TEKNIIKKA JA LIIKENNE

Tietotekniikka

Ohjelmistotekniikka

OPINNÄYTETYÖ

TIETOKANNAN TESTIDATAGENERAATTORIN TOTEUTUS GROOVYLLA

Työn tekijä: Ville-Matti Kaartinen
Työn ohjaaja: Simo Silander
Työn ohjaaja: Olli Vanhapiha

Työ hyväksytty: __. __. 2008

Simo Silander
lehtori

ALKULAUSE

Tämä opinnäytetyö tehtiin Ixonos Oy:n Helsingin yksikölle.

Kiitän erityisesti Ixonosin puolella toiminutta ohjaajaani Olli Vanhapihaa. Olli antoi esimerkillisesti tukea työn eri vaiheisiin.

Myös Simo Silander ansaitsee mitä lämpimimmät kiitokset. Kiitos Simo.

Helsingissä 16.12.2008

Ville-Matti Kaartinen

Metropolia

OPINNÄYTETYÖN TIIVISTELMÄ

Työn tekijä: Ville-Matti Kaartinen

Työn nimi: Tietokannan testidatageneraattorin toteutus Groovylla

Päivämäärä: 12.11.2008 **Sivumäärä:** 51 s. + 10 liitettä

Koulutusohjelma: Tietotekniikka

Ammatillinen suuntautuminen: Ohjelmistotekniikka

Työn ohjaaja: lehtori Simo Silander

Työn ohjaaja: Olli Vanhapiha

Tässä insinöörityössä suunniteltiin ja toteutettiin Ixonosille tietokannan testidatan generaattori. Ohjelma generoi testidatan, joka voidaan kirjoittaa tiedostoon ja sen avulla syöttää tietokantaan. Työ ohjelmoitiin Groovy-skriptauskielellä.

Ixonosin projektissa oli vastaavanlainen sovellus käytössä, mutta siinä oli muutamia ongelmia. Pääongelma oli, että ohjelmalla ei pystynyt generoimaan helposti testidataa, jossa taulut viittaavat toisiinsa.

Työn tavoitteena oli kehittää vanhan sovelluksen tilalle uusi sovellus, joka selviytyisi myös viittauksista eri taulujen välillä. Ohjelman piti olla myös helposti laajennettavissa ohjelmoimalla. Ixonos halusi myös lisää tietoa skriptauskielistä ja niiden hyödyistä. Tavoitteena oli saada kokemusta varsinkin Groovy-skriptauskielestä. Selvitettävänä oli, nopeuttaako tai helpottaako Groovy ohjelmointityötä.

Työ aloitettiin tutustumalla skriptauskieliin ja erikoistumalla Groovy-kieleen. Kun Groovy-kieleen oli tutustuttu, aloitettiin työn suunnittelu ja toteutus. Uusi ohjelma suunniteltiin niin, että tietokannasta muodostetaan malli keskusmuistiin. Malli sisältää tiedot tietokannan tauluista ja sarakkeista. Malli sisältää myös tiedot siitä, minkälaista tietoa mihinkin kohtaan halutaan generoida. Koska malli muodostetaan muistiin, on viittaukset mahdollista toteuttaa. Mallin muodostamisen jälkeen siihen generoidaan arvot ja siitä kirjoitetaan vastaavat INSERT-lauseet tiedostoon.

Tuloksena syntyi uusi testidatan generaattori, jonka avulla saadaan generoitua testidataa, jossa on viittauksia tietokannan eri taulujen välillä. Testidatan generointiin kehitettiin myös erilaisia generaattoreita, jotka liitetään malliin. Generaattorit ovat kukin erikoistuneet omanlaistensa arvojen generointiin. Ohjelma on myös helposti laajennettavissa.

Avainsanat: Groovy, Skriptauskielet, Tietokannan testidata, Testidatan generointi

Metropolia

ABSTRACT

Name: Ville-Matti Kaartinen

Title: Implementing of Database Test Data Generator with Groovy

Date: 12th November 2008 **Number of pages:** 51 pages + 10 attachments.

Department: Information Technology **Study Programme:** Software Engineering

Instructor: Olli Vanhapiha, Group Manager

Supervisor: Simo Silander, lecturer

The purpose of this final project was to design and implement a Database Test Data Generator program using Groovy programming language. This project was carried out for Ixonos Plc.

The purpose of this project was to implement a program which generates random test data for database purposes. The generated data can be written to a file and then inserted to a database. The implemented program is going to replace the old software which has been written for the same purpose. Problems aroused with the old program when developers wanted to generate data where different tables refer to each other.

Ixonos wanted to find out if there were some benefits of using scripting languages. Groovy is a dynamic object-oriented programming language which can be used as a scripting language for the Java Platform. The purpose was to find out if Groovy can make the software implementation easier and faster.

The project started with getting to know the scripting languages and especially Groovy. The design and implementation phases started when there was enough knowledge about Groovy.

The study was successful in implementing the new Test Data Generator. The new program handles the situations where the developer needs to refer from one table to another. There are different generators implemented also, each generating data in its own way. The new program can also be easily extended.

Keywords: Groovy, Scripting languages, Database test data, Test data generation

SISÄLLYS

1 JOHDANTO

Tämän insinöörityön tavoitteena on kehittää sovellus, joka tuottaa satunnaista testitietoa tietokantaa varten. Generoitu testidata kirjoitetaan tiedostoon, joka voidaan syöttää tietokantaan. Ixonosin projektissa on käytössä jo vastaava sovellus, mutta siinä on muutamia puutteita. Sovelluksella ei ole mahdollista luoda helposti testidataa, joka sisältäisi viitteitä eri tietokannan taulujen välille. Ohjelmointityö toteutetaan Groovy-komentosarjakielellä, joten tavoitteena on myös tutustua komentosarjakieliin ja erityisesti Groovyyn.

Korvattava sovellus toimii niin, että se generoi tietokannan testirivejä toisto-lauseesta saadun indeksinumeron perusteella. Näin ollen sovelluksella ei ole muuta tilatietoa kuin hetkellinen indeksiarvo. Tämän takia testitietoon on vaikea saada generoitua aitoja viittauksia eri taulujen välille. Uusien taulujen määrittely on myös työlästä, koska jokaista sarakkeen arvoa vasten on kehitettävä oma arvon generointimekanismi.

Tämän työn tavoitteena on kehittää vanhan sovelluksen tilalle uusi ja parempi sovellus, joka osaisi hoitaa myös viittaukset taulujen välillä. Kehitystyön mahdollisina hyötyinä ovat testidatan laadun parantuminen ja sovelluksen selviytyminen haastavammista tilanteista kuin ennen. Testidatan laadun parantumisen myötä tietokantaa käyttävä sovellus tulee paremmin testatuksi. Tavoitteena on myös se, että ohjelma olisi kehittäjäystävällinen, eli se olisi helposti laajennettavissa ja muokattavissa.

Ohjelman ohjelmointikieleksi valittiin Groovy, jota voidaan käyttää Javan komentosarjakielenä. Groovy valittiin sen takia, että yritys haluaisi lisää tietoa siitä, olisiko Groovysta hyötyä myös muussa kehitystyössä. Selvitettävänä on, nopeuttaako ja helpottaako Groovy ohjelmointia ja ongelmien ratkaisemista.

Tässä työssä käsitellään aluksi yleisesti komentosarjakieliä ja sen jälkeen tarkemmin Groovy-kieltä ja sen ominaisuuksia. Groovy-kielen esittelyn jälkeen työssä selostetaan uusittavan ohjelman ongelma ja ratkaisu siihen. Seuraavaksi työssä selostetaan, kuinka uusi ohjelma toteutettiin. Lopuksi esitellään, kuinka ohjelmaa käytetään, ja käydään läpi tulokset ja yhteenveto.

2 JOHDATUS KOMENTOSARJAKIELIIN

2.1 Yleistä komentosarjakielistä

Komentosarjakielellä tarkoitetaan useimmiten ohjelmointikieltä, jolla voidaan ohjelmoida jotain toista kieltä tai sovellusta [7]. Komentosarjakielen avulla voidaan automatisoida tehtäviä ilman varsinaisen ohjelmointikielen apua [8]. Komentosarjakielen tunnetumpi nimi on skriptauskieli. Normaalit ohjelmat erotetaan pääasiassa komentosarjakielillä kirjoitetuista ohjelmista lähinnä siitä, että ne ovat riippumattomia muista ohjelmista [7].

Komentosarjat tulkitaan usein lennossa lähdekoodista tai esikäännetään tavukoodiksi, joka tulkitaan. Normaalit ohjelmat käännetään taas sen koneen konekielelle, missä ohjelmaa on tarkoitus ajaa. [7]

2.2 Historia

Lähes kaikissa käyttöjärjestelmissä on tekstipohjainen käyttöliittymä, jolla voidaan kirjoittaa komentoja ja käynnistää ohjelmia [8]. Komentosarjakielet kehitettiin toistuvien tehtävien ohjelmointiin eli automatisointiin. Komentosarjakielien ominaisuuksia ovat

- suorituksen parametrisointi
- muuttujien käyttö
- ehdollinen suoritus
- toisto. [8]

Yleisesti tunnettuja komentosarjakieliä ovat Unix-ympäristön komentokielet (sh, bash, ksh, zsh, csh, …) ja MS-DOS ajoilta tutut BAT-päätteiset komentosarjatiedostot [10; 9].

Perinteisesti komentosarjakielet ovat erottuneet nopeista esim. C-kielellä kirjoitetuista ohjelmista yksinkertaisuuden, hitauden ja turvallisuuden osalta, kuten Bourne Shell- ja Awk-komentosarjakielet osoittavat [7]. Ohjelman kirjoittaminen C-kielellä on paljon vaikeampaa kuin komentosarjakielellä.

Nykyään komentosarjakielet erottuvat perinteisistä ohjelmointikielistä lähinnä siitä, että niissä ei ole staattista tyypitystä, eli tietotyyppejä ei tarkasteta ennen suoritusta [7]. Komentosarjakielillä kirjoitettuja ohjelmia ei tarvitse myöskään kääntää, vaan ne yleensä tulkitaan. Uusimmissa

komentosarjakielissä (esim. Perl, Python ja Ruby) käytetään myös sellaisia abstraktioita, joita ei ole mielekästä kääntää, kuten koodin dynaaminen evaluointi, joka tarkoittaa sitä, että koodia tulkitaan tai käsitellään ajonaikaisesti [11].

Komentosarjakieli-termi on yleistynyt tarkoittamaan kaikkia kieliä, joilla on helppo tehdä käyttöjärjestelmän komennon tyyppisiä toimintoja helposti ilman ohjelman kääntämistä. Komentosarjakielestä on tulossa synonyymi tulkattavalle ohjelmointikielelle. [8]

2.3 Dynaamiset ohjelmointikielet

Dynaamiset ohjelmointikielet suorittavat ajonaikaisesti monia sellaisia toimenpiteitä, joita muut ohjelmointikielet suorittavat käännösvaiheessa. Toimenpiteisiin kuuluu esim. uuden koodin lisääminen luokkiin, tai tyypityksen vaihto. [12]

Dynaamisia ohjelmointikieliä ovat mm. Forth, Groovy, Lisp, Perl, PHP, Ruby, Scala, Smalltalk ja Visual Basic [12]. Monet dynaamiset ohjelmointikielet ovat dynaamisesti tyypitettyjä (eli tietotyypit tarkistetaan ajonaikaisesti), mutta eivät kaikki.

Dynaamisissa kielissä on havaittavissa muutamia yhteisiä ominaisuuksia [12]. Yksikään niistä ei ole vaatimus kielen määrittämiseksi dynaamiseksi, mutta useimmissa näitä ominaisuuksia on erilaisilla muunnelmilla. Ominaisuuksia ovat:

- Objektin tyyppiä voidaan vaihtaa ajonaikaisesti eli ilman ohjelman käännöstä.
- Ohjelmakoodia voidaan lisätä tai muuttaa ajonaikaisesti eli ilman käännösvaihetta.
- Kielessä on funktionaalisista kielistä tuttuja ominaisuuksia kuten sulkeumat.
- Kielessä tuetaan reflektiota, eli ohjelma voi tarkastella ja muuttaa sen omaa rakennettaan ja käyttäytymistään. [14]

Tässä työssä käytettävässä Groovy-kielessä on tuki näille kaikille ominaisuuksille. Groovyn ominaisuuksista kerrotaan lisää seuraavassa kappaleessa.

3 GROOVY

3.1 Yleistä

Groovy on ketterä ja dynaaminen olio-ohjelmointikieli, joka on rakennettu Java-alustan päälle. Groovyssa on monia Pythonista, Rubystä ja Smalltalkista tuttuja piirteitä, tuoden ne Java-ohjelmoijan saataville. Groovya luonnehditaan usein komentosarjakieleksi, mutta se on monien mielestä kielen aliarviointia. Groovylla voidaan ohjelmoida helposti esimerkiksi web-sovelluksia, se voidaan kääntää tavukoodiksi ja sillä saa helposti lisämaustetta Ant-build-tiedostoihin. [1, s. 4.] Groovylla voi tietenkin kirjoittaa vaikka kokonaisia sovelluksia.

Groovy on ohjelmoitu Javalla, ja tietyt osat Groovylla itsellään. Groovy on siis tiukasti sidottu Java-alustaan. Voitaisiin ajatella, että Groovy on ikään kuin erityistyyppistä Java-kieltä. Kaikki Javan mahdollisuudet ovat käytettävissä kaikkine kirjastoineen. Groovya ollaan standardoimassa Java-alustan kylkeen skriptauskieleksi nimikkeellä JSR-241 [2].

Groovy tekee paljon kulissien takana saavuttaakseen sen ketteryyden ja dynaamisen luonteen. Vaikka kaiken voisi tehdä Javalla, se voi tuntua turhauttavalta sen jälkeen, kun tietää miten sen voisi tehdä helpommin Groovylla. [1, s. 4.]

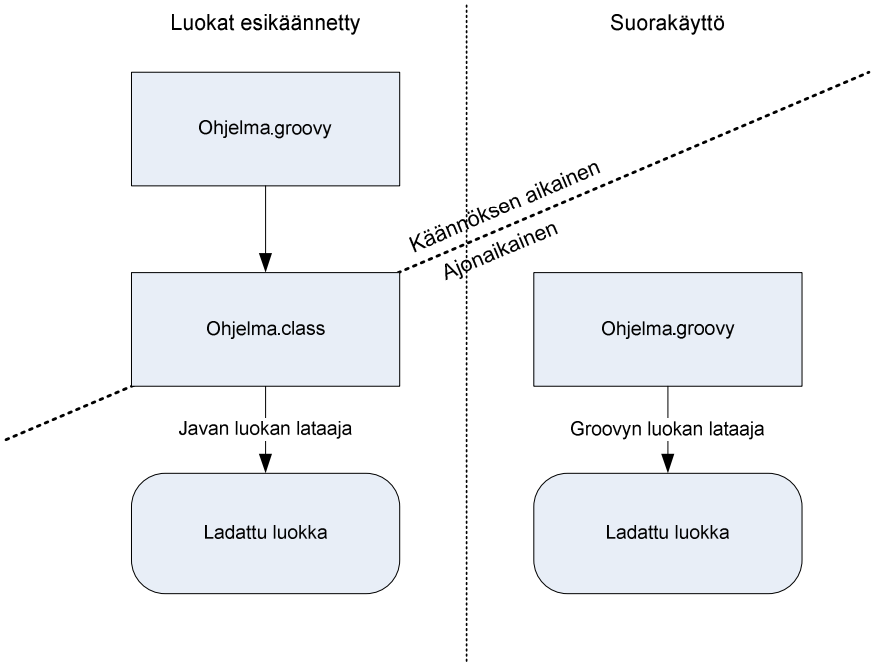
3.2 Groovy ohjelmointikielenä

3.2.1 Groovy on dynaaminen kieli

Groovya ajetaan Javan virtuaalikoneessa niin kuin normaaleja Java-luokkiakin ja se käyttää Javan kirjastoja. Se on ajonaikaisesta näkökulmasta katsottaessa periaatteessa normaalia Javaa, joka tarvitsee mukanaan ylimääräisen jar-paketin, jossa on riippuvaisuudet mukana.

Groovya voidaan ajaa Javan virtuaalikoneessa kahdella tavalla (kuva 1):

- .groovy–tiedostot voidaan kääntää `groovyc`–komennolla Java-luokkatiedostoiksi, jolloin ne voidaan liittää luokkapolkuun ja hakea Javan luokan lataajalla (*ClassLoader*).
- .groovy-tiedostoja käytetään suoraan, jolloin käytetään Groovyn omaa luokan lataajaa (*GroovyClassLoader*). [1, s. 48.]



Kuva 1. Groovy-luokkien ajaminen [1, s. 48.]

Toisin kuin muita komentosarjakieliä Groovya ei suoriteta rivi riviltä tulkiten, vaan koodi jäsennetään ja käännetään tavukoodiksi, joka lopuksi suoritetaan.

