

**Pysyvyyden toteuttaminen Java-sovelluksissa erityisesti ORM-  
menetelmän avulla**

Lauri Pekkala

Tampereen yliopisto  
Tietojenkäsittelytieteiden laitos  
Tietojenkäsittelyoppi /  
Ohjelmistokehitys  
Pro gradu -tutkielma  
Joulukuu 2004

Tampereen yliopisto

Tietojenkäsittelytieteiden laitos

Tietojenkäsittelyoppi / Ohjelmistokehitys

Lauri Pekkala: Pysyvyyden toteuttaminen Java-sovelluksissa erityisesti ORM-  
menetelmän avulla

Pro gradu -tutkielma, 77 sivua, 1 liitesivu

Joulukuu 2004

---

Tutkielmassa esitellään joitakin menetelmiä, joilla pysyvyyden voi toteuttaa Java-sovelluksissa. Tarkemmin käsitellään ORM-menetelmää, joka abstrahoi oliopohjaisen sovelluksen taustalla olevan relaatiotietokannan mahdollistaen siten olioparadigman mukaisen ohjelmoinnin. Suurin osa pysyvyyden toteuttamisen yksityiskohdista, kuten SQL-lauseiden luomisesta ja suorittamisesta, siirretään ORM-kerroksen vastuulle. ORM-menetelmän toteuttamiseksi käytännössä on suositeltavaa käyttää valmista ohjelmistokehystä. Tätä tarkoitusta varten esitellään kehys, jonka nimi on Hibernate. Hibernaten yleisemmän esittelyn lisäksi kerrotaan myös sen käyttämisestä, sekä roolista kerrosarkkitehtuurissa esimerkkisovelluksen avulla. Lopuksi kerrotaan vielä, miten tutkielmaa tukeva ohjelmistoyritys, Solita Oy, on soveltanut Hibernaten avulla menestyksekkäästi ORM-menetelmää.

Avainsanat ja -sanonnat: Pysyvyys, ORM, Java, Hibernate, Oliot, Relaatio-  
tietokanta

## Kiitokset

Haluan kiittää Tapio Niemeä tutkielman ohjaamisesta ja tarkastamisesta, sekä Jyrki Nummenmaata tutkielman tarkastamisesta. Kiitän myös työnantajaani Solita Oy:tä saamastani mahdollisuudesta tehdä tutkielmaa työajalla. Työnantajani puolelta kiitän aiheen hyväksyneitä henkilöitä, Harri Kulmalaa ja Raimo Arvolaa. Hyvistä neuvoista ja kommentteista haluan antaa erityisen suuret kiitokset Juha Komulaiselle, Pirkka Jokelalle ja Elina Huhtalalle. Kiitän lisäksi kaikkia muitakin työtovereitani hyvän työilmapiirin luomisesta. Suuret kiitokset kuuluvat myös vanhemmilleni, jotka ovat aina kannustaneet minua peruskoulu,- lukio,- ja yliopistovuosieni aikana. Tuki, jota olen saanut ystäviltäni ja sukulaisiltani on samoin ollut todella tärkeää. Suurimmat kiitokset menevät kuitenkin rakkaalle vaimolleni Lauralle, joka on auttanut jaksamaan toisinaan raskaidenkin opiskeluvuosieni yli.

## Sisällys

1.	Johdanto.....	1
2.	Pysyvyys.....	3
2.1.	Sarjallistaminen.....	3
2.1.1.	Serializable-rajapinta.....	3
2.1.2.	Esimerkki.....	4
2.1.3.	Sarjallistaminen käytännössä.....	5
2.2.	Prevayler.....	5
2.2.1.	Muistin käyttäminen ja POJO-oliot.....	5
2.2.2.	Palautumismekanismi.....	6
2.2.3.	Esimerkkisovellus.....	7
2.2.4.	Yhteenvedo.....	9
2.3.	Oliotietokannat.....	9
2.3.1.	Oliomallista.....	10
2.3.2.	Ozone.....	11
2.3.3.	Perusteluita oliotietokannan käyttämiselle.....	14
2.4.	Relaatiotietokannat.....	17
2.4.1.	Relaatiomalli.....	17
2.4.2.	Relaatioalgebra ja SQL.....	19
2.4.3.	JDBC.....	20
3.	ORM.....	23
3.1.	Taustaa.....	23
3.2.	Olio- ja relaatioparadigmojen välinen yhteensopivuusongelma.....	24
3.3.	Mitä ORM on?.....	25
3.4.	Pysyvyysmekanismin tarvitsema informaatio ja metadata.....	27
3.5.	Attribuuttien kuvaaminen.....	27
3.6.	Suhteiden kuvaaminen.....	29
3.6.1.	1:1.....	30
3.6.2.	1:n.....	31
3.6.3.	n:m.....	32
3.7.	Periytymisen kuvaaminen.....	32
3.7.1.	Yksi taulu koko luokkahierarkialle.....	33
3.7.2.	Yksi taulu konkreettista luokkaa kohti.....	33
3.7.3.	Yksi taulu luokkaa kohti.....	34
3.7.4.	Luokkien kuvaaminen geneeriseen taulurakenteeseen.....	35
3.8.	Tietokannan suunnittelusta.....	36
4.	ORM-kehykset.....	38
4.1.	Ohjelmistokehyksistä.....	38

4.2.	Mapper ja Data Mapper .....	39
4.3.	Identity Mapper .....	40
4.4.	Unit of Work.....	40
4.5.	Lazy Load .....	41
5.	Hibernate .....	43
5.1.	Arkkitehtuuri ja tärkeimmät rajapinnat.....	43
5.2.	Konfigurointi ja käyttöönotto .....	46
5.3.	Kuvausten määrittäminen.....	46
5.4.	Välimuistista.....	47
5.5.	HQL ja Criteria.....	48
5.6.	Proxy-mallin soveltamisesta .....	51
6.	Hibernate käytännössä .....	55
6.1.	Esimerkkisovellus .....	55
6.2.	Pysyvyyskerros .....	56
6.2.1.	Kohdealueen oliomalli, XDocletit ja POJO-oliot.....	56
6.2.2.	Kuvausten generointi .....	61
6.2.3.	Tietokantakaavion generointi.....	62
6.2.4.	DAO-luokkien toteuttaminen.....	63
6.3.	Palvelukerros.....	65
6.4.	Www-kerros.....	66
6.4.1.	DispatcherServlet.....	66
6.4.2.	Kontrollerit .....	67
6.4.3.	JSP-sivut.....	68
7.	Kokemuksia Hibernaten käyttämisestä Solitan projekteissa .....	69
7.1.	Soveltuvuus .....	69
7.2.	Hyvät puolet.....	70
7.3.	Ongelmia ja ratkaisuja .....	71
7.4.	Parannusta aiempiin menetelmiin?.....	72
7.5.	Yhteenvedo .....	73
8.	Yhteenvedo .....	76
	Viiteluettelo .....	78
	Liite A: Termit ja lyhenteet .....	82

## 1. Johdanto

Useimmille sovelluksille tiedon pysyvä tallettaminen on olennaista. Oliopohjaisten sovellusten yhteydessä puhutaan olioiden pysyvyydestä (persistence), eli yleisesti siitä, että pysyvien olioiden tila voidaan pitää tallessa kiintolevyllä ja se voidaan palauttaa keskusmuistiin, kun olioita halutaan käsitellä.

Suuri osa nykyaikaisista ohjelmistoprojekteista toteutetaankin oliopohjaisia menetelmiä hyödyntäen. Tietojenkäsittelyn ala on tunnetusti mennyt vauhdilla eteenpäin ja erityisen vauhdikkaasti on edistytty viimeisen kymmenen vuoden aikana. Oliopohjaisten menetelmien suosiota on osaltaan kasvattanut Javan suosion valtava nousu, joka alkoi Netscapen hankittua Javan lisenssin Sun Microsystemsiltä ja rakennettua sille tuen Navigator 2.0 selaimen vuonna 1995. Tämä takasi Javalle nopean leviämisen tuolloin maailman suosituimman selaimen mukana. Tänä päivänä Java on eräs maailman tunnetuimmista ja käytetyimmistä ohjelmointikielistä. Tässäkin tutkielmassa Javalla on suuri rooli.

Olioiden pysyvyyden voi toteuttaa useilla eri tavoilla ja menetelmää kannattaakin harkita tapauskohtaisesti, sillä esimerkiksi toteutettavan sovelluksen koon ja käyttötarkoituksen pitäisi aina vaikuttaa menetelmän valintaan. Erityisen tärkeitä kyseiset päätökset ovat yrityksille, jotka pyrkivät toteuttamaan luotettavia, turvallisia, tehokkaita ja helposti ylläpidettäviä sovelluksia kustannustehokkaasti ja kilpailukykyisesti.

Paterson ja Haddow [Paterson ja Haddow, 2004] luettelevat artikkelissaan seuraavia tapoja toteuttaa olioiden pysyvyys Java-projekteissa: Sarjallistaminen, tallentaminen relaatiotietokantaan, tallentaminen oliotietokantaan, ORM-kehysten (object relational mapping) käyttäminen ja sarjallistamiseen pohjautuvan Prevaler-sovelluskirjaston toteuttaman menetelmän (prevalence) käyttäminen.

Sarjallistamisella tarkoitetaan olioiden tilan tallentamista tavuista koostuvaan sekvenssiin ja rakenteen purkamista tarvittaessa jälleen olioiksi. Relaatiotietokantaan tallentaminen erilaisten tietokantarajapintojen avulla lienee yleisin tapa sovelluksen pysyvyyden toteuttamiseen. Oliotietokannat puolestaan tallentavat oliot kunkin sovelluksen oliomallin mukaisesti. Vähemmän tunnettu lähestymistapa pysyvyyden toteuttamiseen on taas Prevalerin malli [Prevaler, 2004], jossa olioiden tilaa muuttavat komennot sarjallistetaan ja kirjoitetaan lokitiedostoon olioiden sijaan. Tämän tiedoston perusteella voidaan oliomalli luoda uudestaan, mikäli keskusmuistissa oleva oliorakenne menetetään.

ORM-kehukset, kuten esimerkiksi Hibernate [Hibernate, 2004], ovat puolestaan välineitä, jotka mahdollistavat olioparadigman mukaisen ohjelmoinnin, vaikka tieto tallennetaankin relaatiotietokantaan. Olio- ja relaatioparadigmat eivät ole suoraan yhteensopivia, mutta muiden muassa Amblerin [Ambler, 2003] mukaan yhteensopivuutta koskevat ongelmat voidaan ratkaista, mikäli ymmärretään prosessi, jonka avulla kohdealueen oliomalli liitetään relaatiotietokannan kaavioon, ja osataan toteuttaa tuo liitos oikein. Tässä tutkielmassa pääpaino on tällä menetelmällä.

Tutkielman toisessa luvussa esitellään tarkemmin edellä mainittuja pysyvyyden toteutustapoja lukuun ottamatta ORM-menetelmää, jota käsitellään kolmannessa luvussa. Neljännessä luvussa paneudutaan ORM-kehysten (frameworks) arkkitehtuuriin, jonka jälkeen esitellään Hibernate, joka on teknisesti hyvin toteutettu, helposti saatavilla oleva, ilmainen ja suosittu avoimen lähdekoodin ORM-kehys. Hibernaten toimintatavan ja käytön tunteminen auttaa myös yleisemmällä tasolla ymmärtämään, mistä ORM-menetelmässä on kyse. Siitä, mikä Hibernaten rooli on sovelluksen kerrosarkkitehtuurissa, kerrotaan kuudennessa luvussa pienen esimerkkisovelluksen avulla. Tutkielmaa tukevalla yrityksellä, Solita Oy:llä, on paljon kokemusta Hibernaten käyttämisestä erilaisissa projekteissa. Tutkielman seitsemännessä luvussa kerrotaankin kyselyyn ja haastatteluihin perustuen näistä pääsääntöisesti positiivisista kokemuksista, joita ORM-menetelmästä ja erityisesti Hibernatesta on saatu erilaisissa projekteissa.

Tutkielman tavoitteena on siis ensinnäkin luoda katsaus joihinkin pysyvyyden toteuttamisen menetelmiin Java-sovelluksissa, mutta painottaa näistä ohjelmistoyrityksen näkökulmasta tärkeintä, ORM-menetelmää, sekä menetelmän toteuttavaa kehystä, Hibernatea. Menetelmän käyttökelpoisuutta perustellaan eri tavoin tutkielman edetessä, mutta käytännön kokemuksista kertovan seitsemännen luvun tehtävänä on lopulta vakuuttaa, että ORM ja Hibernate soveltuvat erinomaisesti ohjelmistoyrityksen toteuttamiin projekteihin.

## 2. Pysyvyys

Pysyvyydellä tarkoitetaan sitä, että järjestelmä mahdollistaa olion tilan tallentamisen. Käsite tunnetaan arkikielessä myös nimellä persistenssi. Tässä luvussa perehdytään pysyvyyden toteuttamiseen Java-sovelluksissa muiden kuin ORM-menetelmän avulla, mihin keskitytään perusteellisemmin kolmanesta luvusta alkaen.

Luvun kohdat noudattavat järjestystä, jossa teknisesti yksinkertaisimmat ratkaisut esitellään ennen monimutkaisempia. Siksi ensimmäisessä kohdassa perehdytään sarjallistamiseen, jossa on kyse olioiden tilan tallentamisesta tiedostoon. Seuraavaksi käsitellään toista, myös sarjallistamiseen perustuvaa menetelmää, joka kuitenkin poikkeaa olennaisesti perinteisestä sarjallistamisesta. Kyseisen menetelmän toteuttavan sovelluskirjaston nimi on Prevayler.

Näiden jälkeen esitellään teknisesti mutkikkaammat menetelmät, eli olio- ja relaatiotietokannat. Vaikka oliotietokannat eivät missään vaiheessa ole saavuttaneet erityisen suurta suosiota, ei niitä kuitenkaan ole syytä sivuuttaa, sillä ei ole varmaa, mikä niiden rooli tulee olemaan jatkossa. Relatiotietokannat ovat puolestaan vuosikymmenten mittaan saavuttaneet valtavan suosion erityisesti suurten tietojärjestelmien taustalla.

### 2.1. Sarjallistaminen

Sarjallistaminen (serialization) on Javan ominaisuus, jonka avulla sarjallistettavaksi kelpaavat oliot voidaan tallentaa esimerkiksi tiedostoon. Ainoa ehto luokan sarjallistuvuudelle on, että se toteuttaa Serializable-rajapinnan. Tällöin myös kaikki kyseisen luokan aliluokat ovat sarjallistettavia. Vaikka kaikki rajapinnan toteuttavat oliot ovatkin sarjallistettavia, ei sarjallistaminen aina ole mahdollista. Vastuu sarjallistamisesta on aina lopulta ohjelmoijalla.

#### 2.1.1. Serializable-rajapinta

Serializable-rajapinta ei sisällä yhtään funktiomäärittelyä. Rajapinnan tehtävänä on vain kertoa, että sen toteuttava luokka voidaan sarjallistaa, eli kirjoittaa tavuvirtaan (bytestream) [Javadoc, 2003]. Jos virtaan yritetään kirjoittaa sarjallistumatonta oliota, tapahtuu poikkeus (NotSerializableException). Mikäli luokka tarvitsee erityistä käsittelyä sarjallistamisessa, tulee sen toteuttaa esimerkissä 1 mainitut funktiot omalla tavallaan.

- 
1. `private void writeObject(java.io.ObjectOutputStream out) throws IOException;`
  2. `private void readObject(java.io.ObjectInputStream in)`
-

- 
3. `ClassNotFoundException;` `throws IOException,`

---

### Esimerkki 1. Funktiot `readObject` ja `writeObject`

---

Luokka voi myös tarvittaessa toteuttaa `Serializable`-rajapintaa laajentavan `Externalizable`-rajapinnan, ja jos näin on, käytetään sarjallistamismekanismissa `Externalizable`-rajapinnan metodeita `readExternal(ObjectInput in)` ja `writeExternal(ObjectOutput out)`, jotka luokan on toteutettava. `Externalizable`-rajapinnan toteuttaminen antaa sen toteuttavalle luokalle vapaat kädet virtojen formaatin ja sisällön muodostamiseen.

#### 2.1.2. Esimerkki

Esimerkeissä 2 ja 3 esitellään normaalia sarjallistamista ja annetaan Javan huolehtia virran formaatista ja sisällöstä. Esimerkissä 2 kirjoitetaan `OwnClass`-luokan olio `obj` tiedostoon `filename.dat` ja esimerkissä 3 se luetaan takaisin muistiin.

---

```
1. File file = new File("filename.dat");
2. FileOutputStream fileOut = new FileOutputStream(file);
3. ObjectOutputStream outputStream = new ObjectOutputStream(fileOut);
4. OwnClass obj = new OwnClass();
5. outputStream.writeObject(obj);
6. outputStream.flush();
7. outputStream.close();
```

---

### Esimerkki 2. Olion sarjallistaminen

Rivillä 1 luodaan `File`-luokan olio. Toisella rivillä luodaan tulostusvirta (`output stream`), jolle annetaan tiedosto rakentimessa. Varsinaisen sarjallistamisen tekee seuraavaksi luotava `ObjectOutputStream`-luokan olio, jolle annetaan parametrina rivillä 2 luotu tulostevirta. Tulostevirtaan voidaan nyt kirjoittaa mielivaltainen määrä mitä tahansa olioita `writeObject(Object)` funktion avulla. Tässä esimerkissä virtaan kirjoitetaan vain yksi olio, `OwnClass` tyyppinen olio `obj`. Lopuksi tulostevirrälle suoritetaan metodit `flush()` ja `close()`, joista ensimmäinen varmistaa, että puskuri on tyhjentynyt virtaan ja jälkimmäinen sulkee virran.

---

```
1. File file = new File("filename.dat");
2. FileInputStream fileInputStream = new FileInputStream(file);
3. ObjectInputStream objectIO = new ObjectInputStream(fileInputStream);
4. OwnClass obj = (OwnClass)objectIO.readObject();
```

---

### Esimerkki 3. Olion lukeminen

Myös tavuvirran lukeminen on suoraviivaista; tarvitaan vain `InputStream`-tyyppisiä luokkia. Olion lukemiseksi tavuvirrasta tarvitsee kuitenkin tehdä tyyppimuunnos oikeaan luokkaan, joten etukäteen on oltava tiedossa, minkä

tyyppisiä olioita virta pitää sisällään. Järjestyksen, jolla luetaan, pitää siis olla täsmälleen sama kuin järjestyksen, jolla on sarjallistettu.

### 2.1.3. Sarjallistaminen käytännössä

Sarjallistamista käytetään pääosin etäkutsujen (RMI) yhteydessä olioiden väliseen kommunikaatioon sokettien (socket) kautta. Lisäksi sarjallistamisen avulla voidaan toteuttaa pysyvyys pienemmissä, paikallisissa sovelluksissa, jolloin sovellus voi ajon jossain vaiheessa sarjallistaa joukon olioita ja ottaa ne taas myöhemmin käyttöön [Javadoc, 2003]. Sarjallistamista voidaan joskus myös käyttää yhdessä relaatiotietokantojen kanssa.

## 2.2. Prevyler

Toisenlainen, myös sarjallistamiseen perustuva menetelmä on Prevylerin malli, jossa perusideana on muistin käyttö ja transaktioiden sarjallistaminen. Prevylerin ensimmäinen versio tuli saataville loppuvuodesta 2001 avoimen lähdekoodin projektina [Villela, 2002]. Kuriositeettina mainittakoon, että vuonna 2002 Prevylerissa (versio 1.3.0) oli ainoastaan noin 350 riviä ohjelmakoodia. Nykyisessä versiossa on noin 1000 riviä.

Prevylerin [Prevyler, 2004] kotisivujen mukaan kyseessä on sovellus, joka on todella helppokäyttöinen, tähän mennessä nopein ja läpinäkyvin pysyvyyden, vikasietoisuuden, sekä kuormantasauksen (load balance) arkkitehtuuri POJO (Plain Old Java Object)-olioille. Edelleen sivuilla esitetään testituloksia, joiden mukaan Prevylerilla tehdyt kyselyt ovat 9000 kertaa Oraclea ja 3000 kertaa MySQL:ää nopeampia niiden käyttäessä JDBC-rajapintaa. Monet Prevylerin käyttäjät ovat kritisoineet tuloksia Prevylerin sivustolla, joten tuloksiin kannattaa suhtautua tietyllä varauksella. Kaikesta huolimatta, oli testit suoritettu miten tahansa, ovat niiden tulokset siinä määrin kiinnostavia, että menetelmään tutustuminen on kannattavaa.

### 2.2.1. Muistin käyttäminen ja POJO-oliot

Prevyler perustuu muistin hyväksikäyttöön. RAM on nykyään halpaa ja muistiteknologia kehittyä kaiken aikaa, joten suurten muistimäärien hankkiminen on mahdollista. Prevylerissa oliomalli pidetään kokonaisuudessaan koko ohjelman suorittamisen ajan muistissa ja koska muistin käyttö on paljon kovalevyn käyttöä nopeampaa, Prevylerilla tehdyt kyselyt ovat siten erittäin nopeita.

Prevyler ei sisällä erityisiä kyselykieliä tai kyselyiden suorittamismekanismeja, vaan sitä käytettäessä voidaan soveltaa kaikkia niitä menetelmiä tiedon käsittelemiseen, joita voidaan käyttää POJO-olioille. POJO tulee sanasta "Plain Old Java Object" ja on alun perin Martin Fowlerin ja muiden vuonna 2002 käyttöönottama termi tavanomaisesta, perinteisestä ja riippumattomasta

Java oliosta [Green, 2004]. POJO-oliolla on liiketoimintametoodeita (business methods), jotka määrittelevät oliion käyttäytymisen ja ominaisuuksia, jotka määrittelevät sen tilan. Osa ominaisuuksista toimii viitteinä muihin POJO-oliioihin [Bauer ja King, 2004]. Eräs hyvä kirjasto muistissa olevien olioiden käsittelemiseen on muun muassa Jakarta Commons Collections [Jakarta CC, 2004].

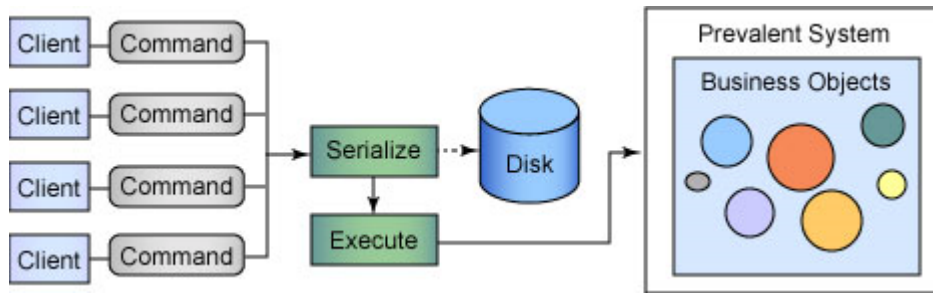
Ainoat rajoitteet palvelinpuolen tallennettaville oliioille ovat, että niiden tulee olla sarjallistettavia (kohta 2.1) ja deterministisiä. Deterministisyys tarkoittaa sitä, että olioiden tulee aina tuottaa samat tulokset samoilla komennoilla [Villela, 2002]. Jälkimmäinen vaatimus ei sikäli ole erityisen merkittävä, että lähes kaikki POJO-oliot ovat valmiiksi deterministisiä. Epädeterministisiä oliioita ovat esimerkiksi sellaiset, jotka käsittelevät esimerkiksi tiedostoja, soketteja (socket) ja järjestelmän kelloa. Tällaisten olioiden tulee kuitenkin harvoin jos koskaan olla pysyviä. Asiakaspuolen sovelluksen liiketoimintalogiikkaa käsittelevien komentojen on puolestaan oltava sarjallistettavia Transaction-luokan oliioita.

### 2.2.2. Palautumismekanismi

Muistin käyttöön pääasiallisena tiedontallennusmedian liittyä tietysti ainakin se ongelma, että mikäli virta katkeaa, muistipiiri tyhjäntyy. Näin ollen tarvitaan mekanismi, jonka avulla keskeytyneen sovelluksen oliomalli voidaan palauttaa entiseen tilaansa. Prevaयरin idea on juuri tässä; se sisältää kyseisen mekanismin.

Ennen mekanismin selittämistä, määritellään jo edelläkin mainittu transaktion käsite Prevaयरin yhteydessä. Yksinkertaisesti selitettynä Prevaयरin transaktio on sellainen operaatio, joka muuttaa järjestelmän tilaa jollakin tavalla. Käytännössä transaktiot kapseloidaan olioiksi, jotka toteuttavat Transaction-rajapinnan.

Mekanismi perustuu siihen, että ennen minkään transaktion päättämistä oliioille asti, sarjallistetaan se levyn lokitiedostoon (kuva 1, s. 7). Järjestelmä voi myös tarvittaessa muodostaa oliosta snap shot-tiedoston, joka sisältää koko oliomallin sen hetkessä tilassaan [Villela, 2002]. Jos sovellus kaatuu, haetaan aluksi sen viimeisin tallennettu tila tuosta tiedostosta, mikäli sellainen on saatavilla. Sitten lokitiedostossa olevista transaktioista ajetaan ne, jotka on tehty viimeisen oliomallin tilan tallennuksen jälkeen. Transaktiot suoritetaan oliioille aivan kuin ne olisivat tulleet järjestelmän asiakasohjelmilta. Tämän proseduurin jälkeen järjestelmä on samassa tilassa kuin ennen kaatumista.



Kuva 1. Transaktiot sarjallistetaan ennen suorittamista [Villela, 2002]

Edellä mainitun Jakarta Commons Collections-kirjaston ohella tietorakenteiden selaamiseen voidaan käyttää myös XML-pohjaisia menetelmiä, sillä Prevayler tukee snap shot-tiedostojen tallentamista myös XML-muotoon. Näin ollen siis mitä tahansa XML-tiedostoja hallitsevaa ohjelmaa voidaan käyttää tiedoston käsittelemiseen.

### 2.2.3. Esimerkkisovellus

Seuraavassa esimerkissä havainnollistetaan Prevaylerin peruskäyttöä yksinkertaisessa ympäristössä, jossa on mukana luokat Employee ja Department. Osastolla voi olla useita työntekijöitä, mutta työntekijä kuuluu vain yhteen osastoon. Kummallakin on viite toisiinsa.

```

5. String baseDir = "C:/MySystem/data"
6. Department dep = new Department("Department 1");
7. Employee emp = new Employee("Larry", "King", 21, "200679-1121", dep);
8. dep.addEmployee(emp);
9.
10. Prevayler prevayler =
11.     PrevaylerFactory.createPrevayler(new MyPrevalentSystem(), baseDir)
12.     MyPrevalentSystem myPrevalentSystem =
13.         (MyPrevalentSystem) prevayler.prevalentSystem();
14.
15. AddDepartmentTransaction dt = new AddDepartmentTransaction(dep);
16. AddEmployeeTransaction et = new AddEmployeeTransaction(mick);
17.
18. prevayler.execute(dt);
19. prevayler.execute(et);
20.
21. Iterator it = myPrevalentSystem.getEmployees().iterator();
22. while(it.hasNext()) {
23.     Employee emp = (Employee)it.next(
24.     System.out.println(emp.getName());
25. }

```

### Esimerkki 4. Testiohjelma Prevayleria varten

Esimerkissä 4 on testiohjelma, jossa aluksi luodaan oliot työntekijälle ja osastolle, muodostetaan assosiaatiot niiden välille, sekä tallennetaan ja luetaan ne tietorakenteesta.

Riveillä 6 ja 7 PrevaylerFactory luo uuden ilmentymän Prevaylerista. Funktio createFactory tarvitsee uuden olion tietomallista (esimerkki 5, s. 8), jota ohjelma tulee käyttämään. Mikäli mallin transaktiologeja tai snap shot-

tiedostoja on olemassa, palauttaa funktio ohjelman tilan niiden mukaiseksi. Sen lisäksi funktiolle annetaan kansio, josta transaktiolokit ja mahdolliset snapshot-tiedostot löytyvät. Riveillä 8 ja 9 kutsutaan Prevaler-olion prevalentSystem-metodia, jolla createPrevaler-funktion konstruoimaan tietomalliin päästään käsiksi. Funktio createPrevaler tekee siis todella paljon töitä käyttäjän huomaamatta.

Testiohjelman riveillä 11 ja 12 muodostetaan kaksi transaktiota, joista ensimmäinen osaa lisätä järjestelmään osaston ja jälkimmäinen työntekijän (esimerkki 6). Riveillä 14 ja 15 suoritetaan transaktiot Prevalerin kautta, jotta ne kirjautuvat lokiin. Lopussa työntekijälista käydään läpi ja tulostetaan listan työntekijöiden nimet konsoliin.

---

```

1. public class MyPrevalentSystem implements Serializable {
2.
3.     private final Set departments;
4.     private final Set employees;
5.
6.     public MyPrevalentSystem() {
7.         employees = new HashSet();
8.         departments = new HashSet();
9.     }
10.
11.    public Set getDepartments() {
12.        return departments;
13.    }
14.
15.    public Set getEmployees() {
16.        return employees;
17.    }
18. }
```

---

### Esimerkki 5. Tietomalli

Esimerkin 5 tietomalli on hyvin yksinkertainen ja sisältää ainoastaan listat tallessa pidettäville työntekijöille ja osastoille. Jotta niihin pääsisi ulkoa päin käsiksi, pitää myös get-metodien olla mukana toteutuksessa.

---

```

1. public final class AddEmployeeTransaction implements Transaction {
2.
3.     private Employee employee;
4.
5.     public AddEmployeeTransaction(Employee employee) {
6.         this.employee = employee;
7.     }
8.
9.     public void executeOn(Object prevalentSystem, Date date) {
10.        MyPrevalentSystem myPrevalentSystem=
11.            (MyPrevalentSystem)prevalentSystem;
12.
13.        myPrevalentSystem.getEmployees().add(employee);
14.    }
15. }
```

---

### Esimerkki 6. Transaktio, jolla järjestelmään lisätään työntekijä

Transaktion on toteutettava joko Transaction- tai TransactionWithQuery-rajapinta. Työntekijän lisäävä transaktio esimerkissä 6 toteuttaa näistä vaihto-

ehdoista edellisen. Rajapinnan toteuttaminen edellyttää metodin `executeOn(Object, Date)` toteuttamista. Kun pääohjelmassa kutsutaan `prevayler.execute(AddEmployeeTransaction)`, ohjautuu metodin suoritus `executeOn()`-metodille. Osaston lisäävä transaktio on vastaavanlainen. `TransactionWithQuery:n executeAndQuery` on muuten `executeOn`-metodin kaltainen, paitsi että se osaa palauttaa arvon ja aiheuttaa poikkeuksen (`throw exception`). Työntekijän tapauksessa se voisi esimerkiksi palauttaa lisätyn työntekijän.

#### 2.2.4. Yhteenveto

Prevaylerin toteuttamaa menetelmää on kritisoitu muun muassa muistin valtavasta käytöstä varsinkin suuremmissa sovelluksissa [Prevayler, 2004], mutta kuten on jo todettu, ei muisti nykyään ole erityisen kallista. Myös eräs niistä asioista, joista järjestelmää voi kritisoida, on palautumisprosessin tehokkuus. Kymmenien tai satojen tuhansien transaktioiden ajaminen ei välttämättä ole erityisen nopeaa, mikäli järjestelmä kaatuu. Myöskään snapshot-tiedoston muodostaminen ei ole erityisen tehokasta, mikäli tallennettavaa tietoa on paljon. Tiedoston tekeminen kannattaa Prevaylerin tekijöiden mukaan ajoittaa tästä syystä esimerkiksi yöhön tai muuten hiljaiseen ajankohtaan.

