hyväksymispäivä arvosana

arvostelija

Metatiedon hallinnan ja reflektion yhdistäminen väliohjelmistoissa

Janne Metso

Helsinki 1. marraskuuta 2003

Seminaaritutkielma

HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty	Laitos — Institution — Department
Matemaattisluonnontieteellinen tiedekunta	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Janne Metso

Työn nimi — Arbetets titel — Title

Metatiedon hallinnan ja reflektion yhdistäminen väliohjelmistoissa

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Seminaaritutkielma	1. marraskuuta 2003	18 + 1

Tiivistelmä — Referat — Abstract

Väliohjelmistoja käytettäessä järjestelmää kuvataan metatiedon avulla. Tätä metatietoa käytetään järjestelmän rakenteen ja tilan selville saamiseen. Hajautettuille järjestelmille asetettujen vaatimusten täyttämiseksi reflektiiviset väliohjelmistot ovat tarpeellisia. Metatiedon avulla reflektiivisiä ominaisuuksia voidaan tuoda väliohjelmistoihin. Erityisesti se onnistuu hallitsemalla metatietoa sellaisten metamallien avulla, jotka tukevat reflektiivisyyden erilaisia piirteitä. Erillisillä metamalleilla saadaan myös reflektiivisyyteen liittyviä ongelmia jaettua pienempiin osiin.

Avainsanat — Nyckelord — Keywords
middleware, reflection, meta-information management

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Väliohjelmistoilla helpotetaan hajautettujen järjestelmien välistä kommunikaatiota. Helpottaminen tapahtuu tuomalla uusi abstraktiotaso ohjelmoijan käyttöön. Tällä abstraktiotasolla piilotetaan kommunikointiin liittyvät ongelmat sovellusohjelmoijilta ja siten helpotetaan ohjelmistojen suunnittelua.

Tähän asti väliohjelmistot ovat kuitenkin tarjoneet suhteellisen staattisen tavan tehdä niiden päälle rakennettuja ohjelmistoja. Tästä on seurannut se, että aina kun ohjelmistoihin on tarvinnut lisätä uusia ominaisuuksia on sitä pitänyt muuttaa huomattavasti. Tämän vuoksi olisikin tarvetta sellaisille väliohjelmistoille, jotka osaavat adaptoitua ympäristöönsä ja niitä käyttävien ohjelmistojen tarpeisiin [KCB+00].

Hajautetut komponenteille tarpeellisia ominaisuuksia ovat seuraavat ominaisuudet: koordinaatio vaatimukset, turvallisuus, luotettavuus, suorituskyky ja tapahtumien kirjaamisella [Tri02]. Kaikkii näihin tarpeisiin yritetään vastata lisäämällä reflektiota väliohjelmistoihin. Reflektiivisillä ominaisuuksilla voidaan vastata kaikkiin näihin tarpeisiin.

Väliohjelmistoissa kuvataan järjestelmän rakennetta metatiedon avulla. Metatiedolla voidaan kuvata järjestelmän rakenteen lisäksi myös järjestelmän ominaisuuksia. Perinteisesti metatieto on tallennettu erityisiin varastoihin joiden avulla metatietoa on helppo käsitellä. Metatieto voidaan jakaa osa- alueisiin joiden avulla sen hallinta on helpompaa.

Metatietoa ja sen hallintaa voidaan käyttää reflektiivisten ominaisuuksien tuottamiseen. Metatieto tarvitsee jakaa sopiviin osa-alueisiin, jotka tukevat reflektion erilaisia tarpeita. Tämän jaon avulla metatiedon hallinnasta tulee osa reflektiivistä koneistoa. Reflektiivisyydessä on kaksi eri aluetta rakenteellinen reflektio ja reflektiivinen käyttäytyminen. Molempia näitä alueita voidaan tukea metatiedon avulla.

Tässä tutkielmassa palvelulla tarkoitetaan sellaista oliota tai joukkoa olioita jotka tarjoavat yhteisen rajapinnan. Rajapinnan avulla voidaan tuottaa jokin yhdeksi kokonaisuudeksi luokiteltava joukko operaatiota joilla voidaan käsitellä samaa tietoa. Yhteinen rajapinta voi muodostua joko erillisenä määritellystä rajapinnasta josta oliot toteuttavat osan

tai muodostamalla olioiden rajapinnnoissa olevista sopivista operaatioista koostamalla.

Komponentilla tarkoitetaan tässä tutkielmassa ohjelmakoodia, jolla on hyvin määritelty rajapinta. Lisäksi komponentti sulkee sisäänsä tarkoin määritellyn toiminnallisuuden. Esimerkkeinä komponenteista voidaan pitää J2EE-standardin mukaisia Javan papuja (*beans*) ja CORBA:n komponenttimallin mukaisia komponentteja.

Tässä tutkielmassa perehdytään aluksi pintapuolisesti väliohjelmistoissa olevaan metatietoon. Sen jälkeen perehdytään väliohjelmistojen reflektiivisyyteen. Kolmantena asiana käsitellään metatiedon hallintaa väliohjelmistoissa ja lopuksi tarkastelaan miten metatiedon hallintaa voidaan käyttää apuna reflektiivisyyden toteutumiseen väliohjelmistoissa.

2 Metatieto väliohjelmistoissa

Tässä osassa luodaan nopea katsaus väliohjelmistoissa käytettyyn metatietoon. Metatieto voidaan kuvata kaikkea sovelluksien rakenteesta niiden tilaan ja itse väliohjelmistonkin toimintaan. Esimerkkejä metatiedosta ovat: tyyppitieto, rajapinnat ja politiikat. Lisäksi metatieto voidaan jakaa erillisiin tasoihin. Mainittuihin esimerkkeihin ja metatiedon tasoihin tutustutaan seuraavaksi.