Groovy voi lukea *ajonaikaisesti* .groovy-tiedostoja ikään kuin ne olisivat luokkatiedostoja. Luokat voidaan myös generoida *etukäteen* Groovyn kääntäjällä, kuten edellä todettiin.

Oletetaan että käytössä olisi Groovy-komentosarja nimeltään OmaSkripti.groovy. Tämä komentosarja voitaisiin suorittaa komennolla: `groovy OmaSkripti.groovy`. Komentosarjan suorituksessa käydään läpi seuraavat vaiheet:

1. OmaSkripti.groovy syötetään jäsentimelle.
2. Jäsennin generoi komentosarjasta abstraktin syntaksipuun, joka edustaa täysin skriptin koodia.
3. Groovyn luokkageneraattori lukee syntaksipuun, ja muodostaa vastaavan Java-tavukoodin. Generaattori voi muodostaa useita luokkia komentosarjan sisällöstä riippuen. Tämän jälkeen luokat ovat käytettävissä Groovyn luokan lataajan avulla.
4. Komentosarja kutsutaan Javan virtuaalikoneen suoritettavaksi, samaan tapaan kuin suoritettaisiin komento: `java OmaSkripti`.

Kaikki tämä hoidetaan taustalla ja saa Groovyn kanssa työskentelyn tuntumaan siltä kuin olisi tekemisissä lennossa tulkitun kielen kanssa. Asia ei kuitenkaan ole näin, vaan luokat käännetään aina ennen ajamista, ja ne eivät muutu ajonaikaisesti. [1, s. 51.]

Groovyn kääntäjä muodostaa tavukoodia, joka on sisällöltään erilaista mitä Java-kääntäjä kääntäisi - ei muodon, vaan sisällön kannalta. Oletetaan, että Groovy-koodissa olisi metodi nimeltään ”foo”. Groovyn kääntäjä ei muodosta tavukoodia, joka kutsuu suoraan metodia, vaan se kutsuu foo-metodia MetaClass-luokan avulla [1, s. 52].

Koodilistaus 1. Metodin suoritus.

```
getMetaClass().invokeMethod(this, ”foo”, EMPTY_PARAMS_ARRAY)
```

Kaikki Groovy-luokat toteuttavat GroovyObject-rajapinnan, joka toimii MetaClass-luokan kanssa yhteistyössä. Groovyssa kaikista luokista voi kaivaa MetaClass-objektin esiin, joka sisältää metatietoa Groovy-luokasta, kuten että mitä metodeja ja kenttiä se sisältää. MetaClass-objektia säilytetään keskitetysti Groovyn MetaClassRegistry-luokassa [1, s. 217].

Kun metodikutsut ohjataan objektin MetaClass-luokan kautta, on mahdollista mm. uudelleenohjata, lisätä, poistaa ja tulkita metodeja ajonaikaisesti (koodilistaus 1). Groovyn metaohjelmointimahdollisuudet on toteutettu muun muuassa Javan reflektiota hyödyntäen.

3.2.2 Groovyssa kaikki ovat objekteja

Javassa on erotettu primitiivi- ja viitetyypit erikseen. Primitiivityyppeihin kuuluvat mm. int, double, char sekä byte ja viitetyyppeihin taas esim. Object ja String. Primitiivityypin muuttujan arvona voi olla esim. numero, kirjain tai totuusarvo. Primitiivityypeillä ei ole metodeja, joita voisi kutsua. Primitiivityyppejä ei voi myöskään käyttää, jos Java odottaa esimerkiksi jossain metodissa saavansa java.lang.Object -tyyppisen objektin. Ratkaisuna käytetään käärijäluokkaa, joka käärii primitiivityypin oliotyypiksi. Esimerkiksi int-tyypin käärijäluokka on java.lang.Integer.

Javassa käytetään erikseen olioita ja primitiivityyppejä sen takia, että suorituskyky olisi parempi ja myös C++-kielen perintö on vaikuttanut asiaan.

Groovyssa kaikki tyypit ovat luokkia. Näin Groovy helpottaa ohjelmoijan työtä ja samalla lisää tietokoneen kuormaa. Groovyssa ei siis käytetä primitiivityyppejä ollenkaan, vaan esimerkiksi numerot käsitellään pelkästään käärijäluokkien avulla.

Groovyssa on mahdollista kuitenkin määritellä muuttujia ikään kuin ne olisivat primitiivityyppejä, kuten 'int num = 4', mutta todellisuudessa, kulissien takana kuitenkin käytetään käärijätyyppiä [1, s. 57]. Tyyppien määrittely tehdään ohjelmoijalle helpoksi, mutta primitiivityyppien hylkääminen heikentää suorituskykyä.

3.2.3 Dynaaminen tyypitys

Staattisella tyypityksellä tarkoitetaan sitä, että tietotyypit tarkistetaan käännöshetkellä [3]. Java, C/C++, C#, Fortran, Pascal ja Haskell ovat esimerkkejä staattisesti tyypitetyistä kielistä. Koodilistaus 2 esittää, kuinka Java-kielessä määritetään muuttuja eksplisiittisesti kokonaislukumuuttujaksi.

Koodilistaus 2. Kokonaislukumuuttujan määritys Java-kielellä.

```
int numero = 4;
```

Dynaaminen tyypitys tarkoittaa taas sitä, että tietotyypit tarkistetaan ajonaikaisesti. Muun muuassa Smalltalk, Python, PHP, Ruby ja Groovy ovat dynaamisesti tyypitettyjä [4]. Koodilistaus 3 osoittaa, kuinka muuttuja voidaan määritellä dynaamiseksi Groovyssa.

Koodilistaus 3. Dynaaminen tietotyyppi.

```
def muuttuja = 4
muuttuja = ”Simo”
```

Ensin määritellään dynaaminen tietotyyppi ”muuttuja”, jolle annetaan arvoksi 4. Tällöin dynaaminen muuttuja pitää sisällään ajonaikaisesti java.lang.Integer-tyyppisen olion, jonka arvo on 4. Seuraavaksi samalle muuttujalle annetaan arvo ”Simo”, jolloin ajonaikainen muuttuja on java.lang.String-tyyppinen. Dynaaminen muuttuja on ikään kuin säiliö, jonka sisälle voidaan laittaa erityyppisiä muuttujia.

Staattisen tyypityksen avulla tyyppivirheet tunnistetaan käännösvaiheessa, jolloin kääntäjä voi generoida tehokasta koodia, koska tyyppejä ei tarvitse tarkistaa enää ajonaikaisesti [4]. Näin koodin pitäisi olla myös hieman luotettavampaa. Staattisen tyypityksen avulla kehitystyökalu antaa myös enemmän tietoa muuttujista ja metodien parametreista [5].

Dynaaminen tyypitys taas voi nopeuttaa kääntäjää tai skriptin tulkitsijaa, koska lähdekoodin muutoksen jälkeisiä tarkistuksia ei tarvitse tehdä niin paljon.

3.2.4 Groovyn Closure-objekti

Groovyn *Closure (*sulkeuma) on pienehkö listaus koodia, joka on kääritty objektiksi [5]. Sulkeumalla tarkoitetaan yleisesti funktiota, joka sitoo määrittely-ympäristönsä [13]. Sulkeuma käyttäytyy niin kuin normaali metodi. Sille voi siis määritellä parametrit, ja se voi palauttaa jonkun arvon. Koodilistaus 4 näyttää yksinkertaisen esimerkin Closure-objektista.

Koodilistaus 4. Closure-objekti luodaan ja sitä kutsutaan.

```
doSum = { a, b ->
   return a + b
}
println doSum(2, 7) // tulostaa 9
```

Esimerkissä doSum-muuttuja on siis sulkeuma, joka saa arvokseen kaarisulkeiden sisällön. Sulkeuman alussa määritellään parametrit a ja b. Objektin varsinainen sisältö kirjoitetaan ”->” merkkitokenin jälkeen. Sulkeumaa voi kutsua suoraan samalla tavalla kuin metodiakin mutta koska se on objekti, niin sen voi laittaa eteenpäin parametriksi, jollekin metodille, tai tallentaa vaikka listaan [1, s. 125].

Kappaleen alussa mainittiin että sulkeuma sitoo määrittely-ympäristönsä. Koodilistaus 5 selventää tilannetta. Jos ABind-luokan giveA-sulkeuma tallennetaan objektiksi, se palauttaa aina luvun 42, vaikka itse ABind-luokan ilmentymää ei varsinaisesti olisi käytettävissä. Täten sulkeuma muistaa määrittely-ympäristönsä ja säilyttää viittauksen ABind-luokan ilmentymään.

Koodilistaus 5. Sulkeuma sitoo määrittely-ympäristönsä.

```
public class ABind {
  private int a = 42
  def giveA = { return a }
}
```

Groovyn Closure-objekteja käytetään Collection-rajapintaa toteuttavissa luokissa. Koodilistaus 6 näyttää, kuinka List-tyyppinen kokoelma voidaan käydä läpi Groovyssa, ja tulostaa sen sisältämät elementit.

Koodilistaus 6. Listan elementtien läpikäynti ja tulostus.

```
def list = ['a', 'b', 'c']
list.each { e ->
   println e
}
```

Esimerkissä luodaan ensin ArrayList-tyyppinen lista, ja sen jälkeen kutsutaan sen each-sulkeumaa. Kaarisulkeissa e-niminen muuttuja saa listaa iteroitaessa elementin arvokseen, ja näin se voidaan tulostaa. Muuttuja voidaan jättää myös esittelemättä, jolloin voidaan kutsua suoraan it-nimistä muuttujaa (koodilistaus 7).

Koodilistaus 7. Listan elementtien läpikäynti ja tulostus käyttäen it-muuttujaa.

```
list.each { println it }
```

3.3 Groovyn erikoisuudet

3.3.1 Groovyn property

Groovyssa on mahdollista kutsua ja asettaa olion muuttujia web-ohjelmoinnista tutuilla tavoilla.

Koodilistaus 8. Book-luokan määritelmä.

```
class Book {
   String title
}
```

Koodilistaus 8 esittää Book-luokan toteutuksen. Luokalla on title-niminen *property*. Jos Java-luokassa on määritelty muuttujalle getName()- ja setName()-metodit, on kyseessä property. Groovyssa property määritetään jättämällä näkyvyysmääre pois. Book-luokan title-muuttuja on siis property. Groovy muodostaa property-tyypeille automaattisesti setterit ja getterit kulissien takana.

Koodilistaus 9. Book-luokan käyttäminen.

```
def book = new Book()
book.title = “Groovy on nastaa”
println book.title
```

Koodilistaus 9 näyttää, kuinka Book-luokan ilmentymän title-muuttujaa voidaan kutsua. Muuttujalle annetaan uusi arvo ja se tulostetaan.

3.3.2 Groovyn GString

Groovyssa merkkijonot merkitään joko heittomerkein tai lainausmerkein. Lainausmerkein merkitty merkkijono on GString-tyyppinen. GString-merkkijonoon on mahdollista merkitä muuttujia, joiden arvot haetaan kun merkkijonoa käytetään. Heittomerkein merkitty merkkijono on perinteinen String-merkkijono.

Koodilistaus 10. GString-merkkijonon käyttö.

```
def table = ‘EX_TABLE’
def insert = ”INSERT INTO $table”
println insert
```

Koodilistaus 10 näyttää, kuinka GString-merkkijonoa käytetään. Ensin määritellään normaali merkkijono, ja se lisätään GString-merkkijonon sisälle. Tuloksena on merkkijono: ’INSERT INTO EX_TABLE’.

3.4 Esimerkkitilanteita joissa Groovy helpottaa ohjelmointia

Groovyssa on paljon ohjelmoijan työtä helpottavia toimintoja. Erilaisten tietorakenteiden läpikäyntiin on käytettävissä paljon erilaisia tapoja. Myös XML-tiedostojen muodostaminen ja jäsentäminen on helppoa.

Moni asia hoituu yhdellä rivillä Groovyssa. Esimerkkitapauksena luetaan CSV-tiedosto, jossa on lueteltu maiden tietoja pilkuilla erotettuina tietoina ja

tulostetaan sieltä haluttu arvo. Koodilistaus 11 esittää, minkälainen on eräs CSV-tiedoston rivi.

Koodilistaus 11. CSV-tiedoston rivi.

```
AND,Andorra,Europe,Southern Europe
```

Koodilistaus 12 näyttää, miten CSV-tiedoston kaikki rivit luetaan, ja jokainen rivi paloitellaan String-luokan split()-metodin avulla.

Koodilistaus 12. CSV-tiedoston luku ja paloittelu split()-metodilla.

```
List countries = new File('country.csv').readLines().collect {
it.split(',')[1] }
```

Operaation jälkeen countries-lista sisältää kaikki CSV-tiedoston maat. Vastaava Java-ohjelma olisi pidempi, koska ensin täytyisi lukea tiedosto muistiin ja käydä se rivi riviltä läpi. Lopuksi täytyisi vielä muistaa sulkea tiedosto oikeaoppisesti. Groovy-ohjelmointikielessä on lisätty Javan File-luokkaan tässä käytetty readLines()-metodi, joka palauttaa tiedoston rivit listana ja huolehtii tiedoston lukemisesta sekä sulkemisesta. Tiedostosta saadut rivit annetaan collect-sulkeumalle, joka suorittaa listan jokaiselle alkiolle String-luokan split-operaation.

4 TIETOKANNAN TESTIDATAN GENERAATTORI

4.1 Johdantoa

Tämän insinöörityön konkreettinen työ on kehittää ohjelma, joka generoi testidataa, joka voidaan syöttää tietokantaan. Koska ohjelma on tarkoitettu lähinnä kehittäjien käyttöön, tarkoituksena ei ole tehdä sille käyttöliittymää. Ohjelmaa siis käytetään, konfiguroidaan ja laajennetaan ohjelmoimalla. Ixonosin projektissa on jo käytössä vastaava sovellus, mutta siinä on muutamia puutteita.

Tietokannassa on esimerkiksi yrityksiä (Company) ja asiakkaita (Customer). Jokaisella yrityksellä on useita asiakkaita, ja asiakkaat voivat viitata eteenpäin muihin tauluihin. Ixonosin projektissa tällä hetkellä käytössä olevalla ohjelmalla ei voi generoida testidataa, jossa viittaukset toimisivat

järkevästi. Tarkoituksena olisi kehittää sellainen tietokannan testidatan generaattori, joka hoitaisi myös viittaukset taulujen välillä mallikkaasti.

Tietokannan testidatan generaattori pitäisi olla myös helposti laajennettavissa ja käytettävissä muissakin projekteissa. Koska ohjelman käyttötarkoituksia laajennetaan ohjelmoimalla, tulisi ohjelmoinnissa ottaa huomioon myös laajentamisen helppous.

Korvattava generaattori on ohjelmoitu Java-kielellä, joten siinä mielessä Groovy sopii hyvin komentosarjakieleksi tähän projektiin. Groovyn oppimiskynnys Java-ohjelmoijille on pieni, koska siinä voidaan tarvittaessa käyttää Java-syntaksiakin.

4.2 Ongelma ja työn tavoite

4.2.1 Korvattavan ohjelman toimintaperiaate

Korvattava ohjelma on kirjoitettu Javalla. Ohjelmaa käytetään komentoriviltä antamalla sille parametrina, minkä taulun tietoja halutaan generoida ja montako riviä generoidaan. Taulut on ohjelmoitu erillisiin luokkiin. Luokissa on määritetty, minkälaista tietoa mihinkin sarakkeelle generoidaan. Ohjelma kirjoittaa halutut tietokannan INSERT-lauseet tiedostoon.

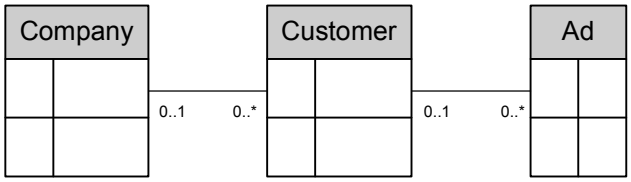
Generointiohjelmassa on toistolause, jota toistetaan niin monta kertaa kuin rivejä halutaan generoida. Toistolause kutsuu halutun taulun toteutusta, antamalla sille sen hetkisen indeksiarvon. Taulutoteutus generoi halutulle riville satunnaista testitietoa. Tietokannan rivejä kirjoitetaan tiedostoon sitä mukaa, kun lauseita generoidaan toistolauseessa.

Ongelmaksi ovat muodostuneet viittaukset taulujen välillä. Koska testidatan generointiohjelmalla ei ole muuta tilatietoa kuin sen hetkinen indeksi, on viittausten toteuttaminen hankalaa. Uusien taulujen määrittely on myös työlästä, koska jokaiseen kenttään täytyy keksiä tai kehittää oma arvon generointimekanismi.

4.2.2 Tavoite

Työn tavoitteena on kehittää testidatan generaattori joka osaisi generoida myös viitteet taulujen välille. Taulujen sarakkeet täytyisi määritellä ensin, ja sieltä viittaukset toisiin tauluihin.

