Erno Mononen

Ortogonaalinen säilyvyys ja relaatiotietokannat

Tietojärjestelmätieteen
kandidaatintutkielma

6.3.2005

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Mononen, Erno Samuli

Ortogonaalinen säilyvyys, relaatiotietokannat. / Erno Mononen

Jyväskylä: Jyväskylän yliopisto, 2005.

63 s.

Kandidaatintutkielma

Tässä tutkielmassa selvitetään ortogonaalisen säilyvyyden kriteerit olio-relaationaalisessa järjestelmässä, esitellään olio- ja relaatiomallin eroavaisuudet sekä näistä eroavaisuuksista johtuvat mallien yhteensovittamisen ongelmat. Tutkielmassa myös tarkastellaan näiden ongelmien ratkaisuksi esitettyjen suunnittelu- ja toteutusmallien hyviä ja huonoja puolia.

AVAINSANAT: ortogonaalinen säilyvyys, näkymätön säilyvyys, olio-relaatiomuuntaja.

SISÄLLYSLUETTELO

1 JOHDANTO

Merkittävä osa etenkin yritysjärjestelmiksi luokiteltavista sovelluksista toteutetaan nykyään oliopohjaisilla ohjelmointikielillä, kuten Javalla, C++:lla, C#:lla tai Smalltalkilla, ja useimpien näiden sovellusten täytyy pystyä säilömään tieto pysyvästi. Säilyvyyden hoitamiseen on kolme eri korkean tason tapaa: porttikäytäväpohjainen olioiden säilöminen relaatiotietokantaan, olio-relaatiotietokanta ja oliotietokanta (Srinivasan ja Chang, 1997). Mahdollisuuksina on nähty myös säilyvyyden lisääminen ohjelmointikieleen ja integroidut tietokantajärjestelmät, mutta niiden ei katsota olevan käytännöllisesti (kaupallisesti) merkittäviä vaihtoehtoja (Carey ja DeWitt, 1996). Hiljattain kiinnostusta on herättänyt lisäksi tapa, jossa tieto elää vain keskusmuistissa ja ainoastaan tietoa muokkaavat komennot tallennetaan (Wuestefeld, 2003), mutta silläkään ei ole toistaiseksi jalansijaa etenkään yritysjärjestelmiä kehitettäessä.

Oliotietokantoihin kohdistuneista suurista odotuksista huolimatta oli niiden markkinaosuus vuonna 2002 edelleen vain prosentin verran (eWeek, 2003). Olio-relaatiotietokannat sen sijaan ovat kyllä varsin suosittuja, mutta niiden olio-ominaisuuksien käyttöä on rajoittanut standardien puute. Yritysjärjestelmissä yleisimmin käytetty tietovarasto onkin edelleen relaatiotietokanta (Hohmann, 2003 s. 150). Tutkielmassa tarkasteltavaksi ohjelmointialustaksi valittu Java2 Enterprise Edition (jäljempänä J2EE) on puolestaan suosittu vaihtoehto yritysjärjestelmien kehittämisessä, ja myös J2EE–sovellusten yleisin tietovarasto on relaatiotietokanta (Johnson, 2002 s. 255). Näiden lähtökohtaisesti kovin erilaisten teknologioiden yhteensovittaminen hidastaa sovelluksien kehittämistä merkittävästi: Roos (2003, s. 8) arvioi sovelluskehittäjien käyttävän 40-60 prosenttia ajastaan olioiden säilömiseen liittyvän ohjelmakoodin kirjoittamiseen, Brown ja Whitenack (1996) puolestaan esittävät 25-50 prosentin ohjelmakoodista liittyvän näiden eri mallien yhdistämiseen. Tällaisessa *olio-relaationaalisessa järjestelmässä*

olioiden säilömiseen liittyvän työmäärän johdosta varsinainen sovelluksen suunnittelu jää monesti taka-alalle (Evans, 2003 s. 107).

Näiden lukujen valossa ei ole kovin vaikea uskoa, että luvussa kaksi lähemmin tarkasteltavassa Atkinsonin (1999) ortogonaalisen säilyvyyden hypoteesissa on jotain perää. Ortogonaalisen säilyvyyden toteuttavia ratkaisuja on kehitetty Java–alustalle, kuten Pjama ja Gemstone/J (Marquez et al. 2001), mutta olio-relaationaalisessa järjestelmässä ortogonaalista säilyvyyttä ei ole tiettävästi ole täysimittaisesti saavutettu. Tällä hetkellä ei ole nähtävissä, että relaatiotietokannoista taikka olioteknologioista oltaisiin luopumassa (Ambler, 2002), joten näiden mallien yhdistäminen tulee olemaan myös vastaisuudessa sovelluskehittäjien haasteena. Näistä seikoista johtuen ortogonaalinen säilyvyys olio-relaationaalisessa järjestelmässä on varsin mielenkiintoinen tutkimuskohde.

1.1 Tutkimuksen tausta

Sovelluskehittäjän rooli olio-relationaalisessa järjestelmässä ei ole aina kiitollinen – itse liiketoimintaongelmien ratkaiseminen jää usein taka-alalle ajan huvetessa infrastruktuuriin liittyvien ongelmien ratkaisemiseksi. Turhauttavaa on myös pidättäytyminen eleganteista oliomalleista, joissa hyödynnetään täysipainoisesti olio-ohjelmoinnin tarjoamia mahdollisuuksia, koska niiden muuntaminen relaatiotietokantaan olisi liian työlästä. On helppo ajatella, että täysimittainen ortogonaalinen säilyvyys olio-relaationaalisessa järjestelmässä parantaisi sovelluskehittäjän tuottavuutta ja sovellusten ylläpidettävyyttä huomattavasti.

Viime aikoina Java–alustalle onkin kehitetty uusia standardeja ja sovelluskehyksiä helpottamaan sovelluskehittäjien työtä olio-relaationaalisessa järjestelmässä. Aiheeseen liittyviä standardeja ovat esimerkiksi osana J2EE–spesifikaatioon kuuluva Enterprise Java Beans Container Managed Persistence

(EJB CMP) -määritys (Sun Microsystems, 2001) ja Java Data Objects (jäljempänä JDO) (Sun Microsystems, 2003). Lisäksi avoimen lähdekoodin yhteisössä on kehitetty useita näkymättömään säilyvyyteen tähtääviä sovelluskehyksiä, kuten Object Relational Bridge (Apache Software Foundation, 2004), Castor (ExoLab Group, 2004) ja Hibernate (Hibernate, 2004). EJB CMP –määrityksen kaupallisten toteutusten lisäksi tarjolla on monia muita, edistyneempiä ominaisuuksia tarjoavia kaupallisia säilyvyyssovelluskehyksiä, kuten Oraclen Toplink (Oracle Corporation, 2004) ja Cocobase (Thought Inc., 2004). EJB:n kohdepohjaisten komponenttien luokkasuunnittelulle asettamat rajoitukset ovat osaltaan johtaneet olioperiaatteiden vastaisesti toteutettuihin sovelluksiin (Fowler, 2003), ja tällä hetkellä näyttäisikin siltä, että tavallisilla Java-olioilla (engl. Plain Ordinary Java Object, jäljempänä POJO) toimivat näkymättömän säilyvyyden takaavat ratkaisut ovat kasvattamassa suosiotaan EJB:n kustannuksella (Tate, 2004).

Näkymätön säilyvyys on askel kohti ortogonaalista säilyvyyttä, mutta kuten luvussa kaksi tulemme huomaamaan, se on kuitenkin vain osa ortogonaalisen säilyvyyden määritelmää.

1.2 Tutkimusongelma, tutkimusalueen rajaus ja tutkimuksen tulokset

Tämän tutkielman tutkimusongelma on: Mitkä ovat olio- ja relaatiomallien väliset yhteensopivuusongelmat? Lisäksi tutkimus pyrkii vastaamaan seuraaviin kysymyksiin:

- Mitkä ovat ortogonaalisen säilyvyyden vaatimukset olio-relaationaalisessa järjestelmässä?
- Millaisia suunnittelu- ja toteutusmalleja on esitetty mallien epäyhteensopivuudesta johtuvien ongelmien ratkaisemiseksi?

Tutkimuksen luvuissa 3, 4 ja 5 esiteltävät suunnittelumallit soveltuvat sellaisenaan useille eri ohjelmointialustoille, mutta ensisijaisesti tutkielmassa keskitytään Java –alustaan.

Tutkielma tuottaa myös olio-relaationaalisen järjestelmän ortogonaalisen säilyvyyden vaatimukset sekä esittää olio- ja relaatiomallin epäyhteensopivuuksien ratkaisumallit.

1.3 Tutkielman rakenne

Tutkielman luvussa kaksi rakennetaan ortogonaalisen säilyvyyden vaatimukset olio-relaationaaliselle järjestelmälle Atkinsonin (1999) esittämien vaatimusten pohjalta.

Luku kolme esittelee arkkitehtuuriset vaihtoehdot olio- ja relaatiomallien irtikytkemiseksi.

Luvussa neljä tutkitaan olio- ja relaatiomallin välisen muuntamisen rakennemalleja. Luvussa tarkastellaan yksityiskohtaisesti eri muunnostapoja, tuodaan esille kunkin eri muunnostavan edut ja haitat sekä arvioidaan eri muunnostapojen soveltuvuutta eri tilanteisiin.

Luku viisi syventyy olio- ja relaatiomallin identiteettien eroavaisuuksiin, inkrementaaliseen lataamiseen sekä transaktioiden tutkimiseen.

1.4 Tutkimusmetodit

Tutkimus suoritetaan pääosin kirjallisuuskatsauksena. Luvun seitsemän osalta tutkimusote on osin konstruktiivinen.

2 ARVIOINNIN VIITEKEHYS

Tässä luvussa esitellään ortogonaalisen säilyvyyden käsite ja kirjallisuudessa esitetyt vaatimukset tuotantotasoiselle ortogonaalisen säilyvyyden takaavalle järjestelmälle.

2.1 Ortogonaalinen säilyvyys

Atkinsonin esittämä *ortogonaalisen säilyvyyden* hypoteesi on seuraava: Jos sovelluskehittäjille taataan hyvin toteutettu ja hyvin tuettu ortogonaalisesti säilyvä ohjelmointialusta, niin siitä seuraa merkittävä kehittäjien tuottavuuden kasvu ja operationaalinen suorituskyky tulee olemaan tyydyttävä (Atkinson, 1999 s. 8). Ortogonaalinen säilyvyys puolestaan koostuu seuraavan kolmen periaatteen soveltamisesta (Atkinson ja Morrison, 1995):

- *Näkymättömän säilyvyyden periaate.* Ohjelman muoto on riippumaton sen käsittelemän tiedon pitkäikäisyydestä. Ohjelmat näyttävät samalta riippumatta siitä, käsittelevätkö ne lyhyt- vai pitkäikäistä tietoa.
- *Tietotyyppiriippumattomuuden periaate.* Kaikilla tieto-olioilla tulisi olla mahdollisuus täysimittaiseen säilyvyyteen niiden tyypistä riippumatta. Ei ole mitään erityistapauksia, joissa olio ei saa olla pitkäikäinen tai ohimenevä (engl. transient).
- *Säilyvyyden tunnistamisen periaate.* Valinta siitä, kuinka tunnistaa ja päästä käsiksi säilyviin olioihin on ortogonaalinen järjestelmän muille ominaisuuksille. Tunnistamisen mekanismi ei liity tyyppijärjestelmään.

Seuraavissa alakohdissa tarkastelemme kutakin periaatetta yksityiskohtaisemmin.

2.1.1 Näkymättömän säilyvyyden periaate

Näkymätön säilyvyys (engl. persistence independence) vapauttaa ohjelmoijan tiedon eri tallennusvälineiden välisen liikuttamisen taakasta ja ohjelmoimasta muunnoksia pitkäkestoisen ja lyhytkestoisen esitysten välillä (Atkinson ja Morrison, 1995). Näkymättömän säilyvyyden periaate tarkoittaa, että lähde- ja tavukoodiin ei pitäisi tarvita mitään muutoksia, jotta se toimisi pitkäkestoisella tiedolla (Atkinson, 1999 s. 3). Lisäksi kielen semantiikan täytyy säilyä muuttumattomana säilyvyyttä käytettäessä, esimerkiksi evaluointijärjestyksen, tyyppiturvallisuuden ja olioidentiteetin täytyy olla täsmälleen samoja koodille, joka operoi pitkäkestoisten olioiden kanssa, kuin ne ovat koodille, joka operoi ohimenevien olioiden kanssa (Atkinson, 1999 s. 3). Jos tämä vaatimus ei täyty, menetetään luotettavuus ja koodin uudelleenkäyttö, koska tuotua koodia täytyy muuttaa, jotta se toimisi säilötyn tiedon kanssa (Atkinson, 1999 s. 3). Näkymätön säilyvyys minimoi niiden seikkojen määrää, joita sovelluskehittäjät joutuvat opettelemaan ennen kuin he voivat soveltaa säilyvyyttä pitkäkestoisen tiedon luontiin (Atkinson, 1999 s. 3).

2.1.2 Tietotyyppiriippumattomuuden periaate

Tietotyyppiriippumattomuus on apukeino tietomallinnukseen siinä mielessä, että se varmistaa sen, että tietomalli voi olla täydellinen ja riippumaton mallinnettavan tiedon säilyvyydestä. Esimerkiksi joukkotietotyypit abstrahoivat koon suhteen, ja ovat siksi yleisesti käytettyjä säilyvyyden takaavissa ohjelmointikielissä helpottamaan massiivisten kokoelmien käsittelyä. Tietotyyppiriippumattomuus sisältää myös kielen suunnitteluperiaatteen tyyppijärjestelmän täydellisyydestä. Ohjelmoijan ei tulisi siis joutua kohtamaan tilannetta, jossa joukko tietyntyyppisiä elementtejä voidaan säilöä, mutta toisentyyppisiä elementtejä ei voida. (Atkinson ja Morrison, 1995).

2.1.3 Säilyvyyden tunnistamisen periaate

Säilötyn tiedon tunnistamiseksi on tutkittu monia keinoja. Jotkut edellyttävät säilyvyyden assosioimista säilymistilan varaajan tai muuttujan kanssa, tai sitten tiettyä tyyppiä luokan määritelmässä. Näkymättömän säilyvyyden periaatteen mukaisesti mikään näistä ei ole sallittu. Ne ovat myöskin sopimattomia muista syistä. (Atkinson ja Morrison, 1995)

Ohjelmointikielissä, joissa datan elinikä (engl. extent) voi poiketa sen näkyvyysalueesta (engl. scope), esiintyy jo suppea säilyvyys. Edellä mainitun kaltaisia olioita säilytetään yleensä keossa (engl. heap), josta ne ovat saatavilla niin kauan kuin sovellus ja suoritusympäristö sisältävät tarpeeksi tietoa niihin viittaamiseksi. Tätä suppeaa säilyvyyttä voidaan laajentaa siten, että olioiden sallitaan säilyä pysyvästi. Tekniikka, jota käytetään nykyään tämän toteuttamiseksi, on *tunnistaminen tavoitettavuuden perusteella*. Sen mukaan säilyvät oliot tunnistetaan sen perusteella, että ovatko ne tavoitettavissa säilöttävästä kantaoliosta (engl. persistent root) käsin. Analogia automaattisen roskienkeruun kanssa on ilmeinen. (Atkinson ja Morrison, 1995)

2.2 Ortogonaalisen säilyvyyden vaatimukset olio-relationaalisessa yritysjärjestelmässä

Yritysjärjestelmien vaatimukset poikkeavat monilta osin yhden käyttäjän sovelluksien vaatimuksista, eikä ortogonaalisen säilyvyyden periaatteiden toteuttaminen sellaisinaan takaa käyttökelpoista järjestelmää. Atkinson (1999) esittää vaatimukset tuotantotasoiselle ortogonaaliselle järjestelmälle Pjama–projektin yhteydessä. Pjaman tarkoituksena oli tuottaa ortogonaalisen säilyvyyden takaava ympäristö Java–alustan pohjalta, missä se suurilta osin onnistuikin (Atkinson ja Jordan, 1999), joskaan ei kaupallisessa mielessä. Pjama ei kuitenkaan ollut olio-relaationaalinen alusta, josta olemme tämän tutkimuksen puitteissa kiinnostuneita. Atkinsonin Pjamalle esittämiä vaatimuksia voidaan silti soveltaa myös olio-relaationaaliseen järjestelmään.