Prevayler ei tietenkään sovellu kaikkeen tiedon tallentamiseen, mutta sen malli tarjoaa joissakin tapauksissa relaatiotietokantoja huomattavasti tehokkaamman ja yksinkertaisemman vaihtoehdon. Vastaavan menetelmän toteuttaminen ei ole erityisen mutkikasta, mutta koska Prevayler on avoimen lähdekoodin hyväksi havaittu sovellus, miksi tehdä sellaista uudestaan? Jos Prevayler ei tarjoa kaikkia niitä ominaisuuksia, joita käyttäjä mahdollisesti tarvitsee, on ominaisuudet mahdollista tarvittaessa kirjoittaa itse.

Paras sovellusalue sarjallistamisen tapaan Prevaylerille lienee pienemmät sovellukset, jotka toimivat paikallisesti. Relaatiotietokannan käyttö varsinkin isompien ja hajautettujen arkkitehtuurien taustalla on monessakin mielessä perustellumpaa kuin Prevaylerin.

### 2.3. Oliotietokannat

Joskus 1990-luvun alussa, kun oliopohjaisen sovelluskehityksen suosio alkoi nousta, tulivat myös laajempaan tietoisuuteen toisenlaiset, oliomallin mukaiset tietokantaratkaisut. Pian ajateltiin, että oliotietokannoista tulisi trendi [Fong 1997] ja että ne jättäisivät relaatiotietokannat varjoonsa oliopohjaisten sovellusten pysyvyyden toteuttamisen menetelmänä. Kuten tiedetään, ei näin ole käynyt, eikä useimpien mielestä todennäköisesti tule käymäänkään (muun muassa [Barry, 2004]). Kaikesta huolimatta on niiden perusteiden tunteminen hyödyllistä.

### 2.3.1. Oliomallista

Oliomallilla tarkoitetaan yleisesti tietyn kohdealueen mallia (domain model), joka on normaalisti esitetty UML-kaavioiden avulla (kuva 21, s. 57), mutta tässä oliomallista pyritään puhumaan yleisemmin ja erityisesti tietokannan näkökulmasta. Voidaanko tällaista oliomallia esittää relaatiomallin (alakohta 2.4.1) tavoin yhtenä yleisenä kokonaisuutena? Oliomalli kun määritellään eri osapuolten keskuudessa hieman eri tavoin, ja määritelmien mukaiset toteutukset vaihtelevat myös paljon. Näin ollen oliomallista ei ole ollut [Date, 1995], eikä edelleenkään ole olemassa yhtä ja kaikilta osin yleisesti hyväksyttyä, saati sitten matemaattisesti määriteltyä versiota.

Oliotietokantojen standardointia on yritetty toteuttaa määrittelemällä yleisiä linjoja yhteiselle kyselykielelle ja oliomallille. Eräs tällainen standardi, ODMG (object database management group) saatiin valmiiksi vuonna 2001 [ODMG, 2001]. Standardi ei käytännössä kuitenkaan ole erityisen käyttökelpoinen, eikä se ole saavuttanut laajaa suosiota, joten sitä ei käsitellä tässä tutkielmassa.

Daten [Date, 1995] mukaan puhuttaessa oliopohjaisuudesta, viitataan lähinnä kokoelmaan ideoista, jotka ovat sidoksissa toisiinsa. Termin monikäsitteisyttä lisää hänen mukaansa edelleen se, että termiä käytetään puhuttaessa tietyistä graafisista käyttöliittymistä, tietyistä ohjelmointityyleistä, tietyistä ohjelmointikielistä, tietyistä analyysi- ja suunnittelumenetelmistä ja niin edelleen. Purkaaksemme hieman tätä monikäsitteisyttä, hahmotellaan seuraavassa taulukossa hieman yleistä, oliotietokantakeskeistä mallia.

Taulukko 1. Oliomallin ominaisuuksia

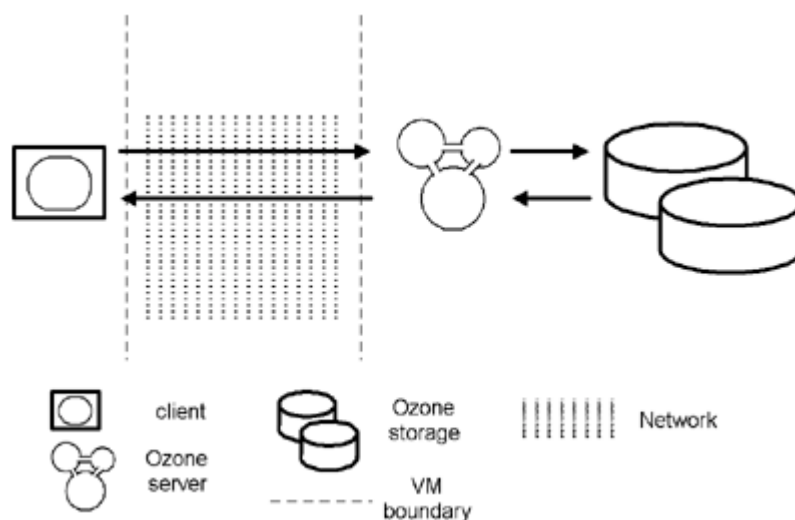
Ominaisuus	Selitys
Olio(object)	OODBMS käsittelee olioita. Oliot ovat rekursiivisia, eli ne voivat pitää sisällään toisia olioita. Olioilla on jokin tyyppi (luokka) ja ne ovat kapseloituja. Olio voi kuvata joko mitä tahansa reaali maailman asiaa tai abstraktia kokonaisuutta. Olio koostuu attribuuteista, joiden arvot muodostavat olion tilan. Olio sisältää funktioita, joiden avulla tilaa voidaan muuttaa.
Rakenne	Olioiden tulee voida olla assosiaatioissa keskenään, eli muodostaa rakenteita. OODBMS:n tulee pystyä kirjoittamaan oliograafin rakenne levyille ja lukemaan se levyiltä takaisin muistiin.
OID (object identity)	OODBMS:n tulee tarjota yksilöllinen ja muuttumaton tunniste kaikille tietokantaan tallennettaville olioille. OID ei näy käyttäjälle, vaan järjestelmä käyttää sitä erotellakseen oliot toisistaan.
Kapselointi	Olion sisäinen rakenne tulee voida piilottaa ja sen tilaa voida muuttaa vain tiettyjen metodikutsujen kautta. Käyttäjälle tulee tarjota rajapinta, joka sisältää tilaa muuttavien toimintojen otsikot (signature).
Nimeäminen (naming)	Kun olio tallennetaan, se pitää nimetä. Nimeämisessä oliolle annetaan yksilöllinen, pysyvä nimi, minkä kautta ohjelmat voivat saada

	olion käyttöönsä.
Tavoitettavuus (reachability)	Koska ei ole järkevää nimetä kaikkia olioita satunnaisesti, tulee nimeämisessä voida käyttää tavoitettavuutta. Olio B on tavoitettava oliosta A, mikäli joukko viittauksia oliograafissa johtavat A:sta B:hen.
Luokkien määrittelyminen	Uusia luokkia tulee voida määrittellä ja uusien luokkien määrittelyminen tulee onnistua olemassa olevien luokkien pohjalta.
Periytyminen	Kun toteutetaan uusi luokka, joka on osittain samankaltainen kuin jokin olemassa olevista, tulee siihen voida periä olemassa olevan luokan ominaisuudet. Perittävää luokkaa kutsutaan yliluokaksi (superclass) ja periytettyä luokkaa aliluokaksi (subclass).
Monimuotoisuus	Samoja operaattoreiden nimiä tai symboleita tulee voida sitoa useampiin erilaisiin funktioiden toteutuksiin. Esimerkiksi jos operaattoria "+" käytetään Integer-tyyppisten olioiden yhteydessä, kutsutaan toteutusta, joka laskee aritmeettisen yhteenlaskuoperaation. Jos sitä taas käytetään joukkojen yhteydessä, kutsutaan toteutusta, joka muodostaa joukoista unionin. Jos tyypit tiedetään ennalta, puhutaan aikaisesta sidonnasta (early binding). Jos tyypit tiedetään vasta ajon aikana, puhutaan myöhäisestä sidonnasta.

Yllä luetelluista ominaisuuksista suuri osa on peräisin Elmasrin ja Navathen [Elmasri ja Navathe, 2000] sekä Daten [Date, 1995] kirjoista, mutta mallia on muunneltu palvelemaan tämän tutkielman tarkoituksia. Taulukossa ei ole mainittu joitakin mahdollisia OODBMS:n ominaisuuksia, kuten moniperintää tai valikoivaa perintää, sillä esimerkiksi Java ei kielenä tue näitä perinnän tyyppejä, ja mallilla pyritään esittämään mahdollisimman yleisiä OODBMS:n ominaisuuksia.

### 2.3.2. Ozone

Ozone [Ozone, 2004] on ilmainen, avoimen lähdekoodin Java-pohjainen oliotietokanta. Ozone tukee transaktioita, hajautusta sekä käyttöoikeuksien hallintaa ja sitä voidaan ajaa erillisenä sovelluspalvelimena (kuva 2) tai upottaa paikallisesti toimivaan sovellukseen. Se ei toteuta mitään uusia olio- tai laskentamalleja eikä kyselykieliä. Ozonen tarkoituksena on, että ohjelmoija voi keskittyä tekemään puhtaita oliopohjaisia sovelluksia ympäristössä, jossa pysyvyyden toteutus on sovelluksen näkökulmasta piilotettu. Ozonen malli ei oletta, että oliot ja niiden sisältämä data olisivat erillään toisistaan, kuten esimerkiksi EJB-mallissa oletetaan. Lisäksi Ozone tukee datan tallentamista XML-muotoon [Braeutigam *et al.*, 2003]. Tässä kohdassa esitellään lyhyesti Ozonen käyttöä esimerkkien avulla.



Kuva 2. Ozone etäpalvelimena [Duchesne ja Nyfelt, 2001].

Ozone on Braeutigamin ja muiden [Braeutigam *et al.*, 2003] mukaan ”yhden instanssin arkkitehtuuri”. Tämä tarkoittaa sitä, että kustakin pysyvästä tietokantaoliosta on tietokantapalvelimeen tallennettu vain yksi ilmentymä, jota kontrolloidaan välittäjäolion (kuva 18, s. 52) kautta. Välittäjä edustaa oikeaa oliota, joten asiakassovellus käyttää sitä, kuten se käyttäisi varsinaista tietokannassa olevaa oliota.

Ozonea käytettäessä tehdään aluksi rajapinnat talteen kirjoitettaville olioille. Seuraavassa esimerkissä on määritelty rajapinta luokalle Car.

```

1. public interface Car extends OzoneRemote {
2.     public void setName( String name );      /*update*/
3.     public String name();
4.     public void setYearOfConst( int year ); /*update*/
5.     public int age();
6. }

```

Esimerkki 7. Luokan Car rajapinta [Braeutigam *et al.*, 2003]

Varsinaisia tietokantaolioita käsitellään Ozonessa rajapintojen kautta. Vain niillä metodeilla, joita rajapintaan on määritelty, voidaan käsitellä rajapinnan takana olevaa varsinaista tietokantaoliota. Rajapinnan pitää periä OzoneRemote-rajapinta, sekä kertoa, mitkä metodit ovat sellaisia, jotka muuttavat olion tilaa. Tilaa muuttavat oliot kerrotaan Ozonelle merkitsemällä metodin perään ”/\*update\*/”. Vaihtoehtoisesti voitaisiin käyttää tiettyä itse määriteltyä merkkijonoa funktion nimessä tai metatietoa luokasta sisältävää XML-muotoista OCD (Ozone class descriptor)-tiedostoa.

Esimerkissä 8 esitellään Ozonen mukainen Car-rajapinnan toteutus.

```

1. public class CarImpl extends OzoneObject implements Car {
2.     final static long serialVersionUID = 1L;
3.     private String _name;
4.     private int _yearOfConst;

```

---

```
6.     public CarImpl() {
7.         _name = new String( "" );
8.         _yearOfConst = 0;
9.     }
10.    public String name() {
11.        return _name;
12.    }
13.    public void setName( String name ) {
14.        _name = name;
15.    }
16.    public void setYearOfConst( int year ) {
17.        _yearOfConst = year;
18.    }
19.    public int age() {
20.        Calendar cal = Calendar.getInstance();
21.        return cal.get (Calendar.YEAR) - _yearOfConst;
22.    }
23. }
```

---

#### Esimerkki 8. Rajapinnan Car toteutus [Braeutigam *et al.*, 2003]

Kaikkien tietokantaolioiden pitää periä `OzoneObject` ja toteuttaa rajapinta, jonka kautta oliota käsitellään. `CarImpl` (esimerkki 8) on luokka, joka toteuttaa rajapinnan `Car`. Toteutusluokissa ei myöskään saa käyttää `this`-avainsanaa, vaan ne pitää korvata Ozonen funktiolla `self`. On myös huomattava, että kaikkien metodeille annettavien parametrien on oltava sarjallistuvia.

Seuraavaksi luokat pitää kääntää, jotta tietokantaluokille (tässä tapauksessa pelkälle luokalle `CarImpl`) voidaan myöhemmässä vaiheessa luoda Proxy-oliot (kuva 18, s. 52). Proxy-olioita eli välittäjiä käytetään sovelluksissa tietokantaolioiden korvikkeina. Niiden tehtävänä on välittää metodikutsut tietokantayhteyden yli tietokantaolioille. Niillä on sama rajapinta kuin varsinaisilla olioilla, joten niiden käyttäminen on samanlaista kuin varsinaisten. Välittäjät luodaan ohjelmalla "OPP", Ozone Post Processor. Se muodostaa halutuista luokista välittäjät, joiden etuliitteenä on luokan nimi, keskellä alaviiva ja lopuksi sana "Proxy". Esimerkin luokasta tulisi näin ollen `Car_Proxy.class`. Sen lisäksi OPP luo tarvittavat Factory-luokat.

Seuraavassa esimerkissä näytetään, miten näin luotua tietokantaa voidaan käyttää.

---

```
1. ExternalDatabase db =
2. ExternalDatabase.openDatabase("ozonedb:remote://localhost:3333" );
3.
4. db.reloadClasses();
5.
6. Car car = (Car)(db.createObject( CarImpl.class.getName(), 0,
7.                               "my_first_car" ));
8. car.setName( "gottfried" );
9. car.setYearOfConst( 1957 );
10.
11. Car car = (Car)(db.objectForName( "my_first_car" ));
12. if (car != null) {
13.     db.deleteObject( car );
14. } else {
15.     System.out.println( "Object my_first_car not found." );
16. }
17. db.close();
```

---

### Esimerkki 9. Ozonen käyttäminen

Aluksi luodaan yhteys tietokantapalvelimeen. Sen jälkeen tietokantaluokat ladataan muistiin. Tämä on välttämätöntä sen vuoksi, mikäli luokkakoodia olisi muutettu ja palvelinta ei olisi käynnistetty muutosten jälkeen uudestaan, eivät muutokset olisi voimassa. Toisin sanoen ilman kyseistä metodikutsua olisi palvelimen välimuistissa vanhat binääriluokat. Tämän operaation jälkeen luodaan uusi auto-olio Ozonen `createObject`-metodin avulla. Kun auto on luotu, voidaan sen tilaa muuttaa, eli asettaa olion attribuuteille arvoja niiden funktioiden avulla, jotka on edellä merkitty tilaa muuttaviksi funktioksi. Muutosten tallentamisesta ei erikseen tarvitse huolehtia.

Kun jokin pysyvä olio halutaan hakea kannasta takaisin muistiin, etsitään oliota graafista olion nimen perusteella `objectForName`-funktion avulla. Jos sitä ei löydy, palautuu `null`. Esimerkissä, mikäli olio löytyy, poistetaan se kokonaan muistista. Tällöin se poistuu myös tietokannasta. Lopuksi sovellus käännetään ja sitä pitää ajaa Ozonen oman virtuaalikoneen avulla (Ozone Java Virtual Machine).

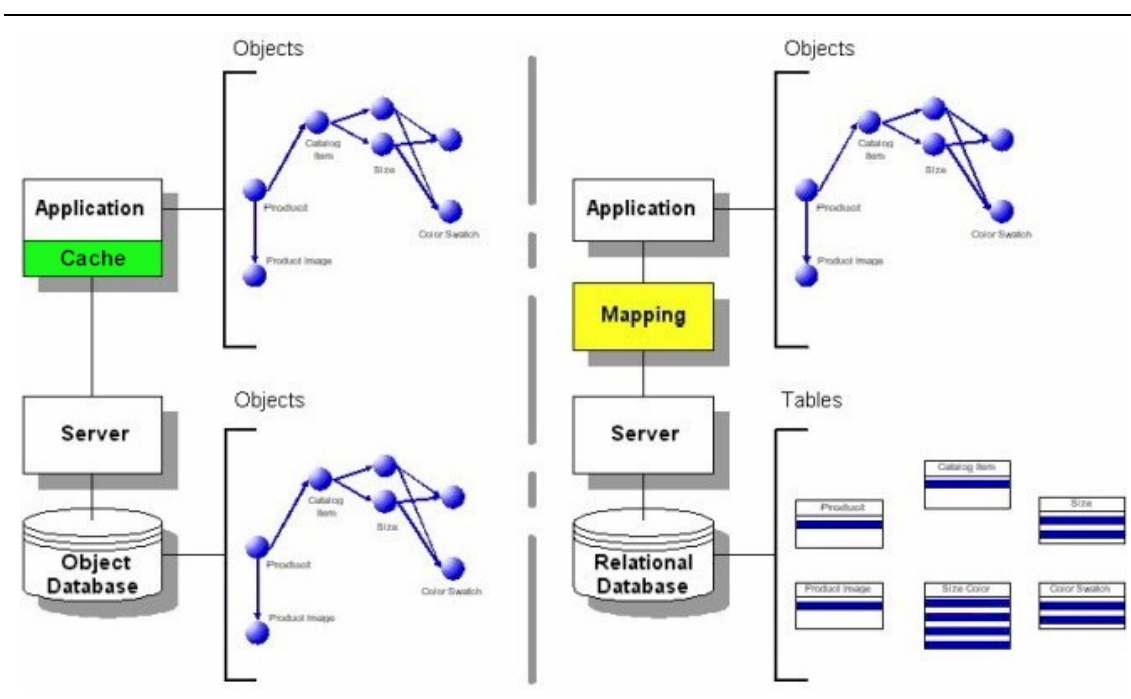
Kun siis olion luomiseksi käytetään tiettyä Ozonen tarjoamaa tehdas (factory)-metodia (`createObject`), luo Ozone toteutusluokan (`CarImpl`) olion palvelimen päähän, sekä välittäjäolion asiakkaan päähän. Nämä oliot toimivat ajon aikana yhteistyössä keskenään.

#### 2.3.3. Perusteluita oliotietokannan käyttämiselle

Oliotietokannan käyttämiselle voidaan löytää useita järkevän tuntuksia perusteita. Eräs argumentti oliotietokantojen puolesta on muun muassa se, että oliopohjaisen sovelluksen ja oliotietokannan välillä ei parhaimmillaan esiinny samankaltaista yhteensopimattomuutta kuin oliopohjaisen sovelluksen ja relaatiotietokannan välillä. Tämän lisäksi eräs oliotietokantojen avainominaisuus on, että ne sallivat sekä oliomallin rakenteen, että olioihin

sovellettavien operaatioiden määrittämisen tietokantaan [Elmasri ja Navathe, 2000]. Relaatiotietokantaa on sitä vastoin voitu yleensä käyttää vain säiliönä datalle. Useimmat nykyisistä relaatiotietokantojen toteutuksista tosin mahdollistavat joidenkin oliotietokantamaisten ominaisuuksien, kuten käyttäjän määrittelemien tietotyyppien tai proseduurien, käyttämisen.

Kun oliopohjaista sovellusta käytetään relaatiotietokantojen kanssa, vaaditaan sovelluksen arkkitehtuuriin lisäkerros, joka poistaa yhteensopivuusongelmat (kuva 3). Tätä kerrosta käsitellään luvusta 3 alkaen otsikolla ORM (object relational mapping). Vastaavaa kerrosta ei Barryn [Barry, 2004] mukaan tarvita, mikäli käytetään oliotietokantaa.



Kuva 3. "Yhteensopivuusongelmaa ei ole oliotietokantoja käytettäessä." [Barry, 2004]

Oliotietokannoilla pyritään Prevalyerin tavoin saavuttamaan läpinäkyvyyttä pysyvyyden suhteen, eli sitä, että pysyvien olioiden käsitteleminen ei poikkeaisi pelkästään muistissa olevien olioiden käsittelemisestä. Näin ollen SQL:n kaltaisten kyselykielten, sekä ODBC:n (Open Database Connectivity), ADO:n (ActiveX Data Objects) tai JDBC:n (Java Database Connectivity) kaltaisten matalan tason rajapintojen käyttäminen ei ole tarpeellista. Oliotietokannoissa tietoa sen sijaan käsitellään hakemalla aluksi tietty juuri tietorakenteeseen, joka esitetään tyypillisesti graafina, vektorina, hajautustaulukkona tai joukkona. Tämän jälkeen haluttuun tietoon päästään käsiksi navigoimalla tietorakenteessa [Obasanjo, 2001]. Kyselykielen, OQL:n (object query language) käyttäminen tosin on mahdollista joissakin oliotietokantaratkaisuissa, mutta periaatteessa puhtaassa oliopohjaisessa sovelluksessa, joka käyttää puhdasta oliotietokantaa, ei esimerkissä 10 esitetyn

kaltainen ohjelmointi ole mielekästä oliopohjaisten sovellusten ideologian kannalta, sillä kuten huomataan, ei tyyli poikkea olennaisesti esimerkiksi suorasta JDBC:n käyttämisestä.

---

```

1. OQLQuery query = new OQLQuery(
2.     "select x from Person x where x.name = \"Doug Barry\"");
3. Collection result = (Collection) query.execute();
4. Iterator iter = result.iterator();

```

---

#### Esimerkki 10. OQL-kysely [Barry, 2004].

Oliotietokannan käyttämistä voidaan ehkä parhaiten perustella luettelemalla syitä, miksi ei välttämättä kannata käyttää relaatiotietokantaa. Seuraavassa kohdassa esiteltävän oliotietokannan Ozonen tekijöiden löytämiä relaatiotietokantojen heikkouksia, jotka ovat motivoineet heitä puhtaan oliotietokannan toteuttamiseen, on lueteltu seuraavassa listassa [Braeutigam *et al.*, 2003]:

1. SQL:n upottaminen ohjelmakoodiin on äärimmäisen rumaa ja vaikeasti ylläpidettävää.
2. Kääntäjän on mahdotonta testata SQL:n syntaksia.
3. Relaatiotietokantaa käytetään tyypillisesti siten, että luodaan SQL-lause, joka palvelimen pitää parsia ja prosessoida. Lopuksi palvelin lähettää kyselyn tulokset takaisin. Tämä ei ole tehokasta.
4. Kun dataa muutetaan useissa paikoissa, pitää muodostaa monimutkaisia SQL-lauseita.
5. Jos lause tai transaktio epäonnistuu, saatetaan joutua lukemaan uudestaan vanhaa dataa RDBMS:stä, jotta sama data saataisiin takaisin muistiin. Tämä on selkeä synkronointiongelma.
6. Oliot tarvitsee kuvata relaatioiksi ja takaisin.
7. Jos ohjelmakoodia halutaan suorittaa palvelimen puolella, pitää käyttää jotain toimittajakohtaista kieltä proseduurien, makrojen tms. ajamiseen.
8. SQL kysely saattaa joskus palauttaa odottamattoman tuloksen. Esimerkiksi jos Oracleen tallennetaan JDBC:n avulla tyhjä merkkijono (`java.lang.String ("")`) ja haetaan se myöhemmin, saadaan takaisin merkkijono, joka on arvoltaan null.

Totuus on kuitenkin se, että relaatiotietokantoja käytetään useimmissa projekteissa huolimatta oliotietokantojen tarjoamasta vaihtoehdosta. Fowlerin [Fowler, 2003] mielestä suurin syy siihen, että oliotietokantoja ei haluta käyttää, on riski, joka niiden käyttöön liittyy. Relaatiotietokannat kun ovat olleet

järjestelmien taustalla jo pitkään ja niiden käytöstä on runsaasti kokemusta. Lisäksi relaatiotietokantojen taustalla on matemaattinen malli (alakohta 2.4.1). Fowler mainitsee myös, että SQL tarjoaa yhteisen standardirajapinnan kaikenlaisille työkaluille. Oliotietokannat eivät taas muodosta läheskään niin selkeää ja yhtenäistä ryhmää kuin relaatiotietokannat. Eri osapuolet määrittelevät oliomallin eri tavoin, mistä taas seuraa, että tavat määrittää oliotietokannan rakenne poikkeavat radikaalistikin toisistaan. McFarlandin ja muiden [McFarland *et al.*, 1999] mukaan oliotietokantojen rakenne-eroista johtuen kannan valinta riippuu kuitenkin aina sovellusalueesta, jossa sitä aiotaan käyttää.

## 2.4. Relaatiotietokannat

Tässä kohdassa käydään aluksi lyhyesti läpi relaatiomallin peruskäsitteitä ja ominaisuuksia. Kohdan tiedot perustuvat pääsääntöisesti Elmasrin ja Navathen [Elmasri ja Navathe, 2000] sekä Daten [Date, 1995] kirjoihin, joissa tulkitaan Codd'n [Codd, 1970] kehittämää relaatiomallia. Lisäksi apuna on käytetty Niemen [Niemi, 2003] luennoiman tietokantoja käsittelevän kurssin luentomateriaalia. Toisessa kohdassa kerrotaan JDBC (Java Database Connectivity)-ajureista ja relaatiotietokannan käyttämisestä JDBC-ohjelmointirajapinnan (Application Interface, API) avulla.

### 2.4.1. Relaatiomalli

IBM:n tutkimusosastolla työskennellyt Edgar F. ("Ted") Codd julkaisi joukkooppiin ja ensimmäisen kertaluvun predikaattilogiikkaan pohjautuvan relaatiomallin vuonna 1970 ACM:n julkaisemassa artikkelissa "A Relational Model of Data for Large Shared Data Banks" [Codd, 1970]. Malli saavutti heti suuren suosion sen yksinkertaisuuden ja matemaattisen täsmällisyyden ansiosta.

Relaatiomalli esittää tietokannan kokoelmana relaatioita. Relaatio muistuttaa taulukkoa, jonka soluissa on arvoja. Kukin taulukon rivi esittää joukon arvoja, jotka ovat suhteessa keskenään. Rivin semanttinen tulkinta on usein jokin todellinen kohde tai suhde. Yksi rivi voi esimerkiksi edustaa yhtä työntekijää. Tietokannan taulu ja relaatiomallin relaatio ovat joukko-opillisesti eri asioita, sillä taulussa voi olla samoja rivejä ja riveillä on tietty järjestys. Relaatiossa sama rivi voi esiintyä vain kerran ja joukon alkioiden järjestyksellä ei ole merkitystä.

Formaalissa relaatiomallin terminologiassa riviä vastaavaa käsitettä kutsutaan monikoksi (tuple), sarakkeen otsikkoa vastaavaa käsitettä attribuutiksi ja taulua vastaavaa käsitettä relaatioksi. Käytännössä attribuutilla tarkoitetaan siis saraketta. Tietotyyppiä, joka kuvaa sarakkeessa esiintyvien arvojen tyyppiä, kutsutaan arvoalueeksi (domain). Arvoalue koostuu joukosta atomisia arvoja. Atomisella tarkoitetaan sitä, että kukin arvoalueen arvo on

jakamaton. Arvoalue määritellään tyypillisesti määrittelemällä tietotyyppi, johon arvojen on kuuluttava. Jos arvoalueen atomisuus toteutuu, on tietokanta ensimmäisessä normaalimuodossa. Esimerkki arvoalueen loogisesta määrittelystä voisi olla vaikkapa työntekijöiden nimien muodostama joukko, jota voitaisiin kutsua esimerkiksi nimellä "employee\_names".

Relaation nimi ja sen attribuutit muodostavat relaation kaavion (schema). Relaatiolla on myös asteluku, joka on sama kuin sen attribuuttien määrä. Alla olevan esimerkin yleisen kaavion asteluku olisi näin ollen  $n$  ja kaavion "employees" puolestaan 2.

Esimerkki kaaviosta: employee (name, age)

Esimerkki yksittäisestä relaatiosta: employees {<Joe, 25>, <John, 34>}

Relaation rivit yksilöivää attribuuttien joukkoa kutsutaan superavaimeksi. Kaikki sellaiset attribuuttien joukot, jotka yksilöivät relaation rivit, ovat siis superavaimia. Avain on sellainen superavain, josta ei voi poistaa yhtäkään attribuuttia menettämättä superavain-ominaisuutta. Jos avaimia on useita, valitaan jokin pääavaimeksi (primary key). Relaatioita kuvataan yleisesti seuraavasti:

Relaation kaavio:  $R(A_1, A_2, \dots, A_n)$

Relaatiokaavion instanssi:  $r(R) = \{t_1, t_2, \dots, t_m\}$

Yksittäinen relaatio  $r(R)$  on joukko  $n$ -monikoita ( $n$ -tuples)  $r = \{t_1, t_2, \dots, t_m\}$ . Kukin  $n$ -monikko  $t$  on järjestetty  $n$ :n alkion joukko  $t = \langle v_1, v_2, \dots, v_n \rangle$ , missä kukin arvo  $v_i$  ( $1 \leq i \leq n$ ) on arvoalueen  $\text{dom}(A_i)$  alkio tai erityinen tyhjäarvo (null). Null edustaa sellaista attribuuttia, jonka arvo on tuntematon tai arvoa ei ole olemassa tietyllä rivillä [Elmasri ja Navathe, 2000]. Monikon sisältämien attribuuttien keskinäisellä järjestyksellä on siis tämän määritelmän mukaan merkitystä, mutta monikkojen keskinäisellä järjestyksellä ei ole matemaattisen relaatiomallin kannalta merkitystä.

Vaihtoehdoisen määritelmän mukaan monikon sisältämien attribuuttien keskinäisellä järjestyksellä ei ole merkitystä. Vaihtoehdoisessa määritelmässä relaation kaavio merkitään joukoksi attribuutteja  $R = \{A_1, A_2, \dots, A_n\}$  ja yksittäinen relaatio  $r(R)$  on äärellinen joukko kuvauksia  $r = \{t_1, t_2, \dots, t_m\}$ , missä  $t_i$  on kuvaus joukolta  $R$  joukkoon  $D$ , joka taas on attribuuttien arvoalueiden unioni, eli  $D = \text{dom}(A_1) \cup \text{dom}(A_2) \cup \dots \cup \text{dom}(A_n)$ . Tässä määritelmässä rivi voidaan tulkita joukoksi nimi-arvo pareja, missä kukin pari ilmoittaa kuvauksen arvon attribuutilta  $A_i$  arvoon  $v_i$  arvoalueesta  $\text{dom}(A_i)$ . Näin ollen järjestyksellä ei ole merkitystä, sillä nimi ja arvo esiintyvät aina yhdessä [Elmasri ja Navathe, 2000].