Esimerkkinä tarkastellaan tietokantamallia (kuva 2), jossa yrityksellä (Company) on useita asiakkaita (Customer). Asiakkailla on vastaavasti useita mainoksia (Ad).



Kuva 2. Company-Customer-Ad-tietokantamalli.

Jotta kuvan mukaiseen tietokantaan voitaisiin generoida testidataa, pitää tietokannan malli ensin saada määriteltyä jotenkin. Malli sisältää tiedot tauluista ja niiden sarakkeista, ja mitä arvoja tauluissa on.

Koodilistaus 13 esittää, miltä kyseisen mallin (kuva 2) luontilausekkeet voisivat näyttää. Listauksessa kullakin yrityksellä on yksi asiakas, ja kullakin asiakkaalla yksi mainos. Yrityksiä luodaan kaksi kappaletta. Tässä työssä pyritään toteuttamaan sellainen ohjelma, joka luo nämä tarvittavat tietokannan rivin luontilauseet (INSERT-lauseet).

Koodilistaus 13. Esimerkki INSERT-lauseita Company-Customer-Ad -mallista.

```
INSERT INTO COMPANY(COMPANY_ID, COMPANY_NAME)
  VALUES(1, 'Ixonos');

INSERT INTO COMPANY(COMPANY_ID, COMPANY_NAME)
  VALUES(2, 'Mermit');

INSERT INTO CUSTOMER(COMPANY_ID, ID, AGE, COUNTRY, NAME)
  VALUES(1, 1, 22, 'Germany', 'Peter Gnek');

INSERT INTO CUSTOMER(COMPANY_ID, ID, AGE, COUNTRY, NAME)
  VALUES(2, 2, 18, 'Iceland', 'Sylvar Åsterblom');

INSERT INTO AD(CONTENT, AD_ID, CUSTOMER_ID)
  VALUES('adcontent_1', 1, 1);

INSERT INTO AD(CONTENT, AD_ID, CUSTOMER_ID)
  VALUES('adcontent_2', 2, 2);

COMMIT;
```

Suunniteltavan ohjelman täytyisi siis selvitä viittauksista eri taulujen välillä, ja sen täytyisi osata myös generoida satunnaista testidataa, kuten merkkijonoja ja numeroita.

4.2.3 Ratkaisun suunnittelu

Suunniteltavaa testidatan generaattoria käytettäisiin seuraavalla tavalla:

1. Tarkastellaan olemassa olevaa tietokantaa, ja sen sisältämiä tauluja, mihin tietoja halutaan generoida. Selvitetään sarakkeiden nimet ja tietotyypit. Selvitetään myös viittaukset eri taulujen välillä.
2. Ohjelmoidaan ohjelmaan vastaavat taulut, ja asetetaan ne generoimaan sopivia arvoja kullekin sarakkeelle.
3. Määritellään viittaukset eri taulujen välille.
4. Käynnistetään testidatan generaattori, jolloin se luo tiedoston joka sisältää INSERT-lauseita (kuten koodilistaus 13).

Ohjelma täytyy käytännössä suunnitella niin, että tietokannasta luodaan malli keskusmuistiin. Mallin luomisella muistiin yritetään ratkaista viittausten toteutusongelma. Jos mallia ei luotaisi muistiin, olisi vaikea viitata mihinkään taulujen välillä, mitä ei ole olemassa. Koska objektit luodaan ensin muistiin, on niistä helpompi viitata toisten objektien sisältämiin arvoihin.

Malli sisältäisi tietokannan taulujen tiedot, kuten että mitä sarakkeita taulussa on, ja mitä rivit sisältävät. Jokaista taulua pitäisi vastata yksi objekti muistissa, ja siihen voi liittyä useita rivejä.

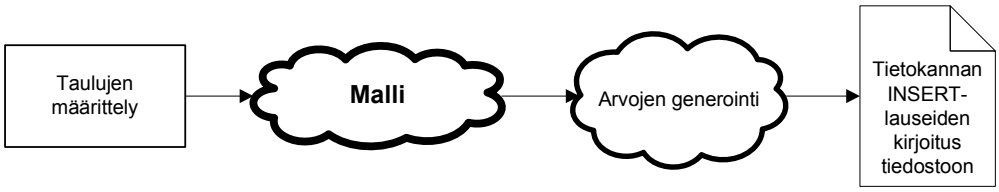
Kun tietokannasta luodaan malli muistiin, voidaan määritellä taulujen väliset yhteydet. Arvot voidaan generoida myöhemmässä vaiheessa, ennen kuin tietokantaa varten luotavat INSERT-lauseet muodostetaan. Tietokannasta rakennetaan ikään kuin selkäranka, joka kuvastaa, mistä komponenteista se koostuu.

Tarkoituksena on, että malli ei sisällä sen muodostamisen jälkeen vielä varsinaisia arvoja. Malli sisältää vain ohjeet siitä, minkälaisia arvoja mihinkin kohtaan halutaan generoida. Mallissa on myös määritelty, mitkä kentät viittaavat toisiinsa eri taulujen välillä, jolloin viittausongelma saadaan ratkaistua.

4.3 Suunnitelma

Vanhan ohjelman tilalle täytyi suunnitella ja kehittää uusi ohjelma. Lähtökohtana uuden ohjelman suunnitteluun oli se, että koko tietokannan halutusta aineistosta piti pystyä luomaan malli keskusmuistiin. Myös malli täytyisi saada jollain sarjallistettua lopullisiksi tietokannan luontilauseiksi.

Sarjallistamisella tarkoitetaan tässä työssä sitä, että mallin sisältämät tiedot täytyisi saada tulostettua konkreettisiksi tietokannan luontilauseiksi. Malli on olio, joka sisältää muita olioita. Strategia on se, että ensin tietokannasta luodaan sitä kuvaava malli muistiin, ja sen jälkeen mallista kirjoitetaan sellaiset INSERT-lauseet, jotka vastaavat mallia.



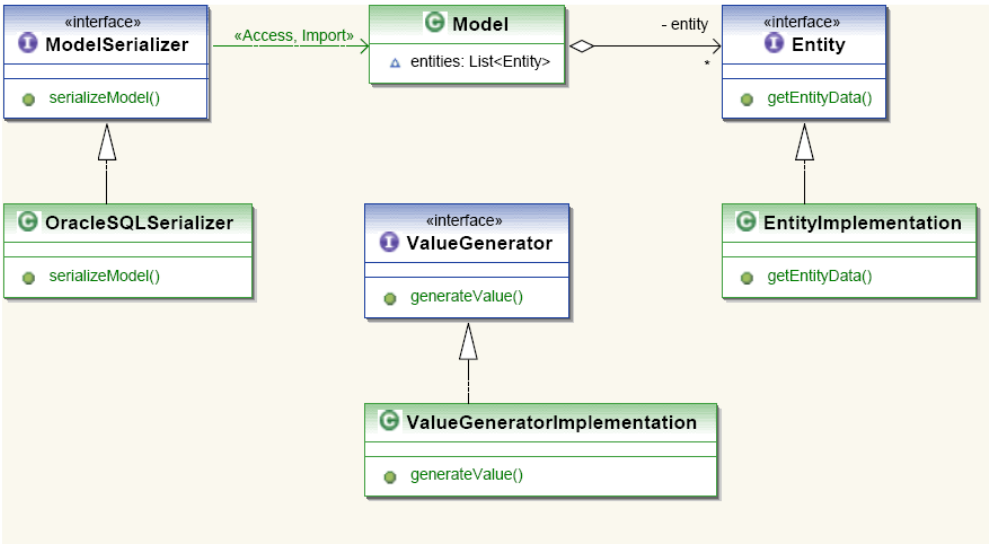
Kuva 3. Mallista luodaan INSERT-lauseet.

Ohjelma muodostaa taulujen määrittelyjen perusteella mallin muistiin (kuva 3). Malli sisältää tiedot siitä, mitkä kentät viittaavat toisiinsa ja mihin generoidaan mitäkin tietoa. Seuraavaksi tiettyihin mallin objekteihin generoidaan varsinaiset arvot, jonka jälkeen mallin perusteella voidaan kirjoittaa INSERT-lauseet.

4.3.1 Suunnittelun lähtökohdat

Aluksi suunniteltiin luokkakaavio yleisellä tasolla (kuva 4). Ohjelman keskeinen osa on Model-luokka, joka edustaa tietokannan mallia. Malli sisältää Entity-objekteja, jotka edustavat loppujen lopuksi taulujen yksittäisiä rivejä. Esimerkiksi, jos mallissa olisi yritys (Company) ja asiakas (Customer), niin yksittäiset yritykset ja asiakkaat olisivat Entity-objekteja. Nämä sarjallistuvat lopuksi yksittäisiksi taulun riveiksi.

Malli pitäisi myös sarjallistaa varsinaisiksi luontilauseiksi, joten sitä varten suunniteltiin ModelSerializer-rajapinta, jossa on serializeModel-metodi. Rajapinnan toteuttajan tehtävänä on kirjoittaa varsinaiset INSERT-lauseet lukemalla mallista tarvittavat tiedot.



Kuva 4. Yleisen tason luokkakaavio.

Luvussa 4.2.3 todettiin, että malli sisältää vain ohjeet siitä, minkälaisia arvoja mihinkin kohtaan halutaan generoida. Varsinaisia arvoja generoivat generaattoriluokat, jotka toteuttavat ValueGenerator-rajapinnan. Generaattorit liitetään tiettyihin komponentteihin mallissa, jolloin ne palauttavat satunnaisia arvoja niitä kutsuttaessa.

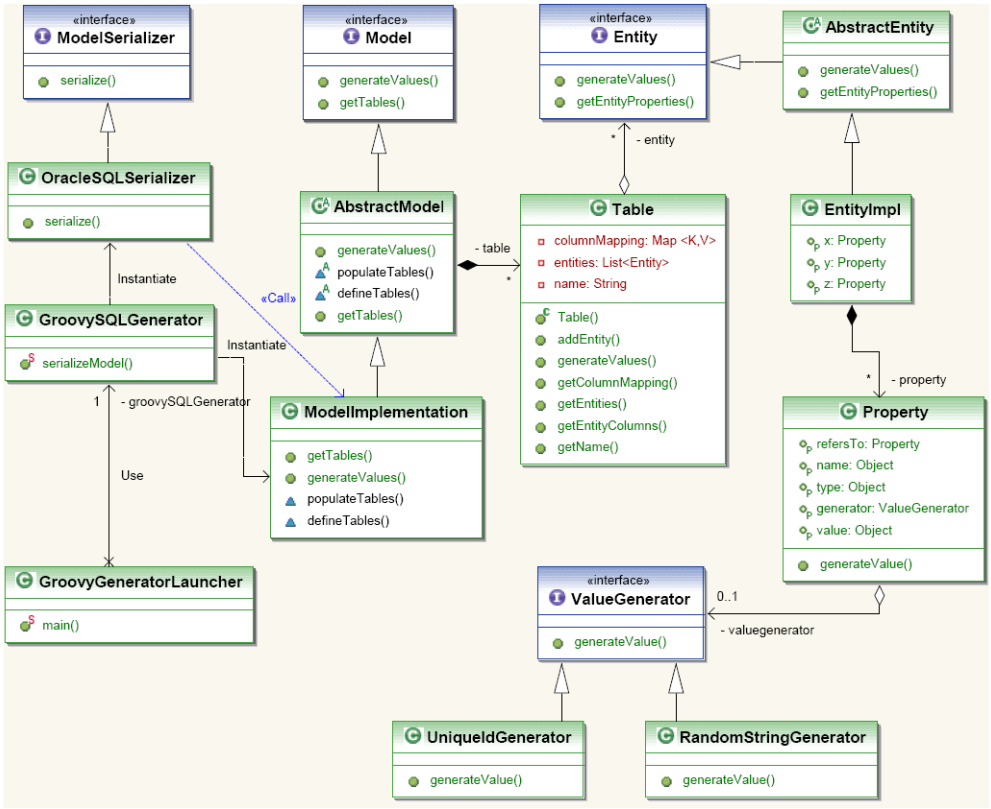
Jokainen generaattoriluokka on erikoistunut generoimaan tietynlaisia arvoja. Joku generaattori voisi generoida esimerkiksi satunnaisia merkkijonoja, ja joku toinen taas numeroita. Rajapinnan ansiosta kaikki generaattorit toimivat samalla tavalla palauttamalla arvon sen generateValue()-metodia kutsuttaessa.

Testidatageneraattoria suunniteltaessa ja ohjelmoitaessa käytettiin hyödyksi yleisen tason luokkakaaviota, josta ohjelman toimintaa lähdettiin tarkentamaan. Ohjelmoijien työn helpotukseksi mietittiin myös ratkaisua Entity-luokan toteutukselle, jotta Entity-rajapintaa toteuttava luokka olisi mahdollisimman helppo määritellä.

Suunnitelman tässä vaiheessa ei suunniteltu vielä viittausmekanismia, vaan se jätettiin suunniteltavaksi ja toteutettavaksi myöhemmin. Vaikka viittausmekanismi onkin tärkeä osa työtä, niin ajateltiin, että sen voisi suunnitella sitten, kun ohjelman päälinjaukset ovat selkeämmät.

4.4 Tekninen toteutus

4.4.1 Luokkakaavio



Kuva 5. Luokkakaavio.

Kuva 5 esittää testidatageneraattorin luokkakaaviota. Kappaleessa 4.3.1 esiteltiin yleisen tason luokkakaavio, joka on nyt tarkennettu astetta tarkemmaksi luokkakaavioksi. Tietokannan malli (ModelImplementation) on määritetty nyt abstraktin luokan, ja siihen liittyvien Table-komponenttien avulla. Table-luokka kuvastaa yksittäistä taulua tietokannassa (taulukko 1). AbstractModel-luokassa on valmista toiminnallisuutta, joka helpottaa mallin (ModelImplementation) luomista.

Taulukko 1. Mallia koskevat käsitteet.

Ominaisuus tietokannassa	**Vastaavuus mallissa**
Tietokannan taulu	Table-objekti
Taulun nimi	Table-objektin name-arvo

Taulun rivi	Entity-objekti
Taulun sarakkeen nimi	Property-objektin name-arvo
Taulun sarakkeen arvo tietyllä rivillä	Property-objektin value-arvo

ModelImplementation koostuu pääosin Table-objekteista. Table-objektit koostuvat pääosin Entity-objekteista. Entity-objekti voi edustaa esim. yksittäistä Company- tai Customer-taulun riviä (taulukko 1). Entity-rajapinta määrittelee sen, miten Entity-objektin tietoihin päästään käsiksi. Entity-rajapintaa toteuttavaa luokkaa ohjelmoitaessa peritään AbstractEntity-luokka (joka toteuttaa Entity-rajapinnan) ja määritellään Entity-toteuttajalle arvot. AbstractEntity-luokassa on valmista toiminnallisuutta, joka helpottaa Entity-rajapinnan toteuttajan ohjelmoimista.

Entity-objektin arvot ovat Property-luokan tyyppisiä. Entity-objekti vastaa taulun riviä (taulukko 1), jolloin Property-objektit ovat ikään kuin rivin sarakkeiden tietoja. Property-objektissa on määritetty nimi, tyyppi, generaattori, arvo ja viittausarvo (joka voi viitata toiseen Property-objektiin) sekä muuttujat. Property-objektin arvo eli value-muuttuja sisältää Entity-objektin tietyn sarakkeen arvon lopuksi, kun arvoja generoidaan. Property-objektiin liitetään generaattori, joka generoi arvoja.

Entity-luokka on tarkentunut yleisen tason luokkakaaviosta sisältämään Property-objekteja. Myös Model-luokka sisältää Table-objekteja, jotka määrittelevät tietokannan taulut.

ModelSerializer-rajapinnan avulla mallit voidaan sarjallistaa kannan luontilauseiksi yhtenäisellä tavalla. Rajapinnassa on määritelty serialize-metodi, joka ottaa parametrina Model-objektin.

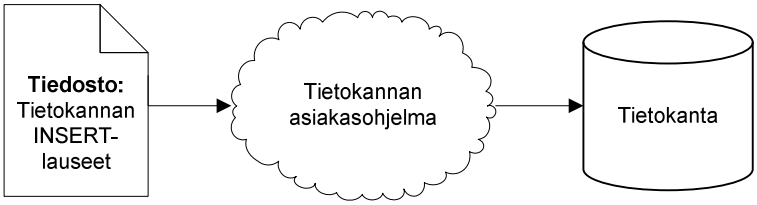
Ohjelma voidaan käynnistää esim. komentoriviltä pääohjelman sisältävän GroovyGeneratorLauncher-luokan avulla. Pääohjelma on kirjoitettu Javalla, jotta ohjelmaa voisi ajaa ilman Groovyn asennusta tietokoneelle.

