Java ja olioiden pysyvyys

Jarkko Rouvinen

31.8.2004

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

Tiivistelmä

Pysyväksi olioksi sanotaan oliota, joka voi säilyttää tilansa sovelluksen suorituskertojen välillä. Pysyvä olio on täten riippumaton sovelluksen elinkaaresta. Pysyvyyteen kehitettyä universaalia ja riippumatonta teoriaa kutsutaan ortogonaaliseksi persistenssiksi. Olioiden pysyvyyden toteuttamiseen on esitetty lukuisia ohjelmointikieli-, alusta- ja toimittajariippuvaisia sekä -riippumattomia tekniikoita. Olioiden tietoja voidaan tallentaa binaaritiedostoihin, tekstitiedostoihin ja tietokantoihin. Useimmiten jokaisen tietolähteen käyttö vaatii omanlaisen tekniikan sovelluksen ohjelmoinnissa.

Tässä pro gradu -tutkielmassa pyritään selvittämään, kuinka olioiden pysyvyys voidaan toteuttaa Java-kielessä. Tutkielman alussa tutkimuskenttään tutustutaan laajemmin kirjallisuuden avulla, jonka jälkeen esitellään neljä erilaista J2SE-tason tekniikkaa. Tutkielmassa käsiteltävät tekniikat ovat Javan olioiden sarjallistaminen, XML-sidonnan arkkitehtuuri, tietokantarajapinta ja dataoliot. Kustakin tekniikasta selvitetään niiden käytön periaatteet, edut ja haitat sovelluskehityksessä sekä soveltuvuusalue. Käytännön esimerkkejä olioiden pysyvyyden toteuttamisesta esitetään laaditun esimerkkisovelluksen lähdekoodin avulla.

ACM-luokat (ACM Computing Classification System, 1998 version): D.1.5, E.2, E.5, H2

Avainsanat: Java, pysyvä olio, ortogonaalinen persistenssi, olioiden sarjallistaminen, XML-sidonta, tietokantarajapinta, dataoliot

Sisältö

1 Johdanto

Tietokoneissa on periaatteessa kahdenlaista muistia. Primaarisessa säilytystilassa, keskusmuistissa, säilytetään sovelluksien ajonaikaisia tietoja. Keskusmuisti on nopeaa, mutta sen sisältö pyyhkiytyy tyhjäksi aina virrankatkaisun yhteydessä. Sekundaariseksi säilytystilaksi sanotaan talletustilaa, joka on primaarista hitaampaa, mutta jonne kirjoitetut tiedot säilyvät virrankatkaisusta huolimatta. Näin ollen sekundaarisessa talletustilassa olevaa dataa voidaan nimittää pysyväksi tiedoksi.

Pysyvät tiedot ovat useiden sovellusten ydin. Sovellukset käyttävät ja ylläpitävät järjestelmään tallennettuja tietoja. Liiketoimintasäännöt, teknologiat sekä järjestelmät ikääntyvät ja vaihtuvat, mutta sen sijaan tiedot hylätään harvoin. Päinvastoin, vanhat tiedot yleensä sulautetaan osaksi uusia järjestelmiä. Kuitenkin vaikka data ei katoa, sitä käsittelevät tekniikat muuttuvat.

Tiedon tallentamista erilaisiin tiedostorakenteisiin tutkittiin jo 1960-luvulla (Dzubak & Warburton, 1965). Tietojen tallennuksessa käytettiin organisoituja rakenteita ja tällöin luotiin käsitteet järjestämätön tiedosto, peräkkäistiedosto ja hajatiedosto (esim. Connolly & Begg, 2005). Tiedon määrän kasvaessa tiedostopohjaiset ratkaisut todettiin riittämättömäksi ja 1970-luvulla ratkaisuksi kehiteltiin relaatiotietokannat (Sperko, 2003). Myöhemmin tulivat vielä oliotietokannat (Cattell & al., 2000).

Tiedon tallennustekniikan ohella myös ohjelmointiparadigmat ovat kehittyneet. Järjestelmien ja niihin liittyvien vaatimusten kasvaessa on proseduraalisista ja funktionaalisista ajatusmalleista siirrytty olioparadigmaan. Tämän johdosta ohjelmointikielet, kuten Pascal, ovat korvaantumassa kokonaisuuksia paremmin käsittelevillä olio-ohjelmointikielillä, kuten esimerkiksi Javalla (Koskimies, 2000). Olioparadigma on kuitenkin tuonut omat erityispiirteensä tiedon tallentamiseen.

Olioiden tietojen tallentamiseksi, eli olioiden pysyvyyden toteuttamiseksi, on esitetty useita vaihtoehtoja. Kaikilla olio-ohjelmointikielillä on omat erityispiirteensä ja teknologiansa. Sovelluksessa pysyvyyden toteuttamiseen käytettävän tekniikan valinnassa täytyy ottaa huomioon useita tekijöitä, koska sovelluksen vaatimukset asettavat epäsuorasti vaatimuksia myös

käytettävälle teknologialle. Baldwinia (2003) mukaillen joudutaan harkitsemaan muun muassa, että halutaanko tietoa jakaa useiden sovellusten kesken, onko säilytettävä tieto vain ohjelman tilaa käsittelevää vai tarvitaanko monimutkaisia kyselyitä, sekä mitkä ovat talletettavan ja käsiteltävän tiedon volyymit. Näin ollen sovelluksessa käytettävän tekniikan valinta ei ole yksiselitteistä.

Java tarjoaa joukon tekniikoita toteuttaa tietojen tallentamisen ja palauttamisen ohjelman käyttöön. Tämän tutkielman tekemisellä haluan ottaa selvää näistä mahdollisuuksista, koska Java on monipuolinen, kehittyvä ja alati suositumpi olio-ohjelmointikieli. Toivon tämän tutkielman hyödyttävän ohjelmistotuottajia havaitsemaan ensinnäkin olioiden pysyvyyden moniulotteisuuden, toiseksi esittelemään Javan yleisimmät tekniikat ja kolmanneksi näkemään näiden tekniikoiden suurimmat eroavaisuudet.

1.1 Tutkimusongelma

Tutkielman päämääränä on selvittää, miten olioiden pysyvyys voidaan toteuttaa Java-kielessä. Pyrin löytämään yleisimmät tekniikat, niiden käytön periaatteet, ominaisuudet, vaatimukset, käyttökohteet sekä edut ja haitat. Tutkielma etsii vastausta kysymyksiin kuinka paljon tekniikat eroavat toisistaan opittavuudeltaan, käyttötarkoituksiltaan ja miten niitä käyttävät järjestelmät poikkeavat toisistaan toteutukseltaan ja ylläpidettävyydeltään.

Tutkielma on rajattu Java 2 Standard Edition (J2SE) -tason tarjoamiin välineisiin olioiden pysyvyyden toteuttamisessa. Edelleen tutkimuksen ulkopuolelle jäävät Java 2 Micro Edition (J2ME) ja Java 2 Enterprise Edition (J2EE) -tason ratkaisut. Näin ollen esimerkiksi Enterprise JavaBeans (EJB) ja Record Management System (RMS) -teknologiat jäävät tutkielmassa käsittelemättä. Syy rajaukseen on työmäärän kohtuullistaminen sekä J2SE-tason muodostama luonnollinen rajaus.

Javan laajennukset, kuten PJama tai AspectJ, sekä kaupalliset ja avoimen lähdekoodin tuotteet olioiden pysyvyyden toteuttamisessa on myös rajattu pois. Näkökulma on itse Java-kielessä, sen standardeissa ja sovelluksen ohjelmoijan työssä. Tutkielma keskittyy Javan omien, Sun Microsystemsin tukemien, työkalujen esittelyyn ja vertailuun.

1.2 Materiaali ja metodologia

Materiaalina teoreettisessa tarkastelussa käytän olioiden pysyvyydestä kirjoitettuja tieteellisiä artikkeleita ja konferenssiesitelmiä, jotka ovat saatavilla muun muassa Academic Pressin, ACM:n ja IEEE/IEE:n tietokannoista. Yksittäisistä tekniikoista kertovina lähteinä käytän myös niistä laadittuja käsikirjoja. Lisäksi katson luotettavaksi Sun Microsystemsin Internet-sivustoilla olevat tiedot ja dokumentaatiot sekä niissä viitatut artikkelit ja sivustot.

Metodologialtaan tutkielma on konstruktiivinen, komparatiivinen ja se sisältää empiirisen osuuden. Tutkielmaa varten on tehty kokeellinen esimerkkisovellus, joka on dokumentoitu tutkielmasta riippumattomana osana, tietojenkäsittelytieteen laudatur-harjoitustyönä (Rouvinen, 2004). Esimerkkisovelluksen rakenne on esitetty tämän tutkielman liitteen 1 luokkakaaviossa. Sovelluksen tarkoituksena on osoittaa olioiden pysyvyyden todellinen implementointi käyttäen tutkielmassa esiteltyjä tekniikoita. Sovelluksen tekemisellä ei pyritty selvittämään mitattavissa olevia kvantitatiivisia asioita tekniikoista, vaan yritettiin saada kvalitatiivista tietoa niiden opittavuudesta, käytöstä ja ylläpidettävyydestä. Saatujen kokemusten ja kirjallisuuden perusteella yritetään osoittaa sovelluskehittäjän kannalta olennaisimmat tekniikoiden väliset eroavaisuudet.

1.3 Tutkielman rakenne

Tämä pro gradu -tutkielma koostuu johdantoluvun lisäksi seitsemästä muusta luvusta. Luvussa 2 esitellään tutkielman lähtökohtia tutustumalla tutkielman viitekehykseen, pysyvien tietojen tutkimuskenttään ja olioiden pysyvyyteen liittyviin teorioihin. Lisäksi esitetään perustelut tutkittavien Java-teknologioiden valinnalle. Luvut 3, 4, 5, ja 6 käsittelevät kukin yhtä J2SE-tason teknologiaa, jotka mahdollistavat Java-olioiden pysyvyyden. Tarkasteltavat tekniikat ovat olioiden sarjallistaminen (Java Object Serialization, JOS), XML-sidonnan arkkitehtuuri (Java Architecture for XML Binding, JAXB), tietokantarajapinta (Java Database Connectivity, JDBC) ja dataoliot (Java Data Objects, JDO). Luvussa 7 vertaillaan edellä mainittujen teknologioiden piirteitä. Tutkielman johtopäätökset esitetään luvussa 8.

2 Olioiden pysyvyys

Tässä luvussa luodaan katsaus tutkielmaan liittyvään kirjallisuuteen. Aluksi kohdissa 2.1 ja 2.2 esitellään muutamia tutkielmassa yleisesti käytettyjä olio-ohjelmoinnin ja olioiden pysyvyyden peruskäsitteitä. Tämän jälkeen kohdassa 2.3 tarkastellaan olioparadigman ja olioiden pysyvyyden välistä suhdetta. Luvun jälkipuoliskolla kohdassa 2.4 tutustutaan teoriaan nimeltä ortogonaalinen persistenssi ja kohdassa 2.5 syvennytään olioiden pysyvyyden toteuttamiseen.

2.1 Olioparadigman termistöä

Seuraavaksi määritellään tutkielman kannalta oleelliset olioparadigman käsitteet. Keskeisiä termejä ovat olio, luokka, ilmentymä, tila, olion elinikä ja pysyvät oliot.

2.1.1 Olio

Olioksi (object) voidaan Koskimiehen (2000) mukaan sanoa ympäristöstään erottuvaa kokonaisuutta, jolla olla on oma identiteetti, sisäinen rakenne ja suhde tiettyyn ympäristöön. Käsitteenä olio on niin yleinen, että se voi edustaa lähes mitä hyvänsä ohjelman osaa tai sovelluksen käsitettä. Teknisesti ajatellen olio on ohjelman strukturaalinen perusyksikkö, joka säilyttää tietoa ja sillä on tiedon käsittelyyn liittyviä toimintoja.

2.1.2 Luokka, ilmentymä, attribuutti ja operaatio

Oliokielessä, kuten Javassa, olioiden ominaisuudet määritellään pääsääntöisesti olion *luokassa* (class). Täten jokaisella oliolla on yksikäsitteinen luokka, jonka mukaan olio on luotu. Luokka on samankaltaisten olioiden malli (Laine, 1996). Olio on tämän luokan *ilmentymä* (instance). Luokka kertoo, mitä tietoa ja toimintoja sen ilmentymillä on. Tietoja kutsutaan *attribuuteiksi* ja toimintoja *metodeiksi* tai *operaatioiksi* (Koskimies, 2000). Luokkaa voidaan ajatella esimerkiksi aihiona, jonka mukaan oliot luodaan.

2.1.3 Olion tila

Olio voi tallentaa tietoa olion attribuutteihin, nimettyihin tietokenttiin. Attribuuttien arvojen yhdistelmiä kutsutaan olion *tiloiksi* (state) (Koskimies, 2000). Tila on vaihe olion *elinkaaressa* (Laine, 1996). Olion tila voi muuttua operaation suorituksen johdosta. Kaikilla luokan ilmentymillä on samat attribuutit ja operaatiot, mutta olion tilan muodostavien attribuuttien arvot voivat vaihdella.

2.1.4 Olion elinikä ja elinkaari

Olion *elinikä* tarkoittaa sitä olion ominaisuutta, joka määrää, miten pitkään luokan ilmentymä on olemassa luontinsa jälkeen (Koskimies, 2000). Vastaavasti *elinkaareksi* (life-cycle) kutsutaan erilaisista tiloista muodostuvaa polkua olion synnystä sen poistumiseen (Laine, 1996).

Koskimiestä (2000) lainaten olio on olemassa luontinsa jälkeen, kunnes se hävitetään ohjelmoijan toimesta eksplisiittisesti tai järjestelmän toimesta automaattisesti. Olioita voidaan kategorioida niiden eliniän perusteella. *Väliaikaiseksi* sanotaan oliota, jotka tuhotaan järjestelmän toimesta heti, kun sitä ei enää tarvita siinä operaatiossa, jota varten se on luotu. *Alistettu olio* hävitetään jonkin toisen olion hävittämisen yhteydessä ja *paikallinen olio* lohkosta tai rutiinista poistumisen yhteydessä. *Vapaat oliot* ovat viittaamattomia, ja siten hyödyttömiä, olioita, jotka järjestelmä tuhoaa *roskien keruussa*. *Staattinen olio* säilyy kunnes ohjelma päättyy. Mikäli olio jää olemaan ohjelman suorituksen jälkeen, sitä kutsutaan *pysyväksi* eli *persistensiksi olioksi* (persistent object). Tässä tutkielmassa keskitytään pysyviin olioihin.

2.1.5 Pysyvät oliot

Persistenssillä tarkoitetaan pysyvyyttä tai säilyvyyttä. Olioiden tapauksessa persistenssi tarkoittaa yksittäisen olion elinkaaren riippumattomuutta sen luoneen ohjelman suorituksesta (Atkinson & al., 1996a). Olion tiedot, siis attribuuttien arvot, säilyvät vaikka ohjelman suoritus lakkaisi. Nämä tiedot voidaan palauttaa järjestelmän käyttöön myöhemmin.

Java-oliot häviävät normaalisti järjestelmän suorittamassa roskien keruussa, jos ne ovat muuttuneet vapaiksi olioiksi (Koskimies, 2000). Mikäli Java-olio säilyy läpi ohjelman suorituksen, se joka tapauksessa hävitetään järjestelmän toimesta ohjelman suorituksen päättyessä.

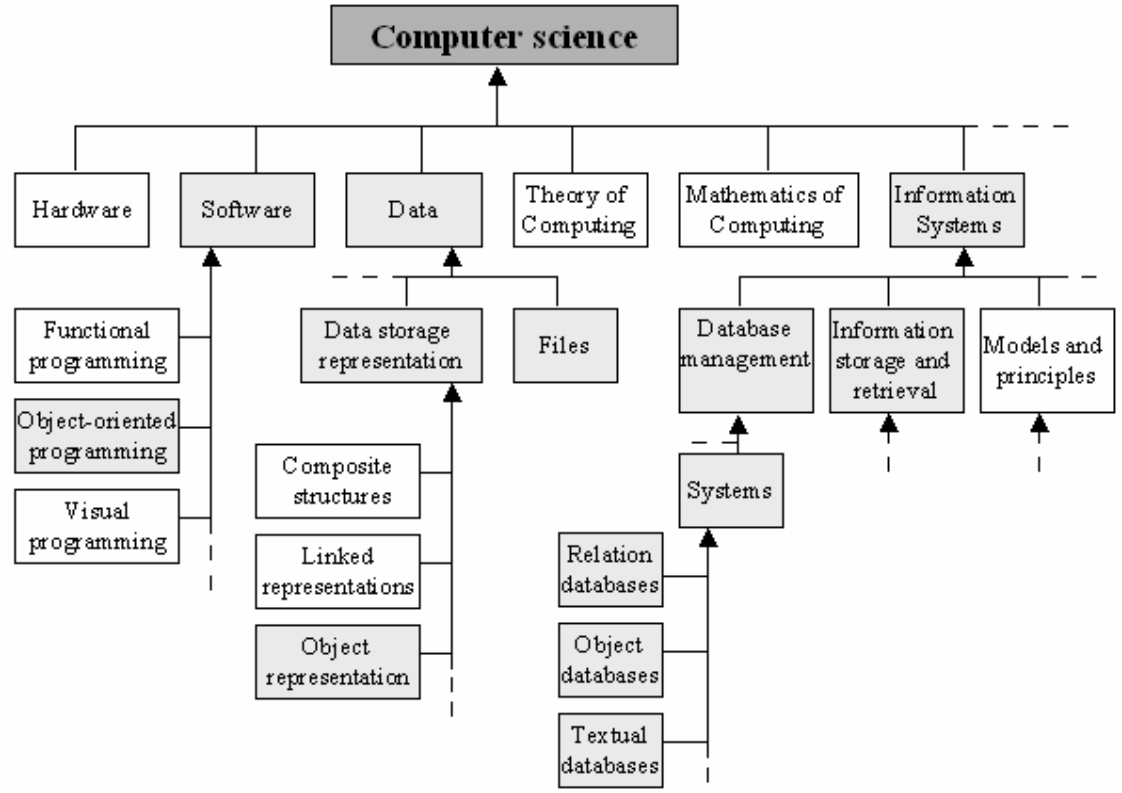
2.2 Olioiden pysyvyyden tutkimuksesta

Olioiden pysyvyys voidaan määritellä yksiselitteisesti, mutta olioiden tai tiedon säilyvyyttä tutkimusalueena ei. Pysyvien järjestelmien tutkimus ei ole yksiulotteista. Seuraavassa esitellään perusteita näille väittämille.

Tiedon pysyvyyteen liittyvät tutkimukset käsittelevät useaa tietojenkäsittelytieteen osa-aluetta, kuten esimerkiksi ohjelmointikielen toteutusta, algoritmien suunnittelua, ohjelmointia, tiedonhallintaa ja hajautettuja järjestelmiä (Persistent Programming Research Group, 1998). Yleisesti ottaen tutkimukset keskittyvät pysyvien järjestelmien suunnitteluun, toteutukseen ja käyttöön. Ne pyrkivät selvittämään, kuinka parhaiten voidaan tukea pitkäikäisten, tietointensiivisten tietojärjestelmien toteutusta. Hienojakoisemmin tutkimuskenttä voidaan esittää esimerkiksi seuraavasti (Connor & Nettles, 1996; Persistent Programming Research Group, 1998):

- suuret pysyvät järjestelmät (large scale persistent systems)
- pysyvien olioiden varastointi (persistent object stores)
- metodologiat pysyvyyden ohjelmointiin (methodologies for persistent programming)
- pysyvyyttä tukevat arkkitehtuurit (architectures for supporting persistence)
- tietotyypit ja tiedon pysyvyys (type systems and persistence)
- pysyvän tiedon kehitys (evolution of persistent data)
- pysyvät sovellukset ja pysyvän teknologian sovellukset (persistent applications and applications of persistence technology)
- yhdenmukaisuus, pysyvyys ja tapahtumat (concurrency, persistence and transactions)
- työkalut pysyvien järjestelmien hallintaan (tools for the management of persistent systems)
- pysyvyyttä tukevat lähdekoodin kääntäjät (compilers for languages supporting persistence)
- pysyvyys ja tietokannan ohjelmointikielet (persistent and database programming languages)
- pysyvyys ja WWW (persistence and the WWW).

Tämä tutkielma keskittyy sellaisiin Javan tekniikoihin, joilla sovellukset voivat luoda pysyviä olioita. Edellä esitetyssä listassa tämä sijoittuu metodologioiden sekä pysyvän teknologian sovelluksien osa-alueisiin. Tässä tutkielmassa ei käsitellä näiden tekniikoiden ohjelmoijalta piilotettuja toteuttamisaspekteja tai eroavaisuuksia. Tutkielman aihe on täten vain suppea osajoukko tiedon säilyvyyteen liittyvästä tutkimustyöstä. Kuvassa 2.1 on esitetty tutkielman aiheen sijoittuminen vuoden 1998 ACM-luokitukseen (Association for Computing Machinery, 2003). Kuvan tarkoituksena on hahmottaa lukijalle tutkielman yleinen viitekehys.



Kuva 2.1: Olioiden pysyvyys ja ACM-luokitus

Kuvassa 2.1 esitetty jaottelu ei ole täydellinen ACM-luokittelu. Harmaalla ovat merkittyinä ne tietojenkäsittelyn osa-alueet, joita tutkielma sivuaa. Katkoviiva kuvaa, että ACM-luokittelun sisältö on esitettyä laajempi, mutta kyseiset osat eivät enää kuulu tutkielman aiheeseen.

2.3 Olioparadigma ja olioiden pysyvyys

Olioiden pysyvyys ei ole suoraviivaista olion attribuuttien tallentamista. Pysyviä olioita käsitellessä joudutaan huomioimaan muun muassa olioiden identifioimiseen, olioiden välisiin yhteyksiin, tiedon kapselointiin ja luokkien periytymiseen liittyviä tekijöitä. Tässä kohdassa on lyhyt katsaus siitä, mitä seikkoja joudutaan huomioimaan, kun olioista tehdään pysyviä.

Jokaisella oliolla on sen identifioiva tunniste, *viite* (Koskimies, 2000). Olioiden identiteetin tunnistaminen on eräs tärkeimmistä asioista käsitellessä pysyviä olioita (Srinivasan & Chang, 1997). Luokasta voidaan tehdä rajaton määrä ilmentymiä, joiden tilat ovat samat, mutta niillä on eri viite. Mikäli nämä oliot tehdään pysyviksi, tulee tallennusmediatasolla huomioida, että nämä ilmentymät ovat erillisiä, eivätkä ehkä saa korvata toisiaan, vaikkakin niiden tiedot ovat samoja. Vastaavasti olioita palautettaessa ohjelman käyttöön, tulisi voida identifioida samansisältöiset oliot eri ilmentymiksi.

Olio koostuu harvoin vain primitiivityyppisistä attribuuteista. Olion attribuutin arvona voi olla viite toiseen olioon. Esimerkiksi luokalla `Henkilö` voi olla attribuutteina `puoliso` tyyppiä `Henkilö`, `auto` tyyppiä `Auto` ja `puhelinnumerot` tyyppiä `Merkkijonotaulukko`. Edelleen luokalla `Auto` voi olla attribuutteinaan mielivaltainen määrä viittauksia muihin olioihin. Mikäli `Henkilö`-luokan ilmentymä tallennetaan, tulee myös edellä mainitut olioviittaukset tallentaa. Yleisemmällä tasolla voidaan sanoa, että olioiden tallennuksessa tulee aina huomioida niiden väliset yhteydet ja tunnistaa kaikki pysyvät oliot yhteneväksi kokonaisuudeksi. Tarvitaan keinot huolehtia viittauksien rekursiivisista käsittelyistä, erilaisten suhteiden havaitsemisesta ja mahdollisista tiedon konversioista tallennusmedian ja ohjelman käyttämän esitystavan välillä (Srinivasan & Chang, 1997).

Periytymisellä olioparadigmassa tarkoitetaan luokan `A` piirteiden siirtymistä jollekin toiselle luokalle `B` (Koskimies, 2000). Tällöin voidaan sanoa `A`:n olevan *yliluokka* luokalle `B`. Oliota tallennettaessa joudutaan tallentamaan myös yliluokkiin kuuluvat tiedot. Ne ovat osa käsiteltävän olion tilaa. Yliluokista täytyy huomioida vähintään attribuutit, mutta riippuen tallennustavasta, myös yliluokkien rakenne voidaan joutua tallentamaan (Srinivasan & Chang, 1997).

Tiedon ja siihen liittyvien toimintojen pakkaamista yhdeksi suojatuksi kokonaisuudeksi kutsutaan mukaan ohjelmistotekniikassa usein *kapseloinniksi* (encapsulation) (Koskimies, 2000).

Myös luokkien voidaan ajatella olevan kapseloituja. Tehtäessä olioista pysyviä, joudutaan harkitsemaan, kuinka kapseloituun tietoon päästään käsiksi (Srinivasan & Chang, 1997). Tarjoaako luokka saantimetodit tarvittaviin attribuutteihin vai ovatko attribuutit julkisia ja miten ei-pysyvät attribuutit piilotetaan pysyvyyden toteutukselta.

2.4 Ortogonaalinen persistenssi

Edellä on käsitelty olioparadigman ja pysyvien olioiden välistä suhdetta. Seuraavaksi tutustutaan teoriaan, joka määrittelee tavoiteltavia ominaisuuksia pysyville järjestelmille. Teorian esittämät säännöt ovat ohjelmointiparadigmasta riippumattomia, mutta teoria on huomioitu esimerkiksi Javan kehityksessä.

Vuonna 1983 Atkinson, Bailey ja Chisholm laativat artikkelin tiedon pysyvyyttä tukevista periaatteista. Tarkoituksena oli luoda systeemi, jolla voitaisiin tarjota sovelluksille tiedon pysyvyys ohjelmointikielen tarjoamana palveluna. Ideaa simuloimaan työryhmä kehitti S-algol-kielen päälle ohjelmointikielen PS-algol (Atkinson & al., 1983). Samaisessa artikkelissa työryhmä määritteli *ortogonaalisen persistenssin* eli järjestelmän riippumattoman pysyvyyden. Riippumattoman pysyvyyden periaatteiden mukaisesti suunnitellut ja toteutetut järjestelmät käsittelevät tietoa samalla tavalla riippumatta tiedon eliniästä, tyypistä tai koosta (Atkinson & al., 1983). Oliotasolla voidaan ajatella, että pysyvyys olion ominaisuutena on riippumaton kaikista muista olion ominaisuuksista (Atkinson & al., 1996a).

Riippumattoman pysyvyyden periaatteita on myöhemmin sovellettu kokeellisesti eri ohjelmointikielisissä toteutuksissa, esimerkiksi Modula-3:n PM3-laajennoksessa (Hosking & Chen, 1999). Java-kieleen ortogonaalisen persistenssin periaatteet integroitiin 1990-luvun puolivälissä, kun Sun Microsystems ja Glasgowin yliopisto ryhtyivät yhteistyössä tutkimaan olioiden pysyvyyttä. Tämä yhteistyö synnytti Java-kieleen kokeellisen ympäristön PJava[1] (Atkinson & al., 1996a). Ajatuksena oli tehdä tiedon pysyvyydestä ohjelmointitasolla huomaamaton ominaisuus Java-kieleen. Näin pyrittiin minimoimaan sovelluskehittäjän työ olioita tallennettaessa ja palautettaessa. Huolimatta useiden vuosien kehityksestä, tämä tutkimuskäyttöön suunnattu kokeilu ei kuitenkaan levinnyt yleiseen käyttöön.

[1] Pjava tunnetaan nykyisin nimellä PJama (Sun Microsystems, 2000c).

2.4.1 Ortogonaalisen persistenssin periaatteet

Ortogonaalisen persistenssin tavoite on tarjota yksikertainen, yhdenmukainen, malli sovelluksiin, jotka käsittelevät pitkäikäistä dataa (Atkinson & al., 1996a; Atkinson & al., 1996b). Vastuu tietojen säilyvyydestä asetetaan paljolti sovelluksen ympäristön vastuulle, kuten esimerkiksi Javan virtuaalikoneelle (McCaughey & Lunney, 2003). Malli sisältää kolme seuraavaksi esiteltävää periaatetta (Atkinsson & al., 1983; Atkinsson & al., 1996a): riippumaton pysyvyys, tietotyyppien ortogonaalisuus ja pysyvyyden tunnistaminen.

Riippumattoman pysyvyyden (persistence independence) periaatteen mukaan kaikella ohjelmakoodilla tulisi olla sama muoto, riippumatta sen käsittelemän tiedon kestosta (Atkinson & al., 1996a). Pitäisi olla esimerkiksi mahdollista kutsua samaa proseduuria käyttäen argumenttina niin pysyvää kuin myös väliaikaista oliota. Lisäksi periaatteesta voidaan johtaa, että tiedon siirtämistä tietolähteestä ohjelman käyttöön ja takaisin ei tulisi ohjelmoida eksplisiittisesti (McCaughey & Lunney, 2003). Matalan tason operaatiot piilotetaan sovelluksen kehitystasolta. Tällöin ohjelmoijan ei tarvitse tehdä lähdekoodiin merkittäviä muutoksia saadakseen ohjelman käyttämien luokkien esiintymistä pysyviä.

Tietotyyppien ortogonaalisuuden (data type orthogonality) periaatteen mukaan tiedon tyyppi ei saa vaikuttaa sen pysyvyyteen (Atkinsson & al., 1983). Kaikille tietotyypeille on tarjottava samat elinkaaret. Javassa on kahdenlaisia tietotyyppejä: *primitiivityyppejä* (primitive types) ja *osoitintyyppejä* (reference types) (Koskimies, 2000). Primitiivityyppisiä muuttujia ovat esimerkiksi `byte`, `boolean` ja `int`. Osoitintyyppisten muuttujien sisältöinä ovat taas oliot ja olion viittaukset toisiin olioihin. Periaatteen mukaan pysyvyys on taattava kaiken tyyppiselle tiedolle.