Arvoalueet ovat siis joukkoja, kuten esimerkiksi kokonaislukujen joukko. Ensimmäisen määritelmän mukaan yksittäiset relaatiot koostuvat joukoista järjestettyjä joukkoja. Tällöin esimerkiksi yksittäisen relaation sisältämän järjestetyn joukon  $t$  alkio  $v_2$  kuuluu arvoalueeseen  $\text{dom}(A_2)$  ja jos  $\text{dom}(A_2)$  viittaa kokonaislukuihin, on myös  $v_2$  jokin kokonaisluku. Näin ollen yksittäinen relaatio  $r(R)$  on  $n:n$  asteen matemaattinen relaatio arvoalueiden  $\text{dom}(A_1), \text{dom}(A_2), \dots, \text{dom}(A_n)$  välillä, mikä on  $R:n$  määrittämien arvoalueiden karteesisen tulon osajoukko:

$$r(R) = U (\text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n))$$

Edellä mainittujen asioiden pohjalta on helppoa päätellä, että relaatiotietokanta määritellään joukoksi relaatioita ja relaatiotietokannan kaavio muodostuu joukosta relaatiokaavioita.

#### 2.4.2. Relaatioalgebra ja SQL

Relaatiotietokantaan voidaan suorittaa kyselyitä. Relaatioalgebra on kyselykieli, jolla kyselyt suoritetaan. Se sisältää omien operaatioidensa lisäksi myös joukko-opin operaatiot, jotka ovat unioni, leikkaus, erotus ja karteesinen tulo. Merkittävimmät relaatioalgebran omat operaatiot ovat valinta (select), jossa valitaan vain tietyt rivit, projektio (projection), jossa valitaan tietyt attribuutit ja liitos (join), jossa yhdistetään eri relaatioissa olevaa dataa. Kaikki relaatioalgebran operaatiot ottavat syötteenä aina relaatioita ja tulosteena on aina relaatio, eli operaatiojoukko on suljettu. Tämän vuoksi operaatioita voidaan ketjuttaa aritmeettisten operaatioiden tavoin [Niemi, 2003].

Käytännön sovellukset lähes mistä tahansa asioista rikkovat usein teorioita ja niin myös SQL, jonka IBM alun perin kehitti soveltamaan Codd'n relaatioalgebraa. Oracle esitteli maailman ensimmäisen kaupallisen IBM:n SQL:n toteutuksen vuonna 1979, jonka jälkeen niin tekivät myös monet muut toimittajat. Vuonna 1986 muodosti ANSI virallisen SQL-standardin ja sitä seuraavana vuonna ISO ratifioi sen. Standardiin on sen jälkeen tullut muutoksia vuosina 1989, 1992, 1999 ja 2003. Osaa muutoksista ei kuitenkaan tueta laajalti, eikä varsinkaan kiistanalaisia oliopohjaisia ominaisuuksia, joita standardiin lisättiin vuonna 1999.

Useimmat toimittajat toteuttavat standardin epätäydellisesti ja määrittelevät omissa kielissään runsaasti epästandardeja ominaisuuksia. Toimittajakohtaiset erot johtuvat pitkälti standardin monimutkaisuudesta ja laajuudesta, mutta toisaalta myös sen puutteellisuudesta: Standardi jättää joitakin tärkeitä ominaisuuksia kokonaan määrittelemättä, kuten indeksien toteutuksen. Eräs vitsi kielestä kuuluukin seuraavasti: "SQL is neither structured, nor a language" [Wikipedia, 2004a]. Vitsillä on totuuspohjansa, sillä

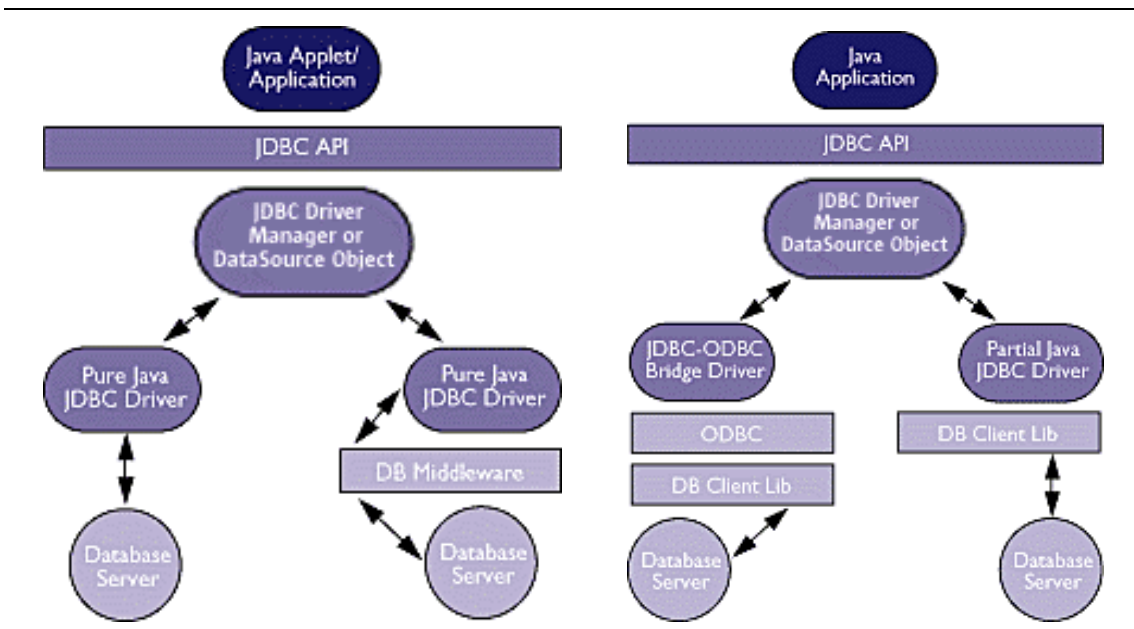
puhdas SQL ei ole aito ohjelmointikieli (vaikka se kylläkin on kieli), koska se ei ole Turing-täydellinen.

### 2.4.3. JDBC

JDBC (Java Database Connectivity) on Sunin kehittämä Java-ohjelmoiden keskuudessa erittäin tunnettu ohjelmointirajapinta (API), joka toimii siltana Java-sovellusten ja useimpien tietokantojen välillä. JDBC API mahdollistaa kolme asiaa: Yhteyden ottamisen relaatiotietokantaan tai mihin tahansa muuhun taulukkomaiseen tietolähteeseen, SQL-kyselyiden lähettämisen tietokantaan, sekä kyselyiden tulosten vastaanottamisen ja prosessoinnin [JDBC, 2004].

API voidaan jakaa kahteen osaan; se määrittelee erillisen rajapinnan sovelluskehittäjille ja matalamman tason rajapinnan ajureiden kirjoittajille [JDBC, 2004]. Tietokantaspesifinen ajuri tehdään näin ollen toteuttamalla luokkakirjaston tietyt rajapinnat. JDBC-ohjelmointirajapintaa käyttävä asiakas voi siten käyttää mitä tahansa tietokantaa, jolle nuo rajapinnat on toteutettu.

JDBC-ajurit voidaan jakaa kahteen kategoriaan: sovelluskehittäjille suunnattuihin ja tietokanta- tai "middleware"-sovellusten toimittajille suunnattuihin ajureihin [Sliwa, 1999]. Kaikille ajureille yhteistä on se, että ne konvertoivat sovelluksessa tehtyjä JDBC-komentoja johonkin muotoon. Ajureiden jako mainittujen kategorioiden alle edelleen neljään ryhmään perustuu siihen, mihin muotoon konversio tehdään (kuva 4).



Kuva 4. JDBC-ajureiden tyypit [JDBC, 2004]

Tyyppin 1 ajuria kutsutaan JDBC-ODBC (Open Database Connectivity) sillaksi. Sen tyyppisen ajurin avulla voidaan ottaa yhteys tietokantaan yhden tai useamman ODBC-ajurin kautta. Ajuri muuntaa sovelluksen JDBC-pyyntöt

ODBC-pyyntöiksi ja takaisin. Kuvan 1 oikean puoleisen kaavion vasemmalla puolella havainnollistetaan tämän tyyppisen ajurin toimintaa. Tyypin 2 ajuria sanotaan osittaiseksi Java ajuriksi. Ajuri konvertoi JDBC-kutsut asiakaskoneen tietyn tietokannan API-kutsuiksi. Tämä vaihtoehto nähdään oikean puoleisen kaavion oikealla puolella [Sliwa, 1999].

Tyypin 3 ajuria sanotaan puhtaaksi Java ajuriksi "middlewareelle". Tämän tyyppisen ajurin toimintaa havainnollistetaan vasemmanpuoleisen kaavion oikealla puolella. Ajuri muuntaa JDBC-kutsut middleware-toimittajan protokollan mukaisiksi. Middlewaren vastuulla on sitten muuntaa kutsut tietokantaspesifiseen muotoon. Neljännen tyypin ajuria kutsutaan aidoksi Java-ajuriksi, joka käyttää suoraan tietokantaa. Tällainen ajuri muuntaa JDBC-kutsut suoraan paketeiksi, jotka lähetetään verkon yli tietyn tietokannan protokollan mukaisesti. Ajuri mahdollistaa suoran kutsun tekemisen asiakaskoneelta tietokantaan [Sliwa, 1999].

Tietokantatoimittajat määrittelevät ajureissaan tietysti myös omia rajapintojaan, jotka laajentavat varsinaista JDBC-kirjastoa. Avoimen lähdekoodin PostgreSQL-tietokannan ajuri määrittelee esimerkiksi rajapinnan PGConnection, mikä laajentaa Java-API:n Connection-luokan rajapintaa. PostgreSQL:n JDBC-ajuri palauttaa pelkästään sellaisia Connection-olioita, jotka toteuttavat rajapinnan PGConnection.

Käytännössä JDBC-ajureita on kirjoitettu jo lähes kaikille tunnetuille tietokannoille mukaan lukien kaikki Microsoftin määrittelemän ODBC:tä käyttävät tietolähteet [Sliwa, 1999], joten niiden itsenäinen toteuttaminen ei juuri koskaan ole tarpeellista. JDBC API:n käyttäminen sovelluskoodissa on sen sijaan huomattavasti yleisempää, joskin ei enää nykyään erityisen suosittua, sillä pysyvyyden toteuttamiseksi Java-sovelluksissa on kehitetty useita muitakin menetelmiä, joista osaa onkin esitelty jo edellä ja vielä eräs tärkeä menetelmä, eli ORM ja erityisesti Hibernate, on vielä käsittelemättä. Hibernate käyttää kuitenkin JDBC:tä, joten sen toiminnan tunteminen on tärkeää Hibernaten syvällisen ymmärtämisen kannalta. Seuraavassa on vielä yksinkertaisia esimerkkejä JDBC:n käytöstä.

---

```

1. DataSource ds = (DataSource) envContext.lookup("jdbc/ora10g");
2. Connection conn = ds.getConnection();
3.
4. . . .
5.
6. String INSERT = "insert into employees (NAME) values (?)";
7. PreparedStatement pstmt = conn.prepareStatement(INSERT);
8. pstmt.setString(1, "John Smith");
9. pstmt.executeUpdate();
10. pstmt.getGeneratedKeys();

```

---

Esimerkki 11. JDBC-esimerkkejä

Yllä olevan esimerkin ylemmässä listauksessa haetaan tietokantayhteys tietokantayhteyksien säiliöstä (connection pool) JNDI:n (Java Naming and Directory Interface) avulla. Alemmassa suoritetaan tietokantapäivitys muodostamalla aluksi SQL-lause, jolle voi antaa yhden parametrin. PreparedStatement-luokan setString-metodilla parametri asetetaan ja executeUpdate-metodilla ajetaan varsinainen päivitys. Metodi getGeneratedKeys on tullut mukaan JDBC-versioon 3.0, jossa määritellään, että avain tulee palauttaa INSERT-operaation yhteydessä ilman ylimääräisen SELECT-lauseen suorittamista [Ashmore, 2004]. Kaikki tietokantatoimittajat eivät kuitenkaan ainakaan toistaiseksi tue tätä ominaisuutta.

Puhtaan JDBC-koodin sijaan on ohjelmakoodiin ollut myös mahdollista upottaa niin sanottua staattista SQL:ää. Tätä tarkoitusta varten on kehitetty SQLj-standardi, josta ainakin Oraclella on ollut toteutus. SQLj perustuu esiprosessointiin; ennen ohjelman kääntämistä ajetaan SQLj käännösohjelma, joka korvaa SQLj lauseet JDBC API:n mukaisiksi lauseiksi. SQLj:n etuna on ollut muun muassa se, että SQL:n syntaksi tarkastetaan sovelluksen kääntämisen yhteydessä. Selkeä haitta taas on ainakin se, että malli antaa mahdollisuuden toteuttaa vaikeaselkoista ja hankalasti testattavaa ohjelmakoodia. SQLj:n käyttäminen ei olekaan yleistynyt Java-yhteisössä, minkä vuoksi ainakaan Oracle ei enää tue sitä uusimmissa tietokantaversioissaan [Wang, 2004]. DB2 puolestaan tarjoaa edelleen tuen SQLj:n käyttämiselle uusimmissa tietokannoissaan.

### 3. ORM

Edellisessä luvussa on käsitelty sarjallistamista, Prevyleria, oliotietokantoja, sekä relaatiotietokantojen suoraa käsittelyä JDBC:n avulla Java-sovelluksen pysyvyyskerroksen toteuttajina. Tästä luvusta eteenpäin keskitytään myös relaatiotietokantoihin, mutta tapa käyttää niitä muuttuu merkittävästi. Abstraktiotasoa nostetaan piilottamalla tietokanta mahdollisimman näkymättömäksi. Oliotietokantoja ei haluta käyttää, mutta tietokannan käsittelyssä halutaan kuitenkin soveltaa oliotietokantamaista tyyliä, eli pyritään luomaan eräänlainen virtuaalinen oliotietokanta [Wikipedia, 2004b]. Tähän vaativaan haasteeseen pyritään vastaamaan ORM:n, eli olioiden ja relaatioiden välisen yhteyden kuvaamisen avulla.

#### 3.1. Taustaa

ORM ei ole erityisen uusi idea, vaan olioiden ja relaatioiden yhteistoimintaa on tutkittu jo useita vuosia. Eräs suhteellisen vanha aihetta käsittelevä tutkimus on Blahan ja muiden [Blahan *et al.*, 1988] julkaisema artikkeli, jossa käsitellään relaatiotietokannan suunnittelua olipohjaisia menetelmiä käyttäen. Tutkimuksessa käydään läpi useita ORM:n osa-alueita, kuten luokkien välisten assosiaatioiden kuvaamista relaatiotietokantaan.

Myöhemmin muun muassa Fong [Fong, 1995] on kuvannut tutkimuksessaan menetelmää, jossa EER (enhanced entity relationship)-malli liitetään OMT (object modeling technique)-malliin, sekä Fahl ja Risch [Fahl ja Risch, 1996] ovat esitelleet menetelmää, jossa relaatiotietokantaa voidaan käsitellä olionäkymän kautta. Myös Brown ja Whitenack [Brown ja Whitenack, 1996] kehittivät suunnittelumalleja, joiden avulla saatiin kuvattua Smalltalkin olioita relaatiotietokannan tauluihin. Mallit kuvataan tarkemmin heidän kehittämänsä Crossing Chasms-suunnittelukielen spesifikaatiossa.

ORM on kuitenkin levinnyt laajemman käyttäjäkunnan keskuuteen vasta viime vuosina osittain ehkä siksi, että menetelmän toteuttamista varten on tullut saataville hyvin toteutettuja kehyksiä, kuten esimerkiksi myöhemmin esiteltävä Hibernate, jotka ovat tehneet menetelmän käyttöönotosta helpompaa kuin aiemmin.

Date [Date, 1995] sanoo teoksessaan seuraavaa: "Jotkin sellaiset ominaisuudet, joita tarvitaan relaatiotietokannoissa, ovat olleet jo kauan oliopohjaisissa ohjelmointikielissä. Näin ollen oliotietokantojen tutkimus on luonnollista, mutta siitä ei kuitenkaan seuraa, että oliotietokannat korvaisivat relaatiotietokannat. Sitä vastoin meidän tulisi etsiä jotain näiden kahden mallin välimuotoa, eli tapaa, jolla nämä kaksi mallia voitaisiin yhdistää toistensa kanssa siten, että molemmista teknologioista saataisiin irti niiden parhaimmat

puolet.” Date ehdottaa menetelmää, josta hän käyttää nimeä OO/RR (object orientation/relational rapprochement). Siinä itse tietokanta toteutettaisiin molemmat mallit huomioiden, mutta malli pohjautuisi pääasiassa kuitenkin relaatiomalliin. ORM:ssä ei kuitenkaan ole kyse Daten näkemyksen mukaisesta ”hybriditietokantamallista”, jossa sekä olio- että relaatiomallin tarjoamat hyödyt pyritään saavuttamaan muuttamalla tietokannan rakennetta. ORM:n käyttäminen ei siis edellytä tällaisia muutoksia, vaan tietokanta voi edelleen olla puhdas relaatiotietokanta ja sovellus puhdas oliopohjainen sovellus. Siitäkin huolimatta molemmista malleista voidaan saada niiden tarjoamat edut.

### 3.2. Olio- ja relaatioparadigmojen välinen yhteensopivuusongelma

ORM:n toteuttamiseksi käytännössä on ratkaistava useita ongelmia. Eräänä ongelmana on muun muassa se, että todelliset järjestelmät toteuttavat teoreettiset mallit epätäydellisesti. Esimerkiksi relaatiotietokannat eivät ole noudattaneet niihin liittyviä teorioita täydellisesti koskaan. Toisaalta taas, kun oliomallinnusta ei ole standardoitu, toteuttaa kukin ympäristö oliomallin omalla tavallaan, mikä on myös omiaan aiheuttamaan lisävaikeuksia relaatio- ja oliomallin integroimisen välillä.

Ambler [Ambler, 2000] ottaa puolestaan esille tärkeän näkökulman, jonka mukaan relaatiotietokannat pohjautuvat matemaattisiin teoreettisiin malleihin (alakohta 2.4.1), kun taas olioparadigma pohjautuu enimmäkseen tietojenkäsittelyn periaatteisiin, kuten kapselointiin, riippuvuuksien minimointiin ohjelman osien välillä (coupling) ja koossapysyvyyteen (cohesion) (taulukko 1, s. 10). Kun ohjelmistotekniikan pyrkimykset, kuten uudelleenkäytettävyys, ja matemaattiset teoriat, kuten joukko-oppi kohtaavat, on yhteyden muodostaminen näiden mallien välille ymmärrettävästi haasteellista. Ambler toteaa lisäksi seuraavaa: ”Olioparadigmassa olioiden välillä kuljetaan suhteiden kautta kun taas relaatioparadigmassa dataa joudutaan kaksinkertaistamaan, jotta taulujen rivejä voidaan liittää toisiinsa. Tämä perustavaa laatua oleva eroavaisuus ei luonnostaan johda näiden kahden mallin ideaaliseen kombinaatioon.”

Olio- ja relaatiomallien yhteydessä keskustellaankin yleisesti yhteensopivuusongelmasta (object/relational paradigm mismatch). Nämä ongelmat on koottu lyhyine selityksineen seuraavaan taulukkoon. Taulukko perustuu Bauerin ja Kingin [Bauer ja King 2004] löytämiin mallien eroihin.

Taulukko 2. Paradigmojen yhteensopimattomuuden osatekijät

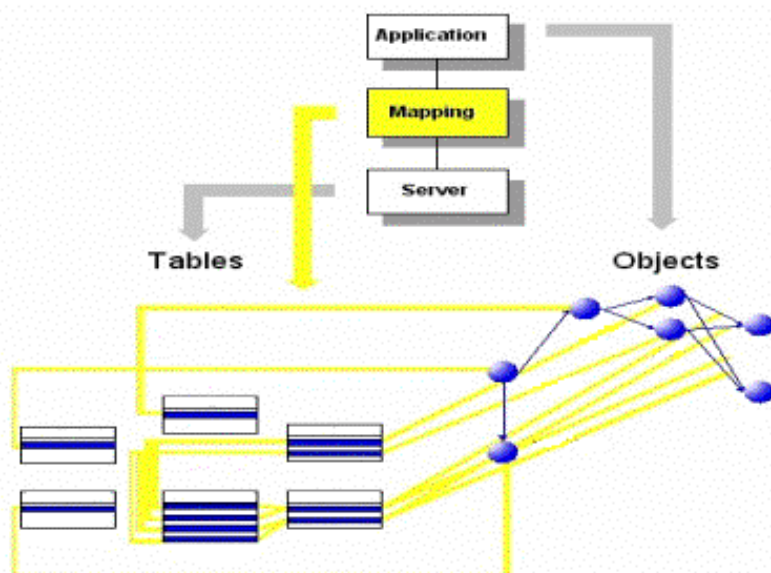
Ongelma	Selitys
Granulariteetti	Hyvillä olio- ja relaatiomalleilla on erilainen granulariteetti (tiedon karkeusaste ts.abstraktiotaso).
Periytyminen	Relaatiotietokannat eivät tarjoa suoraa tukea

	periytymiselle eivätkä siten myös <i>polymorfisille assosiaatioille</i> (luokan X polymorfinen assosiaatio abstraktin luokan Y kanssa tarkoittaa sitä, että X voi ajon aikana olla assosiaatiossa minkä tahansa Y:n aliluokan kanssa) tai <i>polymorfisille kyselyille</i> (kyselyt, jotka kohdistetaan abstraktille ylliluokalle X ja saadaan tuloksina X:n aliluokkia).
Yksilöllisyys	Kumpikaan Javan yksilöllisyyden tunnistusmetodeista, funktio <code>equals()</code> tai operaattori <code>"=="</code> , ei ole ekvivalentti relaatiomallin primääriseen avaimeen perustuvan mallin kanssa.
Assosiaatiot	Oliomallissa assosiaatiot perustuvat olioviitteisiin ja viitejoukkoihin. Relaatiomallissa assosiaatio perustuu sarakkeeseen, joka sisältää vierasavaimia. Relaatiomallin assosiaatiot eivät ole luonteeltaan suunnattuja, kun taas oliomallin ovat. Oliomallissa assosiaatio saattaa näyttää suhteelta n:m, mutta silti ei voi suoraan päätellä, että näin on. Taulujen assosiaatiot ovat taas aina joko 1:1 tai 1:n. Relaatiomallissa n:m-suhteen esittämiseen tarvitaan erillinen linkkitaulu.
Navigointi oliograafissa	Oliomallissa navigoidaan viitteiden avulla oliosta toiseen. Relaatiomallissa on etukäteen tiedettävä, miltä tasolta graafia halutaan tietoa. Taulujen lukumäärä liitoksessa kertoo navigoitavissa olevan graafin syvyyden. Kyselyssä voidaan esimerkiksi haluta aluksi työntekijöiden (n kpl) tiedot, mutta myöhemmin halutaankin tarkat tiedot työntekijöiden osastoista. Tällöin joudutaan suorittamaan oma kysely kullekin työntekijälle, jolloin kyselyitä on pahimmillaan n+1 kappaletta (puhutaan n+1 valinnan ongelmasta).
Päivitykset	Oliomallissa päivitykset koskevat aina vain yhtä oliota kerrallaan kun taas SQL:n <code>update</code> -lauseella voidaan päivittää useita eri olioiden tilaa edustavia rivejä kerrallaan.

### 3.3. Mitä ORM on?

Fussel [Fussell, 2000] näkee ORM:n muunnosprosessina olio- ja relaatiomallien, sekä niitä tukevien järjestelmien välillä. Tämän prosessin tunteminen edellyttää syvällistä ymmärrystä menetelmien yhtäläisyyksistä ja eroista. Optimaalisiin tilanteisiin ei todellisuudessa päästä, mutta olio- ja relaatiomallit voidaan tietyillä menetelmillä saada vastaamaan toisiaan melko hyvin.

Alla olevassa yksinkertaisessa kuvassa havainnollistetaan yleisellä tasolla palvelimella sijaitsevan relaatiotietokannan sisältämien taulujen ja asiakkaan koneella sijaitsevan sovelluksen olioiden suhteita OR-kuvauksessa.



Kuva 5. ORM [Barry, 2004a]

Kuvassa olevat pallot esittävät sovelluksen olioita ja viivat niiden välillä olioiden välisiä suhteita. Kuvassa olevat suorakaiteet esittävät tietokantatauluja. ORM tapahtuu sovelluksen ja palvelimen välissä ja siinä liitetään sovelluksen oliot palvelimen tietokantatauluihin.

Tässä tutkielmassa ORM:n tarkkana määritelmänä käytetään Bauerin ja Kingin [Bauer ja King, 2004] määritelmää, jonka mukaan ORM on käsite, joka kuvaa automaattista ratkaisua olio- ja relaatioparadigman yhteensopimattomuusongelmaan. Manuaaliset ratkaisut eivät siis kuulu kyseisen määritelmän piiriin ja myös tässä tutkielmassa noudatetaan samaa linjaa. Manuaalinen ratkaisu tarkoittaa sellaisia ratkaisuja, joissa oliot ja relaatiot liitetään joka kerta käsin suoraan SQL:n ja JDBC:n avulla. Bauerin ja Kingin pohjalta muotoiltu ja yleistetty määritelmä on esitetty täsmällisemmin seuraavassa:

ORM (object/relational mapping) on oliopohjaisille sovelluksille tarkoitettu, toteutukseltaan sovellukselle näkymätön automaattinen menetelmä, jonka avulla olioiden tila voidaan säilöä relaatiotietokantaan.

Tähän muotoiltu määritelmä jättää tarkoituksella ulkopuolelle sen kohdan Bauerin ja Kingin alkuperäisestä määritelmästä, jossa mainitaan, että mallinnuksen apuna tarvitaan metadataa. Määritelmä on haluttu tässä pitää mahdollisimman yksinkertaisena, eikä siihen ole lisätty vastauksia "miten" kysymyksiin, joihin vastaamiseen edes pintapuolisesti kuluu suuri osa tästäkin tutkielmasta.

Kuten tähän asti esitetystä voidaan päätellä, olioiden ja tietokannan relaatioiden (ts. taulujen) toisiinsa liittämiseen liittyy useita huomioitavia osa-

alueita. Tärkeimpiä ratkaistavia asioita ovat käytännössä ongelmat, jotka liittyvät periytymisen kuvaamiseen, päätöksen tekoon luokkien lukumäärästä taulua kohti, joukkojen kuvaamiseen, olioiden ja tietokannan tietotyyppien liittämiseen sekä olioiden välisten suhteiden kuvaamiseen [Barry, 2004b]. Seuraavissa kohdissa pohditaan ratkaisuja näihin asioihin.

### 3.4. Pysyvyyshä mekanismin tarvitsema informaatio ja metadata

Olioiden kuvaaminen tietokantaan ei onnistu ilman informaatiota, joka kertoo, miten tallentaminen tehdään. Tällaista informaatiota on Amblerin [Ambler, 2003] mukaan muun muassa olioiden avaimet (erityisesti surrogaattiavaimet), samanaikaisuuden hallintaan tarvittavat tiedot (esim. aikaleimat ja versionumerointi) sekä pysyvyyttä ilmaisevat totuusarvot.

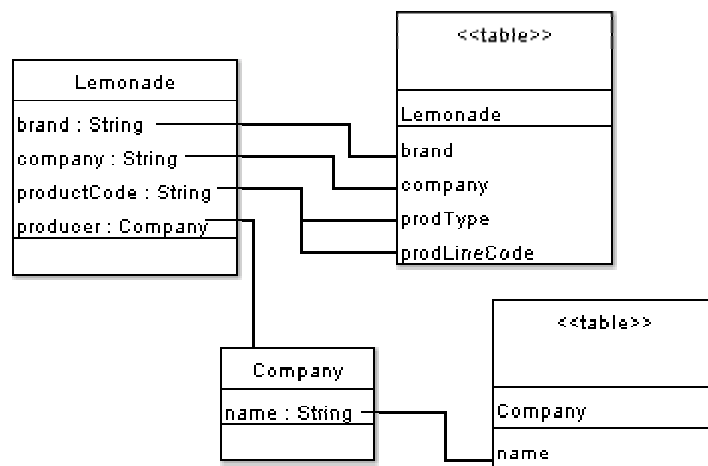
Avaimia tarvitaan erottelemaan oliot toisistaan, versionumeroita ja aikaleimoja voidaan käyttää niin sanotun optimistisen lukituksen (optimistic locking) toteuttamiseen (jos käyttäjä A ehtii päivittää tiedon X ennen käyttäjää B, ilmoitetaan B:lle, että tämän tulee ladata tieto X uudestaan ja aloittaa sovellustransaktio alusta). Pysyvyyttä ilmaisevalla totuusarvolla tarkoitetaan tässä yhteydessä sellaista boolean-tyyppistä muuttujaa (esim. isPersistent), jonka tehtävä on kertoa, onko olio jo tallessa tietokannassa. Pysyvyyshä mekanismin tarvitsema data kuvataan tietokantaan attribuuttien tavoin, ja attribuuttien kuvaamista käsitellään seuraavassa kohdassa.

Metadata on informaatiota datasta. Metadataa tarvitaan, jotta järjestelmälle voidaan kertoa, miten attribuutit, suhteet ynnä muut liittyvät relaatiotietokantaan. Kullakin ORM-kehityksellä on omat menetelmänsä metadatan suhteen, ja tämän tutkielman esimerkkikehitys Hibernate käyttää siinä usein XML-tiedostoja (esimerkki 29, s. 61). Hibernatesta kerrotaan lisää viidennessä luvussa.

### 3.5. Attribuuttien kuvaaminen

Luokan attribuutti kuvataan yhdestä useampaan tietokannan sarakkeeseen. Yksinkertaisinta tämä on silloin, kun attribuutin tietotyyppi vastaa tietokannan tietotyyppiä (esim. String ja VARCHAR). Kaikkia attribuutteja ei aina tarvitse kuvata lainkaan, sillä niistä kaikki eivät välttämättä ole pysyviä. Ambler [Ambler, 2000] mainitsee esimerkkinä tällaisesta luokan "Invoice" attribuutin "grandTotal", jota käytetään erilaisiin laskennallisiin tarkoituksiin, mutta ei tallenneta tietokantaan.

Luokalla voi myös olla attribuuttina jokin toinen luokka, jonka sisältämät attribuutit liitetään tietokannassa nolasta useampaan sarakkeeseen. Kuvassa 6 on esimerkki tällaisesta tilanteesta, jossa Lemonade-luokalla on attribuuttina Company, jonka attribuutit kuvataan edelleen omaan tauluunsa.



Kuva 6. Attribuuttien kuvaaminen

Useat luokan attribuutit voidaan vaihtoehtoisesti liittää myös samaan tietokannan sarakkeeseen. Kuvassa 6 on esimerkki tällaisesta tapauksesta, jossa attribuutti "productCode" esittää tuotekoodia, joka koostuu useista eri tietokentistä. Sillä, mistä osista tuotekoodi koostuu, ei ole luokan Lemonade kannalta merkitystä. Tietokantaan halutaan kuitenkin eritellä tuotekoodin kaksi osaa, jotka ovat tuotteen tyyppi ja tuotteen viivakoodi.

Attribuuttien sijaan voitaisiin periaatteessa puhua myös luokan ominaisuuksista, sillä käytännössä tietokantaan voidaan liittää myös yksittäisten attribuuttien sijasta luokan ominaisuuksia, joita voidaan kuvata yhtenä tai useampana funktiona [Ambler, 2003]. Asian ymmärtämiseksi kuitenkin riittää, että puhutaan attribuuteista.