4.4.2 Ohjelman käyttäminen sovelluskehittäjän näkökulmasta

Tällä ohjelmalla ei ole käyttöliittymää, joten sitä käytetään ohjelmoimalla. Jotta sovelluskehittäjä saa haluamansa testiasetelman rakennettua, tulee hänen suorittaa seuraavat vaiheet:

1. Sovelluskehittäjä tarkastelee niitä tietokannan tauluja, mihin hän haluaa testidataa generoida.
2. Sovelluskehittäjä tarkastaa, mitä ja minkä nimisiä sarakkeita taulussa on ja minkälaisia arvoja niihin on syytä syöttää.
3. Sovelluskehittäjä ohjelmoi tietokantaa vastaavan mallin perimällä AbstractModel-luokan, ja määrittelemällä siihen tietokannan taulut ja niiden halutut sisällöt.
4. Sovelluskehittäjä suorittaa pääohjelman antamalla sille sopivanlaisia parametreja.

Tuloksena ohjelma muodostaa tiedoston, joka sisältää tietokannan INSERT-lauseet. Tämä tiedosto voidaan syöttää tietokantaan tietokannan asiakasohjelman avulla, kuten kuva 6 havainnollistaa.



Kuva 6. INSERT-lauseiden syöttäminen tietokantaan.

Työn loppupuolella on esimerkki, jossa näytetään kuinka malli luodaan ohjelmoiden. Esimerkkimalli toteutetaan luvussa 4.9.

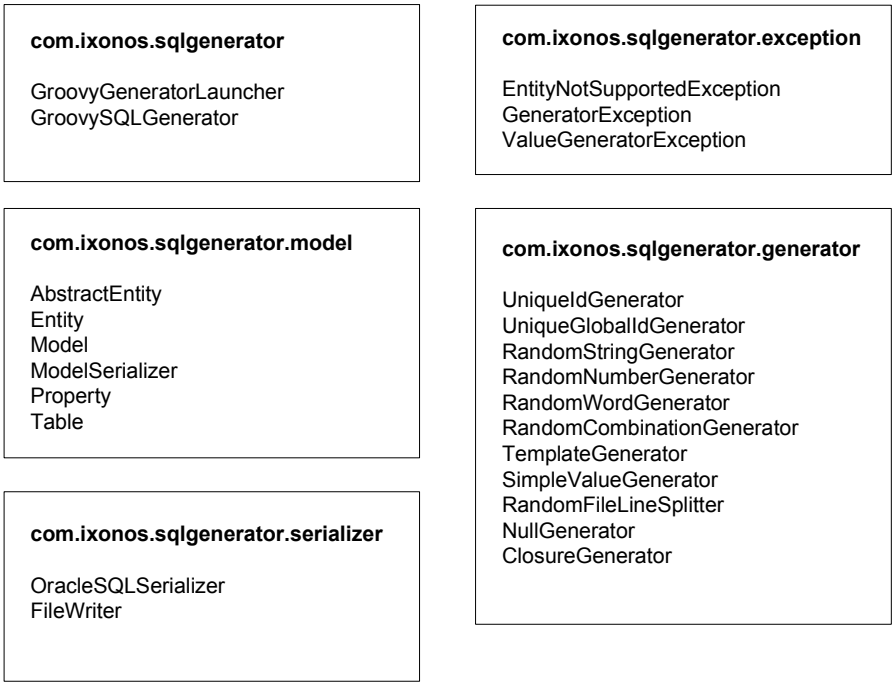
4.4.3 Projektin rakenne ja pakkauksiin ryhmittely

Testidatageneraattori on kirjoitettu hyvin pitkälle Java-tyyliin. Luokat jotka ovat toiminnallisuudeltaan samanlaisia tai muuten kuuluvat samaan kategoriaan, laitetaan usein samaan Java-pakkaukseen (package) [6]. Pakkauksiin ryhmittely antaa pakkauksen komponenteille yksilöllisen nimiavaruuden ja luokilla on pääsy toistensa protected-näkyvyysmääreellä määritettyihin jäseniin [6].

Generaattorin luokat on jaoteltu seuraaviin pakkauksiin:

- `com.ixonos.sqlgenerator`

- com.ixonos.sqlgenerator.exception
- com.ixonos.sqlgenerator.generator
- com.ixonos.sqlgenerator.model
- com.ixonos.sqlgenerator.serializer.



Kuva 7. Pakkaukset ja niiden sisällöt.

Kaikki pakkaukset ovat juuripakkauksen com.ixonos.sqlgenerator alla. Juuripakkauksessa on pääohjelma, jolla generaattorin saa käynnistettyä. Pakkauksessa com.ixonos.sqlgenerator.exception ovat kaikki projektissa käytettävät poikkeusluokat (kuva 7). Yksi selkeimmistä pakkauksista on com.ixonos.sqlgenerator.generator -pakkaus, jossa on määritelty kaikki ValueGenerator-rajapintaa toteuttavat testidatan generaattorit. Ohjelman selkäranka sijaitsee com.ixonos.sqlgenerator.model -pakkauksessa. Siellä on ohjelman keskeisimmät komponentit, kuten Model, Table, Entity ja Property. Pakkaus sisältää myös Entity- ja Model-luokkien abstraktit toteutukset.

Generaattorin lähdekoodi on sijoitettu projektihakemiston src-hakemistoon. Ohjelman koodia vasten on kirjoitettu työtä tehdessä yksikkötestejä JUnit-sovelluskehyksen mukaisesti. Yksikkötestit ovat test-hakemistossa, ja testattavat luokat ovat vastaavissa pakkauksissa kuin src-hakemistossa.

4.4.4 Työn toteutuksen kulku

Työn ohjelmoinnissa käytettiin avoimen lähdekoodin Eclipse-ohjelmointiympäristöä, jota käytetään yleisesti Java-ohjelmoinnissa. Käytössä oli Eclipsen standard-versio ja siihen oli asennettu Groovy-laajennus.

Eclipsen Groovy-laajennus mahdollistaa Groovy-koodin kirjoittamisen suoraan Eclipsessä niin, että koodin syntaksi korostetaan ja väritetään tuoden apua koodin lukemiseen. Eclipse osaa myös huomauttaa, jos koodissa on virheitä, mutta vain muutamissa tapauksissa. Eclipse ei huomaa Groovy-koodista virheitä niin hyvin kuin puhdasta Java-koodia kirjoitettaessa aiheuttaen välillä ongelmia. Muutenkin laajennuksessa oli ohjelmointivirheitä, ja se kaatoi useasti Eclipsen mukanaan.

Käytössä oli myös Eclipsen Subclipse-laajennus, jolla koodin sai tallennettua helposti Ixonosin SVN-versionhallintajärjestelmään. SVN:n käyttö helpotti suuresti työskentelyä vaihtelevien työskentelypaikkojen vuoksi. Versionhallintajärjestelmän ansiosta ideointi oli helppoa, kun tiedosti sen, että aina on mahdollisuus palata aiempaan versioon, mikäli tulisi ongelmia.

4.5 Ohjelmamoduulit

Seuraavaksi työtä tarkastellaan hieman tarkemmalla tasolla - käydään läpi ohjelmakoodia pakkauskohtaisesti ja tarkastellaan ohjelmakoodia tärkeimmistä luokista. Tarkastellaan ensimmäiseksi com.ixonos.sqlgenerator.model -pakkauksen sisältöä.

4.5.1 Entity.groovy

Entity on rajapinta, jonka avulla kaikista Entity-toteutuksista saadaan palautettua siihen liitetyt Property-objektit.

Koodilistaus 14. Entity-rajapinta.

```
interface Entity {

   public List getEntityProperties()

   public void generateValues()

}
```

Entity on siis objekti, joka edustaa esim. yksittäistä asiakasta Customer-taulussa, kuten todetaan luvussa 4.4.1. Kun halutaan generoida Entity-objektille arvot, kutsutaan sen generateValues()-metodia (koodilistaus 14). Entity-objektilta pyydetään sen sisältämät Property-objektit kutsumalla sen getEntityProperties()-metodia.

4.5.2 Property.groovy

Koodilistaus 15. Property-luokka.

```
class Property {

   def name
   def type
   def value
   ValueGenerator generator
   Property refersTo
}
```

Property-objekti on luokka, jonka avulla Entity määritellään. Yksittäinen Property on loppujen lopuksi tietokannan tietyllä rivillä olevan sarakkeen määritelmä. Esim. asiakkaalla (Customer) voisi olla Property nimeltään *name*, jolloin se sisältäisi asiakkaan nimen value-muuttujassaan lopuksi, kun arvot ovat generoitu. Entity-objektilla on niin monta Property-objektia, kuin tietokannan vastaavassa taulussa on sarakkeita.

Koodilistaus 15 on Property-luokan määritelmä ja siitä on nähtävissä luokan muuttujat. Property-objektin *name*-muuttuja on Propertyn nimi ja se tutkitaan ja asetetaan automaattisesti AbstractEntity-luokassa. Property-objektin *type*-muuttujan avulla määritetään Property-objektin tyyppi. Tyyppi on yleensä esim. *String*- tai *long*-tyyppinen. Tämän tiedon avulla voidaan lopuksi, tietokannan luontilauseita muodostettaessa, tehdä päätelmiä esim. siitä, että tulostetaanko kyseisen Property-objektin arvon ympärille heittomerkit vai ei. Arvo täytyy yleensä ympäröidä heittomerkeillä,§ jos arvo on merkkijono-tyyppinen.

Property-objektin *value*-muuttuja on aina Property-objekteja muodostettaessa tyhjä, eli se viittaa *null*-arvoon. Muuttuja saa arvon vasta lopuksi, kun arvoja generoidaan. Generoinnin suorittaa *generator*-muuttujan sisältämä ValueGenerator-rajapintaa toteuttava generaattori. Generaattoria ei tarvitse kuitenkaan määritellä, mikäli *refersTo*-muuttuja on asetettu. Viittausmekanismi on toteutettu Property-objektin sisältämän *refersTo*-

objektin avulla. Koska *refersTo* on Property-luokan tyyppinen muuttuja, se voidaan asettaa viittaamaan toiseen Property-objektiin. Viittausmekanismista kerrotaan tarkemmin myöhemmässä luvussa 4.6.2.

4.5.3 *AbstractEntity.groovy*

Kuten kappaleessa 4.4.1 todettiin, Entity-objekti kuvastaa tietokannan taulun yksittäistä riviä mallissa. Entity-rajapinnan toteuttaja ohjelmoidaan kun rakennetaan mallia. Tietokannan rivillä olevien sarakkeiden arvot määritellään Property-objekteihin. Täten Entity sisältää Property-objekteja. AbstractEntity-luokka on tarkoitettu helpottamaan Entity-rajapinnan toteutusta. Luokka osaa palauttaa aliluokassa määritellyt Property-objektit automaattisesti.

Koodilistaus 16. Entity-toteuttajan abstrakti kantaluokka.

```
abstract class AbstractEntity implements Entity {

   public void generateValues() {
     for (Property p in getEntityProperties()) {
       p.generateValue()
     }
   }

   public List getEntityProperties() {
     List allProperties = this.getMetaClass().properties
     List properties = allProperties.findAll {
       it.getAt('type').toString() == "class
                     com.ixonos.sqlgenerator.model.Property"
     }
     List results = []
     for (p in properties) {
       Property prop = p.getProperty(this)
       if (prop.name == null) prop.name = p.getAt('name')
       results.add(prop)
     }
     return results
   }
}
```

Uutta Entity-rajapinnan toteuttajaluokkaa ohjelmoitaessa peritään aina ensin AbstractEntity, joka on abstrakti luokka (koodilistaus 16). AbstractEntity-luokassa on valmis toteutus Entity-luokan Property-objektien palauttamiseksi joka on getEntityProperties(). Kyseisessä toteutuksessa on hyödynnetty Groovy-ohjelmointikielen metaohjelmointimahdollisuuksia.

Mikäli Entity-rajapinnan getEntityProperties()-metodi toteutettaisiin perinteisellä tavalla, olisi ohjelmoijan aina huolehdittava siitä, että Entity-

toteuttaja palauttaa aina kaikki Property-objektit List-tyyppisenä kokoelmana. Ohjelmoijan täytyisi lisätä luokkaan aina jokin tietorakenne, joka sitten lopuksi palautetaan niin kuin Entity-rajapinta sen määrittelee. Tämä toisi Entity-toteutukseen lisää koodia ja mahdollistaisi virheiden syntymisen helpommin.

Groovyssa kaikista luokista voi kaivaa MetaClass-objektin esiin, kuten kappaleessa 3.2.1 todettiin. AbstractEntityn getEntityProperties()-metodin toteutus etsii ja palauttaa automaattisesti toteuttajaluokan Property-tyyppiset objektit MetaClass-luokan avulla. Näin toteuttajaluokkaan tulee vähemmän koodia, ja se on paljon selkeämpi.

Kyseisessä metodissa tehdään myös toinen temppu hyödyntäen metaohjelmointia. Samalla, kun Property-objekteja haetaan luokasta ja tallennetaan listaan, metodissa asetetaan Property-objektin *name*-muuttujan arvoksi muuttujan nimi. Tätä voidaan hyödyntää sitten lopuksi sarakkeiden nimien muodostamisessa.

4.5.4 Model.groovy

Model-rajapinta yhtenäistää mallien sisällön ja arvojen generoinnin.

Koodilistaus 17. Model-rajapinta.

```
    interface Model {

       public List<Table> getTables()

       public void generateValues()

       public void setRepeat(int repeat)
}
```

Tätä Model-rajapintaa käyttää ModelSerializer-rajapinta, jonka avulla malli voidaan sarjallistaa esim. tietokannan luontilauseiksi.

Ohjelman malli suunniteltiin koostuvan tarvittaessa useista Table-komponenteista. Tämän takia itse Model-rajapinta jää yksinkertaiseksi. Koodilistaus 17 esittää rajapinnan, jossa on getTables()-metodi, joka palauttaa List-kokoelmana kaikki malliin liitetyt Table-objektit. Mallin taulujen sisältämät arvot voidaan generoida kutsumalla mallin generateValues()-metodia.

Mallin setRepeat()-metodi mahdollistaa parametrin liittämiseen malliin ohjelmaa kutsuttaessa komentoriviltä. Mallissa olevasta repeat-arvosta selostetaan lisää seuraavassa luvussa 4.5.5.

4.5.5 AbstractModel.groovy

Varsinaista Model-toteuttajaa ohjelmoitaessa olisi suotavaa periä aina AbstractModel-luokka. AbstractModel-luokassa on valmista toiminnallisuutta, joka helpottaa mallin luomista.

Koodilistaus 18. Model-toteuttajan abstrakti kantaluokka.

```
public abstract class AbstractModel implements Model {

   List<Table> tables = []

   int repeat

   abstract void defineTables()

   abstract void populateRow()
```

AbstractModel-luokassa on tables-niminen kenttä, joka on ArrayList-tyyppinen tietorakenne, johon luodut taulut lisätään automaattisesti. (koodilistaus 18). Luokassa on kaksi abstraktia metodia: defineTables() ja populateRow().

Mallin taulut on tarkoitus instantioida defineTables()-metodissa. Taulujen varsinainen sisältö määritellään populateRow()-metodin sisälle. Malli määritellään sillä tavalla, että populateRow()-metodi sisältää vain yhden luontikierroksen. Komentorivillä annettava *repeat*-niminen arvo määrittää sen, montako kertaa populateRow()-metodia kutsutaan.

Täten testiaineiston laajuutta voidaan kätevästi säädellä ohjelman ajovaiheessa. Yhdestä mallitoteutuksesta voidaan generoida niin suuri testiaineisto kuin halutaan. Mallissa voidaan määritellä esimerkiksi vain yksi yritys. Tällöin komentoriviltä voidaan pyytää suoraan niin monta yritystä kuin halutaan. Jos mallissa määritellään yritys, joka sisältää kaksi asiakasta, niin yhdellä mallin kutsukierroksella luodaan 3 riviä luontilauseita. Tällöin repeat-arvolla 100 ohjelma loisi 300 luontilausetta.

Koodilistaus 19. Table-objektien automaattinen haku.

```
private void prepareModel() {
  defineTables()

  repeat.times {
    populateRow()
  }

  List allProperties = this.getMetaClass().properties
  List properties = allProperties.findAll {
    it.getAt('type').toString() == "class
                    com.ixonos.sqlgenerator.model.Table"
  }
  for (t in properties) {
    Table table = t.getProperty(this)
    tables.add(table)
  }
  prepared = true
}
```

Koodilistaus 19 näyttää, kuinka Table-objektit haetaan toteuttavasta luokasta automaattisesti. Table-objektit tulee määritellä toteuttavaan luokkaan kentiksi, eli niillä täytyy olla julkiset get()-metodit, jotta prepareModel()-metodi osaa hakea kyseiset objektit.

Mallin valmistelussa (koodilistaus 19) toteutetaan Template Method -suunnittelumallia [15], sillä luokan abstrakteja metodeita kutsutaan tietyssä järjestyksessä. Ensin kutsutaan defineTables()-metodia, jolloin Table-objektit instantioidaan. Seuraavaksi mallin populateRow()-metodia kutsutaan repeat-arvon määräämän lukuarvon verran. Taulut on määritelty ja täytetty Entity-objekteilla kyseisen vaiheen jälkeen.