Pysyvyyden tunnistamisen (persistence identification) periaatteen mukaan pitkäikäisen tiedon tunnistamiseen täytyy olla suoraviivainen ja johdonmukainen mekanismi (Atkinson & al., 1983). Tunnistaminen on prosessi, millä selvitetään, mitkä kaikki oliot tulee säilyttää tallennettavan olion mukana niiden välisten yhteyksien johdosta. Tästä periaatteesta esitetään kirjallisuudessa myös kuvaavampi muoto *transitiivinen pysyvyys* (Atkinson & al., 1996a). Tällä tarkoitetaan, että pysyviä olioita ovat myös kaikki ne oliot, joihin on viittaus pysyvästä oliosta. Näistä viittauksista puhutaan *saavutettavuutena*. Jos esimerkiksi olio `c`, luokan `C` ilmentymä, tehdään pysyväksi, on tarpeellista varmistaa, että kaikki tieto, jota voidaan tarvita `c`:n

operointiin ja joka ei ole jo pysyvää, tehdään samanaikaisesti pysyväksi (Atkinson & al., 1996a). Tämä lisätieto sisältää luokan `C`, kaikki `C`:n yliluokat, kaikki niiden *staattisiksi*[2] määritellyt attribuutit ja saavutetut luokat sekä nykyiset `c`:n attribuuttien arvot mukaan lukien yliluokkien attribuuttien arvot.

2.4.2 Ortogonaalisen persistenssin merkityksestä

Ortogonaalisen persistenssin kolmen periaatteen kehityksen päämääränä Atkinsonin & al. (1996a) mukaan on ollut saavuttaa parempaa tuottavuutta ohjelmistotuotannossa: periaatteiden tavoitteena on yksinkertaisuus, ylläpidettävyys sekä työmäärän vähentäminen yksittäisten sovellusten kehitysvaiheissa. Käytännössä kuitenkin täydellisen ortogonaalisesti persistenssin järjestelmän kehittäminen on Hoskingin & Chenin (1999) mielestä erittäin vaikeaa. Täydellisen riippumattoman pysyvyyden ongelmana on, että usein sovelluksen ohjelmoijalle tulee antaa kontrolli, mitä olioita milloinkin noudetaan tietolähteestä sekä palautetaan tietolähteeseen. Antamalla kontrolli ohjelmoijalle, rikkoontuu riippumattomuuden periaate. Toinen heidän argumenttinsa on, että täydellinen tietotyyppien riippumattomuus ei käytännössä ole mahdollista eikä edes toivottavaa. Tietorakenteet voivat esimerkiksi sisältää ohjelman ajonaikaisia ominaisuuksia, kuten avattuja tiedostovirtoja tai tietoliikenneportteihin sidottuja soketteja. Näiden tallentaminen ei ole tarkoituksenmukaista.

Huolimatta edellä esitetyistä kritiikeistä, mielestäni ortogonaalinen persistenssi on teoriana merkittävä. Se kuvaa hyvin ne tiedon pysyvyyden perusaspektit ja tavoitteet, joita pysyvien tietojärjestelmien ja pysyvyyttä toteuttavien teknologioiden kehittelyssä tulisi ottaa huomioon. Ortogonaalisen persistenssin teoria on vaikuttanut myös Javan kehitykseen - PJaman muodossa. Teoriaa ei ehkä ole täysin mahdollista toteuttaa käytännössä, mutta ainakin se on hyvä periaatteellinen malli tiedon säilyvyydelle.

[2] Luokan staattinen (static) ominaisuus tarkoittaa luokassa määriteltyä piirrettä, joka ei liity yksittäisiin luokan ilmentymiin, vaan on kaikkien luokan ilmentymien yhteiskäytössä (Koskimies, 2000).

2.5 Pysyvyyden toteuttaminen

Edellä olioiden pysyvyyttä on käsitelty periaatteellisella tasolla. Tästä eteenpäin asiaa tarkastellaan käytännön toteutusten ja teknologioiden näkökulmasta.

Pysyvyyttä voidaan ajatella osana olioiden ominaisuutta tai ulkoisena palveluna, joka saavutetaan rajapintojen kautta. Tietokantajärjestelmien tarjoajat näkevät useimmiten olioiden pysyvyyden ohjelmointikielestä erillään olevana asiana, kun taas esimerkiksi Java-kieli itsessään pitää sitä kielen sisäisenä ominaisuutena. Jälkimmäiseen tapaukseen liittyy edelleen kysymyksiä: tulisiko olioiden pysyvyys toteutua pääosin kääntäjän, tulkin vai ohjelmoijan toimesta. Mikäli järjestelmän kehityksessä halutaan noudattaa ortogonaalinen persistenssin mallia, vastuu olioiden pysyvyydestä tulisi olla mahdollisimman vähän ohjelmoijan vastuulla.

Näkökulmaeroista ja erilaisista tarpeista johtuen olioiden pysyvyyden toteuttamiseen on olio-ohjelmoinnissa useita tekniikoita. Oliot voidaan tallentaa muun muassa binaaritiedostoina tai rakenteellisina tekstidokumentteina, kuten XML-tiedostoina, paikalliseen tiedostojärjestelmään, samassa verkossa olevien muiden koneiden muistiin tai tietokantaan. Tietokantavarastoinnissa mahdollisuutena on käyttää joko relaatiotietokannanhallintajärjestelmää, oliotietokannanhallintajärjestelmää tai oliorelaatiotietokannanhallintajärjestelmää. Eri tallennusjärjestelmät johtavat useimmiten sovellustasolla erilaisten toteutusteknologioiden valintaan.

Sovelluksen käyttämän olioiden tallentamistekniikan valinta on kriittinen osa sovelluskehitystä. Tällöin luodaan perusta sille, kuinka liiketoimintaoliot (business objects) viedään tietolähteeseen, palautetaan sieltä, kuinka niiden tiedot päivitetään sekä kuinka ne poistetaan tietolähteestä. Oikean tekniikan valitseminen on moniulotteinen ongelma. Ratkaisuun päästään valitsemalla tekniikka sen kokonaisvaltaisen sopivuuden mukaan. Sopivuuteen vaikuttavat seikat voidaan ryhmitellä kuuteen pääryhmään (Srinivasan & Chang, 1997; Baldwin, 2003; Sperko, 2003; Hosking & Chen, 1999; Connolly & Begg, 2005):

1. Kuinka liiketoimintaoliot muunnetaan tietolähteen edellyttämään muotoon? Tarvitseeko muunnosta toteuttaa sovelluksen ohjelmoinnin tasolla ja kuinka hienojakoista tiedon käsittely on?
2. Mikä on käytettävä kyselytekniikka, jolla oliot palautetaan tietolähteestä?

3. Miten tapahtumienhallinta hoidetaan? Kuinka varmistetaan tapahtumien ACID-ominaisuudet, eli tietoa käsittelevien tapahtumien atomisuus (atomicity), tiedon eheys (consistency), eristäminen (isolation) ja säilyvyys (durability).
4. Kuinka suorituskykyinen tekniikka on?
5. Kuinka ylläpidettävä tekniikka käyttävästä ratkaisusta saadaan? Mikä on ylläpidettävyys tiedon skeeman (schema), tallennusmedian tai tarvittavan kolmannen osapuolen toimittajan vaihtuessa?
6. Tekniikan yleinen käytettävyys, kuten ohjelmistokehityksen helppous, tarvittavan konfiguroinnin määrä ja testattavuus.

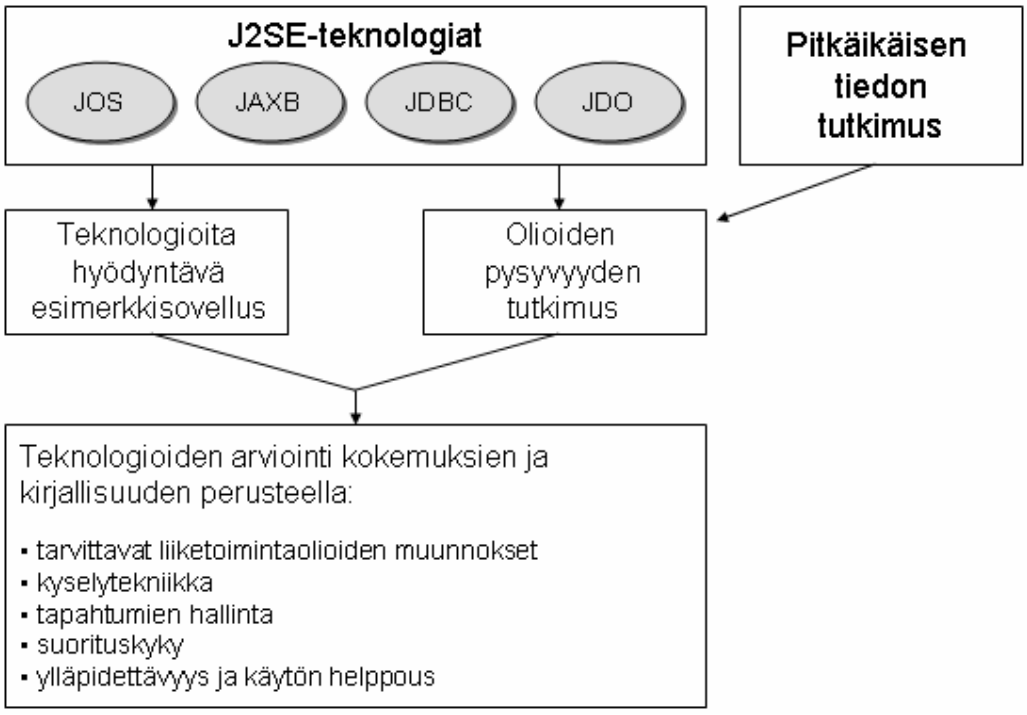
Näitä seikkoja voidaan käyttää indikaattoreina osoittamaan tekniikoiden teknistä soveltuvuutta. Toteutettavan sovelluksen vaatimuksia ja teknisiä piirteitä vertailemalla voidaan helpottaa päätöksentekoa. Vertailun kannalta relevantteja muuttujia tekniikoissa ovat siis tarvittavat liiketoimintaolioiden muunnokset, kyselytekniikka, tapahtumien hallinta, suorituskyky, ylläpidettävyys ja käytettävyys. Tutkimusongelmassani esittämiin tavoitteisiin kuuluu tutkittavien tekniikoiden eroavaisuuksien selvittäminen ja tulenkin kuvaamaan tekniikoita edellä mainittuja muuttujia käyttäen. Myöhemmin esittämiäni arvioita tekniikoista ei tulisi kuitenkaan pitää absoluuttisina totuuksina, juuri johtuen asian moniulotteisuudesta ja järjestelmien vaatimuksien eroavaisuuksista. Lisäksi eräänä tämän tutkielman tarkoituksena on osoittaa, kuinka lähellä käsiteltävät tekniikat ovat ortogonaalista persistenssiä. Tällä on tarkoitus tukea käsitystä, kuinka oliokeskeistä ja helppoa pysyvyyden toteuttaminen valituilla tekniikoilla on.

2.5.1 Java ja pysyvyys

Jatkossa tarkastellaan, miten erilaiset tekniikat soveltuvat sovelluskehittäjän näkökulmasta Java-olioiden tallentamiseen. J2SE tarjoaa tähän useita vaihtoehtoja, joista tähän tutkielmaan on valittu neljä erilaista teknologiaa. Valitut tekniikat ovat tarkoitettu hieman erilaisiin käyttötarkoituksiin, vaikka yhtäältä ne kaikki ovat luotuja olioiden pysyvyyden toteuttamiseen. Läpikäymällä neljä erilaista tapaa pyritään antamaan kokonaisvaltaisempi kuva sovelluskehittäjän mahdollisuuksista.

Käsiteltävät tekniikat ovat kohdan 1.3 mukaisesti JOS, JAXB, JDBC ja JDO. Tarkastelun tarkoituksena ei ole tarjota täydellistä käyttöohjetta tai käsikirjaa käsiteltäviin tekniikoihin, vaan verrata Java-olion pysyvyyden toteuttamista. Alan kirjallisuuden perusteella edellä mai-

nituista tekniikoista JDBC vaikuttaa parhaimmin tunnetulta. JOS on edellistä yksinkertaisempi ja vanhempi väline, kun taas JDO vaikuttaa laajasti hyväksytyltä tekniikalta olioiden tuntumattoman pysyvyyden tavoittelussa (Object Data Management Group, 2004; Sun Microsystems, 2004). Java kehitettiin alun perin verkkosovelluksia varten ja XML-formaatti on yleinen tapa siirtää tietoa verkossa. Tästä syystä esitellään yksi tekniikka, JAXB, olioiden tallentamiseksi XML-dokumentteihin.



Kuva 2.2: Tutkielman viitekehys.

Kuvassa 2.2 esitetään tutkielman viitekehys, joka kuvaa tutkielman jäsentymistä kokemusperäiseen sekä teoreettiseen osaan. Viitekehys osoittaa myös tässä tutkielmassa tarkasteltavat teknologiat olioiden pysyvyyden toteuttamiseen.

3 Olioiden sarjallistaminen

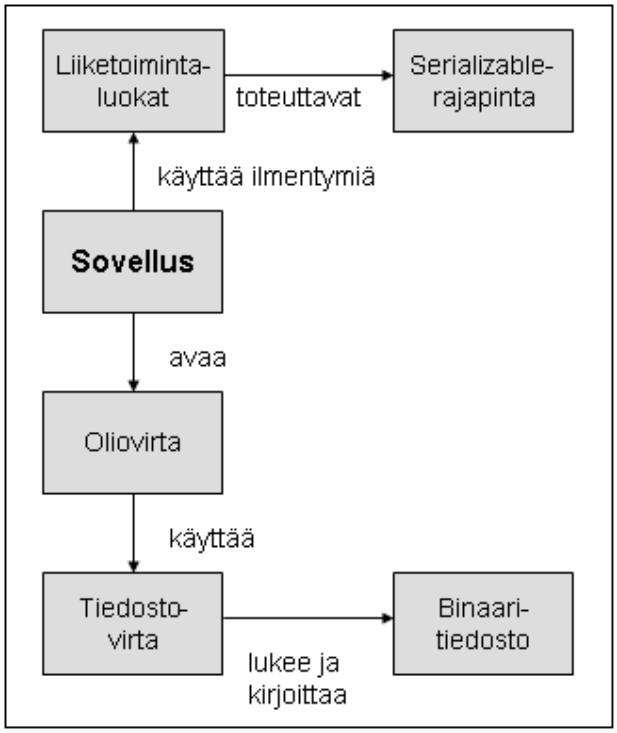
Edellä käsiteltiin olioiden pysyvyyttä yleisellä tasolla ja seuraavaksi asiaan tutustutaan käytännön näkökulmasta. Tässä luvussa esitellään Java Object Serialization (JOS) -teknologia, joka mahdollistaa Java-olioiden pysyvyyden. Ensin teknologiaan tutustutaan yleisesti kohdassa 3.1 ja tämän jälkeen kohdissa 3.2 ja 3.3 tarkastellaan esimerkkejä, kuinka oliot voidaan viedä tietolähteeseen sekä kuinka ne voidaan lukea sieltä. Luvun lopuksi kohdassa 3.4 pohditaan teknologian soveltuvuusaluetta.

3.1 Yleiskuvaus

Java-ohjelmointikielessä primitiivityyppinen data voidaan tallentaa matalan tason tietovirroilla tiedostojärjestelmään. Tällöin ohjelmoijalla on täysi kontrolli siitä, miten pysyvyys toteutuu. Tekniikka on kuitenkin työläs ja erityisen hankala, mikäli tallennetaan olioita, joilla on attribuutteinaan toisia olioita.

Ratkaisuksi tähän Sun Microsystems esitteli Java 1.1 versiossa yksinkertaisen ja laajennettavan sovellusliittymän olioiden *sarjallistamiseksi* (Opyrchal & Prakash, 1999). Olioiden sarjallistamisella tarkoitetaan olioarvojen kirjoittamista tietovirtaan, esimerkiksi tiedostoon (Sun Microsystems, 1997). Samalla tekniikalla olion tila voidaan myös palauttaa lukemalla se tietovirrasta.

Javan olioiden sarjallistaminen tukee niin yksittäisen olion kuin myös toisiinsa yhteyksissä olevien olioiden joukon, *olioverkon*, siirtämistä tietovirtaan (Sun Microsystems, 1997). Tietovirta voidaan kirjoittaa tiedostoon tai se voidaan yhtä lailla lähettää tietoverkossa toiselle tietokoneelle. Käytännössä tämä bittivirta sisältää kaiken sen tiedon, mitä tarvitaan vastaavan olioverkon uudelleen luomiseksi: olioiden luokkakuvaukset sekä kaikkien attribuuttien arvot, lukuun ottamatta nimenomaisesti ei-pysyviksi merkittyjä attribuutteja. Sarjallistettuja olioita voidaan lukea ja muokata millä tahansa toisella Java-sovelluksella. Kuvassa 3.1 on esitetty käsitteellisellä tasolla olioiden sarjallistamisen toimintaperiaate. Asiaan perehdytään yksityiskohtaisemmin tämän luvun kohdassa 3.2.



Kuva 3.1: Olioiden sarjallistamisen toimintaperiaate.

Ensi silmäyksellä sarjallistamismekanismi voi vaikuttaa tehokkaalta ja helpolta tekniikalta Java-olioiden tallentamiseksi. Sarjallistaminen on yksinkertaista, mutta tallennusmediatasolla se on vain binaaritiedostopohjainen ratkaisu. Sitä ei ole kehitetty tietokantojen haastajiksi tai tietovarastoinnin taidon näytteeksi. Kurotsuchin (1997) mukaan sarjallistamista voidaan enemminkin ajatella apumekanismina muille sofistikoituneille tekniikoille. Muun muassa RMI ja JavaBeans teknologiat käyttävät sarjallistamista olioarvojen siirtoon.

3.2 Pysyvyyden toteuttaminen

Seuraavaksi tarkastellaan, kuinka sovelluksen käyttämät oliot voidaan tallentaa binaaritiedostoon ja miten ne voidaan myöhemmin lukea sieltä sovelluksen käyttöön. Aluksi esitetään pysyvään luokkaan kohdistuvat vaatimukset.

Sarjallistamismekanismi on kiinteä osa Sun Microsystemsin Java-toteutusta ja helppo käyttää. Javan virtuaalikone huolehtii tietovirran käsittelyn matalan tason yksityiskohdista. Yksinkertaisimmillaan luokka saa automaattisen sarjallistamistuen, kun se toteuttaa standardi rajapin-

nan `java.io.Serializable`. Tämän rajapinnan käyttäminen ei vaadi luokkaa toteuttamaan ainoatakaan sen metodia. Toteuttaminen (`implements`) on vain merkinnällistä.

```
package jop;  // java ja olioiden pysyvyys

import java.io.*;

public class LevyPapu implements Serializable {

  // Luokkaan ei tarvita muuta sarjallistamiseen liittyvää koodia.
  // Lisää attribuutit ja metodit tähän.
}
```

Kuva 3.2: Sarjallistettava luokka `LevyPapu`.

Kuvassa 3.2 on esimerkki `Serializable`-rajapinnan toteuttavasta luokasta. Luokan `LevyPapu` ilmentymiä käyttävä luokka tai metodi voisi nyt sarjallistaa käyttämänsä ilmentymän tietovirtaan. Tarkempi luokan kuvaus on esitetty liitteenä 2.

Rajapinnan `Serializable` toteuttamisella luokka saa järjestelmän tarjoamat metodit olioiden kirjoittamiseen ja lukemiseen (Sun Microsystems, 1997). Samalla luokan kirjoittaja ilmaisee, että hän on huomioinut sarjallistamisen säännöt luokan toteutuksessa. Perussääntö on, että olion sarjallistamisessa kaikki sen primitiivityyppiset attribuutit sekä oliot myös sarjallistetaan. Mikäli attribuutti on merkattu avainsanalla `transient` tai `static`, sitä ei sarjallisteta. Lisäksi mikäli attribuuttina olevaa oliota ei ole merkattu edellä mainituilla avainsanoilla, tulee myös sen toteuttaa rajapinta `Serializable`. Ortogonaalisen persistenssin pysyvyyden tunnistamisen periaatteen mukaisesti sarjallistaminen koskee kaikkia niitä attribuutteja, mihin pysyvällä oliolla on kosketus.

Javan standardiluokista suurin osa toteuttaa `Serializable`-rajapinnan (Sun Microsystems, 2000a) ja näin ollen ne voidaan sarjallistaa sovelluksen oman luokan mukana. On kuitenkin olemassa eräitä poikkeuksia. Java ei mahdollista tiettyjen ajonaikaisten järjestelmäriippuvaisten olioiden sarjallistamista. Tällaisia ovat muun muassa säikeet (`Thread`) ja tietovirrat (esim. `OutputStream`, `InputStream`) (Sun Microsystems, 2000a). Ortogonaalisen persistenssin mukaan myös nämä tulisi voida tehdä pysyviksi, mutta käytännössä tämä on mahdotonta ja tarpeetonta. Myöskään Javan perusluokka `java.lang.Object` ei toteuta `Serializable`-rajapintaa. Tästä syystä ohjelmoijan omat luokat eivät suoraan saa sarjallistamismekanismia,

vaan rajapinta tulee toteuttaa eksplisiittisesti. Mikäli sarjallistettavalla luokalla on jokin ei-sarjallistettavan luokan esiintymä ominaisuutena, tulee se merkitä avainsanalla `transient`. Samoin kaikki arkaluonteinen ei tallennettavaksi tarkoitettu tieto tulisi merkitä ei-pysyväksi, eli väliaikaiseksi.

3.2.1 Olioiden tallentaminen

Olioiden sarjallistamiseen on omat oliotietovirtansa `ObjectInputStream` ja `ObjectOutputStream`, jotka käyttävät taustalla jotakin binaarivirtaa (Sun Microsystems, 2000a). Varsinainen olioiden tallennus ja luku suoritetaan useimmiten näiden oliotietovirtojen metodeilla `writeObject` ja `readObject`. Kokonaisuudessaan olioiden tiedostoon sarjallistamisessa tarvitaan seuraavia toimenpiteitä:

1. Valitse validi tiedoston nimi.
2. Käyttäen tiedoston nimeä avaa virta `FileOutputStream`.
3. Liitä virta `FileOutputStream` virtaan `ObjecOutputStream`.
4. Kirjoita haluamasi oliot virtaan `ObjecOutputStream`.
5. Sulje virta.

Tällöin poikkeuksettomassa suorituksessa olioverkko kirjoittautuu binaaritiedostoon. Tiedoston lukemisessa täytyy huomioida, että se on peräkkäistiedosto ja täten olioarvot saadaan samassa järjestyksessä kuin ne kirjoitettiinkin. Kuvassa 3.3 on esimerkki kokoelman tiedostoon sarjallistamisesta. Ajateltakoon, että kokoelman `olioKokoelma` luokka on `ListArray` ja kokoelma sisältää vain kuvan 3.2 mukaisia `LevyPapu`-olioita.

Kokoelman sarjallistaminen on mahdollista, koska Java-luokka `ListArray` toteuttaa rajapinnan `Serializable`. Kuvan 3.3 ohjelmakoodilistauksessa näkyy, että kokonaisen olioverkon kirjoittaminen voidaan tehdä yhdellä `writeObject`-operaation kutsulla. Yhden `olioKokoelma`-olion sarjallistamisessa tulevat mukaan kaikki siihen yhteydessä olevat oliot ja niiden attribuutit.

```
private boolean sulje() {

  FileOutputStream   fos = null;
  ObjectOutputStream out = null;

  try {
    fos = new FileOutputStream(this.tiedostonNimi);
    out = new ObjectOutputStream(fos);
    out.writeObject(this.olioKokoelma);    /* kirjoita olio virtaan */
    out.flush();
    out.close();

  /* käsittele poikkeukset */
  } catch (FileNotFoundException fileExc)  {
      /* tiedostoa tiedostonNimi ei voida avata eikä luoda */
      fileExc.printStackTrace();
      return false;
  } catch (NotSerializableException notSerExc) {
      /* käsiteltävä olio ei toteuta Serializable-rajapintaa */
      notSerExc.printStackTrace();
      return false;
  } catch(IOException IOExc) {
      /* virhe tietovirran käsittelyssä */
      IOExc.printStackTrace();
      return false;
  } catch (Exception exc) {
      exc.printStackTrace();
      return false;
  }

  return true;
}
```

Kuva 3.3: Olioiden pysyvyyden toteuttaminen sarjallistamismekanismilla (Rouvinen, 2004).

3.2.2 Olioiden palauttaminen

Edellä kuvattiin olioiden kirjoittaminen binaaritiedostoon. Seuraavaksi esitellään, miten oliot voidaan lukea takaisin sovelluksen käyttöön.

Kuvassa 3.4 on esimerkki, kuinka kuvan 3.3 ohjelmakoodilla kirjoitettu `ListArray`-olio ja sen mukana kaikki `LevyPapu`-oliot voidaan palauttaa tiedostosta ohjelman käyttöön. Palauttaminen tehdään periaatteessa yhdellä komennolla: `readObject`. Tämä lukee tietovirrasta yhden olion, joka tässä tapauksessa aiheuttaa koko virran sisällön käsittävän olioverkon lukemisen.

```
private boolean alusta() {

  FileInputStream fis  = null;
  ObjectInputStream in = null;

  try {
    fis = new FileInputStream(this.tiedostonNimi);
    in  = new ObjectInputStream(fis);
    this.olioKokoelma = (ArrayList) in.readObject();   /* lue oliot */
    in.close();

  /* käsittele poikkeukset */
  } catch (FileNotFoundException fileExc)  {
      /* kohdetiedostoa ei voida avata lukemista varten */
      return false;
  } catch (java.io.EOFException eofExc) {
      /* tiedosto avattiin, mutta se oli tyhjä, ei olioita */
      return false;
  } catch(IOException IOExc) {
      /* virhe tietovirran käsittelyssä */
      IOExc.printStackTrace();
    return false;
  } catch (ClassCastException castExc) {
      /* virrasta luettu olio ei ole tyyppiä ArrayList */
      castExc.printStackTrace();
      return false;
  } catch (Exception exc) {
      exc.printStackTrace();
      return false;
  }

  return true;
}
```

Kuva 3.4: Olioiden palauttaminen sovelluksen käyttöön (Rouvinen, 2004).

Kuvissa 3.3 ja 3.4 näkyvät myös tärkeimmät sarjallisuuteen liittyvät poikkeukset. `NotSerializableException` esiintyy, mikäli jokin sarjallistettavista olioista ei toteuta `Serializable`-rajapintaa ja sitä ei voida sarjallistaa. Mikäli tietovirrasta yritetään lukea jotain sellaista, mitä sieltä ei käsittelykohdan mukaan voida lukea tai sijoittaa vastaanottavaan muuttujaan, syntyy `ClassCastException`. Edelleen luettaessa virrasta olio, jolle ei löydy ajon aikana vastaavaa luokkaa eikä täten oliota voida rakentaa, syntyy poikkeus `ClassNotFoundException`. Esitetyt tiedostonkäsittelypoikkeukset eivät suoranaisesti liity sarjallistamiseen.

3.3 Pysyvyyden kontrollointi

Silloin tällöin voi tulla tarvetta muuttaa sarjallistamismekanismin perustoiminnallisuutta. Tässä kohdassa kuvataan olioiden pysyvyyden kontrolloinnin mahdollisuudet.

Luokka voi myös itse toteuttaa `writeObject` ja `readObject` operaatiot (Sun Microsystems, 1997). Tämä on tarpeellista silloin, kun tallennusmuoto poikkeaa standardista. Näin on esimerkiksi, jos tallennettava olio haluaa muuttaa tietojaan tai suorittaa jonkin operaation ennen talletusta. Jos luokalle halutaan antaa täysi kontrolli tallennusmuotoon, luokan täytyy toteuttaa `java.io` paketin rajapinta `Externalizable` rajapinnan `Serializable` sijasta (Sun Microsystems, 1997). Tätä voidaan käyttää, mikäli halutaan suorittaa esimerkiksi tiedon kryptausta tai pakkausta. `Externalizable`-rajapintaan ei sisälly automaattista tallennus- tai lukuoperaatioita, joten ohjelmoijan on toteutettava ne täydellisesti itse. Toteuttavia metodeja on kaksi: `writeExternal(ObjectOutput)` ja `readExternal(ObjectInput)`. Lisäksi ohjelmoijan tulee itse huolehtia kaiken tiedon, kuten myös yli- ja aliluokkien käsittelystä.

Varsinaisen kirjoitus- ja lukukontrolloinnin lisäksi sarjallistamisessa on muutama erikoispiirre, joista ohjelmoijan on syytä tietää. Ensimmäinen on se, että tietovirtaan kirjoitetut oliot ovat kopioita käsiteltävistä olioista (Sun Microsystems, 2000a). Tästä seuraa, että mikäli olio kirjoitetaan virtaan, jonka jälkeen oliota muutetaan, eivät muutokset automaattisesti reflektoidu virrassa olevaan olioon. Ohjelmoijan tulee aina tarvittaessa kirjoittaa olio uudelleen virtaan.

Toinen erikoispiirre sarjallistamisessa on luokkien versionhallinta. Sarjallistamismekanismi laskee jokaiselle sarjallistettavalle oliolle versionumeron (`serialVersionUID`) sen luokkarakenteen perusteella (Sun Microsystems, 1997). Luokkaa muutettaessa muuttuu sen ilmentymien versionumero. Oletetaan esimerkiksi, että sovellus sarjallistaa luokan `A` ilmentymän `a`, jonka jälkeen luokkaan `A` lisätään uusi attribuutti. Nyt sovellus ei voi enää lukea aiemmin sarjallistettua oliota `a` luokan `A` ilmentymään, koska `a`:n rakenne ei vastaa uudistetun luokan rakennetta. Rakenteiden eroavaisuus huomataan vertaamalla käsiteltävien olioiden versionumeroita. Ohjelmoija voi kiertää versionumerotarkkailun - ja automaattiset tarkistukset - lisäämällä luokkaan oman `serialVersionUID`-attribuutin ja antamalla sille halutun versio-

numeron (Kurotsuchi, 1997). Tällöin edellisessä esimerkkitapauksessa järjestelmä olisi lukenut olion `a` ja antanut uudelle attribuutille oletusarvon.