Tyyppitiedolla tarkoitetaan palvelun tyyppiä. Palelun tyyppi muodostuu yleensä palvelun tarjoamasta rajapinnasta ja joukosta ominaisuuksia, jotka on määritelty jollakin tavalla. Lisäksi voidaan vielä määritellä semanttisia ominaisuuksia, joita palvelun toiminta vastaa. Tyyppi voidaan myös ilmoittaa luokkana esimerkkinä Javan String-luokka. Kuitenkin monimutkaisemmat palvelut koostuvat useista olioista jolloin pelkkä olion luokka ei riitä.

Useimmissa väliohjelmisto alustoissa on määritelty kieli, jonka avulla voidaan kuvata järjestelmän osia [CB00]. Esimerkiksi CORBA ympäristöissä on IDL-kieli, jonka avulla voidaan kuvata rajapintoja. Rajapinnoilla kuvataan palvelun tarjoamia operaatioita. Yksi palvelu voi koostua useasta rajapinnasta. Jos palvelu koostuu useammasta oliosta (komponentista), palvelun rajapinnan muodostavat ne operaatiot jotka kustakin komponen-

tista halutaan julkaista.

Politiikoilla kuvataan palveluiden ominaisuuksia ja tarpeita. Nämä ovat niinsanottuja ei tominniallisia (*non-functional*) ominaisuuksia. Esimerkiksi palvelun tarvitsemat resurssien määrät voidaan kuvata politiikkojen avulla.

Taso	Entiteetti	Esimerkki
3	Meta-meta-tyypit	Luokat, viittaukset, rajoitteet, pakkaukset, assosiaatiot
2	Meta-tyypit	Luokka 'StrInterfaceDef'
1	Tyypit	Jokin rajapintatyyppi
0	Instanssit	Olemassa olevan luokan rajapinta

Taulukko 1: Metatiedon tasot MOF mallissa [CB00].

Talukossa 1 on esitetty OMG:n (Object Management Group) MOF-standardin [OMG99] (Meta-Object Facility) mukaiset metatiedon tasot ja esimerkkejä niille kuuluvista kokonaisuuksista. MOF on geneerinen metatiedon esittämiseen tarkoitettu standardi [CB00]. MOF lajittelee metatiedon neljään tasoon, joista taso 0 ei oikeastaan ole enää metatietoa. Tasolla 1 on perustyypit eli luokkien määrittelemät tyypit. Tasolla 2 on metatyypit, eli esimerkiksi siellä voidaan määritellä jollekin luokalle tyyppit (jonkin tyyppinen rajapinta). Tasolla 3 on meta-metatyypit joilla voidaan asettaa rajoitteita, pakkauksia ja assosiaatoita. Esimerkiksi voidaan rajoittaa metatyyppien ominaisuuksia.

3 Reflektiivisyys väliohjelmistoissa

Tässä osassa tarkastellaan mitä kaikkia asioita reflektiivisyyteen kuuluu väliohjelmistoissa. Reflektiiviset ominaisuudet ovat hyvin lähellä adaptoitumista. Adaptiivisuuteen on vain yhdistetty metatiedon tuottaminen järjestelmän tilasta. Kun sitten metatietoa muutetaan niin järjestelmän tilan tulisi muuttua vastaamaan muutettua metatietoa. Tärkeä ominaisuus on myös väliohjelmistoa käyttävien ohjelmien mahdollisuus käsitellä itse

metatietoa ja siten käsitellä omaa toimintaa. Väliohjelmistoissa reflektio vastaa kysymykseen miten adaptoidutaan, mutta ei siihen mitä adaptoidaan ja milloin [KCB+00].

Reflektiivisyydessä yhtä tärkeä ominaisuus kuin ohjelmien käsitys itsestään on väliohjelmistoalustan käsitys itsestään. Väliohjelmiston muutosten pitää heijastua metatietoon aivan kuten ohjelmistojen muutostenkin. Väliohjelmistojen muuttuminen voidaan jakaa neljään vaiheeseen [BBI+00]:

1. *Väliohjelmiston muutosten olosuhteiden havainnointi.* Muutoksen voi laukaista kaksi tekijää a) haluttujen väliohjelmiston ominaisuuksien muuttaminen ja b) ympäristön muuttaminen. Ensimmäinen johtuu lähinnä sovelluksen muuttumisesta ja jälkimmäinen suoritusympäristön tai ohjelmiston muutoksista vaatien väliohjelmiston muuttamista.

2. *Halutun muutoksen mukaisen uuden väliohjelmiston rakenteen tuottaminen.* Kun on päätetty, että väliohjelmiston pitää muuttua, pitää laskea minkälainen vaikutus muutoksilla on tämänhetkiseen väliohjelmiston tilaan. Muutoksen monimutkaisuudesta riippuu kuinka paljon joudutaan tekemään. Esimerkiksi pelkän konnektorin vaihtaminen ei vaadi suuria toimenpiteitä.

3. *Päättäminen milloin on turvallista muuttaa väliohjelmistoa.* Muunnosprosessin pitää taata sovellusten pysyminen konsistentissa tilassa väliohjelmiston muunnoksen aikana. Esimerkiksi ohjelmien, jotka tarjoavat transaktionaalisia toiminnan pitäisi häiriintyä mahdollisimman vähän muunnoksen aikana.

4. *Itse väliohjelmiston muutoksen tekeminen.* Tämä vaihe on hyvin riippuvainen alla olevan väliohjelmiston reflektiivisistä ominaisuuksista. Kuitenkin sopivan metatiedon avulla näitä toimintoja voidaan tukea.