Mallin sisältämät taulut haetaan samanlaisella toimenpiteellä kuin luvussa 4.5.3 esitellyssä AbstractEntity-luokassa haetaan Property-objektit. Tällä kertaa MetaClass-luokan avulla haetaan Table-komponentteja.

Koodilistaus 20. Mallin Entity-objektien arvojen generointi.

```
public void generateValues() {
  if (!prepared) prepareModel()
  for (Table table in tables) {
    println "Generating values for ${table.getName()}"
    table.generateValues()
  }
}
```

AbstractModel-luokka toteuttaa Model-rajapinnan generateValues()-metodin käymällä kaikki siihen liitetyt Table-komponentit läpi ja kutsumalla niiden

generateValues()-metodia (koodilistaus 20). Ennen arvojen generointia kutsutaan tarvittaessa prepareModel()-metodia, joka valmistelee mallin. Näin arvot saadaan generoitua. Arvojen generointi selostetaan tarkemmin myöhemmässä luvussa 4.6.1.

4.5.6 Table.groovy

Malli koostuu Table-objekteista, ja täten Table on yksi tämän ohjelman tärkeimmistä luokista. Jokainen malliin liitetty Table-objekti kuvastaa yhtä tietokannan taulua ja sen sisältämää tietoa. Mallissa eri taulut voidaan laittaa viittaamaan toisiinsa.

Koodilistaus 21. Table-komponentin alustus.

```
public class Table {

  String name
  private List<Entity> entities = []
  private Map columnMapping = [:]

  public Table(String name, Map columnMapping) {
    this.name = name
    this.columnMapping = columnMapping
  }

  public Table(String name) {
    this.name = name
  }

  public Map getColumnMapping() {
      return columnMapping
  }
```

Table voidaan alustaa joko antamalla pelkkä taulun nimi konstruktorin parametrina tai määrittämällä nimi ja columnMapping-objekti, joka on Map-tyyppinen (koodilistaus 21). Objekti columnMap on HashMap-tyyppinen avain-arvo-taulukko, jonka avulla määritetään tietokannan sarakkeiden nimet. Kyseisen objektin avaimeksi kirjoitetaan Property-objektin nimi (joka siis haetaan automaattisesti niin kuin luvussa 4.5.3 todettiin) ja arvoksi haluttu sarakkeen nimi tietokannassa. Avain ja arvo ovat kumpikin merkkijonotyyppisiä.

Mikäli Table-objektin alustaa pelkällä nimellä, niin silloin columnMapping-objekti jää alustamatta. Se alustetaan kuitenkin heti, kun ensimmäinen Entity-objekti liitetään Table-objektiin.

Taulun sisältämä tieto on siihen liitetyissä Entity-objekteissa. Yksittäinen Entity-objekti vastaa yhtä taulun riviä, kun mallia sarjallistetaan. Table-luokan getEntities()-metodi palauttaa tauluun liitetyt Entity-objektit (koodilistaus 22). Taulun sisältämien Entity-objektien arvot voidaan generoida kutsumalla sen generateValues()-metodia.

Koodilistaus 22. Table-luokan generateValues()- ja getEntities()-metodit.

```
public void generateValues() {
  for (e in entities) {
    e.generateValues()
  }
}

public List<Entity> getEntities() {
  return this.entities
}
```

Tauluun voidaan lisätä Entity-objekteja vain addEntity()-metodin avulla (koodilistaus 23). Tällöin voidaan tarkistaa, että lisättävä Entity-objekti on oikean tyyppinen. Tarkistus tehdään niin, että tutkitaan Entity:n sisältämien Property-kenttien määrä ja verrataan sitä columnMapping-objektin kokoon. Mikäli kenttiä on eri määrä kuin odotettiin, heitetään poikkeus.

Koodilistaus 23. Table-luokan addEntity()-metodi.

```
public void addEntity(Entity entity) {
  if (columnMapping.size() == 0) {
    entity.getEntityProperties().each {
       columnMapping[it.name] = it.name
    }
  }
  if (entity.getEntityProperties().size() !=
                    columnMapping.size()) {
    throw new EntityNotSupportedException("Amount of
    properties ${entity.getEntityProperties().size()}doesn't
    match with amount of columns ${columnMapping.size()}.")
  }
  entities.add(entity)
}
```

Metodin alussa tehdään myös pieni toimenpide. Mikäli columnMapping-objektin sarakkeita ei ole vielä määritetty, ne määritetään nyt Entity-objektin sisältämien Property-objektien nimikenttien (name) avulla. Käytännössä tämä siis tapahtuu vain kerran eli silloin, kun ensimmäistä Entity-objektia liitetään tauluun. Tämän toimenpiteen avulla columnMapping-objektia ei ole

pakko määritellä (eli antaa sarakkeille nimiä), vaan voidaan käyttää suoraan niitä muuttujien nimiä, jotka on Entity-objektissa määritetty.

Koodilistaus 24. Table-luokan getEntityColumns()-metodi.

```
  public String getEntityColumns(Entity e) {
    return (e.getEntityProperties().collect {
      columnMapping[it.name] }).join(", ")
  }
```

Table-luokassa on myös apumetodi nimeltään getEntityColumns(), joka palauttaa annetun Entity-objektin Property-objekteja vastaavat arvot columnMapping-objektista, pilkulla erotettuna (koodilistaus 24). Näin mallin toteuttaja saa helposti oikean nimiset sarakkeet luontilausetta varten. Tässä on esimerkki, jossa Groovy auttaa toden teolla ohjelmoinnissa. Groovyn join()-metodi tekee tässä toimenpiteen ja palauttaa listan merkkijonona halutulla erotinmerkillä varustettuna.

4.5.7 ModelSerializer.groovy

Koodilistaus 25. ModelSerializer-rajapinta.

```
interface ModelSerializer {

   public def serialize(Model model)

}
```

ModelSerializer-rajapinnan tehtävänä on yhtenäistää mallin sarjallistaminen, jotta pääohjelmasta voidaan kutsua kaikkia sarjallistajia yhdenmukaisesti. Rajapinnan serialize()-metodi ottaa parametrinaan Model-rajapinnan tyyppisen mallin sekä palauttaa def-tyypin, eli se voi palauttaa mitä vain (koodilistaus 25).

Tarkoituksena on, että eri sarjallistajia voisi toteuttaa useita ja valita sitten pääohjelmaa käytettäessä että mitä sarjallistajaa käyttää. Sarjallistaja voi tulostaa luontilauseita suoraan näytölle, tai vaikka Junit-yksikkötestien käyttöä varten. Myös eri tietokannat tarvitsevat erilaisia muotoiluja kannan luontilauseille.

4.5.8 OracleSQLSerializer.groovy

Projektissa, jota varten tämä työ tehtiin, on käytössä Oracle-tietokanta, joten sitä varten toteutettiin sarjallistaja. OracleSQLSerializer toteuttaa

ModelSerializer-rajapinnan ja palauttaa kannan INSERT-lauseet merkkijonona. Merkkijono on helppo kirjoittaa myöhemmin tiedostoon Groovy-kielellä.

OracleSQLSerializer sijaitsee com.ixonos.sqlgenerator.serializer–pakkauksessa. Luokassa käytetään Groovyn mallipohjaominaisuuksia, jolloin voidaan luoda kaavain (template), johon syötetään tietoa muuttujien tilalle. Kaavain on määritetty luokan jäsenmuuttujaksi (koodilistaus 26).

Koodilistaus 26. OracleSQLSerializer-luokan toteutus.

```
class OracleSQLSerializer implements ModelSerializer {

def SQLTemplate =
      'INSERT INTO $tablename($columns) VALUES($values);'
  def COMMIT_CMD = 'COMMIT;'
  int commitInterval = 5000
  def engine
  def template

  OracleSQLSerializer() {
    engine = new groovy.text.SimpleTemplateEngine()
    template = engine.createTemplate(SQLTemplate)
  }
```

SQLTemplate-merkkijono on kaavain, jossa muuttujien edessä on dollarimerkki. Näin ollen `$tablename`, `$columns` ja `$values` ovat muuttujia, joiden tilalle saadaan syötettyä haluttua tekstiä. Pohja luodaan Groovyn SimpleTemplateEngine-luokan avulla konstruktorissa (koodilistaus 26).

Koodilistaus 27. OracleSQLSerializer-luokan serialize()-metodi.

```
public StringBuffer serialize(Model model) {
    model.generateValues()

    StringBuffer result = new StringBuffer()
    def counter = 1

    for (Table table in model.tables) {
      for (Entity entity in table.entities) {
...
```

Koodilistaus 27 esittää ModelSerializer-rajapinnan vaatiman serialize()-metodin toteutuksen. Metodissa kutsutaan Model-objektin generateValues()-metodia, jolloin mallin arvot generoidaan. Seuraavaksi mallin sisältämät taulut käydään läpi ja taulujen sisältämät Entity-objektit.

Koodilistaus 27 jatkuu.

```
...
        List values = entity.getEntityProperties().collect {
             it.type == String ? "'"+it.value+"'" : it.value }

        def binding = [
            "tablename" : table.name,
            "columns" : table.getEntityColumns(entity),
            "values" : values.join(', ') ]

        result.append(template.make(binding).toString()+ "\n")

        if (counter++ % commitInterval == 0) {
          result.append(COMMIT_CMD + "\n")
        }
      }
    }
    result.append(COMMIT_CMD)

    return result
  }
}
```

Entity-objektista kerätään arvot listaksi, jotta ne voitaisiin lisätä pohjaan. Arvo tarkistetaan, ja mikäli se todetaan merkkijono-tyyppiseksi, sen ympärille lisätään heittomerkit. Listan collect-sulkeuma on kätevä tällaisissa tapauksissa, joissa lista täytyy käydä läpi ja tehdä listan jokaiselle arvolle jokin toimenpide.

Seuraavaksi muodostetaan binding-objekti, joka on Map-tyyppinen (koodilistaus 27). Siinä avaimena toimii pohjan muuttuja ja arvona kyseisen muuttujan haluttu arvo. Tässä käytetään hyväksi Table-luokan getEntityColumns()-apumetodia, jolloin `$columns` -muuttujaan saadaan oikeat sarakkeiden arvot kätevästi pilkulla erotettuina. Arvolistaus saadaan aikaan, kun käytetään aiemmin määritettyä listaa, jonka join()-metodia kutsutaan.

Table-luokassa on apumetodi vain sarakkeiden nimien hakemiselle. Syy tähän on se, että arvojen listauslogiikka haluttiin jättää sarjallistajan tehtäväksi. Sarakkeiden nimiin ei yleensä tule mitään erikoista, kuten pilkkuja, toisin kuin arvoihin. Eri tietokannoissa voi olla eri tavalla esitettynä eri arvot.

OracleSQLSerializer-luokan alussa määritetty commitInterval-muuttuja ilmaisee sen, kuinka monen luontilauseen jälkeen väliin syötetään määritetty tietokannan commit-lause.

4.5.9 *Poikkeusluokat*

Pakkauksessa `com.ixonos.sqlgenerator.exception` on määritetty poikkeusluokat. Poikkeusluokkia on 3 kappaletta:

- `GeneratorException.groovy`
- `EntityNotSupportedException.groovy`
- `ValueGeneratorException.groovy`

GeneratorException-luokka on abstrakti, joten se on tarkoitettu muiden poikkeusten kantaluokaksi (koodilistaus 28). Näin ollen uusia poikkeuksia on helppo toteuttaa.

EntityNotSupportedException-luokka on tarkoitettu ilmaisemaan poikkeustilannetta, kun Table-objektiin yritetään lisätä sellaista Entity-objektia, jonka Property-objektien määrä ei vastaa taulun sarakkeiden määritystä. ValueGeneratorException-luokka on tarkoitettu tilanteisiin, joissa ValueGenerator-rajapintaa toteuttava generaattori kohtaa virhetilanteen.

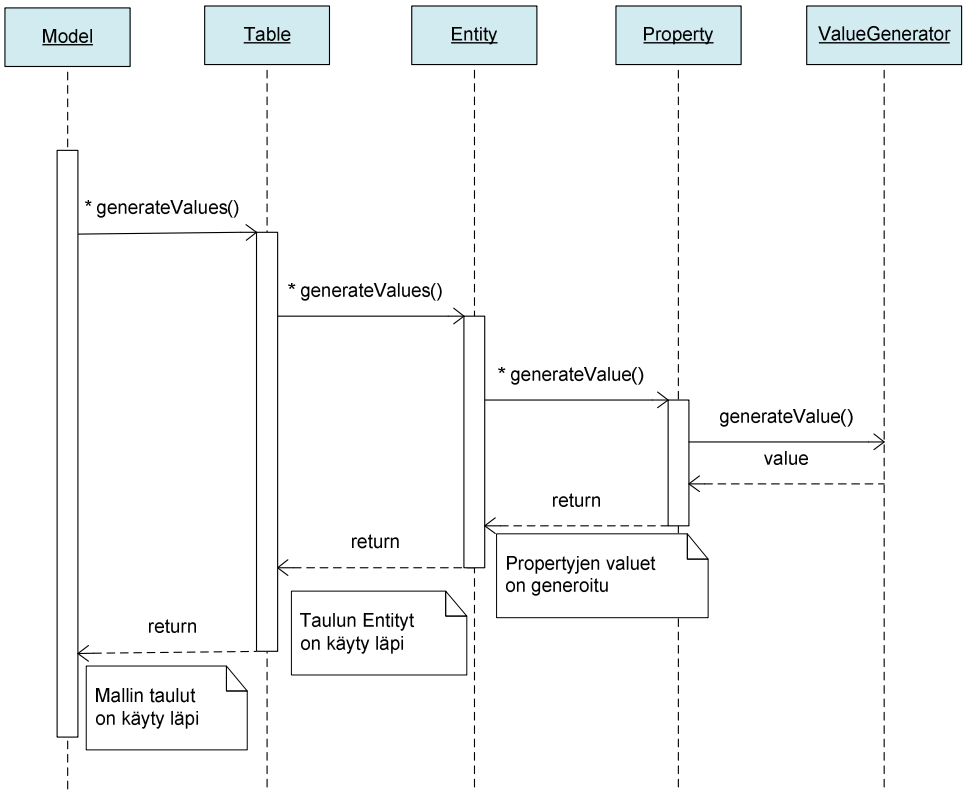
Koodilistaus 28. Poikkeusten kantaluokka.

```
public abstract class GeneratorException extends Exception {

   public GeneratorException(String message) {
     super("GeneratorException: " + message);
   }

   public GeneratorException(Throwable cause) {
     super(cause);
   }

   public GeneratorException(String message, Throwable cause){
     super(message, cause);
   }
}
```

4.6 Arvojen generointi ja viittaus

4.6.1 *Arvojen generointi*

Tietokannan testidatageneraattori on suunniteltu siten, että tietokannasta luodaan malli muistiin ja Property-objektit saavat arvonsa vasta sitten, kun mallin generateValues()-metodia kutsutaan. Tämä tapahtuu yleensä vasta ModelSerializer-rajapintaa toteuttavassa sarjallistajassa. Kuva 8 on sekvenssikaavio, joka havainnollistaa sitä, kuinka arvot generoidaan.



Kuva 8. Arvojen generoinnin sekvenssikaavio.

Mallin generateValues()-metodin kutsuvaiheessa käydään läpi siihen liitetyt Table-objektit, ja kutsutaan niiden generateValues()-metodia. Table-objekti käy läpi siihen liitetyt Entity-objektit ja kutsuu niiden generateValues()-metodia. Entity käy läpi taas siihen liitetyt Property-objektit, ja kutsuu niiden generateValue()-metodia.

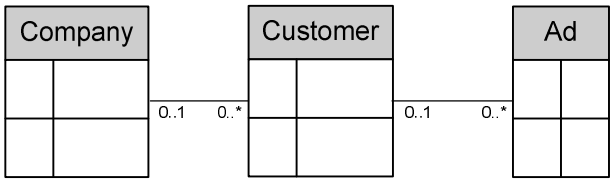
Property-luokan generateValue()-metodi on siis generoinnin varsinainen toteuttaja. Property-objektin value-muuttuja saa arvonsa generator-objektilta. Arvo generoidaan vain, jos arvoa ei ole vielä generoitu. Arvo on voitu jo generoida, mikäli kyseessä on viitearvo. Viittausmekanismista kerrotaan lisää seuraavassa luvussa 4.6.2.