3.4 Soveltuvuusalue

Javan sarjallistamisominaisuus ei toteuta ortogonaalista persistenssiä, vaikkakin on hyvin lähellä sitä. Tekniikka rikkoo riippumattoman pysyvyyden periaatetta, koska luokan tulee toteuttaa `Serializable`-rajapinta saadakseen tuen sarjallisuuteen. Käytännössä tämä merkitsee, että sovelluksen käyttämistä luokista, jotka eivät toteuta tätä rajapintaa ja joiden lähdekoodia ei ole saatavilla, ei saada sarjallistamismekanismilla pysyviä. Lisäksi luokkia muutettaessa törmätään sarjallistettujen olioiden versiointiongelmiin. Jälkimmäinen ongelma voidaan kylläkin kiertää.

Sarjallistamismekanismilla luodut pysyvät oliot eivät ole siirrettäviä. Tietovirtaa voi lukea vain Java-ohjelmalla ja lukijan tulee tietää peräkkäisyyteen perustuvan virran sisältö (Kurotsuchi, 1997). Toinen vakavampi puute on se, että sarjallistaminen ei tue tallennetun tiedon osittaista käsittelyä. Siispä yksittäisen talletetun olion tietoja haettaessa tai muutettaessa koko olioverkko on ladattava virtuaalikoneen muistiin ja etsittävä haluttu olio manuualisesti. Sarjallistamismekanismin toteutusta on myös väitetty tehottomaksi muistinkäytön ja levylle kirjoittamisnopeuden suhteen (Opyrchal & Prakash, 1999). Eräissä kokeissa ohjelmoijan toteuttama suoraan tietovirtaan kirjoittaminen on ollut yli puolet nopeampaa kuin sarjallistamisen oliovirtojen käyttö (Sun Microsystems, 2000a).

Tämä tekniikka sopii periaatteessa erittäin moniin pysyvän tiedon toteutuksiin. Se on helppo käyttää ja eikä vaadi erillisasennuksia. Käytännössä suuret tietomäärät ja niihin kohdistuvat haut eivät sovellu sarjallistamismenetelmälle johtuen peräkkäistietuemaisuudesta. Sarjallistaminen on omiaan yksinkertaisissa sovelluksissa, joissa ei ole tehokkuusvaatimuksia tai todella suuria tietomääriä, esimerkiksi tapauksissa, joissa pysyvä olio ladataan sovelluksen käynnistyessä ja kirjoitetaan sovelluksen sulkeutuessa.

4 XML-sidonnan arkkitehtuuri

Edellä esitelty Javan sarjallistamismekanismi on yksi vaihtoehto luotaessa pysyviä olioita. Tässä luvussa esitetään toinen vaihtoehto, Java Architecture for XML Binding -teknologia (JAXB). Teknologiaan tutustutaan ensin yleisesti kohdissa 4.1 ja 4.2 ja tämän jälkeen kohdissa 4.3 ja 4.4 tarkastellaan esimerkkejä, kuinka oliot voidaan viedä tietolähteinä käytettäviin XML-dokumentteihin sekä kuinka ne voidaan lukea sieltä. Luvun lopussa pohditaan teknologian soveltuvuusaluetta.

4.1 Yleiskuvaus

Extensible Markup Language (XML) on ympäristöriippumaton metakieli rakenteisten tietojen kuvaamiseen. XML-kielellä kirjoitettuja dokumentteja käytetään siirrettäessä tietoa heterogeenisten sovellusten ja organisaatioiden välillä. Java-ohjelmointikieli ja XML sopivat hyvin yhteen verkkosovellusten rakentamisessa: Java tarjoaa kielen ohjelmointiin ja XML tiedon esittämiseen. Seuraavaksi on muutama sana XML-dokumenteista ja tämän jälkeen keskitymme itse JAXB-teknologiaan.

4.1.1 XML-dokumenteista

XML-dokumentti voidaan kuvata puuksi *elementtejä*. Elementti voi sisältää muita elementtejä, tekstiä tai näiden yhdistelmiä sekä lisäksi elementillä voi olla attribuutteja, jotka koostuvat avaimesta ja arvosta. Elementti voi viitata toiseen elementtiin *tunnisteattribuuteilla* ja näin voidaan kuvata mielivaltaisia verkkorakenteita. XML-dokumentin täytyy noudattaa XML-spesifikaation mukaisia muodostamissääntöjä. Sanotaan, että XML-dokumentin tulee olla *oikeamuotoinen* eli rakenteellisesti hyvin muodostunut. Jotta XML-dokumentteja voitaisiin käyttää mielekkäästi, tulee oikeamuotoisuuden lisäksi dokumenttien rakenteen ja sisällön olla määritelty niin, että kaikki osapuolet ymmärtävät ne täsmälleen samalla tavalla. Tämä voidaan saavuttaa käyttämällä skeemaa eli tiedon kaavaa. Skeema on joukko sääntöjä, jotka määräävät XML-dokumentin rakenteen ja sen komponenttien sisällön: elementit, attribuutit ja tekstin. Toisin sanoen skeema on XML-dokumentin syntaksin ja semantiikan määrittelevä spesifikaatio. XML-dokumentin sanotaan olevan *validi* jos ja vain jos se täyttää skeeman asettamat rajoitteet (Sun Microsystems, 2003d).

XML-spesifikaatio kuvaa itse yhden alikielen skeemojen määrittelemiseen: document-type definitions (DTD) (Sun Microsystems, 2003d). On kuitenkin huomattu, että DTD on varsin riittämätön monimutkaisen tiedon esittämiseen. Tästä syystä on kehitetty rikkaampia kieliä, kuten esimerkiksi World Wide Web Consortiumin XML Schema Language. Tällä kielellä voidaan DTD-kieltä helpommin muun muassa kuvata tietotyyppejä sekä rakenteiden välisiä yhteyksiä (Ort & Mehta, 2003).

4.1.2 JAXB-teknologiasta

Java Community Process julkaisi vuonna 2003 ensimmäisen virallisen version spesifikaatiosta The Java Architecture for XML Binding. Tätä ennen jo parin vuoden ajan oli julkaistu epävirallisia versioita ja lähiaikoina on ilmestymässä spesifikaation toinen versio. Teknologia siis elää voimakkaasti sekä kehittyy jatkuvasti. Sun Microsystems on integroinut JAXB-teknologian sen omaan verkkopalveluiden kehityspakettiin (Java Web Services Development Pack), mutta sitä voidaan käyttää myös ilman paketin muita teknologioita.

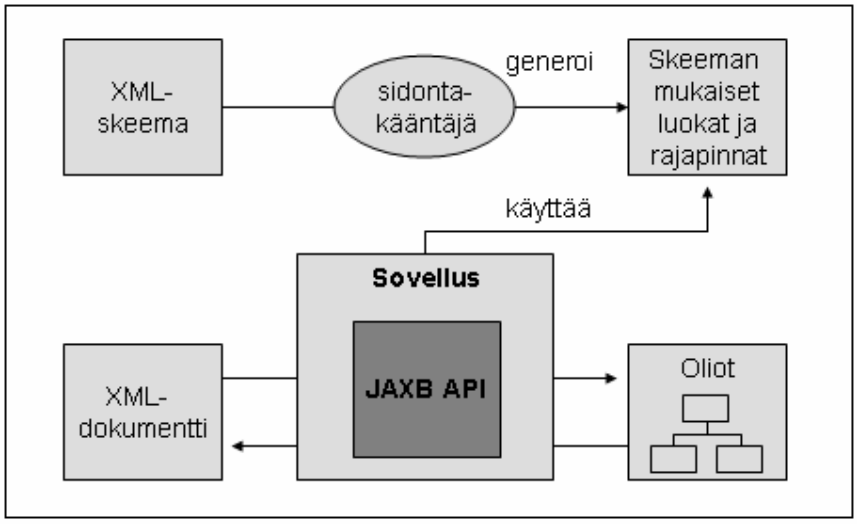
Sun Microsystemsin (2003e) mukaan JAXB-teknologian avulla sovelluskehittäjät voivat helposti sisällyttää Java-sovelluksiin XML-dokumenttien käsittelyn. Se tarjoaa kätevän tavan sitoa XML-skeemat Javan käyttämään tiedon esitysmuotoon, oliomalliin. Teknologia mahdollistaa XML-dokumenttien lukemisen sovelluksen käyttämään oliorakenteeseen sekä tämän rakenteen sisällön kirjoittamisen takaisin XML-tiedostoon. Tämä tarkoittaa, että sovelluskehittäjät saavat XML-teknologian tuomat edut tietämättä paljoakaan XML-ohjelmointitekniikoista. Tuloksena saadaan hyvin siirrettäviä, kevyitä ja helposti ylläpidettäviä sovelluksia sekä verkkopalveluita. Sanotaan myös, että tämä teknologia voi syrjäyttää tähän asti käytetyt Simple API for XML Processing (SAX) ja Document Object Model (DOM) käsittelymallit (Sun Microsystems, 2003e).

4.2 JAXB-arkkitehtuuri

XML-sidonnan arkkitehtuuri voidaan jakaa kahteen osaan. Sidontakääntäjää käytetään ennen sovelluksen käyttöä ja JAXB-sovellusliittymä hoitaa ajonaikaiset toiminnot XML-dokumenttien käsittelyssä. Seuraavassa on tarkempi kuvaus aiheesta.

JAXB-teknologian ydin on *sidontakääntäjä* (binding compiler), joka lukee pysyviä luokkia kuvaavan XML-skeeman ja generoi sitä vastaavan Java-oliomallin (Sun Microsystems, 2003e). Generointi suoritetaan kerran, ennen sovelluksen suorittamista. Sovelluksen ajon aikana XML-dokumentin sisältö ladataan tämän hierarkkisen mallin mukaiseen *sisältöpuuhun* (content tree). Toinen puolisko teknologiasta käsittää JAXB-sovellusliittymän, minkä kautta sovellukset voivat käsitellä sisältöpuussa olevien generoitujen luokkien ilmentymiä (Sun Microsystems, 2003e). Tämä sovellusliittymä kätkee sovellukselta toteutukseen liittyvät yksityiskohdat ja tarjoaa standardoidun tavan suorittaa perusoperaatioita, kuten XML-dokumentin lukemisen ja kirjoittamisen.

Kuvassa 4.1 esitetään JAXB-teknologian pääpiirteittäinen käyttö. Kuva koostuu seuraavista kokonaisuuksista: XML-skeema, sidontakääntäjä, skeeman mukaiset luokat ja rajapinnat, XML-dokumentti, sisältöpuu ja JAXB-sovellusliittymää käyttävä sovellus. XML-skeema on kuvaus elementeistä, attribuuteista ja muista kokonaisuuksista XML-dokumentissa. Toisin sanoen skeeman tulisi kuvata pysyvät liiketoimintaluokat, niiden väliset yhteydet sekä mahdollinen hierarkia. Käytettävä kieli skeeman kuvaukseen on W3C XML Schema Language (Sun Microsystems, 2003d). Sidontakääntäjä generoi XML-skeeman perusteella joukon rajapintoja sekä niiden mukaisia Java-luokkia. Kukin luokka tarjoaa pääsyn vastaavaan skeeman komponentin sisältöön JavaBeans-tyylisesti haku- ja asetusmetodeilla (`get` ja `set`) (Sun Microsystems, 2003d). JAXB API:n avulla käsiteltävät XML-dokumentit ladataan näiden rajapintojen ja luokkien mukaiseen sisältöpuuhun.



Kuva 4.1: JAXB-teknologian käyttö (Ort & Mehta, 2003).

Sun Microsystemsiä (2003e) mukaillen *JAXB-sidontaympäristön toteutus* (JAXB binding framework implementation) on ajonaikainen sovellusliittymän toteutus, joka vastaa XML-dokumenttien sisällön *takaisinjärjestämisestä* (unmarshalling) ja *järjestämisestä* (marshalling). Takaisinjärjestämiseksi kutsutaan XML-dokumentin lataamista sovelluksen käyttämään sisältöpuuhun. Olioiden kirjoittamista sisältöpuusta XML-tiedostoon kutsutaan vastaavasti järjestämiseksi. Kuvassa 4.1 näitä operaatiota on kuvattu sovelluksen läpi kulkevilla nuolilla. Toteutus on myös vastuussa XML-dokumenttien validiteetin tarkastamisesta. Java-sovellukset käyttävät JAXB-sidontaympäristön toteutusta pakkauksen `javax.xml.bind` kautta. Sun Microsystems on toteuttanut tämän pakkauksen luokat ja tätä toteutusta kutsutaan nimellä JAXB Standard Implementation. Yhtäältä spesifikaatio on avoin kaikille, joten kuka tahansa voi luoda oman toteutuksensa.

4.3 Pysyvyyden toteuttaminen

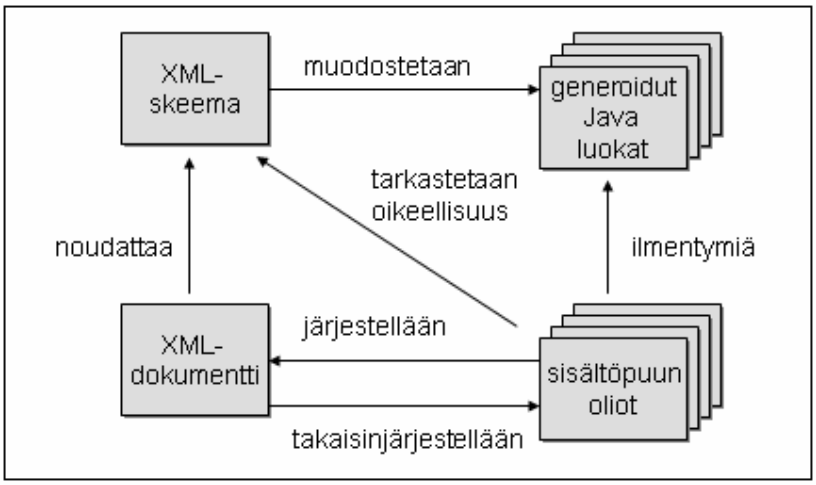
Olioiden pysyvyyden toteuttaminen JAXB-teknologialla vaatii osaamista XML-skeeman kirjoittamisesta ja ymmärrystä koko teknologiasta. Itse sovelluksessa XML-dokumenttien käsittely on tämän jälkeen helppoa. Tässä kohdassa kuvataan vaiheet olioiden tallentamiseksi ja palauttamiseksi.

JAXB-teknologiaa käyttävän sovelluksen rakentaminen aloitetaan suunnittelemalla pysyvät liiketoimintaluokat. Näiden luokkakuvausten perustella kirjoitetaan XML-skeema, joka annetaan syötteeksi sidontakääntäjälle. Kuvassa 4.2 on esimerkki `LevyPapu`-liiketoimintaluokkaa vastaavasta skeemasta. Kuvan skeeman rakenne muodostuu pääelementistä `LevyKokoelma`, joka sisältää `LevyPapu`-tyyppisiä liiketoimintaluokkaa kuvaavia `Levy`-elementtejä. `Levy`-elementtejä voi esiintyä skeeman mukaisessa XML-dokumentissa peräkkäin kuinka monta tahansa. `Levy`-elementillä on pakollinen `nimi`-attribuutti, muut attribuutit ovat valinnaisia. On syytä huomioida, että luokkien generointi sidontakääntäjällä synnyttää rajapinnan `LevyPapu` ja sen toteuttavan luokan `LevyPapuImpl`, mutta jälkimmäisen toteutus ei ole sama kuin luvussa 3 esitellyn `LevyPapu`-liiketoimintaluokan. Luokan toteutuksessa on JAXB-teknologiaan liittyviä lisäyksiä eikä `LevyPapuImpl`-ilmentymiä voida suoraan käyttää `LevyPapu`-luokan ilmentymien sijaan.

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:jaxb="http://java.sun.com/xml/ns/jaxb" jaxb:version="1.0">

<xs:element name="LevyKokoelma">
   <xs:complexType>
      <xs:sequence>
         <xs:element name="Levy" type="LevyPapu" minOccurs="0"
            maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
</xs:element>

<xs:complexType name="LevyPapu">
  <xs:sequence>
    <xs:element name="nimi" type="xs:string"/>
    <xs:element name="sisalto" type="xs:string" minOccurs="0"
        maxOccurs="1"/>
    <xs:element name="levynKoko" type="xs:double" minOccurs="0"
        maxOccurs="1"/>
    <xs:element name="onUudelleenKirjoitettavissa" type="xs:boolean"
        minOccurs="0" maxOccurs="1"/>
  </xs:sequence>
</xs:complexType>

</xs:schema>
```

Kuva 4.2: XML-skeema `levyt.xsd`.

Kuvassa 4.3 selvitetään sidontaan liittyvien käsitteiden välistä yhteyttä. JAXB-sovellusliittymä on vastuussa kuvassa näkyvistä olioiden käsittelystä ja sidontakääntäjä luonnollisesti generoitujen luokkien muodostamisesta annetun XML-skeeman perusteella. Kuvasta voimme lisäksi huomata, että XML-skeeman tulee olla saatavilla myös sovelluksen ajon aikana, ei vain generoitaessa rajapintoja ja luokkia.



Kuva 4.3: JAXB-teknologian käsitteiden välinen suhde toisiinsa (Sun Microsystems, 2003e).

Luokkien ja rajapintojen generoinnin jälkeen voidaan rakentaa niitä käyttävä sovellus. Sovelluksen portti JAXB API:in ja näin ollen XML-dokumenttien käsittelyyn on sovellusliittymän abstrakti luokka `JAXBContent`. Tämän luokan kautta saadaan tarvittavat oliot XML-dokumenttien järjestämiseksi, takaisinjärjestämiseksi ja sisällön validiteetin tarkastamiseksi (Ort & Mehta, 2003). `JAXBContent`-ilmentymä luodaan tämän luokan staattisella metodilla `newInstance(String contextPath)`, missä parametrina annettava merkkijono on käytettävä *kontekstipolku* (context path). Kontekstipolku on lista yhdestä tai useammasta pakkauksen nimestä, jotka sisältävät XML-skeemasta generoidut rajapinnat. Metodi lataa muistiin rajapintojen määrittelemän rakenteen. Rakenne muodostaa sisältöpuun rungon.

4.3.1 Olioiden tallentaminen

Sovellus voi lisätä sisältöpuuhun vain sidontakääntäjän generoimien luokkien ilmentymiä, mutta muuten sisältöpuuta ja sen olioita voidaan käyttää, kuten mitä tahansa muita sovelluksen olioita. Sisältöpuu voi olla muodostettu automaattisesti takaisinjärjestämisen yhteydessä tai luotu manuaalisesti. Käytettävät operaatiot olioiden pysyvyyden toteuttamiseksi ovat samat riippumatta siitä, kuinka sisältöpuu on ladattu. Seuraavaksi tarkastelemme sisältöpuun manuaalista muodostamista ja tämän jälkeen tutustumme olioiden tallentamiseen.

Sovelluksen ohjelmoija voi muodostaa sisältöpuun sidontakääntäjän generoiman luokan `ObjectFactory` avulla. Tämän luokan toteutus riippuu niin XML-skeemasta kuin JAXB-sidontaympäristön toteutuksestakin. Ohjelmoijalle sisältöpuun muodostaminen on kuitenkin varsin yksinkertaista (Ort & Mehta, 2003):

1. Luo uusi ilmentymä `ObjectFactory`-luokasta.
2. Luo XML-dokumentin elementit, siis sisältöpuun oliot, kutsumalla `ObjectFactory`-olion tarjoamia `create`-metodeita (esimerkiksi `createLevyKokoelma`).
3. Lisää sisältöpuun olioille haluamasi arvot.

Elementtejä sisältöpuuhun voidaan luoda haluttu määrä. Sisältöpuun edustamaan elementtijoukkoon päästään käsiksi generoiduilla hakumetodeilla (`get`) ja tätä joukkoa voidaan muokata vastaavilla asetusmetodeilla (`set`). Kuvassa 4.4 esitetään ohjelmakooditasolla edellä ku-

vattu uuden sisältöpuun muodostaminen ja olioiden lisääminen siihen. Kuvan operaatiossa sisältöpuu on kertakäyttöinen eikä sovellus käytä luotua rakennetta myöhemmin.

Kuvassa 4.4 osoitetaan myös sisältöpuun järjestäminen ja oikeellisuuden tarkastaminen, toisin sanoen olioiden pysyvyyden toteuttaminen. XML-dokumentin järjestäminen suoritetaan rajapinnan `Marshaller` avulla. Sen toteuttava ilmentymä pyydetään `JAXBContent`-oliolta metodilla `createMarshaller()`. XML-dokumentti voidaan järjestellä muun muassa paikalliseen tiedostoon, tietovirtaan, merkkijonoon tai DOM-solmuun (Sun Microsystems, 2003d). Lisäksi järjestämisen visuaaliseen lopputulokseen, siis taittoon, on mahdollista vaikuttaa. Tästä on esimerkki kuvassa 4.4, missä `jarjestaja`-oliota pyydetään metodilla `setProperty` lisäämään rivinvaihdot ja sisennykset muodostettavaan XML-tiedostoon. Itse järjestäminen suoritetaan yhdellä operaatiokutsulla: `marshal`. Kuvassa 4.4 järjestäminen tuottaa sisältöpuun mukaisen XML-tiedoston. Ennen järjestämistä voidaan suorittaa sisältöpuun oikeellisuuden tarkistaminen `Validator`-rajapinnan metodilla `validate`. Tämä rajapinta tarjoaa myös mahdollisuuden tarkastella mahdollisia sisältöpuun virheitä tarkemminkin (Sun Microsystems, 2003d).

Mikäli kuvan 4.4 luokalla olisi attribuuttinaan myös JAXB-sisältöpuuolio ja tätä ilmentymää ylläpidettäisiin läpi luokan ilmentymän elinkaaren, ei erillistä uuden sisältöpuun rakentamista ja listan kopiointia tarvitsisi suorittaa kuvan esittämässä `jarjesta`-operaatiossa. Kuvalla 4.4 halutaan esimerkin kautta osoittaa, kuinka nämä tehtävät voidaan tehdä.

```
public boolean jarjesta() {

  JAXBContext jaxbSisalto= null; /* portti JAXB API:in */
  Marshaller  jarjestaja = null; /* vastaa sisältöpuun järjestämisestä */

  /* skeemasta generoitu tyyppi, XML-dokumentin juurielementti */
  jop.jaxb.LevyKokoelma levyKokoelma = null;

  try {
    /* aseta sisältöpolku (context path) jop.jaxb–pakkaukseen  */
    jaxbSisalto = JAXBContext.newInstance("jop.jaxb");

    /* luo järjestäjä-olio */
    jarjestaja  = jaxbContext.createMarshaller();
    jarjestaja.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                           new Boolean(true));
    Validator validifioija = jaxbContext.createValidator();

    /* (1) luo uusi sisältöpuu, (2) luo juurielementti, (3) hae viittaus
    sisältöön */
    ObjectFactory objectFactory = new ObjectFactory();
    levyKokoelma   = (jop.jaxb.LevyKokoelma)
                      objectFactory.createLevyKokoelma();
    List levyLista = levyKokoelma.getLevy(); /* hae LevyPapu-kokoelma */

    /* Kopioi olion käyttämän listan sisältö XML-elementtien listaan.
     * Listan elementit ovat jop.jaxb.LevyPapu-luokan ilmentymiä.
     */
    levyLista.add((Collection)this.olioKokoelma);

    if (false == validator.validate(levyKokoelma)) {
      throw new JAXBException("Sisältöpuu ei ole validi");
    }

    /* järjestä oliot XML-tiedostoon,levyKokoelma on juurielementti */
    jarjestaja.marshal(levyKokoelma,
      new FileOutputStream(this.tiedostonNimi));

  } catch (JAXBException jaxbExc) {
      jaxbExc.printStackTrace();
      return false;
  } catch (Exception Exc) {
      Exc.printStackTrace();
      return false;
  }
  return true;
}
```

Kuva 4.4: Sisältöpuun luominen ja sen järjestäminen XML-tiedostoon (Rouvinen, 2004).

4.3.2 Olioiden palauttaminen

JAXB-tekniikassa takaisinjärjestäminen tarkoittaa olemassaolevan XML-dokumentin sisältöä vastaavan oliorakenteen muodostamista. Toisin sanoen tällä tarkoitetaan XML-dokumenttiin tallennettujen olioiden palauttamista sovelluksen käyttöön. XML-dokumentin takaisinjärjes-

täminen voidaan tehdä useista lähteistä. Tässä kohdassa käsiteltävä tiedostosta lukeminen on yksi vaihtoehto.

Kuvassa 4.5 on kuvattu operaatio XML-tiedoston lukemiseksi sisältöpuuhun. Tämä takaisinjärjestäminen tehdään seuraavasti. Järjestämisen tavoin on JAXB API:n luokka `JAXBContent` portti takaisinjärjestämiseen. `JAXBContent`-luokan ilmentymään ladataan sisältöpuun runko `newInstance`-metodilla ja tältä ilmentymältä pyydetään varsinaisen työn suorittavaa `UnMarshaller`-rajapinnan mukaista oliota. Tämän olion `unmarshal`-operaatiolla pyydetään JAXB API:a lukemaan XML-dokumentti operaatiolle parametrina annetusta lähteestä. Kuvan 4.5 `takaisinjarjesta`-operaatiossa lähteenä käytetään tietovirtaa. Muodostettuun oliorakenteeseen päästään käsiksi hakumetodeilla, esimerkiksi kuvassa 4.5 lause `levyKokoelma.getLevy()` palauttaa `java.util.List`-listan `LevyKokoelma`-elementin sisälle kuuluvista `Levy`-elementeistä. Palautettu lista ei ole kopio, vaan kaikki listan käsittelyt kohdistuvat suoraan sisältöpuuhun.

XML-dokumentin oikeellisuus voidaan haluttaessa tarkastaa takaisinjärjestämisen yhteydessä tai tämän suorituksen jälkeen (Sun Microsystems, 2003d). Kuvassa 4.5 tarkastus asetetaan päälle operaatiokutsulla `setValidating(true)`, jolloin tarkastus suoritetaan takaisinjärjestämisen ohessa. Spesifikaation mukaan JAXB-sidontaympäristön toteutuksen tulee raportoida takaisinjärjestämisessä havaituista virheistä, mutta se jättää toteutuksen päätettäväksi, keskeytetäänkö olioiden koostaminen vai jatketaanko XML-dokumentin lukemista seuraavasta kelvollisesta kohdasta (Sun Microsystems, 2003d).

Takaisinjärjestämisessä palautettu oliorakennetta voidaan käyttää sellaisenaan osana sovellusta, tai sen sopivat osat voidaan kopioida luokan ilmentymän staattisiin rakenteisiin. Jälkimmäistä keinoa käytetään kuvassa 4.5: `jop.jaxb.LevyPapu`-oliot lisätään olion omaan listaan. Tällöin `JAXBContent`-ilmentymä on vain väliaikainen olio.