Toisinaan tarvitaan myös yhteistä, niin sanottua staattista attribuuttia, jota kaikki luokan ilmentymät voivat käyttää. Staattisten attribuuttien kuvaamiseksi tietokantaan on olemassa neljä erilaista mallia [Ambler, 2003]. Ensimmäinen ja yksinkertaisin tapa on käyttää yhtä tietokantataulua yhtä staattista attribuuttia kohti. Taulussa on vain yksi tietokenttä, jossa pidetään kirjaa halutun luokan attribuutin tilasta. Toinen vaihtoehto on käyttää yhtä taulua yhden luokan kaikkia staattisia attribuutteja kohti. Taulussa on tällöin yksi rivi, mutta useampia sarakkeita. Tällöin jokainen tietokenttä huolehtii omasta attribuutistaan. Kolmas vaihtoehto on, että käytetään edellä mainitun kaltaista yhden rivin ja useamman sarakkeen taulua koko oliomallin kaikkien staattisten attribuuttien kuvaamiseen. Viimeinen vaihtoehto on käyttää monirivistä ja monisarakkeista ratkaisua koko oliomallin staattisten attribuuttien kuvaamiseen. Tällaisen taulun yksi rivi sisältää aina yhden staattisen attribuutin kuvauksen, joka koostuu luokan nimestä, attribuutin nimestä sekä tietysti ylläpidettävästä attribuutin arvosta.

### 3.6. Suhteiden kuvaaminen

Olioiden välisiä suhdetyyppejä on olemassa ainoastaan kolme kappaletta: Assosiaatio, aggregaatio ja kompositio. Suhteet voidaan Amblerin [Ambler, 2003] mukaan jaotella kahteen kategoriaan, joista ensimmäinen perustuu suhteiden moninkertaisuuteen ja toinen niiden suuntiin. Ensimmäiseen ryhmään kuuluvat suhteet 1:1, 1:n ja n:m. Sillä, onko kerroin oikeasti 0 tai 1, ei ole merkitystä, sillä tällöin kerroin merkitään aina 1:ksi. Myöskään sillä, onko kerroin todellisuudessa 2, 3 tai n ei ole merkitystä.

Amblerin jaottelun mukaan toisen ryhmän muodostavat yksi- ja kaksisuuntainen suhde. Ryhmä ei oikeastaan ole samalla käsitetasolla ensimmäisen ryhmän kanssa. Yksisuuntaisella suhteella tarkoitetaan oliomallissa näet sitä, että olio A tietää sen kanssa suhteessa olevien oliojoukon B olioista, mutta joukon B oliot eivät tiedä A:n olemassa olosta. Kaksisuuntaisessa suhteessa joukon B oliot taas tietävät A:sta. Esimerkiksi luvun 6 esimerkkisovelluksessa (kuva 21, s. 57) työntekijät tietävät osaston, johon he kuuluvat. Osaston ei kuitenkaan tarvitse välttämättä tietää omista työntekijöistään. Kyseessä on siis aluksi yksisuuntainen suhde. Sovelluksen elinkaaren varrella voi kuitenkin tulla tarvetta sille, että osasto tietää työntekijänsä. Tällöin suhde muuttuu kaksisuuntaiseksi. Suhteiden suuntiin liittyvä ongelma on siis lähinnä toteutustekninen, eikä selkeästi suunnitteluvaiheessa eroteltava suhteen kertoimien ratkaisemisen kaltainen asia. Suunnitteluvaiheessa on näet tärkeää pystyä luomaan mahdollisimman muuttumaton oliomalli kohdeavaruuden olioista, ja tällöin olioiden välisten suhteiden kertoimien ratkaiseminen on suuntaa olennaisempaa, sillä kuten on edellä osoitettu, voi suunta olla oliolta A oliolle B tai oliolta B oliolle A tai molempiin suuntiin. Siinä mielessä suuntakysymys on tärkeitä kuitenkin huomioida, että relaatiotietokannoissa kaikki suhteet ovat aina kaksisuuntaisia toisin kuin oliograafeissa, joissa suunnat on ilmoitettava erikseen.

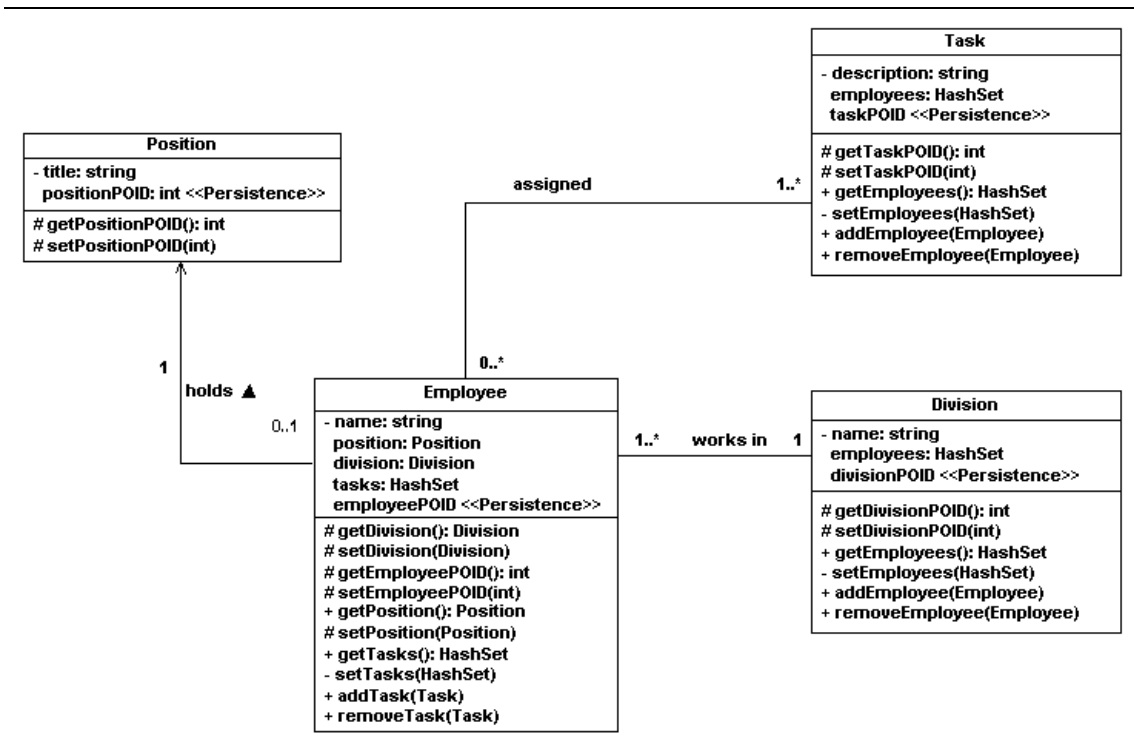
Oliomallin olioiden väliset suhteet toteutetaan Amblerin mukaan viitteinä toisiin olioihin ja operaatioihin. Kun kerroin on 1, toteutetaan suhde olioviitteen sekä get- ja set-operaatioiden avulla. Siinä tapauksessa, että kerroin on n, toteutetaan suhde listan ja sitä käsittelevien funktioiden avulla. Esimerkiksi luvun 6 esimerkkisovelluksessa Department-luokka sisältää listan employees, johon voidaan tallentaa osaston työntekijöitä.

Relaatioiden väliset suhteet toteutetaan puolestaan vierasavainten avulla. 1:1 suhteessa vierasavain täytyy löytyä jommasta kummasta suhteen taulusta. Periaatteessa ei ole väliä, kumpaan tauluun se sijoitetaan, kunhan tarvittava liitos voidaan vain tehdä. Suhde 1:n toteutetaan taas siten, että suhteen n-puolen tauluun lisätään vierasavain suhteen 1-puolen tauluun. Suhdetta n:m varten taas tarvitaan ylimääräinen taulu.

Amblerin mukaan hyvä perussääntö suhteiden kuvaamisessa on se, että kertoimet tulisi pitää muunnosprosessissa samoina. Siten esimerkiksi 1:n-oliosuhde kuvautuu 1:n-datasuhteeksi. Säännössä pitäytyminen ei kuitenkaan ole välttämätöntä, sillä esimerkiksi 1:1-suhteen voi halutessaan kuvata 1:n-datasuhteeksi, sillä 1:1-datasuhde on 1:n-datasuhteen osajoukko. Samoin datasuhde 1:n on n:m-suhteen osajoukko.

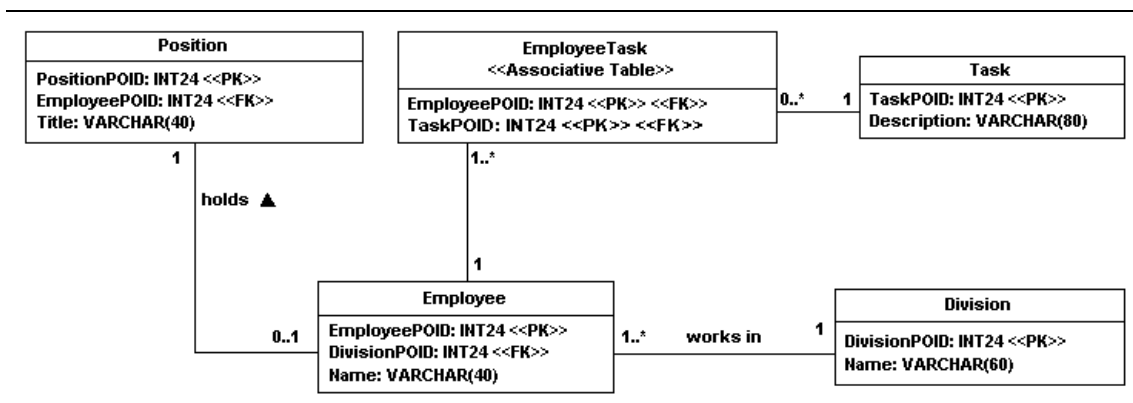
### 3.6.1. 1:1

Yksinkertainen suhde, ja siten myös suoraviivaisin mallintaa, on suhde 1:1. Suhteen molemmissa päissä on siis maksimikertoimena 1. Amblerin [Ambler, 2003] esimerkissä (kuva 7) löytyy tällainen suhde luokkien Employee ja Position välillä.



Kuva 7. Kohdealueen oliomalli [Ambler, 2003]

Oliosuhteen 1:1 kuvaaminen on hyvin suoraviivaista, sillä kuten kuvasta 8 nähdään, tehdään relaatiotietokantaan vain kumpaakin luokkaa vastaavat taulut, joista jompaan kumpaan tauluun lisätään vierasavain toiseen tauluun. Alla olevassa kaaviossa vierasavain (EmployeePOID) on lisätty tauluun Position.



Kuva 8. Vastaava relaatiomalli [Ambler, 2003]

Tavalliset attribuutit kuvataan myös yksikäsitteisesti, kuten kuvista huomataan. Esimerkiksi luokan Position attribuutti "title" kuvataan relaation Position sarakkeeksi "title". Esimerkkisovelluksesta (luku 6) löytyy lisäksi poikkeuksellinen esimerkki 1:1-suhteesta, jossa kummallekin oliosuhteen osapuolelle ei ole omaa taulua tietokannassa.

### 3.6.2. 1:n

Toinen mallinnettava suhde on suhde 1:n, jossa suhteen toisen puolen kerroin on maksimissaan 1 ja toisen puolen kerroin on vähintään 1. Kuvassa 7 on tällainen suhde luokkien Employee ja Division välillä. Suhde toteutetaan siten, että suhteen n-puolelle laitetaan vierasavain osoittamaan suhteen 1-puoleen. Siitä syystä tauluun Employee lisätty vierasavain (DivisionPOID) tauluun Division, kuten kuvasta 8 havaitaan.

Jos relaatiotietokannan kaavio olisi jo olemassa, voitaisiin sen sarakkeiden määrittelyistä heti päätellä suhteiden tyyppisiä. Seuraavanlainen SQL-määrittely tarkoittaa esimerkiksi aina suhdetta 1:n:

---

```
DIVISION_POID BIGINT FOREIGN KEY REFERENCES DIVISION
```

---

### Esimerkki 12. Sarakkeen määrittely 1

Kerroin voidaan siis päätellä vierasavaimen määrittelystä. Koska määreessä ei ole UNIQUE avainsanaa, voi samalla id:llä olevia rivejä olla taulussa useita. Seuraavasta suhdemäärittelystä voidaan siis päätellä, että siinä tarkoitetaan suhdetta 1:1:

---

```
EMPLOYEE_POID BIGINT UNIQUE FOREIGN KEY REFERENCES EMPLOYEE
```

---

### Esimerkki 13. Sarakkeen määrittely 2

### 3.6.3. n:m

Kolmas suhde on suhde n:m, jossa suhteen kummankin puolen kerroin on vähintään 1. Kohdealueen oliomallia havainnollistavassa kuvassa 7 on tällainen suhde luokkien Employee ja Task välillä. Esimerkkioliomallin mukaan yhdellä tehtävällä voi siis olla useita tekijöitä ja yhdellä työntekijällä voi olla useita tehtäviä.

Suhteen n:m kuvaamiseksi on olemassa kaksi mahdollisuutta. Ensimmäinen vaihtoehto on lisätä vierasavaimen sarake toiseen tauluun useita kertoja. Tämä ei kuitenkaan ole järkevä vaihtoehto, sillä jos esimerkiksi työntekijälle haluttaisiin lisätä 7 tehtävää, vaikka tehtävän TaskPOID sarakkeita olisi Employee taulussa vain 5, ei lisääminen tietenkään olisi mahdollista [Ambler, 2003].

Toinen ja oikeastaan ainoa järkevä tapa on ottaa mukaan erillinen suhdetaulu, jossa rivejä muodostavat parit "primäärinen avain taulusta A-primäärinen avain taulusta B" [Ambler, 2003]. Esimerkissä tällainen suhdetaulu on EmployeeTask, joka sisältää pääavaimet kummastakin taulusta. Nyt työntekijälle voi ajonaikana lisätä tehtäviä mielivaltaisen määrän. Perusidea on siis muuntaa n:m-suhde kahdeksi 1:n-suhteeksi, missä siis alkuperäinen taulu on aina suhteen 1-puoli ja uusi suhdetaulu on suhteen n-puoli aivan kuten kuvassa 8. Kaksi luokkaa kuvataan siis kolmeksi tietokantatauluksi.

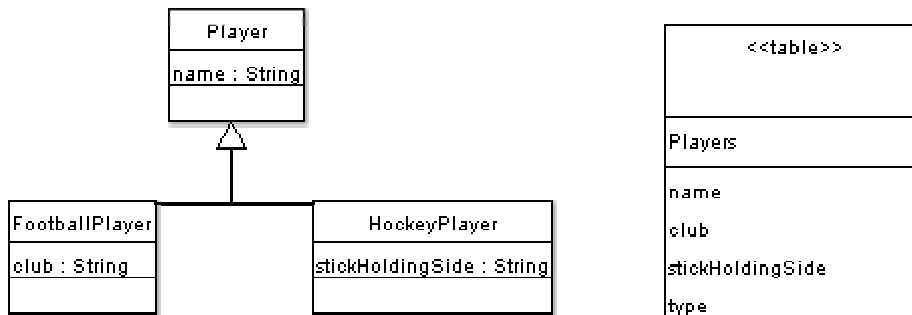
Rekursiivisten suhteiden kuvaamisessa tietokantaan ei ole teoriassa mitään eroa yllä kuvattujen suhteiden kuvaamisen kanssa. Suhteessa n:m molemmat suhdetaulun sarakkeet ovat vain vierasavaimia samaan tauluun ja suhteessa 1:n on taulussa A vierasavain B, joka viittaa taulun A sarakkeeseen C.

## 3.7. Periytyksen kuvaaminen

Relaatiotietokannoissa ei perinteisesti ole suoraa tukea periytykselle, vaikka SQL-standardia laajennettiin vuonna 1999 mahdollistamaan myös tietokannan taulujen periytyksen toisista tauluista. Siitä syystä oliomallin periytyksirakenteiden tietokantaan kuvaamiseen tutustuminen on perusteltua. Liiallista luokkien konkreettisen periytyksen käyttämistä tulisi tosin välttää [Johnson, 2003] muun muassa särkyvän yliluokan ongelman [Koskimies, 2003] vuoksi. Rajapintojen käyttäminen ainakin Javalla tehtäessä on Johnsonin mukaan lähes aina parempi tapa myös silloin, vaikka aluksi vaikuttaisi paremmalta periyttää uusi luokka konkreettisesti toisesta luokasta. Kaikesta huolimatta, periytyksen kuvaamiseksi on olemassa kolme yleisintä vaihtoehtoa, jotka esitellään seuraavissa alikohdissa. Eri kuvaamistapoja koskevat tiedot perustuvat pääsääntöisesti Amblerin [Ambler, 2003], Bauerin ja Kingin [Bauer ja King, 2003] ja Fowlerin [Fowler, 2003] teoksiin. Alakohtien kaaviot on muokattu Fowlerin esimerkkien pohjalta.

### 3.7.1. Yksi taulu koko luokkahierarkialle

Ensimmäinen vaihtoehto on käyttää ainoastaan yhtä tietokannan taulua kuvaamaan kokonaista luokkahierarkiaa. Fowler on pukenut menetelmän malliksi, jota hän kutsuu nimellä "Single Table Inheritance". Parhaiten asia ymmärretään seuraavaan kaavioon avulla:



Kuva 9. Single Table Inheritance

Kuten kuvasta nähdään, merkitään luokkahierarkiaa vastaavan taulun nimeksi "Player", jonka sarakkeiksi laitetaan kaikkien luokkien kaikki attribuutit. Hierarkiataso merkitään "type"-sarakkeeseen, jota Bauer ja King [Bauer ja King, 2004] kutsuvat tyyppi diskriminaattoriksi (type discriminator). Se saisi esimerkkitapauksessa arvot "FootballPlayer" tai "HockeyPlayer" tai niitä vastaavat ennalta sovitut numeeriset arvot.

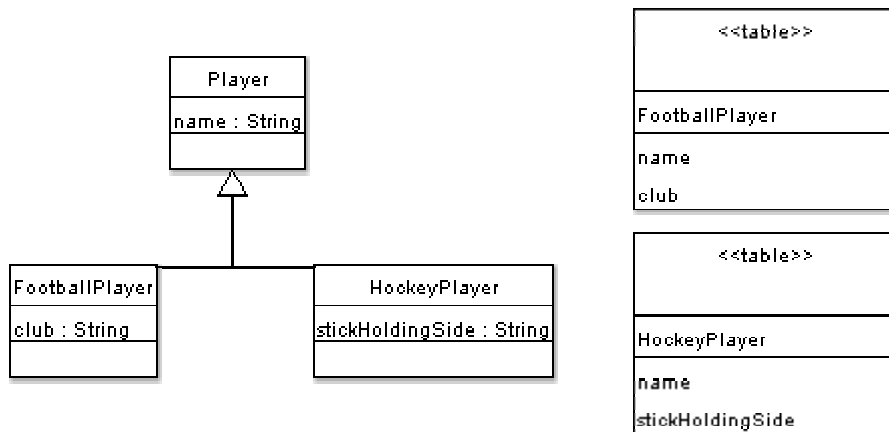
Liitosten määrä kyselyissä on eräs tekijä, joka vaikuttaa usein kyselyiden tehokkuuteen. Tämän vaihtoehdon yksi etu onkin se, että liitoksia ei tarvitse tehdä lainkaan. Menetelmä on myös yksinkertainen, sillä tietokannassa on vain yksi taulu, josta pitää huolehtia. Muutosten tekeminen hierarkiatasoissa on lisäksi yksinkertaista, sillä tietokantataulun rakenteeseen ei tällöin tarvitse tehdä muutoksia.

Toisaalta huonoa tässä mallissa on se, että taulu paisuu monesti todella suureksi, eikä siitä ole helppoa erotella, mitkä kentät kuuluvat millekin oliomallin luokalle. Hukkatilaa tulee usein myös paljon, sillä osaa sarakkeista käyttävät vain jotkut aliluokat. Tällöin monissa kentissä on null- tai tyhjiä arvoja. Tosin eri tietokantojen optimointimenetelmät osaavat käsitellä tätä ongelmaa eri tavoin. Tästä kuitenkin seuraa se, että aliluokkien attribuutteja vastaavat sarakkeet on merkittävä null-arvot salliviksi, mikä saattaa aiheuttaa ongelmia tietokannan eheydessä (integrity). Nimiavaruus on myös hyvin rajallinen, joskin tämän ongelman voi ratkaista erilaisten nimeämiskäytäntöjen avulla.

### 3.7.2. Yksi taulu konkreettista luokkaa kohti

Toinen vaihtoehto on käyttää yhtä taulua jokaista konkreettista luokkaa kohti. Toisin sanoen abstrakteille luokkahierarkian luokille ei luoda omia tauluja.

Tätä menetelmää Fowler kutsuu nimellä "Concrete Table Inheritance". Seuraava kuva havainnollistaa menetelmää:



Kuva 10. Concrete Table Inheritance

Kaaviosta nähdään, että tässä tapauksessa luokkahierarkian ylimmällä luokalla ei ole omaa taulua, koska se ei ole sellainen luokka, josta luodaan olioita. Sitä vastoin kummallakin juuren perivällä luokalla on tietokannassa oma vastaava taulunsa, jonka nimi on vastaavan luokan nimi ja jonka sarakkeet ovat vastaavan luokan attribuutteja. Lisäksi kussakin taulussa on mukana yliluokan attribuutit, eli tässä tapauksessa attribuutti "name".

Tämän mallin suurin ongelma on, että se ei tue polymorfisia assosiaatioita kovinkaan hyvin. Polymorfinen assosiaatio tarkoittaa konkreettisen luokan assosiaatiota abstraktin luokan kanssa. Tällöin konkreettinen luokka voi ajon aikana olla assosiaatiossa minkä tahansa abstraktin yliluokan aliluokan kanssa. Kuten edellä jo puhuttiin, esitetään assosiaatiot tietokannoissa vierasavainten avulla. Jos sitten kaikki aliluokat kuvataan eri tauluihin, ei polymorfista assosiaatiota niiden yliluokkaan voida esittää yhtenä vierasavainsuhteena. Myös polymorfiset kyselyt aiheuttavat ongelmia. Polymorfinen kysely on sellainen kysely, jossa kysely kohdistetaan abstraktiin yliluokkaan ja tulosjoukkoon tulee aliluokkien ilmentymiä.

Menetelmään liittyy myös merkittävä käsitteellisen tason ongelma, nimittäin mikäli yliluokkaan lisätään, tai siitä poistetaan attribuutteja, pitää muutokset heijastaa myös kaikkiin muihin tauluihin. Tämä aiheuttaa ylimääräistä ylläpitotyötä, sekä vaikeuttaa tietokannan eheysrajoitteiden (integrity constraints) tekemistä.

### 3.7.3. Yksi taulu luokkaa kohti

Viimeinen vaihtoehto periytyamisen kuvaamiseksi on käyttää yhtä taulua kaikkia oliomallin luokkia kohti olivat ne sitten abstrakteja tai konkreettisia. Fowlerin nimeämä malli tälle strategialle on nimeltään "Class Table Inheritance". Mallin nimen ja edellisten kohtien perusteella ei ole vaikeaa

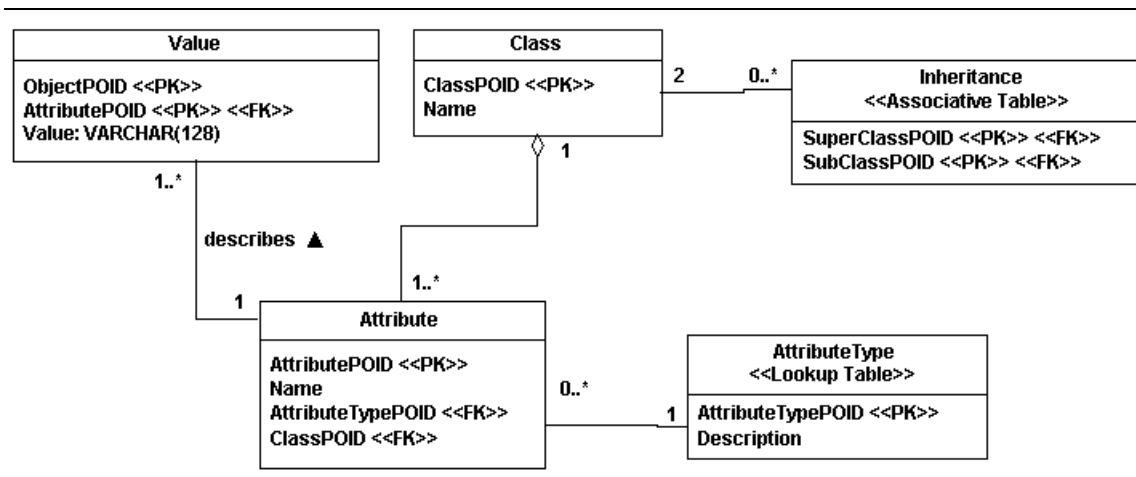
päätellä, mistä tässä vaihtoehdossa on kyse. Kaavion esittäminenkään ei ole tarpeellista, sillä lisäämällä kaavioon 10 taulun "Player" ja poistamalla "name" sarakkeet olemassa olevista tauluista, sekä lisäämällä "name" sarakkeen uuteen tauluun, päästään tässä mallissa esitettyyn tilanteeseen esimerkin kohdalla. Jokaiselle luokkahierarkian luokalle luodaan siis täsmälleen vastaavat taulut tietokantaan. Tauluissa on toisinsanoin vain sellaisia sarakkeita, joita ei ole peritty.

Tässä mallissa relaatiomalli on normalisoitu. Skeeman kehittäminen ja eheysrajoitteiden määrittäminen on näin ollen suoraviivaista. Polymorfinen assosiaatio voidaan helposti esittää vierasavaimen avulla.

Mallin ilmeisin ongelma on tehokkuus, mikäli hierarkiassa on paljon luokkia ja vastaavat tietokantataulut paisuvat suuriksi. Tällöin on ilmeistä, että joudutaan joko tekemään useita erillisiä kyselyitä useisiin tauluihin tai käyttämään useita liitoksia. Edellinen vaihtoehto ei välttämättä ole paras mahdollinen, sillä se aiheuttaa runsaasti liikennettä sovelluksen ja tietokannan välillä. Jälkimmäinen vaihtoehto on taas siinä mielessä huono, että liitoksen tekeminen on tietokannasta riippumatta lähes poikkeuksetta raskas operaatio ja mitä enemmän liitoksia joudutaan tekemään, sitä raskaampaa ja hitaampaa kyselyiden suorittaminen on. Useiden liitosten tekeminen voi myös sen vuoksi olla hidasta, että tietokanta joutuu etsimään optimaalisen liitosjärjestyksen. Sopivien indeksien avulla liitosten tekemiseen kuluva aika voidaan kuitenkin vähentää.

#### **3.7.4. Luokkien kuvaaminen geneeriseen taulurakenteeseen**

Ambler [Ambler, 2003] esittelee edellä mainittujen yleisimpien, useissa lähteissä mainittujen menetelmien lisäksi vielä yhden menetelmän, jota voidaan soveltaa sekä periytymisen kuvaamiseen, mutta myös muihinkin tarkoituksiin. Tässä esiteltävää geneeristä menetelmää kutsutaan toisinaan myös metadata-pohjaiseksi malliksi. Alla olevassa kuvassa nähdään geneerinen tietokantaskeema, johon voidaan tallentaa attribuuttien arvoja ja joka mahdollistaa periytymisrakenteiden kuvaamisen. Mallia voitaisiin myös tarvittaessa laajentaa tukemaan suhteiden kuvaamista.



Kuva 11. Geneerinen rakenne [Ambler, 2003]

Oliomallissa olevat luokat tallennetaan tauluun Class. Luokkien attribuutit tallennetaan tauluun Attribute. Attribuuttien arvot laitetaan tauluun Value ja attribuuttien tyypit tauluun AttributeType. Viimeksi mainittua taulua tarvitaan, jotta attribuutti osataan konvertoida Value-kentän varchar-tyypiseksi. Luokkien periytymishierarkia taas tallennetaan tauluun Inheritance siten, että SubclassPOID viittaa sen luokan aliluokkaan, johon SuperClassPOID viittaa. Inheritance-tiluun tulee siis lisätä yksi rivi aina yhtä periytymissuhdetta kohti.

Mallin kiistattomia etuja on erityisesti sen lähes rajaton dynaamisuus. Huonoa mallissa on sen monimutkaisuus; yhden olion rakentamiseksi joudutaan tietoa hakemaan useista tauluista ja useilta riveiltä. Myös raportoinnin järjestäminen tällaista tietokantaa vasten voi muodostua liian suureksi haasteeksi. Mallia käytettäessä menetetään oikeastaan kaikki relaatiotietokannan tarjoamat edut. Jos haitoistakin huolimatta mallia halutaan käyttää, sopii se mutkikkaille sovelluksille, jotka käsittelevät pientä tietomäärää, tai sellaisille sovelluksille, missä tietokannassa ei tarvitse käydä usein, jolloin välimuistin (cache) käyttäminen on mahdollista toiminnan nopeuttamiseksi.

### 3.8. Tietokannan suunnittelusta

Käytännössä on olemassa 3 eri tapausta, joissa ORM-menetelmää sovelletaan [Fowler, 2003]:

1. Tietokantaskeema luodaan oliomallin luomisen yhteydessä tyhjästä
2. Tietokantaskeema on jo olemassa, mutta sitä ei voi muuttaa.
3. Tietokantaskeema on jo olemassa ja siihen voidaan tarvittaessa tehdä muutoksia.

Toinen tapaus on tyypillinen esimerkiksi silloin, kun ohjelmistoyrityksen pitää toteuttaa sovellus asiakkaan olemassa olevan tietojärjestelmän päälle. Tällaisissa tapauksissa minkäänlaisten muutosten tekeminen varsinaiseen kantaan saattaa olla kiellettyä. Kolmas tapaus tarkoittaa sitä, että asiakasyritys saattaa suostua muutokseen, mikäli niiden tekemiseen on riittävän hyvät perusteet. Muutosten tekeminen on siis neuvottelukysymys. Ensimmäinen tapaus on kehittäjien kannalta optimaalinen, sillä siinä sekä tietokannan että oliomallin voi suunnitella alusta asti ORM-menetelmää silmällä pitäen.

Käytännössä tilanne voi myös sijoittua tapauksen välille. Uusi tietojärjestelmätoimittaja voi näet suunnitella kokonaan oman tietokanta-kaavionsa omalle sovellukselleen ja liittää sen näkymien ja vierasavainten avulla varsinaiseen alkuperäiseen tietokantaan.

Fowlerin mukaan tietokantaskeeman suunnittelua tulee käsitellä nimenomaan olioiden pysyvyyden toteuttamisen näkökulmasta. Tietokantaskeemaa kannattaa rakentaa lyhyissä sykleissä yhdessä oliomallin kanssa. Parhaaseen tulokseen päästään, kun aina ensin ajatellaan oliomallia ja vasta sitten fyysistä tietomallia.

## 4. ORM-kehukset

Kolmannen luvun määritelmän automatiikkaa koskevan vaatimuksen täyttämiseksi tarvitaan sovelluskehys, jonka vastuulle voidaan siirtää suuri osa menetelmän toteuttamiseen liittyvistä tehtävistä. Tässä kohdassa esitellään yleisellä tasolla, mitä kaikkea ORM-kehysten vastuulla on ja miten hyvä kehys rakentuu. Rakenteen kuvaamisessa ei pyritä täydellisyyteen, vaan esitellään vain joitakin perusajatuksia. Motivaationa luvulle on se, että ymmärtämällä ORM-kehysten rakenteen, ymmärretään, miten olioiden ja relaatioiden yhteistoiminta automatisoidaan, ja edelleen ORM:n käytännön toteuttaminen valmiiden kehysten avulla on sen seurauksena helpompaa. Viidennessä luvussa tutustutaan tarkemmin tässä luvussa esiteltyjä malleja soveltavaan Hibernate-kehykseen, joka on tällä hetkellä ORM-kehysten parhaimmista.

