Aku Kalajo

EJB:n eri käyttötapojen tehokkuus tiedonsiirrossa tietokantaan käyttäen OC4J palvelinta

Tietojärjestelmätieteen
pro gradu –tutkielma
26.4.2002

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä 2001

TIIVISTELMÄ

Kalajo, Aku-Joona
EJB:n eri käyttötapojen tehokkuus tiedonsiirrossa tietokantaan käyttäen OC4J palvelinta./Aku Kalajo
Jyväskylä: Jyväskylän yliopisto, 2002.
100 s.
Tietojärjestelmätieteen pro gradu –tutkielma

Enterprise JavaBeans (EJB)-komponenttimalli on hyvin yleinen ja suuren suosion saanut ratkaisu hajautetuissa järjestelmissä. Mallin keskeisimpiä piirteitä ovat komponenttipohjainen toteutus ja komponenttien hajautus. Tässä tutkielmassa toteutetaan konstruktio testausohjelmistosta, jossa tietoa viedään EJB-mallia eri tavoin käyttäen tietokantaan ja selvitetään konstruktiolla tehtävän testauksen avulla EJB-mallin käyttötapojen suorituskykyeroja. Tutkimuksen keskeisimpänä mielenkiinnon kohteena on kaksi yleisimmin käytettyä EJB-komponenttien käyttötapaa siirrettäessä tietoa tietokantaan. Tutkimuksen motiivina ovat yleisesti havaitut tehokkuusongelmat toisessa näistä käyttötavoista etenkin siirrettäessä suuria määriä tietoa tietokantaan. Tutkimuksessa selvitetään, miten nämä suorituskykyerot muodostuvat siirryttäessä yhä suurempiin tietomääriin ja kuinka merkittäviä nämä erot ovat. Tutkimus on osin konstruktiivinen, osin empiirinen. Konstruktio-osassa suunnitellaan ja toteutetaan konstruktio, jossa suorituskykyerojen testaaminen tapahtuu. Empiirisessä osassa suoritetaan testaus käyttäen Oracle Containers for J2EE (OC4J)-palvelinta ja analysoidaan testin tulokset.

Tutkielman keskeisimpinä tuotoksina ovat testauksen viitekehys ja siihen pohjautuvan testausohjelmiston rakentaminen sekä tapauskohtaisten testaustulosten analyysi ja vertailu. Tutkimuksen tärkeimpänä tuloksena havaittiin EJB-komponenttimallin eri käyttötapojen suorituskykyerojen olevan merkittävä suurien tietomassojen siirrossa.

AVAINSANAT: monitasoarkkitehtuuri, komponenttimalli, Enterprise JavaBeans, Java 2 Enterprise Edition, suorituskykytestaus, Oracle tietokanta