Tarkastelemme seuraavissa alakohdissa näitä kymmentä Atkinsonin esittämää vaatimusta yksityiskohtaisesti ja pohdimme samalla, millä tavoin ne ovat relevantteja Java–pohjaisessa olio-relaationaalisessa järjestelmässä.

2.2.1 Ortogonaalisuus (engl. orthogonality)

Kaikkien luokkien kaikilla ilmentymillä täytyy olla täysi oikeus säilyvyyteen (Atkinson, 1999). Olio-relaationaalisessa järjestelmässä ortogonaalisuuden toteutumisen voidaan katsoa olevan säilyvyyssovelluskehyksen tehtävä. Java–alustalla täysimittainen ortogonaalisuus tarkoittaa, että myös sellaisten luokkien ilmentymien, kuten `java.lang.Thread`, `java.lang.Class`, `java.lang.Throwable` ja graafisten käyttöliittymäkomponenttien tulisi voida olla säilöttyjä. Atkinson ja Jordan (1998) mainitsevat edellä lueteltujen luokkien ilmentymien säilömisen olevan erityisen haasteellista.

Javassa on pääasiassa kahdenlaisia tietotyyppejä, primitiivi- ja viitetyyppejä. Primitiivityypit eivät aiheuta suuria ongelmia säilyvyyden suhteen, mutta toisaalta viitetyypit, jotka ovat viitteitä luokista luotuihin olioihin ovat ongelmallisempia. Olion käyttäytyminen määrittyy suurelta osin oliota vastaavassa luokassa määriteltyjen metodien perusteella, joten olion tila on tiiviisti sidoksissa sen luokkaan. Tämä näyttäisi johtavan siihen, että luokan koodi säilötään sen ilmentymien kanssa. Niin muodoin tämä olion ja luokan välinen sidonta pitää säilyttää tietotyyppiriippumattomuuden tukemiseksi. (Lunney ja McCaughey, 2003)

2.2.2 Näkymätön säilyvyys (engl. persistence independence)

Kielen täytyy olla täysin muuttumaton (syntaksi, semantiikka, luokkakirjasto), jotta tuodut ohjelmat ja kirjastot toimivat oikein, riippumatta siitä, että tuodaanko ne lähde- vai tavukoodimuodossa (Atkinson, 1999). Olio-relaationaalisessa järjestelmässä näkymättömän säilyvyyden toteutuminen on enimmäkseen riippuvainen säilyvyyssovelluskehyksestä.

Java –alustalla näkymätön säilyvyys tarkoittaa sitä, että mitään lähdekoodia ei pitäisi täytyä muokata, jotta luokka tai sen ilmentymät voivat tulla säilötyksi (Lunney ja McCaughey, 2003). Mahdollisiksi muokkauksiksi lasketaan periytyminen erityisistä luokista, erityiset metodit jotka liittyvät säilömiseen tai erityiset konstruktorit (Lunney ja McCaughey, 2003). Atkinson ja Jordan (1998) listaavat vaatimukseksi myös sen, että mitään erityisiä metodikutsuja olion siirtämiseksi säiliöön ei tule tarvita.

Java –alusta määrittää julkisen formaatin käännetyille luokille, joka on siten avoin kääntämisen jälkeiselle prosessoinnille ja tarjoaa mahdollisuuden säilyvyyden tuomiseen. Tällä tavalla lähdekoodia ei tarvitse muokata, mikä on yksi ortogonaalisen säilyvyyden tavoitteista. Tuloksena saatavaa tavukoodia ei kuitenkaan voida hyödyntää nimenomaisen järjestelmän ulkopuolella, mikä eliminoi Java –alustan ”kirjoita kerran, aja kaikkialla” –päämäärän. Lisäksi tämä menetelmä vaatii jälkiprosessoijan jakelemisen sovelluksen kanssa ja jälkiprosessoitujen luokkien lataamisen muokatulla luokanlataajalla (engl. class loader), mikä rikkoo näkymättömän säilyvyyden periaatetta. (Lunney ja McCaughey, 2003)

2.2.3 Pysyvyys (engl. durability)

Sovelluskehittäjien täytyy voida luottaa siihen, että järjestelmä ei kadota tietoja (Atkinson, 1999). Tiedon pitkäaikainen pysyvyys on olio-relaationaalisessa järjestelmässä pääasiallisesti relaatiotietokannan tehtävä. Tämän tutkimuksen kannalta pysyvyys ei siten ole erityisen kiinnostava kriteeri; relaatiotietokannat edustavat pitkään käytössä ollutta ja luotettavaksi todettua teknologiaa, tietojen pysyvyyden niissä voidaan tämän tutkimuksen puitteissa katsoa olevan ilmeistä.

2.2.4 Skaalautuvuus (engl. scalability)

Kehittäjien ei tulisi kohdata rajoitteita, jotka estävät heidän sovelluksiensa toimimista (Atkinson, 1999). Olio-relaationaalisesssa järjestelmässä skaalautuvuus liittyy sekä säilyvyyssovelluskehykseen että relaatiotietokantaan. Tässä tutkimuksessa relaatiotietokantaan liittyvät skaalautumisseikat ohitetaan kuitenkin aiheen laajuuden vuoksi. Todettakoon, että olio-ohjelmointikielillä toteutettujen järjestelmien säilyvyysratkaisuista relaatiotietokannan on katsottu olevan parhaiten skaalautuva (Carey ja DeWitt, 1996).

Olio-relaatiomuuntajan skaalautuminen on yhtä lailla laaja aihe, eikä siihenkään voida tutkimuksen puitteissa perehtyä syvällisesti. Sovelluskehyksien arvioinnissa tuomme kuitenkin esille muutaman skaalautuvuuteen vaikuttavan seikan: välimuististrategiat, ryvästyksen sekä lukituksen väliohjelmistossa.

2.2.5 Kaavan muuttuminen (engl. schema evolution)

Järjestelmän täytyy tukea mahdollisuutta tehdä muutoksia minkä tahansa luokan määritelmään sekä sitä vastaaviin ilmentymiin (Atkinson, 1999). Meyer (1997, s. 1041-1042) esittää kolme lähestymistapaa kaavan muuttumisesta aiheutuvien ongelmien ratkaisemiseksi: vanhojen olioiden hylkääminen tyystin, migraatiopolku vanhasta mallista uuteen mallin sekä vanhojen olioiden lennosta muuntaminen. Meyer hylkää perustellusti kaksi ensin mainittua tapaa käyttökelvottomina jo käytössä oleville sovelluksille, joten arvioinnissa tulemme kiinnittämään huomiota sovelluskehyksien kykyyn suoriutua vanhojen olioiden muuntamisesta.

2.2.6 Alustan muuttaminen (engl. platform migration)

Olemassa olevien sovellusten ja niiden tietojen siirtäminen uudelle alustalle (uusi kone, käyttöjärjestelmä tai JVM) täytyy olla mahdollista (Atkinson, 1999).

Olio-relaationaalisessa järjestelmässä tätä kriteeriä voidaan lisäksi soveltaa siten, että säilyvyyssovelluskehyksen tulisi toimia useilla virtuaalikoneilla sekä useampien eri valmistajien relaatiokantojen kanssa.

2.2.7 Jatkuvuus (engl. endurance)

Alustan täytyy tukea jatkuvaa toimintaa (Atkinson, 1999). Jatkuvan toiminnan voidaan ajatella muodostuvan seuraavista seikoista, jotka ovat tulleet esille jo aiemmissakin kohdissa:

- Ryvästäminen. Yhdellä koneella toimiva järjestelmä on alttiimpi käyttöhäiriöille kuin useammalla koneella toimiva.
- Kaavan muuttuminen. Muutosten ei tulisi aiheuttaa pitkiä käyttökatkoja taikka tietojen menettämistä.

2.2.8 Avoimuus (engl. openness)

Sovellusten kytkeminen muihin järjestelmiin täytyy olla mahdollista (Atkinson, 1999). Java –alusta sinänsä tarjoaa erilaisia mahdollisuuksia muihin järjestelmiin kytkeytymiseksi, kuten J2EE Connector Architecture (JCA), Java Message Service (JMS) ja Java Database Connectivity (JDBC) (Kassem ym., 2000 luku 6). On luontevaa edellyttää myös olio-relaationaaliselta järjestelmältä tukea näille tekniikoille. Lisäksi vaatimusta voidaan tarkentaa siten, että olio-relaatiomuuntajan tulisi tukea useamman tietovaraston yhtäaikaista hyödyntämistä.

2.2.9 Transaktionaalisuus (engl. transactional)

Transaktionaalinen toiminta on edellytys pysyvyyden ja yhtäaikaisuuden yhdistämiselle (Atkinson, 1999). Nock (2003, s. 376-377) esittelee neljä eri lähestymistapaa olio-relaationaalisen järjestelmän transaktionaalisuuden hoitamiseen, joita tulemme käyttämään pohjana sovelluskehyksien

arvioinnissa: transaktio, optimistinen lukitseminen, pessimistinen lukitseminen ja täydentävä transaktio. Tarkastelemme näitä malleja yksityiskohtaisemmin tämän tutkielman luvussa 5.

2.2.10 Suorituskyky (engl. performance)

Suorituskyvyn täytyy olla hyväksyttävää (Atkinson, 1999). Olio-relaationaalisessa yritysjärjestelmässä suorituskyky koostuu useasta eri osa-alueesta. Tämän tutkimuksen kannalta kiinnostavimpia ovat säilyvyyssovelluskehykseen liittyvät seikat:

- Muuntamisominaisuudet. Oliomalli voidaan muuntaa moneksi erilaisiksi relaatiomalleiksi, joiden suorituskyky ja ylläpidettävyys voivat poiketa huomattavasti. Eri muuntamisvaihtoehtojen monipuolinen tukeminen on täten toivottavaa. Muuntamisominaisuuksiin kuuluvat luokkahierarkian, olioiden välisten suhteiden ja luokkakohtaisten jäsenmuuttujien muuntaminen. Olio- ja relaatiomallin väliseen muuntamiseen liittyviä seikkoja tarkastellaan yksityiskohtaisemmin luvussa 4.
- Välimuisti. Turhien tietokantaoperaatioiden eliminoiminen on yksi tehokkaimmista tavoista parantaa suorituskykyä (Nock, 2003 s. 267). Olio-relaationaalisessa järjestelmässä välimuistitusta voi esiintyä monella eri tasolla: relaatiotietokannassa, tietokannan ajurissa sekä usealla eri tasolla säilyvyyssovelluskehyksessä. Kuten aiemmin mainittiin, tutkimuksen kannalta kiinnostavaa on säilyvyyssovelluskehyksen sisällä esiintyvä välimuistitus.
- Kyselyominaisuudet. Tiedon saantiin liittyvät operaatiot voivat helposti olla koko yritysjärjestelmän eniten aikaa kuluttavia (Nock, 2003 s. xxiv), joten olio-relaationaalisessa järjestelmässä on tärkeää pystyä hakemaan ja muokkaamaan säilöttyjä olioita tehokkaasti - yleensä tietokantakyselyjen määrä pyritään minimoimaan. Luonnollinen

vertailukohde olio-relaationaalisen järjestelmän kyselyominaisuuksille ovat SQL:n tarjoamat ominaisuudet.

- Inkrementaalinen lataaminen (Atkinson ja Morrison, 1995). Inkrementaalinen lataaminen voidaan toteuttaa usealla eri tapaa, joiden vaikutus suorituskykyyn vaihtelee. Inkrementaalisen lataamisen toteutustavoista enemmän luvussa 5.

2.2.11 Muita vaatimuksia

Atkinsonin esittämien vaatimusten lisäksi voidaan olio-relaationaalisen järjestelmän arvioinnissa ottaa huomioon joitain erityisseikkoja. Cabibbo ja Porcelli (2004) esittelevät kolme eri tapaa olio-relaationaalisen järjestelmän kehittämiseen: relaatiomallin luominen oliomallin pohjalta, oliomallin luominen relaatiomallin pohjalta sekä olio- ja relaatiomallin itsenäinen kehittäminen (engl. meet-in-the-middle). Olio-relaationaalinen järjestelmä saatetaan kehittää jo olemassa olevan relaatiokannan päälle, minkä johdosta tietokannan rakennetta ei välttämättä voida muokata sovelluksen ehdoilla (Cabibbo ja Porcelli, 2004). Näin ollen yksi arvioinnissa käytettävä kriteeri on, miten eri kehittämismalleja tuetaan. Tähän liittyy se, vaaditaanko tietokantaan erityisiä sarakkeita taikka tauluja sekä myös kohdassa 2.2.10 mainitut muuntamisominaisuudet – jos tietokantarakenne on annettu, ei säilyvyyssovelluskehyksen tulisi vaatia luonnotonta oliomallia toimiakseen.

Toinen merkittävä seikka on järjestelmän tuki olion identiteetille. Kuten tulemme luvussa 5 huomamaan, olion identiteetti poikkeaa olio- ja relaatiomallissa huomattavasti toisistaan, ja toimivan tunnistemekanismin puuttuessa saattaa sovelluskehitys vaikeutua selkeästi. Tämän johdosta säilyvyyssovelluskehykseltä voidaan vaatia tukea käyttäjän kannalta läpinäkyvälle olion identiteetille.

2.3 Yhteenveto

Tässä luvussa esittelimme ortogonaalisen säilyvyyden käsitteen ja tarkastelimme tuotantotasoiselle ortogonaalisen säilyvyyden takaavalle järjestelmälle esitettyjä vaatimuksia. Pohdimme samalla, kuinka nämä esitetyt vaatimukset voidaan huomioida olio-relaationaalisessa järjestelmässä. Jäljempänä tässä tutkielmassa tulemme arvioimaan sovelluskehyksiä sekä suunnittelumalleja näiden vaatimusten pohjalta.

3 OLIO- JA RELAATIOMALLIN VÄLINEN IRTIKYTKENTÄ

Tässä luvussa tarkastelemme olio- ja relaatiomallin väliseen irtikytkentään (engl. decoupling) esitettyjä suunnittelu- ja toteutusmalleja. Tutkielman kannalta mielenkiintoisin näistä malleista on kohdassa 3.4 käsiteltävä olio-relaatiomuuntaja, jolla saavutetaan periaatteessa täydellinen irtikytkentä olio- ja relaatiomallin välillä.