4.6.2 Arvojen viittausmekanismi

Viittausmekanismi on uuden generaattorin hyödyllisimpiä ominaisuuksia. Mekanismin avulla taulun Entity-objekteja voidaan asettaa viittaamaan toisien taulujen Entity-objekteihin. Viittaukset asetetaan Model-rajapintaa

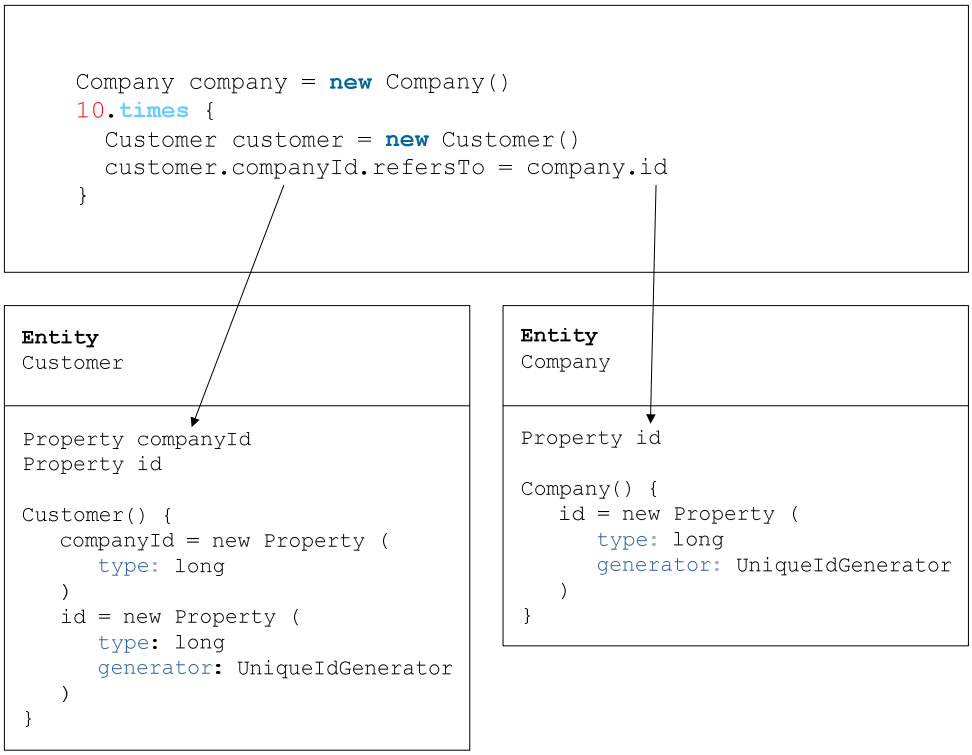
toteuttavassa luokassa, kun mallia rakennetaan. Viitattavan Entity-objektin arvo haetaan lopuksi, kun arvoja generoidaan.

Esimerkkinä tarkastellaan yksinkertaista, kuvitteellista tietokannan mallia (kuva 9), jossa yrityksellä (Company) on useita asiakkaita (Customer). Asiakkaihin on liitetty useita mainoksia (Ad).



Kuva 9. Esimerkki tietokantamalli.

Kuva 10 esittää, kuinka tietokantageneraattorin malli muodostetaan yrityksen ja asiakkaan suhteesta kooditasolla.



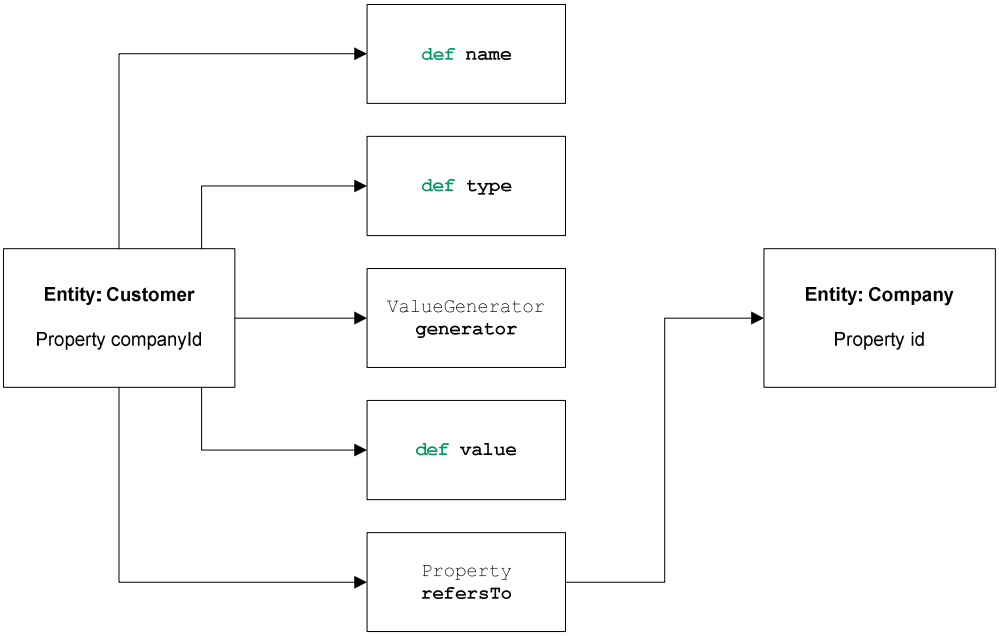
Kuva 10. Esimerkkinä Customer-Company-viittaus.

Kuva 10 on koodiesimerkki, jossa luodaan ensin Company-objekti ja sitä kohden tehdään 10 Customer-objektia. Jokainen Customer-objekti laitetaan viittaamaan alussa luotuun Company-objektiin. Koodissa käytetään Groovy-

ohjelmointikielen suoraa viittausta objektin kenttään kirjoittamalla kentän nimi. Menetelmä esiteltiin luvussa 3.3.1.

Viittaus toteutetaan tässä mallissa asettamalla Customer-objektin companyId-kentän Property-objektin refersTo-arvo viittaamaan Company-objektin *id*-nimiseen Property-objektiin. Tämä on mahdollista, koska jokaisella Property-objektilla on refersTo-kenttä, joka on Property-tyyppinen.

Kuva 11 esittää kyseistä tilannetta jossa Customer-objektin companyId-objektilla on muuttujia sisällään. Tällöin refersTo-objektin arvoksi voidaan asettaa toinen Property-objekti, joka on tässä tapauksessa Company-objektin id-Property.



Kuva 11. Property viittaa toiseen Property-objektiin.

Viittauksen asettamisen jälkeen viitattava objekti on vain refersTo-objektiin tallennettuna. Varsinainen viitattava arvo haetaan vasta sitten, kun mallin arvoja generoidaan. Generoinnin suorittaa lopulta Property-luokan generateValue()-metodi, jossa viitattava arvokin haetaan (koodilistaus 29).

Koodilistaus 29. Property-objektin generateValue()-metodi.

```
  public void generateValue() {
    if (value == null) {
      if (refersTo == null) {
        value = generator?.generateValue()
      } else {
        if (refersTo.value == null) {
          refersTo.value = refersTo.generator?.generateValue()
        }
        value = refersTo.value
      }
    }
  }
```

Koodilistaus 29 esittää generateValue()-metodin, jossa arvo generoidaan joko käyttäen objektiin liitettyä generaattoria tai refersTo-objektia. Mikäli refersTo-arvoa ei ole asetettu, niin silloin Property-objektin arvon generointiin käytetään sen generaattoria. Mikäli taas refersTo-objekti on asetettu, eli kyseessä on viittausarvo, niin silloin käydään tarkistamassa, onko viitattu arvo jo generoitu. Mikäli kyseistä arvoa ei ole vielä generoitu, se generoidaan käyttämällä viitattavan Property-objektin generaattoria. Tämän jälkeen Property-objektin value-kenttä saa saman arvon kuin viitattava Property-objekti.

4.7 Generaattorit

Pakkaus com.ixonos.sqlgenerator.generator on tarkoitettu kaikille ValueGenerator-rajapintaa toteuttaville generaattoreille. ValueGenerator-rajapinnassa (koodilistaus 30) on määritelty generateValue()-metodi, jonka tarkoitus on yhtenäistää kaikkien generaattoreiden kutsuntatapa.

Koodilistaus 30. ValueGenerator-rajapinta.

```
interface ValueGenerator {

   public def generateValue()

}
```

Generaattori asetetaan Property-objektin generator-kentän arvoksi, jolloin siltä pyydetään arvo lopuksi kun arvoja generoidaan.

Pakkauksessa ovat seuraavat generaattorit:

- UniqueIdGenerator
- UniqueGlobalIdGenerator

- RandomStringGenerator
- RandomNumberGenerator
- RandomWordGenerator
- RandomCombinationGenerator
- TemplateGenerator
- SimpleValueGenerator
- RandomFileLineSplitter
- NullGenerator
- ClosureGenerator.

Generaattorit ovat toteutettu suurimmaksi osaksi siten, että ne saavat konstruktorissa yhden tai useampia parametreja, joiden avulla generaattorit generoivat dataa. Parametrit on annettava generaattorin luontivaiheessa, koska rajapinnan määrittelemä generateValue()-metodi ei ota parametria vastaan. Generaattoreiden toteutukset ovat liitteinä 1-11.

4.7.1 UniqueIdGenerator ja UniqueGlobalIdGenerator

UniqueIdGenerator-luokka on tarkoitettu uniikkien kokonaislukujen generoimiseen. Uniikkeja lukuja voidaan käyttää tietokannan id-sarakkeissa. UniqueIdGenerator-luokka generoi uniikkeja arvoja instanssikohtaisesti, eli sille voidaan antaa alkuarvo, josta se lähtee kasvattamaan lukua. UniqueIdGenerator-luokan toteutus on nähtävissä liitteessä 1.

UniqueGlobalIdGenerator on taas Singleton-suunnittelumallilla toteutettu uniikkien arvojen generaattori. UniqueGlobalIdGenerator-luokasta on vain yksi ilmentymä Javan virtuaalikoneessa käynnissä, joten siltä saadaan uniikkeja arvoja mistä päin sovellusta tahansa. UniqueGlobalIdGenerator-luokan toteutus on nähtävissä liitteessä 2.

Useimmiten Property-objektit varustetaan kuitenkin instanssikohtaisella UniqueIdGenerator-objektilla. Tällöin UniqueIdGenerator-instanssi täytyy muistaa asettaa Property-luokan staattiseksi jäseneksi, jolloin generaattori antaa uniikkeja arvoja luokkakohtaisesti.

4.7.2 RandomStringGenerator ja RandomNumberGenerator

RandomStringGenerator-luokan (liite 3) avulla voidaan generoida merkkijonoja, jotka koostuvat satunnaisista merkeistä. Generaattoria voi

käyttää eri tavoin, riippuen kuinka se on alustettu. RandomStringGenerator voidaan alustaa vaihtoehtoisilla tavoilla:

- Annetaan vain merkkijonon pituus, jolloin satunnaisia merkkejä generoidaan pituuden osoittaman määrän.
- Määritellään merkkijonon pituus ja sallitut merkit, jolloin generoidaan merkkejä vain merkkijoukosta, joka on annettu.
- Määritellään merkkijonon pituus ja aloitus- sekä lopetusmerkki, jolloin ilmaistaan, miltä väliltä merkkejä valitaan satunnaisesti merkkijonoon.

Koodilistaus 31 esittää, kuinka RandomStringGenerator-luokka voidaan alustaa. Alustuksen jälkeen Generaattori palauttaa merkkijonoja, jotka koostuvat 50:stä a:n ja z:n välillä olevasta merkistä.

Koodilistaus 31. RandomStringGenerator-luokan alustus.

```
new RandomStringGenerator(50, 'a'..'z')
```

RandomNumberGenerator-luokan avulla voidaan generoida satunnaisia lukuja annetusta arvojoukosta. RandomNumberGenerator (liite 4) alustetaan antamalla sille Groovyn IntRange-tyyppinen objekti, joka edustaa lukujoukkoa. Vaihtoehtoisesti toisena parametrina voidaan syöttää merkkijonona malli, jonka avulla numero voidaan formatoida esittämään Javan DecimalFormat-luokan avulla numeroa.

Koodilistaus 32. RandomNumberGenerator-luokan alustus.

```
new RandomNumberGenerator(1..31, '00')
```

Koodilistaus 32 osoittaa, kuinka RandomNumberGenerator-luokka alustetaan generoimaan lukuja arvojoukosta {1, 2, 3, ... , 31} ja antamaan luvut formatoituna niin, että ne sisältävät 2 merkkiä.

4.7.3 RandomWordGenerator ja RandomCombinationGenerator

RandomWordGenerator-luokka (liite 5) generoi satunnaisia arvoja sille annetusta List-tyyppisestä kokoelmasta. Kokoelma on luokan jäsenmuuttuja, ja se määritellään konstruktorissa.

RandomCombinationGenerator-luokka (liite 6) generoi satunnaisen arvoparin kahdesta sille syötetystä listasta. Listojen lisäksi määritellään erotinmerkki, joka syötetään arvojen väliin. RandomCombinationGenerator-

luokka käyttää apunaan RandomWordGenerator-luokkaa muodostaessaan satunnaista arvoparia.

Koodilistaus 33. RandomCombinationGenerator.

```
def etu = ['Aarne', 'Maija', 'Simo']
def suku = ['Multisilta', 'Reinikainen', 'Laho']
def generator = new RandomCombinationGenerator(etu, suku, " ")
```

Koodilistaus 33 näyttää, kuinka RandomCombinationGenerator-luokka alustetaan kahdella listalla ja erotinmerkillä, joka on välilyönti. Generaattori voisi palauttaa generateValue()-metodin kutsuvaiheessa esimerkiksi merkkijonon ”Aarne Reinikainen”.

Generaattoreita voisi käyttää myös kätevämmin, muodostamalla listat jonkin Groovyn sulkeuman avulla. Lista voitaisiin muodostaa esimerkiksi etunimet.csv-tiedoston sisällöstä helposti (koodilistaus 34).

Koodilistaus 34. Listan muodostaminen tiedoston sisällöstä.

```
new File('etunimet.csv').readLines()
```

4.7.4 *TemplateGenerator ja SimpleValueGenerator*

TemplateGenerator-luokan (liite 7) avulla toisen generaattorin sisältö voidaan syöttää kaavaimena toimivan merkkijonon sisään. Generaattoria käytetään määrittelemällä sille merkkijonona malli, joka sisältää avainsanan ”*$generator*”. Generaattori ottaa konstruktorissa parametrina sekä kaavaimen, että toisen generaattorin ilmentymän. Arvoja generoitaessa TemplateGenerator-luokka syöttää siihen liitetyn toisen generaattorin arvon kaavaimen avainsanan tilalle.

Koodilistaus 35. TemplateGenerator-luokan alustus.

```
def template = "to_date('2008/09/\${generator}',
'yyyy/mm/dd')"

def numberGen = new RandomNumberGenerator(1..30, '00')

def templateGen = new TemplateGenerator(template, numberGen)
```

Koodilistaus 35 esittää, kuinka TemplateGenerator-luokkaa voidaan hyödyntää esim. satunnaisten päivämäärien generoinnissa. Ensin määritellään kaavain, jossa on tietokantaspesifinen päivämäärien esitystapa. Merkkijonoon syötetään ”${generator}”-merkintä, jolloin kaavainmoottori

osaa syöttää myöhemmin sen tilalle halutun arvon. TemplateGenerator-luokka alustetaan kaavaimella ja satunnaisten lukujen arpomiseen erikoistuneella generaattorilla.

Generaattorin (koodilistaus 35) generateValue()-metodia kutsuttaessa 3 kertaa generaattori tulostaa esim. seuraavat arvot:

- `to_date('2008/09/26', 'yyyy/mm/dd')`
- `to_date('2008/09/15', 'yyyy/mm/dd')`
- `to_date('2008/09/05', 'yyyy/mm/dd')`

SimpleValueGenerator-luokka (liite 8) on yksinkertainen generaattori, jonka avulla voidaan hyödyntää Groovyn GString-merkkijonoja, jotka tulivat tutuksi kappaleessa 3.3.2. Generaattori voidaan alustaa dynaamista sisältöä omaavalla GString-merkkijonolla, jolloin se palauttaa merkkijonon sisällön arvoja generoitaessa.

Koodilistaus 36. SimpleValueGenerator-luokan alustus.

```
def simpleGen = new SimpleValueGenerator("TODAY='${new
java.util.Date()}'")
```

Koodilistaus 36 näyttää, kuinka SimpleValueGenerator alustetaan dynaamisella GString-merkkijonolla. Dollarimerkin jälkeen alkavan kaarisulkujen määräämän osion sisältö evaluoidaan, kun generaattorista luodaan instanssi.

TemplateGenerator- ja SimpleValueGenerator-luokat ovat monikäyttöisiä koska ne toimivat kaavaimina, mikäli on tarve generoida hieman monimutkaisempia arvoja. SimpleValueGenerator-luokkaa voidaan käyttää myös muihin tarkoituksiin kuin GString-merkkijonoihin, koska se ottaa konstruktorissa parametrinaan def-tyyppisen parametrin, jolloin arvo voi olla mitä tahansa.

4.7.5 RandomFileLineSplitter

RandomFileLineSplitter-luokan (liite 9) avulla voidaan hyödyntää CSV-tiedoston sisältöä helposti, satunnaisen datan generoinnissa. RandomFileLineSplitter-luokka ottaa konstruktorissa parametrikseen tiedoston nimen, säännöllisen lausekkeen ja pilkotun merkkijonon taulukon indeksiarvon.