```
private boolean takaisinjarjesta() {

  JAXBContext  jaxbSisalto   = null;  /* portti JAXB API:in */
  Unmarshaller takJarjestaja = null;  /* Vastaa sisältöpuun
                                         takaisinjärjestämisestä */
  /* skeemasta generoitu tyyppi, XML-dokumentin juurielementti */
  jop.jaxb.LevyKokoelma levyKokoelma = null;

  try {
    jaxbContext = JAXBContext.newInstance("jop.jaxb");

    /* Takaisinjärjestä XML-dokumentti ja tarkasta sen oikeellisuus.
     * Takaisinjärjestäminen palauttaa pääelementtiä vastaavan olion.
     */
    takJarjestaja = jaxbContext.createUnmarshaller();
    takJarjestaja.setValidating(true);    /* aseta tarkistus päälle */
    discCollection = (jop.jaxb.LevyKokoelma)
      takJarjestaja.unmarshal(new FileInputStream(this.tiedostonNimi));

    this.olioKokoelma = new ArrayList();          /* katso kuva 4.4 */

    /* lisää kaikki XML-dokum. Levy-elementit olion omaan listaan   */
    this.olioKokoelma.add((Collection)levyKokoelma.getLevy());

  } catch (JAXBException jaxbExc) {
      jaxbExc.printStackTrace();
      return false;
  } catch (Exception e) {
      e.printStackTrace();
      return false;
  }

  return true;
}
```

Kuva 4.5: XML-dokumentin lukeminen sisältöpuuhun (Rouvinen, 2004).

4.4 Pysyvyyden kontrollointi

Ohjelmoijat voivat halutessaan muuttaa JAXB-spesifikaation mukaista toiminnallisuutta. Tässä kohdassa kuvataan olioiden pysyvyyden kontrolloinnin mahdollisuudet.

JAXB-spesifikaatio määrittelee perussäännöt XML-skeeman ja Java-oliomallin väliselle sidonnalle. Tällä tarkoitetaan, että spesifikaatiossa määrätään XML-skeeman komponenttien ja Javan esitysmuodon välinen yhteys. Esimerkiksi XML-tietotyyppi `xsd:string` on sidottu Javan luokkaan `java.lang.String`. Kaikkien JAXB-sidontakääntäjien tulee noudattaa näitä sääntöjä. Useimmissa tapauksissa sidonnan perussäännöt ovat riittävät, mutta JAXB-teknologia mahdollistaa myös sidontasääntöjen muuttamisen. Sidontaan voidaan vaikuttaa antamalla sidontakääntäjälle syötteeksi sidontasääntöjen korvaustiedosto XML-skeeman ge-

neroinnin yhteydessä tai kirjoittamalla haluttuja sidonnan erityispiirteitä suoraan XML-skeemaan (Ort & Mehta, 2003).

Syitä sidonnan modifiointiin on useita. Mikäli ohjelmoija kokee, ettei spesifikaatiossa määritelty sidonta XML-tietotyypin ja Java-tietotyypin välillä ole sopiva, hän on vapaa muuttamaan sitä (Sun Microsystems, 2003e). Ohjelmoija voi esimerkiksi määrätä käytettävän primitiivityyppiä `int` luokan `Integer` sijasta. Toinen syy on, että vakiosidontasäännöt muodostavat listan (`java.util.List`) hierarkkisesti samalla tasolla olevista samantyyppisistä elementeistä. Sidontasääntöjen muokkaamisella ohjelmoija voi määritellä käytettävän jotain muuta tarkoitukseen paremmin sopivaa tietorakennetta (Sun Microsystems, 2003e). Lisäksi ohjelmoija voi määritellä haluamiaan yhteyksiä XML-dokumentin attribuuttien arvojen ja Javan määriteltyjen vakioiden välille.

Omia sidonnan määrityksiä voidaan lisäksi käyttää, mikäli liiketoimintaluokassa on joitain sellaisia ajonaikaisia ominaisuuksia, joita ei voida tallentaa XML-tiedostoon, kuten esimerkiksi tiedostovirtoja. Tällöin XML-skeemassa kuvataan liiketoimintaluokka tallennettavien tietojen osalta ja sidonnan laajennuksissa kuvataan generoitavaan luokkaan lisättävät muut ominaisuudet. Laajennukset annetaan syötteenä sidontakääntäjälle skeeman ohessa. Näin ollen oliomallin järjestämisessä XML-dokumenttiin tiedostovirrat jätetään huomioimatta ja XML-dokumentin lukemisessa JAXB-toteutus asettaa oliomallin tiedostovirta-ominaisuudet null-arvoihin.

4.5 Soveltuvuusalue

Javan JAXB-tekniikka ei toteuta ortogonaalista persistenssiä. Sidontakääntäjällä generoidut luokkarungot ovat toteutusriippuvaisia ja tämä rikkoo riippumattomuuden periaatetta. Pysyvyyden tunnistamisen periaatteen rikkoo se, että ohjelmoijan on määriteltävä eksplisiittisesti kaikki pysyvät luokat skeemassa. Lisäksi tekniikalla on hankala tallentaa jo olemassaolevien luokkien ilmentymiä, koska niistä tulee aluksi kirjoittaa XML-skeema, josta sitten generoidaan sidontakääntäjällä sisältöpuun käyttämät luokat. Sovellus ei voi lukea XML-dokumenttia sen rakenteen kaltaiseen ohjelmoijan itsensä tuottamaan oliomalliin, sillä sisältöpuun rajapinnat ja luokat tulee olla muodostettu sidontakääntäjällä.

Toisaalta JAXB tarjoaa esivalmistelujen jälkeen oliomaisen ja lähdekooditasolla yksinkertaisen tavan käsitellä XML-dokumentteja. JAXB ei kuitenkaan ole ainoa teknologia XML-datan käsittelyyn. Tarjolla on niin Sun Microsystemsin aiemmin kehittämiä kuin myös JAXB-tekniikkaa vastaavia avoimen lähdekoodin ratkaisuja[3] sekä kaupallisia[4] toteutuksia. Sunin SAX-teknologia edellyttää osaamista ja ymmärtämistä XML:sta, koska se on yksityiskohtainen jäsennin XML-dokumenttien läpikäymiseen. SAX on toisaalta nopea sekä tehokas muistinkäytön suhteen, koska se ei luo ylläpidettävää oliomallia dokumentin elementeistä (Sun Microsystems, 2003d). Toinen Sunin teknologia on DOM. Sen perusajatuksena on luoda ajonaikainen oliopuu, johon XML-dokumentti ladataan (Sun Microsystems, 2003d). Se on oliolähtöinen tapa käsitellä XML-dataa, mutta sen käyttö vaatii sovelluksen puolella tarkan tietämyksen käytettävän XML-dokumentin rakenteesta. Lisäksi muodostettava oliopuu vie merkittävästi enemmän muistia kuin vastaava JAXB-sisältöpuu (Ort & Mehta, 2003). JAXB-teknologian etu on, että sovelluksen ohjelmoijien ei tarvitse välittää XML-datan yksityiskohdista tai dokumentin rakenteesta. Ohjelmoijat käyttävät vain tarjolla olevia luokkia ja niiden metodeita.

XML-dokumentti olioiden varastona tarjoaa hyvän mahdollisuuden siirtää tietoja täysin erilaisten sovellusten välillä. Lisäksi tallennettua dataa voidaan lukea vapaasti, koska tiedot ovat (rakenteisessa) tekstitiedostossa, eivätkä esimerkiksi JOS-teknologian käyttämässä binaarimuodossa. Tällöin olemassaolevia XML-dokumentteja, siis oliovarastoja, voidaan muokata myös ilman sovellusta. Mikäli tiedostojärjestelmän tarjoama tietoturva tai tapahtumien hallinta nähdään riittämättömäksi, on mahdollista käyttää myös XML-tietokantoja (Graves, 2002).

JAXB-teknologian käyttö ei ole kuitenkaan perusteltua, mikäli ainoa tarkoitus on olioiden pysyvyyden toteuttaminen. Syitä tähän on useita. Ensinnäkin suorituskyvyllisesti JAXB on tehottomampi kuin JOS (Hericko & al., 2003). Toiseksi JAXB-teknologian käyttämiseksi tarvitaan osaamista XML-skeeman määrittelystä W3C XML Schema Language -kielellä. Henkilökohtaisen kokemukseni perusteella yksinkertaisen luokan määrittelyn tekeminen ei ole ylivoimaista, mutta ohjelmoijan tulee hyväksyä ajatus yhden ylimääräisen kielen opettelusta. Kolmas syy on, että JAXB ei tarjoa kyselymekanismia sisältöpuun dataan eikä myöskään tukea millekään XML-kyselykielelle (esimerkiksi XPath tai XQL). Ohjelmoija joutuu

[3] Esimerkiksi Castor, `http://www.castor.org/` (30.8.2004).

[4] Esimerkiksi JSX, `http://www.jsx.org/` (30.8.2004).

itse toteuttamaan haut XML-dokumentista muodostettuihin olioihin. Tämä on triviaalia listan käsittelyä, mutta vaatii ylimääräistä työtä.

Ylläpidettävyys JAXB-sovelluksissa on kaksitahoista: sovelluksen lähdekoodin ylläpidettävyys ei ole raskasta, mutta sisältöpuu mahdollistaa vain yhden version pysyvistä liiketoimintaluokista. Lähdekoodin ylläpidettävyyttä helpottaa se, että Java-kääntäjä huomaa sidontakääntäjän generoimista luokista aiempiin versioihin ristiriidassa olevat seikat, kuten luokan attribuutin poistamisen tai sen tyypin vaihtamisen. Oletetaan esimerkiksi, että sovellus käyttää pysyvää luokkaa `A`, jolla on attribuutit `a1` sekä `a2`, ja täten myös generoidut metodit `getA1()` sekä `getA2()`. Oletetaan lisäksi, että sovellus käyttää molempia metodeita. Jos ohjelmoija poistaa luokkaa kuvaavasta skeemasta attribuutin `a2` ja generoi sidontakääntäjällä uuden luokan `A`, niin tähän uuteen pysyvään luokkaan ei tule metodia `getA2()`. Mikäli sovellus käännettäisiin nyt uudelleen, huomaisi Java-kääntäjä ristiriidan luokan `A` ja sovelluksen välillä, koska luokassa `A` ei ole enää sovelluksen aiemmin käyttämää metodia `getA2()`.

Täten monet ajonaikaiset virheet saadaan poimittua pois helposti eikä skeeman muokkaamisen takia tarvitse käydä koko sovellusta yksityiskohtaisesti läpi. Toisaalta skeeman muokkaaminen synnyttää sovellukselle uuden sisältöpuun eli oliomallin rakenne muuttuu. Tämä johtaa siihen, että aiemmin kirjoitetut XML-dokumentit eivät ole enää valideja, jolloin niiden takaisinjärjestäminen uuteen sisältöpuuhun ei välttämättä onnistu. Ylläpidettävyyden ongelmat kasautuvat siis vanhempaa versiota olevien XML-dokumenttien käsittelyyn. Tämän ratkaisemiseksi joudutaan sovellukseen lataamaan niin uusi kuin myös vanha oliomalli tai konvertoimaan vanhat XML-dokumentit uuden skeeman mukaisiksi.

XML-sidonnan arkkitehtuuri sopii sovelluksiin, joiden pysyvien olioiden tulee olla helposti jaettavissa muille osapuolille. Sen käyttäminen vaatii joukon jar-paketteja sekä perehtymisen sidontakääntäjän toimintaan, mutta tietojenkäsittely säilyy oliomaisella tasolla. Tiedon viemiset ja tuomiset tietolähteeseen hoituvat muutamilla peruskomennoilla. Käytännössä suuret tietomäärät ja niihin kohdistuvat haut eivät sovellu JAXB-teknologialle, varsinkaan mikäli sovellukseen kohdistuu suorituskykyvaatimuksia.

5 Tietokantarajapinta

Tässä luvussa kerrotaan Javan standarditeknologiasta, joka on tarkoitettu ulkoisten tietolähteiden hallintaan eli muun muassa sovelluksen käsittelemien tietojen tallentamiseen. Teknologiasta on tullut de-facto -standardi erityisesti relaatiotietokantojen käsittelyssä. Se tunnetaan nimellä Java Database Connectivity (JDBC). Luvun alussa kohdassa 5.1 olevan yleiskuvauksen jälkeen esitetään kohdassa 5.2 JDBC-arkkitehtuuri ja tämän jälkeen kohdassa 5.3 on esimerkkejä, kuinka sovelluksen oliot voidaan tehdä pysyviksi. Tämän luvun lopuksi kohdassa 5.4 pohditaan tarkasteltavan teknologian soveltuvuusaluetta.

5.1 Yleiskuvaus

Aluksi selvennetään käsitettä relaatiotietokanta ja tämän jälkeen aloitetaan tutustuminen tietokantarajapintaan.

Relaatiotietokannat ovat relaatiotietokannanhallintajärjestelmien hallinnoimia tietolähteitä. Relaatiotietokantoja käytetään, kun käsiteltävät tietomäärät ovat suuria ja niiden käsittelyyn liittyy tehokkuus-, luotettavuus- ja turvallisuusvaatimuksia. Relaatiotietokantojen ajatusmalli kehitettiin 1970-luvulla ja nykyään ne ovat ehkä kaikkein yleisin tapa tallentaa sovellusten käsittelemää tietoa (Sperko, 2003). Relaatiotietokannassa tiedot jaotellaan *tauluihin*, joissa on *sarakkeita* ja *rivejä*. Yksittäinen taulu on kokoelma tietoja. Taulun rivi kuvaa kokoelman yhden elementin tiedot. Taulun sarake kuvaa elementin yhden ominaisuuden.

Eri toimittajien tietokantajärjestelmät ovat toteutukseltaan erilaisia. Yhteisiä tekijöitä ovat tarkoitus ja samaan kyselykieleen perustuminen. Jokainen toimittaja tarjoaa oman yksilöllisen sovellusliittymän (API), jonka kautta ohjelmat voivat olla vuorovaikutuksessa tietokannan kanssa. Siirrettävyyden ja yksinkertaisemman ohjelmavoitavuuden vuoksi tietokannanhallintajärjestelmien ja sovellusten väliin on kehitetty standardeja rajapintoja (Henry, 2001). Tietokannan toimittaja tarjoaa kantaan standardin mukaisen ajurin, jota käyttämällä sovellukset voivat olla vuorovaikutuksessa tietokannan kanssa.

Sun Microsystems julkaisi vuonna 1997 oman toimittaja- ja alustariippumattoman sovellusliittymäspesifikaation, JDBC, Java-ohjelmointikielen ja tietokantojen välille. JDBC on stan-

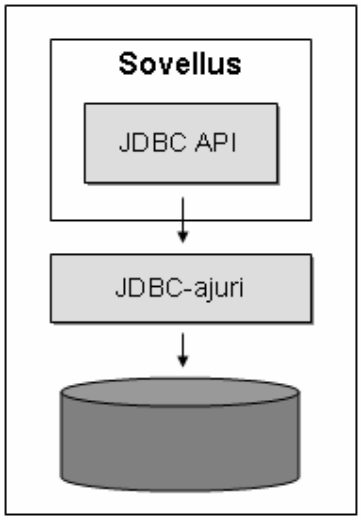
dardi, joka antaa Java-sovelluksille mahdollisuuden käyttää erilaisia ulkoisia tietolähteitä tietämättä tietolähteen toteutuksellisia yksityiskohtia (Sun Microsystems, 2001). Pääosissa tapauksista tämä tietolähde on relaatiotietokanta, jota käsitellään SQL-kielellä. Tietolähteeksi voidaan kuitenkin valjastaa JDBC-ajuriteknologian avulla myös tiedostojärjestelmä tai oliotietokanta. Nykyään kaikki markkinoiden johtavat tietokanta- sekä väliohjelmistotoimittajat tukevat JDBC-teknologiaa tarjoten sovelluksille sopivan JDBC-ajurin (Sun Microsystems, 2000b).

5.2 JDBC-arkkitehtuuri

JDBC-teknologia pohjautuu Opengroupin X/Open SQL Call Level Interface (SQL CLI) -määrityksiin (Sun Microsystems, 2001). SQL CLI standardissa kuvataan säännöt SQL-kielen ja ohjelmointikielen väliseen liitäntään. Lisäksi JDBC on sitoutunut noudattamaan SQL99 standardeja.

JDBC-spesifikaatio voidaan jakaa arkkitehtonisesti kahteen päätasoon: sovellusliittymään sekä sovelluksen ja tietolähteen välisen vuorovaikutuksen hoitavaan toteutukseen (Sun Microsystems, 2000b). JDBC Application Programming Interface (JDBC API) on ohjelmistorajapintaspesifikaatio Java-sovellusten ohjelmoijille. Se on kokoelma rajapintakuvauksia ja abstrakteja luokkia, jotka esittelevät operaatiot yhteyden luomiseksi ja SQL-lauseiden kohdistamiseksi tietolähteeseen sekä operaatiot SQL-lauseiden synnyttämien tulosjoukkojen käsittelyyn. Matalammalla tasolla toimivat JDBC-ajurit toteuttavat JDBC API:n rajapinnat ja luokat mahdollistaen tietolähteen ja sovelluksen välisen vuorovaikutuksen. JDBC-ajureiden toteuttamiseen käytettävä rajapinta on nimeltään JDBC driver API.

Kuvassa 5.1 esittelen näkemykseni JDBC-teknologian yleisarkkitehtuurista. Kuvassa selvennetään Java-sovelluksen, JDBC API:n, JDBC-ajurien ja tietokannan välistä yhteyttä. JDBC API:ia voidaan käyttää niin kaksi- kuin kolmikerrosarkkitehtuurisissa ratkaisuissa. JDBC-tekniikan kannalta ei ole merkitystä, käytetäänkö sitä suoraan käyttöliittymän, liiketoimintalogiikan vai sovelluslogiikan tasolta (Sun Microsystems, 2001). Toisaalta tällä on merkitystä itse sovellukselle ylläpidettävyyden ja muunneltavuuden kannalta.



Kuva 5.1: JDBC-teknologian arkkitehtuuri.

JDBC-teknologialla Java-sovellukset voivat käsitellä tietokantoja saaden kaikki tietokantoihin liittyvät edut. Sperkoa (2003) lainatakseni tietokannanhallintajärjestelmä pitää huolen muun muassa tiedon suojauksesta, kyselyiden suorituksesta sekä käyttäjien ja transaktioiden hallinnasta. JDBC-sovellusliittymän avulla voidaan luoda tietokantaan tauluja, lisätä, poistaa ja päivittää niiden tietoja sekä suorittaa kyselyitä. Lisäksi kyselyiden tuloksia voidaan selata ja muokata siten, että muutokset reflektoituvat tietokantaan. Transaktioille voidaan asettaa kirjanmerkkejä ja niitä voidaan hajauttaa useaan tietokantaan. Lisäksi yksi tämän teknologian vahvoista puolista on siirrettävyys ja toimittajariippumattomuus. Sovellus voi vaihtaa käytettävää tietolähdettä esimerkiksi Oraclen tietokannasta MySQL-kantaan vaihtamalla JDBC-ajuria. Ohjelmakooditasolla joudutaan vaihtamaan vain merkkijono, jossa määritellään ladattavan JDBC-ajurin nimi.

Sun Microsystems on määritellyt neljä erilaista JDBC-ajurityyppiä, joilla kullakin on oma tapansa hoitaa vuorovaikutus sovelluksen ja tietolähteen välillä (Sun Microsystems, 2001). Tietolähteen toimittajat voivat toteuttaa ajurit ODBC-rajapinnan läpi, kohdetietokannan oman sovellusliittymän kautta tai tulkkaamalla JDBC-kutsut tietokantariippumattomaksi protokollaksi tai tietokantaspesifiseksi verkkoprotokollaksi. Sovelluksen käyttämän ajurin valinta riippuu toimittajan tuesta ja sovelluksen tarpeista. Useimmiten toimittajat tarjoavat laajan joukon JDBC-ajureita erilaisilla ominaisuuksilla varustettuina (Sun Microsystems, 2000b).

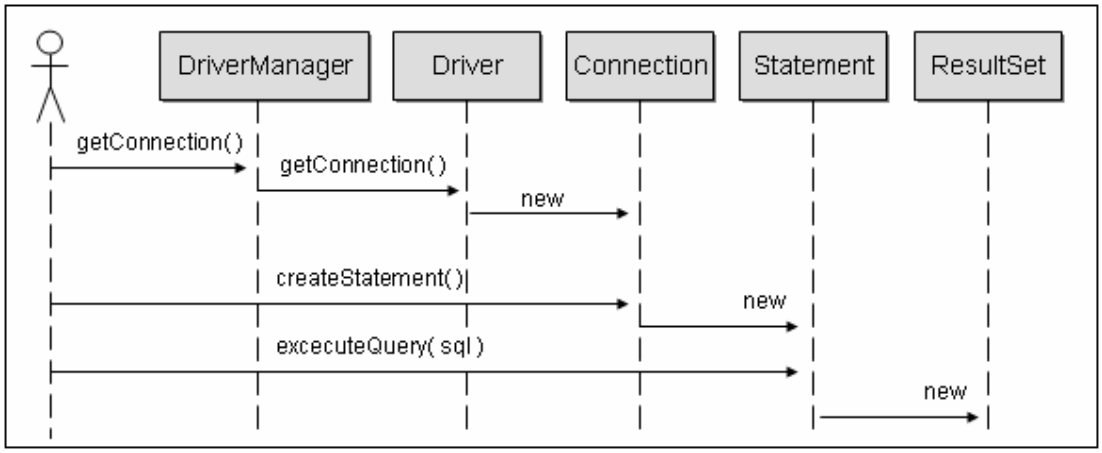
5.3 Pysyvyyden toteuttaminen

Seuraavaksi tarkastellaan, kuinka oliot voidaan tallentaa JDBC-tekniikkaa käyttämällä ja miten nämä oliot voidaan myöhemmin palauttaa sovelluksen käyttöön. Aluksi esitetään JDBC-sovellusliittymän tärkeimmät rajapinnat ja luokat, jonka jälkeen voidaan syventyä niiden käyttöön.

JDBC-sovellusliittymä on kiinteä osa Sun Microsystemsin Java-toteutusta: JDBC API on sijoitettu pakkauksiin `java.sql` ja `javax.sql`. Tarvittavan JDBC-ajurin voi yleensä ladata tietokannan toimittajan Internet-sivuilta. Näin ollen JDBC on varsin helppo ottaa käyttöön. Itse olioiden pysyvyyden toteuttamiseen tarvittavia rajapintoja ja luokkia on suhteellisen vähän, etenkin mikäli sovellus tarvitsee vain teknologian perusominaisuuksia. Yksinkertaisimmillaan JDBC-sovellusliittymää käyttäen olion pysyvyys voidaan saavuttaa seuraavalla toimenpidesarjalla:

1. Luo yhteys tietokantaan.
2. Tuota SQL-lause tarvittavien tietojen syöttämiseksi.
3. Suorita muodostettu SQL-lause.

Olion palauttamisessa suoritettava lause vaihdetaan SQL-syntaksin mukaiseen kyselyyn ja lauseen suorittaminen palauttaa tietolähteestä vastaavan oliojoukon tiedot. Kuvassa 5.2 on esitetty jälkimmäisen toimenpidesarjan suorittamiseen käytetyt JDBC API:n luokat sekä rajapinnat ja niiden väliset yhteydet.



Kuva 5.2 JDBC API:n pääluokat ja niiden välinen vuorovaikutus (Sperko, 2003).

Kuvassa 5.2 osoitetaan tärkeimpien JDBC-sovellusliittymän luokkien ja rajapintojen välinen yhteys. Sperko (2003) tulkitsee kuvaa seuraavasti. `DriverManager`-luokka, eli *ajurinhoitaja*, on vastuussa yhteydenpidosta ladattuun JDBC-ajuriin (`Driver`) ja uusien yhteyksien muodostamisesta tietolähteeseen. Kaikki vuorovaikutus ajurin kanssa tapahtuu `DriverManager`-luokan staattisten metodien kautta. Rajapintaa `Connection` käytetään ylläpitämään asiakasyhteyttä tietolähteeseen. `Connection`-rajapinnan toteuttaa JDBC-ajurin toimittaja eikä sovelluksen ohjelmoijan tarvitse tietää toteutuksen todellisia yksityiskohtia. Kaikki tiedon käsittely, kuten lisäys tai päivitys, tapahtuu tämän rajapinnan kautta, pyytämällä siltä rajapinnan `Statement` toteuttava olio. Rajapintaa `Statement` käytetään tekemään varsinainen työ tietolähteeseen. Se käärii SQL-lauseet vaadittuun muotoon ja suorittaa ne tietolähteeseen. Sitä voidaan käyttää muuttamaan tietolähteen sisältöä tai palauttamaan sieltä tietoa. SQL-kyselyiden tulokset palautuvat rajapinnan `ResultSet` toteuttavassa oliossa. `ResultSet`-rajapinta toimii osoittimena tietolähteeseen ja sitä voidaan käyttää iteroitaessa kyselyn tuloksia, esimerkiksi tietokannan rivejä. Sun Microsystemsin (2000b) mukaan yhteyden muodostaminen tietolähteeseen onnistuu myös `DataSource`-rajapinnan kautta, joka mahdollistaa `DriverManager`:ia paremman siirrettävyyden ja ylläpidettävyyden.

Seuraavissa kohdissa tarkastellaan esimerkkien kautta JDBC-teknologian käyttöä olioiden pysyvyyden toteuttamisessa. Käsiteltäviä esimerkkejä ovat yhteyden muodostaminen, olion pysyvyyden toteuttaminen ja tietojen hakeminen.

5.3.1 Yhteyden muodostaminen

Yhteyden muodostaminen tietokantaan on perustehtävä suoritettaessa mitä tahansa muita operaatioita. Se aloitetaan JDBC-ajurin latauksella. JDBC-ajuri ladataan ohjelman käyttöön suorittamalla luokan `Class` staattinen metodi `forName(String className)`. Metodilla etsitään ja ladataan käytettävä JDBC-ajurin luokka. Luokkaa ensimmäistä kertaa ladattaessa järjestelmä tekee kaikki tarvittavat staattiset alustukset ja rekisteröi ajurin ajurinhoitajaan. Rekisteröinnin jälkeen ajuri on valmis käsittelemään sovelluksen tekemät yhteyspyynnöt (Henry, 2001).

Ajurin latauksen jälkeen sovelluksen tulee muodostaa yhteys käytettävään tietokantaan. Yhteys saadaan `DriverManager`-luokan `getConnection`-metodilla, jolle annettaan paramet-

reiksi URL (Unified Resource Locator), käytettävä käyttäjätunnus ja salasana. URL on yhteyden muodostamiseen käytettävä merkkijono ja se muotoa `jdbc:aliprotokolla:alinimi`. Esimerkiksi luotaessa yhteys paikalliseen MySQL-tietokantaan, URL voisi olla `jdbc:myslq://localhost:3306/dbname`. Tässä aliprotokolla `mysql` kertoo ajurinhoitajalle, mitä rekisteröityä ajuria tulee käyttää ja alinimi `//localhost:3306/dbname` kertoo ajurille tietokannan sijainnin, käytettävän portin ja tietokannan nimen. Alinimen perässä olevaa portin numeroa ei ole välttämätöntä antaa, mikäli käytetään MySQL:n oletusporttia 3306 (Sperko, 2003).

Mikäli luokka muodostaa toistuvasti uuden yhteyden tietokantaan voidaan luokkaan tehdä oma metodi ajurin latausta ja yhteyden muodostamista varten. Tällainen metodi esitetään kuvassa 5.3. Se lataa ajurin, luo uuden yhteyden tietokantaan sekä palauttaa yhteysolion (Connection) metodin kutsujalle. Mikäli suorituksessa tapahtuu virhe, metodi nostattaa `SQLException`-poikkeuksen. Kuvan 5.3 metodissa oletetaan, että yhteyden muodostamisessa käytetään aina tietokantaa `jop`, sen käyttäjätunnusta `kayttajatunnus` ja salasanaa `salasana`. Metodin käyttäjän tulee itse huolehtia tarpeettoman yhteyden sulkemisesta kuvan 5.4 mukaisesti.

```
private Connection haeYhteys() throws SQLException  {

  Connection connection = null; // rajapinta java.slq.Connection

  try {
    /* Lataa ajuri, mikäli sitä ei ole ladattu kertaakaan */
    Class.forName("com.mysql.jdbc.Driver").newInstance();

    connection = DriverManager.getConnection(
           “jdbc:mysql://localhost:3306/” + “jop”,
           “kayttajatunnus”, “salasana”);

  } catch (Exception exc) {
      throw new SQLException(exc.getMessage());
  }

  return connection;
}
```

Kuva 5.3: JDBC-ajurin lataus ja yhteysolion muodostaminen (Rouvinen, 2004).

Uuden yhteyden muodostaminen tiedetään melko raskaaksi operaatioksi ja tällöin on järkevää käyttää luotuja yhteyksiä useasti (Sperko, 2003). Yhteyksien uudelleenkäyttö on mahdollista yhteysvarannon (Connection Pooling) avulla. Yhteysvaranto on palvelintekniikka, jossa so-

vellus pyytää yhteysoliota varannosta tarpeen mukaan ja palauttaa sen sinne käytön jälkeen. Varanto pitää huolta tarvittavien yhteyksien määrästä ja tilasta.

5.3.2 Olioiden tallentaminen

Onnistuneen yhteyden muodostamisen jälkeen voidaan yhteysoliolta pyytää lauseoliota, jonka kautta voidaan suorittaa SQL-kielen mukaisia lauseita tietokantaan ja tarkastella niiden synnyttämiä tuloksia. Olio saadaan rajapinnan `Connection` metodilla `createStatement()`. Tämä palauttaa rajapinnan `Statement` mukaisen lauseolion. Yksinkertaisimmissa tapauksissa lauseen ajoon riittää pelkän `execute(String sql)` tai `executeUpdate(String sql)` -metodin kutsuminen. Riippuen metodista, lauseen suorittaminen palauttaa totuusarvon, kokonaisluvun tai tulosjoukon `ResultSet`. Kuvassa 5.4 osoitetaan, kuinka liiketoimintaolio saadaan pysyväksi näiden rajapintojen avulla.

Kuten kuvasta 5.4 voidaan huomata, JDBC-teknologia on hyvin hienojakoinen tapa käsitellä olioiden tietoa. Tietokannan taulun skeema perustuu olion rakenteeseen ja sovellustasolla tämä skeema on tunnettava tarkasti. Vaikka JDBC-ajurit huolehtivatkin tietotyyppien konversioista sovelluksen ja tietolähteen välillä, tulee olion tallennuksessa sen attribuutit syöttää kyselyyn täsmälleen oikeassa järjestyksessä.

Kuvan 5.4 koodilistauksessa olevassa `finalize`-lohkossa kutsutaan esimerkkiluokan omaa operaatiota, joka hoitaa varattujen resurssien vapauttamisen. On erittäin tärkeää kutsua `close()`-metodia kaikille tuotetuille tulosjoukoille, lause- ja yhteysolioille. Muutoin voidaan törmätä tilanteeseen, että sovelluksen ajon aikana nämä resurssit loppuvat eikä sovellus pysty suorittamaan sille annettuja tehtäviä (Sun Microsystems, 2001).

Operaatiot liiketoimintaolion poistamiseksi tietolähteestä ja sen tietojen päivittämiseksi ovat hyvin samankaltaisia kuvan 5.4 operaation kanssa. Ne voidaan toteuttaa pelkkää suoritettavaa SQL-lausetta muuttamalla. Mikäli halutaan tehdä tarkistuksia, kuten onko olio tietokannassa ennen poistamista, joudutaan suorittamaan lisäkyselyitä. Tällainen tarkistuskysely voidaan suorittaa kuvan 5.5 esittämällä tavalla.