Yksi yleisimmistä yritysjärjestelmien kehittämisessä käytetyistä arkkitehtuureista on alijärjestelmien looginen ja fyysinen sovittaminen kerroksiin (Hohmann, 2003 s. 147). Laajalti käytetty kerrosarkkitehtuuri koostuu neljästä sovelluskerroksesta, jotka ovat esitystapa-, sovellus-, liiketoiminta- ja infrastruktuurikerros (Wirfs-Brock ja McKean, 2003 s.29). Tässä kerrosjaossa liiketoimintakerros sisältää sovelluksen liiketoimintaluokat, jotka puolestaan sisältävät säilöttävän datan, kun taas infrastruktuurikerroksessa sijaitsevat sovelluksen käyttämät tietovarastot, kuten relaatiotietokanta. Olio-relaationaalisessa yritysjärjestelmässä olio- ja relaatiomallin välinen irtikytkentä on täten oleellisinta liiketoimintakerroksen ja infrastruktuurikerroksen välillä, sillä muissa kerroksissa olevien olioiden ei yleensä tarvitse olla säilöttyjä.

Nock (2003) esittää liiketoiminta- ja tietovarastokerroksen väliseen irtikytkentään kolme mallia: porttikäytävä, toiminnallinen tietue ja olio-relaatiomuuntaja. Näiden lisäksi mielenkiintoinen irtikytkentämalli on Evansin (2003, s. 106-114) esittämä oliosäilö (engl. repository). Seuraavissa alakohdissa syvennytään kuhunkin eri vaihtoehtoon ja tutkitaan, kuinka ne suhteutuvat luvussa kaksi esitettyihin vaatimuksiin.

3.1 Porttikäytävä

Porttikäytävätyyppinen (engl. gateway) liiketoimintaluokkien ja relaatiomallin välinen irtikytkentä koteloi säilötyn tiedon käsittelyyn liittyvät yksityiskohdat yhteen komponenttiin, paljastaen vain loogiset operaatiot. Sovelluksen

ohjelmakoodissa säilyy kytkentä alla olevaan tietomalliin, mutta se on irtikytketty säilötyn tiedon käsittelyyn liittyvistä tehtävistä. (Nock, 2003 s.9)

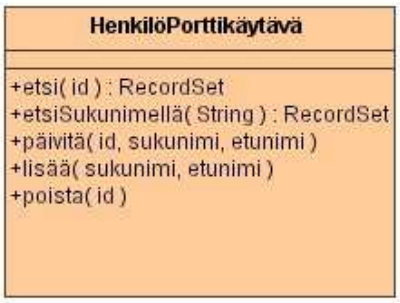
Kirjallisuudessa on esitelty eri nimillä useita malleja, jotka ovat suurelta osin vastaavia Nockin mallin kanssa. Sun Microsystemsin (2002) Data Access Object (DAO) -suunnittelumalli on käytännössä sama asia kuin Nockin porttikäytävä. Fowler (2002) esittelee irtikytkentään kaksi erilaista porttikäytäväpohjaista mallia. Seuraavissa alakohdissa tarkastelemme näitä lähemmin.

3.1.1 Rivikohtainen porttikäytävä

Rivikohtainen porttikäytävä on porttikäytävä –suunnittelumallin mukainen ratkaisu säilöntälogiikan toteuttamiseen oliopohjaisissa järjestelmissä. Ideana on, että kukin (säilötty) olio vastaa yhden taulun yhtä riviä tietokannassa. Tällä oliolla on tietokannan taulun sarakkeita vastaavat attribuutit sekä metodit tallentamista, poistamista, päivittämistä ja hakuja varten. (Fowler, 2002 s. 152)

3.1.2 Taulukohtainen porttikäytävä

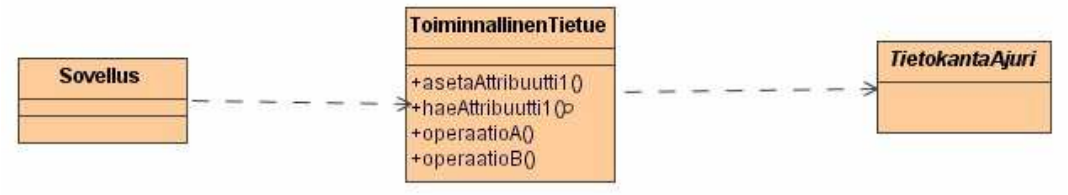
Taulukohtainen porttikäytävä on toinen porttikäytävään perustuva arkkitehtuurinen vaihtoehto irtikytkennälle. Erona rivikohtaiseen porttikäytävään on se, että taulukohtainen porttikäytävä vastaa yhden rivin sijasta koko tietokannan taulua. Se pitää sisällään kaikki SQL –lauseet yksittäisen taulun tai näkymän katsomiseen, poistamiseen, päivittämiseen ja lisäämiseen. Muut oliot kutsuvat porttikäytävän metodeita tietokannan ja sovelluksen väliseen vuorovaikutukseen. (Fowler, 2002 s. 144)



KUVIO 1. Taulukohtainen porttikäytävä (Fowler, 2002 s. 144)

3.2 Toiminnallinen tietue

Toiminnallinen tietue (engl. active record) koteloi tietomallin ja tiedon käsittelemisen yksityiskohdat relevantteihin liiketoimintaolioiden toteutuksiin. Toiminnallinen tietue vapauttaa sovelluskoodin kaikesta suorasta vuorovaikutuksesta tietokannan kanssa. (Nock, 2003 s. 33).



KUVIO 2. Toiminnallinen tietue -suunnittelumallin staattinen rakenne (Nock, 2003 s.38).

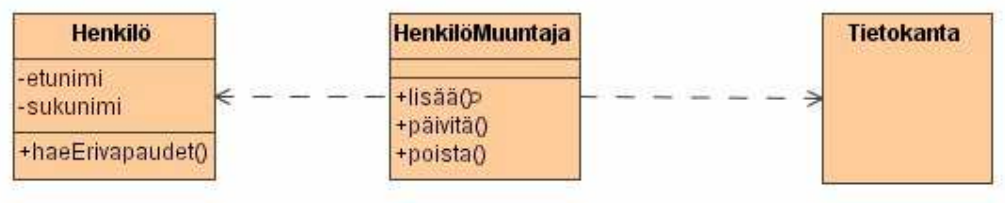
Toiminnallisella tietueella saavutetaan siistimpi sovelluskoodi, sillä pelkästään liiketoimintaolioita käyttävä sovelluskoodi on selkeämpää ja helpompaa ylläpitää kuin koodi, joka käsittelee sekä tietomallin että tiedonsaannin yksityiskohtia. Sovelluskoodi on eriytetty tietomallista, minkä johdosta tietomallin muuttaminen on vaivattomampaa. Lisäksi toiminnallinen tietue ryhmittää toisiinsa liittyvät tiedonmuokkausoperaatiot yhteen komponenttiin,

joten näiden operaatioiden puutteiden tunnistaminen ja korjaaminen on helpompaa. (Nock, 2003 s. 40)

Haittapuolena toiminnallisessa tietueessa on tiedonmuokkausoperaatioiden hajautuminen useaan liiketoimintaolioon, mistä johtuen kokonaisvaltaisten tiedonmuokkausstrategioiden, kuten lauseiden (engl. statement) välimuistittaminen, vaatii toistuvaa koodia jokaisessa liiketoimintaolion toteutuksessa (Nock, 2003 s. 40). Samoin kuin porttikäytävämallissa, sovelluskoodin tiedonmuokkaus on rajoitettu liiketoimintaolion operaatioihin (Nock, 2003 s. 40). Toiminnallinen tietue myös kytkee oliomallin tiiviisti relaatiomalliin, joten sekä olio- että relaatiomallin muuttaminen on vaikeaa projektin edetessä (Fowler, 2002 s.162). Toiminnallinen tietue on hyvä valinta silloin, kun liiketoimintalogiikka ei ole liian monimutkaista, vaan muodostuu lähinnä olioiden luomisesta, lukemisesta, päivittämisestä ja poistamisesta (Fowler, 2002 s. 161).

3.3 Olio-relaatiomuuntaja

Olio-relaatiomuuntaja koteloi liiketoimintaolioiden ja relaationaalisen datan välisen muunnoksen yhteen komponenttiin. Olio-relaatiomuuntaja irtikytkee sekä sovelluskoodiin että liiketoimintaoliot alla olevasta tietomallista ja tiedonsaannin yksityiskohdista. Sitä sovellettaessa vastuu oliopiirteiden ja relaationaalisen datan välisestä muuntamisesta tulee erilliselle komponentille, jota voidaan muuttaa sovelluksesta ja liiketoimintaolioista riippumatta. Olio-relaatiomuuntaja on yleisesti määritetty käyttäen abstraktiota, joka piilottaa kaikki muuntamisen yksityiskohdat sovelluksilta. (Nock, 2003 s.53-55)



KUVIO 3. Oliorelaatiomuuntaja (Fowler, 2002 s. 165)

Olio-relaatiomuuntaja on vastuussa liiketoimintaolioiden ja relaationaalisen datan välisestä muuntamisesta ja käyttää yleensä suoraan fyysistä tietokanta-ajuria. Yleinen analogia oliokäsitteiden ja relaatiomallin välillä on seuraava (Nock, 2003 s.56):

TAULUKKO 1. Olio- ja relaatiokäsitteiden välinen analogia

Oliokäsite	**Relaatiotietokannan käsite**
Luokka	Taulu
Olio	Rivi
Attribuutti	Sarake

Vastaavuus ei kuitenkaan useimmiten ole näin suoraviivaista, kuten tässä tutkielmassa tullaan jäljempänä huomaamaan.

Olio-relaatiomuuntaja tarjoaa seuraavat edut:

- Fyysinen tietomalli ja tiedonsaannin monimutkaisuus on piilotettu sovelluslogiikalta ja liiketoimintaolioilta. Tämän johdosta nämä komponentit pysyvät paremmin keskittyneinä liiketoimintaprosessien mallintamiseen (Nock, 2003 s. 58)
- Tietomalliin voidaan tehdä muutoksia koskematta sovelluskoodiin tai liiketoimintaolioiden määrittelyihin (Nock, 2003 s.58)

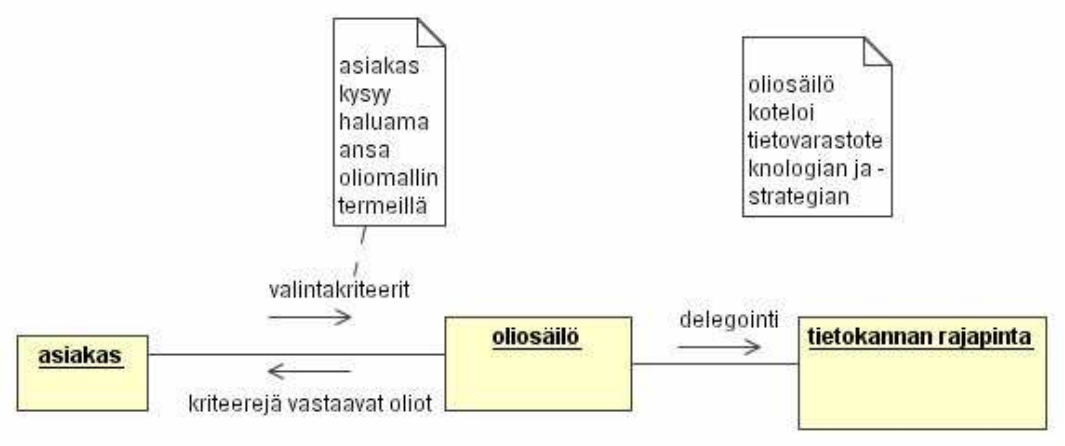
- Mahdollistaa liiketoimintaolioiden käytön useiden tietomallien kanssa, ilman että liiketoimintaolioita tarvitsi muuttaa (Nock, 2003 s. 58)
- Monimutkaisten liiketoimintaoliomallien yhteydessä tietokantariippuvainen toiminnallisuuden irtikytkentä on arvokasta (Fowler, 2002 s. 170-171).

Olio-relaatiomuuntaja tietää yhtä uutta sovelluskerrosta sovellukseen, mikä saattaa olla tarpeeton kyseen ollessa suhteellisen yksinkertaisesta liiketoiminnallisuudesta (Fowler, 2002 s. 170).

3.4 Oliosäilö

Evans (2003, s.106-114) tarjoaa liiketoiminta- ja tietovarastokerrosten väliseen irtikytkentään ratkaisuksi mallia, jossa olioihin päästään käsiksi pyytämällä niitä halutuilla kriteereillä oliosäilöstä, joka koteloi sisäänsä olioiden tietovarastosta hakemisen yksityiskohdat. Tällainen oliosäilö käyttäytyy kuten kokoelma, mutta tarjoten kehittyneemmät kyselyominaisuudet (Evans, 2003 s.108). Evansin mallissa ideana on, että säilöstä päästään käsiksi oliomallin kantaolioihin, joista voidaan navigoida oliomallin muihin olioihin. Evansin kantaolio (Evans, 2003 s. 89) on pitkälle yhteneväinen Atkinsonin ja Morrisonin (1995) kantaolion kanssa.

Oliosäilöstä saatavilla olevia olioita haetaan olioiden attribuuttien arvoihin perustuvien kyselyjen avulla (Evans, 2003 s.107). Joustava ja elegantti tapa kyselyjen muodostamiseen on *spesifikaation* käyttäminen (Evans, 2003 s.109-110). Kuviossa 4 on kuva toimintaperiaatteesta.



KUVIO 4. Oliosäilön toimintaperiaate (Evans, 2003 s.108).

3.5 Yhteenveto

Ortogonaalisen säilyvyyden kannalta tässä luvussa irtikytkennälle esitetyt mallit poikkeavat toisistaan merkittävästi. Porttikäytäväpohjaisilla malleilla on vaikea saavuttaa ortogonaalisen säilyvyyden vaatimuksia, koska niitä käytettäessä säilyy kytkentä alla olevaan tietomalliin. Toiminnallista tietuetta sovellettaessa taasen sovelluskoodi ei täytä näkymättömän säilyvyyden vaatimuksia, koska luokkiin joudutaan lisäämään säilömiseen liittyvää ohjelmakoodia. Olio-relaatiomuuntaja sen sijaan tarjoaa edellytykset ortogonaalisen säilyvyyden vaatimusten täyttämiselle, se ei periaatteessa vaadi mitään säilömiseen liittyvää koodia sovellusluokkiin ja sen avulla voidaan myös saavuttaa täydellinen irtikytkentä olio- ja relaatiomallin välillä. Toisaalta se on esitellyistä malleista selkeästi monimutkaisin toteutuksen kannalta.

Kohdassa 3.4 esitelty oliosäilö vaatii olioiden säilömistä varten jonkin mekanismin. Kuten yllä todettiin, ortogonaaliseen säilyvyyteen pyrittäessä esitellyistä vaihtoehdoista kyseeseen tulee olio-relaatiomuuntaja. Oliosäilön ja olio-relaatiomuuntajan yhdistelmä mahdollistaa olio- ja relaatiomallin irtikytkemisen sekä tavan päästä käsiksi säilöttyihin kantaolioihin.