Arvoja generoitaessa generaattori hakee ensin annetusta tiedostosta satunnaisen rivin, ja pilkkoo sen String-luokan split-metodin avulla. Pilkonta suoritetaan parametrina annetun säännöllisen lausekkeen perusteella. Varsinainen palautettava arvo on *split*-metodin tuottaman merkkijonotaulukon alkio, joka määräytyy parametrina annetun indeksiarvon mukaan.

Koodilistaus 37. RandomFileLineSplitter-luokan käyttö.

```
def splitterGen = new RandomFileLineSplitter('country.csv',
                                                ",", 1)
```

Koodilistaus 37 näyttää, kuinka RandomFileLineSplitter-luokkaa voidaan käyttää. Luokka alustetaan country.csv-nimisellä tiedostolla, joka tuli tutuksi kappaleessa 3.3. Erotinmerkiksi määritellään pilkku, ja tiedostosta valitaan sarakkeen indeksiksi 1. Näin ollen generaattorin generateValues()-metodia kutsuttaessa se palauttaa maiden nimiä tiedostosta.

CSV-tiedostojen käyttö testidatan generoinnissa on kätevä keino, jos halutaan käyttää oikeata dataa jota on paljon saatavilla. Näin testidataan tulee paljon vaihtelua, mutta samalla tieto on järkevää.

4.7.6 NullGenerator

NullGenerator-luokka (liite 10) generoi *null*-arvoja satunnaisesti datan joukkoon. Generaattori ottaa konstruktorissa parametreina ValueGenerator-rajapintaa toteuttavan generaattorin, ja prosenttiluvun. NullGenerator-luokka palauttaa arvoja siihen liitetystä generaattorista, ja välillä null-arvoja, parametrina saadun prosenttiluvun mukaisesti.

Syöttämällä null-arvoja satunnaisesti testidatan joukkoon voidaan kokeilla, selviytyykö testattava sovellus tilanteesta, jolloin jokin arvo ei olekaan määritetty.

Koodilistaus 38. NullGenerator-luokan alustus.

```
def gen = new NullGenerator(new SimpleValueGenerator("test"),
4)
```

Koodilistaus 38 näyttää, kuinka NullGenerator-luokka alustetaan generaattorilla, ja luvulla 4. Mikäli määritellyn generaattorin

generateValues()-metodia kutsuttaisiin 10000 kertaa, se palauttaisi keskimäärin noin 9600 kertaa ”test”-merkkijonon, ja 400 kertaa null-arvon.

4.7.7 ClosureGenerator

ClosureGenerator-luokka (liite 11) generoi arvoja sulkeuman perusteella. Sulkeumia käsiteltiin luvussa 3.2.4. ClosureGenerator-luokka vastaanottaa konstruktorissa parametrina Closure-objektin. Luokka voidaan alustaa myös Closure-objektilla ja sille syötettävällä parametrilla.

Koodilistaus 39. ClosureGenerator-luokan käyttö.

```
Closure addRandomDigit = { text ->
   def addedList = text.asType(List).collect {
      it + (Math.random() * 10).asType(int)
   }
   String result = ""
   addedList.each { result += it }
   return result
}

def gen = new ClosureGenerator(addRandomDigit, "test")
```

Koodilistaus 39 esittää, kuinka ClosureGenerator-luokkaa voidaan käyttää. Alussa määritellään sulkeuma, joka ottaa parametrin sisäänsä. Parametria käsitellään niin kuin se olisi merkkijono, ja sen jokaisen merkin väliin lisätään satunnainen luku joukosta 0-9. Lopuksi palautetaan käsitelty merkkijono. Yllä määritellyn ClosureGenerator-luokan ilmentymän generateValue()-metodia kutsuttaessa, generaattori tulostaa esim. seuraavanlaisia arvoja:

- `t6e1s3t5`
- `t9e9s9t6`
- `t1e8s4t0`
- `t2e2s6t8.`

Sulkeuma voi sisältää minkälaista toiminnallisuutta tahansa, joten ClosureGenerator-luokan ansiosta ei ole aina järkevää ohjelmoida uutta generaattori-luokkaa, kun haetaan uutta toiminnallisuutta.

4.8 Ohjelman käyttö

Ohjelmaa on tarkoitus ajaa komentoriviltä, antamalla pääohjelmalle vain tarvittavat parametrit. Ohjelmalla ei ole graafista käyttöliittymää niin kuin

todettiin luvussa 4.1. Pääohjelma sijaitsee com.ixonos.sqlgenerator–pakkauksessa, ja se on kirjoitettu Java-ohjelmointikielellä.

Pääohjelmaa ei ole toteutettu Groovy-ohjelmointikielellä, koska ohjelmasta ei haluttu liian Groovy-riippuvaista. Groovy-kielellä kirjoitetun ohjelman ajaminen vaatii Groovyn asentamisen tietokoneelle, joka toisi lisää vaivaa ohjelman käyttöön. Ohjelma on toteutettu niin, että Javalla kirjoitettu pääohjelma GroovyGeneratorLauncher.java kutsuu Groovy-kielellä kirjoitettua GroovySQLGenerator.groovy-luokkaa välittäen parametrit komentoriviltä sille. Javalla kirjoitettu pääohjelma pystyy kutsumaan Groovylla kirjoitettua ohjelmaa, koska Groovy-riippuvaisuus on toteutettu lisäämällä luokkapolkuun Groovyn jar-paketti, joka sisältää kaikki Groovyn ajamiseen tarvittavat luokat. Ohjelman ajamiseen tarvitaan siis vain Javan ajonaikainen ympäristö (Java Runtime Environment).

Pääohjelmaa ajetaan komentoriviltä ja sille annetaan ajamisen yhteydessä 4 parametria:

- Malli, joka on Model-luokan toteuttaja.
- Sarjallistaja, joka on ModelSerializer-luokan toteuttaja.
- Tiedoston nimi, johon sarjallistajan tiedot kirjoitetaan.
- Kokonaislukuarvo, joka määrittää sen, montako kertaa Model-luokan toteuttajan mallia kutsutaan.

Ohjelman kokoaminen tapahtuu Ant-työkalun avulla. Ennen ohjelman ajoa on ajettava seuraava käsky: `ant clean jar`. Kyseinen käsky tuhoaa mahdolliset edelliset käännöstiedostot ja kääntää lähdekoodit sekä pakkaa ne jar-tiedostoksi. Paketoitu jar-tiedosto sijoitetaan *dist*-hakemistoon, ja ohjelmaa ajetaan käyttämällä sitä.

Koodilistaus 40. Pääohjelman ajaminen komentoriviltä.

```
java -cp dist/GroovySQLGenerator.jar:lib/groovy-all-1.5.7.jar
   com.ixonos.sqlgenerator.GroovyGeneratorLauncher
   com.ixonos.sqlgenerator.admodel.AdModel
   com.ixonos.sqlgenerator.serializer.OracleSQLSerializer
   "admodel-insert-statements.sql"
   100
```

Koodilistaus 40 esittää, kuinka pääohjelmaa voidaan ajaa komentoriviltä. Luokkapolkuun täytyy määritellä *groovy-all-x.x.x.jar*-paketin sijainti, jotta Groovy-riippuvaisuus saadaan täytettyä. Luokkapolkuun lisätään myös dist-hakemistossa sijaitseva jar-paketti, joka sisältää kaikki projektissa käytettävät luokat. Komentoriviltä kutsutaan pääohjelmaa, ja sille annetaan parametriksi malli, sarjallistaja, tiedoston nimi ja repeat-arvo. Luvussa 4.5.5 esiteltiin repeat-arvon merkitys.

Ohjelmaa ajettaessa komentoriville tulostuu tietoja ohjelman eri suoritusvaiheista (koodilistaus 41). Näytölle tulostetaan viesti, kun taulun sisältämien Entity-objektien arvoja aletaan generoida. Mikäli mallissa on hyvin paljon objekteja sisällä, voi mallin generoinnissa mennä kauan, joten on hyvä tietää, missä ohjelma on menossa.

Koodilistaus 41. Ohjelman suoritus.

```
Generating values for sum_ad_delivery
Generating values for d_ad
Generating values for d_customer
Serializing class com.ixonos.sqlgenerator.admodel.AdModel
Generated 940 rows.
Done.

Wrote to file: admodel-insert-statements.sql
```

4.9 Mallin luonti ja ohjelman laajennus

Tässä kappaleessa tutustutaan mallin luontiin. Jotta ohjelmaa voidaan käyttää tietokannan testidatan generointiin, niin tietokannasta täytyy ohjelmoida malli. Oman mallin luonti käsittää seuraavat vaiheet:

- Mallille luodaan oma pakkaus (package).
- Tietokantaa vastaavat Entity-luokat ohjelmoidaan.
- Haluttua testidataa vastaava malli luodaan ohjelmoimalla.

Seuraavaksi toteutetaan yksinkertainen esimerkki, kappaleessa 4.6.2 esitellystä Company-Customer-Ad –mallista (yritys, asiakas, mainos). Uutta toteutusta varten luodaan pakkaus ”ccamodel”, ja toteutetaan tietokannan tauluja vastaavat Entity-luokat.

Yrityksellä voisi olla tietokannassa *id*-kenttä, jonka oletetaan olevan uniikki perusavain. Yritykselle määritellään myös *name*-kenttä, johon syötetään yrityksen nimi.

Koodilistaus 42. Company.groovy –luokan toteutus.

```
package com.ixonos.sqlgenerator.ccamodel

import com.ixonos.sqlgenerator.model.*
import com.ixonos.sqlgenerator.generator.*

class Company extends AbstractEntity {

  Property id
  Property name

  Company() {
    id = new Property( type: long, generator:
                new UniqueGlobalIdGenerator())
    name = new Property( type: String, generator:
                new RandomWordGenerator(['Ixonos', 'Mermit']))
  }
}
```

Koodilistaus 42 esittää Company-luokan toteutuksen. Luokka perii AbstractEntity-luokan, ja siinä määritellään kaksi Property-objektia: *id* ja *name*. Property-objektit luodaan konstruktorissa, ja niille määritellään tyypit ja generaattorit. Koska id-kenttä kuvastaa perusavainta, sille määritellään generaattoriksi uniikkien avainten generaattori. Luokan name-kenttä asetetaan merkkijono-tyyppiseksi, ja sille määritellään generaattoriksi satunnaisten merkkijonojen generointiin erikoistunut generaattori.

Koodilistaus 43. Customer.groovy -luokan toteutus.

```
package com.ixonos.sqlgenerator.ccamodel

import com.ixonos.sqlgenerator.model.*
import com.ixonos.sqlgenerator.generator.*

class Customer extends AbstractEntity {

  Property companyId
  Property id
  Property name
  Property age
  Property country

  // Customer-luokan generaattori
  static def customerIdGenerator = new UniqueIdGenerator(1)

  Customer() {
    companyId = new Property()
```

```
    id = new Property( type: long, generator:
            customerIdGenerator )
    name = new Property( type: String, generator:
            new RandomStringGenerator(10, 'a'..'z') )
    age = new Property( type: long, generator:
            new RandomNumberGenerator(18..35) )
    country = new Property( type: String, generator:
            new RandomFileLineSplitter('country.csv', ',', 1))
  }
}
```

Koodilistaus 43 esittää Customer-luokan toteutuksen. Luokkaan määritellään *companyId*-niminen Property-objekti, johon voidaan määritellä yrityksen id, johon asiakas kuuluu. Property-objekti alustetaan, mutta siihen ei määritellä mitään arvoja, koska se asetetaan myöhemmin viittaamaan yrityksen id-kenttään. Luokkaan määritellään myös Customer-luokkakohtainen avaingeneraattori, joka alustetaan alkamaan numerosta yksi.

Koodilistaus 44. Ad.groovy -luokan toteutus.

```
package com.ixonos.sqlgenerator.ccamodel

import com.ixonos.sqlgenerator.model.*
import com.ixonos.sqlgenerator.generator.*

class Ad extends AbstractEntity {

  Property id
  Property customerId
  Property content

  // Ad-luokan generaattori
  static def adIdGenerator = new UniqueIdGenerator(1)

  Ad() {
    id = new Property( type: long, generator: adIdGenerator )
    content = new Property( type: String, generator:
                        new RandomStringGenerator(20, 'abc') )
    customerId = new Property()
  }
}
```

Myös Ad-luokassa määritellään luokkakohtainen id-generaattori (koodilistaus 44). Ad-luokan customerId-Property asetetaan viittaamaan asiakkaaseen seuraavaksi kun luodaan malli. Ad-luokan customerId-niminen Property-objekti jätetään taas tyhjäksi, koska se asetetaan mallissa viittausarvoksi.

Koodilistaus 45 esittää CCAModel-luokan toteutuksen, eli Customer-Company-Ad –mallin. Luokka perii AbstractModel-luokan, ja luokassa määritellään Table-komponentit sen ominaisuuksiksi.

Luokan defineTables()-metodissa määritellään ja instantioidaan luokan sisältämät taulut. Company- ja Ad-tauluja varten luotaville Table-objekteille määritellään sarakkeiden nimet uusiksi. Tässä kohdassa on huomioitava, että Customer-taulua vastaava Table-objekti alustetaan ilman sarakkeiden nimiä. Tällöin sarakkeiden nimet tulevat automaattisesti Entity-objektin sisällön perusteella, kuten luvussa 4.5.6 esitettiin.

Koodilistaus 45. CCAModel.groovy -luokan toteutus.

```
package com.ixonos.sqlgenerator.ccamodel

import com.ixonos.sqlgenerator.model.*

class CCAModel extends AbstractModel {

  Table companyTable
  Table customerTable
  Table adTable

  public void defineTables() {
    def companyColumns = [ 'id' : 'COMPANY_ID',
                           'name' : 'COMPANY_NAME' ]
    def adColumns = [ 'id' : 'AD_ID',
                      'customerId' : 'CUSTOMER_ID',
                      'content' : 'CONTENT' ]
    companyTable = new Table("D_COMPANY", companyColumns)
    customerTable = new Table("D_CUSTOMER")
    adTable = new Table("D_AD", adColumns)
  }

  public void populateRow() {
    Company company = new Company()
    companyTable.addEntity(company)
    2.times {
      Customer customer = new Customer()
      customer.companyId.refersTo = company.id
      customerTable.addEntity(customer)
      2.times {
        Ad ad = new Ad()
        ad.customerId.refersTo = customer.id
        adTable.addEntity(ad)
      }
    }
  }
}
```

CCAModel-luokan *populateRow*()-metodissa (koodilistaus 45) luodaan Company-luokasta ilmentymä, joka lisätään sitä vastaavaan Table-objektiin addEntity()-metodin avulla. Seuraavaksi jokaista Company-luokan ilmentymää vasten luodaan kaksi Customer-luokan ilmentymää. Asiakkaiden *companyId*-objekti asetetaan viittaamaan yrityksen *id*-objektiin. Asiakas lisätään sitä vastaavaan Table-objektiin. Jokaista asiakasta vasten luodaan myös kaksi Ad-luokan ilmentymää. Mainosten *customerId*-objekti asetetaan

viittaamaan vastaavasti asiakkaan *id*-objektiin. Mainos lisätään myös sitä vastaavaan Table-objektiin.

Malli on määritetty tässä luokassa siten, että jokaista populateRow()-metodin kutsukertaa vasten luodaan yksi yritys, kaksi asiakasta ja neljä mainosta. Jos ohjelmaa ajetaan komentoriviltä repeat-arvolla yksi, niin silloin ohjelma generoi 7 riviä luontilauseita, ja repeat-arvolla kaksi taas 14 riviä lauseita (commit-käskyjä ei lasketa mukaan). Luvussa 4.5.5 esiteltiin repeat-arvon toimintaperiaate.

Kappaleessa 5.1 on esitetty tulokset, kun tietokannan testidatageneraattorille on annettu tässä kappaleessa määritetty malli (koodilistaus 45) sarjallistettavaksi OracleSQLSerializer-luokan avulla repeat-arvolla kaksi.

5 TULOKSET

5.1 Uusi tietokannan testidatageneraattori

Suunnittelun ja ohjelmoinnin tuloksena syntyi uusi testidatan generointiin erikoistunut sovellus. Tavoitteena oli mahdollistaa viittaukset taulujen välillä. Tavoite täyttyi, kuten kappaleen 4.9 perusteella voidaan todeta.

Koodilistaus 46 esittää kappaleessa 4.9 esitellyn mallin perusteella generoitua testidataa. Generaattoria on kutsuttu repeat-arvolla kaksi, jolloin se generoi 14 riviä tietokannan INSERT-lauseita mallista.