Väliohjelmistoa muutettaessa on otettava huomioon muutenkin sovellusten toiminta. Muutosten tapahtuessa väliohjelmiston pitää siirtyä turvallisesta tilasta toiseen turvalliseen tilaan [Ven02]. Turvallisella tilalla tarkoitetaan tilaa, jossa väliohjelmiston muutokset eivät aiheuta käynnissä oleville sovelluksille ongelmia. Jos kuitenkin sovelluksia

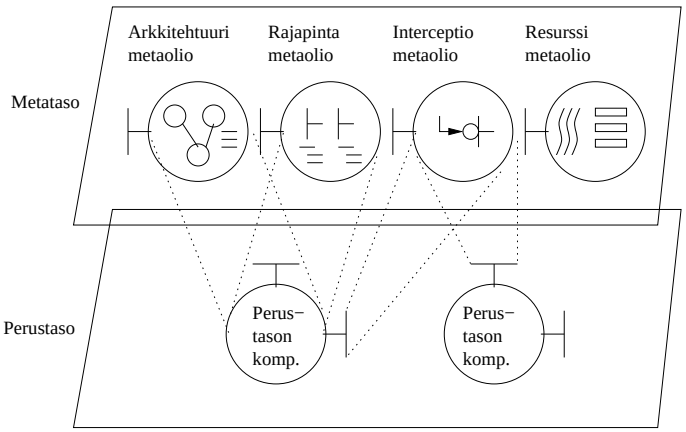
suoritetaan parhaillaan on suoritettavien operaatioiden semantiikan säilyttävä ennallaan [Ven02, BBI+00]. On huomattavaa, että semantiikan säilyttäminen on kova vaatimus. Sen vuoksi onkin ehdotettu, että muutoksia tehtäisiin vain ennaltamäärätyissä pisteissä sovellusten suoritusta [BBI+00]. Etukäteen määritellyjen adaptaatiopisteiden avulla voidaan varmistua siitä, että adaptoinnin tapahtuessa ei yhdenkään sovelluksen toiminta häiriinny. Jokaiselle sovellukselle pitää määritellä nämä pisteet erikseen. Käytännössä väliohjelmiston ja sitä käyttävien sovellusten toiminta pysäytetään ennalta määritettyyn adaptaatiopisteeseen, suoritetaan muutokset ja jatketaan pisteestä eteenpäin.

Väliohjelmistojen adaptoinnille on voitava asettaa rajoitteita. Rajoitteilla voidaan vaikuttaa siihen miten väliohjelmisto adaptoituu. Lisäksi rajoitteilla vaikutetaan siihen miten paljon väliohjelmisto voi muuttua. On tärkeää että adaptointi säilyttää annetut rajoitteet [BCB+02]. Jos rajoitteita ei noudateta, voi ohjelmien toiminta vaarantua käytettyjen palvelujen muuttuessa tunnistamattomiksi niiden adaptoituessa.

4 Metatiedon hallinta

Tässä osassa tarkastellaan miten metatietoa voidaan hallinta väliohjelmistoissa. Väliohjelmistoissa perustason (*base-level*) tietoa voidaan hallita metatason (*meta-level*) tiedon avulla. Reflektiiviset ominaisuudet jakaantuvat aiheryhmiin, joista voidaan muodostaa sopivia metamalleja. Tämän tyyppinen metatiedon pilkkominen sopiviksi metamalleiksi antaa helpommat lähtökohdat vaadittavien ominaisuuksien saavuttamiseen.

Kuvassa 1 on esitelty väliohjelmistojen meta-avaruuden jakaminen metamallieihin. Kuvassa on kaksi tasoa perustaso ja metataso. Perustasolla olevat komponentit käyttävaät metatason malleja itsensä kuvaamiseen. Kuvassa olevat metamallit ovat arkkitehtuurimetamalli, kapselointi-metamalli, interceptiometamalli ja resurssimetamalli. Näistä metamalleista kapselointi-metamalli ja arkkitehtuuri-metamalli tukevat reflektiivistä rakennetta ja interceptio-metamalli ja resurssimetamalli tukevat reflektiivistä käyttäytymistä.



Kuva 1: Väliohjelmistojen meta-avaruus [BCB+02].

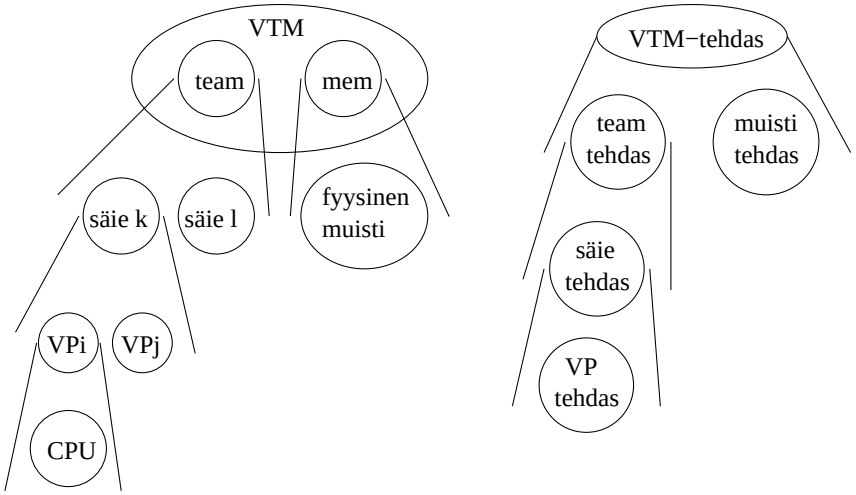
4.1 Resurssi-metamalli

Resurssi-metamallilla hallitaan järjestelmän käytössä olevia resursseja ja niiden allokointia väliohjelmistoa käyttävien sovellusten tarpeisiin. Metamalli jakaa järjestelmän resursseihin ja tehtäviin. Tehtävät ovat loogisia aktiviteetteja jotka käyttävät resursseja. Resurssit ovat joko yksinkertaisia (eli esimerkiksi muistia) tai monimutkaisia (eli esimerkiksi säikeitä tai puskureita datan käsittelyyn) [DB00].

Väliohjelmiston avulla keskenään kommunikoivien komponenttien vaatimuksina ovat ainakin seuraavat ominaisuudet: säikeiden käsittely, viestien vastaanotto ja jonottaminen [CBC98]. Näitä ominaisuuksia pitää pystyä tukemaan resurssi-metamallin avulla. Resurssi-metamallin avulla voidaan myös muokata sovelluksen resurssien käyttöä [BCB+02]. Yksinkertaisesti allokoimalla sovelluksen käyttöön enemmän resursseja sovelluksen saamat resurssit lisääntyvät ja sen saama palvelun laatu paranee.