```
public boolean tallennaLevy(LevyPapu levy) {

  Connection connection = null;
  Statement  statement  = null;
  int        tulos      = 0;

  if ((null == levy) || ("".compareTo(levy.getNimi()) == 0)) {
    return false; /* ei levyä tai levyn nimi on tyhjä */
  }

  try {
    connection = haeYhteys();                  /* ks. kuva 5.3 */
    statement  = connection.createStatement(); /* muodosta lauseolio */

    String sql = "INSERT INTO " + TAULUN_NIMI + " VALUES "
          + "("
          + "'" + levy.getNimi()     + "',"
          + "'" + levy.getSisalto()  + "',"
          + "'" + levy.getLevynKoko() + "',"
          + "'" + (true==levy.onkoUudelleenKirjoitettavissa() ? 1:0)+"'"
          + ")";

    tulos = statement.executeUpdate(sql);      /* suorita SQL-lause */
    if (tulos != 1) {
      throw new SQLException("Levyoliota ei saatu pysyväksi!");
    }

  } catch (SQLException SQLExc) {
      SQLExc.printStackTrace();
      return false;
  } catch (Exception exc) {
      exc.printStackTrace();
      return false;
  } finally {
      closeAll(connection, statement, null);  /* vapauta resurssit */
  }

  return true;
}
```

Kuva 5.4: `LevyPapu`-luokan esiintymän tallentaminen tietokantatauluun (Rouvinen, 2004).

5.3.3 Olioiden palauttaminen ja kyselyt

JDBC-teknologian avulla tietolähteeseen voidaan suorittaa kyselyitä. SQL-pohjaiset haut mahdollistavat tarkan tietojen poimimisen sekä erilaisten olioiden osajoukkojen käsittelyn. Hakumekanismi on monipuolinen, mutta valitettavasti ei oliosuuntautunut. Itse sovelluksen ei tarvitse huomioida kyselyiden optimointeja tai sitä, kuinka suuria tietomääriä käsitellään. Ne ovat tietokannanhallintajärjestelmän ja JDBC-ajurin vastuulla.

Oliot palautetaan sovelluksen käyttöön tietokannasta suorittamalla JDBC-kysely. Kyselyyn määritellään halutut hakukriteerit ja kriteereitä vastanneesta tulosjoukosta muodostetaan sovelluksen käyttöön vastaavat oliot. Kyselyn suorittaminen on identtistä olion tallentamisen kanssa yhteyden ja lauseolion luomisen osalta. Suoritettava kysely muodostetaan SQL-kielen sääntöjen ja kohdetietokannan perusteella. Kyselyt suoritetaan `executeQuery(String sql)`-metodilla, joka palauttaa `ResultSet`-tulosjoukon. Tulosjoukko pitää sisällään SQL-kyselyn vastauksen. Tulosjoukko-olion tietoja voidaan iteroida riveittäin ja rivin sarakkeiden arvoihin päästään käsiksi saantimetodeilla (Sperko, 2003). Toisin sanoen rivin sarakearvoihin päästään käsiksi metodeilla, kuten `getString(String columnName)`, `getBoolean(String columnName)`. Vaihtoehtoisesti sarakearvoja voidaan pyytää sarakkeen järjestysnumeron mukaan, esimerkiksi `getString(1)` palauttaisi rivin ensimmäisen merkkijonotyyppisen sarakkeen arvon.

Kuvan 5.5 esimerkkimetodin kyselyn tulosjoukko sisältää sen `LevyPapu`-olioiden taulun rivin, missä `nimi`-kenttä täsmää metodin parametrina saamaan merkkijonoon. Mikäli täsmäystä ei ole, tulosjoukko on tyhjä. Tulosjoukon rivistä voidaan muodostaa uusi `LevyPapu`-olio poimimalla rivin kaikki tiedot olion rakentajaan. Relaatiotietokannasta ei siis saada valmiita olioita, vaan ne on rakennettava sovelluksessa manuaalisesti uudelleen.

`ResultSet`-rajapinnan toteuttamaa oliota voidaan käyttää myös muuhun kuin vain tietojen tarkasteluun. Tulosjoukosta voidaan pyytää sen *metadata* (Sun Microsystems, 2001). Metadata kuvaa tulosjoukon, esimerkiksi tietokantataulun, rakenteen. Tämän saanti mahdollistaa joustavien toteutusten laatimisen. Riippuen tulosjoukon ominaisuuksista, sen kautta voidaan myös poistaa tietokannan rivejä sekä päivittää kenttien arvoja (Sun Microsystems, 2001). Samoin tulosjoukkoa voi olla mahdollista selata tarpeen mukaan eteen- ja taaksepäin.

```
public LevyPapu etsiLevy(String nimi) {

  LevyPapu   levy       = null;
  Connection connection = null;
  Statement  statement  = null;
  ResultSet  resultSet  = null;

  if (null == nimi) {
    return null;
  }

  /* Etsi rivi, millä 'nimi' täsmää parametrina annettuun merkkijonoon*/
  try {
    connection = haeYhteys();                          /* ks. kuva 5.3 */
    statement  = connection.createStatement();

    String sql = "SELECT * FROM " + TABLE_NAME + " WHERE "
        + "nimi = '" + nimi + "'";

    resultSet = statement.executeQuery(sql);

    /* siirry tulosjoukon ensimmäiselle riville ja luo uusi levy */
    if (resultSet.next()) {
      levy = new LevyPapu(resultSet.getString("nimi"),
               resultSet.getString("sisalto"),
               resultSet.getDouble("koko"),
               resultSet.getBoolean("onkoUudelleenKirjoitettavissa"));
      /* voitaisiin tehdä myös:
         "new LevyPapu(resultSet.getString(1), resultSet.getString(2).."
      */
    }

  } catch (SQLException SQLExc) {
      SQLExc.printStackTrace();
  } catch (Exception exc) {
      exc.printStackTrace();
  } finally {
      closeAll(connection, statement, resultSet);
  }

  return levy;
}
```

Kuva 5.5: Liiketoimintaolion etsiminen ja palauttaminen tietokannasta.

Kyselyitä voidaan suorittaa tietolähteeseen muutoinkin kuin vain yksi staattinen lause kerrallaan. *Valmistellut lauseet* (prepared statements) ovat lauseita, joihin määritellään kyselyn runko ja kenttien arvot annetaan dynaamisesti muuttujien arvojen perusteella. Niitä on järkevä käyttää, mikäli samaa kyselyä suoritetaan toistuvasti eri arvoilla (Sperko, 2003). JDBC mahdollistaa myös kyselyiden eräajon. Tällöin useita SQL-lauseita, esimerkiksi `INSERT`, kootaan yhteen lauseolioon, joka suorittaa ne yhtenä eränä.

5.4 Soveltuvuusalue

Tässä luvussa on käsitelty JDBC-teknologiasta vain pieni osa. JDBC-spesifikaatio sisältää laajan joukon rajapintoja ja operaatioita vaativaan käyttöön, joskin käytännössä useat ominaisuudet ja niiden toteutukset riippuvat paljolti käytettävästä ajurista ja sen tuesta (Sperko, 2003). Edellä esitetyillä operaatioilla voidaan kuitenkin toteuttaa olioiden pysyvyys ja näin ollen lisäominaisuuksista keskustelu ei toisi lisäarvoa tälle tutkielmalle. Sitä vastoin seuraavassa on pohdintoja JDBC-teknologian hyvistä ja huonoista puolista sekä käyttötarkoituksesta.

Tietokantarajapinta on siis eräänlainen abstraktio, jolla häivytetään tiedon tallennuksen yksityiskohtia pois, esimerkiksi tietokannan fyysistä sijaintia tai toimittajaa ei ole välttämätöntä tietää. JDBC:n näkymä tietokantaan on keskittynyt relaatiomalliin, joka on valitettavasti hyvin kaukana oliokeskeisestä näkymästä. Tämän johdosta joudutaan usein toteuttamaan ylimääräinen sovelluskerros tietokannan ja sovelluksen välille huolehtimaan Java-olioiden *pilkkomisesta* (decomposing) (Sperko, 2003). Pilkkomisessa olion ominaisuudet paloitellaan pienemmiksi osiksi, siten että olio voidaan tallentaa relaatiotietokantaan. Vastaavasti on toteutettava pilkkomisen käänteinen operaatio, joka lukee tietokannan sarakkeita ja rivejä ja muodostaa niistä olioita.

Lisäkerros voidaan nähdä joko ylimääräisenä työnä tai uudelleenkäytettävyyttä sekä ylläpidettävyyttä parantavana komponenttina. Hyvällä olioparadigman mukaisella suunnittelulla lisäkerroksesta saadaan niin kutsuttu *tiedonhallinnan kerros* (data management tier), joka erottaa sovelluslogiikan ja pysyvyyden toteuttamisen toisistaan. Kerrostamisen suunnittelussa käytetään usein *suunnittelumalleja* (Sperko, 2003).

Tietokantarajapinnan käyttö on luonnollinen valinta, mikäli sovelluksen tulee käsitellä relaatiotietokannassa jo olemassaolevaa tietoa, koska se on suunniteltu relaatiotietokantojen käyttöön. SQL-tuen ansiosta se on myös hyvä valinta, mikäli sovelluksen tulee tehdä paljon laskentatehoa vaativia mielivaltaisia kyselyitä suureen määrään rakenteista tietoa. Sun Microsystemsiä (2000b) mukaillen JDBC:n eduksi voidaan katsoa myös toimittajariippumattomuus, helppo opittavuus sekä sen kiinteä yhteys Java-ympäristöön. Ohjelmakoodi on siirrettävää eikä erillisasennuksia ei tarvita.

JDBC-tekniikan käyttö ei kuitenkaan ole perusteltua tavoitteen ollessa pelkkä olioiden pysyvyyden saavuttaminen. Mikäli ohjelma ei tarvitse tietokannan palveluita, tai kyselyitä pysyvään tietoon, ei JDBC:stä saada vastaavaa hyötyä työmäärään nähden. SQL-kieli ei ole sidoksissa Java-kieleen ja näin ollen teknologian käyttö vaatii ylimääräistä osaamista yhdestä täysin erilaisesta ohjelmointikielestä. Lisäksi oliomallia käännettäessä relaatiomalliin yleensä törmätään pienempiin tai suurempiin ylläpito- sekä mallinnusongelmiin (Baldwin, 2003). Toisaalta tietokantatasolla muunnokseen voidaan käyttää tarkoitukseen sopivia tietokannan generoijia ja sovellustasolla suunnittelumalleja. JDBC-teknologia vaatii myös tarkkaa tuntemusta käytettävän tietokannan skeemasta. Kaikki muutokset tietokannan taulujen rakenteeseen aiheuttavat tarkistuksen ja mahdollisesti muutostarpeen kaikkiin sovellukseen sulautettuihin SQL-lauseisiin. JDBC ei toteuta miltään osin ortogonaalisen persistenssin periaatteita, koska ohjelmoija on täysin tietoinen olioiden pysyvyydestä.

6 Dataoliot

Olioiden pysyvyyden saavuttamiseksi on tähän mennessä tutustuttu kahteen tiedostopohjaiseen ratkaisuun ja yhteen pääosin relaatiotietokantojen käsittelyyn tarkoitettuun tekniikkaan. Neljäntenä tekniikkana olioiden pysyvyyden toteuttamiseen esitellään Java Data Objects (JDO). Tässä luvussa selvitetään aluksi kohdassa 6.1 JDO-teknologian kehittämiseen johtaneita syitä sekä kuvataan sen yleisarkkitehtuuri kohdassa 6.2. Tämän jälkeen tätä teknologiaa tarkastellaan yksityiskohtaisemmin esimerkkien kautta kohdassa 6.3. Kuten muidenkin edellä käsiteltyjen tekniikoiden kohdalla, myös tämän luvun lopussa kohdassa 6.4 on pohdintaa teknologian hyvistä ja huonoista puolista sekä mahdollisista käyttökohteista.

6.1 Yleiskuvaus

Tuntumattomuus (transparent) olioiden pysyvyydessä tarkoittaa mahdollisuutta käsitellä tietolähteeseen tallennettuja olioita suoraan olio-ohjelmointikielen kautta (Cattell & al., 2000). Sovelluksessa olioiden käsittelyyn ei tällöin tarvita tietokantakohtaista alikieltä, kuten SQL. Tietolähteen pysyviä olioita tulisi voida käsitellä muistissa olevien väliaikaisten olioiden tavoin. Tuntumattomuuden tavoitteena on saavuttaa oliokeskeisiä ratkaisuja ja välttää toteutuksen monimutkaisuutta.

Object Data Management Group (ODMG) on kehittänyt toimittajariippumattoman standardin oliotietokantojen käsittelyyn sekä olio-relaatiosidonnan toteuttamiseen (Cattell & al., 2000). ODMG-spesifikaatio määrittelee siirrettävän, olio-ohjelmointikieliriippumattoman ja sovelluksen ohjelmoijalle tuntumattoman tavan saavuttaa olioiden pysyvyys. Perusideana on kapseloida tietolähteeseen liittyvät yksityiskohdat, kuten tietotyypit tai yhteyden hallinta, ja antaa sovelluksen kehittäjän keskittyä liiketoimintamalleihin ja oliomallinnukseen. Tiedon käsittelyn yksityiskohdat jäävät ODMG-spesifikaation toteuttajan vastuulle.

ODMG-standardin pohjalta Java Commynity Process kehitti vuonna 2001 julkaistun Java Data Objects -spesifikaation (Samson, 2001). Se on periaatteiltaan hyvin esikuvansa kaltainen, mutta se on laadittu täysin Java-keskeiseksi ja näin siitä on saatu kevyempi. Se luo rajapinnan sovellusten ja tietolähteiden välille, kuten JDBC-tekniikkakin, mutta tekee sen sovelluksen kannalta yksinkertaisemmin. JDO nähdään sovelluskehitystä nopeuttavana lisäkerrok-

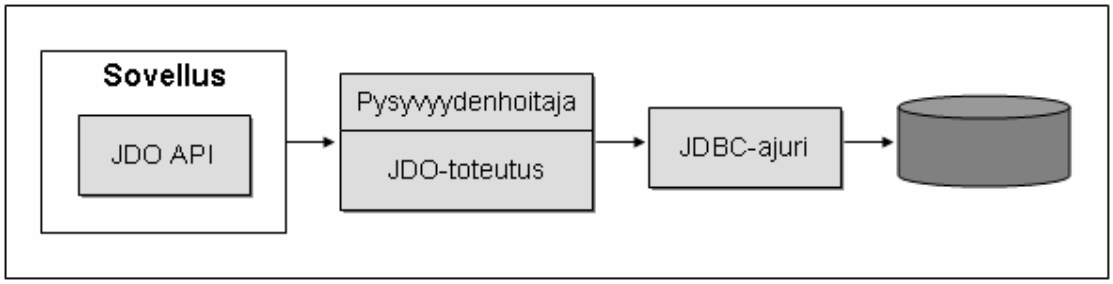
sena tiedon varastoinnissa (Sun Microsystems, 2003c). JDO-teknologialla sovelluksista väitetään saavan hyvin ylläpidettäviä sekä siirrettäviä.

JDO-teknologian suunnittelussa on Samsonin (2001) mukaan tavoiteltu kolmea seikkaa: tuntumattomuutta, universaaliutta ja Java-ympäristöriippumattomuutta. Tuntumattomuudella tarkoitetaan, että minkä tahansa Java-luokan ilmentymä voidaan tehdä pysyväksi, pysyvän tiedon käsittely vaatii vain sovelluskohtaisen ohjelmointikielen osaamisen (Java) ja että pysyvyyden toteuttaminen ei vaadi tietämystä olion fyysisestä esityksestä tietolähteessä. Universaaliudella tarkoitetaan, että standardi on Java-kielen tavoin ympäristöriippumaton sekä lisäksi tietolähderiippumaton. Tietolähderiippumattomuus voidaan käsittää niin, että sovelluksen kannalta ei ole merkitystä käytetäänkö tietolähteenä binaaritiedostoa, relaatiotietokantaa, Excel-taulukkoa vai tietoverkkoa. Java-ympäristöriippumattomuudella tarkoitetaan, että JDO-teknologiaa voidaan käyttää niin J2ME-, J2SE- kuin J2EE -sovelluksissa (Samson, 2001).

6.2 JDO-arkkitehtuuri

Arkkitehtonisesti JDO-tekniikka voidaan jakaa JDO-sovellusliittymää käyttävään sovellukseen, JDO-toteutukseen ja tietolähteeseen.

Varsinaisen olioiden pysyvyyden suorittaa *JDO-toteutus* (JDO Implementation). Se on tietolähteen toimittajan tai kolmannen osapuolen tuottama JDO-spesifikaation määrittelemien rajapintojen ja luokkien toteutus, joka vastaa sovelluksen ja tietolähteen välisestä vuorovaikutuksesta. JDO-toteutusta voi verrata esimerkiksi JDBC-teknologian JDBC-ajuriin. Ohjelmia voidaan kirjoittaa JDO API:n mukaisesti myös ilman JDO-toteutusta, mutta niitä ei voi suorittaa ilman sitä. Sun Microsystems on tehnyt yksinkertaisen binaaritiedostopohjaisen esimerkkitoteutuksen (JDORI), joka osoittaa, kuinka toimittajien toteutuksien tulisi toimia (Sun Microsystems, 2003a). JDO-toteutus voi myös käyttää apunaan jotain olemassaolevaa rajapintaa lisäten sen päälle yhden kerroksen. Kuvassa 6.1 on esimerkki JDO-toteutuksen sijoittautumisesta sovelluksen ja tietolähteen JDBC-ajurin väliin.



Kuva 6.1: Sovelluksen, JDO-toteutuksen, JDBC-ajurin ja tietolähteen välinen yhteys.

Valtaosa JDO-toteutuksista käyttää relaatiotietokantoja tietolähteinä, mutta oliotietokantoja ja tiedostorajapintaa hyödyntäviä on myös tarjolla (Sun Microsystems, 2003a). Esimerkkeinä toteutuksista voisin mainita LIBeLIS LiDO Community Edition, Kodo JDO ja JPOX. JDO-sovellusliittymää käyttävä sovellus voi vaihtaa JDO-toteutuksta toiseen muuttamatta sovelluksen lähdekoodeja (Monday & Edwards, 2002; Sperko, 2003). Asiaa voi tosin vaikeuttaa toteutusriippuvaisten ominaisuuksien käyttö.

Karkeajakoisesti katsottuna JDO-sovellusliittymää käyttävät ohjelmat koostuvat seuraavista komponenteista: itse sovellus, liiketoimintaoliot, pysyvyydenhoitaja, tietolähde ja metadata liiketoimintaluokista (Monday & Edwards, 2002). Sovellus käyttää JDO-sovellusliittymää liiketoimintaolioiden käsittelyyn. Liiketoimintaolioita voi olla väliaikaisia, siis ei-pysyviä sekä pysyvyydenhoitajan käsittelemiä pysyviä olioita. *Pysyvyydenhoitaja* on osa JDO-toteutusta, ja sitä käytetään `PersistenceManager`-rajapinnan kautta olioiden viemiseen tietolähteeseen sekä niiden noutamiseen sieltä. Tietolähteen tehtävä on säilöä pysyvät liiketoimintaoliot. Luokan *metadata* on XML-muotoinen kuvaus sovelluksen pysyvien luokkien attribuuteista, luokkien mahdollisista yhteyksistä ja JDO-toteutukseen liittyvistä yksityiskohdista.

6.3 Pysyvyyden toteuttaminen

Seuraavaksi tarkastellaan, kuinka sovelluksen käyttämät oliot voidaan tallentaa tietolähteeseen ja miten ne voidaan myöhemmin lukea sieltä sovelluksen käyttöön. Aluksi kuvataan prosessia yleisellä tasolla ja tämän jälkeen esitellään tallentaminen sekä palauttaminen lähdekooditasolla.

Sovelluksen liiketoimintaluokat suunnitellaan ja kirjoitetaan normaalisti ilman, että niihin tarvitsee lisätä mitään erityistä toiminnallisuutta JDO:n takia. Periaatteessa luokat voidaan kirjoittaa tietämättä niiden ilmentymien tietojen päätyvän esimerkiksi oliotietokantaan. JDO-teknologialla voidaan tehdä pysyväksi minkä tahansa luokan ilmentymä pois luettuina muutamat poikkeukset. JDO-toteutuksien ei tarvitse pystyä tiettyjen ajonaikaisten järjestelmäriippuvaisten olioiden pysyvyyden toteuttamiseen. Tällaisia ovat muun muassa luokkien `java.net.Socket` ja `java.lang.Thread` oliot sekä niistä perittyjen luokkien oliot (Sperko, 2003). Luokkien kirjoituksen jälkeen Java-kieliset lähdekoodit käännetään *tavukoodisiksi* (byte code) tiedostoiksi (.class) käyttäen mitä tahansa standardia Java-kääntäjää (Monday & Edwards, 2002).

Luokkien pysyvät tiedot määritellään erillisessä XML-tiedostossa (Sun Microsystems, 2003b). Tämän tiedoston tarkoituksena on kertoa JDO-muokkaimelle ja JDO-toteutukselle, mitkä sovelluksen luokat ovat pysyviä, mihin pakkaukseen ne kuuluvat ja mitä erityistä huomioitavaa olioiden tallentamisessa on. Yksinkertaisimmissa tapauksissa XML-kuvaus sisältää vain pysyvien liiketoimintaluokkien ja pakkausten nimet. Monimutkaisimmissa kuvataan tarkasti kaikki luokkien attribuutit ja JDO-toteutukseen liittyvät tiedot. Tarkat säännöt tälle kuvaukselle voidaan löytää JDO-spesifikaatiosta (Sun Microsystems, 2003b). Esimerkki kuvan 3.1 `LevyPapu`-luokkaa vastaavasta XML-kuvauksesta esitetään kuvassa 6.2.

```
<?xml version="1.0"?>
<! DOCTYPE jdo PUBLIC "-//Sun Microsystems, Inc.//DTD Java Data Objects
Metadata 1.0//EN" "http://java.sun.com/dtd/jdo_1_0.dtd">
<jdo>
  <package name="jop">
    <class name="LevyPapu">
    </class>
  </package>
</jdo>
```

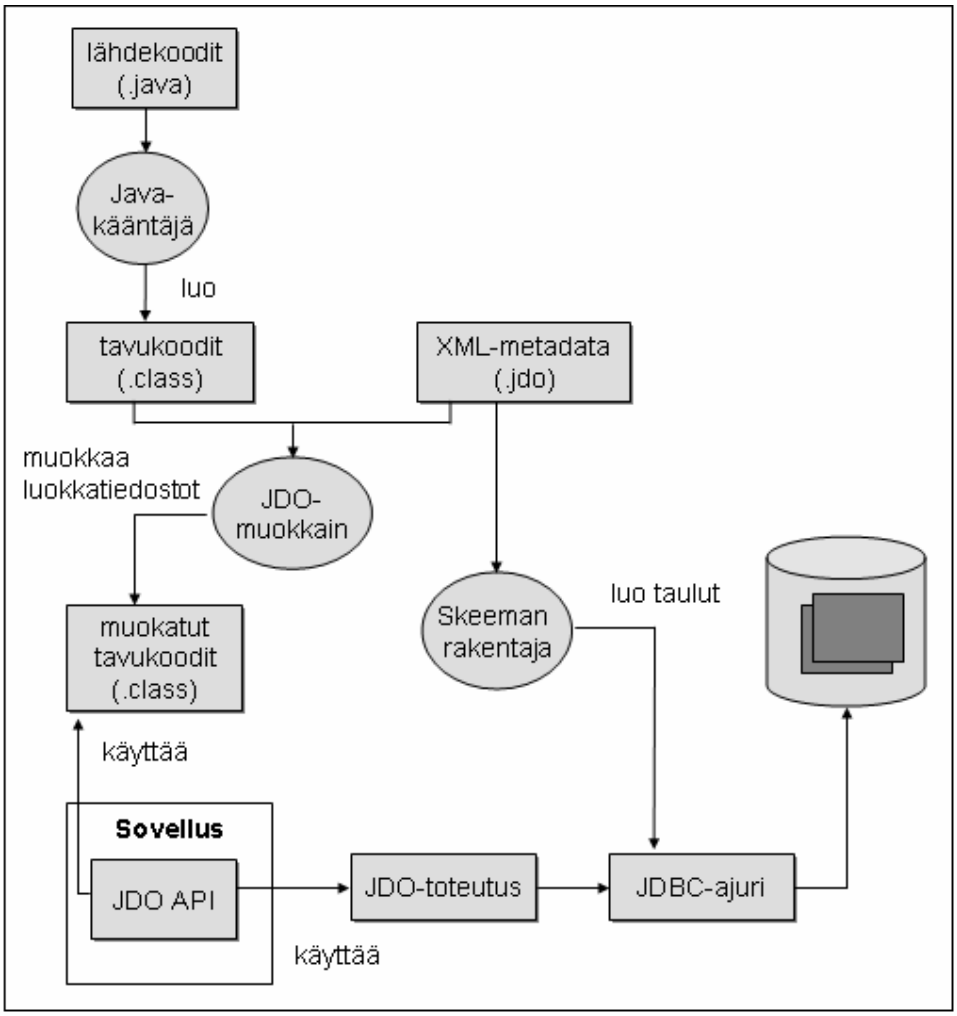
Kuva 6.2: `LevyPapu`-luokan metadata `LevyPapu.jdo`.

Kuvan 6.2 metadata-kuvauksessa määritellään, että pysyvä luokka `LevyPapu` kuuluu pakkaukseen `jop` ja ettei luokan tallennuksessa tarvitse huomioida mitään erikoistekijöitä, vaan luokan kaikki attribuutit voidaan tehdä pysyviksi sellaisenaan.

Metadata-tiedosto annetaan syötteeksi JDO-toimittajan tarjoamalle *JDO-muokkaimelle* (JDO class enchancer). JDO-muokkain käsittelee pysyvien luokkien tiedostoja siten, että ne toteuttavat rajapinnan `PersistenceCapable`. Näin niiden ilmentymät ovat JDO-toteutuksen hallittavissa sovelluksen suorituksen aikana. JDO-muokkaimia on kahdenlaisia: lähdekoodimuokkaimia ja tavukoodimuokkaimia (Sperko, 2003; Monday & Edwards, 2002). Ensimmäinen tekee muutoksia luokan lähdekoodiin ja periaatteessa luokan kirjoittaja voisi itsekin kirjoittaa luokan tähän muotoon ilman lähdekoodimuokkainta. Jälkimmäinen tekee tarvittavat muutokset binaaritasolla käännettyihin luokkatiedostoihin ja näin luokan lähdekoodi säilyy muuttumattomana. Jälkimmäinen tapa on JDO-toteuttajien keskuudessa suositumpi (Monday & Edwards, 2002). Riippuen JDO-muokkaimesta se voi samalla tuottaa XML-kuvauksen perusteella kantakaavan eli tietokannan skeeman (database schema) ja taulujen luontilausekkeet.

Liiketoimintaluokkien muokkauksen jälkeen voidaan kirjoittaa JDO API:a käyttävä sovellus, joka tallentaa, noutaa ja hakee muokattujen luokkien ilmentymiä. JDO-sovellusliittymän käyttö ei vaadi toimiakseen erityisympäristöä, joten sovellus voidaan suorittaa normaalia Java-virtuaalikonetta käyttäen. Sovelluksen ajon aikana JDO-toteutus huolehtii pysyvyyteen liittyvistä yksityiskohdista. Eräs huolehdittava yksityiskohta muun muassa on olioiden identiteetti: JDO-toteutus lisää pysyville olioille yksilöijän, joilla se voi yhdistää sovelluksen käyttämät oliot tietolähteessä sijaitseviin vastaaviin tietoihin (Sun Microsystems, 2003b).

Kuvassa 6.3 selvennetään edellä kuvattua prosessia, jolla liiketoimintaolioista saadaan pysyviä. Kuva 6.3 perustuu kuvan 6.1 tavoin olettamukselle, että JDO-toteutus käyttää toiminnassaan tietokannan toimittajan tarjoamaa JDBC-ajuria. Kuvassa on esitetty myös tietokantataulujen generointi XML-kuvauksen perusteella, mutta riippuen tallennusmediasta tämä sivupolku voi olla tarpeeton. Lisäksi eräät JDO-toteutukset voivat tarpeen vaatiessa luoda tietokantataulut sovelluksen ajon aikana, jolloin ennakkogenerointia ei tarvitse suorittaa.



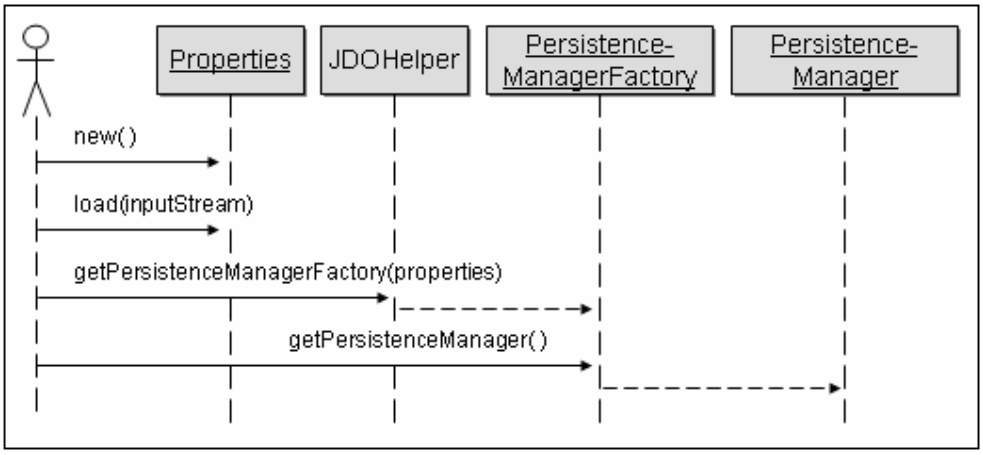
Kuva 6.3: Olioiden pysyvyyden toteuttaminen JDO-tekniikalla (Monday & Edwards, 2002).

JDO-tekniikan käyttöprosessi välivaiheineen ei ole samanlainen kuin JAXB-tekniikan. Pääerona on, että JAXB-tekniikassa pysyvien luokkien lähdekoodin generoi sidontakääntäjä annetun XML-skeeman perusteella, kun taas JDO-tekniikassa ohjelmoija kirjoittaa luokkien lähdekoodin ja JDO-muokkain tekee tarvittavat muutokset lähdekoodiin tai käännettyyn binaaritiedostoon annetun XML-metadatan perusteella.

6.3.1 Yhteyden muodostaminen

Edellä kuvattiin prosessi, jolla liiketoimintaluokista saadaan pysyviä luokkia. Seuraavaksi esitetään tarkemmin, kuinka näitä luokkia voidaan käyttää JDO-sovellusliittymää käyttävissä sovelluksissa.

JDO-toteutus huolehtii sovelluksen ja tietolähteen välisestä yhteydestä. Yhteyden yksityiskohdista vastuussa oleva `PersistenceManagerFactory`-olio saadaan luokan `JDOHelper` staattisella metodilla `getPersistenceManagerFactory`. Tälle metodille annetaan parametrina luokan `Properties` ilmentymä, jossa määritellään yhteyteen liittyvät yksityiskohdat, kuten esimerkiksi käytettävän JDO-toteutuksen nimi, tietolähde, mahdollinen tietolähteen käyttäjätunnus ja salasana. Nämä ominaisuudet voidaan määritellä erillisessä tekstitiedostossa, jonka perusteella tarvittava `Properties`-olio luodaan. Tällä tavoin mitään tarkkoja tietolähteeseen liittyviä tietoja ei tarvitse kirjoittaa lähdekoodiin ja esimerkiksi JDO-toteutuksen vaihto onnistuu tekstitiedostoa muuttamalla. Näiden vaiheiden jälkeen sovellus voi pyytää `PersistenceManagerFactory`-oliolta `PeristenceManager`-oliota, jonka kautta pysyvien olioiden käsittely tehdään. Edellä kuvatut vaiheet on esitetty kuvassa 6.4. Kuvassa esitellään samalla sovelluksen kirjoittajan kannalta tärkeimmät JDO API:n rajapinnat.



Kuva 6.4: Yhteyden luominen tietolähteeseen (Sperko, 2003).

Kuvan 6.4 toimenpiteet on esitetty lähdekooditasolla kuvassa 6.5. Sovellukseen voidaan kirjoittaa alustusoperaatio, joka lataa `PersistenceManagerFactory`-olion sekä operaatio, jolla muut luokan operaatiot saavat tarvittaessa uuden yhteyden pysyvyydenhoitajaan. Kuvassa 6.5 on ohjelmakoodilistaus tällaisista operaatioista.

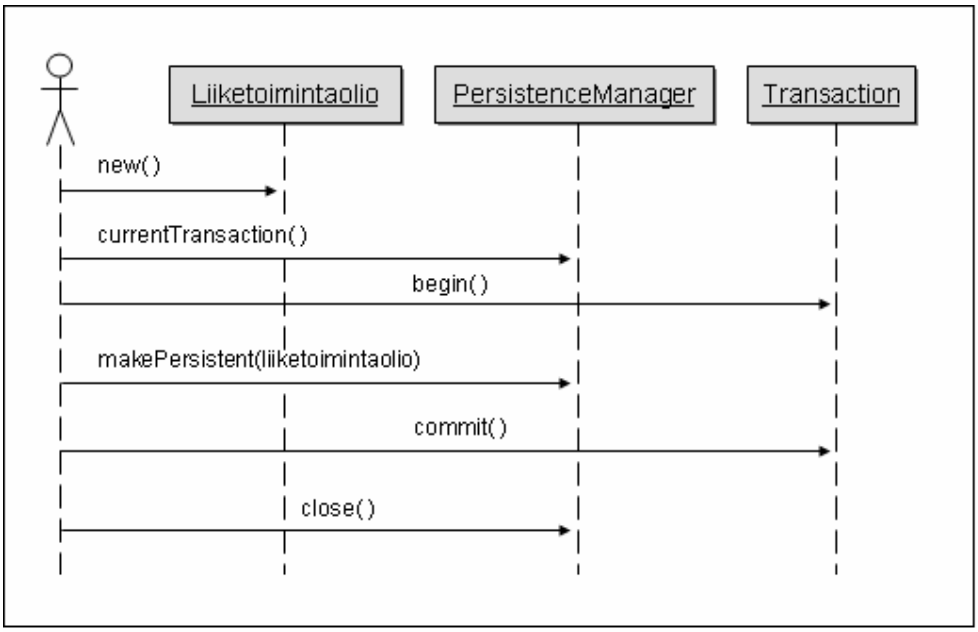
```
/* Lataa ominaisuustiedosto ja luo PersistenceManagerFactory-olio */
private boolean alusta() {