4 OLIO- JA RELAATIOMALLIN VÄLISEEN MUUNTAMISEEN LIITTYVÄT RAKENNEMALLIT

Tässä luvussa tarkastellaan, kuinka oliomallille ominaiset piirteet, kuten periytyminen ja koostaminen, voidaan sovittaa relaatiomalliin. Kunkin eri ratkaisuvaihtoehdon kohdalla tutkitaan sen heikkouksia ja vahvuuksia sekä soveltuvuutta eri tilanteisiin. Luvun tarkoituksena on osoittaa, että oliomalli voidaan eroavaisuuksista huolimatta muuntaa relaatiotietokantaan. Luvussa esitettävät muunnostavat toimivat myös pohjana eri säilyvyyssovelluskehyksien arviointia varten – eri muunnostavat soveltuvat eri tilanteisiin, joten muunnostapojen monipuolinen tukeminen on hyvin toivottava ominaisuus säilyvyyssovelluskehykselle.

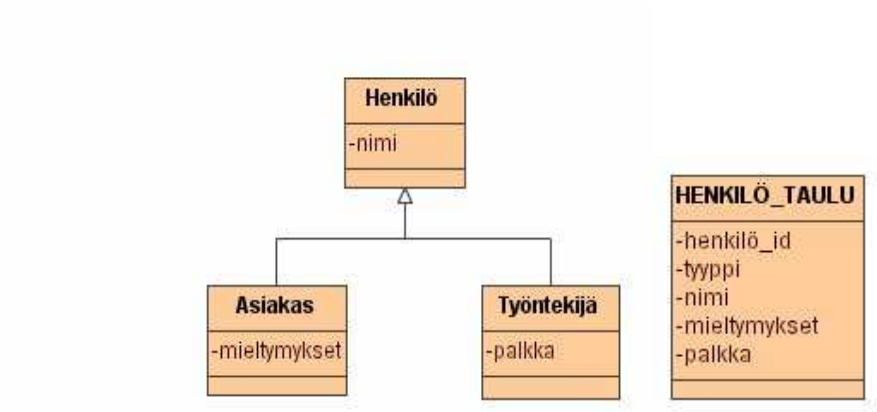
4.1 Periytyminen

Relaatiotietokannat eivät tue attribuuttien periytymistä. On mahdotonta tehdä todellista yhdestä yhteen -muunnosta taulun ja luokan välillä, kun luokka perii attribuutteja joltain toiselta luokalta tai jos muut luokat periytyvät siitä. (Brown ja Whitenack, 1996)

Jo yksinkertainenkin periytymishierarkia riittää siis muuttamaan taulukossa 1 esitellyt yleiset analogiat olio- ja relaatiokäsitteiden välillä. Periytymisen muuntamista relaatiotietokantaan on käsitelty kirjallisuudessa kiitettävästi ja ratkaisuksi on esitetty monia eri tapoja. Seuraavissa alakohdissa esitetään viisi eri tapaa luokkahierarkian muuntamiseksi relaatiokantaan, kukin omana kohtanaan, mutta on huomioitava, että käytännössä näitä malleja voidaan käyttää yhdessä - ei vain eri luokkahierarkioissa vaan myös yhden luokkahierarkian sisällä. Kunkin alakohdan yhteydessä pohditaan, mitä etuja ja haittoja kyseisestä muunnostavasta on, ja missä tilanteissa se on soveltuva. Lopuksi esitetään taulukkomuotoinen yhteenveto periytymisen eri muunnostavoista.

4.1.1 Periytymishierarkia yhdessä taulussa

Keskeinen idea yhden taulun periytymishierarkiassa on se, että kaikki hierarkian luokkien jäsenmuuttujat sijoitetaan samaan tauluun siten, että kutakin luokkahierarkiassa spesifiä attribuuttia vastaa taulussa sarake. Kukin hierarkian luokka vastaa yhtä riviä tässä taulussa; kullekin luokalle tarpeettomat sarakkeet jätetään tyhjiksi sillä rivillä. (Fowler, 2002 s.278). Eri luokkien tunnistamiseksi toisistaan tarvitaan tauluun myös tyypin ilmaiseva sarake, joka voi olla esimerkiksi luokan nimi (Fowler, 2002 s.278), tai useampi boolean–tyyppinen sarake (Ambler, 2003). Jälkimmäisellä mallilla on se etu, että yksi rivi voi olla useampaa tyyppiä ilman, että tarvitsee muodostaa uusi nimi tyypille jokaista yhdistelmää kohden (Ambler, 2003).



KUVIO 5. Perintähierarkia yhdessä taulussa (Ambler, 2003)

Yhden taulun periytymishierarkiassa voidaan katsoa olevan seuraavat vahvuudet:

- Tietokannassa on vain yksi taulu huolehdittavana (Fowler, 2002 s. 279)

- Tiedon hakemisessa ei tarvita liitoksia (Fowler, 2002 s. 279). Tästä johtuen tiedon hakeminen on yleensä nopeaa, jos taulun koko ei ole kohtuuttoman suuri.
- Jäsenmuuttujien siirtäminen luokkahierarkiassa ylös- tai alaspäin ei aiheuta muutoksia tietokantaan (Fowler, 2002 s. 279).
- Uusien luokkien lisääminen on helppoa, tarvitsee vain lisätä tarvittavat sarakkeet tauluun (Ambler, 2003)
- Tukee monimuotoisuutta yksinkertaisesti vaihtamalla rivin tyyppiä (tyyppisarakkeen arvoa) (Ambler, 2003)
- Ad-hoc -raportointi on helppoa, koska kaikki tieto löytyy yhdestä taulusta (Ambler, 2003)

Vastaavasti yhden taulun periytymishierarkiaan voidaan katsoa liittyvän seuraavat heikkoudet:

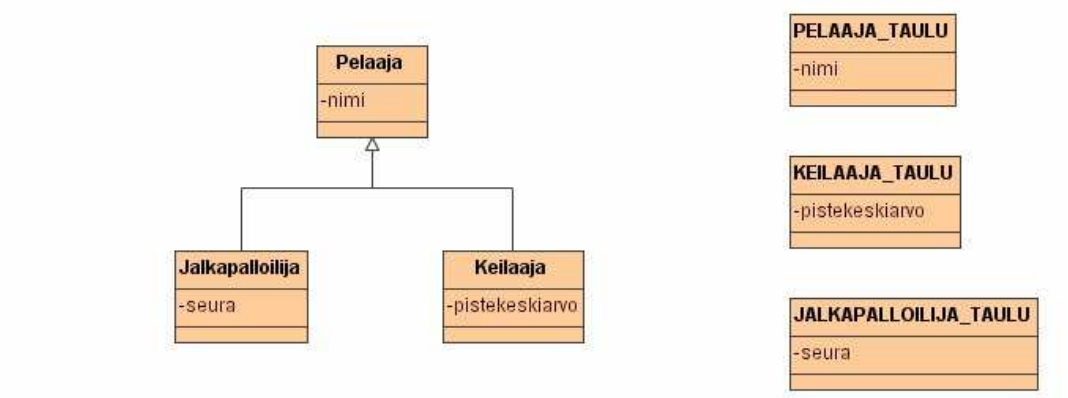
- Taulun sarakkeet ovat joissain tapauksissa merkityksellisiä ja joissain eivät, mikä voi olla hämmentävää taulua suoraan käyttäville (Fowler, 2002 s.279)
- Vain joidenkin aliluokkien käyttämät sarakkeet jäävät tyhjäksi muiden luokkien tapauksessa, mistä aiheutuu hukattua tilaa tietokannassa. Kuinka suuri ongelma tämä käytännössä on riippuu siitä, miten hyvin käytettävä tietokannan hallintajärjestelmä osaa tiivistää (pakata) hukatun tilan (Fowler, 2002 s. 279)
- Taulu saattaa kasvaa liian isoksi, millä voi olla negatiivinen vaikutus suorituskykyyn (Fowler, 2002 s. 279)
- Sarakkeiden nimeämisessä on käytettävissä vain yksi nimiavaruus (engl. namespace) (Fowler, 2002 s. 279)
- Kytkentä luokkahierarkian sisällä kasvaa, koska kaikki luokat ovat suoraan sidottuja samaan tauluun. Muutos yhdessä luokassa voi

johtaa muutokseen taulussa, mikä taas voi vaikuttaa luokkahierarkian muihin luokkiin (Ambler, 2003).

- Tyypin (siis luokan tyypin) ilmaisemisesta tulee hankalaa, kun tyyppien välillä on merkittävää limittymistä (Ambler, 2003).
- Lisäksi, koska kaikki taulun sarakkeet eivät ole relevantteja kaikille luokille, ei voida täysimittaisesti hyödyntää kannan rajoitteita. Esimerkiksi NOT NULL –rajoitteen käyttäminen ei ole mahdollista, jos kyseenomainen sarake/attribuutti ei esiinny kaikissa luokkahierarkian luokissa.

4.1.2 Taulu luokkaa kohden

Tässä luokkahierarkian muunnostavassa luodaan taulu jokaista hierarkian luokkaa kohden siten, että myös mahdollisesti abstrakteille yliluokille on omat taulunsa (vertaa 4.1.3 Taulu konkreettista luokkaa kohden). Kukin luokan attribuutti vastaa saraketta luokkaa vastaavassa taulussa.



KUVIO 6. Taulu luokkaa kohden muunnostapa (Fowler, 2002 s. 285)

Aliluokka koostuu siis sekä sen yliluokan taulujen sarakkeista sekä oman taulunsa sarakkeista. Tästä seuraa, että näiden taulujen rivit on voitava yhdistää jollain tavalla haettaessa jonkin konkreettisen luokan ilmentymää. Eräs ratkaisu on käyttää luokkahierarkialle yhteistä pääavainta (joka on siis määritelty

luokkahierarkian ylintä luokkaa vastaavassa taulussa) kaikissa luokkahierarkiaan liittyvissä tauluissa (Brown ja Whitenack, 1996). Toinen vaihtoehto on se, että näillä tauluilla on omat pääavaimensa, jolloin käytetään viiteavaimia yliluokan tauluun rivien yhdistämiseksi. (Fowler, 2002 s. 286).

Taulu luokkaa kohden –muunnostavassa voidaan katsoa olevan seuraavat vahvuudet:

- Taulut ovat helppoja ymmärtää, koska kaikki sarakkeet ovat relevantteja eikä tilahukkaa esiinny (Fowler, 2002 s. 286).
- Liiketoimintaluokkien ja taulujen välinen yhteys on suoraviivaista (Fowler, 2002 s. 286).
- Tukee monimuotoisuutta hyvin (Ambler, 2003).
- Yliluokkien muokkaaminen ja aliluokkien lisääminen on helppoa, koska vain yhtä taulua täytyy muokata (tai lisätä yksi taulu aliluokan tapauksessa) (Ambler, 2003).
- Tiedon määrä (tietokannan koko) kasvaa suorassa suhteessa olioiden määrän kasvuun (Ambler, 2003).

Heikkouksia voidaan katsoa olevan seuraavien seikkojen:

- Jo kohtalaisen syvissä luokkahierarkioissa luokkien lataamiseen vaadittavat useiden taulujen liitokset voivat olla raskaita (Brown ja Whitenack, 1996).
- Luokkien jäsenmuuttujien siirtäminen luokkahierarkiassa ylös- tai alaspäin vaatii muutoksia tietokantaan (Fowler, 2002 s. 286).
- Yliluokkien taulut saattavat muodostua (suorituskyvyllisiksi) pullonkauloiksi, koska niitä käsitellään tiheästi (Keller, 1997).
- Korkea normalisointiaste voi tehdä ad-hoc –kyselyjen tekemisen hankalaksi (Fowler, 2002 s. 286). Tätä voidaan helpottaa kuitenkin näkymien luomisella (Ambler, 2003).

- Tietokannassa tarvitaan monta taulua, yksi jokaista luokkaa kohden ja lisäksi suhteiden ylläpitämiseen tarvittavat taulut (Ambler, 2003).

4.1.3 Taulu konkreettista luokkaa kohden

Ajateltaessa tauluja olioilmentymän näkökulmasta on järkevä tapa ottaa kukin muistissa oleva olio ja muuntaa se yhdeksi riviksi tietokannassa. Tästä seuraa taulu konkreettista luokkaa kohden -muunnostapa, jossa on taulu jokaista konkreettista luokkaa kohden. (Fowler, 2002 s. 293) (Keller (1997) käyttää tästä muunnostavasta nimitystä 'taulu periytymispolkua kohden').

Jokainen konkreettista luokkaa vastaava taulu sisältää myös mahdollisesti abstraktien yliluokkien attribuutit sarakkeina. Huomioitavaa on, että yliluokan ei välttämättä tarvitse olla abstrakti: voi olla myös niin, että sitä ei muunneta (eikä täten myöskään säilötä) tietokantaan lainkaan.

Taulu konkreettista luokkaa kohden -muunnostavassa on seuraavat vahvuudet:

- Kukin taulu kaikki sisältää tarvittavat sarakkeet, eikä tauluissa ole epäoleellisia sarakkeita – tästä johtuen taulu on helppokäyttöinen muille sovelluksille. (Fowler, 2002 s.295). Tauluun on myös helppo suorittaa ad hoc –kyselyjä (Ambler, 2003).
- Tiedon hakuun ei tarvita liitoksia muihin tauluihin (Fowler, 2002 s.295).
- Taulua käsitellään vain, kun sitä vastaavaa luokkaa käsitellään, mikä hajauttaa käsittelykuormaa (engl. access load) (Fowler, 2002 s.295). Tästä ja edellisestä kohdasta seuraa, että tämä muunnostapa on yleensä suorituskykyinen. (Ambler, 2003)
- Muunnostapa tarjoaa optimaalisen tilankulutuksen, sillä tauluissa ei tarvita lainkaan ylimääräisiä sarakkeita. (Keller, 1997)

Heikkoudet taasen ovat:

- Pääavaimet saattavat olla hankalia käsitellä (Fowler, 2002 s.295)
- Suhteita abstrakteihin luokkiin ei voi pakottaa tietokannassa (Fowler, 2002 s.295)
- Jos jäsenmuuttujia siirretään ylös- tai alaspäin luokkahierarkiassa täytyy myös tauluja muuttaa. Muutosten tarve on vähäisempi kuin taulu luokkaa kohden –muunnostavassa, mutta toisin kuin taulu luokkahierarkiaa kohden –muunnostavassa, muutoksia joudutaan tekemään. (Fowler, 2002 s.295)
- Jos yliluokan attribuutti muuttuu, joudutaan muuttamaan kaikkia niitä tauluja, joissa tämän attribuutti on, koska yliluokan attribuutit ovat toistettuna mahdollisesti useissa tauluissa. (Fowler, 2002 s.296)
- Tehtäessä hakuja yliluokkaan, joudutaan katsomaan (tarkistamaan) kaikki (aliluokkia vastaavat) taulut, mikä johtaa useampiin tietokantaoperaatioihin (tai erikoiseen liitokseen). (Fowler, 2002 s.296)
- Kun olio muuttaa rooliaan, esimerkiksi asiakkaasta työntekijäksi, joudutaan se kopioimaan oikeaan tauluun ja antamaan sille uusi pääavainarvo. (Ambler, 2003)
- Usean roolin tukeminen ja samalla tietoeheyden säilyttäminen on vaikeaa. Esimerkiksi, mihin tallennetaan jonkun sellaisen nimi, joka on sekä asiakas ja työntekijä? (Ambler, 2003)