SISÄLLYSLUETTELO

1 JOHDANTO 1

2 EJB-KOMPONENTTIMALLI 4

2.1 SUORITUSYMPÄRISTÖ 6

2.1.1 Palvelimen ja säiliön tehtävät 6

2.1.2 Hajautus 7

2.1.3 Rajapinnat 8

2.1.4 Ajonaikainen ympäristö 9

2.2 EJB-KOMPONENTIT 10

2.2.1 Istuntopohjainen komponentti 11

2.2.2 Kohdepohjainen komponentti 15

2.3 KOMPONENTIT SUORITUSYMPÄRISTÖSSÄ 19

2.3.1 Sijoittaminen ympäristöön 19

2.3.2 Asiakassovelluksen näkökulma 20

2.3.3 Poikkeusten käsittely 20

3 TESTAUKSEN VIITEKEHYS 22

3.1 KOMPONENTTI TESTAUKSEN VIITEKEHYKSESSÄ 22

3.2 VIITEKEHYKSEN MÄÄRITYS 24

3.2.1 Asiakasohjelma 25

3.2.2 EJB-komponenttien käyttötavat 25

3.2.3 Tietokanta ja tietotyypit 27

3.2.4 Viitekehyksen arviointia 29

3.3 EJB:N KÄYTTÖTAPOJEN VALINTA 30

3.3.1 Tilallinen ja tilaton istuntopohjainen komponentti 31

3.3.2 CMP- ja BMP-komponentti 32

3.3.3 Käyttötapojen vertailua 32

3.3.4 Tietokantayhteyden hallinta komponentissa 34

4 TESTAUSYMPÄRISTÖ 36

4.1 TESTAUSOHJELMISTON VAATIMUKSISTA 36

4.2 KOKONAISARKKITEHTUURI 38

4.2.1 Kehittämisnäkymä 40

4.2.2 Skenaarionäkymä ... *41*
4.3 ASIAKASOHJELMA ... 43
4.4 EJB-KOMPONENTIT ... 44
4.4.1 sEJBInsert ... *44*
4.4.2 eEJBData ... *46*
4.4.3 sEJBParser ... *47*
4.4.4 Kuvaustiedosto ... *47*
4.5 TIETOKANTA ... 48
4.6 YMPÄRISTÖ ... 49
4.6.1 Oracle9iAS palvelin ... *49*
4.6.2 Laitteisto ... *50*
5 TESTITAPAUKSET JA TESTAAMINEN ... 51
5.1 TESTITAPAUKSET ... 51
5.1.1 Testausmateriaalin valinnan lähtökohdat ... *51*
5.1.2 Testausmateriaalin esittely ... *53*
5.1.3 Testitapausten esittely ... *54*
5.2 TESTAUSTAPAHTUMAT ... 55
5.2.1 Testaustapahtuman käsite ... *55*
5.2.2 Vieraat havainnot ... *56*
5.3 TESTIN TULOKSET ... 57
5.3.1 Havaintoja testauksessa ... *57*
5.3.2 Testauksen eteneminen ... *58*
5.3.3 Tulokset ... *58*
6 TULOSTEN ARVIOINTI ... 62
6.1 ANALYYSIN VIITEKEHYS ... 62
6.1.1 Testitapakohtainen analyysin viitekehys ... *63*
6.1.2 Testitapojen vertailun viitekehys ... *64*
6.2 TESTITAPAKOHTAINEN ANALYYSI ... 66
6.2.1 Suoraviivainen puhdas tapa ... *66*
6.2.2 Oppikirjamainen puhdas tapa ... *67*
6.2.3 Suoraviivainen käytännön tapa ... *68*
6.2.4 Oppikirjamainen käytännön tapa ... *69*

6.2.5 Mittausten reliabiliteetti ja validiteetti ... *70*
6.3 TESTITAPOJEN VERTAILUA ... 71
6.3.1 Pienen ja ison mittakaavan vertailua ... *72*
6.3.2 Puhtaan ja käytännön tavan vertailua ... *74*
6.3.3 Suoraviivaisen ja oppikirjamaisen tavan vertailua ... *75*
6.4 JOHTOPÄÄTÖKSET ... 78
6.4.1 Vertailun yhteenveto ... *78*
6.4.2 Tulosten merkityksen arviointia ... *79*
7 YHTEENVETO ... 81
LÄHDELUETTELO ... 84

KUVIEN LUETTELO

Kuva 2.1: Asiakasohjelman ja EJB-komponentin välinen kommunikointi 7
Kuva 2.2: Tilattoman istuntopohjaisen komponentin elinkaari..... 13
Kuva 2.3: Tilallisen istuntopohjaisen komponentin elinkaari. 14
Kuva 2.4: Kohdepohjaisen EJB-komponentin elinkaari. 18
Kuva 4.1: Testausohjelmiston komponentit ja niiden riippuvuudet..... 40
Kuva 4.2: Testausohjelmiston ajonaikainen toiminta..... 42
Kuva 5.1: Testitapausten ensimmäisen läpikäynnin tulokset..... 59
Kuva 5.2: Testin lopulliset tulokset..... 61
Kuva 6.1: Suoraviivaisen puhtaan tavan regressioanalyysin tulokset..... 67
Kuva 6.2: Oppikirjamaisen puhtaan tavan regressioanalyysit..... 68
Kuva 6.3: Suoraviivaisen käytännön tavan regressioanalyysin tulokset..... 69
Kuva 6.4: Oppikirjamaisen käytännön tavan regressioanalyysin tulokset. 70
Kuva 6.5: Pienen ja ison mittakaavan aineistojen kuvaajat..... 73
Kuva 6.6: Kulmakertoimien prosentuaaliset kasvut siirryttäessä suoraviivaisen tavan aineistoista oppikirjamaisen tavan aineistoihin. 74
Kuva 6.7: Suoraviivaisen ja oppikirjamaisen tavan suorituskykyerojen muodostuminen. 76