Resurssi-metamalli muodostaa komposition resursseista ja siten itseasiassa metametamallin. Tässä piilee reflektio tornin vaara [CBC98]. Sen vuoksi onkin oltava varovainen metamallin käyttämisessä ja käyttäytymisen muokkauksessa. Kuitenkin normaalissa tapauksessa tästä ei pitäisi olla suurta vaaraa kunhan asia pidetään mielessä.

Kuvassa 2 a) on esitetty OpenORB:ssa [BCA+01] käytetty resurssi-metamallin hierarkia. OpenORB:n metamalli on jaettu erillisiin tasoihin. VTM (Virtual Task Machine) esittää re-



a) Abstraktien resurssien hierarkia b) Tehtaiden hierarkia

Kuva 2: Eräs resurssi-metamallin mahdollinen hierarkia

surssien hallinan yksikköjä suorittavassa järjestelmässä. Käytännössä VTM käsittää yksikölle annetut resurssit ja käytössä olevien resurssien hallintapolitiikat. VTM rakentuu abstraktien resurssien, jotka esittää prosessoria ja muistia, päälle. Tässä mallissa käytetään team-abstraktiota useiden säikeiden muodostamista tehtävistä. Yksi säie voi taas olla suorituksessa yhdellä tai useammalla virtuaali prosessorilla. Fyysistä prosessoria sitten vaihdellaan taas virtuaaliprosessorien avulla. Allokoimalla useita virtuaali prosessoreita yhdelle säikeelle voidaan säikeen saamaa suoritusaikaa kasvattaa.

Kuvassa 2 b) on esitetty OpenORB:ssa käytetyn resurssi-metamalliin liittyvien tehtaiden hierarkia. Koska tehtaita käytetään resurssien luomiseen, muistuttaa tehtaiden hierarkia suoraan abstraktien resurssien hierarkiaa.

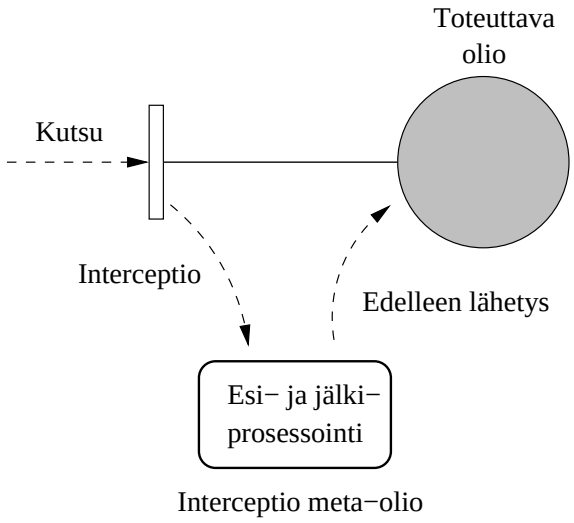
4.2 Interceptio-metamalli

Interceptio-metamalli tarjoaa mahdollisuuden lisätä interceptoreita komponenttien väliseen kommunikointiin. Interceptorilla tarkoitetaan ohjelmakoodia, joka liitetään kahden komponentin väliin siten, että kaikki liikenne komponenttien välillä kulkee interceptorin kautta.

Interceptorit liitetään komponenttien rajapintoihin [BCB+02]. Tarkemmin sanoen interceptorit liitetään paikallisiin sidontoihin. Tosin sidonta voi olla ulottua kahteen erilliseen palveluun muodostaen hajautetun sidonnan. Näin voidaan tarkkailla myös kahden komponentin välistä kommunikaatiota.

Interceptio metamallin avulla komponenttien käyttäytymistä voidaan muuttaa. Muuttaminen tapahtuu lisäämällä interceptori ja siten käytännössä muuttaen joko esi- tai jälkikäyttäytymistä [BCB+02]. Esikäyttäytymisellä tarkoitetaan lisätoimenpiteitä ennen varsinaiselle komponentille menevää kutsua ja jälkikäyttäytymisellä lisätoimenpiteitä varsinaisen komponentti-kutsun jälkeen.

Interceptorit tarjoavat myös välineet komponenttien välisen kommunikoinnin tarkkailuun [BCB+02]. Koska interceptori tulee komponenttien väliin ja kaikki kommunikaatio kulkee niiden kautta, liikenteessä tapahtuvia muutoksia voidaan seurata ja raportoida eteenpäin.



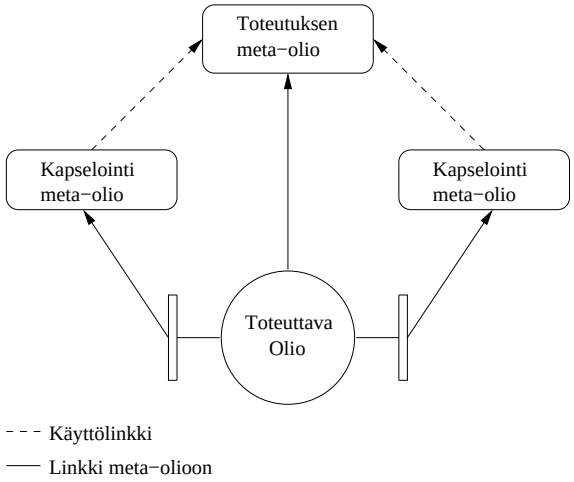
Kuva 3: Interceptio olion kutsussa [CBC98].

Kuvassa 3 nähdään kuinka interceptio metamallin mukaisesti toteuttavalle oliolle lisätään käyttäytymistä. Kuvassa oliolle tulee kutsu joka kuitenkin ohjataan ensin interceptoivalle oliolle esiprosessointia varte. Prosessoinnin jälkeen kutsu jatkaa eteenpäin varsinaiselle toteuttavalle oliolle.