4.1.4 Periytymishierarkia binääridatana

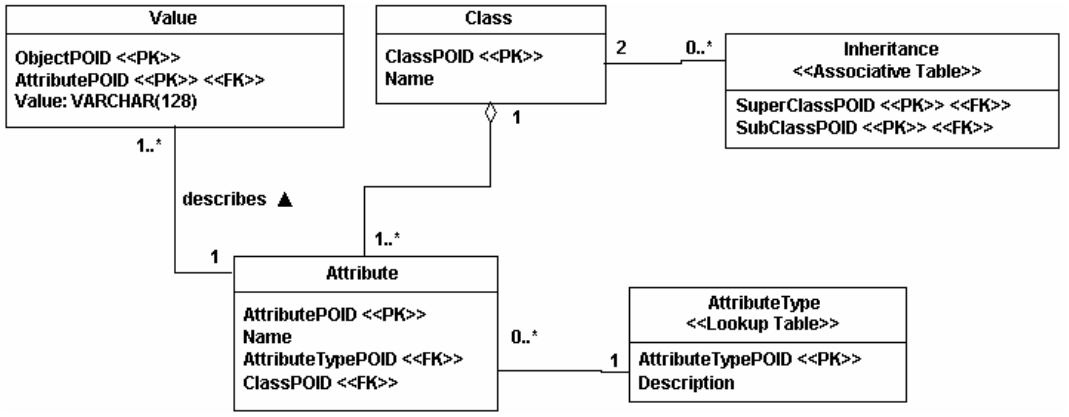
Yksi vaihtoehto luokkahierarkian muuntamiseksi relaatiokantaan on sarjallistaa se ja tallentaa binääridatana sarakkeeseen. Tapa voi olla toimiva, jos luokkahierarkia ei sisällä kovin monia ja/tai kooltaan isoja luokkia. Siihen

pätevät samat ongelmat kuin sarjallistamiseen yleensä: hakuja varten joudutaan lataamaan muistiin kaikki hierarkian luokat, eikä tieto ole saatavilla ad hoc –kyselyihin tai raportointiin. Huomioitavaa on, että tätä muunnostapaa voitaisiin käyttää myös yhdessä muiden muunnostapojen kanssa. (Keller, 1997)

Samaa periaatetta voidaan soveltaa myös koostamisen muuntamiseen. Binääridatana tallentaminen tarkoittaa käytännössä useista tietokannan hallintajärjestelmän tarjoamista ominaisuuksista luopumista, eikä sitä täten voida pitää suositeltavana mallina normaalitilanteessa.

4.1.5 Periytymishierarkia yleisessä taulurakenteessa

Ambler (2003) esittelee myös metatietolähestymistapana tunnetun tavan muuntaa periytymishierarkia yleiseen taulurakenteeseen. Tämä tapa soveltuu luokkien muuntamiseen tietokantaan yleisellä tasolla, ei vain luokkahierarkian muuntamiseen (Ambler, 2003). Ideana tässä muunnostavassa on, että arvoille, jäsenmuuttujille, jäsenmuuttujien tyypeille ja luokkien tunnisteille on oma taulunsa.



KUVIO 7. Periytymishierarkia yleisessä taulurakenteessa (Ambler, 2003).

Muunnostavassa on seuraavat vahvuudet (Ambler, 2003):

- Toimii hyvin, kun tietokantayhteydet on kätketty vakaan säilyvyyssovelluskehyksen sisään.

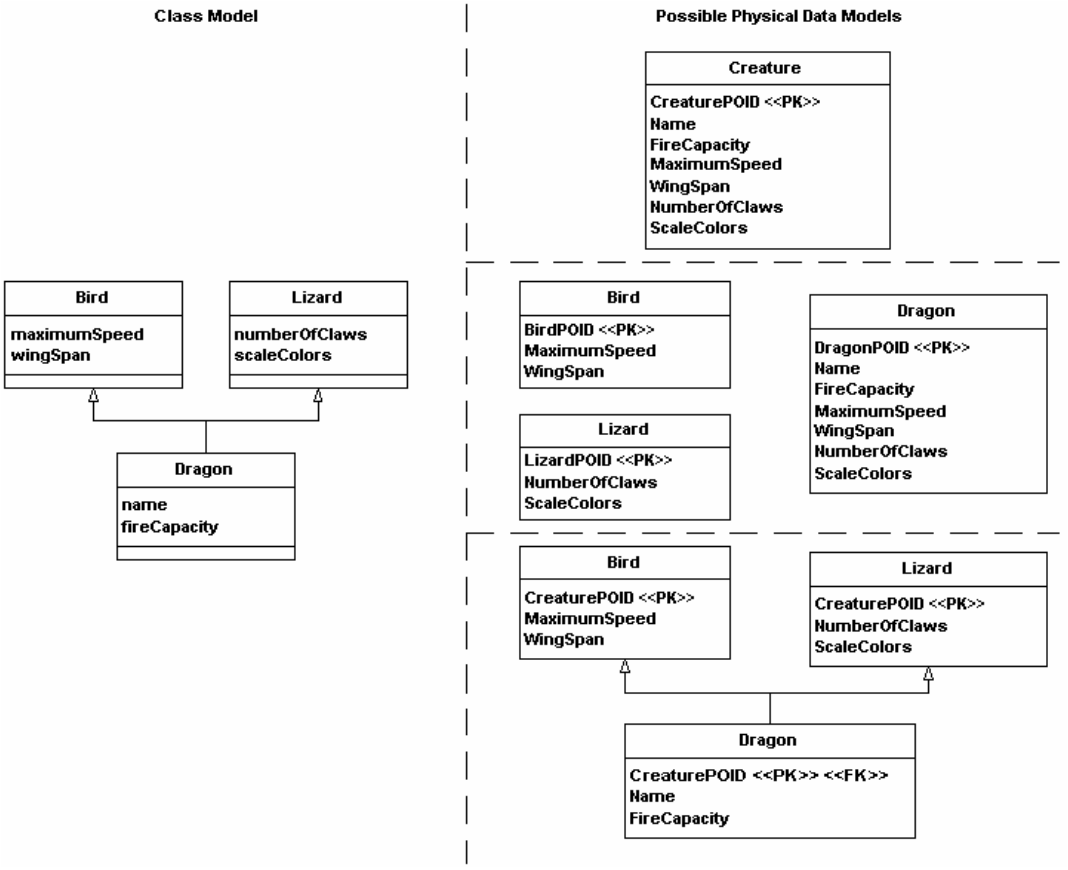
- Tapaa voidaan laajentaa tukemaan monenlaisia muunnoksia, kuten suhteiden muunnosta.
- Erittäin joustava, mahdollistaa olioiden tallentamistavan muuttamisen nopeasti, koska ei tarvitse kuin päivittää metatietoa luokka, perintä, muuttuja ja muuttujan tyyppi –tauluissa.

Heikkouksiksi voidaan lukea seuraavat seikat (Ambler, 2003):

- Erittäin edistynyt tekniikka, joka voi olla aluksi vaikea toteuttaa.
- Toimii vain pienillä määrillä dataa, koska yhden olion rakentamiseksi joudutaan käsittelemään useita rivejä tietokannassa. Tästä johtuen myös raportointi (ja ad hoc –kyselyt) voi olla erittäin vaikeata.
- Vaatii todennäköisesti hallinnointisovelluksen metatiedon ylläpitämiseen.

4.1.6 Moniperinnän muuntaminen

Moniperinnän muuntaminen relaatiomalliin ei juuri poikkea yksittäisperinnästä; periaatteessa kaikki samat muunnostavat ovat sovellettavissa myös moniperinnän osalta. Seuraavassa Amblerin (2003) esittämä kuvio moniperinnän muuntamiselle käyttäen seuraavia muuntamistapoja: periytymishierarkia yhdessä taulussa, taulu luokkaa kohden ja taulu konkreettista luokkaa kohden (kuviossa ei tosin ole mukana yhtään abstraktia luokkaa). Kuviossa olevasta Creature -taulusta puuttuu myös luokan tyypin ilmaiseva sarake.



KUVIO 8. Moniperinnän muuntaminen (Ambler, 2003)

4.1.7 Yhteenveto periytymisen muunnostavoista

Taulukossa 2 on koottu yhteen esitetyt muunnostavat. Samassa yhteydessä esitetään myös viitteellinen suositus siitä, missä tapauksissa mikin muunnostapa voi olla soveltuva (taulukko on mukaelma Amblerin (2003) vastaavasta täydennettynä).

TAULUKKO 2. Periytymisen muunnostavat

Tapa	**Milloin tulisi käyttää**
Luokkahierarkia yhdessä taulussa	Toimii hyvin, kun luokkahierarkia on yksinkertainen ja/tai matala eikä hierarkian luokkien tyyppien välillä ole juuri limittäytymistä
Taulu luokkaa kohden	Kun hierarkian luokkien tyyppien välillä on limittäytymistä tai tyypin muuttuminen on tavallista
Taulu konkreettista luokkaa kohden	Kun luokkien tyyppien muuttuminen tai limittyminen on harvinaista
Luokkahierarkia yleisessä taulurakenteessa	Kun oliomalli on monimutkainen ja käsiteltävä tietomäärä vähäinen, tai kun pysyvän tiedon käsittely ei ole kovin yleistä tai kun tieto voidaan ladata välimuistiin etukäteen
Luokkahierarkia binääridatana	Kun hierarkia ei sisällä kooltaan suuria luokkia ja hierarkian yksittäisiin luokkiin ei tarvitse kohdistaa kyselyjä

4.2 Suhteet

Brown ja Whitenack (1996) luokittelevat oliomallin luokkien väliset suhteet seuraavasti:

- yhdestä yhteen (esimerkiksi mies – vaimo)
- yhdestä moneen (äiti – lapsi)
- monesta moneen (esi-isä – lapsi)

- ternaarinen tai n-äärinen (opiskelija–luokka–professori)
- tarkennettu suhde (yritys–toimisto–henkilö)

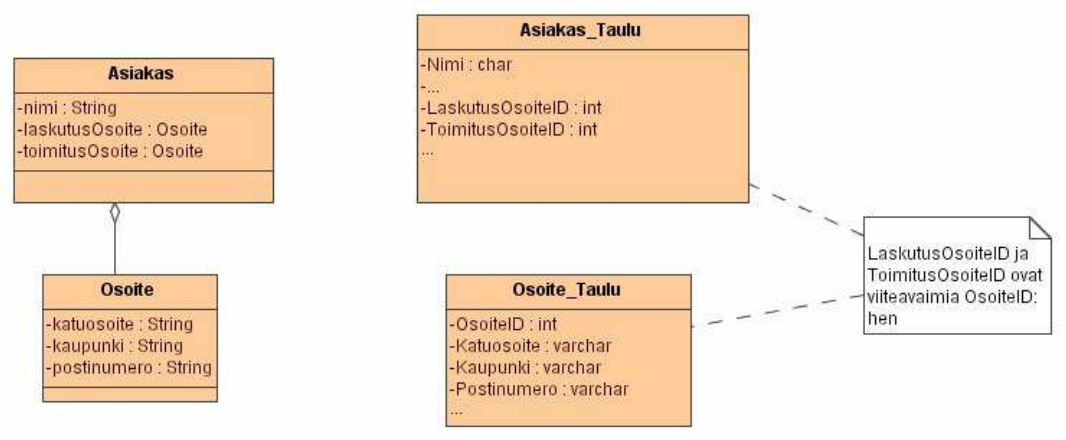
Olioiden väliset suhteet voivat ilmaista koostetta, suhteen ominaisuuksia tai niillä voi olla oma erityinen merkityksensä (esimerkiksi avioliitto on erityinen suhde miehen ja naisen välillä) (Brown & Whitenack, 1996). Oliomallissa kaikki nämä suhteet voivat olla joko yksi- tai kaksisuuntaisia, mutta relaatiotietokannoissa kaikki suhteet ovat kaksisuuntaisia (Ambler, 2003).

Oliomallissa suhteet esitetään olioviitteiden ja operaatioiden yhdistelmänä. Suhteen ollessa yhdestä yhteen, suhde toteutetaan olioviitteenä olioon, kun taas yhdestä moneen ja monesta moneen –suhteet toteutetaan kokoelmatyyppisenä attribuuttina ja operaatioina tuon attribuutin sisältämän kokoelman muokkaamiseen. Relaatiotietokannoissa puolestaan suhteita ylläpidetään käyttämällä viiteavaimia. (Ambler, 2003)

Seuraavissa alakohdissa esitellään neljä eri vaihtoehtoa oliomallin suhteiden muuntamiselle relaatiotietokantaan.

4.2.1 Yhdestä yhteen

Yhdestä yhteen -suhteessa olevat olioviitteet voidaan ilmaista relaatiotietokannassa suoraviivaisesti viiteavaimilla. Yhdestä yhteen -suhteessa tämä tarkoittaa esimerkiksi kuvion 9 kaltaista tilannetta.



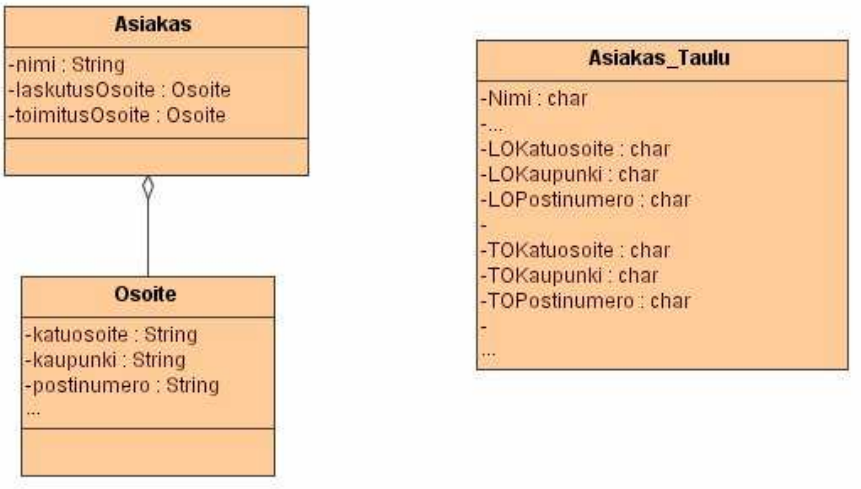
KUVIO 9. Yhdestä yhteen suhde viiteavainsuhteeksi muunnettuna (kuvio on mukaelma Kellerin (1997) vastaavasta).

Muunnostavalla on seuraavat seuraamukset (Keller, 1997):

- Vaatii joko liitoksen tai kaksi erillistä hakua tietokantaan viitatun olion lataamiseksi, joten suorituskyky ei ole ideaalinen.
- Ylläpidettävyys on hyvä, koska koosteen osana olevaan olioon (ja sitä vastaavaan tauluun) voidaan tehdä muutoksia koskematta koosteolioon.
- Koosteen osat eivät tule automaattisesti poistetuksi koosteoliota poistettaessa. Niiden poistamiseksi tarvitaan laukaisin tietokannassa tai sovelluslogiikkaa.
- Ad hoc –kyselyitä on helppo suorittaa koosteen osana oleviin olioihin, koska ne sijaitsevat omassa taulussa.

4.2.2 Yhdestä yhteen koostesuhde

Muunnettaessa oliomallin yhdestä yhteen koostesuhdetta relaatiokantaan, voidaan osana koostetta olevan olion attribuutit sijoittaa sarakkeiksi koosteoliota vastaavaan tauluun, edellyttäen, että oliolla ei ole olennaista omaa identiteettiä, eli se on niin sanotusti arvo-olio (engl. value object, Evans 2003, s.70-71).



