, JSR-163, 2003
http://www.jcp.org/en/jsr/detail?id=163

[Sun03b] Sun: “A Metadata Facility for Java Programming Language“, JSR-175, 2003
http://www.jcp.org/en/jsr/detail?id=175

[SVJ+02] Davy Suvée, Wim Vanderperren, Viviaine Jonckers: “JasCo: an Aspect-oriented approach tailored for Component-based software development”, AOSD 2003

[TOH+99] Peri Tarr et al: “N Degrees of Seperation: Multidimensional Seperation of Concerns”, Proceedings of the 21st International Conference on Software Engineering, May 1999

[Wha01] John Whaley: “System Checkpointing using Reflection and Program analysis”, Reflection 2001, Springer-Verlag, Lecture Notes on Computer Systems 2192, 2001

[WPS01] Nanbor Wang, Kirthika Parameswaran, Douglas Schmidt: “The Design and Performance of Meta-Programming Mechanisms for Object Request Broker Middleware”, COOTS’01, Jan/Feb 2001

Liite 1: AspectJ esimerkki

Muokattu lähteestä [Lad02]:

// Sovelluskoodi

```
// HelloWorld.java
public class HelloWorld {
    public static void say(String message) {
        System.out.println(message);
    }
    public static void sayToPerson(String message, String name) {
        System.out.println(name + ", " + message);
    }
}
```

// Leikkauspiste päämetodiin ja vihje ennen ja jälkeen metodikutsua sijoitettavaksi metodikutsuiksi

```
// MannersAspect.java
public aspect MannersAspect {
  pointcut callSayMessage() : call(public static void HelloWorld.say*(..));
  before() : callSayMessage() {
    System.out.println("Good day!");
  }
  after() : callSayMessage() {
    System.out.println("Thank you!");
  }
}
```

// Vaihtoehtoinen leikkauspiste metodille sayToPerson ja vihje tilalle suoritettavaksi metodikutsuksi

```
// JapaneseMannersAspect.java
public aspect JapaneseMannersAspect {
   pointcut callSayMessageToPerson(String person) : call(* HelloWorld.sayToPerson(String, String)) && args(*, person);
   Void around(String person) : callSayMessageToPerson(person) {
        proceed(person + "-san");
    }
}
```

Liite 2: JasCo esimerkki

Muokattu lähteestä [SVJ+02]:

// Sovelluskoodi

```
// HelloWorld.java
public class HelloWorld {
    public static void say(String message) {
        System.out.println(message);
    }
    public static void sayToPerson(String message, String name) {
        System.out.println(name + ", " + message);
    }
}
```

// Yhdistäjä-Suhde AspectBeanin koukun ja sovelluksen välillä

```
// Connector.jasco
connector HelloAccessControl {
// mikä AspectBean koukko
   AccessManager.AccessControl control=
// minkä sovellusmetodin yhteydessä suoritetaan
      new AccessManager.AccessControl (* HelloWorld.* (*));
// mikä AspectBeanin koukku
   control.before ();
}
```

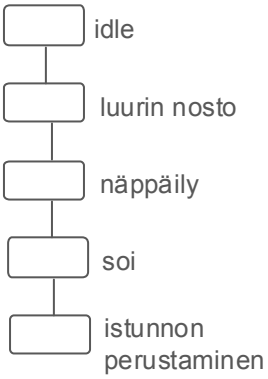
// AspectBean-Koukkujen yhteiskäyttämä koodi sekä kaikki koukut, jota AspectBean toteuttaa

```
// AccessManager.jasco
class AccessManager {
  . . .
  hook AccessControl {
// milloin suoritetaan
    AccessControl (metodi(args)) {
    execute (); }
// mitä suoritetaan
    before () {
     . . .}
  }
}
```

Liite 3: Hyper/J ja käyttötapauskohtainen sovelluskehitys

Muokattu lähteestä [Jak03]:

// Perussovellus: Istunnon luonti



// Ensimmäinen laajennus: Istunnon laatutakeiden määritys

ennen istunnon perustamista, **jälkeen** soi

-määritä laatuvaatimukset (per osapuoli)

- tee tietoliikenteen varaukset (per osapuoli)

- ilmoita varauksen onnistumisesta tai epäonnistumisesta toiselle osapuolelle (per osapuoli)

// Toinen laajennus: Kolmannen osapuolen puhelun hallinta

ennen istunnon perustamista, **jälkeen** soi

-määritä soittajan x vaatimukset

-aloita istunto kohteeseen y

-lähetä x:n istunnon vaatimukset y:lle

-lähetä y:n vaatimukset x:lle istunnon avauspyyntö

-kuittaa x:n hyväksyntä y:lle

-kuittaa y:n hyväksyntä x:lle

// Kolmas laajennus: hätäpuhelujen mahdollistus

ennen soi, **jälkeen** näppäily

-tarkista numero

-jos hätänumero, muuta tunnistetietoja ennen istunnon aloittamista

-tarkista lähin hätäkeskus

-tarkista verkon hätäpalvelut

-aloita istunnon alustus

Liite 4: JBoss AOP-konfiguraatioesimerkki

Muokattu lähteestä [Bur03c]:

```
<?xml version="1.0" encoding="UTF-8"?>

<aop>
<!-- *** Pointcuts *** -->
<method-pointcut class="org.jboss.POJO" methodName="set*.*">
  <interceptors>
     <interceptor class="org.jboss.TransactionInterceptor"/>
  </interceptors>
</method-pointcut>


<!-- metadata -->
<class-metadata group="transaction" class="org.jboss.test.aop.bean.POJO.*">
  <default>
    <trans-attribute>Required</trans-attribute>
  </default>
  <method name="setMethod1.*">
    <trans-attribute>RequiresNew</trans-attribute>
  </method>
  <method name="setMethod2.*">
    <trans-attribute>Supports</trans-attribute>
  </method>
</class-metadata>

<method-pointcut group="transaction" attr="trans-attribute">
  <interceptors>
     <interceptor class="org.jboss.TransactionInterceptor"/>
  </interceptors>
</method-pointcut>
</aop>
```