  Properties ominaisuudet = new Properties();
  try {
    /* lataa ominaisuustiedosto (tekstitiedosto) muistiin */
    InputStream in =
      ClassLoader.getSystemResourceAsStream("jdo.properties");
    ominaisuudet.load(in);

    /* muodosta yhteydenhallitsija, sijoita se olion ominaisuudeksi */
    this.persistenceManagerFactory =
      JDOHelper.getPersistenceManagerFactory(ominaisuudet);

  } catch (Exception e) {
      e.printStackTrace();
      this.persistenceManagerFactory = null;
      return false;
  }

  return true;
}

/* Palauta pysyvyydenhoitaja –olio (PersistenceManager) */
public PersistenceManager haePysyvyydenHoitaja() {
  return this.persistenceManagerFactory.getPersistenceManager();
}
```

Kuva 6.5: Esimerkkioperaatiot, joilla muodostetaan yhteys tietolähteeseen ja pyydetään pysyvyydenhoitajaa (Rouvinen, 2004).

Kuvan 6.5 operaatioista `alusta`-metodia kutsuttaisiin vain kerran sovelluksen käynnistyessä ja `haePysyvyydenHoitaja`-metodia aina, kun sovellus haluaisi tehdä erillisiä operaatioita pysyville olioille. `PersistenceManager`- ja `PersistenceManagerFactory` -olioiden varaamat resurssit tulee vapauttaa niiden `close`-operaatiolla. On kuitenkin otettava huomioon, että näiden luokkien välinen yhteys on yhden-suhde-yhteen. Toisin sanoen kerran suljettua pysyvyydenhoitajaa ei saada avattua kutsumalla uudestaan `getPersistenceManager`-operaatiota. Sulkeminen voidaan siten tehdä esimerkiksi luokan `finalize()`-operaatiossa.

6.3.2 Olioiden tallentaminen

JDO-tekniikalla olioiden tallentaminen on esivalmistelujen jälkeen varsin yksinkertaista. Sovellus käyttää pysyvyydenhoitajan tarjoamia metodeita olion viemiseksi tietolähteeseen ja JDO-toteutus on vastuussa liiketoimintaolion pysyvyyden toteuttamisen yksityiskohdista, kuten esimerkiksi olion pilkkomisesta relaatiotietokantaan. Kuvassa 6.6 on esitetty liiketoimintaolion tallentaminen tietolähteeseen sekvenssikaavion avulla.



Kuva 6.6: Liiketoimintaolion tallentaminen JDO-tekniikalla (Sperko, 2003).

Kuvassa 6.7 esitetään yllä oleva tallentamisprosessi lähdekooditasolla. Varsinainen työ olion pysyväksi saamiseksi sisältää seuraavat vaiheet:

1. Luo tallennettava olio.
2. Ota yhteys pysyvyydenhoitajaan.
3. Pyydä pysyvyydenhoitajalta viittaus tapahtumankäsittelijään ja aloita uusi tapahtuma.
4. Tee olio pysyväksi pysyvyydenhoitajan metodilla `makePersistent`.
5. Hyväksy tapahtuma.

Pysyvyydenhoitajan sulkeminen ei ole tarpeellista, mikäli kyseisellä pysyvyydenhoitajalla käsitellään vielä muita pysyviä olioita. Mikäli suorituksessa syntyy poikkeus, on tapahtuma peruutettava tai se jää avoimeksi. Avoinna oleva tapahtuma estää uuden tapahtuman aloittamisen. Poikkeuksen johdosta suoritettava tapahtuman peruuttaminen voidaan tehdä kuvan 6.7 tavoin operaation `finally{}`-lohkossa.

```
public boolean tallennaLevy(LevyPapu levy) {

  PersistenceManager pm       = null;
  Object             oid      = null;
  boolean            palautus = true;

  if ((null == levy) || ("".compareTo(levy.getNimi()) == 0)) {
    return false;               /* ei levyä tai levyn nimi on tyhjä */
  }

  /* tarkasta onko levy jo tallennettu (sovelluskohtainen tarkistus) */
  oid = haeOID(disc);
  if (null != oid) {
      return true;
  }

  try {
    pm = haePysyvyydenHoitaja();            /* ks. kuva 6.5 */
    pm.currentTransaction().begin();        /* aloita uusi tapahtuma */
    pm.makePersistent(levy);                /* tee olio pysyväksi */
    pm.currentTransaction().commit();       /* hyväksy tapahtuma */
  } catch (Exception e) {
      e.printStackTrace();
      palautus = false;                     /* epäonnistunut suoritus */
  } finally {          /* mikäli tapahtuma on sulkematta, peruuta se */
      if ((null != pm) && (null != pm.currentTransaction())
        && pm.currentTransaction().isActive()) {
        pm.currentTransaction().rollback();
    }
  }

  return palautus;
}
```

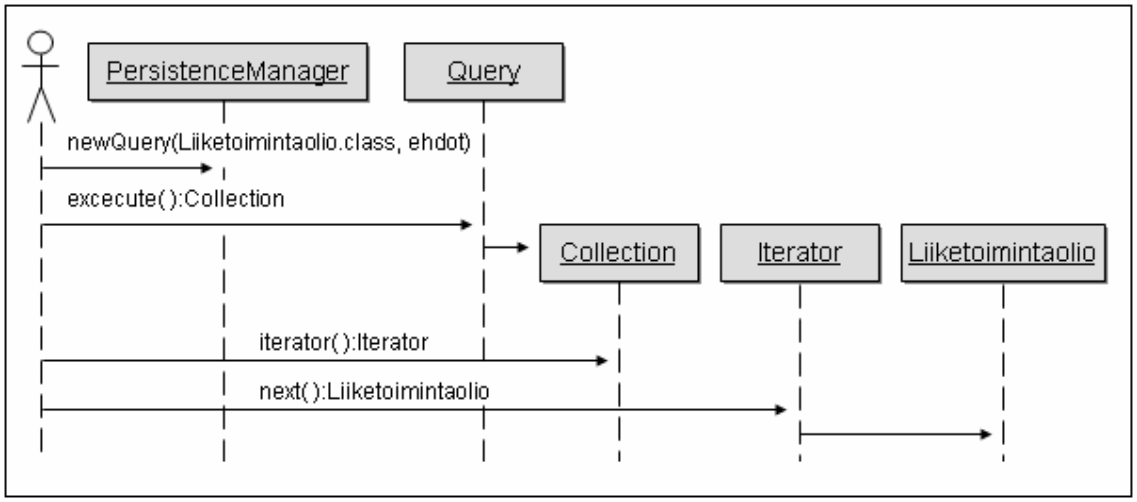
Kuva 6.7: `LevyPapu`-liiketoimintaolion tallentaminen tietolähteeseen (Rouvinen, 2004).

Kuvan 6.7 operaatiossa tarkistetaan duplikaatin tallennusyritys. Toisaalta tämä tarkistus on tarpeeton, sillä operaatio `makePersistent(Object pc)` nostattaa poikkeuksen, mikäli kyseinen olio on jo pysyvä. Tarkistus tehdään olioiden identiteettien perusteella. Tekemällä tarkistus itse on varmistettu, ettei tietolähteeseen viedä samansisältöisiä olioita ja siten syntyneet poikkeukset ovat aina todellisia käsittelyvirheitä.

Kun olio on kerran tehty pysyväksi metodilla `makePersistence()`, voidaan sen tietoja muokata avoinna olevissa tapahtumissa ja muutokset reflektoituvat automaattisesti tietolähteeseen. Tietojen päivittämisestä on vastuussa pysyvyydenhoitaja, eli JDO-toteutus. Pysyvä olio saadaan muokattavaksi tietolähteestä esimerkiksi kyselyllä kohdan 6.3.3 mukaisesti.

6.3.3 Olioiden palauttaminen ja kyselyt

JDO-teknologian avulla tietolähteeseen voidaan suorittaa kyselyitä. Käytettävä kyselykieli on JDO Query Language (JDOQL) ja kyselyitä suoritetaan `Query`-rajapinnan kautta (Monday & Edwards, 2002). Tämän rajapinnan toteuttava olio saadaan pysyvyydenhoitajalta siis `PersistenceManager`-oliolta. Kyselyillä voidaan hakea haluttujen luokkien ilmentymiä ja kyselyihin voidaan asettaa rajoitteita eli ehtoja. JDOQL-kyselyt ovat hyvin monipuolisia, mutta peruskäytön oppiminen on helppoa. Yleiskuvaus kyselyyn käytettävien luokkien välisistä yhteyksistä esitetään kuvassa 6.8.



Kuva 6.8: Kyselyiden suorittaminen (Sperko, 2003).

Kuten kuvasta 6.8 ilmenee, pysyvät oliot palautetaan sovelluksen käyttöön tietolähteestä suorittamalla JDOQL-kysely. Kyselyn tyyppi asetetaan sen luontihetkellä (kuvassa 6.8 metodilla `newQuery`). Tyypillä tarkoitetaan, minkä luokan ilmentymiä kyselyllä on tarkoitus etsiä ja mitä ehtoja ilmentymien valitsemiselle asetetaan. Kyselyn suorittamisen jälkeen kriteereitä vastanneesta tulosjoukosta poimitaan sovelluksen käyttöön halutut oliot. Tämä prosessi esitetään lähdekooditasolla kuvassa 6.9.

```
public LevyPapu etsiLevy(String nimi) {

  PersistenceManager pm   = null;
  LevyPapu           levy = null;

  if ((null == nimi) || ("".compareTo(nimi) == 0)) {
    return null;
  }

  try {
    pm = haePysyvyydenHoitaja();                      /* ks. kuva 6.5 */
    pm.currentTransaction().begin();
    /* luo uusi kysely: hae luokan LevyPapu oliot, joiden nimi täsmää
       parametrina annettuun merkkijonoon */
    Query query = pm.newQuery(jop.LevyPapu.class,
      "nimi == \"" + nimi + "\"");
    Collection tulosjoukko = (Collection) query.execute();

    Iterator iteraattori = tulosjoukko.iterator();
    if (iteraattori.hasNext()) {
      levy = (LevyPapu) iteraattori.next();
      pm.makeTransient(levy);     /* irrota olio pysyvyydenhoitajasta */
    }

    pm.currentTransaction().commit();
  } catch (Exception e) {
      e.printStackTrace();
      levy = null;
  } finally {
      if ((null != pm) && (null != pm.currentTransaction())
          && pm.currentTransaction().isActive()) {
          pm.currentTransaction().rollback();
      }
  }

  return levy;
}
```

Kuva 6.9: `LevyPapu`-olion etsiminen tietolähteestä JDOQL-kyselyllä (Rouvinen, 2004).

Kuvan 6.9 esimerkkimetodissa kyselyn tulosjoukko sisältää sen `LevyPapu`-luokan ilmentymän, minkä `nimi`-attribuutti täsmää metodille parametrina annettuun merkkijonoon. Mikäli täsmäystä ei ole, tulosjoukko on tyhjä eikä sitä voida iteroida. Kuvassa oleva operaatiokutsu `makeTransient(Object pc)` irrottaa pysyvän olion pysyvyydenhoitajan alaisuudesta. Tämä tarkoittaa sitä, että pysyvyydenhoitaja poistaa olion identiteetin eikä pysyvyydenhoitaja tämän jälkeen seuraa olion vaiheita. Näin ollen luokan ilmentymä, joka saa `etsiLevy`-operaation palauttaman olion, voi käsitellä oliota haluamallaan tavalla ilman, että muutokset heijastuisivat suoraan tietolähteeseen. Riippuen käytetystä JDO-toteutuksesta tämä irrotus voi olla myös välttämätön, että oliota voidaan ylipäätään käsitellä tapahtuman ulkopuolella (tapahtuma suljetaan operaation lopussa). Irrotusta edellyttää esimerkiksi JDO-toteutus JPOX.

6.3.4 Pysyvän olion poistaminen

Pysyvän olion poistaminen tietolähteestä suoritetaan, kuten olion sinne lisääminen. Prosessi on muuten yhteneväinen, mutta lisäämisoperaation sijasta sovelluksen täytyy suorittaa pysyvyydenhoitajan tarjoama poistamisoperaatio: `deletePersistent(Object pc)`. Tällä metodilla pysyvyydenhoitaja poistaa pysyvän olion tietolähteestä sekä irrottaa käsiteltävän olion alaisuudestaan.

```
public boolean poistaLevy(LevyPapu levy) {

  PersistenceManager pm              = null;
  LevyPapu           poistettavaLevy = null;
  Object             oid             = null;
  boolean            palautus        = true;

  if ((null == levy) || ("".compareTo(levy.getNimi()) == 0)) {
    return false;
  }

  /* etsi liiketoimintaoliota vastaava pysyvä olio tietolähteestä,
     palauta pysyvän olion JDO-identiteetti. */
  oid = haeOID(levy);
  if (null == oid) {
    return true;      /* ei vastaavaa oliota, olio on jo poistettu */
  }

  try {
    pm = haePysyvyydenHoitaja();                    /* ks. kuva 6.5 */
    pm.currentTransaction().begin();
    /* Pyydä pysyvä olio selvitetyn olion identiteetin perusteella */
    poistettavaLevy = (LevyPapu) pm.getObjectById(oid, true);
    pm.deletePersistent(poistettavaLevy);
    pm.currentTransaction().commit();
  } catch (Exception e) {
      e.printStackTrace();
      palautus = false;
  } finally {
    if ((null != pm) && (null != pm.currentTransaction())
        && pm.currentTransaction().isActive()) {
      pm.currentTransaction().rollback();
    }
  }