### 4.1. Ohjelmistokehysistä

Ohjelmistokehys on kuin auton runko, jonka ympärille voidaan asentaa erilaisia koreja ja moottoreita. Kehys ei yleensä toimi itsenäisesti (kuten ei auton runkokaan), vaan vaatii ohjelmakoodia, jolla siinä olevat aukot täytetään. Erään kehysten määritelmän mukaan kehys on uudelleenkäytettävä, puoliksi valmis sovellus, josta voidaan erikoistaa toisia sovelluksia eri tarkoituksiin [Fayad *et al.* 1999].

Fayad *et al.* luettelevat tiettyjä etuja, joita kehukset parhaimmillaan tarjoavat. Selitysten suomentamisessa on käytetty apuna Seppäsen [Seppänen, 2001] luentomateriaalia.

1. Modulaarisuus: Sovelluskohtainen toiminnallisuus on kapseloitu ja kätketty vakiomuotoisten rajapintojen taakse.
2. Uudelleenkäytettävyys: Rajapinnat mahdollistavat geneeristen komponenttien valmistamisen.
3. Laajennettavuus: "Koukku"-menetelmän (hook) soveltaminen mahdollistaa sovelluskohtaisen toiminnallisuuden toteuttamisen.
4. Suorituksen hallinnan käänteisyys: Tapahtumienkäsittelijät mahdollistavat sovelluskohtaisen tapahtumien käsittelyn siten, että kehys osaa päättää, mitä sovelluskohtaisia menetelmiä pitää kutsua. Näin ollen suorituksen hallinta on kehysten vastuulla kehittäjän sijaan.

Kehysten tulisi näin ollen nostaa tuottavuutta vähentämällä kirjoitettavan ohjelmakoodin määrää ja parantamalla sovellusten laatua. Tulevissa kohdissa esitelläänkin lähinnä Fowlerin [Fowler, 2003] pohjalta joitakin ORM-kehysten

rakentamisessa käytettäviä suunnittelumalleja, joita muun muassa Hibernate soveltaa.

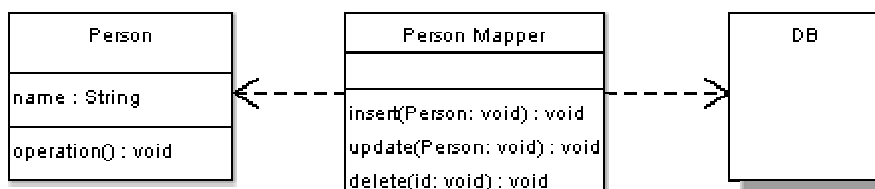
Kerrosarkkitehtuurin hyötyjä on muun muassa se, että kullakin kerroksella on omat selkeästi rajatut vastuunsa. Nämä hyödyt tulevat ilmi sitä paremmin, mitä suuremmista sovelluksista on kyse. Yksi näistä kerrosarkkitehtuurin kerroksista on lähes aina pysyvyyden toteuttava kerros, jonka taustalla on usein relaatiotietokanta. Jotta kohdealueen malli (Domain Model) ja tämä pysyvyyden toteutustapa voidaan pitää toisistaan erillään, voidaan käyttää erityistä suunnittelumallia, josta Fowler [Fowler, 2003] käyttää nimeä "Data Mapper". Data Mapper käsittelee kaikkea lataamista ja tallentamista tietokannan ja mallin välillä ja sallii muutosten tekemisen kumpaankin ilman, että muutokset vaikuttavat toiseen osapuoleen. Data Mapper onkin yksi niistä malleista, joita hyvä ORM-kehys hyödyntää. Seuraavassa kohdassa esitelläänkin mallia "Mapper" ja sen erikoistapausta Data Mapper.

## 4.2. Mapper ja Data Mapper

Mapper on malli, jonka perusteella tehdään olioita, jotka huolehtivat kahden itsenäisen komponentin välisestä kommunikaatiosta kätkemällä kommunikaation yksityiskohdat. Mapperin tarkoituksena on pitää erilliset komponentit mahdollisimman irrallaan toisistaan. Tämä poistaa komponenttien väliset riippuvuudet, mikä on ylläpidon kannalta tärkeää.

Ongelmana on tietysti se, kumpi olioista herättää tarvittaessa Mapperin. Ongelma ratkeaa käyttämällä lisäkomponenttia, joka käyttää Mapperia. Toinen vaihtoehto on käyttää Observer-mallia, jossa Mapperista tehdään jomman kumman kommunikoivan osapuolen tapahtumien tarkkailija (event listener). Näin Mapper alkaa toimia, kun kuunneltavalle osapuolelle tehdään jokin toimenpide, joka aiheuttaa tapahtuman.

Data Mapper (DM) on puolestaan Mapperin erikoistapaus, joka muodostaa tyypillisen ORM-kehysten ytimen. Seuraavassa kaaviossa on esimerkki DM-mallista.



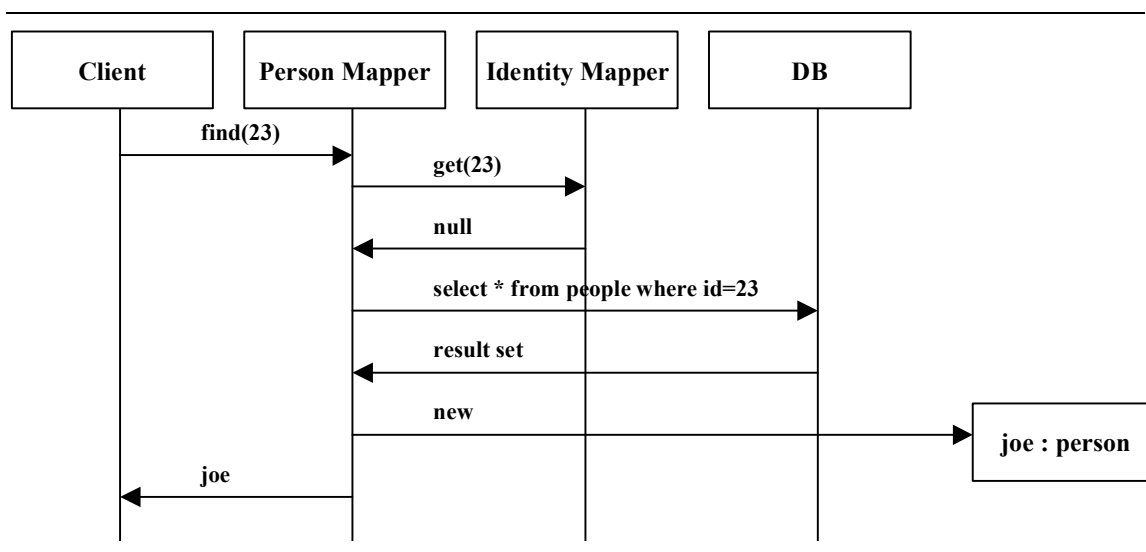
Kuva 12. Person Mapper

DM muodostuu oikeasti useammista Mappereista, jotka siirtävät dataa olioiden ja tietokannan välillä pitäen ne erillään toisistaan ja DM:stä itsestään. Peruste DM:n sovittamiseksi tietokannan ja sovelluksen väliin on se, että sen avulla voidaan suurelta osin ratkaista kohdassa 3.2 kuvattu olioiden ja relaatioiden

väläinen yhteensopivuusongelma. DM:n avulla saavutetaan lisäksi tietokannan ja sovelluksen välinen riippumattomuus, sillä DM:n olennaiset tehtävät ovat tiedon välittäminen komponenttien välillä ja tietokannan erottaminen sovelluksesta.

### 4.3. Identity Mapper

DM:n yhteydessä käytetään myös Identity Mapper (IM)-mallia. IM varmistaa, että kukin olio ladataan vain kerran pitämällä jo ladattuja olioita tallessa. IM on ensimmäinen paikka, mistä olioita etsitään, kun niihin viitataan. Jos olio löytyy IM:sta, ei sitä tarvitse ladata tietokannasta. Kuvassa 13 näkyy, miten Person-luokan oliota haetaan tunnuksella 23. Vasta kun todetaan, että IM palauttaa null, suoritetaan kysely tietokantaan.



Kuva 13. Person-olion ("Joe") hakeminen tietokannasta [Fowler, 2003]

Yllä olevan kaavion esimerkki on hyvin yksinkertainen, sillä kuvaus on suoritettu siten, että olion luokkaa vastaa tietokannassa samanniminen taulu ja olion attribuutteja vastaavat taulussa samannimiset sarakkeet. Todellisuudessa tilanne on usein monimutkaisempi, sillä DM:n tarvitsee esimerkiksi toisinaan kuvata yksi luokka useampiin tauluihin ja osata toimia oikein perinnän kuvaamisessa.

### 4.4. Unit of Work

Kuvauksista huolehtivan komponentin on päivitysten yhteydessä oltava jatkuvasti tietoinen, mitkä oliot ovat muuttuneet, mitä uusia olioita on tehty ja mitä olioita on poistettu järjestelmästä. Kaiken lisäksi kaikki tämä on sovitettava transaktionaaliseen kehykseen, sillä jokaisen pienenkin muutoksen kirjoittaminen jatkuvasti tietokantaan on tehotonta ja turhaa. Ratkaisuna tähän ongelmaan on malli, jonka nimi on Unit of Work (UoW).

---

Unit of Work
registerNew(Object: void) : void
registerDirty(object: void) : void
registerClean(Object: void) : void
registerDeleted(Object: void) : void
commit() : void

---

Kuva 14. Unit of Work [Fowler, 2003]

UoW:n tehtävänä on ylläpitää listaa sellaisista olioista, joita käsitellään liiketoimintaloogisten-transaktioiden (business transaction) kautta. Huomautettakoon, että tässä ei tarkoiteta tietokantatransaktioita, joskin usein tietokanta- ja liiketoimintalooginen transaktio tarkoittavat käytännössä samaa asiaa. UoW huolehtii myös muutosten kirjoittamisesta tietokantaan, sekä ratkaisee samanaikaisuuteen (concurrency) liittyvät ongelmat. UoW pitää siis kirjaa kaikesta mahdollisesti tietokantaan vaikuttavista toimenpiteistä, mitä käyttäjä tekee transaktion aikana. Kun käyttäjä on valmis, selvittää UoW, mitä kaikkia tietokantamuutoksia käyttäjän toimien ansiosta tulee tehdä.

UoW luodaan aina siinä vaiheessa, kun aletaan tehdä jotain sellaista, joka voi vaikuttaa tietokantaan. Joka kerta luotaessa, muutettaessa tai poistettaessa olioita, tehdään ilmoitus UoW:lle. UoW:lle voidaan myös tarvittaessa ilmoittaa muistiin luetut oliot, jotta se voi suorittaa yhdenmukaisuustestausta transaktioiden aikana.

UoW:n perusajatus on se, että vasta tietokantaan suoritettavan commit-komennon yhteydessä UoW päättää, mitä tehdään. Yleensä se avaa transaktion, tekee yhteiskäyttöön liittyviä tarkistuksia ja kirjoittaa muutokset tietokantaan. Sovelluskehittäjän ei UoW:n ansiosta tarvitse koskaan kutsua eksplisiittisesti tietokantaa päivittäviä funktioita. Ainoa asia, mitä sovelluskehittäjän tulee muistaa, on rekisteröidä tarvittavat oliot UoW:lle, jotta se tietää, minkä olioiden muutoksia pitää valvoa.

#### 4.5. Lazy Load

Tietokantakyselyn myötä ladataan usein haettavaan olioon liittyviä muitakin olioita. Jotta koko oliograafia ei turhaan ladattaisi muistiin, on raja vedettävä jonnekin. Tuon rajan vetäminen on DM:n vastuulla. DM siis päättää, kuinka paljon olioita ladataan kerralla muistiin. Apuna tehtävän suorittamisessa käytetään mallia nimeltä "Lazy Load".

Lazy Load on olio, joka ei sisällä kaikkea tarvittavaa dataa, mutta tietää, miten siihen päästään käsiksi. Se keskeyttää latausprosessin halutusta kohdasta jättäen merkin oliograafiin siten, että dataa voidaan ladata aina vain käytön

yhteydessä. Haku on siinä mielessä nimensä mukaisesti laiska, että on mahdollista, että osaa datasta ei tarvitse koskaan hakea, jolloin latausprosessi voi "laiskotella" mahdollisimman kauan. Mallin voi toteuttaa muun muassa tunnettua Proxy-mallia [Gamma *et.al*, 1995] hyväksikäyttäen. Hibernate käyttää tätä Proxy-malliin perustuvaa toteutusta, ja siitä on kerrottu hieman tarkemmin kohdassa 5.6.

Edellä esiteltyjen mallien lisäksi ORM-kehys sisältää toki runsaasti muitakin malleja sekä niiden variaatioita. ORM-kehysten rakentaminen on vaativa tehtävä ja tässä esiteltyt mallit kuvaavat vain joitakin ongelmia ja niiden ratkaisuja, joita kehysten rakentamisen yhteydessä kohdataan. Seuraavassa luvussa esitellään erästä olemassa olevaa ORM-kehystä, Hibernatea.

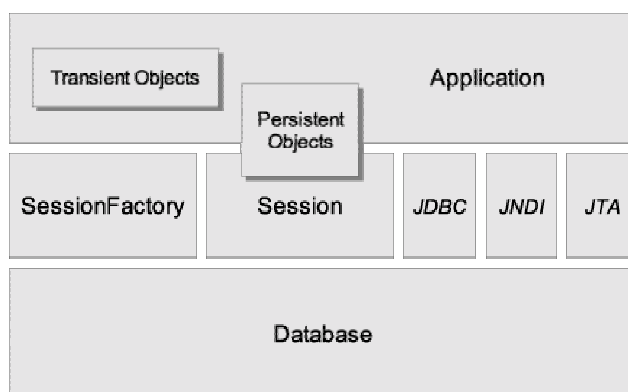
## 5. Hibernate

Kuten edellisen luvun sisällöstä voidaan päätellä, ORM-kehysten tarkoituksena on vähentää ja yksinkertaistaa sovellusten kirjoittamista sekä helpottaa niiden ylläpitoa. Sovelluksista tulee usein myös tehokkaampia kehysten edesauttaman tietokannan käsittelyn optimoinnin ansiosta. ORM-kehyksistä ainakin tässä esiteltävä Hibernate yhdessä JDBC-rajapinnan kanssa mahdollistavat lisäksi tietokantariippumattomuuden, mikä taas johtaa siihen, että tietokanta-toimittajan vaihtaminen ei ole enää aikaa vievä ja kallis operaatio. Kehykset eivät kuitenkaan nopeuta tai paranna sovelluksia automaattisesti; yleiset ORM-menetelmän periaatteet, sekä kunkin kehiksen erityiset piirteet tulee hallita hyvin. Tässä luvussa käydään läpi Hibernaten tärkeimpiä käsitteitä, yleisimpiä ajonaikaisia arkkitehtuureja, sekä tärkeimpiä ominaisuuksia pääasiassa Hibernaten tekijöiden Bauerin ja Kingin [Bauer ja King, 2004] kirjoittaman kirjan, sekä Hibernaten dokumentaation [Hibernate, 2004b] pohjalta.

### 5.1. Arkkitehtuuri ja tärkeimmät rajapinnat

Hibernaten ajonaikainen arkkitehtuuri vaihtelee käyttötarkoituksen mukaan. Joskus Hibernatesta halutaan hyödyntää vain pientä osaa (kuva 15), kun taas toisinaan sitä halutaan käyttää kokonaisvaltaisemmin (kuva 16).

Ensimmäisessä kuvassa esitellään esimerkki kevyestä arkkitehtuurista, jossa Hibernate ei ole vastuussa JDBC-yhteyksistä eikä transaktioiden hallinnasta, vaan vastuu niistä on sovelluksella. Sen sijaan Hibernate on vastuussa sessioista, joista puhutaan lisää alempana.



Kuva 15. Kevyt arkkitehtuuri [Hibernate, 2004b]

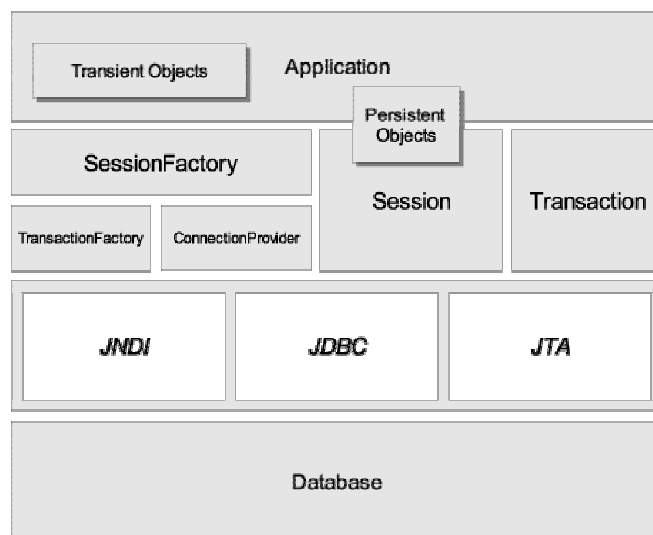
SessionFactory on rajapinta, jonka toteutuksen avulla luodaan Hibernate-sessioita (ilmentymiä Session-rajapinnan toteutuksesta). SessionFactory-rajapintaa voidaan tarvittaessa käyttää useissa säikeissä (thread safe). Rajapinnan toteutuksesta luodaan tyypillisesti ainoastaan yksi ilmentymä

sovellusta kohti sovelluksen lataamisen yhteydessä, sillä sen luominen on raskas operaatio. Mikäli sovellus kuitenkin käyttää useita tietokantoja, tarvitaan yksi SessionFactory jokaista käytettävää tietokantaa kohti. SessionFactory pitää myös yllä välimuistia, johon tallennetaan generoituja SQL-lauseita, sekä Hibernaten ajon aikana tarvitsemaa metadataa. Sen lisäksi SessionFactory pitää tarvittaessa yllä niin sanottua toisen tason välimuistia (second level cache).

Session on puolestaan rajapinta, jonka ilmentymien luominen ja tuhoaminen on halpaa. Session-ilmentymiä tarvitaan jatkuvasti. Niitä ei ole suunniteltu toimimaan monisäikeisesti (not threadsafe), joten tähän asiaan on kiinnitettävä huomiota jo suunnitteluvaiheessa. Hibernaten sessio-käsite sijoittuu merkitykseltään yhteyden (connection) ja transaktion välille. Sessio on sellaisten muistissa olevien olioiden säiliö, mitkä ovat tekemisissä yksittäisen työn yksikön (single unit of work) kanssa. Hibernate reagoi olion pysyvyydessä tapahtuviin muutoksiin vain tällaisen työn yksikön sisällä. Sessiota kutsutaan myös pysyvyyden hallitsijaksi (persistence manager), sillä sen avulla hallitaan muun muassa haku- ja tallennusoperaatioita.

Kuvasta 15 nähdään myös, että sovelluskerros (application) sisältää olioita, joiden edessä on sana "transient". Tämä tarkoittaa sitä, että oliot voivat pysyvyyden näkökulmasta olla tietyllä hetkellä pysyviä (persistent, P) tai transientteja (transient, T). Näiden kahden lisäksi oliot voivat myös olla irrallisia (detached, D). Kun olio on tilassa P, on Hibernate vastuussa olion tilan tallentamisesta tietokantaan. Tila D tarkoittaa sitä, että olio on aiemmin ollut tilassa P, mutta on "irrotettu" siitä, jolloin olion tila ei ole enää Hibernaten vastuulla. Olion tila voi siis muuttua hetkellisestä pysyväksi ja pysyvästä irralliseksi. Sama voi tapahtua myös toisinpäin, eli tila muuntuu irrallisesta jälleen pysyväksi ja edelleen hetkelliseksi. Tilojen T ja D oliot ovat Javan roskienkeruun piirissä. Näiden käsitteiden tunteminen ja tilasiirtymien ymmärtäminen on Hibernaten käytön kannalta erittäin olennaista [Bauer ja King, 2004].

Kuvassa 16 esitellään kokonaisvaltaisempi arkkitehtuuri, jossa Hibernaten olennaisimmista kirjastoista käytetään suurinta osaa.



Kuva 16. Kokonaisvaltainen arkkitehtuuri [Hibernate, 2004b]

Yllä olevan kuvan arkkitehtuurissa sovellus ei enää olekaan itse vastuussa JDBC-yhteyksien ja transaktioiden hallinnasta, vaan vastuu niistä on siirretty Hibernatelle. Mukaan on kuvassa 15 esiteltyyn arkkitehtuuriin verrattuna otettu rajapinnat **Transaction** ja **TransactionFactory** sekä **ConnectionProvider**.

Edellä puhuttiin yksittäisistä työn yksiköistä. Hibernatessa tällaista yksittäistä työn yksikköä kutsutaan transaktioksi (tässä on huomattava, että on olemassa erikseen sovellustransaktioita ja tietokantatransaktioita, jotka tarkoittavat toisinaan käytännössä samaa asiaa, mutta eivät aina). Transaktio päättyy aina joko sen vahvistamiseen (**commit**) tai perumiseen (**roll back**). Mikäli saman transaktion sisällä suoritetaan useita tietokantaoperaatioita, tulee transaktiolle ilmoittaa alku ja loppu. Jos transaktiota ei voida syystä tai toisesta suorittaa loppuun asti, perutaan kaikki sen aikana tehdyt toimenpiteet. **Transaction**-rajapinta abstrahoi **JDBC**, **JTA** (Java Transaction API) tai **CORBA** (Common Object Request Broker Architecture) transaktioiden käsittelyn. Sessio voi tarvittaessa käyttää samanaikaisesti useitakin transaktioita [Hibernate ref, 2004]. Transaktioiden käyttämiseen liittyy monia tärkeitä asioita, kuten isolaatiotasot ja erilaiset lukitusmenetelmät. Niiden ymmärtäminen on tärkeää, mutta ei tämän tutkielman kannalta olennaista. **Transaction**-rajapinnan avulla Hibernate-sovellusten alustan vaihtaminen on helppoa. **TransactionFactory** rajapinnan toteutuksen tehtävänä on luoda ilmentymiä **Transaction**-rajapinnan toteutuksista. **ConnectionProvider** puolestaan tarjoaa ja säilöö JDBC-yhteyksiä.

Tässä kohdassa mainittujen rajapintojen lisäksi muita tärkeitä Hibernaten rajapintoja ja ovat **Configuration**,- **Query**,- **Criteria**,- takaisinkutsu (**callback**),- sekä laajennus (**extension**)-rajapinnat. **Configuration**-olion kautta saadaan sovelluksessa selville kuvaukset sisältävien dokumenttien sijainti, sekä Hibernaten muut asetukset. Näiden tietojen perusteella voidaan sitten luoda

SessionFactory. Query ja Criteria rajapintoja käsitellään hieman tarkemmin kohdassa 5.5. Takaisinkutsurajapintojen avulla sovellus voi reagoida olioiden tilassa tapahtuviin muutoksiin. Interceptor on eräs tällainen rajapinta. Laajennusrajapintojen avulla Hibernate sallii useiden ominaisuuksien toteuttamisen itsenäisesti sen sijaan, että käytettäisiin sisäänrakennettuja menetelmiä. Tällaisia itse toteutettavia laajennusrajapintoja ovat IdentifierGenerator (avainten generointi), Dialect (SQL-kielen eri versioiden tulkinta), Cache ja CacheProvider (välimuististrategiat), ConnectionProvider (esitelty edellä), transaktioihin liittyvät rajapinnat (esitelty edellä), ClassPersister-rajapinnat (ORM-strategia), PropertyAccessor (ominaisuuksien käsittely) ja ProxyFactory (välittäjien luonti).

## 5.2. Konfigurointi ja käyttöönotto

Hibernaten käyttöönotto onnistuu lähes missä tahansa Java-ympäristössä riippumatta siitä, tarjoaako ympäristö automaattiset transaktiot, vai pitääkö sovelluksen huolehtia niistä itsenäisesti. Koska Hibernate pyrkii abstrahoimaan käytettyä ympäristöä, on Hibernaten käyttäminen samanlaista riippumatta käytettävästä ympäristöstä. Ainoa eroavaisuus eri ympäristöissä toimivien Hibernatea käyttävien sovellusten välillä on konfiguraatitiedostossa (hibernate.properties), johon määritellään ympäristöspesifiset asiat, kuten mikä tietokanta on käytössä, mikä JDBC-ajuri on käytössä, miten saadaan JDBC-yhteyksiä, kuinka suuri yhteyssäiliö (connection pool) on käytössä tai minkälainen strategia valitaan transaktioiden suhteen. Konfiguroitavuuden parhaita puolia on muun muassa ympäristöriippumattomuus, ylläpidon vaivattomuus sekä riippumattomuus kolmansista osapuolista.

## 5.3. Kuvausten määrittäminen

Hibernaten avulla on mahdollista käyttää kaikkia luvussa 3 esiteltyjä kuvaustapoja, tosin alakohdassa 3.7.4 esitettyä mallia ei ole Hibernaten vakio luokilla mahdollista toteuttaa, eikä sen käyttäminen ole suositeltavaa. Kuvaukset määritellään Hibernatessa tyypillisesti XML-muodossa (esimerkki 29, s.61). Kuvaustiedostot (hbm.xml) sisältävät metadatan, jonka avulla kerrotaan, miten kohdealueen mallin luokat, luokkien väliset assosiaatiot, luokkahierarkiat ja luokkien attribuutit, sekä joukot ja perintähierarkiat liitetään tietokantaan.

Hibernaten XML-tiedostojen rakenne on pyritty tekemään mahdollisimman yksinkertaiseksi. Tiedostoista on haluttu tehdä luettavia ja riippumattomia mistään erityisistä XML-editoreista. Kuvausten tekemisen helppous ja tiedoston yksinkertaisuus perustuu pitkälti Hibernaten käyttämiin järkeviin oletusarvoihin ja sen suorittamaan reflektioon perustuvaan parametrien arvaamiseen. Tämä tarkoittaa sitä, että jos käyttäjä ei ole määrittänyt

kuvaustiedostossa luokan attribuutille tietotyyppiä, tarkistaa Hibernate tietotyyppin luokan ohjelmakoodista ja asettaa tyyppin sen perusteella.

Kuvaukset on myös mahdollista toteuttaa XDoclet-tagien avulla (esimerkki 24) [XDoclet, 2004]. Tällöin kuvaukset sijoitetaan tageina luokan get-metodien kommentteihin. Eräs ongelma tagien kohdalla kuitenkin on, että tagikirjaston päivitykset eivät aina ole ajan tasalla uusimpien Hibernaten versioiden kanssa. Näin ollen XDoclettien avulla ei voida välttämättä tehdä kaikkea sitä, mitä käsin kirjoittaen voidaan. Eräs selkeä hyvä puoli on taas se, että kuvaukset ovat sovelluskoodin yhteydessä, mikä tarkoittaa sitä, että sovelluskehittäjän ei tarvitse editoida kahta tai useampaa tiedostoa samanaikaisesti toteuttaessaan kuvauksia. Toisesta näkökulmasta katsoen voidaan kuitenkin pitää parempana tapaa, jossa sovelluskoodi pidetään mahdollisimman minimaalisena, eli ohjelmakoodissa ei määritellä mitään muuta kuin olion tarvitsemat ominaisuudet. XDoclet generoi jokataapauksessa hbl.xml-tiedoston, jota Hibernate käyttää. Tästä johtuen XML-tiedoston rakenteen opetteleminen ei ole pahasta kenellekään huolimatta menetelmästä, jota sen tekemiseen käytetään. XDoclet-tageista ja niiden perusteella generoiduista XML-kuvauksista on esimerkkejä seuraavassa luvussa.

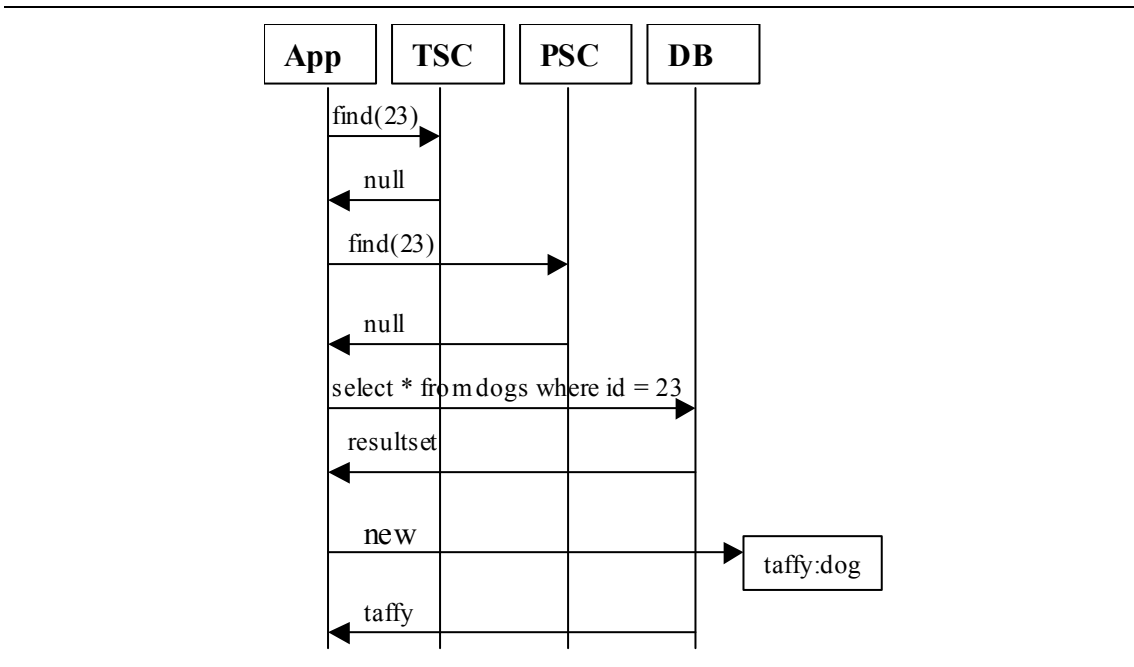
#### 5.4. Välimuistista

Kohdan 5.1 yhteydessä puhuttiin hieman erittäin olennaisista käsitteistä, sessioista ja transaktioista. Niihin liittyy vielä yksi asia, joka on toistaiseksi jäänyt käsittelemättä. Kyseessä on erittäin olennainen käsite liittyen ORM-menetelmään ja sen tehokkaaseen soveltamiseen käytännössä; välimuisti (cache). Välimuistin avulla saavutetaan selvää etua tehokkuudessa verrattuna vaikkapa suoraan JDBC-rajapinnan käyttämiseen. Välimuisti on myös olennainen tekijä sovelluksen skaalautuvuuden kannalta. Hibernate toteuttaa välimuistiarkkitehtuurin, joka perustuu seuraaviin, yleisesti tunnettuihin välimuistin toteutusperiaatteisiin.

Välimuisti on paikallinen kopio tietokannan tilasta. Kopio voi fyysisesti sijaita joko kiintolevyllä tai keskusmuistissa. Olennaista on se, että tietovarasto on sovelluksen lähellä. Erilaiset välimuistit voidaan jakaa kolmeen tyyppiin: Transaktiotason välimuisti (Transaction Scope Cache, TSC), prosessitason välimuisti (Process Scope Cache, PSC) ja klusteritason välimuisti (Cluster Scope Cache, CSC). TSC on liitetty yhteen työn yksikköön (unit of work). Se on myös voimassa ainoastaan yhden työn yksikön (unit of work) suorittamisen ajan. TSC on käytössä aina ja sen toteuttaa Identity Mapper (kohta 4.3). PSC on taas välimuisti, jonka useampi työn yksikkö jakaa keskenään. Tästä seuraa, että PSC-välimuistia käyttää mahdollisesti useampi samaan aikaan ajettava transaktio. CSC on puolestaan välimuisti, jonka jakaa useampi prosessi samalla tietokoneella tai jonka jakaa useampi tietokone samassa klusterissa (joukko

yhteen liitettyjä tietokoneita). CSC ei kovinkaan usein ole tarpeellinen, sillä sen käyttäminen on monesti vain marginaalisesti tavanomaista tietokannan käsittelyä nopeampaa.