4.3 Kapselointi-metamalli

Kapselointi-metamalli tarjoaa pääsyn komponentin rajapintoihin. Rajapinnat ovat nimenomaan kopmonentin käyttämät ja tarjoamat rajapinnat. Lisäksi metamalli tarjoaa välineet tietyn komponentin kaikkien rajapintojen luetteluun ja lisäksi mahdollisuuden selvittää komponentin tyyppi. Metamalli siis tarjoaa Java-kielen tyypiset dynaamiset komponentin rajapintojen tarkasteluominaisuudet [BCB+02].

Metamalli on kuitenkin kieliriippuvainen. Esimerkiksi käännetyissä kielissä kuten C tarkastelu voi olla rajoitettu vain IDL-rajapintakuvauksiin. Tulkatuissa kielissa (kuten Java ja Python) laajempi tarkastelu voi olla mahdollista. Laajemmalla tarkastelulla voidaan tarkoittaa esimerkiksi metodien lisäämistä tai poistamista komponentin rajapinnasta.



Kuva 4: Kapselointi meta-olio ja siihen liittyvät meta-oliot [CBC98].

Kuvassa 4 on kuvattu kapselointi-metamallin meta-olio ja siihen liittyvät muut meta-oliot. Kuvassa on olio, joka toteuttaa jonkun tietyn tominnallisuuden (esimerkiksi laskutuksen). Oliolla on kaksi rajapintaa. Olioon sekä sen toteuttamiin rajapintoihin on liitetty meta-oliot jotka kuvaavat niitä. Toteutusta kuvaava meta-olio käyttää vielä hyväkseen rajapintoja kuvaavia meta-olioita.

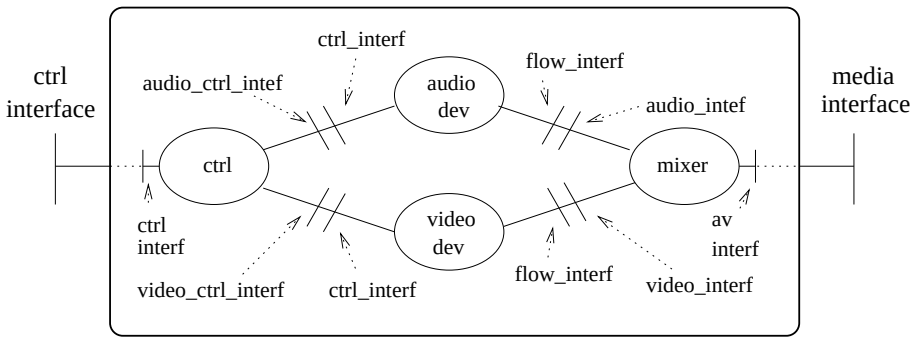
4.4 Arkkitehtuuri-metamalli

Arkkitehtuurilla tarkoitetaan tässä tapauksessa yhden palvelun muodostumisesta useammasta oliosta. Arkkitehtuuri-metamalli tarjoaa mahdollisuuden muokata komponentin ohjelmistoarkkitehtuuria [BCB+02]. Kun komponentti koostuu useammasta oliosta, voidaan olioita vaihtaa ja siten muuttaa komponentin toimintaa. Esimerkiksi komponentti, joka lähettää multimediavirtaa (kuvaa ja ääntä), voi vaihtaa kuvanpakkausalgoritmia kesken suorituksen havaitessaan siihen tarpeen. Olio vaihdetaan arkkitehtuurissa toiseen arkkitehtuuriin sopivaan olioon, joka toteuttaa toisen pakkausalgoritmin.

Arkkitehturaaliseen adaptiointiin tarvitaan myös rajoitteita [BCB+02]. Näitä rajoitteita saadaan aikaan esimerkiksi tyyppitarkasteluilla mutta myös muita rajoitteita voidaan asettaa. Edellisen esimerkin mukaisesti valittaessa arkkitehtuuriin uutta oliota, joka hoitaa kuvanpakkausta, valitaan sellainen olio jonka tyyppi on sama kuin edellisen käytetyn olion. Pelkkä tyyppi tarkastelu ei kuitenkaan ole tarpeeksi vaan mukaan pitää ottaa laajempi näkemys adaptaatioon. Esimerkiksi pakkausalgoritmin vaihtaminen voi vaatia kommunikaatiokumppanin rakenteeseen samanlaisia muutoksia purkualgoritmin vaihtamiseksi vastaamaan pakkausalgoritmia. Lisäksi voi olla hyödyllistä säilyttää joitakin arkkitehturaalisia ominaisuuksia kuten tiettyjä suodattimia (*filters*). Yksi vaihtoehto tällaisten rajoitusten säilyttämiseksi onkin kirjata ne eksplisiittisesti arkkitehtuuriin [CBC98].

Arkkitehtuurimetamalli on esitetty oliograafina, jossa oliot ovat yhdistetty toisiinsa sidosten avulla [CBC98]. Arkkitehtuurissa olevat sidokset ovat alustasta riippumattomia. Esitystapa mahdollistaa arkkitehtuurin rakenteen tarkastelun ja muuttamisen. Metamallin kannalta komponentit ovat joko yksinkertaisia (koostuen vain yhdestä oliosta) tai monimutkaisia (koostuen monesta oliosta). Jos komponentti on yksinkertainen sillä ei ole arkkitehtuurikuvausta. Normaalisti komponentin sisäinen kommunikaatio on käyttäjälle näkymätöntä, mutta arkkitehtuuri-metamallin avulla sisäistä kommunikaatiota voidaan muuntaa ajonaikaisesti ja sen tyyppiä (esimerkiksi käytetäänkö informaatiovirtaa (*informationstream*) vai ryhmäkommunikaatiota) tutkia.

Kuvassa 5 on komponentti joka koostuu useammasta osasta. Komponentilla on ulospäin



Kuva 5: Esimerkki komponentin sisäisestä arkkitehtuurista [CB00].

kaksi rajapintaa hallinta- (ctrl_interface) ja mediarajapinta. Komponentin ytimen muodostaa kaksi oliota joiden avulla saadaan tuotettua kuvaa ja ääntä. Tuottaja-olioiden hallintarajapinnat on yhdisetty uudella oliolla joka tarjoaa rajapinnan molempien edellisten olioiden hallintaan. Myös tuottaja-olioiden ääni- ja kuvavirtarajapinnat on yhdistetty uuden mixer-olion avulla. Mixer-olion rajapinnan avulla komponentti pystyy tuottamaan sekä kuvaa että ääntä. Lisäksi kuvaan on merkitty kaikki käytetyt rajapinnat.