  return palautus;
}
```

Kuva 6.10: Pysyvän olion poistaminen tietolähteestä (Rouvinen, 2004).

Prosessi liiketoimintaolion poistamiseen on esitetty lähdekooditasolla kuvassa 6.10. Lisätyötä olion poistamisessa tuo halutun olion noutaminen poistettavaksi. Haku suoritetaan metodilla `haeOID`. Tämä joudutaan tekemään, koska operaation parametrinaan saama liiketoimintaolio

voi olla irrotettu pysyvyydenhoitajasta, jolloin oliolla ei ole JDO-identiteettiä eikä sitä täten voida poistaa. Jos käytettävissä olisi irrottamaton olio, se voitaisiin tietenkin poistaa suoraan ilman olion identiteetin selvittämistä.

Mikäli pysyvyydenhoitajaa pyydetään poistamaan olio, jolla ei ole tunnistettavaa JDO-identiteettiä, syntyy poikkeus `JDOException`. Tämä `deletePersistent`-operaation nostama-poikkeus ilmaisee, että pysyvyydenhoitaja ei voi poistaa parametrina annettua oliota. Useimmiten tämä tarkoittaa, ettei kyseistä oliota löydetä tietolähteestä.

6.4 Soveltuvuusalue

Dataoliot-tekniikka tarjoaa oliokeskeisen ja lähes tuntumattoman tavan käsitellä pysyviä olioita. Teknologiaa käyttävät sovellukset ovat hyvin siirrettäviä ja sen käyttö sopii erilaisiin arkkitehtuureihin. Yksityiskohdat olioiden tietojen käsittelyssä on piilotettu sovelluksen tai komponentin kehittäjältä. Tiedon viennit, palautukset, kyselyt ja muokkaamiset ovat melko yksinkertaisia. Etuna voidaan nähdä myös kyselyissä käytettävä JDOQL-kyselykieli, koska tämän syntaksi perustuu Javaan. Pysyvyyteen tarvittava koodin määrä on suhteellisen pientä verrattuna muihin vastaaviin tekniikoihin (Brown, 2002).

Biggsin (2001) mukaan JDO-tekniikka vähentää selvästi ohjelmistotuotantoprosessin kustannuksia. Se yksinkertaistaa ohjelmakoodia verrattuna muun muassa JDBC-teknologiaan ja näin lyhentää sovelluksen kehitysaikaa sekä helpottaa ylläpitoa. JDO antaa mahdollisuuden ohjelmoijille keskittyä Java-kieleen eikä tietolähteeseen liittyviin yksityiskohtiin. Tämä mahdollistaa virheettömämmän ohjelmakoodin sekä vähentää organisaation koulutuskustannuksia. Lisäksi toisin kuin JDBC-teknologiassa, ei JDO-toteutuksissa ole nähdäkseni tarpeellista tai edes järkevää rakentaa erillistä kerrosta olioiden pysyvyyden hallintaan. JDO-operaatiot tulisi sulauttaa jo osaksi sovelluslogiikkakerrosta ja näin luoda yksinkertaisempia toteutuksia.

JDO-tekniikka sopii asiakas-palvelin- sekä RMI-arkkitehtuurien mukaisiin hallinnoimattomiin ympäristöihin, missä sovelluspalvelinta ei ole käytössä. Sun Microsystemsin (2003c) mukaan JDO voidaan myös integroida palvelimille suunnatun EJB-komponenttiteknologian kanssa. Näin ollen JDO-teknologiaa ehdotetaan rinnasteiseksi tai jopa korvaajaksi olioiden pysyvyyden toteuttavalle Container Managed Persistence –teknologialle (CMP). Tällöin

JDO:n etuja CMP-teknologiaan verrattuna ovat muun muassa kevyempi toteutus, helpompi testattavuus sekä Java-tyyliset haut SQL:n sijaan.

JDO-tekniikkaa voidaan pitää vaihtoehtona JDBC-tekniikalle, mutta se ei korvaa sitä kokonaan. Mikäli tarvitaan tietokantakohtaista koodia, on JDBC parempi valinta (Biggs, 2001). Samoin on, mikäli JDO-toteutuksen hoitama olioiden ja relaation välinen *kuvaus* (mapping) on suorituskyvyllisesti liian raskasta. Lisäksi ohjelmoijan tulee huolehtia, että käytetään sellaisia JDO-toteutuksen asetuksia, jotka takaavat optimaalisen suorituskyvyn kyseiseen systeemiin (Monday & Edwards, 2002).

Hyvistä ominaisuuksistaan huolimatta JDO ei ole täysin ortogonaalisesti persistenssi tekniikka, koska sovelluksen ohjelmoija on tietoinen olioiden pysyvyydestä ja osittain vastuussa sen toteuttamisesta. Lisäksi ohjelmoija joutuu vastaamaan pysyvyyden tunnistamisen onnistumisesta määrittelemällä pysyvien liiketoimintaluokkien metadatan.

7 Pysyvyyden toteutustekniikoiden vertailu

Tähän mennessä on esitetty kirjallisuuden sekä esimerkkisovelluksen lähdekoodien avulla neljä erilaista Javan teknologiaa, joilla voidaan toteuttaa olioiden pysyvyys. Tässä luvussa pyritään osoittamaan sovelluskehittäjän kannalta olennaisimmat teknologioiden väliset eroavaisuudet. Tekniikoiden ominaisuuksia tarkastellaan kohdassa 2.7 esitettyjen relevanttien muuttujien suhteen sekä esimerkkisovelluksen tekemisestä saatujen kokemusten perusteella.

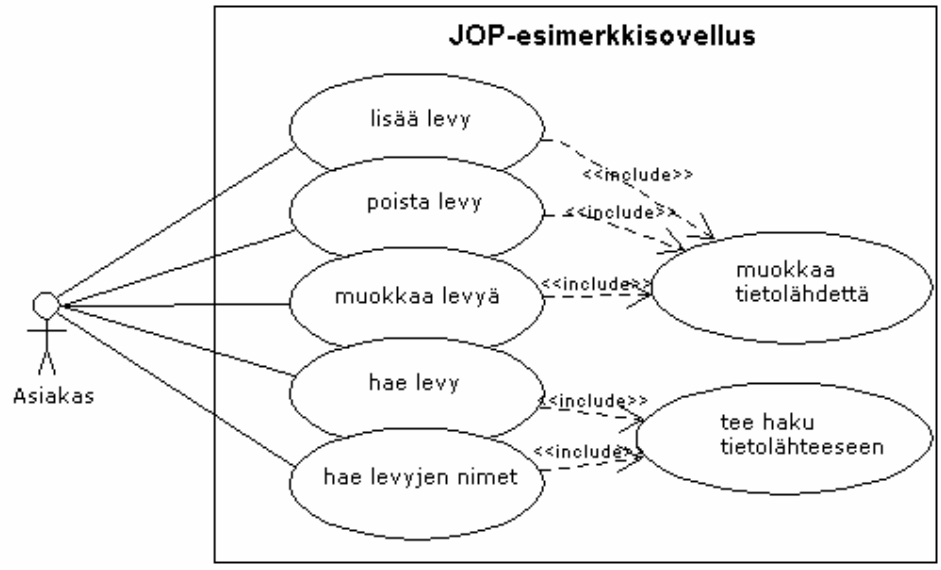
7.1 Esimerkkisovellus

Ennen kuin keskitytään tekniikoiden välisiin eroihin ja niistä saatuihin kokemuksiin, kuvataan tehdyn esimerkkisovelluksen toiminta. Tämä tehdään, jotta lukija ymmärtäisi paremmin, kuinka tekniikoita on verrattu toisiinsa.

Toteutettu esimerkkisovellus (Rouvinen, 2004) on asiakas-palvelin -ohjelmisto, joka demonstroi neljää Javan teknologiaa, joilla voidaan tallentaa olioita tietolähteeseen. Tietolähteinä käytetään binaaritiedostoa, XML-tiedostoa ja relaatiotietokantaa. Liiketoimintaoliot tallennetaan käyttäen JOS-, JAXB-, JDBC- ja JDO -tekniikoita. Sovellus on toiminnallisesti levyluettelo-ohjelma ja sen käsittelemät liiketoimintaoliot ovat kuvitteellisia datalevyjä (levypapuja), kuten CD- tai DVD -levyjä. Jokaisella levyllä on nimi (teksti), sisältökuvaus (teksti), käytetty tila (desimaaliluku) ja tieto siitä voidaanko levylle vielä kirjoittaa lisää tietoa (totuusarvo). Sovelluksella voidaan lisätä, poistaa, muokata ja hakea tietolähteissä olevien datalevyjen tietoja. Esimerkkisovelluksen käyttötapaukset on esitetty kuvassa 7.1.

Java ja olioiden pysyvyys -esimerkkisovelluksessa on neljä luokkaa, jotka kukin toteuttavat eri teknologialla kuvassa 7.1 esitetyt levyluettelon toiminnalliset vaatimukset. Nämä neljä luokkaa muodostavat kukin itsenäisen demonstraation liiketoimintaolioiden tallentamisesta ja tallennetun tiedon käsittelystä. JOS-levyluettelo käyttää tallennusmediana binaaritiedostoa, JAXB-levyluettelo käyttää XML-tiedostoa, JDBC- ja JDO[5] -levyluettelot käyttävät MySQL-tietokantaan.

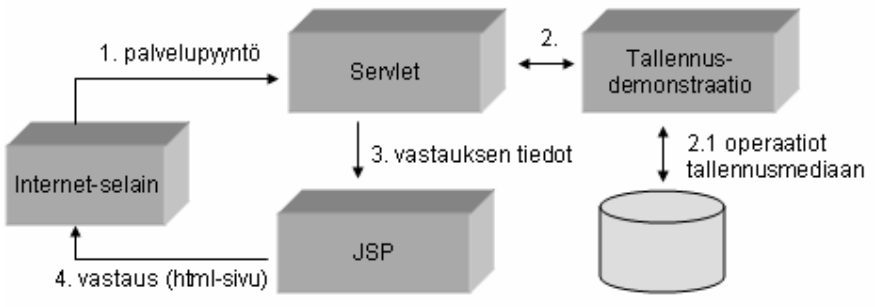
[5] Esimerkkisovelluksessa JDO-toteutuksena on käytetty JPOX-toteutusta (Java Persistent Objects JDO, 2004).



Kuva 7.1: Esimerkkisovelluksen käyttötapaukset (Rouvinen, 2004).

7.1.1 Esimerkkisovelluksen toteutus

Palvelinsovellus toteutettiin Tomcat-sovelluspalvelimelle käyttäen Servlet- ja JSP -tekniikoita. Asiakas käyttää sovellusta Internet-selaimen avulla. Asiakkaan sovellukselle lähettämät komennot käsittelee palvelinsovelma eli servletti. Palvelinsovelman toimintaa ohjaavat asiakkaan komentojen mukana saapuvat parametrit. Palvelinsovelma ohjaa komentoja varsinaisen tietolähteen käsittelyn hoitaville luokille ja lähettää niiden palauttamat tulokset JSP-sovelmalle. Arkkitehtonisesti palvelinsovelma toimii liitteen 1 ja kuvan 7.2 mukaisesti vain kontrollerina, tallennusdemonstraatioluokat (4 kpl) hallitsevat varsinaisen ydintoimintalogiikan ja JSP-sovelma vastaa käyttöliittymän muodostamisesta.



Kuva 7.2: Esimerkkisovelluksen arkkitehtuurinen rakenne (Rouvinen, 2004).

Kuvassa 7.2 osoitetaan sovelluksen arkkitehtuuri ja yhden palvelupyynnön käsittely. Palvelupyyntö alkaa, kun asiakas lähettää komennon sovelluksen käyttöliittymästä, esimerkiksi komennon uuden levyn tallentamiseksi. Servletti ottaa palvelupyynnön vastaan ja lukee sen mukana saapuneet parametrit. Parametreina tulevat käyttöliittymän vuorovaikuttimien arvot sekä arvot, jotka kertovat annetun komennon. Näiden parametrien mukaan palvelinsovelma luo uuden levypapu-olion sekä valitsee tallennusdemonstraatioluokan, eli levykokoelman ja sille tehtävän toiminnon. Levykokoelman palauttama vastaus lähetetään edelleen JSP-sovelmalle, joka vastaa käyttöliittymän ja sen näyttämien tietojen lähettämisestä asiakkaan selaimelle.

7.2 Toteutustekniikoiden vertailua

Seuraavaksi tarkastellaan tekniikoiden välisiä eroavaisuuksia seitsemän piirteen perustella. Lisäksi esitellään esimerkkisovelluksen neljän tallennusdemonstraatioluokan tekemisestä saatuja kokemuksia.

7.2.1 Pysyvän olion muunnos tietolähteen edellyttämään muotoon

Ensimmäinen vertailtava piirre on, kuinka tekniikat suoriutuvat liiketoimintaolion saattamisesta tietolähteen käyttämään esitysmuotoon. Toivottavaa on, että olioita tulisi käsitellä kokonaisina tietoyksikköinä ja riippuvuus muiden tekniikoiden osaamisesta tulisi olla mahdollisimman vähäistä.

JOS-tekniikassa Java-ympäristö huolehtii kokonaisten olioiden kirjoittamisesta virtaan sekä niiden lukemisesta sieltä takaisin sovelluksen käyttöön. Ohjelmoijan ei tarvitse suorittaa muunnosta olion ja tietolähteen esitysmuodon välillä, kuten esimerkiksi pilkkoa olion tietoja. Samoin on myös JAXB- ja JDO -tekniikoissa. Toisaalta kahden jälkimmäisen tekniikan käyttö vaatii ylimääristä työtä, jotta käsiteltävistä luokista saadaan pysyviä. JAXB-tekniikassa ohjelmoijan täytyy kirjoittaa XML-skeema ja generoida pysyvät luokat sidontakääntäjällä. Vastaavasti JDO-tekniikassa ohjelmoijan täytyy tuottaa metadata ja ajaa luokkatiedostot JDO-muokkaimen läpi. Näiden toimien suorittamiseen tarvitaan tekniikkaan sidoksissa olevaa erikoisosaamista. Toimenpiteillä saavutetaan kuitenkin tilanne, jossa sovelluksen ohjelmoijan ei tarvitse tietää käytettävän tietolähteen yksityiskohtia eikä tiedon rakennetta ja tietojen käsittely on oliomaista.

JDBC-tekniikan käyttö ei edellytä luokkien lähdekooditiedostojen muokkausta tai kuvausten kirjoittamista, jotta oliosta saataisiin pysyvä. Täten tekniikan käyttöönotto sovelluksissa on suoraviivaista. Valitettavasti JDBC-ratkaisuissa olioita joudutaan käsittelemään primitiividatan tasolla ja näin ohjelmoija on itse vastuussa olion muunnoksesta relaatiotietokannan taulun muotoon. Jotta liiketoimintaoliot saadaan pysyväksi, on ohjelmoijan tunnettava käytettävän tietolähteen rakenne läpikotaisin. Samoin ohjelmoijien täytyy muodostaa palautettavat oliot itse SQL-kyselyiden palauttamien primitiivitietojen perusteella.

Esimerkkisovelluksen tapauksessa käsiteltäviä pysyviä liiketoimintaluokkia oli vain yksi, joten JDBC-ratkaisu oli helppo toteuttaa, vaikkei tietoja käsiteltykään oliomaisesti. Pysyvän luokan kirjoittaja, tietokannan luoja ja pysyvyyden toteutuksen kirjoittaja olivat sama henkilö, joten primitiividatan käsittelyä ei nähty ongelmalliseksi. Tilanne voisi olla toinen, mikäli käsiteltävät pysyvät luokat olisivat isompia ja niitä olisi enemmän. Tällöin tuntumattoman olioiden käsittelyn edut korostuisivat.

7.2.2 Käytettävä kyselytekniikka olioiden palauttamiseksi

Käsitellyistä teknologioista olioiden sarjallistaminen on mielestäni kaikkein kankein kyselyiden ja olioiden palauttamisen suhteen. Tekniikka ei mahdollista yksittäisen olion suorahakua tietolähteestä, vaan ohjelmoijan on ohjelmoitava tietueiden (olioiden, merkkien, lukujen jne.) lukeminen sarjajärjestyksessä tietovirrasta. Lukemisessa täytyy siis tietää tietovirran sisällön järjestys tai tehdä ylimääräistä työtä selvitettäessä luettujen alkioiden tyyppejä. Vastaavasti tietojen päivittäminen ei onnistu yksittäisen virrasta luetun olion tietoja muuttamalla, vaan muutosten jälkeen koko tietovirta on kirjoitettava uudestaan. Hakuja sekä olioiden päivityksiä voidaan yrittää helpottaa ja tehostaa pitämällä esimerkkisovelluksen tavoin koko tietovirran sisältö sopivassa tietorakenteessa sovelluksen suorituksen ajan. Sovelluksen päättyessä koko rakenne voidaan sarjallistaa (kirjoittaa) tietovirtaan yhdellä komennolla. Tosin tällöin sovelluksen kaatuessa voidaan menettää kaikki tietorakenteeseen kohdistuneet muutokset.

XML-sidonnan arkkitehtuuri ei ole kyselyiden suhteen yhtään olioiden sarjallistamista parempi: se ei mahdollista hakujen tekemistä tietolähteen eli XML-dokumentin tietoihin. Sovelluksen täytyy lukea koko dokumentti muistiin ja etsiä halutut tiedot sisältöpuun olioista. Lisäksi ei ole mahdollista suoraan muuttaa yksittäisen olion - dokumentin elementin - tietoja, vaan dokumentti on kirjoitettava aina kokonaan uudestaan. Olioiden sarjallistamista helpompi

se on mielestäni olioiden palauttamisen suhteen, sillä koko oliojoukon lukeminen ja kirjoittaminen takaisin tehdään aina yhdellä komennolla. Tietolähdettä, siis XML-dokumenttia, käyttävän sovelluksen ei tarvitse tietää käsiteltävien tietojen määrää tai rakennetta.

JDBC-teknologian kyselytekniikka perustuu standardoituun ja tietokantaohjelmoijien hyvin tuntemaan SQL-kieleen. Tietolähteeseen voidaan kohdistaa hyvin monipuolisia kyselyitä, joten sovellus voi palauttaa käyttöönsä kaikki tallennetut oliot tai vain pienen, haluttuja kriteereitä vastaavan osajoukon. Kyselyt lähetetään suoraan tietokantaan ja vastuu niiden suorittamisesta on tietokannanhallintajärjestelmällä, ei sovelluksen ympäristöllä. Kyselyiden tuloksia voidaan iteroida, mutta valitettavasti tulosjoukon alkiot eivät ole olioita, vaan oliot tulee luoda tulosjoukon perusteella. Näin ollen luotuihin olioihin tehdyt muutokset täytyy päivittää tietolähteeseen uudella kyselyllä.

JDO-teknologian käyttö mahdollistaa JDBC-teknologian tavoin kyselyt käytettävään tietolähteeseen. Kyselytekniikka on oliolähtöinen ja kyselykielenä käytettävän JDOQL:n sanotaan olevan Java-ohjelmoijille SQL-kieltä helpompi sen Java-tyylisyyden vuoksi (Monday & Edwards, 2002; Samson, 2001; Sun Microsystems, 2003c). JDO-kyselyt voidaan suorittaa standardin mukaan kahdella tavalla (Sun Microsystems, 2003b): JDO-toteutus välittää koko kyselyn tietokannanhallintajärjestelmälle ja luo uudet oliot palautettujen arvojen perusteella tai JDO-toteutus voi tehdä laveamman kyselyn tietokantaan ja suorittaa itse tietojen koostamisen kyselyn mukaisesti. Jälkimmäistä tapaa toteuttavat JDO-toteutukset valitettavasti vaativat suoraa tietokannanhallintajärjestelmän käyttöä enemmän muistia ja ovat luultavasti tehottomampia kuin kyselyihin suunnitellut ja optimoidut tietokannanhallintajärjestelmät. Olioiden palauttamisessa JDO:n etu on kuitenkin se, että esimerkiksi tietyn luokan kaikki ilmentymät saadaan palautettua tietolähteestä sovelluksen ohjelmoijan näkökulmasta hyvin helposti. Lisäksi palautettuja olioita on mahdollista muokata ja käyttää vapaasti.

7.2.3 Suorituskyky ja skaalautuvuus

Esimerkkisovelluksen tallennusdemonstraatioluokkien suorituskykyisyyttä ei testattu eikä luokkia ole vertailtu toisiinsa tämän osalta. Seuraavaksi esitettävät arviot tekniikoista perustuvatkin kirjallisuudessa esitettyihin faktoihin.

Aluksi verrataan olioiden sarjallistamista ja XML-sidonnan arkkitehtuuria toisiinsa. Herickon & al. (2003) suorittamien testien mukaan JOS on noin viisi kertaa JAXB-tekniikkaa nopeampi. Lisäksi olioiden sarjallistamisen luomat binaaritiedostot ovat noin neljä kertaa pienempiä kuin vastaavat XML-tiedostot. Suorituskyvyssä mitattuna JAXB on siis varsin huono. JAXB-sovelluksen suorituskykyä voidaan yrittää parantaa vaihtamalla JAXB API:n käyttämää XML-jäsennintä. Standarditoteutus käyttää SAX:ia ja tarjolla on useita muita vaihtoehtoja. Jäsentimien välisten suorituskykyjen selville saaminen vie valitettavasti aikaa ja resursseja. Mikäli sovelluksen käsittelemät tietomäärät ovat kuitenkin suuria ja sovelluksessa halutaan suorittaa kyselyitä, ei kumpikaan näistä tiedostopohjaisista tekniikoista pärjää skaalautuvuudessa eikä suorituskyvyssä JDBC- ja JDO -tekniikoille.

JDBC-tekniikkaa käyttävien sovelluksien tietolähteenä on useimmiten relaatiotietokanta (Sun Microsystems, 2001). Tietokanta pitää huolen tietojen hallinnasta ja sovellus käyttää tietokannan palveluita JDCB-sovellusliittymän kautta. Näin ollen JDBC-ratkaisujen tehokkuus on riippuvainen kolmesta asiasta: tietokannanhallintajärjestelmän suorituskyvystä, sille tarjotusta JDBC-ajurista ja ohjelmoijan tekemistä kyselyistä. Tällöin sovellukset, jotka voivat pilkkoa pysyvät oliot helposti relaatiomuotoon, hyötyvät merkittävästi tietokantojen suorituskykyisyydestä. Pullonkaulaksi skaalautuvuudessa voi muodostua käytettävä JDBC-ajuri (Sperko, 2003). Ratkaisuksi tähän sovelluskehittäjä voi oikeastaan vain testata erilaisia ajureita erilaisilla kuormilla ja tilanteilla. Samoin sovelluksen tietokantaan lähettämät SQL-kyselyt voivat olla huonosti laadittuja tai epäsopivia käytettävälle ajurille. Suorituskyvyn parantamiseksi ohjelmoija voi yrittää muokata kyselyitä, mutta itse JDBC ei tarjoa tähän apua.

Loogisen päättelyn perusteella voisi arvioida, että JDO-toteutukset, jotka käyttävät JDBC-ajuria apunaan ovat enintään niin suorituskykyisiä ja skaalautuvia kuin kyseinen ajuri. Toisaalta JDO-teknologia mahdollistaa suurtenkin olioryhmien välimuistissa pitämisen (Sun Microsystems, 2003b), joten tietyissä tapauksissa tällä voidaan saavuttaa suoraa JDBC-ratkaisua parempaa suorituskykyä. Tämä siksi, ettei olioita aina tarvitse noutaa ja luoda uudelleen tietokantaan suoritetun kyselyn perusteella. Tarjolla on myös erityisen suorituskykyisiksi mainostettuja JDO-toteutuksia, esimerkiksi JDO Genie.

7.2.4 Ylläpidettävyys

Seuraavaksi tarkastellaan, kuinka käsiteltävät teknologiat eroavat toisistaan ylläpidettävyydeltään, muun muassa tiedon skeeman vaihtuessa.

JOS seuraa luokkien rakennetta luomalla niille versionumeron (Sun Microsystems, 1997). Mikäli luokkaa muutetaan, ei aiemmin sarjallistettuja olioita voida suoraan lukea muuttuneen versionumeron vuoksi uuden luokan ilmentymäksi. Ohjelmoija voi kiertää automaattisen versionumerotarkistuksen, jonka jälkeen luokkien rakenteisiin voidaan tehdä vähäisiä muutoksia, kuten esimerkiksi uuden kentän lisääminen, vanhan poistaminen tai attribuutin näkyvyysalueen muutos (Sun Microsystems, 2000a). Tällöin sarjallistetun olion ylimääräiset attribuutit hylätään uutta oliota muodostettaessa ja olemattomat kentät saavat oletusarvon. Tämä tarkoittaa myös sitä, että luokasta poistettujen attribuuttien tiedot kadotetaan (loss of information). Mikäli tämä ei haittaa, ei pysyvän luokan muuttaminen aiheuta ylivoimaisia ongelmia koko sovellukselle.

Mikäli käytetään JAXB-tekniikkaa ja liiketoimintaluokka muuttuu, joudutaan kirjoittamaan uusi XML-skeema ja generoimaan sidontakääntäjällä sisältöpuun rajapinnat sekä luokat uudestaan. Uusista kentistä rajapintoihin ja luokkiin tulevat uudet haku- ja asetusmetodit sekä poistettuja vastaavat katoavat. Mikäli sovelluksen kehityksessä käytetään esimerkiksi JBuilderin kaltaista ohjelmaa, joka ilmaisee syntaksivirheet kääntämättömästäkin lähdekoodista, ei muutosten jäljittäminen sovelluksessa pitäisi olla vaikeaa. Koska tiedot on varastoitu tekstimuotoisiin XML-dokumentteihin, ei myöskään mitään varastoituja tietoja kadoteta, kuten JOS-tekniikassa. XML-skeemaan on lisäksi mahdollista lisätä omia sidontasääntöjä, joilla voidaan vaikuttaa, kuinka vanhan skeeman mukainen XML-tiedosto tulisi lukea uuteen sisältöpuuhun. Näin ollen sovelluksen lähdekoodi on mielestäni melko ylläpidettävää, mutta ylimääräistä opettelua ja testaamista aiheuttaa sidontasääntöjen lisääminen XML-skeemaan tai vaihtoehtoinen XML-dokumenttien manuaalinen konvertointi uuden skeeman mukaiseksi.

Relaatiotietokantoja käyttävissä sovelluksissa, missä on vähäinen *tietoriippumattomuus*, törmätään yleensä ongelmiin tiedon skeeman muuttuessa (Connolly & Begg, 2005). Tällä tarkoitetaan tiukkaa yhteyttä käytettävän tietokannan ja sovelluslogiikan välillä. Tähän ongelmaan voidaan usein törmätä JDBC-teknologiaa käyttävissä ratkaisuissa. Tällöin skeeman muuttaminen ei aiheuta ongelmia tallennetuille tiedoille, kuten JOS-ratkaisuissa, vaan sovelluksille.

Sovelluksien lähdekoodiin upotetut SQL-lauseet (merkkijonot) suoritetaan dynaamisesti, joten mahdolliset virheet paljastuvat sovelluksessa vasta ajon aikana. Tämä johtaa siihen, että ohjelmoijan tulee tarkastaa jokainen ohjelman SQL-lause sekä niiden tulosjoukkojen käsittely uuden taulurakenteen mukaiseksi.

JDO-tekniikassa pienet muutokset tiedon skeemaan ovat helppoja: luokan lähdekoodiin tehdään tarvittavat muutokset ja tiedostot ajetaan JDO-muokkaimen läpi. Mikäli tietolähteenä käytetään tietokantaa, muutokset sen taulujen rakenteisiin voidaan tehdä JDO-toteutuksen tarjoamalla skeeman rakentajalla. Itse sovellukseen, joka käyttää liiketoimintaolioita joudutaan lisäämään tai poistamaan tarvittavat haku- ja asetusmetodit, mutta varsinaisiin olioiden palauttamisiin tai takaisin viemisiin ei jouduta koskemaan JDBC:n tavoin. Luonnollisesti sovelluksen JDOQL-kyselyt joudutaan silti tarkistamaan. Mikäli luokkaan tehdään suurempia muutoksia, tai joitain attribuutteja ei haluta pysyviksi, joudutaan myös muokkaamaan metadatatiedostoa. JDO-tekniikalla saadaan silti hyvin ylläpidettäviä ratkaisuja.

7.2.5 Tapahtumien hallinta

Kun käsitellään pitkäikäistä dataa, tulee huolehtia tapahtumien ACID-ominaisuuksista (Connolly & Begg, 2005). Epäonnistuneet tapahtumat tulee perua, käyttäjät eivät saa saada muokkauksen alla olevia tietoja käyttöönsä, tietojen väliset riippuvuudet täytyy olla kunnossa ja tietoihin tehdyt muokkaukset eivät saa kadota.

JOS-teknogian tiedostopohjainen ratkaisu ei tue tapahtumien hallintaa, vaan sovelluksen ohjelmoijan on itse huolehdittava muun muassa usean käyttäjän yhtäaikaisista muokkauksista pysyviin olioihin. Mikäli tiedostoon sarjallistettuja olioita pidetään esimerkkisovelluksen tavoin suorituksen ajan yhdessä tietorakenteessa, voidaan käyttäjien yhdenaikaisia vaikutuksia kontrolloida Javan `synchronized`-lohkoilla. Valitettavasti ajonaikaisen tietorakenteen käyttö ei tue tapahtumien säilyvyyden ominaisuutta, sillä koko tietorakenteen sen hetkinen sisältö voidaan menettää sovelluksen kaatuessa. Lisäksi ohjelmoija on täysin vastuussa muun muassa tapahtumien perumisesta.

JAXB-teknologia ei ole yhtään JOS:ia parempi pysyvyyden hallinnan suhteen. Ohjelmoija on itse vastuussa järjestetyn XML-tiedoston hallinnasta, eli sisältöpuun eheästä käytöstä ja takaisinjärjestämisestä.

Sitä vastoin JDBC-ajurit ja tietolähteinä käytettävät tietokannat ottavat tapahtuman hallintaan liittyvät tekijät huomioon. Osa tästä tapahtuu automaattisesta, mutta ohjelmoija voi JDBC-sovellusliittymän luokkien kautta huolehtia esimerkiksi tapahtumien eristämisestä, tiedon eheydestä ja epäonnistuneiden tapahtumien peruuttamisesta (Sun Microsystems, 2001). Tapahtumien hallintaa on myös mahdollista säätää JDBC-sovellusliittymän metodeilla esimerkiksi siten, että sovellus voi lukea tietokantaan tehdyt muutokset, joita ei ole vielä hyväksytty.

JDO-teknologiassa tapahtumien hallinta on oliomaisempaa ja automatisoidumpaa kuin JDBC-tekniikassa. Kaikki olioiden viemiset ja tuomiset tietolähteen ja sovelluksen välillä tehdään rajapinnan `Transaction` kautta, jonka toteuttava luokka (osa JDO-toteutusta) on vastuussa tapahtumien hallinnasta. Rajapinnan operaatioilla ohjelmoija voi jossain määrin vaikuttaa tapahtumien hallintaan, kuten perutaanko olion vieminen tietolähteeseen tai säilytetäänkö tapahtuman olioita välimuistissa myös tapahtuman päättymisen jälkeen. JDO-tekniikan vahvuus on kuitenkin siinä, että ohjelmoijan tarvitsee harvemmin säätää, tai edes tietää, tapahtuman hallintaan liittyviä yksityiskohtia.

7.2.6 Tekniikan käytettävyys

Seuraavaksi tarkastellaan tekniikoiden vaatimaa työmäärää ja niiden erikoispiirteitä. Tarkasteluun sisällytetään tekniikan opittavuus, pysyvyyden toteuttavan ohjelmakoodin tuottamisen helppous, tarvittavan konfiguroinnin määrä ja ratkaisun testattavuus.

Olioiden sarjallistamiseen perustuvan ratkaisun tekeminen on yksinkertaista, nopeaa ja olioita käsitellään olioina. Ohjelmoijan ei tarvitse tietää tietolähteeseen liittyviä yksityiskohtia. Koska sarjallistaminen on osa Javaa, ei erillisiä konfigurointeja eikä rinnakkaistekniikoita tarvita. Esimerkkisovelluksen JOS-ratkaisun ohjelmakoodimäärää ei suoranaisesti voi verrata JDBC- ja JDO -ratkaisuihin, koska yksittäisten olioiden käsittely tapahtuu tallennusdemonstraatioluokan ylläpitämään listaan eikä suoraan tietolähteeseen (Rouvinen, 2004). Mikäli tavoitteena on vain tallettaa olioita ilman suurempia vaatimuksia, on JOS-ratkaisu verrattavissa JAXB-ratkaisuun. Laskettujen puolipisteiden (;) perusteella ohjelmakoodimäärä esimerkkisovelluksen sarjallistamisratkaisussa on 34 % JAXB-ratkaisua pienempi. Lisäksi tulee ottaa huomioon, että JAXB-teknologian käytössä joudutaan kirjoittamaan myös XML-skeema, joka vaatii XSD-kielen osaamisen.

JAXB-teknologiassa tarvittavan sidontakääntäjän käyttö on sinänsä yksinkertaista, mutta sen ajaminen komentoriviltä on vaivalloista, sillä se vaatii lukuisia jar-paketteja toimintaansa. Esimerkkisovelluksen tekemisessä tuli esille myös, miten tärkeää on nimetä skeeman elementit ja niiden tyypit selkeällä ja johdonmukaisella tavalla. Näistä nimittäin muodostuvat sovelluksessa käytettävän sisältöpuun rajapinnat ja luokat. JAXB:in testattavuuden etuna voidaan JOS-ratkaisuun verrattuna nähdä se, että testaaja voi lukea tietolähteenä käytettävää XML-dokumenttia testin kuluessa. JAXB-tekniikan opittavuus on tosin vaikeampaa, johtuen sen vaatimasta luokkien generoinneista.

JDBC-ratkaisun kehitys on helppoa, mikäli liiketoimintaolioiden pilkkominen ja kokoaminen relaatiotietokannan skeeman muotoon onnistuu helposti, suorituskyvyssä ei havaita ongelmia ja SQL on ohjelmoijalle tuttu. JDBC:n vaatima konfiguroinnin määrä on kohtuullista, sillä tekniikan käyttö ei vaadi erillisasennuksia eikä JDO- tai JAXB -tekniikoiden tapaan välivaiheita sovelluskehityksessä. JDBC on olioiden sarjallistamista vaikeampi oppia, mutta suoraviivaisuutensa takia mielestäni helpompi kuin JAXB.

JDO on oliokeskeinen tekniikka olioiden pysyvyyden toteuttamiseksi. Se antaa ohjelmoijalle mahdollisuuden keskittyä liiketoimintaluokkien kehitykseen ja tarvittava ohjelmakoodi pysyy siistinä ja lyhyenä. Olioiden sarjallistamista ja XML-sidonnan arkkitehtuuria parempi se on siinä mielessä, että se kykenee yksittäisten olioiden käsittelyyn sovelluksen ja tietolähteen välillä. JDO-toteutus huolehtii automaattisesti tapahtumien hallinnasta sekä olioiden muuntamisesta kulloinkin käytettävän tietolähteen vaatimaan muotoon. Liiketoimintaluokkien muokkaus pysyväksi luokaksi ja tarvittavan metadatan kirjoittaminen JDO-toteuttajan ohjeiden avulla ei ole vaikeaa. JDO-metadatan laatiminen on helpompi oppia kuin esimerkiksi JAXB:in XML-skeeman kirjoittaminen. Lisäksi ohjelmoija voi jättää tietokannan generoinnin kokonaan JDO-toteutuksen vastuulle. JAXB-ratkaisuissa ohjelmoija joutuu määrittelemään XML-skeeman liiketoimintaluokkien mukaisiksi ja JDBC:ssä luomaan tietokannan itse. Vaikka tarvittavan konfiguroinnin määrä onkin esimerkiksi JDBC:n vaatimaa konfigurointia suurempi, niin luokkien testauksen määrä vähenee. Tämä johtuu siitä, että JDBC-ratkaisuihin kirjoitetaan usein erillisiä kerroksia pilkkomaan ja kokoamaan oliot (Sperko, 2003), joita ei JDO-ratkaisuissa välttämättä tarvita. Ylimääräistä työtä toisaalta syntyy siitä, että ohjelmoija joutuu tutustumaan käyttämäänsä JDO-toteutuksen ja sen erikoispiirteisiin. Itsessään JDO-sovellusliittymän käyttäminen ja oppiminen eivät ole JDBC-sovellusliittymää vaikeampaa.

JDO-tekniikan työmäärää pyrittiin selvittämään vertaamalla esimerkkisovelluksen JDBC-ratkaisun ohjelmakoodimäärää vastaavaan JDO-ratkaisuun. Laskettujen puolipisteiden (;) perusteella toteutuksien ohjelmakoodimäärät eivät huomattavasti eroa toisistaan. Vertailuun ei laskettu mukaan JDO-ratkaisun vaatimaa ominaisuustiedostoa eikä metadataa. Toisaalta esimerkkisovelluksessa ei käsitelty kuin yhden luokan ilmentymiä ja luultavasti JDO-tekniikan edut korostuvat suuremmilla oliomäärillä, kun taas JDBC:n haitat tulevat tällöin ilmi.

7.2.7 Ortogonaalinen persistenssi

Viimeisenä tarkasteltavana ominaisuutena on ortogonaalinen persistenssi. Aihetta on käsitelty hieman jo kohdissa 3.4, 4.5, 5.4 ja 6.4, mutta tässä kohdassa vertaamme tekniikoita toisiinsa.

Riippumattoman pysyvyyden periaatteen mukaisesti kaikella ohjelmakoodilla tulisi olla sama muoto, riippumatta sen käsittelemän tiedon iästä (Atkinson & al, 1996a). Periaate määrää myös, että tiedon käsittely sovelluksen ja tietolähteen välillä tulisi olla tuntumatonta. Tarkasti ottaen mikään käsiteltävistä tekniikoista ei toteuta tätä. Sarjallistamisessa pysyvän luokan täytyy toteuttaa järjestelmän tarjoama rajapinta. JAXB-sisältöpuun luokat ovat generoituja, joiden pitää olla käytettävän JAXB-ympäristön toteutuksen mukaisia. JDO on melko lähellä tätä periaatetta, sillä sovelluskehittäjä voi kirjoittaa pysyvät luokat haluamallaan tavalla. Tarvittava rajapinnan toteutus tehdään tämän jälkeen JDO-muokkaimella. On näkemysasia, rikkooko binaaritiedostoon tehtävät muokkaukset riippumattoman pysyvyyden periaatteen. JDBC-teknologiassa pysyvien luokkien lähdekoodilla on sama muoto kuin väliaikaisillakin luokilla, mutta ohjelmoija joutuu toteuttamaan olioiden viemisen ja tuomisen tietolähteeseen hyvin eksplisiittisesti. Tämän seikan osalta muut vertailtavat tekniikat ovat lähempänä periaatetta.

Tietotyyppien riippumattomuuden periaatteen mukaan kaikille tietotyypeille on tarjottava samat elinkaaret (Atkinsson & al., 1983). Tätä periaatetta ei toteuta tiukan tulkinnan mukaan yksikään käsiteltävistä tekniikoista, sillä Javassa on luokkia, kuten `Thread`, joiden ilmentymiä ei voida tehdä pysyviksi esitellyillä tekniikoilla. Kaikki tekniikat sisältävät sisäiset mekanismit muuntaa Javan primitiividata (merkkijonot ja luvut) tietolähteen edellyttämään muotoon ja näin tarjota tämän tyyppiselle datalle pysyvyys. Monimutkaisempien olioiden, kuten `java.util.ArrayList` suoraa pysyvyyttä eivät tue kuin JOS ja JDO. JAXB-ratkaisuissa XML-skeemaan joudutaan kirjoittamaan erilliskuvaus oliojoukoista ja JDBC-ratkaisuissa

ohjelmoija on täysin vastuussa olioiden pilkkomisesta primitiividatan tasolle. Näin ollen JOS ja JDO tarjoavat paremman tietotyyppien riippumattomuuden kuin JAXB ja JDBC.

Pysyvyyden tunnistamisen periaatteen mukaisesti pitkäikäisen tiedon tunnistamiseen täytyy olla suoraviivainen ja johdonmukainen mekanismi (Atkinson & al., 1983). Ortogonaalisen persistenssin periaatteista tämä on mielestäni kaikkein tärkein sillä, se kuvaa kuinka helposti kaikki käsiteltävät oliot saadaan pysyviksi. Sarjallistamismekanismi tuntee olioverkkojen käsitteen, jonka mukaan kaikki tallennettavaan olioon yhteydessä olevat tiedot myös tallennetaan. Näin ollen ohjelmoijan työmäärä vähenee, kun jokaista oliota ei tarvitse erikseen kirjoittaa tietovirtaan. JAXB-teknologiassa tallennetaan yhdellä komennolla kaikki sisältöpuun oliot: mikäli ohjelmoija haluaa olion pysyväksi, hän lisää sen sisältöpuuhun. Lisätyn olion yhteydet on määritelty XML-skeemassa, joten mitään varsinaista mekanismia tunnistamiseen ei tarvita sovelluksen ajon aikana. JDO kykenee tekemään pysyväksi kaikki saavutetut oliot, mikäli ne ovat Java-olioita tai niiden luokat on muokattu JDO-muokkaimella. Olioverkon pysyvyys saavutetaan yhden sen olion tallentamisella `makePersistence`-operaatiolla. JDBC-tekniikassa ei ole mitään tukea tunnistamisprosessille, vaan pysyvien tietojen määrä on riippuvainen suoritettavista SQL-lauseista.

Taulukko 7.1: Ortogonaalinen persistenssi ja tarkasteltavat tekniikat.

	JOS	*JAXB*	*JDBC*	*JDO*
Riippumaton pysyvyys	Lähellä; joudutaan toteuttamaan rajapinta	Ei; toteutusriippuvaiset luokat	Ei; vuorovaikutus tietolähteeseen toteutetaan eksplisiittisesti	Lähellä; tarvitaan tavukoodin muokkaus
Tietotyyppien riippumattomuus	Lähellä	Ei	Ei	Lähellä
Pysyvyyden tunnistaminen	Automaattista, mutta luokkien tulee toteuttaa rajapinta	Aina kaikki sisältöpuun oliot	Täydellisesti ohjelmoijan vastuulla	Automaattista, mutta luokat tulee olla muokattu

Taulukkoon 7.1 on koottu tässä kohdassa kerrotut ajatukset. Ominaisuuksia ei ole suoraan pisteytetty paremmuusjärjestykseen ortogonaalisen persistenssin mukaan, mutta taulukosta voidaan saada nopea silmäys tekniikoiden eroista. JDBC ei siis lainkaan huomioi ortogonaalisen persitenssin periaatteita, kun taas muut toteuttavat niitä melko samantasoisesti.

7.3 Ominaisuuksien yhteenveto

Tässä luvussa esitellyt teknologioiden erityispiirteet on vertailtavuuden vuoksi koottu taulukkoon 7.2. Tekniikoita on vaikea asettaa suoraan mihinkään paremmuus järjestykseen, koska ne on luotu osittain erilaisiin tarkoituksiin. Sarjallistamista ja JAXB:ia voidaan toisaalta vertailla toisiinsa kuin myös JBDC:ä ja JDO:ia, sillä ne ovat jossain määrin rinnakkaiset tekniikat.

Taulukko 7.2: Pysyvyyden toteuttamistekniikoiden vertailu.

	JOS	*JAXB*	*JDBC*	*JDO*
Tekniikan käyttämä tietolähde	Binaaritiedosto	XML-tiedosto (tekstitiedosto)	Useimmiten relaatiotietokanta	Useimmiten relaatiotietokanta, mutta voidaan vaihtaa
Opittavuus	Helppo	Sovelluksen puolella helpohko, vaatii tietämyksen XSD:sta	Peruskäyttö yksinkertaista, vaatii tietämyksen SQL:sta	Yksinkertaista ja nopeaa, vaatii osaamista metadatasta
Suorituskyky	Pienissä oliomäärissä riittävä	Tiedoston jäsentäminen hitaahkoa	Hyvä. SQL-lauseiden suoritus tehokasta, mutta tulokset joudutaan muuttamaan olioiksi.	Riippuu JDO-toteutuksesta, JDBC:n veroista.
Tapahtumien hallinta	Ei tukea	Ei tukea	Hyvä, tietokannanhallintajärjestelmä ja ajuri vastuussa	Ei kohdistu vain kyselyihin, kuten JDBC:ssa, vaan pysyviä olioita seurataan automaattisesti koko ajan
Kyselytekniikka	Ei ole	Ei ole	SQL-pohjaiset tietokantakyselyt, tehokkaita yhdisteltäessä dataa	JDOQL-kyselyt, jotka suoritetaan tietokannan ja JDO-toteutuksen yhteistyönä

Taulukko 7.2: Pysyvyyden toteuttamistekniikoiden vertailu (jatk.).

	JOS	*JAXB*	*JDBC*	*JDO*
Pysyvien olioiden muokkaus	Vaatii koko tiedoston lukemisen ja uudelleenkirjoittamisen	Vaatii koko tiedoston lukemisen ja uudelleenkirjoittamisen	Kyselyiden kautta voidaan muokata mitä tahansa tietoja, tosin ei oliomaisesti	Kyselyiden kautta voidaan muokata mitä tahansa oliota
Ylläpidettävyys	Helppo, joskin luokkien versiointi on huomioitava	Helppo, mutta vaatii uuden skeeman ja luokkien generoinnin	Sulautetut SQL-lauseet tulee tarkistaa	Helppo
Konfiguroinnin määrä	Ei tarvita	Tarvittavien jarpakettien järjestäminen, XSD-skeeman kirjoitus, luokkien generointi	JDBC-ajurin (jarpaketti) järjestäminen	JDO-toteutuksen järjestäminen, metadatan kirjoitus, luokkien muokkaus
Ortogonaalisesti peristenssi	Ei, melko lähellä	Ei	Ei	Ei, melko lähellä

JDBC ja JDO voidaan ajatella kilpaileviksi tekniikoiksi. Siinä missä JDO on oliokeskeisempi, on JDBC ehkä tehokkaammassa yhteydessä tietokantaan suorien SQL-lauseiden avulla. JDO-tekniikkaa voidaan myös käyttää binaaritiedostomaisen JOS:in kilpailijana JDORI-toteutuksella ja mikäli on olemassa XML-perusteinen JDO-toteutus, niin siten myös JAXB:in kilpailijana. JDO-tekniikan ongelmaksi on kuitenkin muodostumassa muutaman suuren ohjelmistoalan yrityksen ilmoitus, etteivät ne enää jatkossa halua tukea sitä, vaan keskittyvät EJB-teknologiaan (Sun Microsystems, 2004).

8 Johtopäätökset

Olio on teknisesti ajatellen ohjelman strukturaalinen perusyksikkö, joka säilyttää tietoa ja jolla on tiedon käsittelyyn liittyviä toimintoja. Pysyväksi olioksi sanotaan oliota, joka voi säilyttää tilansa sovelluksen suorituskertojen välillä. Pysyvä olio on täten riippumaton sovelluksen elinkaaresta. Olioiden pysyvyyden käsite liittyy tietojenkäsittelyssä tapahtuvaan tietojen tallentamiseen. Tämän tutkielman tarkoituksena oli selvittää, kuinka olioiden pysyvyys voidaan toteuttaa Java-kielessä. Tutkielman alussa tutkimuskenttään tutustuttiin kirjallisuuden kautta, jonka jälkeen esiteltiin neljä erilaista J2SE-tason tekniikkaa. Tarkasteltavista tekniikoista pyrittiin selvittämään käytön periaatteet, käyttökohteet sekä edut ja haitat.

Java Object Serialization on tämän tutkielman perusteella helpoin, mutta myös rajoitetuin mahdollisuus toteuttaa Java-olioiden pysyvyys. Se sopii periaatteessa erittäin moniin pysyvää tietoa käsitteleviin toteutuksiin. Mekanismi on helppo oppia sekä käyttää, ja se on standardi osa J2SE-ympäristöä. Käytännössä sovellukset, jotka käsittelevät suuria tietomääriä tai tarvitsevat pysyviin tietoihin kohdistuvia suorahakuja, eivät sovellu sarjallistamistekniikalle johtuen sarjatiedostomaisuudesta. Tekniikalla ei ole tukea tapahtumien hallinnalle, joten useat samanaikaiset pysyvään tietoon kohdistuvat operaatiot muodostuvat ongelmallisiksi. Sarjallistaminen on omiaan yksinkertaisissa sovelluksissa, joissa ei ole tehokkuusvaatimuksia tai todella suuria tietomääriä. Soveltuva se on esimerkiksi tapauksissa, joissa olioverkko ladataan muistiin sovelluksen käynnistyessä ja kirjoitetaan tiedostoon sovelluksen sulkeutuessa.

JAXB-tekniikka sopii sovelluksiin, joissa pysyvien olioiden tulee olla helposti jaettavissa muille osapuolille, tai suoraan ohjelmiin, joissa käsitellään XML-dokumentteja. Sen käyttö ei ole niin yksinkertaista kuin JOS-tekniikan, koska XML-skeeman kirjoittaja joutuu opettelemaan XSD-kielen. Tietojenkäsittely on itse sovelluksen ohjelmoinnissa oliomaisella tasolla eikä sovelluksen ohjelmoijan tarvitse enää välittää XML-datan rakenteesta. Pysyvien tietojen kirjoittaminen tietolähteeseen sekä niiden lukeminen tietolähteestä hoituvat muutamilla JAXB-sovellusliittymän perusoperaatiolla. Mikäli XML-dokumenttien jaettavuudella ja luettavuudella ei ole sovellukselle arvoa, on JOS luvuissa 3, 4 ja 7 esitettyjen seikkojen valossa JAXB-tekniikka parempi pysyvien olioiden käsittelyssä.

Yleisesti ottaen JDBC nähdään työvälineeksi käsitellä relaatiotietokantoja Java-sovelluksista. Olioiden pysyvyyden toteuttamiseen JDBC on hyvä valinta, mikäli liiketoimintaluokat voidaan helposti muuntaa relaatiomallin skeemaan. Erityisen hyvä se on, mikäli sovelluksen tulee käsitellä relaatiotietokannassa jo olemassaolevaa tietoa. Valitettavasti JDBC:n näkymä tietokantaan on keskittynyt relaatiomalliin, joten sovelluksen oliot tulee pilkkoa mallin mukaisesti vietäessä niitä tietolähteeseen ja vastaavasti oliot tulee aina koostaa uudelleen kyselyiden tuloksista. Mikäli sovelluksen tarve on vain noutaa pysyvät oliot, tehdä niihin muutoksia ja tämän jälkeen viedä oliot takaisin, on JDO tähän parempi vaihtoehto. Toisaalta, koska tietokannanhallintajärjestelmä on kokonaan vastuussa SQL-lauseiden suorittamisesta, JDBC-tekniikalla voidaan suorittaa laskennallisesti vaativia kyselyitä.

JDO-teknologiaa käyttävät sovellukset ovat hyvin siirrettäviä ja sen käyttö sopii erilaisiin arkkitehtuureihin. Olioiden tallentamiset, palautukset, kyselyt ja muokkaamiset ovat yksinkertaisia. Etuna voidaan nähdä myös kyselyissä käytettävä Java-pohjainen kyselykieli. Opittavuudeltaan JDO-sovellusliitymä on JDBC API:n tasolla, mutta sen käyttäminen vaatii tuntemusta JDO-toteutuksen yksityiskohdista. Käyttökohteiltaan JDO on hyvin monipuolinen, sillä sitä voidaan käyttää erityyppisten tietolähteiden kanssa: esimerkkisovelluksen JDO-levykokoelmaa testattiin binaaritiedostolla ja relaatiotietokannalla (Rouvinen, 2004).

Vaikuttaa siltä, että kaikista esitellyistä tekniikoista JDBC on tunnetuin ja käytetyin. Olioiden sarjallistaminen ja JAXB sijoittuvat täysin erilaisiin käyttökohteisiin kuin JDBC, ja sarjallistamis- sekä JAXB -tekniikoiden tarjoamia mahdollisuuksia tarvitaan huomattavasti vähemmän kuin relaatiotietokantahallintajärjestelmien. Toisaalta taas olioiden sarjallistamista ja JAXB-tekniikkaa voidaan ajatella keskenään eräänlaisina kilpailijoina. JDO monipuolisuudellaan ja helpolla ohjelmoitavuudellaan haastaa vastaavasti kaikki kolme edellistä tekniikkaa.

Yleisesti ottaen erityyppisten tietolähteiden käyttäminen johtaa sovellustasolla eri pysyvyyden toteuttamistekniikoiden valintaan. JDO-standardi on lähtenyt tätä kehitystä vastaan yrittämällä luoda universaalin tavan hoitaa Java-olioiden pysyvyys riippumatta käytettävästä tietolähteestä. JDO-tekniikan yleistymisen esteeksi kuitenkin sanotaan sen tunkeutumista J2EE-tason EJB-teknologian alueelle. Vaikkakin JDO on EJB-teknologiaa kevyempi ja sovelluspalvelimesta vapaa ratkaisu, voi tunnetumpi EJB ehkäistä sen yleisen hyväksynnän ja käytön. Mikäli

tämä ei toteudu, on mahdollista, että JDBC-ajurien päälle rakennetuttuja JDO-ratkaisuja nähdään tulevaisuudessa useissa, hyvin erilaisissa sovelluksissa.

Mikäli sovelluksen vaatimuksissa määritellään yhden tietyn tyyppisen tietolähteen käyttäminen, myös valittava pysyvyyden toteuttava tekniikka on usein hyvin ilmeinen. Mikäli tietolähdettä ei kiinnitetä ennalta, vaan sovellukselle halutaan valita sen vaatimuksiin parhaiten sopiva tallennustekniikka, on tilanne vaikeampi. Toteutettavan sovelluksen vaatimuksia ja menetelmien teknisiä piirteitä vertailemalla voidaan helpottaa tämän päätöksen tekemistä. Vertailun kannalta relevantteja muuttujia tekniikoissa ovat tarvittavat liiketoimintaolioiden muunnokset sovelluksen ja tietolähteen välillä, kyselytekniikka, tapahtumien hallinta, suorituskyky, ylläpidettävyys ja menetelmän helppous sovelluskehityksessä. Olisi houkuttelevaa sanoa, että jokin käsitellyistä tekniikoista olisi näiden piirteiden perusteella universaali ja ylivoimaisesti paras kaikkiin tapauksiin, mutta asia ei ole näin yksinkertainen. On vain joukko erilaisia vaatimuksia ja sovelluskehittäjien kokemuksia.

Olioiden pysyvyyttä voidaan tarkastella kirjallisuuden tai koesovelluksen toteuttamisesta saatujen kokemusten perusteella. Tässä työssä käytettiin molempia lähestymistapoja, joskin käytännön määrä jäi melko pieneksi. Tutkielmaa olisi voitu rajata tarkemmin koskemaan vain yhtä tai kahta kilpailevaa tekniikkaa. Tällöin tutkielman tekijä olisi voinut paneutua käytäntöön tarkemmin ja tuoda tutkielmassa havainnollisia esimerkkejä enemmän julki. Nyt toteutettu esimerkkisovellus on suppea, jonka johdosta tekniikoiden käytännön eroavaisuuksia voi olla vaikea havaita. Mikäli tutkielmaa laajennettaisiin tai jatkettaisiin, olisi ehkä mielenkiintoista tarkastella suurempien oliojoukkojen pysyvyyden toteutusta ja erityisesti toteutuksiin liittyviä suorituskykytekijöitä. Itsessään JDBC- ja JDO -tekniikat ovat myös niin laajoja, että ne tarjoaisivat itsenäisen tutkimusmahdollisuuden.

Viitteet

Association for Computing Machinery (2003) *Top Two Levels of The ACM Computing Classification System (1998)*. 1998 ACM Computing Classification System, `http://www.acm.org/class/1998/` (20.8.2004).

Atkinson, M. P., Daynès, L., Jordan, M. J., Printezis, T., Spence, S. (1996a) An orthogonally persistent Java. *SIGMOD Record* **25**(4), 68-75.

Atkinson, M. P., Jordan, M. J., Daynès, L., Spence, S. (1996b) Design Issues for Persistent Java1: a type-safe, object-oriented, orthogonally persistent system. *POS7 - Seventh International Workshop on Persistent Object Systems: Design, Implementation and Use* (toim. Connor, R. C. H., Nettles, S.), Morgan Kaufmann Publishers, New Jersey, USA, 1-16.

Atkinson, M. P., Bailey, P. J., Chisholm, K. J. (1983) An Approach to Persistent Programming. *The Computer Journal* **26**(4), 360-365.

Baldwin, R. T. (2003) Views, Objects, and Persistence for Accessing a High Volume Global Data Set. *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)* (toim. Chadduck, R.), IEEE Computer Society, Washington, DC, 77-81.

Biggs, M. (2001) JDO streamlines data access in Java. *InfoWorld* **23**(45), 60.

Brown, J. (2002) *An Introduction To Java Data Objects*. Object Computing, Inc. - Java News Brief, `http://www.ociweb.com/jnb/jnbJun2002.html` (2.2.2004).

Cattell, R. G. G., Barry, D. K., Berler, M., Eastman, J., Jordan, D., Russell, C., Schadow, O., Stanienda, T., Velez, F. (2000) *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann Publishers, USA.

Connolly, T., Begg, C. (2005) *Database Systems*. Pearson Education Limited, USA.

Connor, R. C. H., Nettles, S. (1996) *Seventh International Workshop on Persistent Object Systems: Design, Implementation and Use*. University of Pennsylvania, Department of Computer and Information Science, POS6, http://www.cis.upenn.edu/~POS96/cfp.html (1.4.2004).

Dzubak, J., Warburton, C. P. (1965) The Organization of Structured Files. *Communications of the ACM* **8**(7), 446-452.

Graves, M. (2002) *Designing XML Databases*. Prentice Hall PTR, USA.

Henry, K. (2001) *JDBC - Java Database Connectivity*. ACM Crossroads, http://www.acm.org/crossroads/columns/ovp/march2001.html (22.1.2004).

Hericko, M., Juric, M. B., Rozman, I., Beloglavec, S., Zivkovic, A. (2003) Object Serialization Analysis and Comparison in Java and .NET. *ACM SIGPLAN Notices* **38**(8), 44-54.

Hosking, A. L., Chen, J. (1999) PM3: An Orthogonally Persistent Systems Programming Language – Design, Implementation, Performance. *25th Very Large Data Base Endowment Conference* (toim. Atkinson, M. P., Orlowska, M. E., Valduriez, P., Zdonik, S. B., Bordie, M. L.), Morgan Kaufmann Publishers, Edinburgh, UK, 587-598.

Java Persistent Objects JDO (2004) *JPOX Java Persistent Objects JDO*. JPOX Java Persistent Objects JDO, http://www.jpox.org/index.jsp (15.6.2004).

Koskimies, K. (2000) *Oliokirja*. Satku - Kauppakaari, Jyväskylä.

Kurotsuchi, B. T. (1997) *The Wonders of Java Object Serialization*. ACM Crossroads, http://www.acm.org/crossroads/xrds4-2/serial.html (20.1.2004).

Laine, H. (1996) *Oliosanasto*. Helsingin yliopisto, Tietojenkäsittelytieteen laitos, http://www.cs.helsinki.fi/u/laine/oliosanasto/suom_termit.html (1.4.2004).

McCaughey, A., Lunney, T. (2003) Object Persistence in Java. *2nd international conference on Principles and practice of programming in Java* (toim. Waldron, J.), Computer Science Press, Inc., New York, 115-120.

Monday, P., Edwards, J. D. (2002) *Hands-on Java Data Objects*. IBM, developersWorks, `http://www-106.ibm.com/developerworks/java/edu/j-dw-javajdo-i.html` (5.2.2004).

Object Data Management Group (2004) *Object Data Management Group - The Standard for Storing Objects*. ODMG Home Page, `http://www.odmg.org/` (17.8.2004).

Opyrchal, L. Prakash, A. (1999) Efficient object serialization in Java. *19th IEEE International Conference on Distributed Computing Systems Workshops on Electronic Commerce and Web-Based Applications* (toim. Chanson, S., Sun, W., Tygar, J. D.), IEEE Computer Society, Austin, USA, 96-101.

Ort, E., Mehta, B. (2003) *Java Architecture for XML Binding (JAXB)*. A Sun Developer Network Site, `http://java.sun.com/developer/technicalArticles/WebServices/jaxb/index.html` (20.5.2004).

Persistent Programming Research Group (1998) *POS8 - Eighth International Workshop on Persistent Object Systems: Design, Implementation and Use*. The POS8 Workshop, `http://www-ppg.dcs.st-and.ac.uk/Conferences/POS8/CFP.html` (1.4.2004).

Rouvinen, J. (2004) *Java ja olioiden pysyvyys*. Tietojenkäsittelytieteen laudatur-harjoitustyö, Joensuun yliopisto, Tietojenkäsittelytieteen laitos, Joensuu.

Samson, E. (2001), *JDO*. ObjectWeb Consortium, ObjectWeb Conferences, `http://www.objectweb.org/phorum/download.php/36,18/JDO.pdf` (5.2.2004).

Sperko, R. (2003) *Java Persistence for Relational Databases*. Apress, USA.

Srinivasan, V., Chang, D. T. (1997) *Object persistence in object-oriented applications*. IBM Systems Journal, Application Development, `http://www.research.ibm.com/journal/sj/361/srinivasan.html` (1.4.2004).

Sun Microsystems (1997) *Java Object Serialization Specification*. California, USA.

Sun Microsystems (2000a) *Discover the secrets of the Java Serialization API*. A Sun Developer Network Site, `http://java.sun.com/developer/technicalArticles/Programming/serialization` (20.1.2004).

Sun Microsystems (2000b) *JDBC Overview*. A Sun Developer Network Site, `http://java.sun.com/products/jdbc/overview.html` (20.1.2004).

Sun Microsystems (2000c) *PJama Release 1.65*. Sun Microsystems Laboratories, `http://www.sunlabs.com/research/forest/opj.main.html` (5.5.2004).

Sun Microsystems (2001) *JDBC 3.0 Specification*. California, USA.

Sun Microsystems (2003a) *Java Data Objects*. Java Data Objects, `http://access1.sun.com/jdo` (4.2.2004).

Sun Microsystems (2003b) *Java Data Objects (JDO) Specification*. California, USA.

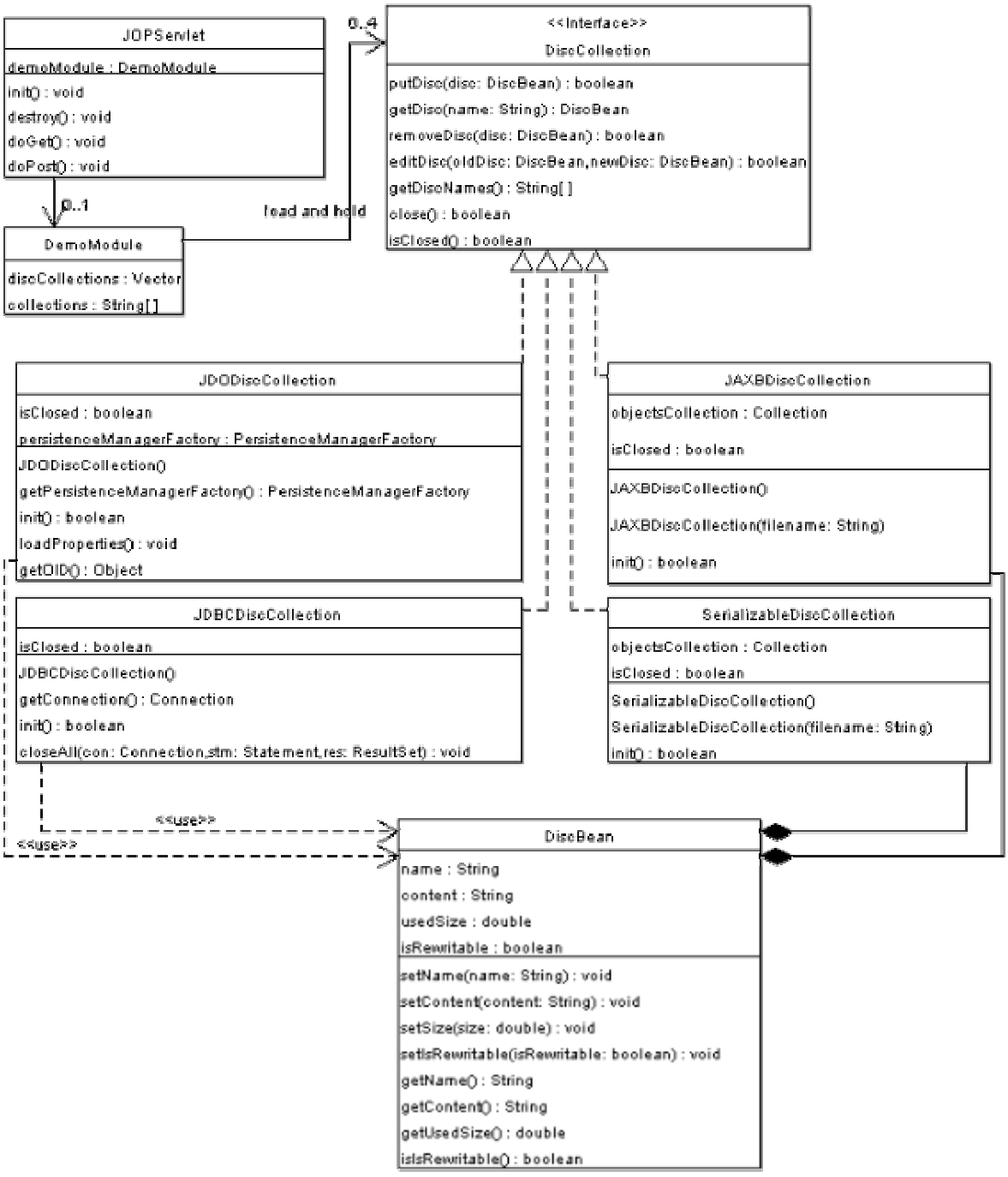
Sun Microsystems (2003c) *Java Data Objects and Transparent Persistence: Where and Why to Use Them*. A Sun Developer Network Site, `http://developers.sun.com/prodtech/javatools/jsstandard/reference/techart/jdo.html` (20.1.2004).

Sun Microsystems (2003d) *The Java Architecture for XML Binding (JAXB) Specification*. California, USA.

Sun Microsystems (2003e) *The Java Web Services Tutorial*. California, USA.

Sun Microsystems (2004) *Java Data Objects 2.0 – An Extension to the JDO Specification, JSR Review Ballot*. Java Community Process, `http://www.jcp.org/en/jsr/results?id=2548` (22.7.2004).

Liite 1: Esimerkkisovelluksen luokkakaavio



Levykokoelmaluokkia, siis tallennusdemostraatioluokkia, ovat `JDODiscCollection`, `JDBCDiscCollection`, `JAXBDiscCollection` ja `SerializableDiscCollection`. Nämä luokat käsittelevät luokan `DiscBean` (suomeksi `LevyPapu`) ilmentymiä.

Liite 2: LevyPapu-luokan lähdekoodi (DiscBean.java)

Esimerkkisovellus on toteutettu englanniksi. Tutkielman esimerkeissä on käytetty luokan `DiscBean` suomenkielistä nimeä `LevyPapu` sekä vastaavia suomenkielisiä attribuuttinimiä ja metodeita.