Seuraavassa kaaviossa havainnollistetaan tapahtumaa, josta käytetään englanniksi termiä "cache miss", joka kaavion esimerkissä tapahtuu itse asiassa kahteen kertaan.



Kuva 17. "Cache miss"

Kaaviossa havainnollistettava "cache miss" (CM) tarkoittaa sitä, että kun oliota halutaan käyttää (kaaviossa dog-luokan "taffy"-oliota), tarkastetaan ensin, onko siitä kopiota joko TSC tai PSC-välimuisteissa. Kaaviossa ensimmäinen CM tulee etsittäessä oliota TSC-välimuistista ja toinen, kun sitä etsitään PSC-välimuistista. Oliio joudutaan lopulta hakemaan normaalisti tietokannasta.

Välimuistin käyttämisellä tavoitellaan vain yhtä asiaa; sillä pyritään välttämään turhia käyntejä tietokantaan. Eniten välimuistin käytöstä hyödytään tietysti sovelluksissa, jotka ovat "vain luku" (read only) tyyppisiä, ja joissa tietokannasta haetaan runsaasti tietoa. Siitä, että tietokantaa voi käyttäjien toimesta vain lukea, ei voi seurata ongelmia tietokannan samanaikaisen käsittelyn kanssa. Sen vuoksi välimuistin käyttö on tällaisissa tapauksissa hyödyllistä.

## 5.5. HQL ja Criteria

HQL (Hibernate query language) on SQL-kieltä muistuttava kyselykieli. HQL on SQL:stä eroten täysin oliopohjainen, eli se ymmärtää perintää, polymorfisuutta ja assosiaatioita [Hibernate ref]. HQL poikkeaa SQL:stä muun muassa siinä, että sen syntaksi on huomattavasti yksinkertaisempi. Hightowerin [Hightower, 2004] mukaan kieli on taas melko samankaltainen

JDO:n JDOQL:n, sekä myös EJBQL:n kanssa. HQL:n toteuttamisessa on pyritty minimaalisuuteen pyrkien silti säilyttämään kielen helppolukuisuus. Yksinkertaisin HQL-kysely on esimerkiksi seuraavanlainen:

---

```
1. from eg.Cat
```

---

#### Esimerkki 14. Yksinkertaisin HQL-lause

Esimerkin kysely palauttaa kaikki luokan eg.Cat ilmentymät. Verrattuna SQL:ään kyselyssä ei ole osaa "SELECT \*". Koska lauseessa ei kerran ole lainkaan ehtoja, on se loogista ymmärtää siten, että suoritetaan hakuoperaatio, joka palauttaa kaikki mainitun luokan ilmentymät.

Koska HQL on oliopohjainen kyselykieli, palauttaa esimerkin kysely myös kaikki luokan eg.Cat aliluokat, mikä taas ei ole aivan itsestään selvää. HQL kysely palauttaa siis kaikkien niiden pysyvien luokkien ilmentymät, jotka perivät luokan tai toteuttavat sen rajapinnan. Tällöin puhutaan polymorfisesta kyselystä.

HQL:n käyttäminen ei kuitenkaan ole pakollista, sillä tavallisten SQL-kyselyiden tekeminen on myös tarvittaessa mahdollista. Toinen vaihtoehtoinen tapa on käyttää Criteria-rajapintaa, jonka avulla voidaan kapseloida hakuehdot yhteen olioon, jonka perusteella kysely tehdään.

Criteria-rajapinta on Hibernateen kokeilumielessä toteutettu vaihtoehtoinen metodi HQL:lle kantakyselyiden toteuttamiseen. Tämä dynaaminen ja oliopohjainen kyselyiden toteuttamisen malli on mukana Hibernatessa, sillä se saattaa sopia HQL:ää paremmin sellaisille, joilla ei ole kokemusta SQL-kielen kaltaisesta syntaksista [Hibernate, 2004b]. Seuraava esimerkki havainnollistaa hieman Criteria-rajapinnan käyttämistä.

---

```
1. Criteria crit = session.createCriteria(Cat.class);
2. crit.add(Expression.eq("color", Color.BLACK));
3. List cats = crit.list();
```

---

#### Esimerkki 15. Criteria-rajapinnan käyttöä

Aluksi luodaan session-olion kautta Cat luokkaan sidottu Criterion ilmentymä. Sen jälkeen crit-olioon voidaan lisätä niin paljon siihen sidottua luokkaa koskevia ehtoja kuin tarvitaan. Edellisessä esimerkissä ehdoksi on laitettu ainoastaan se, että mukaan tulevien kissojen on oltava väriltään mustia. Staattisen Expression-luokan metodi eq() (equals) hoitaa tässä tarvittavan vertailun. Kun ehdot on syötetty, on itse kyselyn suorittaminen erittäin yksinkertaista, sillä kuten nähdään, on crit-oliolla metodi list(), joka palauttaa listan kaikista sellaisista luokan Cat ilmentymistä, jotka ovat mustia.

Muita Expression-luokan metodeita ovat muun muassa `like(String, String)`, `gt(String, Float)`, `in(String, Collection)`, `isNull(String)`, `and()`, `conjunction()`, `disjunction()`, `not(Criterion)` sekä `sql(String)`. Ensimmäinen toimii SQL:n like-operaattorin tavoin, eli ensimmäisenä parametrina ilmoitetaan, mitä haetaan ja jälkimmäisenä annetaan ilmaus, josta osa voi olla korvattu %-merkillä, jolloin se osa voi tarkoittaa mitä vain. Like-funktiota voitaisiin siis käyttää esimerkiksi seuraavasti:

---

```
1. Expression.like("address", "Hämeen%");
```

---

#### Esimerkki 16. Like

Tässä tulosjoukkoon tulisi vain sellaiset oliot, joiden osoitekentän alusta löytyisi merkkijono "Hämeen".

Metodi `gt` (greater than) puolestaan soveltaa annetun kentän annettuun arvoon rajoitetta "suurempi kuin". Ensimmäisenä parametrina annetaan vertailtavan ominaisuuden nimi ja toisena vertailtavan ominaisuuden arvo:

---

```
1. Expression.gt("weight", new Float(minWeight));
```

---

#### Esimerkki 17. Greater than

Tässä esimerkissä tulosjoukkoon tulisi sellaiset oliot, joilla paino on suurempi kuin erikseen määritelty minimipaino `minWeight`. Funktion `sql(String)` avulla voi taas kirjoittaa rajoitteen käsin ilmaistuna SQL-kielellä. `Conjunction()` ryhmittelee ilmaukset yhteen konjunkttiiviseen joukkoon ja `disjunction()` puolestaan disjunkttiiviseen.

In-funktio ottaa mukaan sellaiset oliot, joilla on arvo, joka kuuluu jälkimmäisenä annettuun joukkoon. Ensimmäisenä parametrina annetaan attribuutti, jonka arvoa halutaan vertailla, esimerkiksi

---

```
1. Expression.in("name", Arrays.asList(new
    String[]{"mikko", "niilo", "teppo"});
```

---

#### Esimerkki 18. In

Criteria rajapinta sisältää tietysti myös muita metodeita kuin `add`. Eräs hyvin käyttökelpoinen metodi on `addOrder(Order)`, joka saa parametrinaan `Order`-luokan ilmentymän. Nimestä voi päätellä, että funktio laittaa palautettavan listan parametrina annetun olion mukaiseen järjestykseen. Seuraavassa on esimerkki funktion käytöstä:

---

```
1. crit.addOrder(Order.asc("size"));
```

---

Esimerkissä lista järjestettäisiin nousevaan järjestykseen siten, että ensimmäisinä alkiolina listassa olisivat kooltaan pienimmät.

Käytännössä Criterionin käyttö voisi olla perusteltua esimerkiksi tilanteessa, jossa toteutetaan hakusivua, jossa käyttäjä voi syöttää tai valita erilaisista listoista lukuisia ehtoja suodattamaan tulosjoukkoa. Tällöin ehdot voitaisiin kätevästi lisätä add-metodikutsuilla crit-olioon ja ohjelmakoodista tulisi selkeämpää ja helppolukuisempaa. Sen lisäksi hakuehtoien muuttuminen ei vaikeuttaisi merkittävästi kyselyiden päivittämistä toisin kuin jos suodattaminen tehtäisiin esimerkiksi tavallisilla SQL-kyselyillä.

Criteriaa käytettäessä tärkeimmät luokat ovat siis Session, Criteria, Expression sekä Order. Kokonaisuutena tämä datan suodattamiseen tarkoitettu malli tarjoaa mielenkiintoisen ja olio-ohjelmoijille helposti omaksuttavan tavan käsitellä tietokantaa täysin oliomallin mukaisesti. Tässä esiteltyihin sekä muihin ominaisuuksiin voi tutustua tarkemmin esimerkiksi Hibernaten dokumentaation avulla [Hibernate, 2004b].

### 5.6. Proxy-mallin soveltamisesta

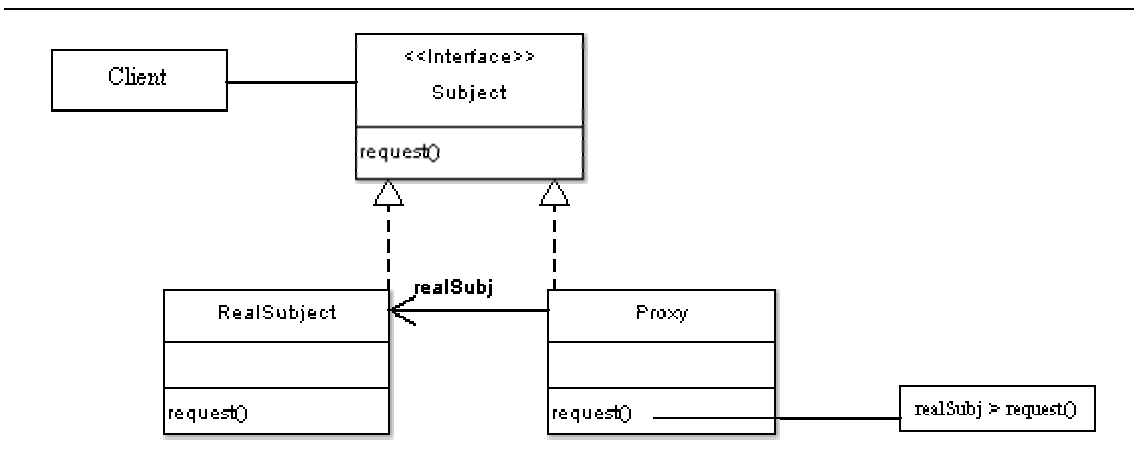
Hibernaten kuvauksiin on mahdollista konfiguroida erilaisia tiedonhakustrategioita (fetching strategies), joista tärkeimmät ovat "lazy" ja "eager". Tiedonhaulla tarkoitetaan sitä, että tietokannasta ladataan muistiin jokin haluttu olio, sekä muita olioita, joihin sillä mahdollisesti on viitteitä. Koska viitteet voivat periaatteessa jatkua loputtomiin kattaen pahimmassa tapauksessa koko tietokannan, halutaan oliograafi katkaista jostain kohtaa, jotta koko tietokantaa ei ladata muistiin. Graafin katkaiseminen automaattisesti, eli "lazy"-strategia, oli esillä jo edellä kohdassa 4.5, kun puhuttiin Lazy Load-mallista.

"Eager" taas on malli, jossa ohjelmoija voi eksplisiittisesti kertoa, mitkä kaikki oliot halutaan ladata tietyn olion mukana, eli toisin sanoen kertoa tarkalleen, mistä kohtaa graafi halutaan katkaista. Jos laiskaa hakua halutaan soveltaa, pitää luokalle sallia välittäjien käyttö asettamalla "lazy=true" luokan kuvaukseen. Mikäli joukolle halutaan sallia "joukko wrapperin", eli joukon oman välittäjän käyttö, pitää joukolle tehdä sama asetukset. Seuraavaksi tarkastellaan hieman perusteellisemmin Hibernaten käyttämän Proxy-suunnittelumallin soveltamista laiskassa haussa. Proxy on tekniikka, jonka avulla toteutetaan edellä mainittu "lazy"-optio.

Proxy tarkoittaa nimensä mukaisesti välittäjää, joka toimii asiakkaan ja varsinaisen luokan ilmentymän välillä. Asiakas suorittaa kaikki pyynnöt välittäjälle, joka ohjaa pyynnöt sille ja luo myös tarvittaessa olion varsinaisen

ilmentymän. Näin asiakkaan ei tarvitse koskaan käyttää suoraan varsinaista oliota, vaan ainoastaan sen välittäjää.

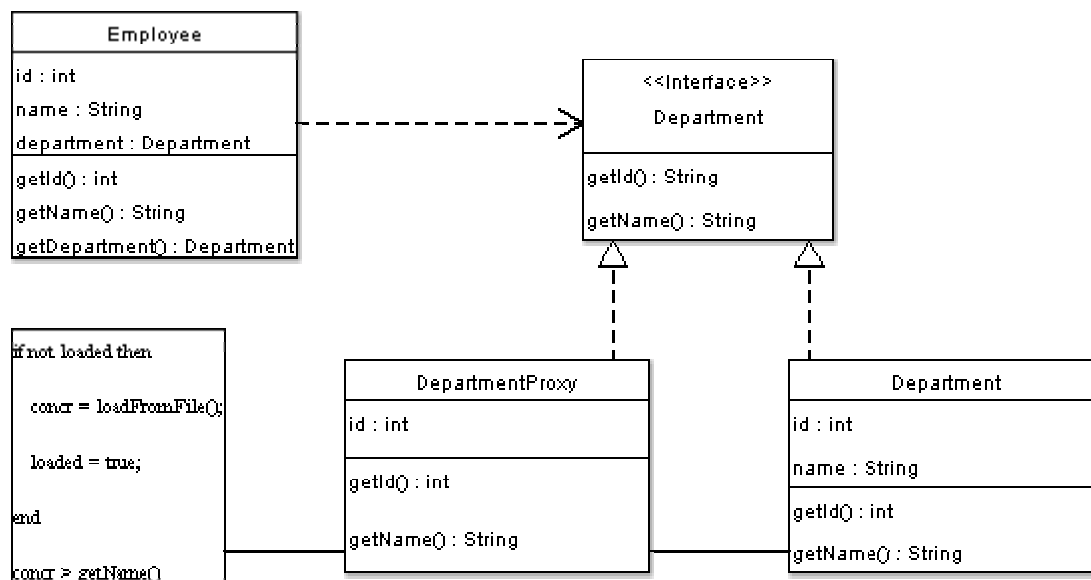
Välittäjän avulla kontrolloidaan siis suoraa pääsyä tiettyyn olioon. Koskimiehen [Koskimies, 2003] mukaan tätä suunnittelumallia saatetaan tarvita esimerkiksi silloin, kun olio on eri koneessa kuin sen palveluiden pyytäjä. Toinen mahdollinen syy on se, että olion käyttöön halutaan liittää erilaista monitorointia. Kolmas syy on taas se, että olion luontia halutaan lykätä mahdollisimman pitkälle, mikäli se on raskas toimenpide. Kuitenkin tärkein syy mallin käyttämiseen Hibernaten laiskan haun yhteydessä on se, että se on ainoa keino oliograafin katkaisemiseen.



Kuva 18. Proxy [Koskimies, 2003]

Hibernaten kohdalla mallin käyttäminen on perusteltua juuri siitä syystä, että olio voi olla suuri ja kallis luoda, mutta siihen halutaan kuitenkin viitata, sillä ei olla varmoja, tarvitaanko sen tarjoamia palveluita. Koskimiehen tekstiä mukaillen voidaan oliolle luoda heti edustaja, ja käyttää viitettä siihen kuin viitettä varsinaiseen olioon, jota ei siis vielä ole olemassa. Edelleen, kun olion palveluita tarvitaan, luo edustaja luokan ilmentymän ja siirtää palvelupyynnöt sille.

UML:n avulla välittäjä voidaan esittää kuvan 18 tavoin. Kaaviosta nähdään, että sekä Proxy, että RealSubject toteuttavat saman rajapinnan. Asiakas ei samasta rajapinnasta johtuen ole tietoinen siitä, onko kyseessä edustaja vai varsinaisen luokan ilmentymä. Välittäjä siirtää pyyntöjä varsinaiselle oliolle tehden samalla mahdollisia muita toimia [Koskimies, 2003].



Kuva 19. Proxy-mallin sovellus

Edellisessä kuvassa on esitetty yksinkertainen malli, jossa on mukana luokat Department, Employee, sekä DepartmentProxy, joka toimii välittäjänä varsinaiseen Department-luokkaan. Molemmat sekä välittäjä että varsinainen Department toteuttavat saman rajapinnan, jolloin Employee-luokan ilmentymällä ei ole tietoa siitä, kumpaa se käyttää. Oletetaan lisäksi, että tietokannassa on kaksi taulua, työntekijä ja osasto. Yhteen osastoon voi tässä esimerkissä kuulua useita työntekijöitä ja yksi työntekijä voi kerrallaan kuulua vain yhteen osastoon, joten suhde tässä on 1:n. Hibernaten mukainen syntaksi työntekijän hakemiseksi id:n perusteella on tällöin seuraava:

---

```
1. Employee employee = (Employee) session.load(Employee.class, id);
```

---

### Esimerkki 20. Työntekijän hakeminen

Osastoa voitaisiin ohjelmakoodissa alkaa käyttää seuraavasti:

---

```
1. Department department = employee.getDepartment();
```

---

### Esimerkki 21. Osaston hakeminen

Tässä vaiheessa Hibernate ei kuitenkaan luo luokan Department ilmentymää, vaan muodostaa välittäjän, jonka kautta varsinainen olio voidaan luoda tarvittaessa. Seuraavaksi voidaan hakea osaston pääavain:

---

```
1. department.getId();
```

---

### Esimerkki 22. Osaston id:n hakeminen

Vielä tässäkin ei tarvitse luoda ilmentymää varsinaisesta oliosta, sillä välittäjä sisältää jo id:n. Vasta, kun oliolta kysytään jotain muuta tietoa, kuten osaston nimeä, luodaan oikea ilmentymä Department luokasta:

---

```
1. department.getName();
```

---

### Esimerkki 23. Osaston nimen hakeminen

Tämänkin jälkeen pyynnöt kulkevat välittäjälle, joka edelleen siirtää pyynnöt varsinaiselle luokalle.

## 6. Hibernate käytännössä

Modernin Java-sovelluksen kerrosarkkitehtuuri sisältää useiden eri toimittajien komponentteja, joiden integroiminen ja käyttäminen ei välttämättä ole yksinkertaista. Siksi toisinaan tyydytään tekemään sovelluksia vanhemmilla menetelmillä, vaikka parempiakin olisi olemassa. Aikaa ei haluta käyttää uusien asioiden opetteluun. Tämä ei kuitenkaan ole viisasta, sillä esimerkiksi juuri ORM:n ja sitä soveltavan Hibernaten merkityksen huomaa vasta, kun sitä osataan käyttää ja siitä on saatu riittävästi kokemusta.

Tämän luvun eräs tarkoitus onkin havainnollistaa käytännössä, miten yksinkertaista Hibernaten käyttöönottoaminen ja peruskäyttäminen on. Tulevissa kohdissa Hibernate asetetaan osaksi kerrosarkkitehtuuria, jolloin ORM-kerroksen sijainti ja rooli arkkitehtuurissa selkeytyvät. Esimerkkitsovelluksen rakentamisen yhteydessä esitellään myös muitakin hyödyllisiä välineitä.

Fowler toteaa Vennersin [Venners, 2002] artikkelissa seuraavasti: "Aluksi on helpompaa tehdä konkreettinen sovellusalueeseen liittyvä esimerkki ja sen jälkeen muunnella (refactor) esimerkkiä soveltumaan myös yleisempiin tilanteisiin." Tämän luvun yksi motivaatio on täsmälleen sama; esimerkin ymmärtämällä helpottuu kokonaisuuden yleinen ymmärtäminen.

### 6.1. Esimerkkisovellus

Seuraavaksi esiteltävä esimerkkisovellus on toteutettu käyttäen hyväksi muun muassa Spring-kehystä [Spring, 2004] ja tietysti tämän tutkielman kannalta olennaisinta, eli Hibernatea. Esimerkkisovelluksen päätason arkkitehtuuri muodostuu kerroksista, joista jokaisella on oma vastuualueensa. Koskimiehen [Koskimies, 2003] mukaan kerrosarkkitehtuuri koostuu jonkin abstrahointiperiaatteen mukaan järjestetyistä tasoista. Hän jatkaa, että tyypillisesti tämä periaate on skaala laitteesta ihmiseen. Ihmistä lähempänä olevat tasot ovat korkeammalla,- ja laitetta lähempänä olevat tasot ovat matalammalla abstraktiotasolla. Kerrostus voi tietysti perustua muihinkin näkökulmiin. Esimerkkisovelluksen arkkitehtuurissa sovelletaan Spring-kehysten mukaista MVC-mallia (Model View Controller), jonka avulla käyttöliittymä ja sovellusdata ovat toisiinsa nähden aina yhtenäisessä tilassa.

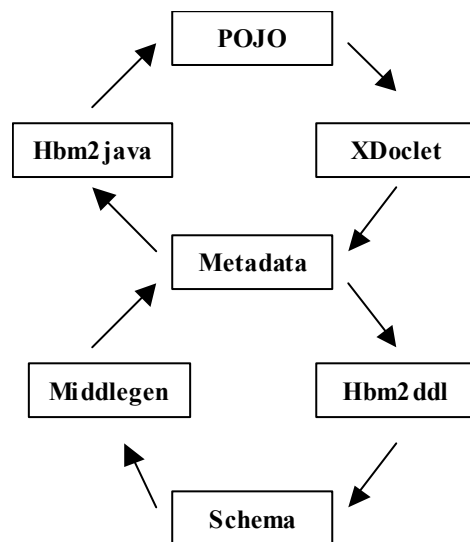
Sovelluksessa toteutetaan aluksi pysyvyyden toteuttava kerros, eli tehdään tarvittavat dataluokat ja liitetään ne Hibernaten avulla tietokantaan. Esimerkin pääpaino on tutkielman aiheen vuoksi juuri pysyvyyserroksen toteuttamisessa. Tulevissa kohdissa kerrotaan kuitenkin myös lyhyesti siitä, miten luodaan liiketoimintaloginen,- ja käyttöliittymäkerros.

Esimerkkisovelluksen ominaisuudet ovat seuraavat:

1. Sovelluksella voidaan lisätä ja poistaa työntekijöitä sekä osastoja.
2. Työntekijälle voidaan myös lisätä ja poistaa tehtäviä.
3. Samaa tehtävää ei voida määrätä kuin yhdelle henkilölle kerrallaan.
4. Jos työntekijän poistaa, tulee tähän liitettyjen tehtävienkin poistua.
5. Tehtävällä on oltava tieto tekijästään.
6. Sellaista osastoa, jolla on työntekijöitä, ei voi poistaa.
7. Työntekijällä on muiden ominaisuuksiensa lisäksi myös yksi osoite, joka esitetään omana luokkanaan, mutta yhtenä työntekijän attribuuttina.

## 6.2. Pysyvyyskerros

Esimerkkisovelluksen pysyvyyskerroksen rakentamisessa edetään top-down ajattelun mukaisesti. Kuvassa 20 tämä tapa on kuvattu oikealla puolella. Aluksi kirjoitetaan sovelluksen kohdealueen luokat. Luokkiin lisätään sitten metatietoa, jonka avulla generoidaan Hibernaten hbm.xml-tiedostot, joiden perusteella generoidaan edelleen tarvittavat tietokantataulut.



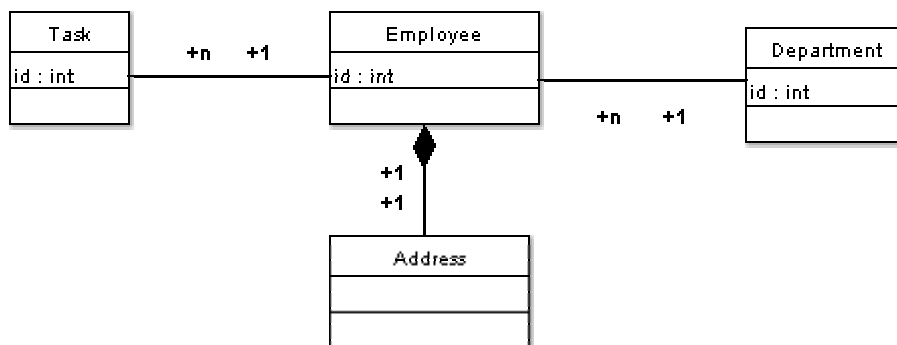
Kuva 20. "Roundtrip"-kehitys

Kaavion vasemmalla puolella aloitetaan pohjalta. Siinä tietokantakaaviota käytetään metadatan generoimiseen, minkä perusteella generoidaan edelleen kohdealueen luokat. Vasemmanpuoleista mallia, jossa olemassa olevan tietokantakaavion perusteella edetään kohti POJO-olioita, ei käsitellä tässä tutkielmassa.

### 6.2.1. Kohdealueen oliomalli, XDocletit ja POJO-oliot

Kohdealueen oliomalli koostuu luokista Employee, Department, Task ja Address. Kukin työntekijä kuuluu täsmälleen yhteen osastoon ja kullakin

osastolla voi olla useita työntekijöitä. Kuvassa 21 ei ole mainittu muita attribuutteja kuin id-attribuutit, sillä sen ainoa tarkoitus on havainnollistaa luokkien välisiä assosiaatioita.



Kuva 21. Kohdealueen luokkakaavio

Kun ympäristö on saatu kuntoon, on ensimmäinen ohjelmointityötä edellyttävä vaihe toteuttaa oliomallin POJO-luokat Employee, Department, Task ja Address. Seuraavassa koodilistauksessa näytetään osa luokan Employee ohjelmakoodista XDoclet-tageilla [XDoclet, 2004] varustettuna. Merkillä '@' kerrotaan, mistä tagista on kyse. Sen jälkeen ilmoitetaan tarvittavat Hibernaten parametrit. Tagien avulla suoritetaan Hibernaten formaatin mukaisten XML-kuvauksen generointi. Ohjelmakoodista on jätetty pois joitakin esimerkin kannalta epäolennaisia set-funktioita.

---

```

1. /** @hibernate.class table = "employees" */
2. public class Employee implements Serializable {
3.
4.     private long id = -1;
5.     private String firstName = "";
6.     private String lastName = "";
7.     private int age = 0;
8.     private Address address = null;
9.     private Department department = null;
10.    private Set tasks = new HashSet();
11.
12.    public Employee() {}
13.
14.    public Employee(String fullName, int age, Address adr, Department dep)
15.    {
16.        this.setFullName(fullName);
17.        this.age = age;
18.        this.address = adr;
19.        this.department = dep;
20.    }
21.    /**
22.     * @hibernate.id generator-class="native" unsaved-value="-1"
23.     */
24.    public long getId() {
25.        return id;
26.    }
27.
28.
29.    /** @hibernate.property */
30.    public String getFullName() {
31.        return firstName + " " + lastName;
32.    }
33.

```

---

---

```

34. public void tokenize(String name) {
35.     StringTokenizer tokenizer = new StringTokenizer(name);
36.     this.firstName = tokenizer.nextToken();
37.     this.lastName = tokenizer.nextToken();
38. }
39.
40. /** @hibernate.property */
41. public int getAge() {
42.     return age;
43. }
44.
45. /** @hibernate.component */
46. public Address getAddress() {
47.     return address;
48. }
49. public void setAddress(Address address) {
50.     this.address = address;
51. }
52.
53. /** @hibernate.many-to-one column="department_id" */
54. public Department getDepartment() {
55.     return department;
56. }
57. public void setDepartment(Department department) {
58.     this.department = department;
59. }
60.
61. /**
62.  * @hibernate.set inverse="true" lazy="true" cascade="all"
63.  * @hibernate.collection-key column="employee_id"
64.  * @hibernate.collection-one-to-many class="fi.solita.test.model.Task"
65.  */
66. public Set getTasks() {
67.     return tasks;
68. }
69. }

```

---

#### Esimerkki 24. Employee-luokka ja XDoclet-tagit

Employee-luokka kuvataan "entity" tyypiksi tagilla `@hibernate.class` (esimerkki 24, rivi 1). Tagille annetaan myös parametriksi tietokantataulun nimi. Kuten kuvauksista näkyy, ovat ne hyvin yksinkertaisia. Tämä johtuu siitä, että XDoclet ja Hibernate asettavat tietyt oletusarvot kaikille tarvittaville parametreille. XDocletin oletusarvot ovat samoja kuin Hibernaten, joten voidaan yhtä hyvin puhua Hibernaten oletusasetuksista. Oletusasetuksia ei kuitenkaan ole pakko ylikuormittaa. Jos Employee-luokan kuvaukseen kirjoitettaisiin myös oletusasetukset, näyttäisi kuvaus seuraavalta:

---

```

1. /**
2.  * @hibernate.class table = "employees" dynamic-update="false"
3.  *                   dynamic-insert="false"
4.  */

```

---

#### Esimerkki 25. Employee-kuvaus ja oletusasetukset

Työntekijälle luodaan keinotekoinen avain (surrogate key). Tällaisen avaimen käyttäminen on hyvä käytäntö useimmissa tapauksissa. Id-parametrin määrittämisessä kerrotaan, että yksilölliset tunnisteet generoidaan "native"-metodilla. Esimerkiksi PostgreSQL:n tapauksessa "native" tarkoittaa sekvenssin "hibernate\_sequence" käyttämistä. Kyseinen sekvenssi on luotava

tietokantaan käsin, mikäli ei käytetä skeeman automaattisesti luovaa Hbm2ddl kirjastoa. Lisäksi ilmoitetaan tallentamattoman rivin id, -1.

Etu,- ja sukunimi kuvataan esimerkin vuoksi samaan tietokannan sarakkeeseen merkitsemällä metodi "getFullName" tagilla `@hibernate.property`. Luokalla voi siis olla useita attribuutteja ja mikään ei estä tallentamasta niitä samaan sarakkeeseen tietokannassa. Luokan sisäistä ominaisuuksien toteutusta ei aina tarvitse näyttää luokan ulkopuolelle. Tämänkin seikan vuoksi on kannattavaa käyttää get- ja set-metodeita (accessor methods). Normaalisti etunimeä ja sukunimeä ei kuitenkaan kannata käsitellä esimerkin näyttämällä tavalla, vaan niitä on parempi käsitellä erikseen. Metodin getFullName kuvaus on myös hyvin yksinkertainen, sillä tietokantaan luotavan sarakkeen nimeksi laitetaan automaattisesti "fullname" ja sen pituus määräytyy Hibernaten määrittämän pituuden oletusarvon mukaan. Vaihtoehtoisesti voisi tämänkin myös kirjoittaa `@hibernate.property column="fullname" length="100"`. Oletusarvoihin nojautuminen lyhentää OR-kuvauksiin liittyvää ohjelmakoodia merkittävästi.