5 Metatiedon hallinta ja reflektio

Tässä osassa nivotaan reflektiivisyys ja metatiedon hallinta yhteen ja tarkastellaan miten metatiedon hallinnan avulla saavutetaan reflektiivisiä ominaisuuksia väliohjelmistoissa. Metatiedon avulla voidaan tukea sekä komponenttien reflektiivistä rakennetta että komponenttien reflektiivistä käyttäytymistä.

Reflektiivinen väliohjelmisto vaatii kehystä, joka koostuu muutamien tarpeiden ottamista huomioon meta-tason mallin suunnittelussa. Tällöin eri osia entiteetin meta-alasta hallitaan jakamalla metataso erillisiiin metamalleihin. Metamallit ovat itsenäisiä ja niitä käsitellään erikseen. Metamallit on esitelty edellisessä osassa. Metamallien avulla reflektiivisä ominaisuuksia voidaan totetutaa aina yhden metamallin sisällä ja on siten eristetty muiden metamallien tarpeista [CB00].

Rakenteellisten metamallien tarkoituksena on järjestelmän konfiguraation reifioiminen.

Tällä tarkoitetaan reifikaation kohteen on oltava yhtäpitävä tyyppivaraston säilytettävän metatiedon kanssa. Tyyppivarasto siis säilyttää kaikkea metatietoa sisällään joka tarvitaan komponentien reifioimiseeen. Esimerkiksi kapselointi-metamallin sisältämät kuvaukset ovat rajapintatyyppejä tyypivaraston sisällä.

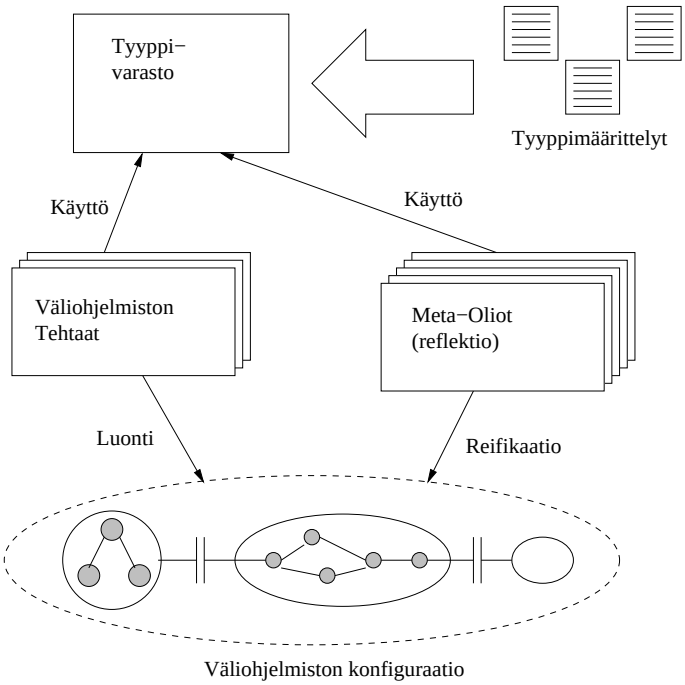
Meta-olion luonnin aikana sen tila luodaan tyyppivarastossa olevan metatiedon avulla. Metatietoa käytetään suoraan olioin luonnin aikana, eikä mitään erillisiä muutoksia tehdä. Arkkitehturaalisen-metamallin tapauksessa tyyppivaraston tietoa käytetään instanssikohtaisten metatiedon tuottamiseen. Käytännössä tämä tapahtuu luomalla kaikki oliograafissa esitetyt oliot ja kiinnittämällä ne toisiinsa graafin esittämällä tavalla.

Instantioinnin jälkeen tyyppivaraston metatietoa voidaan käyttää komponenttien dynaamiseen uudelleen konfigurointiin ja rakenteen muuttamiseen. Uudelleen konfiguraation aikana meta-olio eriyttää tilansa tyyppivaraston metatiedosta ja muokkaa tilansa uudelleen. Tämän jälkeen se luo uuden version alkuperäisestä meta-oliosta ja tallentaa sen uutena tyyppivarastoon. Tällöin järjestelmä etenee käytännössä evolutiivisesti. Samalla luodusta tyypistä tulee meta-olion uusi tyyppi ja siihen viitataan uusien uudelleen konfiguraatioiden aikana [CB00].

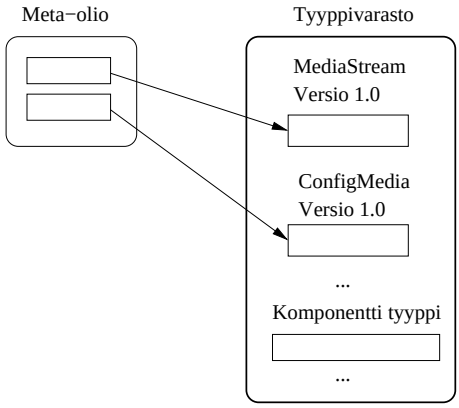
Kuvassa 6 on kuvattu tyyppivaraston rooli konfiguroinninin ja reflektion yhdyspisteenä. Tyyppivarastoon talletetaan tyyppimäärittelyt. Väliohjelmistossa olevat tehtaat käyttävät luodessaan uusia olioita tyyppivarastossa olevia tyyppimäärittelyitä. Meta-oliot käyttävät tyyppivarastoa apunaan muuntautuessaan ja tallettavat uusia versioita perustyypistään tyyppivarastoon. Väliohjelmiston tehtaat luovat uusia olioita ja siten konfiguraatioita väliohjelmistossa toimivaan sovellukseen. Meta-olioiden avulla järjestelmää voidaan reifioida ja siten säilyttää metatiedossa järjestelmän kulloinenkin tila.