KUVIO 10. Osana koostetta olevan olion muuntaminen koosteoliota vastaavaan tauluun (Keller, 1997).

Tällaisella muunnostavalla on seuraavat edut (Keller, 1997):

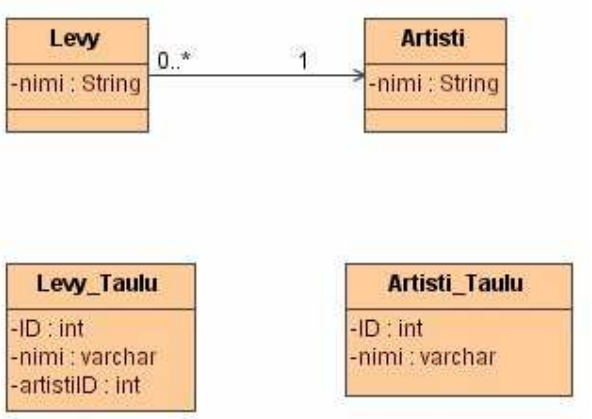
- *Suorituskyky*: Suorituskyky on optimaalinen, sillä haku voidaan kohdistaa vain yhteen tauluun koosteolion osana olevien olioiden hakemiseksi.
- *Tietokannan eheys:* Koosteen osana olevat oliot poistetaan automaattiseesti koosteolion poiston yhteydessä.

Muunnostavasta aiheutuu seuraavat hankaluudet (Keller, 1997):

- *Ylläpidettävyys ja joustavuus:* Mikäli koosteen osana olevan luokan olioita kuuluu useisiin eri luokkaa oleviin koosteolioihin, aiheutuu muunnostavasta ylläpitovaikeuksia, koska koosteen osana olevan olion luokan muuttuminen aiheuttaa muutoksia kaikkiin koosteolioita vastaaviin tauluihin.
- *Ad-hoc kyselyt:* Hakujen kohdistaminen osana koostetta oleviin olioihin on hyvin vaikeaa.

4.2.3 Yhdestä moneen

Kokoelmien suhteen tilanne ei ole ihan yhtä yksinkertainen kuin yhdestä yhteen suhteissa, koska relaatiomalli ei tue kokoelmia (Ambler, 2003). Ratkaisu on kääntää suhteen suunta laittamalla kokoelman omistajaa vastaavan taulun pääavain viiteavaimeksi kokoelmaoliota vastaavaan tauluun, kuten kuviossa 11.



KUVIO 11. Yhdestä moneen viiteavainmuunnoksena (Fowler, 2002 s. 236)

Tällaisen suhteen päivittämisessä on joitakin ongelmia – kuinka ilmaista tietokannalle, mitkä muutokset kokoelmassa pitäisi tallettaa, kun kokoelman omistajaa vaihdetaan. Tähän on kolme eri vaihtoehtoa: (1) poistaminen ja lisääminen, (2) käänteisosoittimen (engl. back pointer) lisääminen ja (3) kokoelman vertaaminen muutoksien havaitsemiseksi. (Fowler, 2002 s. 237).

- Poistaminen ja lisääminen. Tämä vaihtoehto on teknisesti helpoin toteuttaa: kun omistajaa päivitetään, tuhotaan ensin kaikki kokoelmaan liittyvät rivit ja lisätään kokoelmassa olevat oliot (mahdollisesti uudestaan) kantaan. Ilmeinen haittapuoli tässä tavassa on se, että kokoelman olioihin ei voi olla viittauksia mistään muualta. (Fowler, 2002 s.238)
- Käänteisosoittimen lisääminen. Tässä vaihtoehdossa kokoelman olioihin lisätään viite omistajaan, eli tehdään suhteesta kaksisuuntainen.

Oliomalli siis muuttuu, mutta päivittämisen voi nyt hoitaa käyttämällä yksinkertaista tekniikkaa yksiarvoisiin kenttiin suhteen toisessa päässä. (Fowler, 2002 s.238)

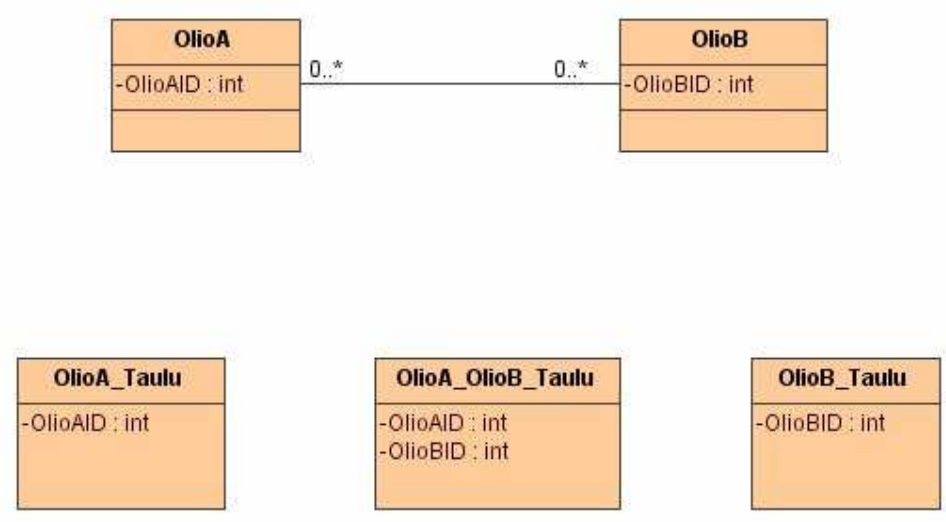
- Kokoelman vertaaminen muutoksien havaitsemiseksi. Tämän voi toteuttaa kahdella tavalla – joko lataamalla tallennustilanteessa kokoelman oliot tietokannasta uudestaan tai tekemällä kopion kokoelmasta kun se alun perin ladataan, ja vertaamalla tätä kopiota päivitettävään. (Fowler, 2002 s.238)

4.2.4 Yhdestä moneen koostesuhde

Yhdestä moneen –koostesuhteen muuntaminen on pitkälle samankaltaista kuin yhdestä moneen –viitesuhteenkin, erona vain se, että koosteen osana olevien olioiden identiteettinä taulussa voidaan käyttää koosteolion identiteettiä. Kuten yhdestä yhteen –koostesuhteessa, tämä edellyttää sitä, että koosteen osana olevalla oliolla ei ole omaa identiteettiä ja että sen elinkaari on sidottu koosteolion elinkaareen.

4.2.5 Monesta moneen -suhteet

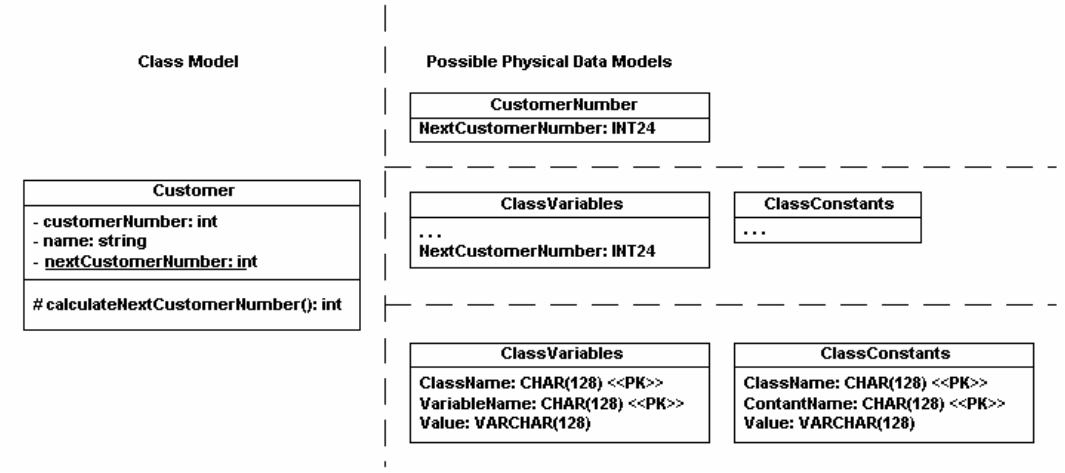
Monesta moneen –suhteiden muuntamiseen ei ole juurikaan vaihtoehtoja: relaatiomallissa tämä vaatii suhdetaulun, joka sisältää yhdistettäviä olioita vastaavien taulujen pääavaimet. Muilta osin muuntaminen tapahtuu kuten yhdestä moneen –suhteessa.



KUVIO 12. Monesta moneen –suhde suhdetauluna (kuvio on mukaelma Kellerin (1997) vastaavasta).

4.3 Luokkakohtaisten attribuuttien muuntaminen

Toisinaan luokat sisältävät luokkakohtaisia attribuutteja, eli muuttujia, jotka ovat yhteisiä kaikille luokan ilmentymille. Näiden muuntaminen poikkeaa normaalien attribuuttien muuntamisesta. (Ambler, 2003)



KUVIO 13. Luokkakohtaisten attribuuttien muuntaminen (Ambler, 2003).

Ambler (2003) esittää luokkakohtaisten attribuuttien muuntamiseen neljä eri tapaa. Seuraavassa taulukkomuotoinen yhteenveto näiden eri tapojen eduista ja haitoista (Ambler, 2003):

TAULUKKO 3. Luokkakohtaisten attribuuttien muuntaminen.

Tapa	**Edut**	**Haitat**
Yksisarakkeinen ja –rivinen taulu	Yksinkertainen, nopea	Voi johtaa useisiin pieniin tauluihin
Monisarakkeinen, yksirivinen taulu yhtä luokkaa kohden	Yksinkertainen, nopea	Voi johtaa useisiin pieniin tauluihin, kuitenkin ei yhtä useaan kuin yksisarakkeisessa tavassa
Monisarakkeinen, yksirivinen taulu kaikkia luokkia varten	Minimaalinen määrä tauluja	Mahdollisuus samanaikaisuusongelmiin, jos monen luokan täytyy päästä dataan käsiksi yhtäaikaisesti. Yksi ratkaisu on lisätä LuokkaVakiot–taulu luettavien attribuuttien erottamiseksi päivitettävistä.

Monirivinen yleisluonteinen malli kaikkia luokkia varten.	Minimaalinen määrä tauluja. Vähentää samanaikaisuusongelmia (olettaen, että tietokanta tukee rivikohtaista lukitusta).	Joudutaan suorittamaan tyyppimuunnoksia. Tietomalli on kytketty luokkien nimiin ja niiden luokkakohtaisiin muuttujiin.

4.4 Yhteenveto

Tarkastelimme tässä luvussa kirjallisuudessa esitettyjä malleja keskeisten oliopiirteiden, periytymisen ja koosteen, muuntamiseksi relaatiomalliin. Huomasimme, että olio- ja relaatiomallin eroista huolimatta niiden rakenteellinen yhteensovittaminen ei ole mahdotonta, taikka edes erityisen monimutkaista periaatteellisella tasolla - tuotantotason tekninen toteutus ei tietysti ole triviaalia. Sovellusten erilaisista tarpeista johtuen esimerkiksi yksikään periytymishierarkian muunnostapa ei ole riittävä kaikkiin tapauksiin – usein myös yhden periytymishierarkian sisällä olisi suorituskyky ja / tai ylläpitosyistä toivottavaa pystyä yhdistelemään eri muunnostapoja.

Luvussa kaksi esiteltyjen vaatimusten valossa eri muunnostapojen tukeminen on ehdottomasti vaadittu ominaisuus säilyvyyssovelluskehykseltä. Esitellyt muunnostavat mahdollistavat sen, että monimutkainenkin oliomalli voidaan periytymishierarkioiden ja suhteiden osalta säilyttää sellaisenaan kuin se on, muutoksia siihen relaatiotietokantaan säilömistä varten ei tarvita, jos vain säilöntäsovelluskehys tukee muunnostapoja monipuolisesti. Tämän luvun kohdassa 4.1.5 esiteltyä mallia periytymishierarkian muuntamiseksi emme kuitenkaan katso välttämättömäksi ominaisuudeksi sovelluskehyksille.

Esitetyllä yleisellä taulurakenteella menetetään “normaalin” relaatiomallin etuja, kuten kyselyjen helppous, ja koska kantamallin täytyy olla täydellisesti sovelluksen hallinnassa yleisen taulurakenteen mallissa, voidaan relaatiokannan järkevyys säilömisalustana kyseenalaistaa tällaisessa tapauksessa. Samoin argumentein kohdassa 4.1.4 esitellyn muuntamistavan tukemisen emme katso olevan erityisen tärkeää. Luvun kaksi vaatimuksia vasten luokkakohtaisten attribuuttien erityiset muunnostavat eivät myöskään ole välttämättömiä.

5 OLION IDENTITEETTI, INKREMENTAALINEN LATAAMINEN JA TRANSAKTIOT

Tässä luvussa käsittelemme sekä olio- että relaatiomallin identiteettiä, niiden eroavaisuuksia, transaktioita ja laajojen oliograafien yhteydessä tarvittavaa inkrementaalista lataamista.

5.1 Olion identiteetti

Identiteetti on se olion ominaisuus, joka erottaa kunkin olion kaikista toisista. Identiteettiä on tutkittu lähes erillään yleiskäyttöisissä ohjelmointikielissä ja tietokantakielissä. Sen merkitys on kasvamassa näiden kahden ympäristön kehittyessä ja sulautuessa. (Copeland ja Khoshafian, 1986).

Olion tunnistaminen on ratkaisevan tärkeä kysymys monissa tietojenkäsittelytieteiden haaroissa, ulottuen käyttöjärjestelmistä tietoverkkoihin, tietokantajärjestelmiin ja ohjelmointikieliin (Wieringa ja de Jonge, 1995). Ohjelmointikielissä olion identiteetin käsite on ollut olemassa kauan, tietokannoissa käsite on tuoreempi (Atkinson ym. 1990).

Atkinson ym. (1990) määrittelevät olion identiteetin seuraavasti: olion identiteetin sisältävässä mallissa olio on olemassa riippumatta sen arvosta. Näin ollen on olemassa kaksi käsitettä olioiden vastaavuudesta: kaksi oliota voivat olla identtisiä (ne ovat sama olio) tai ne voivat olla samanarvoisia (niillä on sama arvo).

Wieringa ja de Jonge (1995) tarkastelevat olion identiteetin käsitettä yleisellä tasolla ja esittävät yleismaailmallisen oliotunnisteen käsitteen, jonka avulla voitaisiin välttyä olioiden tunnistamiseen liittyviltä ongelmilta erillisten järjestelmien yhteydessä. Tällainen yleismaailmallinen oliotunniste on ainoalaatuinen halki maailman ja kaikkien sen tilojen (Wieringa ja de Jonge, 1995). Tämän tutkielman kannalta kiinnostavampaa on kuitenkin olio- ja relaatiomallien olioiden tunnisteisiin liittyvät erovaisuudet sekä niiden

yhteensovittamisen tekniset haasteet, joten seuraavissa alakohdissa syvennymme vain näihin. Yleismaailmallinen oliotunniste on kuitenkin mukana luvun yhteenvedossa esitettävässä taulukossa.

5.1.1 Identiteetti oliomallissa

Identiteetillä oliomallissa tarkoitamme tässä käytännössä olion identiteettiä (olio-)ohjelmointikielessä. Sekaannuksen välttämiseksi lienee syytä vielä selventää, että tässä tarkasteltava identiteetti vastaa lähinnä Wieringan ja de Jongen (1995) sisäistä tunnistetta, ei yleismaailmallista oliotunnistetta.