Osoite kuvataan sovelluksessa omana luokkana, joka sisältää osoitteelle tavanomaisia kenttiä, kuten kadun ja postinumeron. Työntekijälle voidaan ilmoittaa vain yksi osoite, jonka on myös poistuttava, mikäli kyseinen työntekijä poistetaan. Osoite on Hibernaten termejä käyttäen luokan Employee komponentti. Tämä tarkoittaa sitä, että luokan Address elinkaari riippuu luokan Employee elinkaaresta ja sillä ei ole omaa yksilöivää tunnistetta. Address on siis luokka, joka on arvo-tyyppiä (value type) oleva attribuutti. Employee on puolestaan tyyppiä "entity", sillä sen ilmentymillä on yksilölliset avaimet, eivätkä ilmentymät poistu automaattisesti, vaikka ne omistava osasto poistettaisiinkin. Luokkaan Address ei tarvitse lisätä muita tageja kuin sen sisältämien get-metodien normaalit tagit. Seuraavassa esimerkissä on esimerkki kadun nimen merkitsemisestä "property"-tagilla.

---

```

1. /** @hibernate.property */
2. public String getStreetName() {
3.     return streetName;
4. }
```

---

#### Esimerkki 26. getStreetName()

Employee on n:1-suhteen Employee-Department n-puoli. Tämän kuvaamiseksi Employee-luokkaan voidaan esitellä Department-tyyppinen jäsenmuuttuja ja määritellä tagi `@hibernate.many-to-one`. Tagiin voidaan liittää ominaisuus, jossa kerrotaan, minkä niminen sarake tietokantaan halutaan luoda viittaamaan työntekijän osastoon. Tämä merkitään seuraavasti: `"column="department_id"`. Jos työntekijän ei tarvitse tietää osastoaan, ei suhteesta kuitenkaan välttämättä tarvitse tehdä kaksisuuntaista, eikä

Department-luokan lisääminen jäsenmuuttujaksi, sekä kuvauksen tekeminen Employee luokkaan ole pakollista.

Viimeinen Employee-luokan kuvaus koskee työntekijän tehtäviä. Tässä suhteessa Employee on "1"-osapuoli, kun taas luokka Task on suhteen "n"-osapuoli. Nyt tarvitaan tagia @hibernate.set, jolle ilmoitetaan set-määrittäyksille tyypilliset parametrit inverse ja lazy, jotka molemmat merkitään tosiksi. Inverse-avainsanalla ilmoitetaan, että tehtävien joukko on n:1-suhteen toinen puoli. Ilman kyseistä avainsanaa Hibernate yrittäisi suorittaa kaksi erilaista SQL-lausetta, jotka molemmat päivittäisivät samaa vierasavaimen kenttää, kun assosiaatiota käsiteltäisiin ajon aikana. Tämä aiheuttaisi ymmärrettävästi arvaamattomia seurauksia.

Koska Task on tyyppiä "entity", eli sillä on oma id, pitää valita "cascade"-strategia. Tässä tapauksessa halutaan, että kun isäntäolio poistetaan, halutaan myös orvoiksi jääneet lapsioliot poistaa. Sen vuoksi strategiaa kuvataan nimellä "all". Kaksi jäljelle jäävää tagia edustavat Hibernaten <key> ja <one-to-many> tageja, mutta ne ovat niiden "collection"-versiot, koska tässä kuvataan joukkoa. Joukon avaimeksi ilmoitetaan employee\_id ja luokan sisältämien alkioiden tyyppiä Task. Task-luokasta käsin tämä suhde kuvataan seuraavasti:

---

```

1. /** @hibernate.many-to-one column="employee_id" */
2. public Employee getEmp() {
3.     return emp;
4. }
```

---

### Esimerkki 27. Luokan Task Employee-suhteen kuvaaminen

Luokan Employee avulla on nyt esitelty tyypillisiä tapoja toteuttaa ominaisuuksien, assosiaatioiden ja joukkojen kuvauksia. Myös komponentin käsite on tullut selväksi. Seuraavaksi esitellään vielä ote luokan Department get-funktioista, jotka eivät edellä esitellyn perusteella kaipaa lisäselvityksiä.

---

```

1. /** @hibernate.class table = "departments" */
2. public class Department implements Serializable {
3.     . . .
4.     /**
5.      * @hibernate.id
6.      * generator-class="native"
7.      * unsaved-value="-1"
8.      */
9.     public long getId() {
10.         return id;
11.     }
12.
13.     /** @hibernate.property */
14.     public String getName() {
15.         return name;
16.     }
17.
18.     /**
19.      * @hibernate.set cascade="all" lazy="true" inverse="true"
20.      * @hibernate.collection-key column="department_id"
21.      * @hibernate.collection-one-to-many
22.      * class="fi.solita.test.model.Employee"

```

---

---

```
24. public Set getEmployees() {
25.     return employees;
26. }
27. . . .
28. }
```

---

### Esimerkki 28. Ote Department-luokan ohjelmakoodista

#### 6.2.2. Kuvausten generointi

Seuraava vaihe on tuottaa tarvittavat kuvaustiedostot (hbm.xml-tiedostot) ja ne generoidaan XDoclet-tagien perusteella. Apuna tässä käytetään Apachen Mavenia [Maven, 2004]. Hyvin lyhyesti sanottuna Maven on Java-projektin hallintatyökalu. Ajamalla komennon *maven java:compile* projektin juurikansiossa, generoituvat tarvittavat tiedostot ennalta määrättyyn kohteeseen, joka tässä tapauksessa on asetettu osoittamaan oliomallin sisältävään "model"-kansioon.

---

```
1. <hibernate-mapping>
2.   <class name="fi.solita.test.model.Employee" table="employees"
3.     dynamic-update="false" dynamic-insert="false">
4.
5.     <id name="id" column="id" type="long" unsaved-value="-1">
6.       <generator class="native"></generator>
7.     </id>
8.
9.     <property name="fullName" type="java.lang.String"
10.      update="true" insert="true" column="fullName"/>
11.
12.     <property name="age" type="int" update="true" insert="true"
13.      column="age"/>
14.
15.     <component name="address" class="fi.solita.test.model.Address">
16.
17.       <property name="streetName" type="java.lang.String"
18.        update="true" insert="true" column="streetName"/>
19.
20.       <property name="streetNumber" type="java.lang.String"
21.        update="true" insert="true" column="streetNumber"/>
22.
23.       <property name="zipCode" type="java.lang.String" update="true"
24.        insert="true" column="zipCode"/>
25.
26.       <property name="city" type="java.lang.String" update="true"
27.        insert="true" column="city"/>
28.
29.       <property name="country" type="java.lang.String" update="true"
30.        insert="true" column="country"/>
31.
32.     </component>
33.
34.     <many-to-one name="department"
35.      class="fi.solita.test.model.Department"
36.      cascade="none" outer-join="auto" update="true"
37.      insert="true" column="department_id"/>
38.
39.
40.     <set name="tasks" lazy="true" inverse="true"
41.      cascade="all-delete-orphan" sort="unsorted">
42.       <key column="employee_id"/>
43.       <one-to-many class="fi.solita.test.model.Task"/>
44.     </set>
45.
46.   </class>
47. </hibernate-mapping>
```

---

---

```
49. ( To add non XDoclet property mappings, create a file named
50.  hibernate-properties-Employee.xml containing the additional properties
51.  and place it in your merge dir.)
```

---

### Esimerkki 29. Employee.hbm.xml

Tiedostojen Department.hbm.xml ja Task.hbm.xml sisältöä ei ole tarpeellista esitellä tarkemmin, sillä ne eivät sisällä Employee.hbm.xml-tiedostoon verrattuna mitään uutta.

#### 6.2.3. Tietokantakaavion generointi

Hibernaten mukana tulee paketti "net.sf.hibernate.tool.hbm2ddl", joka sisältää SchemaExport (SE)-luokan. Luokka toimii komentoriviltä ajettavana itsenäisenä sovelluksena, tai se voidaan upottaa tarvittaessa myös toiseen sovellukseen. Kaavion (skeeman) generointi on näin ollen mahdollista myös ajonaikaisesti.

SE käyttää generointiin hbm.xml-tiedostoja, jotka esimerkkisovelluksessa on generoitu XDoclet-tagien perusteella. Tietokantaskeeman generointia voidaan siis ohjata editoimalla kuvauksia. Olennainen skeeman generointiin vaikuttava elementti on sarakkeen modifiointiin tarkoitettu "column", joka voidaan laittaa esimerkiksi "id"- tai "property"-elementtien sisälle.

Jos esimerkiksi haluttaisiin luoda työntekijän nimen mukainen indeksi nopeuttamaan nimen mukaisia hakuja, voitaisiin luokan Employee getFullName-funktion tagiin tehdä seuraava muutos:

---

```
1. /**
2.  * @hibernate.property
3.  * @hibernate.column name="fullName" length="300" not-null="true"
4.  *                      index="IND_FULLNAME"
5.  */
6. public String getFullName() {
7.     return firstName + " " + lastName;
8. }
```

---

### Esimerkki 30. getFullName()

Column-tagiin on indeksin lisäksi määritelty tietokantaan luotavan sarakkeen nimi, pituus ja null-arvojen salliminen. Hbm2ddl tekee sarakkeesta automaattisesti tyyppiä VARCHAR, jos ominaisuus on POJO:ssa tyyppiä String. Nyt siis kentän "fullName" pituudeksi tulee 300 merkkiä ja se ei salli null-arvoja. Kentän nimeä ja pituutta ei välttämättä tarvitse ilmoittaa, sillä SchemaExport käyttää erilaisia mahdollisimman rationaalisia oletusarvoja, kuten esimerkiksi yllä olevassa tapauksessa nimi-sarakkeen nimeksi tulisi automaattisestikin "fullname". Jos kentän pituutta ei ilmoiteta, tulee pituudeksi 255.

Työkalua on mahdollista käyttää komentoriviltäkin, mutta esimerkkisovelluksessa käytetään Mavenin Hibernate lisäosan toimintoa (goal)

*hibernate:schema-export*. Lisäosan käyttäminen on yksinkertaista, sillä sitä varten tarvitsee ainoastaan muokata hieman kahta tiedostoa:

1. Mavenin alustustiedostoon `project.properties` lisätään seuraavan rivi:

```
maven.hibernate.properties ${basedir}/src/conf/hibernate.properties
```

2. Hibernaten alustustiedostoon `hibernate.properties` lisätään seuraavat rivit:

```
hibernate.connection.driver_class org.postgresql.Driver
hibernate.connection.url jdbc:postgresql://$server$/$dbname$
hibernate.connection.username $username$
hibernate.connection.password $password$
```

Nyt kun projektin juuressa ajetaan komento *maven hibernate:schema-export*, syntyy tietokanta haluttuun paikkaan. Skripti, jolla kanta luotiin, taltioidaan myös tiedostoon. Seuraavassa esimerkissä on taulun "employees" luontilause.

---

```
1. create table employees (
2.     id int8 not null,
3.     fullName varchar(255) not null,
4.     age int4,
5.     streetName varchar(255),
6.     streetNumber varchar(255),
7.     zipCode varchar(255),
8.     city varchar(255),
9.     country varchar(255),
10.    department_id int8,
11.    primary key (id)
12.)
```

---

### Esimerkki 31. Taulun "employees" generoitu luontiskripti

Kuten nähdään, ovat myös luokan Address kentät mukana sarakkeina, koska luokka oli Employees luokan komponentti. Samoin mukana on viite työntekijän osastoon, sekä not-null määreet työntekijän nimelle ja id:lle.

#### 6.2.4. DAO-luokkien toteuttaminen

Edellä on toteutettu esimerkkisovelluksen pysyvyyden toteuttava kerros, sekä tietokanta, johon oliomallin pysyvyyttä tarvitsevat luokat on sidottu Hibernaten avulla. Seuraavaksi rakennetaan DAO-luokat (Data Access Object).

DAO on standardi J2EE-suunnittelumalli [Sullivan, 2003]. DAO ei kuitenkaan ole samalla tavalla formaali suunnittelumalli kuin esimerkiksi Proxy tai Factory [Gamma *et al.*, 1995]. Oikeastaan puhuttaessa DAO-mallista yleensä, puhutaan tavasta jäsentää sovelluskoodia siten, että pääsy tietokantaan on eristetty omaksi kokonaisuudekseen.

DAO-luokkien avulla voidaan alemman tason tietokannan käsittely erottaa liiketoimintalogiikasta. DAO-luokat sisältävät tyypillisesti niin sanotut CRUD-operaatiot (create, read, update, delete), eli luonti, luku, päivitys ja poistooperaatiot [Sullivan, 2003]. Tyypillinen DAO-toteutus sisältää factory-

luokan, rajapinnan, rajapinnan toteuttavan luokan sekä mahdollisesti DTO (data transfer object)-olioita [Sullivan, 2003].

DTO-olioiden tehtävänä on nimensä mukaisesti toimia olioina, jotka kapseloivat siirrettäviä tietokokonaisuuksia. Bauerin ja Kingin [Bauer ja King, 2004] mukaan DTO-luokkien käyttämiseen liittyy kuitenkin ongelmia; ne aiheuttavat muun muassa ohjelmakoodin monistamista. Bauer kuitenkin jatkaa, että niiden käytölle löytyy myös perusteita; ne auttavat muun muassa erottamaan www- ja liiketoimintalogiikka-kerrokset toisistaan, sekä niitä käyttämällä varmistetaan, että näkymän käyttämä data on haettu ennen kontrollin palauttamista www-kerrokselle. Esimerkissä on muodostettu omat DAO-luokat työntekijälle, osastolle ja työntekijän tehtäville. Työntekijän DAO:n rajapinta on seuraavanlainen:

---

```

1. public interface EmployeeDao {
2.     void addEmployee(Employee employee);
3.     Employee getEmployee(long id);
4.     void updateEmployee(Employee employee);
5.     void removeEmployee(long employeeId);
6.     List getAllEmployees();
7. }
```

---

### Esimerkki 32. Employee-luokan DAO:n rajapinta

Ote EmployeeDao-rajapinnan toteutuksesta on esitetty seuraavassa esimerkissä.

---

```

1. public class EmployeeDaoImpl extends HibernateDaoSupport
2.     implements EmployeeDao {
3.
4.     public void addEmployee(Employee employee) {
5.         getHibernateTemplate().save(employee);
6.     }
7.
8.     public Employee getEmployee(long id) {
9.         return (Employee) getHibernateTemplate().load(Employee.class,
10.             new Long(id));
11.     }
12.
13.     public void updateEmployee(Employee employee) {
14.         getHibernateTemplate().update(employee);
15.     }
16.
17.     public void removeEmployee(long id) {
18.         Employee employee = getEmployee(id);
19.         getHibernateTemplate().delete(employee);
20.     }
21.
22.     public List getAllEmployees() {
23.         return getHibernateTemplate().find("from Employee emp");
24.     }
25.     . . .
26. }
```

---

### Esimerkki 33. Ote DAO-luokan toteutuksesta

Kuten huomataan, on operaatioiden toteutus hyvin lyhyt. Tämä johtuu apuna käytettävän Spring-kehiksen tarjoaman Hibernate-tuen käyttämisestä.

HibernateDaoSupport tarjoaa runsaasti Hibernaten käyttämistä helpottavia funktioita.

Spring on todella monipuolinen ja dynaaminen kehys, jonka tarjoamia palveluita tämänkin luvun esimerkkiä tehdessä on käytetty runsaasti. Esimerkin kannalta olennaisinta on tietää, että Spring sisältää muun muassa tuen ORM-työkaluille, kuten Hibernatelle, JDO:lle ja iBatis-kehykselle, sekä laajan kehysten www-sovelluksille. Jotta Spring tietäisi hbm.xml-tiedostojen sijainnin, pitää tiedostoon applicationContext-hibernate.xml lisätä seuraavassa esimerkissä olevat rivit.

---

```

1. . . .
2.
3. <value>org/TestApp/model/Employee.hbm.xml</value>
4. <value>org/TestApp/model/Department.hbm.xml</value>
5.
6. . . .
7.
8. <bean id="employeeDAO"
9.     class="org.appfuse.persistence.hibernate.EmployeeDAOHibernate">
10.    <property name="sessionFactory"><ref
11.        local="sessionFactory"/></property>
12. </bean>
13. <bean id="departmentDAO"
14.     class="org.appfuse.persistence.hibernate.DepartmentDAOHibernate">
15.    <property name="sessionFactory"><ref local="sessionFactory"/></property>
16. </bean>

```

---

### Esimerkki 34. Informaatiota Spring-kehykselle

#### 6.3. Palvelukerros

Palvelukerrokseen laitetaan yleensä kohdealueen looginen toiminnallisuus, eli niin sanottu liiketoimintalogiikka (business logic), esimerkiksi tässä tapauksessa palvelukerros osaa muodostaa uuden työntekijän, joka lisätään alempien kerrosten palveluita hyväksikäyttäen tietokantaan.

Palvelukerroksen palveluilla on myös muiden kerrosten tapaan julkinen rajapinta, jonka toteuttavaan luokkaan upotetaan varsinainen looginen ohjelmakoodi. Työntekijän muodostamista ja sitä seuraavan palvelupyynnön suorittamista varten on palvelurajapintaan määritelty addEmployee (EmployeeData)-funktio, joka on toteutettu palvelurajapinnan toteuttavaan luokkaan seuraavasti:

---

```

1. public void addEmployee(EmployeeData data) {
2.     Department dep =
3.     departmentDao.getDepartment(data.getDepartmentId());
4.     Employee emp = new Employee(data.getFullName(), data.getAge(),
5.     data.getAddress(), dep);
6.     employeeDao.addEmployee(emp);
7. }

```

---

### Esimerkki 35. Yksi luokan EmployeeServiceImpl funktio

Metodissa nähdään selvästi, miten alemman DAO-kerroksen palveluita käytetään hyväksi hakemalla aluksi työntekijään liitettävä osasto DAO:n kautta. Seuraavaksi luodaan uusi työntekijä käyttämällä operaatioissa Employee-luokan toista rakenninta. Lopuksi käytetään taas DAO-kerroksen palveluita, eli kutsutaan kerroksen vastaavaa työntekijän lisäävää funktiota.

#### 6.4. Www-kerros

Tässä kohdassa esitellään lyhyesti käyttöliittymän toteutukseen liittyviä asioita. Käyttöliittymä ei sinänsä kuulu tutkieman aihepiiriin, mutta täydellisyyden vuoksi asiasta kerrotaan jonkin verran. Esimerkkisovelluksen käyttöliittymä toteutetaan www-pohjaisena. Tyypillisen J2EE:tä soveltavan arkkitehtuurin www-kerros pohjautuu servletteihin, jotka generoidaan sovelluskehittäjän tekemien JSP-sivujen pohjalta [Johnson, 2003]. Kuten aiemmin mainittiin, sovelletaan arkkitehtuurissa Springin web-kehiksen (lyhennetty tässä myöhemmin SWF) mukaista MVC-mallia.

##### 6.4.1. DispatcherServlet

SWF on suunniteltu DispatcherServlet:in (DS) pohjalle. DS jakaa viestejä, joita järjestelmään rekisteröidyt käsittelijät osaavat käsitellä. DS konfiguroidaan \$servlet\_name\$.xml-tiedoston avulla. DS:n konfigurointitiedosto sisältää muun muassa kontrollereiden riippuvuudet ja URL-kuvaukset, joiden avulla tietty URL sidotaan tiettyyn kontrolleriin. Seuraavassa koodiesimerkissä on ote konfigurointitiedostosta. Oteessa on kuvattu kontrollerin UpdateEmployeeController riippuvuudet.

---

```

1. <bean id="updateEmployeeController"
2.   class="fi.solita.test.controller.emp.UpdateEmployeeController">
3.   <property name="employeeService">
4.     <ref bean="employeeService" />
5.   </property>
6.   <property name="departmentService">
7.     <ref bean="departmentService" />
8.   </property>
9.   <property name="commandClass">
10.    <value>
11.      fi.solita.test.controller.emp.EmployeeForm
12.    </value>
13.  </property>
14.  <property name="commandName">
15.    <value>form</value>
16.  </property>
17.  <property name="formView">
18.    <value>employee/update-employee</value>
19.  </property>
20.  <property name="successView">
21.    <value>update-employee-success.action</value>
22.  </property>
23.</bean>

```

---

Esimerkki 36. UpdateEmployeeController-luokan riippuvuudet

## 6.4.2. Kontrollerit

Kontrollerit ovat osa MVC-mallia. Niissä määritellään joko suoraan sovelluksen käyttäytyminen, tai tarjotaan pääsy luokkiin, joissa sovelluksen käyttäytyminen toteutetaan. Kun käyttäjä vaikuttaa käyttöliittymän kautta sovelluksen toimintaan, on kontrolleri se osa MVC-mallista, joka tulkitsee käyttäjän syötteen, muokkaa sitä tarvittavalla tavalla ja päivittää uusilla tiedoilla mallia, minkä sisältö lopulta esitetään käyttäjälle uutena näkymänä [Johnson *et al.*, 2004].

Spring sisältää useita laajennettavia kontrolleri luokkia, joista eräs esimerkisovelluksessa käytetty on nimeltään SimpleFormController. SimpleFormController on komentopohjainen kontrolleri, jonka avulla voidaan tehdä lomake (form) ja sitä vastaavan data olio (DTO). SimpleFormController-luokan avulla voidaan muun muassa määrittää komento-olio, lomaketta vastaavan näkymän nimi, seuraavan näkymän nimi ja niin edelleen [Johnson *et al.*, 2004]. UpdateEmployeeController perii SWF:n luokan SimpleFormController. Luokan UpdateEmployeeController toteutus näkyy esimerkissä 37.

---

```

1. public class UpdateEmployeeController extends SimpleFormController {
2.
3.     private EmployeeService employeeService;
4.     private DeptService departmentService;
5.
6.     protected Object formBackingObject(HttpServletRequest request)
7.         throws Exception {
8.
9.         long id = RequestUtils.getRequiredLongParameter(request, "id");
10.        Employee employee = employeeService.getEmployee(id);
11.        return createForm(employee);
12.    }
13.
14.    protected Map referenceData(HttpServletRequest request)
15.        throws Exception {
16.
17.        Map map = new HashMap();
18.        List departments = departmentService.getAllDepartments();
19.        map.put("departments", departments);
20.        return map;
21.    }
22.
23.    private EmployeeForm createForm(Employee employee) {
24.        EmployeeForm form = new EmployeeForm();
25.        form.setId(employee.getId());
26.        form.setAddress(employee.getAddress());
27.        form.setAge(employee.getAge());
28.        form.setDepartmentName(employee.getDepartment().getDisplayName());
29.        form.setFullName(employee.getFullName());
30.        return form;
31.    }
32.
33.    protected ModelAndView onSubmit(Object employeeCommand)
34.        throws Exception {
35.
36.        EmployeeData employeeData = (EmployeeData) employeeCommand;
37.        employeeService.updateEmployee(employeeData);
38.        return new ModelAndView(new RedirectView(getSuccessView()));
39.    }
40.
41.    public void setEmployeeService(EmployeeService employeeService) {
42.        this.employeeService = employeeService;
43.    }

```

---

---

```
45.     public void setDepartmentService(DeptService deptService) {
46.         this.departmentService = deptService;
47.     }
48.
49. }
```

---

### Esimerkki 37. UpdateEmployeeController

#### 6.4.3. JSP-sivut

JSP-sivut toimivat esimerkkisovelluksen MVC-mallin V-komponenttina. JSP-sivut sisältävät JSP-tageja, HTML-koodia ja puhdasta Java ohjelmakoodia. Kun käyttäjä kutsuu JSP-sivua ensimmäisen kerran, generoi palvelin sivua vastaavan servletin, joka on valmis suorittamaan tehtäviä käyttäjän pyynnöstä.

Kun http-pyyntö suoritetaan, etsitään aluksi `WebApplicationContext` (WAC), joka sisältää joitakin `www`-sovelluksen tarvitsemia ominaisuuksia. Seuraavaksi WAC sidotaan pyyntöön, jotta kontrolleri ja muut pyyntöketjun komponentit pystyvät käyttämään sitä. Seuraavissa vaiheissa pyyntöön sidotaan muita tarvittavia asioita, muun muassa ominaisuus, joka mahdollistaa `www` käyttöliittymän monikielisyden (locale resolver). Sitten, mikäli sopiva käsittelijä löytyy, suoritetaan käsittelijän komentoketju mallin valmistamiseksi. Jos malli palautuu, näkymä renderöidään käyttämällä apuna näkymän ratkaisijaa (view resolver) [Johnson *et al.*, 2004].

## 7. Kokemuksia Hibernaten käyttämisestä Solitan projekteissa

Solita on käyttänyt Hibernatea useissa projekteissa aina vuoden 2003 alusta alkaen. Sinä aikana kehykseen on ehtinyt tutustua jo useita työntekijöitä. Tässä luvussa käydään läpi joillekin Hibernatea käyttäneille Solitan työntekijöille tehdyn kyselyn tuloksia. Käyttökokemukset ovat yleisesti olleet hyvin positiivista. ”Enemmän, nopeammin. Asiakas saa enemmän samalla rahalla”, sanoo muun muassa yksi kyselyyn vastanneista. Ohjelmakoodin ja virheiden määrän on yleisesti todettu vähentyneen sekä tehokkuuden kasvaneen. Hibernate on vastanneiden mielestä pääsääntöisesti helpottanut tietokantalioiden ohjelmointia ja tehnyt siitä nopeampaa ja joustavampaa. Joitakin hankaluuksia on ilmennyt ja niistä kerrotaan tarkemmin kohdassa 7.3. Sitä ennen kerrotaan vastanneiden mielipiteitä Hibernaten soveltuvuusalueista ja sen tuomasta lisäarvosta projekteille. Kohdassa 7.4 vastataan kysymykseen, onko Hibernate tuonut parannusta Solitalla aiempiin käytettyihin menetelmiin. Lopuksi kokemukset vedetään yhteen taulukoiden avulla.

### 7.1. Soveltuvuus

Hibernate soveltuu erityisesti tietokantavetoisiin projekteihin, joissa tietokantaan voidaan ennakoida tulevan runsaasti muutoksia kehityksen aikana. Hibernatea voidaan käyttää lähes kaikissa sellaisissa pysyvyyttä vaativissa sovelluksissa, joissa ei tarvita täydellistä kontrollia suoritettusta SQL:stä. Hibernate sopii hyvin myös esimerkiksi sellaiseen projektiin, jossa kaikkien toteutukseen osallistuvien täytyy käyttää tietokantapalveluita, vaikka osalla projektiryhmästä ei olisikaan erityisen hyvää tietokantaosaamista. Tällaisissa tapauksissa kokeneen käyttäjän vastuulle jää toteuttaa kuvaukset, jolloin muiden ei tarvitse välttämättä olla erityisen tietoisia, mitä oikeastaan tapahtuu. Asiaan liittyy myös kääntöpuoli, josta lisää myöhemmin.

Yleisesti ottaen Hibernate sopii sellaisiin järjestelmiin, joilla on rikas kohdealueen malli sekä paljon transaktioita, jotka koskettavat suhteellisen pientä osaa koko järjestelmän datamäärästä. Rikas malli on sellainen kohdealueen oliomalli, jolla on älykkäitä dataolioita, eli olioita, jotka sisältävät paljon loogisia olion tilaa muuttavia funktioita. Rikkaassa mallissa käytetään myös oliopohjaisuuden ominaisuuksia laajalti ja järkevästi hyväksi. Tällaisissa tapauksissa Hibernate on parhaimmillaan. Yleistettynä Hibernate soveltuu siis lähes kaikkiin sovelluksiin, joiden taustalla on tietokanta, mutta sen käyttäminen ei aina kaikissa sellaisissa sovelluksissa ole perusteltua.

Hibernatea taas ei luonnollisesti käytetä projekteissa, joissa ei tarvita pysyvyyttä, tosin joitakin poikkeuksia on, sillä Hibernatesta voidaan käyttää myös pientä osaa, kuten tutkielman viidennestä luvusta on käynyt ilmi.

Hibernate ei myöskään sovellu sellaisiin sovelluksiin, joissa tietokantaa ei haluta käsitellä olioina. Tällaisia ovat muun muassa raportteja generoivat sovellukset. Toinen sovellusalue, jossa Hibernaten käyttö ei ole suositeltavaa, on eräajot (batch). Myöskään pienissä, vähän tietokantaa käsittelevissä sovelluksissa Hibernaten käyttöönottoaminen ei ole kannattavaa. Tällöin tavanomainen, suora JDBC:n käyttäminen on perusteltua. Hibernatea ei siis yleisesti kannata käyttää silloin, mikäli käytetään vain vähän transaktioita, jotka koskevat suurta osaa tietokannan datasta (kuten raportointisovelluksissa).

Hibernate toimii kyllä edellä mainituissakin tapauksissa, mutta niitä varten on olemassa parempiakin työkaluja.

## 7.2. Hyvät puolet

Hibernate on mahdollistanut hyvien olio-ohjelmointitapojen noudattamisen kohdealueen malliin. Hibernate on myös tuonut läpinäkyvyyttä pysyvyyteen, eli kätkenyt tietokannan sovellukselta lähes kokonaan. Tätä ominaisuutta voidaan pitää hyvien olio-ohjelmointitapojen noudattamisen edellytyksenä. Lisäksi järjestelmistä on tullut testattavampia. Myös optimoinnista on tullut helpompaa johtuen osittain Hibernaten välimuistimallista, joka on hyvin tehokas ja helposti konfiguroitavissa. Oliot ovat myös aina ajan tasalla, ja Hibernate myös varmistaa, että samasta rivistä ei muodosteta useampia olioita.

Erään vastanneen mielestä Hibernate on tehnyt tietokannan käsittelystä todella helppoa. Hän toteaa seuraavaa: "Jos toinen projektin jäsen kirjoittaa Hibernate spesifiset osat, ei toisen tarvitse optimi tapauksessa edes ajatella taustalla olevaa tietokantaa". Eräs toinen vastanneista taas toteaa, että olemassa olevaan oliomalliin on helppoa integroida ominaisuudet pysyvyyden toteuttamiseksi. Muutamassa vastauksessa lisäarvoksi mainitaan myös se, että Hibernate korvaa J2EE-spesifikaatiossa määritellyt EB:t (entity bean).

Eräs vastanneista mainitsee, että erittäin merkittävä Hibernaten tuoma lisä on mahdollisuus vaihtaa tietokantaa hyvin vähällä vaivalla. Myös useiden tietokantojen samanaikainen käyttäminen on aiempaa vaivattomampaa. Vaikka tietokannan kohtalaisen helppo vaihtaminen on mahdollista myös JDBC:n avulla, on siihen verrattuna Hibernaten tuomia lisäarvoja muun muassa ylläpidettävyyys, ohjelmakoodin yksinkertaisuus ja perustoimintojen helppo-käyttöisyys, hän kertoo.

Yksi vielä mainitsematon tärkeä hyöty on myös se, että motivaatio kasvaa, kun käytetään hyvää ja sovelluksen kehittämistä helpottavaa komponenttia. Hibernaten käyttäminen on yleisesti ottaen nopeuttanut kaikin puolin järjestelmän toteutusta. Liiketoiminnan kannalta hyvin tärkeä seikka on edellä mainittujen tekijöiden suora seuraus, kustannussäästöt. Optimitapauksessa päästäänkin selkeisiin kustannussäästöihin, mikäli Hibernatea osataan hyödyntää tehokkaasti.

### 7.3. Ongelmia ja ratkaisuja

Kaikesta hyvästä huolimatta Hibernaten käyttöön liittyy myös hankaluuksia, kuten lähes minkä tahansa työvälineen käyttöön. Eräänä lähes kaikkien vastanneiden mielestä hankalana asiana Hibernaten kohdalla on koettu olevan korkea oppimiskynnys. Työkalu ei useimpien mielestä ole erityisen yksinkertainen ja sen toiminnan tehokas ja hyödyllinen käyttäminen edellyttää runsasta harjoittelua ja kokemusta. Toisaalta taas oppimiskynnys on suhteellisen pieni siihen nähden, mitä kehys tekee.