TAULUKOIDEN LUETTELO

Taulukko 2.1: *javax.ejb.SessionBean* -rajapinnan metodit..... 11
Taulukko 2.2: *javax.ejb.EntityBean* rajapinnan metodit. 17
Taulukko 4.1: *sEJBInsert*-komponentin etärajapinta 45
Taulukko 4.2: *metering_data* -taulun kentät ja niiden tietotyypit sekä niitä vastaavat attribuutit ja attribuuttien tietotyypit EJB-komponentissa..... 49
Taulukko 5.1: Testausmateriaalissa käytetyt kentät. 54
Taulukko 6.1: Aineistojen kulmakertoimien keskiarvot. 72
Taulukko 6.2: Ison mittakaavan aineistojen kuvaajien funktiot..... 76

1 JOHDANTO

Enterprise JavaBeans (EJB)-komponenttimalli on hyvin yleinen ja suuren suosion saanut ratkaisu hajautetuissa järjestelmissä. Se on yksi keskeisimpiä, ellei keskeisin määritys Java 2 Enterprise Edition (J2EE)-ohjelmointialustan määrityksistä. Mallin keskeisimpiä piirteitä ovat komponenttipohjainen toteutus ja komponenttien hajautus. Se tarjoaa ratkaisuja muun muassa tapahtumanhallintaan, monipuoliseen viestinvälitykseen ja näihin liittyviin tietoturvakysymyksiin. Suuren suosionsa myötä EJB-komponenttimallia myös käytetään hyvin laajasti ja monen tyyppisissä sovelluksissa.

Mahdollisten sovellusalueidensa laajan kirjon myötä on oletettavaa, että joissain EJB-komponenttimallia käyttävissä sovelluksissa tullaan tilanteeseen, jossa suurten tietomassojen vienti tietokantaan käy tarpeelliseksi. Usein juuri suurten tietomassojen käsittely ja siirtely muodostuu sovelluksen suorituskyvyn kannalta ohjelmiston kriittisimmäksi osaksi. Tutkielman aiheenvalinnan motiivina on kaupallisessa käytössä oleva järjestelmä, jossa eräajoina siirretään joka vuorokausi erittäin suuria määriä energianmittaustietoa mittareista tietokantaan.

Tämän tutkielman keskeisimpinä tavoitteina on toteuttaa konstruktio testausohjelmistosta, jossa tietoa viedään EJB-mallia eri tavoin käyttäen tietokantaan, sekä selvittää konstruktiolla tehtävän testauksen avulla EJB-mallin käyttötapojen suorituskykyeroja. Tarkoituksena on käyttää sellaisia tietomääriä, joiden testaamisesta saadaan riittävästi aineistoa kuljettamiseen kuluvan ajan lisääntymisen jatkuvuuden havaitsemiseen. Toisin sanoen aineistoa on saatava riittävästi sellaisen kaavan muodostamiseksi, jonka mukaisesti suoritusaika kullakin testaustavalla kasvaa kuljetettavia tietomääriä lisättäessä. EJB-komponenttimallin viimeisin versio on 2.0, joka sisältää kokonaisuudessaan version 1.1 määrityksen ja lisäksi mittavan määrän uusia ominaisuuksia. Tutkimuksen kohteena ovat lähinnä version 1.1 määrityksessä esitellyt ominaisuudet. Tutkielman lopussa otetaan kantaa myös joihinkin version 2.0 tuomiin ominaisuuksiin ja arvioidaan niiden merkitystä tässä tutkimuksessa saaduille tuloksille.

EJB-komponenttimalli tarjoaa ohjelmistosuunnittelijan käyttöön eri tyyppisiä komponentteja, ja niitä voidaan käyttää monin eri tavoin. Tutkimuksen keskeisimpänä mielenkiinnon kohteena on kaksi yleisimmin käytettyä EJB-komponenttien käyttötapaa siirrettäessä tietoa tietokantaan. Toinen näistä käyttötavoista tuo mukanaan ominaisuuksia, jotka tietyissä tilanteissa ovat hyödyllisiä ja joskus jopa välttämättömiä. Toinen tapa puolestaan on suoraviivaisempi ja siten myös tehokkaampi. Tutkimuksen motiivina ovat yleisesti havaitut tehokkuusongelmat ensin maitussa EJB-komponenttien käyttötavassa etenkin siirrettäessä suuria määriä tietoa tietokantaan. Tutkimuksen tarkoituksena on selvittää, miten nämä suorituskykyjen erot muodostuvat siirryttäessä yhä suurempiin tietomääriin ja kuinka merkittäviä nämä erot ovat.