Kuvassa 7 on esitetty meta-olioiden ja tyyppivaraston suhde. Meta-olio kuvaa aina lähtökohdaltaan perusolioita, jolla on perustyyppi tyyppivarastossa. Meta-olio kapseloi toiminnallisuuden kun perusolio käyttää useampaa tyyppiä yhtäaikaa. Muutokset olemassa oleviin rajapinta tyyppieihin vaikuttavat suoraan meta-olion esitykseen.

Tässä mallisssa reflektiivinen uudellen konfiguraatio on mallinnettu tyyppien evoluu-



Kuva 6: Tyyppivarasto konfiguroinnin ja reflektion yhdyspisteenä [CB00].

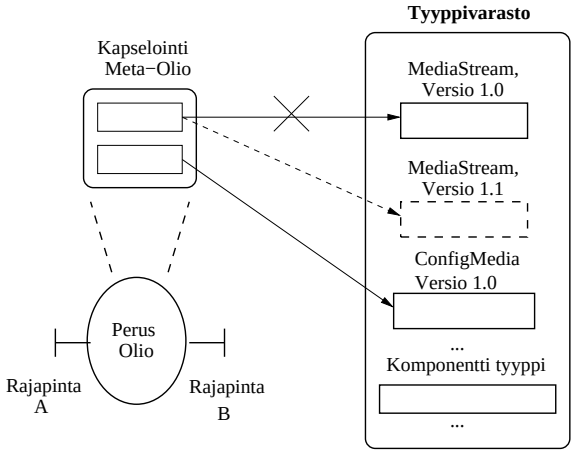


Kuva 7: Tyyppivaraston ja meta-olion suhde.

tion avulla. Tärkeää on huomata, että uudelleen konfiguroinnille voidaan asettaa rajoituksia joiden avulla konfigurointia voidaan hallita. Rajoitteet kuuluvat tyyppivarastossa olevaan tyyppiin. Rajoitukset takaavat arkkitehturaalisen yhtenäisyyden reflektion näkökulmasta. Rajoitteita voidaan luoda erillisillä politiikka-moduuleilla jokaista rajotetta kohden erikseen [Tri02].

5.1 Esimerkki 1: Olion rajapinnnan muuttaminen

Tämä on yksinkertainen esimerkki olion rajapinnnan muuttamisesta reflektiiviesti. Muuttamisessa käytetään hyväksi metatietoa, joka on talletettu tyyppivarastoon.



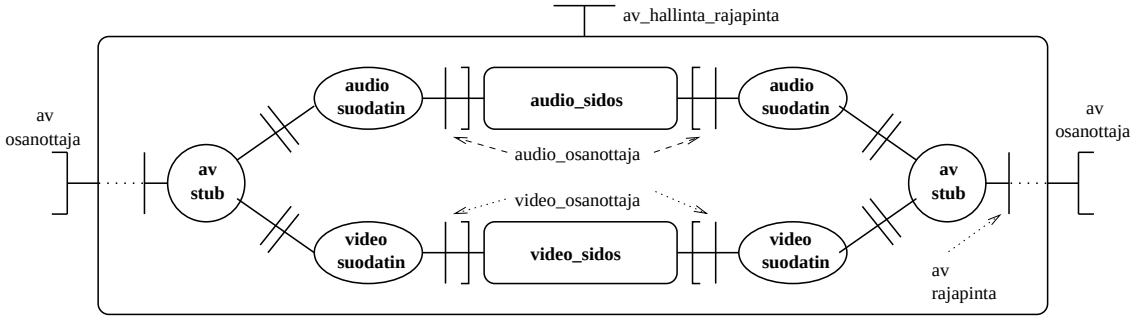
Kuva 8: Esimerkki reflektion toteuttamisesta kehittämällä tyyppiä [CB00].

Kuvassa 8 on perustason olio. Oliota reifioidaan kapselointi-metamallin avulla. Kuvan oliolla on kaksi rajapintaa. Rajapinnan A tyyppi on `MediaStream` ja rajapinnan B tyyppi on `ConfigMedia`. Kun kapselointi meta-olio luodaan tälle oliolle, sen kaksi rajapintatyyppiä etsitään tyyppivarastosta ja niiden vastaavia määrittelyjä käytetään meta-olioin tilan luomiseen. Näin meta-olio tarjoaa tietoa olion rajapinnoista. Esimerkiksi rajapintojen operaatiot ja ominaisuudet. Myöhemmin tietoa voidaan käyttää hyväksi vaikkapa lisäämällä rajapinnan A muuttamiseen siten että se tukee palvelun laadun merkintää operaatioissaan. Muuttaminen aiheuttaa uuden rajapinnan määrittelyn muodostamisen ja sen tallentamisen tyyppivarastoon. Kaikki tämänjälkeiset muutokset koskettavat uutta rajapintatyyppiä (`MediaStream 1.1`).

5.2 Esimerkki 2: Komponentin arkkitehtuurin muuttaminen

Ensimmäisen esimerkin kaltaisesti voidaan yhdistettyä komponenttia uudellen konfiguroida. Tässä tapauksessa metatiedon ja tyyppivarastossa pidetyn tiedon suhde on monimutkaisempi. Meta-olion täytyy metakuvauksen tyyppivarastosta noudettuaan tulki-

ta kuvausta ja saada selville konkreettinen oliograafi jonka avulla komponenttia reifioidaan.



Kuva 9: Useista olioista yhdistetty sidos-komponentti [CB00].