Jos Hibernaten toimintaa ei tunneta kunnolla, saatetaan joutua ylimääräisiin ja yllättäviinkin vaikeuksiin. Tästä esimerkkinä yksi vastanneista mainitsee ensimmäisen Solitan Hibernatea soveltaneen projektin, jossa välimuistia jouduttiin konfiguroimaan hyvän suorituskyvyn takaamiseksi vain sen vuoksi, että osa sovelluksesta ehdittiin aikataulukiireiden vuoksi toteuttamaan ymmärtämättä Hibernaten toimintaa riittävän syvällisesti. Toisaalta konfigurointi ei vastanneen mukaan ollut kuitenkaan liian työlästä ja tänä päivänä kyseisen sovelluksen suorituskyky on erinomainen.

Hibernaten suorittamat tietokantapäivitykset ovat myös aiheuttaneet jonkin verran päänvaivaa. Eräs vastanneista kertoo esimerkiksi Hibernaten INSERT-lauseesta: Hibernate suorittaa aluksi INSERT-lauseen, jossa se ei täytä kaikkia kenttiä. Seuraavaksi se suorittaa UPDATE-lauseen, jossa päivitetään loput kentät. INSERT on saattanut epäonnistua, mikäli osa kentistä ei ole sallinut null-arvoja. UPDATE-lauseen olisi myöhemmin pitänyt päivittää kyseiset kentät. Tämä on aiheuttanut sen, että osa kentistä on pitänyt muuttaa sallimaan null-arvoja, mikä on taas johtanut siihen, että alkuperäisestä tietokantasuunnitelmasta on jouduttu poikkeamaan.

Hibernaten kanssa voidaan hyvinkin joutua tekemisiin edellä kuvatun kaltaisten ongelmien kanssa. Tässä kuvattu ongelma on tosin selvinnyt, sillä se johtui pienestä viasta ohjelmakoodissa. Vian korjaamisen jälkeen ei siis ole ollut tarpeellista poiketa alkuperäisestä tietokantasuunnitelmasta. Tämä kuitenkin kuvastaa ongelmaa, mikä liittyy abstraktiotason nousemiseen; sovelluksen toiminnan ymmärtäminen vaikeutuu. Ohjelmoijalle tämä näkyy siten, että ohjelmakoodista käsin kaikki vaikuttaa olevan kunnossa, mutta siitäkään huolimatta sovellus ei toimi oikein.

Abstraktiotason nouseminen näyttää olevan tähänkin lukuun saatujen vastausten ja haastattelujen perusteella eräs suurimmista tekijöistä useimpiin ongelmiin, joita Hibernaten, mutta myös muiden kehysten käyttäminen tuo mukanaan. Tämä johtuu ensinnäkin siitä, että ohjelman kontrollin kulkemisen seuraaminen tulee vaikeammaksi, koska kehys toimii "mustana laatikkona" (black box), joka saa sisäänsä syötteitä, käsittelee ne omalla tavallaan piilossa sovellukselta laatikon sisällä, ja antaa tulosteita. Avoimen lähdekoodin

kehysten kanssa työskenneltäessä kontrollin kulkemisen seuraaminen on kuitenkin helpompaa, sillä esimerkiksi Eclipse-sovelluskehittimen avulla Hibernaten binääritiedostoihin voi liittää lähdekoodit, jolloin kontrollin ajon aikainen seuraaminen on mahdollista kehittimen debug-toiminnolla aina Hibernaten tasolle asti. Debug-toiminnon käyttäminen ei tietysti ole pakollista, sillä ongelman voi usein selvittää tutustumalla hyvin oletetun ongelman ratkaisun sisältävään kohtaan kehiksen ohjelmakoodissa. Vertauksena Hibernatelle mainittakoon tässä kohtaa maksullinen ORM-työkalu TopLink, jonka mukana lähdekoodia ei saa. Tällöin kyseessä on täysin musta laatikko, jonka sisäistä logiikkaa on mahdotonta tutkia. Tämän voi kuvitella aiheuttavan hankaluuksia.

Abstraktiotason nouseminen tarkoittaa toisekseen myös usein generoitua ohjelmakoodia, joka Hibernaten tapauksessa on SQL-lauseita. Näiden automaattisesti tuotettujen lauseiden ymmärtäminen on tuottanut useimmille hankaluuksia. Esimerkkinä mainittakoon, että toisinaan Hibernate suorittaa aluksi DELETE-lauseen ja vasta sitten INSERT-lauseen. Miksi se ei tee suoraan UPDATE-lausetta? Syy siihen, miksi näin tapahtuu, riippuu tilanteesta. Ongelma kuvastaa kuitenkin hyvin niitä kysymyksiä, joihin Hibernaten kanssa työskennellessä joudutaan joskus etsimään vastauksia.

”Hibernate ei myöskään toimi erityisen hyvin, mikäli tietokannan tauluilla ei ole yksiosaisia pääavaimia”, kertoo eräs kyselyyn vastanneista. Erään toisen vastanneen mukaan Hibernaten käyttöönotto ei välttämättä ole paras valinta sellaisissa projekteissa, joiden tietokanta ei ole hyvin suunniteltu. Eräässäkin projektissa vastaan tullessa tietokannassa oli hänen mukaansa paljon redundanssia ja tauluja ilman pääavaimia, sekä siinä käytettiin useita erilaisia tapoja samojen asioiden esittämiseksi vieläpä siten, että eri tavoilla saatiin erilaisia tuloksia. Tällaisessa tapauksessa olisi saattanut olla helpompaa käyttää esimerkiksi iBatis-kehystä [iBatis, 2004] tai suoraan JDBC-rajapintaa, hän toteaa. Toisten tekemien, laajojen ja useiden XML-kuvaustiedostojen lukeminen ja tulkitseminen on myös koettu joidenkin kohdalla hankalaksi. Tällaiseen tilanteeseen joutuu usein sellainen henkilö, joka menee mukaan jo käynnissä olevaan projektiin.

#### **7.4. Parannusta aiempiin menetelmiin?**

Erään aiemmin Solitalla käytetyn työkalun ominaisuudet edellyttivät, että jokaista tietokannassa olevaa taulua kohti jouduttiin generoimaan 2-3 luokkaa, mikä tarkoitti sitä, että luokkia tuli todella paljon. Jos tauluihin tehtiin pieniäkin muutoksia, jouduttiin kaikki luokat generoimaan uudestaan. Kun tietokantaa muuteltiin ja uudet generoidut luokat laitettiin versionhallintaan, ei sovellus enää yllättäen toiminutkaan niillä, jotka eivät olleet hakeneet uusinta versiota versionhallinnasta. Näin kertoo yksi vastanneista. Hänen mukaansa

Hibernate on huomattavasti ennen käytettyä työkalua joustavampi, eikä käytön yhteydessä esiinny edellä mainitun menetelmän ongelmia, sillä Hibernaten käyttö ei edellytä ylimääräisten luokkien generoimista.

Hibernate on Prevaierin ohella erään vastanneen mukaan ainoa menetelmä, joka tarjoaa hyvän oliopohjaisen kohdealueen mallin. Hibernate on myös EJB:n lisäksi ainoa menetelmä, mikä tarjoaa kunnon tuen välimuistin käytölle eri kerroksissa ainoastaan konfigurointia muuttamalla. EJB:n heikkouksia Hibernaten nähdessä ovat monimutkaisuus ja riippuvuus säiliöstä (container). Pelkkä puhdas JDBC-rajapinnan käyttäminen ei myöskään ole erityisen kannattavaa, sillä jokaisessa projektissa joudutaan usein tekemään samoja asioita uudestaan ja uudestaan. Tämä on selvää ajan haaskausta. Pelkkään JDBC-rajapinnan käyttämiseen nähdessä säästetään Hibernaten avulla aikaa ja vaivaa, helpotetaan testaamista ja ylläpitoa, sekä parannetaan sovelluksen laatua.

## 7.5. Yhteenveto

Vastauksista voi päätellä, että ORM-menetelmää toteuttava Hibernate on yleisesti otettu käyttöön mielenkiinnolla ja toivoen, että se parantaisi tehokkuutta, helpottaisi ohjelmointityötä ja parantaisi ohjelmien laatua. Johdon kannalta Hibernaten on tietysti toivottu aiheuttavan kustannussäästöjä.

Vastauksissa esiintyy kuitenkin myös esimerkkejä Hibernaten liittyvistä pienistä hankaluuksista, joista tässä luvussa on edellä kerrottu. Osaan vastauksista mainituista hankaluuksista törmätään kuitenkin lähes aina työskennellessä relaatiotietojen parissa, eikä niillä ole varsinaisesti tekemistä Hibernaten ominaisuuksien kanssa; huonosti suunnitellut ja laajat tietokannat aiheuttavat aina tiettyjä vaikeuksia, oli niiden käsittelyyn käytetty menetelmä mikä tahansa.

Suurin ongelma tällaisissa tietokannoissa lienee niiden vaikeaselkoisuus, joka aiheuttaa sen, että tietokannan rakenteen oppiminen ja ymmärtäminen on kohtuuttoman hankalaa. Tällöin saattaa tulla houkutus olla opettelematta tietokannan rakennetta kunnolla. On selvää, että myös aikataulupaineet rajoittavat oppimiseen ja asioiden ymmärtämiseen käytettävissä olevaa aikaa. Mikäli ORM-kuvauksia aletaan kuitenkin muodostaa ymmärtämättä riittävän hyvin, mitä tehdään, on selvää, että siitä seuraa hankaluuksia. Jos tähän lisätään vielä se, että kuvausten tekijä ei ole ehtinyt täysin opetella Hibernaten käyttöä eikä näin ymmärrä sen toimintaa, seuraa vielä isompia vaikeuksia.

Vastausten yleisestä linjasta päätellen, on Hibernaten todettu olevan käyttökelpoinen työkalu, joka on lunastanut sille asetettuja odotuksia, mutta jättänyt myös seuraaville versioille parantamisen varaa. Seuraavaan taulukkoon on vielä koottu Hibernaten soveltuvuuteen liittyvät asiat.

Taulukko 3. Hibernaten soveltuvuus

Soveltuu	Ei sovellu
Tietokantavetoisiin projekteihin	Projekteihin, joissa ei tarvita pysyvyyttä
Mikäli tietokantaan voidaan ennakoida tulevan runsaasti muutoksia	Projekteihin, joissa tietokantaa ei haluta käsitellä olioina
Pysyvyyttä vaativiin sovelluksiin, joissa ei tarvita täydellistä kontrollia suoritetusta SQL:stä	Raportointisovelluksiin
Jos kaikkien toteutukseen osallistuvien täytyy käyttää tietokantapalveluita, mutta osalla ei ole vahvaa tietokantaosaamista	Eräajoihin
Yleisesti: Sellaisiin sovelluksiin, joilla on rikas kohdealueen malli sekä paljon transaktioita, jotka koskettavat suhteellisen pientä osaa koko järjestelmän datamäärästä	Yleisesti: Sellaisiin sovelluksiin, joilla on vain vähän transaktioita, jotka koskevat suurta osaa tietokannan datasta.

Seuraavaan taulukkoon on puolestaan koottu Hibernaten hyvät puolet.

Taulukko 4. Hibernaten hyvät puolet

Hyvät puolet
Mahdollistanut hyvien olio-ohjelmointitapojen noudattamisen
Tuonut läpinäkyvyyttä pysyvyyteen
Järjestelmistä tullut testattavampia
Optimointi helpompaa
Oliot ajan tasalla
Samasta rivistä ei muodosteta useita olioita
Tehnyt tietokannan käsittelystä kaikin puolin helpompaa
Olemassa olevaan oliomalliin on helppoa integroida ominaisuudet pysyvyyden toteuttamiseksi
Korvaa EB:t
Mahdollisuus vaihtaa tietokantaa hyvin vähällä vaivalla
Useiden tietokantojen samanaikainen käyttö helpottuu
Ylläpidettävyys
Koodin yksinkertaistuminen
Perustoimintojen helppokäyttöisyys
Motivaatio paranee
Nopeuttaa järjestelmän toteutusta
Tämä kaikki aiheuttaa kustannussäästöjä

Viimeiseen taulukkoon on koottu Hibernaten käyttämiseen liittyviä ongelmia. Ongelmien viereisestä sarakkeesta löytyy ratkaisuehdotus ongelmaan. Huomautettakoon vielä, että monet taulukossa mainituista ongelmista ovat seurannaisvaikutuksia kohtalaisen suuresta oppimiskynnyksestä.

Taulukko 5. Hibernaten ongelmia ja ehdotuksia niiden ratkaisemiseksi

Ongelma	Ratkaisu
Korkea oppimiskynnys	Tehokas opiskelu ja harjoittelemine aivan kuten minkä tahansa muunkin uuden asian kohdalla. Oppimiskynnys on pieni siihen nähden, mitä kehys tekee
Hibernaten suorittamien tietokantapäivitysten ymmärtäminen saattaa olla vaikeaa	Kokemuksen karttuessa kehysten toiminta hahmottuu
Alkuperäisestä tietokantasuunnitelmasta on joskus jouduttu poikkeamaan	Kannattaa yrittää korjata ohjelmakoodia, jotta muutokset voidaan pitää sovellustasolla. Hibernate on hyvin joustava, mutta kaikkien sen ominaisuuksien tunteminen vaatii pitkää kokemusta
Abstraktiotason nousemisen seurauksena ymmärtäminen vaikeutuu	Tähän ei ole ratkaisua. Niin vaan voi käydä, kun abstraktiotasoa nostetaan
Konrollin kulkemisen seuraaminen hankaloituu	Avoimen lähdekoodin kehysten kanssa työskenneltäessä kontrollin kulkemisen seuraaminen ja kehittäminen yleensä on kuitenkin helpompaa
Generoidun SQL:n ymmärtäminen vaikeaa	Vielä vaikeampaa olisi toteuttaa käsin tiettyjä, hyvin monimutkaisia lauseita. Parhaimmista tapauksissa generoitua SQL:ää ei edes tarvitse lukea
Ei toimi hyvin, Mikäli tietokannan tauluilla ei ole yksiosaisia pääavaimia	Joissain tapauksissa kannattaa harkita vaihtoehtoisista kehystä (esim.yksinkertainen JDBC-kehys iBatis) tai suoraa JDBC-rajapinnan käyttämistä
Huonosti suunniteltu tietokanta aiheuttaa vaikeuksia	Vaihtoehtoiset ratkaisut ja erilaisten menetelmien yhdisteleminen saattaa olla avuksi myös tällaisissa tapauksissa. Toisaalta huonosti suunniteltu tietokanta aiheuttaa aina ongelmia menetelmistä riippumatta
Toisten tekemien XML-kuvausten tulkitseminen työlästä	XDoclet-tagien käyttäminen ja pian julkaistavan Hibernate 3:n tukemien annotaatioiden käyttäminen helpottavat tätä ongelmaa
Rekursiivisten suhteiden ja erilaisten "cascade"-strategioiden yhteiskäyttö saattaa aiheuttaa vaikeuksia	Kokemus auttaa myös tämän ongelman ratkaisemisessa

## 8. Yhteenveto

Tutkielmassa on esitelty erilaisia pysyvyyden toteuttamisen menetelmiä Java-sovelluksissa, joista tarkimmin on käsitelty ORM-menetelmää. Olio- ja relaatioteknologioiden sulavan yhdistämisen on huomattu olevan mahdollista, kun tiedetään, mitä ollaan tekemässä ja käytetään oikeita työkaluja. Relaatiotietokantojen käyttämisen sovellusten taustalla on todettu olevan edelleen paras vaihtoehto useimmissa tapauksissa monestakin syystä. Tärkeä syy on muun muassa se, että relaatiotietokantaa voidaan käyttää integraatio-kerroksena useiden sovellusten välillä.

Hyväksi ORM-työkaluksi on ehdotettu Hibernate-kehystä, joka on toteutettu ORM-menetelmää koskevien teorioiden ja toimiviksi havaittujen suunnitelmallien pohjalta. Hibernaten avoin lähdekoodi, laaja käyttäjäkunta, hyvä dokumentaatio ja aktiivisesti eteenpäin viетävä kehitystyö takaavat kehysten laadun myös tulevaisuudessa. Omien sovellusten laadun osatekijä on lisäksi se, että hyviä kehysiä käyttämällä tulee käyttäneeksi alempien tason komponentteja parhaimmalla mahdollisella tavalla. Hibernate pyrkii esimerkiksi käsittelemään erilaisia tietokantoja ja JDBC-rajapintaa mahdollisimman optimaalisesti.

Tutkielmassa on lisäksi todettu, että Hibernate on myös käytännössä aiheuttanut positiivisia vaikutuksia tutkielmaa tukevan yrityksen, Solita Oy:n, ohjelmistokehitysprosessille. Näitä vaikutuksia ovat muun muassa pysyvyyttä varten kirjoitettavan ohjelmakoodin määrän huomattava vähentyminen ja laadun parantuminen, sekä tästä seuraava sovellusten ylläpidettävyyden paraneminen. Myös taustalla olevan tietokantatoimittajan vaihtaminen on tullut merkittävästi aiempaa helpommaksi. Liiketoiminnan näkökulmasta tärkein asia on taas se, että tämä kaikki on tuonut kustannussäästöjä. Olio- ja relaatiomallin yhdistämisen mahdollistavan ORM-menetelmän on näin todettu tajoavan paljon muita menetelmiä paremman mallin pysyvyyden toteuttamiseksi useimmissa, erityisesti suuremmissa Java-sovelluksissa.

Tutkielmassa on kuitenkin myös havaittu, että pysyvyyden toteuttamiseen liittyvät ongelmat eivät poistu itsestään. Esimerkiksi Fowler [Fowler, 2003] toteaa seuraavaa: "Älä oleta, että ORM-työkalu tekee kaiken työn puolestasi. Sen käyttäminen vähentää työmäärää kyllä huomattavasti, mutta tulet silti havaitsemaan, että työkalun konfigurointi ja käyttäminen vaatii joka tapauksessa edelleen paljon työtä." Työtä vaaditaan edelleen muun muassa hyvän tietokantakaavion suunnittelussa ja OR-kuvausten laatimisessa. Eniten työtä vaatii kuitenkin ORM-työkalun ominaisuuksien ja käyttämisen opiskelu. Korkea oppimiskynnys mainittiin edellisessäkin luvussa erääksi Hibernaten

huonoksi puoleksi. Toisaalta kynnyksen todettiin olevan kuitenkin suhteellisen pieni siihen nähden, mihin kaikkeen kehys pystyy.

Hibernate kehittyy jatkuvasti. Kehys tulee muun muassa toteuttamaan tulevaisuudessa julkaistavan EJB 3.0-spesifikaation [King, 2004a]. Hibernaten pian julkaistava versio 3.0 sisältää myös paljon uusia ominaisuuksia (tämän hetkinen versio on 2.1.7). Eräs tärkeimmistä on muun muassa parametrisoitavat suodattimet, joiden avulla voidaan tarkastella oliograafeja, jotka ovat vain osajoukkoja koko tietokannan sisältämästä tiedosta. Myös kuvausten tekemisestä on pyritty tekemään joustavampaa, jotta voitaisiin tukea mahdollisimman laajaa skaalaa erilaisia, jopa erittäin huonosti suunniteltuja tietokantoja.

King [King, 2004b] arvioi artikkelissaan, että yksinkertaisemmat JDBC-kehukset (kuten iBatis) ovat etenemässä kohti tilaa, jossa niistä on käytännössä tullut ORM-kehiksiä ilman mahdollisuutta generoida SQL:ää. Hibernaten versioon 3 on puolestaan lisätty mahdollisuus olla käyttämättä generoituja SQL-lauseita, mikäli niin halutaan. Toisin sanoen Hibernaten kaikkia muita ominaisuuksia voidaan käyttää, mutta SQL voidaan tarvittaessa kokonaan, tai osittain kirjoittaa käsin. Tämä antaa mahdollisuuden käsin tehtävälle lauseiden optimoinnille, mikäli se nähdään tarpeelliseksi.

Eräs uusi 3.0-version ominaisuus on myös tuki äskettäin julkaistun Javan 1.5-version annotaatioille. Tämän ominaisuuden myötä ei tarvita enää erillistä XDoclet-kirjastoa, mikäli metadata halutaan sijoittaa samaan paikkaan ohjelmakoodin kanssa. Hibernaten tekijät näkevät tämän tulevaisuuden mallina. He uskovat, että XML-tiedostoja ei tulevaisuudessa enää tarvita kuvausten määrittämiseen.

## Viiteluettelo

- [Ambler, 2003] Mapping Objects to Relational Databases and Evolutionary Database Development. In: Scott Ambler, *Agile Database Techniques*. Wiley, 2003, Chapters 9 and 14.  
Chapters available as <http://www.agiledata.org/essays/mappingObjects.html> and <http://www.agiledata.org/essays/evolutionaryDevelopment.html>.
- [Ambler, 2000] Scott Ambler, Mapping Objects to Relational Databases. What Do You Need to Know and Why. IBM Inc., July 1, 2000. Available as <http://www-106.ibm.com/developerworks/webservices/library/ws-mapping-to-rdb/>.
- [Apache, 2004] The Apache Software Foundation home page. The Apache Software Foundation, 2004. <http://www.apache.org/>.
- [Ashmore, 2004] Derek C. Ashmore, Best practices for JDBC programming, *Java developer's journal*. Volume 9, 10 (Oct. 2004), 18-22.
- [Barry, 2004a] Douglas Barry, Object-Relational Mapping Articles. Barry and Associates Inc., 2004. Available as <http://www.service-architecture.com/object-relational-mapping/articles/index.html>.
- [Barry, 2004b] Douglas Barry, Object Database Articles. Barry and Associates Inc., 2004. Available as <http://www.service-architecture.com/object-oriented-databases/articles/index.html>.
- [Bauer ja King, 2004] Christian Bauer, Gavin King, *Hibernate in Action*. Manning Publications, 2004.
- [Blaha *et al.*, 1988] Michael R. Blaha, William J. Premerlani ja James E. Rumbaugh, Relational database desing using an object-oriented methodology. *Comm. ACM* 31, 4 (Apr. 1988) 414-427.
- [Braeutigam *et al.*, 2001] Braeutigam Falko, Mueller Gerd, Nyfelt Per, Mekenkamp Leo, Ozone Users Guide. SMB GmbH, 2001. Available as [http://sourceforge.net/docman/display\\_doc.php?docid=10396&group\\_id=39695](http://sourceforge.net/docman/display_doc.php?docid=10396&group_id=39695).
- [Brown ja Whitenack, 1996] John Brown, Bruce Whitenack, Pattern Language for Relational Databases and Smalltalk. Draft, 1995. Available as <http://members.aol.com/kgb1001001/Articles/ObjectRelational/ObjectRelational.htm>.
- [Codd, 1970] Ted Codd, A Relational Model of Data for Large Shared Data Banks. *Comm. ACM* 13, 6 (1970), 377-387.
- [Date, 1995] C.J. Date, *An Introduction to Database Systems*. 6:th edition, Addison-Wesley, 1995.
- [Duchesne ja Nyfelt, 2001] Yanick Duchesne, Per Nyfelt. Ozone developers' guide. SMB GmbH, 2001.

Available as

[http://sourceforge.net/docman/display\\_doc.php?docid=10743&group\\_id=39695](http://sourceforge.net/docman/display_doc.php?docid=10743&group_id=39695).

- [Elmasri ja Navathe, 2000] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2000.
- [Fahl ja Risch, 1997] Gustav Fahl, Tore Risch, Query Processing Over Object Views of Relational Data. *The VLDB Journal* **6** (1997), 261-281.
- [Fayad *et al.*, 1999] Mohamed E. Fayad, Douglas C. Schmidt, Ralph E. Johnson, *Building Application Frameworks*. Wiley, 1999.
- [Fong, 1997] Fong Joseph. Converting Relational to Object-Oriented Databases. *ACM SIGMOD Record*, **26**, 1 (1997), 53-58.
- [Fong, 1995] Fong, Joseph. Mapping Extended Entity Relationship to Object Modeling Technique. *ACM SIGMOD Record*, **24**, 3 (Sept. 1995), 18-22.
- [Fowler, 2003] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- [Fussell, 2000] Mark Fussell, Foundations of Object-Relational Mapping. ChiMu Corporation, 2000.  
Available as <http://www.chimu.com/publications/objectRelational/>.
- [Gamma *et al.*, 2004] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns*. Addison-Wesley, 1995.
- [Green, 2004] Roedy Green, Java Glossary. Canadian Mind Products, 2004.  
Available as <http://mindprod.com/jgloss/jgloss.html>.
- [Heudecker, 2003] Nick Heudecker, Introduction to Hibernate. System Mobile Inc., 2003.  
Available as  
<http://www.systemmobile.com/articles/IntroductionToHibernate.html>.
- [Hibernate, 2004a] Hibernate Framework home page. Hibernate, 2004.  
<http://www.hibernate.org>.
- [Hibernate, 2004b] Hibernate Reference Documentation. Hibernate, 2004.  
Available as  
[http://www.hibernate.org/hib\\_docs/reference/en/pdf/hibernate\\_reference.pdf](http://www.hibernate.org/hib_docs/reference/en/pdf/hibernate_reference.pdf).
- [iBatis, 2004] iBatis Framework home page. Clinton Begin, 2004.  
<http://ibatis.com/>.
- [Javadoc, 2003] API specification for the Java 2 Platform, Standard Edition, version 1.4.2. Sun Microsystems Inc., 2003. Available as  
<http://java.sun.com/j2se/1.4.2/docs/api/>.
- [Jakarta CC, 2004] Jakarta Commons Collections home page. The Apache Software Foundation, 2004. <http://jakarta.apache.org/commons/collections/>.
- [JDBC, 2004] Java Database Connectivity home page. Sun Microsystems Inc., 2004. <http://java.sun.com/products/jdbc/index.jsp>.

- [Johnson *et al.*, 2004] Rod Johnson, Juergen Hoeller, Alef Arendsen, Colin Sampaleanu, Darren Davison, Dmitriy Kopylenko, Thomas Risberg and Mark Pollack, Spring Java/J2EE Application Framework Reference Documentation. Spring, 2004. Available as <http://www.springframework.org/docs/spring-reference.pdf>.
- [Johnson, 2003] Rod Johnson, *J2EE Design and Development*. Wiley, 2003.
- [King, 2004a] Gavin King, Haastattelu uusista Hibernaten ominaisuuksista. Vedos, Marraskuu 11, 2004.  
Saataavilla  
<http://www.javafree.com.br/home/modules.php?name=Content&pa=showpage&pid=41>.
- [King, 2004b] Gavin King, Using Hibernate 3 as a JDBC framework. Draft, August 23, 2004. Available as <http://blog.hibernate.org/cgi-bin/blosxom.cgi/Gavin%20King/customsql.html>.
- [Koskimies, 2003] Kai Koskimies, kurssin Ohjelmistoarkkitehtuurit 2003 luentomoniste. Tampereen Teknillinen Yliopisto, 2003.
- [Maven, 2004] Maven home page. The Apache Software Foundation, 2004. <http://maven.apache.org/>.
- [McFarland *et al.*, 1999] Gregory McFarland, Andres Rudmik and David Lange. Object-Oriented Database Managements Systems Revisited. Modus Operandi Inc, 1999.  
Available as <http://www.dacs.dtic.mil/techs/oodbms2/oodbms2.pdf>.
- [Niemi, 2003] Tapio Niemi. Kurssin Tietokantaohjelmointi luentomateriaali. Tampereen Yliopisto, Tietojenkäsittelytieteiden laitos, 2003.
- [Obasanjo, 2001] Dare Obasanjo, An Exploration of Object Oriented Database Managements Systems. Obasanjo, 2001. Available as <http://www.25hoursaday.com/WhyArentYouUsingAnOODBMS.html>.
- [ODMG, 2001] Object Data Management Group, The Standard for Storing Objects. Object Data Management Group, 2001. Available as <http://www.odmg.org/>.
- [Ozone, 2004] Ozone home page. Ozone, 2004. <http://www.ozone-db.org/>.
- [Paterson ja Haddow, 2004] James Paterson and John Haddow, Approaches to object persistence in Java projects. In: *The 9<sup>th</sup> Annual Conference on Innovation and Technology in Computer Science Education*. Comm. ACM, **36**, 3 (2004), 256-256.
- [Prevayler, 2004] Prevayler home page. Prevayler, 2004. <http://www.prevayler.org>.
- [Seppänen, 2001] Kari Seppänen. Kurssin Ohjelmistojen uudelleenkäyttö luentomateriaali. Espoo-Vantaan teknillinen ammattikorkeakoulu, 2001. Saataavilla <http://users.evtek.fi/~ritvak/ZO00/tunti5.pdf>.

- [Sliwa, 1999] Carol Sliwa. Java Database Connectivity. Computerworld Inc., 1999.  
Available as  
<http://www.computerworld.com/softwaretopics/software/appdev/story/0,10801,43545,00.html>.
- [Spring, 2004] Spring Framework home page. Spring, 2004.  
<http://www.springframework.org/>.
- [Struts, 2004] Struts Framework home page. The Apache Software Foundation, 2004. <http://struts.apache.org/>.
- [Sullivan, 2003] Sean Sullivan, Advanced DAO Programming. Learn Techniques for Building Better DAOs. IBM, Inc. October 7, 2003.  
Available as <http://www-106.ibm.com/developerworks/java/library/j-dao/>.
- [Venners, 2002] Bill Venners, Test-Driven development. A Conversation With Martin Fowler. Draft, December 2, 2002. Available as  
<http://www.artima.com/intv/testdriven.html>.
- [Villela, 2002] Carlos Eduardo Villela, An Introduction to Object Prevalence. IBM Inc., August 1, 2002. Available as <http://www-106.ibm.com/developerworks/web/library/wa-objprev/index.html>.
- [Wang, 2004] Wang, Quan. Oracle SQLj roadmap. Oracle Inc., 2004. Available as [http://www.oracle.com/technology/tech/java/sqlj\\_jdbc/pdf/oracle\\_sqlj\\_roadmap.pdf](http://www.oracle.com/technology/tech/java/sqlj_jdbc/pdf/oracle_sqlj_roadmap.pdf).
- [Wikipedia, 2004a] Wikipedia, The Free Encyclopedia. SQL. Draft, December 8, 2004. Available as <http://en.wikipedia.org/wiki/SQL>.
- [Wikipedia, 2004b] Wikipedia, The Free Encyclopedia. Object-Relational Mapping. Draft, October 15, 2004. Available as  
[http://en.wikipedia.org/wiki/Object-relational\\_mapping](http://en.wikipedia.org/wiki/Object-relational_mapping).
- [XDoclet, 2004] XDoclet home page. Last published October 23, 2004.  
<http://xdoclet.sourceforge.net/xdoclet/index.html>.

Huom. Kaikki URL-osoitteet on tarkistettu 16.12.2004.

**Liite A: Termit ja lyhenteet**

ADO	ActiveX Data Object
DAO	Data Access Object
DTO	Data Transfer Object
EB	Entity Bean
EER	Enhanced Entity-Relationship
EJB	Enterprise Java Bean
HQL	Hibernate Query Language
J2EE	Java 2 Enterprise Edition
JDBC	Java Database Connectivity
JNDI	Java Naming and Directory Interface
ODBC	Open Database Connectivity
OODBMS	Object Oriented Database Management System
ORM	Object Relational Mapping
POJO	Plain Old Java Object
RDBMS	Relational Database Management System
RMI	Remote Method Invocation