Useimmat yleiskäyttöiset ohjelmointikielet ovat suunniteltu huomioimatta säilyvän tiedon käsitettä (Copeland ja Khoshafian, 1986). Tästä syystä ne tarjoavat heikon tuen olion identiteetille ajallisessa ulottuvuudessa, sillä ohjelmointikielen kannalta data elää vain ohjelman suorituksen ajan (Copeland ja Khoshafian, 1986). Tämä pätee myös tämän tutkielman kannalta olennaisimpaan ohjelmointikieleen, Javaan, jossa olioiden samuus perustuu (ainakin useimmilla virtuaalikoneilla) niiden sijaintiin keskusmuistissa, ja sitä voidaan testata == -operaattorilla. Olioiden samanarvoisuutta voidaan puolestaan testata `java.lang.Object` –luokan `equals` –metodilla, jonka käyttäjä voi toteuttaa haluamakseen (tiettyjen rajoitteiden puitteissa) (Bloch, 2001 s. 23), jolloin olioiden samanarvoisuutta voidaan mielekkäästi vertailla myös eri ajokertojen välillä. Javassa voidaan siis tunnistaa oliot, jotka ovat samoja, ja oliot, jotka ovat samanarvoisia.

5.1.2 Identiteetti relaatiomallissa

Relaatiotietokannoissa avain on yhden tai useamman attribuutin arvojen kombinaatio, jonka täytyy olla yksikäsitteinen asianomaisessa rivien joukossa (Wieringa ja de Jonge, 1995). Esimerkiksi tietokanta-avaimen täytyy olla ainoalaatuinen kussakin tietokannan sallitussa tilassa ja relaatioavaimen täytyy

olla ainoalaatuinen kussakin relaation ilmentymässä (Wieringa ja de Jonge, 1995). Copeland ja Khoshafian (1986) arvostelevat relaatiomallin identiteettiä, heidän mukaansa se sekoittaa arvojen ja identiteetin käsitteet. Relaatiomallissa oliolla ei ole muuta identiteettiä kuin sen arvo (Meyer 1997, s. 1052), mikä poikkeaa siis perustavanlaatuisesti oliomallin identiteetistä. Tietysti relaatiomallissakin voidaan ilmaista olion identiteettiä lisäämällä erityinen avainkenttä relaatioon, jonka taataan olevan ainoalaatuinen kullekin oliotyypille (Meyer, 1997, s. 1053). Relaatiomallissa tästä täytyy kuitenkin huolehtia erikseen, kun taas oliomallissa olioilla on tunniste oletusarvoisesti (Meyer, 1997 s.1053).

5.1.3 Yhteenveto identiteettien eroavaisuuksista

Wieringa ja de Jonge (1995) esittävät seuraavan taulukon aiemmissa kohdissa esiteltyjen identiteettien eroavaisuuksista. Taulukossa on mukana heidän esittämänsä yleismaailmallinen oliotunniste.

TAULUKKO 4. Avainten, oliotunnisteiden ja sisäisten tunnisteiden eroavaisuudet (Wieringa ja de Jonge, 1995).

	Avaimet	Oliotunnisteet	Sisäiset tunnisteet (surrogaatit)
1.	Tietokannan käsite	Mallintamisen käsite	Toteutuskäsite
2.	Saattaa sisältää muuttuvaa tietoa	Ei sisällä muuttuvaa tietoa	Ei sisällä muuttuvaa tietoa
3.	Päivitettävä	Ei voida päivittää	Ei voida päivittää
4.	Ainoalaatuinen kussakin tietokannan tilassa (tai relaatiossa)	Ainoalaatuinen halki kaikkien maailman tilojen	Ainoalaatuinen halki kaikkien yhden tietokantajärjestelmän tilojen

5.	Tiedonsiirto-ongelma yleinen	Tiedonsiirto-ongelma harvinainen	Tiedonsiirto-ongelma eri tietokantajärjestelmien välillä
6.	Usein tietokannan käyttäjän asettama	Oliotunnistemekanismin asettama	Tietokantajärjestelmän asettama
7.	Käyttäjälle näkyvä	Käyttäjälle näkyvä	Käyttäjälle näkymätön

5.2 Inkrementaalinen lataaminen

Yleensä olio ei elä eristyksissä muista olioista, vaan vuorovaikuttaa ja on suhteessa muiden oliomallin olioiden kanssa. Luonteva tapa päästä käsiksi johonkin olioon on seurata olioiden välisiä suhteita - esimerkiksi jos käytettävissä luokan *Asiakas* –ilmentymä, ja haluamme selvittää asiakkaan tilaukset, luonteva tapa on pyytää niitä tältä ilmentymältä. Tästä seuraa, että kun olio otetaan säiliöstä keskusmuistiin sovelluksen käsiteltäväksi, niin otetaan myös siihen liittyvät oliot. Lopulta tämä johtaa siihen, että keskusmuistiin saatetaan joutua lataamaan huomattava osa tietokannan sisällöstä. Tämän ongelman välttämiseksi tarvitaan *inkrementaalista lataamista (*engl. lazy load). Inkrementaalinen lataaminen tarkoittaa sitä, että oliot ladataan tietokannasta vasta sitten, kun niitä käsitellään.

Seuraavassa esitellään neljä eri vaihtoehtoa inkrementaalisen lataamisen toteuttamiseksi: laiska lataaminen (engl. lazy initialization), virtuaalinen välittäjä (engl. virtual proxy), arvonhaltija (engl. value holder) ja haamu (engl. ghost).

Laiska lataaminen on yksinkertaisin lähestymistapa. Perusajatus on, että kun johonkin kenttään pyritään pääsemään käsiksi, niin tarkistetaan ensin onko kentän arvo tyhjäarvo. Jos on, niin kentän arvo lasketaan ennen sen

palauttamista. Jotta tämä toimisi, täytyy kaiken kentän käsittelyn – myös luokan sisäisen – tapahtua kentän saantimetodin kautta. Tyhjäarvon käyttäminen lataamattoman kentän arvon ilmaisemiseen toimii hyvin, kunhan se ei ole sallittu arvo kentässä. Muussa tapauksessa lataamattoman arvon ilmaisemiseen täytyy käyttää jotain toista arvoa. (Fowler, 2002 s. 201)

Vähemmän tietokantaan riippuvuutta aiheuttava vaihtoehto on *virtuaalinen välittäjä*. Virtuaalinen välittäjä on olio, joka näyttää siltä oliolta, jonka pitäisi olla kentässä, mutta ei itse asiassa sisällä mitään. Vasta kun jotain sen metodeja kutsutaan, niin oikea olio ladataan tietokannasta. Virtuaalisen välittäjän heikkous on siihen liittyvät identiteettiongelmat. Yhtä oliota kohden voi olla monta virtuaalista välittäjää, joilla on tällöin oma olioidentiteettinsä, vaikka ne esittävät samaa käsitteellistä oliota. (Fowler, 2002 s. 201)

Haamu on varsinainen olio osittaisessa tilassa. Kun olio ladataan tietokannasta, se sisältää vain sen tunnisteen. Yritettäessä käsitellä sen kenttiä, ladataan olion oikea tila tietokannasta. Koko olion tilaa ei tarvitse ladata kerralla, vaan lataaminen voidaan ryhmittää yleisesti haettujen yhdistelmien mukaan. Jos tiedetään, että jotain tietoa tullaan todennäköisesti tarvitsemaan, voidaan se ladata aina kun haamu ladataan ja loput tarpeen mukaan. (Fowler, 2002 s. 202)

Inkrementaalinen lataaminen voi johtaa siihen, että tietokantaan tehdään huomattavasti enemmän hakuja kuin suorituskyvyn kannalta olisi optimaalista. Esimerkiksi, jos ladattava olio sisältää kokoelman muita olioita, jotka ladataan yksi kerrallaan niitä käsiteltäessä, on se hitaampaa kuin koko kokoelman kerralla lataaminen. Hyvä ratkaisu onkin tehdä koko kokoelmasta laiska, ja ladata se kokonaisuudessaan jos jotain sen sisältämää oliota käsitellään. Tämä toimii yleensä hyvin, kunhan kokoelma ei ole kooltaan huomattavan iso. (Fowler, 2002 s. 201).

5.3 Transaktiot

Transaktiot ovat hyvin olennainen asia palvelinpuolen yritysjärjestelmien kehityksessä, etenkin suoritettaessa säilyvyyteen liittyviä operaatioita, kuten päivityksiä tietokantaan (Roman et al, 2002 s.275). Transaktiot eivät ole ainoastaan tietokantaan ja säilyvyyteen liittyviä, mutta tämän tutkielman kannalta kiinnostavinta ovat juuri transaktiot säilyvyyskontekstissa, ja edelleen niihin liittyen nimenomaan olio-relaatiomuuntajan kannalta olennaiset seikat.

Tyypillinen yritysjärjestelmän käyttäjälle tarjoama toiminto, joka mahdollistaa käyttäjälle tiedon hakemisen ja sen päivittämisen sisältää seuraavat askeleet (Nock, 2003 s.372):

1. Käyttäjä valitsee päivitettävän kohteen.
2. Sovellus näyttää valitun kohteen käyttäjälle muokattavassa lomakkeessa.
3. Käyttäjä muokkaa yhtä tai useampaa kenttää lomakkeessa ja tallentaa muutokset.
4. Sovellus tallentaa käyttäjän tekemät muutokset tietokantaan.

Nämä toimenpiteet muodostavat yhdessä yksittäisen työnyksikön (engl. Unit of work). Yleisempi kuvaus työnyksiköstä on (Nock, 2003 s.372):

- *Lukeminen* – Lue tieto tietokannasta ja tee siitä työkopio keskusmuistiin.
- *Laskenta* – Suorita laskenta työkopion sisältämien tietojen pohjalta.
- *Päivitys* – Päivitä laskennan tuloksena saatu tieto takaisin tietokantaan.

Juuri työkopiot aiheuttavat useita samanaikaisuudesta johtuvia ongelmia. Useat sovellusilmentymät luovat erillisiä työkopioita aloittaessaan suorittamaan yksilöllisiä laskentojaan ja päivityksiään (Nock, 2003 s.373). Jos järjestelmä ei hyödynnä mitään menetelmää yhtäaikaisuuden hallintaan, kukin sovellusilmentymä tallentaa tekemänsä muutokset välittämättä lainkaan

muiden tekemistä muutoksista (Nock, 2003 s.373) - jäljempänä viittamme näihin muiden tekemiin kadotettuihin muutoksiin *menetettyinä päivityksinä*. Toisaalta transaktioiden täydellinen eristäminen heikentää yleensä järjestelmän suorituskykyä huomattavasti, eikä monissa tapauksissa täydellistä eristämistä tarvita. Seuraavissa alakohdissa tarkastelemme eristystasoja sekä transaktioiden hallintaan ratkaisuksi esitettyjä suunnittelu- ja toteutusmalleja.

5.3.1 Eristystasot

Tietokannat (ja muut transaktionaaliset järjestelmät) pyrkivät takaamaan, että kukin transaktio on eristetty muista. Tällä päästään siihen, että yksittäisen transaktion kannalta katsottuna näyttää siltä, ettei muita transaktioita ole meneillään. Perinteisesti tämä on toteutettu käyttämällä lukitusta – transaktion sallitaan asettaa lukko tiettyihin tietoihin, estäen väliaikaisesti muiden transaktioiden pääsyyn näihin. (Bauer ja King, 2004 s.161)

Transaktioiden epätäydellinen eristäminen altistaa seuraaville ongelmille (Bauer ja King, s.162):

- *Menetetty päivitys* (engl. lost update) – kuten kohdassa 5.3 esitetty. Tätä esiintyy, kun järjestelmä ei hyödynnä minkäänlaista lukitusta.
- *Likainen luku* (engl. dirty read) – Jokin transaktio lukee toisen transaktion tekemiä muutoksia ennen kuin tämän toinen transaktio suoritettu. Tämä on vaarallista, koska toinen transaktio saatetaan vielä peruuttaa.
- *Toistumaton luku* (engl. unrepeatable read) – Jokin transaktio lukee rivin kahdesti, saaden eri arvot eri kerroilla. Tällainen tilanne saattaa aiheutua, jos jokin toinen transaktio on kirjoittanut riviin ja suoritettu loppuun lukujen välissä.
- *Toinen menetettyjen päivitysten ongelma* (engl. second lost updates problem) – toistamattoman luvun erityistapaus. Tapahtuu, kun kaksi samanaikaista transaktiota lukevat rivin ennen kummankaan transaktion

vahvistamista, ensin toinen kirjoittaa tietonsa ja vahvistaa transaktion, minkä jälkeen toinen kirjoittaa tietonsa ja vahvistaa transaktion – ensimmäisen transaktion tekemät muutokset katoavat.

- *Haamuluku* (engl. phantom read) – Transaktio suorittaa kyselyn kahdesti, ja toisen kyselyn palauttamissa tuloksissa on rivejä, joita ei ollut ensimmäisessä (kyselyn ei tarvitse olla täsmälleen sama). Vastaavasti sama ongelma ilmenee käänteisesti, jos toisen kyselyn tuloksissa ei ole kaikkia niitä rivejä, jotka olivat ensimmäisen kyselyn tuloksissa. Tämä tilanne tapahtuu, kun jokin toinen transaktio lisää uusia rivejä näiden kahden kyselyn välissä.

Näiden tilanteiden aiheuttamien ongelmien välttämiseksi voidaan määritellä eri *transaktioiden eristystasoja*. Eristystasot tarjoavat mahdollisuuden valita sovellukselle sopiva kompromissi tiedon eheyden ja suorituskyvyn väliltä. ANSI SQL –standardin mukaiset eristystasot ovat seuraavat (Bauer ja King, 2004 s. 163):

- *Vahvistattomien luku* (engl. read uncommitted) – sallii likaiset luvut, mutta ei menetettyjä päivityksiä. Transaktio ei voi kirjoittaa riviin, mikäli toinen vahvistamaton transaktio on jo kirjoittanut siihen. Mikä tahansa transaktio voi kuitenkin lukea minkä tahansa rivin. Tämä eristystaso voidaan toteuttaa käyttämällä jakamattomia kirjoituslukkoja.
- *Vahvistettujen luku* (engl. read committed) – sallii toistamattomat luvut, mutta ei likaisia lukuja. Tämä voidaan saavuttaa käyttämällä hetkellisesti jaettuja lukulukkoja ja jakamattomia kirjoituslukkoja. Lukulukot eivät estä muita transaktioita pääsemästä käsiksi riviin, mutta vahvistattomat kirjoittavat transaktiot estävät kaikkien muiden transaktioiden pääsyn riviin.
- *Toistettava luku* (engl. repeatable read) – ei toistumattomia eikä likaisia lukuja. Haamuluvut ovat sen sijaan mahdollisia. Voidaan toteuttaa

käyttämällä jaettuja lukulukkoja ja jakamattomia kirjoituslukkoja. Lukevat transaktion estävät kirjoittavien transaktioiden pääsyn (mutta sallivat muut lukevat transaktiot), ja kirjoittavat transaktiot estävät kaikkien muiden transaktioiden pääsyn.