Sidos-komponentin meta-olion alustuksessa etsitään ensin sidoksen tyyppi tyyppivarastosta. Seuraavaksi meta-oliolle syötetään tieto sidoksen päätepisteistä (oliot, jotka sidos liittää yhteen). Tieto saadaan sidoksen hallintarajapinnasta. Yhdistämällä sidostyypin metatietoa ja tiedon sidoksen päätepisteistä voi meta-olio saada selville yksilölliset nimet kaikille sidoksen rajapinnoista. Meta-olio käyttää tähän samoja sääntöjä kuin sidoksen luova tehdas-oliokin. Nimien selville saamisen jälkeen meta-olio saa selville rajapintojen todelliset viitteet tekemällä nimipalveluhaun järjestelmän nimipalvelimeen. Lopuksi arkkitehtuuri-metamalliin kuuluva meta-olio luo sidoksen oliograafin yhdistämällä osittaiset oliograafit ja yhdistämällä edellisessä askeleessa saatudut rajapintaviitteet todellisiin rajapintoihin. Osittaiset oliograafit saadaan tyyppivarastosta kummankin päätepisteen roolimäärittelyiden avulla.

Sen jälkeen kun meta-olion alustaminen on saatu päätökseen meta-olio pystyy hallitsemaan sidoksen konfiguraatiota useasta kohdasta. Esimerkiksi meta-oliolta voidaan kysyä osaa tai kokonaisuutta sidoksen konfiguraatiosta. Vaihtoehtoisesti meta-oliota voidaan pyytää korvaamaan osa sidoksen konfiguraatiosta sellaisilla olioilla jotka osaavat erilaisen äänenkoodauksen. Huomattavaa on että sidoksen meta-olion muuttaminen aiheuttaa samanlaisen vaikutuksen tyyppivaraston tyyppeihin kuin ensimmäisesssä esimerkissä kuvassa 8 esitetään.

6 Yhteenveto

Väliohjelmistoissa metatieto voidaan jakaa erilaisille tasoille. MOF-malli jakaa metatiedon neljään tasoon. Nämä tasot ovat instanssi taso, tyypit, meta-taso ja metameta-taso. Tasojen avulla tiedon halkinta on helpompaa.

Väliohjelmistoissa reflektiivisyys tarkoittaa sekä väliohjelmiston itsensä sopeutumista muuttuvaan ympäristöön että sovellusten muuttumista. Reflektio onkin hyvin lähellä adaptoitumista. Väliohjelmistojen adaptoinnissa on syytä ottaa huomioon muutama asia. Nämä asiat ovat muutosta tarvitsevien olosuhteiden havainnointi, halutun muutoksen vaikutusten laskeminen, päätöksen tekeminen milloin muutos tehdään ja lopulta itse muutoksen tekeminen. Muutoksen aikana on syytä ottaa huomioon väliohjelmiston ja sitä käyttävien sovellusten tilan säilyminen konsistenttina.

Metatietoa hallitaan jakamalla se sopiviin metamalleihin. Koska metatiedolla on tarkoitus tukea reflektiota jaetaan se sopivasti sellaisiin metamalleihin, joiden avulle reflektiivisten ominaisuuksien tukeminen onnistuu helposti. Resurssi-metamallin avulla voidaan hallita sovellusten resurssien käyttöä ja palvelun laatua. Interceptio-metamallin avulla komponenttien käyttäytymistä voidaan muuttaa lisäämällä esi- tai jälkitoiminnallisuutta. Kapselointi-metamallin avulla olioiden rajapintoihin päästään käsiksi. Arkkitehtuuri-metamallin avulla voidaan muodostaa ja uudelleen konfiguroida monimutkaisia komponentteja, jotka koostuvat useammista olioista.

Yksi tapa yhdistää reflektiiviset ominaisuudet ja metatiedon hallinta on tyyppivaraston käyttäminen näiden kahden maailman yhdyspisteenä. Tällöin tyyppivaraston sisältämän metatiedon avulla voidaan muokata reflektiivisesti sovellusten toimintaa. Tyyppivaraston metatiedon evolutiivisella käyttämisellä voidaan muokata olemassaolevia tyyppejä aivan uudenlaisiksi tyypeiksi ja siten laajentaa käytössä olevan ohjelmiston käyttömahdollisuuksia. Komponentit luodaan aina tyyppivarastossa olevien perustyyppien mukaisiksi . Tämän jälkeen komponetti pystyy itse metatiedon avulla muuntumaan uudeksi tyypiksi.

Lähteet

BBI+00 Gordon S. Blair, Lynne Blair, Valerie Issarny, Petr Tuma, and Apostolos Zarras. The role of software architecture in constraining adaptation in component-based middleware platforms. In *Middleware*, pages 164–184, 2000.

BCA+01 Gordon S. Blair, Geoff Coulson, Anders Andersen, Lynne Blair, Michael Clarke, Fabio Costa, Hector Duran-Limon, Tom Fitzpatrick, Lee Johnston, Rui Moreira, Nikos Parlavantzas, and Katia Saikoski. The design and implementation of open orb 2, 2001.

BCB+02 Gordon S. Blair, Geoff Coulson, Lynne Blair, Hector Duran-Limon, Paul Grace, Rui Moreira, and Nikos Parlavantzas. Reflection, self-awareness and self-healing in openorb. 2002.

CB00 Fabio M. Costa and Gordon S. Blair. Integrating meta-information management and reflection in middleware. In *International Symposium on Distributed Objects and Applications*, pages 133–143, 2000.

CBC98 Fabio M. Costa, Gordon S. Blair, and Geoff Coulson. Experiments with reflective middleware. In *Proceedings of the ECOOP Workshop on Reflective Object-Oriented Programming and Systems (ROOPS '98)*, volume 1543 of *Lecture Notes in Computer Science (LNCS)*, pages 390–391, Brussels, Belgium, 1998. Springer Verlag.

DB00 H. Duran and G. Blair. A resource management framework for adaptive middleware, 2000.

KCB+00 Fabio Kon, Roy H. Campbell, Francisco J. Ballesteros, M. Dennis Mickunas, and Klara Nahrstedt. 2k: A distributed operating system for dynamic heterogeneous environments. 2000.

OMG99 OMG Object Management Group. The meta-object facility specification, 1999.

Tri02 Anand Tripathi. Challenges designing next-generation middleware systems, June 2002.

Ven02 Nalini Venkatasubramanian. Safe 'composability' of middleware services, June 2002.