Koodilistaus 46. Toteutetun ohjelman avulla tuotettua testidataa (CCAModel-malli)

```
INSERT INTO D_COMPANY(COMPANY_ID, COMPANY_NAME)
  VALUES(1, 'Ixonos');
INSERT INTO D_COMPANY(COMPANY_ID, COMPANY_NAME)
  VALUES(2, 'Mermit');

INSERT INTO D_CUSTOMER(companyId, id, age, country, name)
  VALUES(1, 1, 21, 'Ethiopia', 'optxadiwgg');
INSERT INTO D_CUSTOMER(companyId, id, age, country, name)
  VALUES(1, 2, 27, 'Guinea-Bissau', 'dqpbdlzzyw');
INSERT INTO D_CUSTOMER(companyId, id, age, country, name)
  VALUES(2, 3, 31, 'United States', 'rduhffdgdf');
INSERT INTO D_CUSTOMER(companyId, id, age, country, name)
```

```
  VALUES(2, 4, 35, 'Austria', 'yovthbdhlu');

INSERT INTO D_AD(CONTENT, AD_ID, CUSTOMER_ID)
  VALUES('abccbaaabacccbaaaaba', 1, 1);
INSERT INTO D_AD(CONTENT, AD_ID, CUSTOMER_ID)
  VALUES('cccacccabbcaabcbcbac', 2, 1);
INSERT INTO D_AD(CONTENT, AD_ID, CUSTOMER_ID)
  VALUES('bbababbacbbbcabbbaaa', 3, 2);
INSERT INTO D_AD(CONTENT, AD_ID, CUSTOMER_ID)
  VALUES('cbbccacbaacaccabcbab', 4, 2);
INSERT INTO D_AD(CONTENT, AD_ID, CUSTOMER_ID)
  VALUES('babaacabbcaaccccbbac', 5, 3);
INSERT INTO D_AD(CONTENT, AD_ID, CUSTOMER_ID)
  VALUES('bbabaaaaacacaaabccac', 6, 3);
INSERT INTO D_AD(CONTENT, AD_ID, CUSTOMER_ID)
  VALUES('aabccaacbcacbccacccc', 7, 4);
INSERT INTO D_AD(CONTENT, AD_ID, CUSTOMER_ID)
  VALUES('baabacacccabbbaccbaa', 8, 4);

COMMIT;
```

Asiakkaat kuuluvat tiettyihin yrityksiin, ja mainokset tiettyihin asiakkaisiin, kuten koodilistaus 46 osoittaa. Ohjelmoijan on helppo toteuttaa generaattoreita rajapintojen määrittelemillä tavoilla itse, joten myös käytettävyys ja ylläpidettävyys ovat onnistuneet.

Yleisesti ottaen uusi sovellus näyttää toimivan hyvin, ja on hyvä korvaus vanhalle ohjelmalle. On mahdollista, että ohjelmaa jatkokehitetään vielä selviämään astetta haastavammista tietokannan tiedon malleista.

5.2 Ongelmat ohjelmointivaiheessa

Tavoitteena oli myös saada tietoa Groovysta yleisesti. Tavoite täyttyi siltä osin että Groovy tuli hyvin tutuksi työn teon ohessa. Groovy nopeutti useassa tilanteessa ongelmien ratkaisua ja ohjelmointia, mutta aiheutti välillä myös ongelmatilanteita, jotka veivät paljon aikaa. Asiaan vaikuttaa paljon se, että Groovysta ei ollut aiempaa kokemusta.

Groovyn ongelmatilanteet liittyivät usein siihen, että ohjelmointivirheiden selvittäminen ei ollut niin suoraviivaista kuin Javalla. Vaikka Eclipse tukeekin Groovy-kielen debuggausta, oli muutama tilanne sellainen että debuggerista

ei ollut mitään hyötyä. Virheilmoitukset eivät kertoneet vian todellisesta aiheuttajasta juuri mitään, ja tällöin vian selvittämiseen tuhraantui paljon aikaa.

Eclipsen Groovy-liitännäinen kaatoi myös välillä Eclipsen, tai hidasti sitä niin että järjestelmä ei vastannut kymmeniin sekunteihin. Välillä luokkatiedostot eivät kääntyneet oikein, jolloin ohjelmaa ei pystynyt ajamaan. Myös yksikkötestien ajamisessa tuli samanlaisia ongelmia aika ajoin. Kaikki Eclipse-kehitysympäristön ominaisuudet eivät olleet aina käytettävissä, kuten sisällön avustaja. Koodin syötön avustaja hidasti Eclipseä useimmissa tapauksissa niin, että työnteko hidastui merkittävästi. Myös uusien luokkien luonnin yhteydessä tuli usein ongelmia, kun yritti luoda luokkaa velhon avulla, määrittäen perittävän kantaluokan tai toteutettavan rajapinnan. Tämä johti käsin tehtävän työn lisääntymiseen. Eclipsen Groovy-liitännäinen on kuitenkin jatkuvasti kehityksen alla, joten se luultavasti tulee kehittymään parempaan suuntaan.

6 YHTEENVETO

Tässä opinnäytetyössä kehitettiin vanhan tietokannan testidatageneraattorin tilalle uusi sovellus. Tavoitteena oli tehdä sellainen generaattori, joka osaisi käsitellä taulujen väliset viittaukset kätevästi. Tavoitteena oli myös, että ohjelma olisi helposti laajennettavissa ja muokattavissa. Ohjelman ohjelmointikielenä käytettiin Groovya, josta oli myös tavoitteena saada lisää tietoa.

Taulujen välisiin viittauksiin kehitettiin yksinkertainen ja helppo menetelmä mallista käsin. Viittaukset toimivat hyvin, ja ne tekevät tietokannan testidatasta järkevää. Ohjelmassa käytetään myös selviä rajapintoja, jolloin ohjelma on helposti laajennettavissa ja muokattavissa.

Työssä käsiteltiin aluksi yleisesti komentosarjakieliä, ja sen jälkeen tarkemmin Groovy-kieltä ja sen ominaisuuksia. Groovysta saatiin arvokasta kokemusta ja siitä huomattiin sekä hyviä että huonoja puolia. Välillä ongelmatilanteet olivat hankalasti selvitettävissä, mutta välillä Groovy näytti kyntensä ohjelmointiongelmien ratkaisujen helppoudessa ja lyhyydessä.

VIITELUETTELO

[1] Koenig, Dierk – Glover, Andrew, *Groovy in Action*. Manning. 2007.

[2] Laforge, Guillaume, JSR 241: *The Groovy Programming Language*. [Verkkodokumentti, viitattu 12.11.2008]. Saatavilla: http://jcp.org/en/jsr/detail?id=241.

[3] Premshree Pillai, *Introduction to Static and Dynamic Typing*. [Verkkodokumentti, viitattu 12.11.2008]. Saatavilla: http://www.sitepoint.com/article/typing-versus-dynamic-typing/.

[4] Baars, Arthur – Swierstra, Doaitse, *Typing Dynamic Typing*. [Verkkodokumentti, viitattu 22.10.2008]. Saatavilla: http://people.cs.uu.nl/arthurb/data/Dynamic/p189-baars.pdf.

[5] Wikipedia, *Type system*. [Verkkodokumentti, viitattu 22.10.2008]. Saatavilla: http://en.wikipedia.org/wiki/Type_system.

[5] Codehaus Foundation, *Closures*. [Verkkodokumentti, viitattu 27.10.2008]. Saatavilla: http://groovy.codehaus.org/Closures.

[6] Wikipedia, *Java package.* [Verkkodokumentti, viitattu 05.11.2008]. Saatavilla: http://en.wikipedia.org/wiki/Java_package.

[7] Wikipedia, *Scripting language*. [Verkkodokumentti, viitattu 14.11.2008]. Saatavilla: http://en.wikipedia.org/wiki/Scripting_language.

[8] Wikipedia, *Komentosarjakieli*. [Verkkodokumentti, viitattu 14.11.2008]. http://fi.wikipedia.org/wiki/Komentosarjakieli.

[9] Wikipedia, *Unix shell*. [Verkkodokumentti, viitattu 14.11.2008]. Saatavilla: http://en.wikipedia.org/wiki/Unix_shell.

[10] Wikipedia, *Batch file*. [Verkkodokumentti, viitattu 05.11.2008]. Saatavilla: http://en.wikipedia.org/wiki/Batch_file.

[11] Wikipedia, *Ohjelmointi*. [Verkkodokumentti, viitattu 09.11.2008]. Saatavilla: http://fi.wikipedia.org/wiki/Ohjelmointi.

[12] Wikipedia, *Dynamic Programming language*. [Verkkodokumentti, viitattu 05.11.2008]. Saatavilla: http://en.wikipedia.org/wiki/Dynamic_programming_language.

[13] Emberg, Pekka – Rauni Mika, *Sulkeumat*. [Verkkodokumentti, viitattu 11.11.2008]. Saatavilla: http://www.cs.helsinki.fi/u/wikla/OKP/ArtikkelitK07/sulkeumat.pdf.

[14] Wikipedia, *Reflection (computer science)*. [Verkkodokumentti, viitattu 12.11.2008]. Saatavilla: http://en.wikipedia.org/wiki/Reflection_(computer_science).

[15] Gamma, Erich, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. USA. 1995.

```
package com.ixonos.sqlgenerator.generator
import com.ixonos.sqlgenerator.*

/**
 * Generates unique id:s as an instance.
 *
 * @author villek
 *
 */
public class UniqueIdGenerator implements ValueGenerator {

  UniqueIdGenerator(Integer initialValue) {
    this.id = initialValue
  }

  def getCurrentValue() {
    id
  }

  def generateValue() {
    return id++
  }

  private long id

}
```

```
package com.ixonos.sqlgenerator.generator
import com.ixonos.sqlgenerator.*

/**
 * Global UniqueIdGenerator implemented as Singleton.
 *
 * Generates unique IDs, starting with initial value.
 *
 * @author villek
 */

class UniqueGlobalIdGenerator implements ValueGenerator {

  // Hidden constructor
  private UniqueGlobalIdGenerator() {}

  /**
  * Accessor for UniqueIdGenerator
  */
  synchronized static UniqueGlobalIdGenerator getUniqueGlobalIdGenera-
tor() {
    if (ref == null) {
      ref = new UniqueGlobalIdGenerator()
    }
    return ref
  }

  /**
  *  Returns a unique value.
  */
  synchronized def generateValue() {
    return id++
  }

  def clone() {
    throw new CloneNotSupportedException()
  }

  // Initial value
  private static long id = 1
  private static UniqueGlobalIdGenerator ref
}
```

```
package com.ixonos.sqlgenerator.generator

/**
 * Generates random String values.
 * There are 3 modes:
 * 1) If only lenght of the string given, it generates random characters
 * 2) If length & onlyChars is given, it generates a random String using
      only characters in 'onlyChars'
 *     * onlyChars-example: 'aääöz +32"
 * 3) If length & charRange is given, it generates a random String using
      characters in range of 'charRange'
 *     * charRange-example: 'a'..'z'
 */

class RandomStringGenerator implements ValueGenerator {

   /**
   * Generate String with random characters from charRange.
   */
   RandomStringGenerator(int stringLength) {
     length = stringLength ?: 5

     resultMaker = { length ->
       StringBuffer randomString = new StringBuffer()
       length.times {
         randomString.append( ((Math.random() *
           128).asType(Integer)).asType(Character) )

       }
       return randomString
     }
   }

   /**
   * Generates a random String using only characters supplied with
     onlyChars parameter.
   */
   RandomStringGenerator(int stringLength, String onlyChars) {
     length = stringLength ?: 5
     def charList = onlyChars.asType(List)

     resultMaker = { length ->
       StringBuffer randomString = new StringBuffer()
       length.times {
         randomString.append( charList[(Math.random() *
           onlyChars.size()).asType(Integer)] )
       }
       return randomString
     }
   }

   /**
    * Generates a random String using characters in the charRange.
    */
   RandomStringGenerator(int stringLength, List charRange) {
     charRange = charRange ?: 'a'..'c'
     length = stringLength ?: 1
```

```
      resultMaker = { length ->
        StringBuffer b = new StringBuffer()
        length.times {
          b.append(charRange[(Math.random() *
            charRange.size()).asType(int)])
        }
        return b.toString()
      }
    }

    String generateValue() {
      return resultMaker ? resultMaker (length) : null
    }

    private int length
    private Closure resultMaker
}
```

```
package com.ixonos.sqlgenerator.generator
import java.text.*

/**
 * Generates a random number from provided range.
 *
 * Uses DecimalFormat if pattern is provided.
 *
 * @author Ville
 *
 */
class RandomNumberGenerator implements ValueGenerator {

  IntRange range;
  DecimalFormat df

  RandomNumberGenerator(IntRange range) {
    this.range = range
  }

  RandomNumberGenerator(IntRange range, String pattern) {
    this.range = range
    this.df = new DecimalFormat(pattern)
  }

  public def generateValue() {
          def randomValue = range.toList()[(Math.random() *
            range.size()).asType(Integer)]
          df ? df.format(randomValue) : randomValue
  }
}
```

```
package com.ixonos.sqlgenerator.generator
/**
 * Returns random word from array.
 * @author villek
 *
 */

class RandomWordGenerator implements ValueGenerator {

  def wordList = []

  RandomWordGenerator(List list) {
    wordList = list
  }

  /*
  * Returns a random word from the list as String.
  */
  String generateValue() {
    wordList ? wordList[(Math.random() *
            wordList.size()).asType(Integer)].asType(String) : null
  }
}
```

```
package com.ixonos.sqlgenerator.generator
/**
 * Combines a word from both lists. Separates them with
 * provided separator.
 *
 * @author Ville
 *
 */

class RandomCombinationGenerator implements ValueGenerator {

  String separator
  List list1, list2

  RandomCombinationGenerator(List list1, List list2, String separator) {
    this.list1 = list1
    this.list2 = list2
    this.separator = separator
  }

  /**
  * Takes one word from both lists randomly and separates them with
    @separator.
  */
  String generateValue() {
    return (new RandomWordGenerator(list1).generateValue() + separator +
            new RandomWordGenerator(list2).generateValue())
  }
}
```

```
package com.ixonos.sqlgenerator.generator
import com.ixonos.sqlgenerator.exception.*

/**
 * Generates a String value with template.
 * This generator assumes that the template is provided with $generator
   / ${generator}
 *
 * Remember to use 'single quotes' when defining the template.
 *
 * Example template:
 * 'Generators value $generator gets inserted there'
 *
 * @author villek
 *
 */
public class TemplateGenerator implements ValueGenerator {

   String template
   ValueGenerator aGenerator

   TemplateGenerator(String template, ValueGenerator aGenerator) {
     if (!(template =~ "generator")) {
       throw new ValueGeneratorException("${this.getClass()} needs
                                \$generator defined in a template.")
     }
     this.template = template
     this.aGenerator = aGenerator
   }

   String generateValue() {
     def engine = new groovy.text.SimpleTemplateEngine()
     def template = engine.createTemplate(template)
     def binding = [ "generator" : aGenerator?.generateValue() ]
     return template.make(binding)
   }
}
```

```
package com.ixonos.sqlgenerator.generator
import com.ixonos.sqlgenerator.*
/**
 * Simple value generator.
 * Can contain any types of variables.
 *
 * For example, a GString: "System time = ${System.currentTimeMillis()}"
 *
 * @author villek
 */
public class SimpleValueGenerator implements ValueGenerator {

   def x

   SimpleValueGenerator(def x) {
     this.x = x
   }

   def generateValue() {
     return x
   }
}
```

```
package com.ixonos.sqlgenerator.generator

/**
 * Generates a random value from file.
 * Value is splitted by a splitRegexp.
 *
 * @author villek
 *
 */
public class RandomFileLineSplitter implements ValueGenerator {

  String fileName, splitRegexp
  Integer splitNumber

  /**
  * file = Filename to read
  * splitRegexp = Regular expression to split the line
  * splitNumber = element's index in splitted String[]-array
  */
  RandomFileLineSplitter(String file, String splitRegexp, Integer
                                                        splitNumber) {
    this.fileName = file
    this.splitRegexp = splitRegexp
    this.splitNumber = splitNumber
  }

  def generateValue() {
    String value = null
    try {
      value = FileReader.readRandomLineFromFile(fileName)?
                                      .split(splitRegexp)[splitNumber]
    } catch (Exception ex) {
      // returning null value
    }
    return value
  }
}
```

```
package com.ixonos.sqlgenerator.generator
/**
 * Generates null-values randomly to attached ValueGenerator.
 * Percentage is the % amount how often null-values are
 * generated, for about.
 *
 * When generating 10000 values the error margin is -1.5 .. +1.5 %
 *
 * @author villek
 *
 */
class NullGenerator implements ValueGenerator {

   ValueGenerator generator
   int percentage

   NullGenerator(ValueGenerator generator, int percentage) {
     this.generator = generator
     this.percentage = percentage
   }

   def generateValue() {
     int n = (int)(Math.random() * 100)
     if (n < percentage) {
       return null
     }
     return generator.generateValue()
   }
}
```

```groovy
package com.ixonos.sqlgenerator.generator

/**
 * Generates a value from a Closure.
 * Closure is called with a parameter if it is set.
 *
 * @author villek
 *
 */
public class ClosureGenerator implements ValueGenerator {

  // Closure to be executed
  Closure closure

  // Parameter for Closure
  def parameter

  ClosureGenerator(Closure closure, def parameter) {
    this.closure = closure
    this.parameter = parameter
  }

  ClosureGenerator(Closure closure) {
    this.closure = closure
  }

  def generateValue() {
    parameter ? closure.call(parameter) : closure.call()
  }

}
```