- *Sarjallistuva* (engl. serializable) – tarjoaa tiukimman transaktioiden eristyksen. Emuloi sarjallista transaktioiden suorittamista siten, että transaktiot suoritetaan peräjälkeen eikä yhtäaikaisesti. Sarjallistumista ei voida toteuttaa käyttämällä rivikohtaisia lukkoja, jonkin toisen mekanismin täytyy estää juuri lisätyn rivin näkymisen transaktiolle, joka on jo suorittanut kyselyn, jonka olisi pitänyt palauttaa kyseinen rivi (haamulukujen estäminen).

Transaktioiden eristystaso tulisi valita sovelluskohtaisesti, huomioitavia seikkoja ovat sovelluksen suorituskyvylle ja tiedon eheydelle asetetut vaatimukset sekä säilyvän tiedon pääasialliset käyttötavat (onko tieto enimmäkseen vain luettavaa, vai päivitetäänkö sitä usein ja niin edelleen). Myös mahdollinen välimuistin ja tiedon versioinnin käyttö vaikuttavat optimaalisen eristystason valintaan (Bauer ja King, 2004 luku 6). Eristystasot ovat olennainen osa usean käyttäjän transaktionaalista järjestelmää, ja tuen niille voidaan katsoa olevan edellytys olio-relaatiomuuntajalle.

5.3.2 Optimistinen lukitus

Optimistinen lukitus ylläpitää versioinformaatiota estääkseen menetettyjä päivityksiä. Nimestään huolimatta optimistisessa lukituksessa ei lukita tietoa, vaan optimistinen lukitus käyttää sovelluksen semantiikkaa validoidakseen työkopion version ennen tietokantaan päivitystä. (Nock, 2003 s.395-396)

Marinescu (2002, luku 3) esittää erään tavan optimistisen lukituksen toteuttamiseen, versionumeron. Siinä ideana on, että työkopiossa pidetään mukana kokonaislukua, joka ilmaisee mistä versiosta on kyse (versionumeron täytyy löytyä siis myös tietokannasta vastaavasta taulusta). Kun työkopiota

ollaan päivittämässä takaisin kantaan, tarkistetaan onko versionumero kannassa edelleen sama kuin työkopiossa, ja jos on, niin päivitetään tieto työkopiosta kantaan ja samalla kasvatetaan versionumeroa yhdellä. Mikäli versionumero ei ole sama, tarkoittaa se, että jokin toinen prosessi on päivittänyt tietoa työkopion tiedon hakemisen jälkeen. Tässä tapauksessa työkopion sisältämä tieto on siis vanhentunutta, eikä sitä tule päivittää kantaan.

Optimistinen lukitus on siitä hyvä, että se ei itse asiassa lukitse mitään eikä täten vaikuta negatiivisesti sovelluksen skaalautuvuuteen (Nock, 2003 s.399). Lisäksi sitä käytettäessä ei ole vaara orpojen lukitusten syntymiseen, mutta haittapuolena on, että käyttäjät saattavat menettää tekemiään muutoksia (Nock, 2003 s.399). Jos konfliktien todennäköisyys on suuri, saattaa parempi vaihtoehto olla seuraavassa kohdassa esiteltävä pessimistinen lukitus.

5.3.3 Pessimistinen lukitus

Pessimistisessä lukituksessa sovellus liittää lukon työkopioon hakemaansa tietoon estääkseen saman tiedon yhtäaikaisen muokkauksen. Tätä menetelmää kutsutaan pessimistiseksi, koska se olettaa, että muut prosessit yrittävät päivittää samaa tietoa (Nock, 2003 s.407). Pessimistinen lukitus on hyvä vaihtoehto käyttäjän käyttäessä kauan aikaa tiedon muokkaamiseen, etenkin jos muokkauksen uudelleen tekeminen on isotöistä. Edellisessä kohdassa tarkastellussa optimistisessa lukituksessahan versiokonfliktin sattuessa täytyy ladata uusimmat tiedot kannasta ja käyttäjän tekemät muutokset menetettäisiin. Pessimistisen lukitus avulla sovellukset voivat havaita tilanteet, joissa useat asiakkaat ovat muokkaamassa samaa tietoa ja ilmoittaa asiakkaalle tilanteesta ennen kuin muutoksia yritetään tehdä (Nock, 2003 s.409).

Pessimistisessä lukituksessa piilee riski orpoihin lukkoihin, eli sovellusilmentymä saattaa lukita tiedon, mutta laiminlyödä lukon vapauttamisen (Nock, 2003. s. 409). On tosin huomioitava, että tietokannan hallintajärjestelmässä voi olla mekanismi tälläisten tilanteiden selvittämiseen.

5.4 Yhteenveto

Tarkastelimme tässä luvussa olion identiteettiä olio- sekä relaatiomallissa ja osoitimme niiden olevan perustavanlaatuisesti erilaisia. Toimme myös esille näiden eroavaisuuksista johtuvia ongelmia. Säilyvyyssovelluskehyksen arvioimisen kannalta näkymätön tuki olion identiteetille sisältyy kohdassa 2.2.2 esittämäämme näkymättömän säilyvyyden vaatimukseen, eli säilöttävään olioon ei tulisi tarvita lisätä erityisiä identiteettikenttiä.

Osoitimme inkrementaalisen lataamisen olevan olennainen ominaisuus säilyvyyssovelluskehykselle laajojen oliograafien yhteydessä, sillä jos yhden kannasta ladattavan olion mukana tulisi aina kaikki muut siihen liittyvät oliot, ajauduttaisiin nopeasti ongelmiin sekä suorituskyvyn että muistinkulutuksen suhteen. Inkrementaalisen lataamisen toteuttamiseen esitettiin eri tapoja, joilla on kullakin omat puolensa. Osa esimerkiksi sallii attribuuttikohtaisen inkrementaalisen lataamisen ja osa ei. Luvun kaksi vaatimuksissa inkrementaalinen lataaminen tuotiin esille kohdassa 2.2.10.

Tuki eri transaktiostrategioille on myös tärkeä ominaisuus säilyvyyssovelluskehykselle. Eri strategiat sopivat eri tilanteisiin, ja kuten tässä luvussa huomasimme, niin sekä pessimistinen että optimistinen lukitus ovat molemmat lähes välttämättömiä ominaisuuksia. Tuki transaktionaalisuudelle tuotiin vaatimuksena esille kohdassa 2.2.9.

6 YHTEENVETO

Tämän tutkielman ensisijainen tehtävä oli selvittää, mitkä ovat olio- ja relaatiomallien väliset yhteensopivuusongelmat ortogonaaliseen säilyvyyteen pyrittäessä. Lähestyimme ongelmaa rakentamalla luvussa kaksi ortogonaalisen säilyvyyden vaatimukset Atkinsonin esittämien vaatimusten pohjalta. Atkinsonin vaatimukset eivät huomioi olio-relaationaalisen järjestelmän erityispiirteitä, joten täydensimme vaatimuksia niiltä osin.

Luvussa kolme tarkastelimme olio- ja relaatiomallin välistä irtikytkentää ja siihen esitettyjä suunnittelu- ja toteutusmalleja. Näiden mallien irtikytkentä on tärkeää sovelluksen ylläpidettävyyden kannalta, ja välttämätöntä ortogonaaliseen säilyvyyteen pyrittäessä.

Luvussa neljä syvennyimme olio- ja relaatiomallin väliseen muuntamiseen liittyviin rakennemalleihin, kuten periytymishierarkian esittämiseen tietokannassa. Tarkastelimme yksityiskohtaisesti eri ratkaisumallien soveltuvuutta eri tilanteisiin, ja yhteenvetona totesimme, että ortogonaalisen säilyvyyden ollessa tavoitteena on tärkeää, että olio-relaatiomuuntaja tukee lähes kaikkia näistä esitetyistä malleista.

Luku viisi käsitteli olion identiteettiä, inkrementaalista lataamista ja transaktioita. Toimme esille identiteetin eroavaisuudet olio- ja relaatiomallissa, tarkastelimme keinoja inkrementaalisen lataamisen – jonka totesimme olevan välttämätön ominaisuus olio-relaatiomuuntajassa – toteuttamiseen. Lisäksi perehdyimme transaktionaalisuuteen, joka oli yksi luvussa kaksi esittämistämme vaatimuksista.

Jatkotutkimusaiheena voitaisiin tarkastella, millä tasolla eri olio-ohjelmointikielet ja sovelluskehykset tukevat ortogonaalista säilyvyyttä. Erityisesti olisi mielenkiintoista tarkastella dynaamisesti ja staattisesti tyypitettyjen olio-ohjelmointikielten eroavaisuuksia tässä suhteessa. Myös

tuottavuuden ja ylläpidettävyyden vertaaminen eri arkkitehtuurisia malleja (aktiivinen tietue, olio-relaatiomuuntaja) käytettäessä olisi mielenkiintoista.

7 LÄHDELUETTELO

Ambler S., 2002 [online] The Object-Relational Impedance Mismatch [viitattu 19.10.2003]. Saatavilla www:ssä osoitteessa <http://www.agiledata.org/essays/impedanceMismatch.html>

Ambler S., 2003 [online] The Fundamentals of Mapping Objects to Relational Databases [viitattu 19.10.2003]. Saatavilla www:ssä osoitteessa <http://www.agiledata.org/essays/mappingObjects.html>

Apache Software Foundation, 2004 [online] Object/Relational Bridge – OJB [viitattu 22.11.2004]. Saatavilla www:ssä osoitteessa <http://db.apache.org/ojb/>

Atkinson M. & Bancilhon F., DeWitt D., Dittrich K., Mater D., Zdonik S., 1990. The Object-Oriented Database System Manifesto. Proceedings of the First International Conference on Deductive and Object-Oriented Databases. 223-240.

Atkinson M. & Morrison R., 1995. Orthogonally Persistent Object Systems. The VLDB Journal – The International Journal on Very Large Data Bases, Volume 4, Issue 3 (July 1995). 319-402.

Atkinson M., 1999. Persistence and Java – A Balancing Act. Proceedings of the International Symposium on Objects and Databases. 1-31.

Atkinson M. & Jordan M., 1998. [online] Orthogonal Persistence for Java – A mid-term Report [viitattu 6.2.2004]. Saatavilla www:ssä osoitteessa <http://www.sunlabs.com/research/forest/com.sun.labs.pjw3.main.html>

Atkinson M. & Jordan M., 1999.[online] Orthogonal Persistence for the Java Platform – Draft Specification [viitattu 6.2.2004]. Saatavilla www:ssä osoitteessa

<http://www.sunlabs.com/forest/COM.Sun.Labs.Forest.doc.external_www.papers.html>

Bauer C. & King G., 2004. Hibernate in Action. Manning Publications.

Bloch J., 2001. Effective Java: Programming Language Guide. Addison Wesley.

Brown K. & Whitenack B., 1996. Crossing Chasms: A Pattern Language for Object-Relational Integration. Pattern Languages of Program Design 2 , Vlissides, Coplien and Kerth, eds., Addison-Wesley, 1996.

Cabibbo L. & Porcelli R., 2004. M2ORM2: A Model for the Transparent Management of Relationally Persistent Objects. 9th International Workshop, DBPL 2003, Potsdam, Germany. 166-178.

Carey M. & DeWitt D., 1996. Of Objects and Databases: A Decade of Turmoil. Proceedings of the 22nd VLDB Conference Mumbai (Bombay), India.

Copeland G. & Khoshafian S. 1986. Object Identity, Conference proceedings on OOPSLA, Portland, Oregon, United States. 406-416.

Evans E., 2003. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley.

EWeek, 2003. [online] Database Market Grows in Europe [Viitattu 1.1.2004]. Saatavilla www:ssä osoitteessa: <www.eweek.com/article2/0,3959,1225621,00.asp>

ExoLab Group, 2004 [online] The Castor Project [viitattu 22.11.2004]. Saatavilla www:ssä osoitteessa < http://castor.exolab.org/>

Fowler M., 2003 [online] Anemic Domain Model [viitattu 31.12.2003]. Saatavilla www:ssä osoitteessa <http://martinfowler.com/bliki/AnemicDomainModel.html >

Fowler M., 2002. Patterns of Enterprise Application Architecture. Addison-Wesley.

Hibernate, 2003. [online] Feature List [Viitattu 21.12.2003]. Saatavilla www:ssä osoitteessa <http://www.hibernate.org/4.html#A18>

Hohmann L., 2003. Beyond Software Architecture: Creating And Sustaining Winning Solutions. Addison-Wesley.

Johnson R., 2002. J2EE Design and Development. Wrox.

Kassem N. ym., 2000. [online]Designing Enterprise Applications with Java2 Platform, Enterprise Edition [viitattu 4.5.2004]. Saatavilla www:ssä osoitteessa <http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/index.html>

Keller W., 1997. Mapping Objects to Tables: A Pattern Language. Proceedings of the 1997 European Pattern Languages of Programming Conferences, Germany.

Lunney T. & McCaughey A., 2003. Object Persistence in Java. Proceedings of the 2nd international conference on Principles and practice of programming in Java, Kilkenny City, Ireland. 115-120.

Marquez A., Blackburn S., Mercer G. & Zigman J., 2001. [online] Implementing Orthogonally Persistent Java [viitattu 20.5.2004]. Saatavilla www:ssä osoitteessa < http://cs.anu.edu.au/~Steve.Blackburn/pubs/papers/opj-pos9.pdf>

Meyer B., 1997. Object-Oriented Software Construction, 2nd Edition. Prentice Hall.

Nock C., 2003. Data Access Patterns. Addison-Wesley.

Oracle Corporation, 2004. [online] Oracle Toplink [Viitattu 12.10.2004] Saatavilla www:sssä osoitteessa <http://www.oracle.com/technology/products/ias/toplink/index.html>

Roman, Ambler & Jewell, 2002. Mastering Enterprise JavaBeans 2nd edition. John Wiley & Sons, Inc.

Roos R., 2003. Java Data Objects. Addison-Wesley.

Sun Microsystems, 2003. [online] Java Data Objects (JDO) Specification 1.0.1 [Viitattu 20.12.2003]. Saatavilla www:ssä osoitteessa <http://access1.sun.com/jdo/>

Sun Microsystems, 2001. [online] Enterprise JavaBeans Specification , Version 2.0 [Viitattu 4.1.2004]. Saatavilla www:ssä osoitteessa <http://java.sun.com/products/ejb/docs.html>

Sun Microsystems, 2002. [online] Core J2EE Patterns – Data Access Object [Viitattu 31.12.2003]. Saatavilla www:ssä osoitteessa <http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html >

Srinivasan V. & Chang D.T. 1997. [online] Object persistence in object-oriented applications [Viitattu 22.11.2004]. Saatavilla www:ssä osoitteessa <http://www.research.ibm.com/journal/sj/361/srinivasan.html >

Tate B., 2004. [online] For JDO, the Time Is Now [Viitattu 10.5.2004]. Saatavilla www:ssä osoitteessa <http://www.devx.com/Java/Article/20422/1954?pf=true>

Thought Inc, 2004. [online] Cocobase [Viitattu 12.10.2004]. Saatavilla www:ssä osoitteessa <http://www.thoughtinc.com/cber_index.html>

Wieringa R. & de Jonge W., 1995. Object identifiers, keys, and surrogates – object identifiers revisited. Theory and Practice of Object Systems, 1(2) 101-114.

Wirfs-Brock R. & McKean A., 2003. Object Design: Roles, Responsibilities, and Collabrations. Addison-Wesley.

Wuestefeld K., 2003. Do you still use a database? Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications. 101.