```
package jop;                      // Java ja Olioiden Pysyvyys
import java.io.*;

public class DiscBean implements Serializable {

  private String  name;            // levyn nimi
  private String  content;         // levyn sisällön kuvaus
  private double  usedSize;        // levystä käytetty tila
  private boolean isRewritable;    // voidaanko levylle lisätä tietoa

  public DiscBean() {
    this("", "", 0, false);
  }

  public DiscBean(String name, String content, double size,
                  boolean isWritable){
    this.name     = name;
    this.content  = content;
    this.usedSize = size;
    this.isRewritable = isWritable;
  }

  public String getContent()      { return content; }

  public boolean isIsRewritable() { return isRewritable; }

  public String getName()         { return name; }

  public double getUsedSize()     { return usedSize; }


  public void setUsedSize(double usedSize) {
    this.usedSize = usedSize;
  }

  public void setName(String name) {
    this.name = name;
  }

  public void setIsRewritable(boolean isRewritable) {
    this.isRewritable = isRewritable;
  }

  public void setContent(String content) {
    this.content = content;
  }

}
```

Liite 3: Sanasto

API	Application Programmer Interface, sovellusliittymä.
CMP	Container-Managed Persistence. EJB-teknologian käyttämät CMP-pavut hoitavat sovellusten ja tietovaraston välisen vuorovaikutuksen.
DOM	Document Object Model. Alusta- ja kieliriippumaton rajapinta, joka kuvaa dokumentin puumaisen rakenteena. Käytetään XML- ja HTML -dokumenttien käsittelyssä.
EJB	Enterprise JavaBeans. On palvelimille suunnattu Javan komponenttiteknologia.
J2SE	Java 2 Standard Edition. On ympäristö Java-ohjelmistojen ohjelmointiin ja käyttöön, ns. Javan perusversio.
JAXB	Java Architecture for XML Binding. Javan teknologia, joka mahdollistaa XML-dokumenttien oliomaisen käsittelyn.
JDBC	Java Database Connectivity. Universaali ja toimittajariippumaton standardi, jonka avulla Java-sovellukset voivat operoida tietokantojen kanssa.
JDO	Java Data Objects. Tekniikka, joka määrittelee oliomaisen, tuntumattoman ja toimittajariippumattoman tavan tallentaa olioita.
JOS	Java Object Serialization. Yleiskäyttöinen mekanismi, jolla oliot voidaan tallentaa Java-ohjelmien ymmärtämään sarjarakenteiseen binaaritiedostoon.
JSP	JavaServer Pages. Java 2 Enterprise Edition arkkitehtuuriin kuuluva servlettien laajennus. JSP-moottori kääntää JSP-tiedoston servletiksi, silloin kun sitä ensimmäisen kerran tarvitaan.
JSX	Java Serialization to XML. Kaupallinen ratkaisu, jolla voidaan muuntaa Java-olioiden tiedot XML-dokumenteiksi.
MySQL	MySQL on suosittu, tehokas ja vapaa SQL-tietokannan hallintajärjestelmä.
RMI	Remote Method Invocation. Hajautettujen Java-sovellusten tekemiseen kehitetty teknologia, joka mahdollistaa vuorovaikutuksen sovellusten olioiden välillä.
SAX	Simple API for XML. On ensimmäinen laajasti hyväksytty Javan sovellusliit-

	tymä XML-dokumenttien käsittelyyn. On jäsennin XML-datan käsittelyyn.
Servletti	Palvelinsovelma. Web-palvelimella oleva sovelluskohtainen laajennus, joka käsittelee asiakkaan URL-pyyntöjä. Voidaan esimerkiksi muodostaa Internet-sivuja dynaamisesti. Osa Java 2 Enterprise Edition -arkkitehtuuria.
SQL	Structured Query Language on standardoitu relaatiotietokantojen hallintakieli.
Tomcat	Sovelluspalvelin, jolla voidaan ajaa muun muassa servlettejä ja JSP-sivustoja.
XML	Extensible Markup Language. Tietoverkkoja varten kehitetty rakenteisten dokumenttien merkintäkieli, jolla tieto voidaan esittää laite- ja ohjelmariippumattomasti.
XSD	XML Schema Defintion. XML-skeemojen kuvauksessa käytetty kieli, tunnetaan myös nimellä XML Schema Language.