Tutkimus jakautuu kahteen osaan: konstruktiiviseen ja empiiriseen. Konstruktio-osassa suunnitellaan ja toteutetaan konstruktio, jossa edellä kuvattujen suorituskykyerojen testaaminen on mahdollista. Siinä tavoitteena on aikaansaada järjestelmä, jossa asiakasohjelmasta käsin voidaan yhden EJB-komponentin tarjoaman rajapinnan kautta suorittaa tiedon vienti tietokantaan kaikilla testauksen viitekehyksessä määriteltävillä tavoilla. Luvussa kaksi testausohjelmiston määrittely alustetaan esittelemällä tutkielman kannalta keskeisimmät osat J2EE-ohjelmointialustan määrittelemästä EJB-komponenttimallista. Kolmannessa luvussa esitellään testauksen viitekehys, jossa määritellään komponentin käsite ja rajataan testausohjelmistossa käytettävät EJB-komponenttimallin ominaisuudet. Neljännessä luvussa esitellään kolmannen luvun viitekehyksen mukaisesti toteutetun testausohjelmiston arkkitehtuuri sekä komponenttikohtaisesti ne piirteet, jotka ovat tutkielman kannalta kiinnostavia. Tutkielman toinen, empiirinen osa käsittelee testausohjelmistolla tehdystä testauksesta saatuja tuloksia ja niiden analyysia. Empiirinen osa alkaa viidennestä luvusta, jossa kuvataan testaustapahtumat, niissä tehdyt havainnot sekä esitellään testin tulokset. Kuudennessa luvussa määritellään tulosten analyysin ja vertailun viitekehykset, analysoidaan testin tulokset ja esitellään analyysistä tehdyt johtopäätökset.

Tutkimuksen keskeisimpinä tuotoksina ovat EJB-pohjaisen testausohjelmiston rakentaminen sekä analyysi edellä kuvattujen EJB-komponenttien käyttötapojen suorituskykyeroista. Testausohjelmiston rakenteen lähtökohtana on järjestelmä, jossa

energiamittareista etäluettua mittaustietoa siirretään tietokantaan. Tämä käytännön lähtökohta testausohjelmistolle näkyy ohjelmiston sisäisissä sekä testausmateriaalin ja tietokannan nimeämiskäytännöissä. Lähtökohta ja sen mukanaan tuomat ominaisuudet ja ratkaisut testausohjelmistossa pyritään ratkaisemaan siten, etteivät ne vaikuta testauksen tuloksiin. Testausohjelmisto rakennetaan siten, että se on siirrettävissä ohjelmakoodia muuttamatta mihin tahansa EJB-määrityksen mukaiseen ympäristöön. Testaus ja tulosten analyysi puolestaan koskevat ainoastaan Oracle Containers for J2EE (OC4J)-palvelinohjelmistoa, joten empiirinen osuus sisältää vain tapauskohtaista materiaalia eivätkä siinä esitellyt tulokset ole sellaisenaan siirrettävissä johonkin toiseen ympäristöön. Toimintaperiaatteiltaan eri palvelinohjelmistot ovat kuitenkin hyvin samankaltaisia, joten testauksesta saatuja tuloksia voitaneen pitää ainakin suuntaa-antavina tarkasteltaessa jotain toista EJB-määrityksen mukaista palvelinohjelmistoa. Empiirisessä osiossa käytettyjen tilastomenetelmien valintaan on saatu konsultointiapua psykologian tohtori Joni Kettuselta.

2 EJB-KOMPONENTTIMALLI

Java 2 Enterprise Edition (J2EE) on Sun Microsystemsin määrittelemä ohjelmointialusta, joka sisältää kaikki hajautettujen liiketoimintaa tukevien järjestelmien toteuttamiseen tarvittavat sovellusliittymät. Enterprise JavaBeans (EJB)-komponenttimalli on suurin ja tämän tutkielman kannalta keskeisin J2EE:n sisältämistä määrityksistä. EJB-komponenttimallin määrityksistä on julkaistu versiot 1.0, 1.1 ja 2.0, joissa merkittävimmät muutokset ovat laajennoksia edellisiin versioihin nähden. Version 2.0 tuomassa laajennoksessa merkittävimmät muutokset liittyvät asynkroniseen viestinvälitykseen sekä uuteen kyselysyntaksiin (EJB Query Language), eivätkä ne ole tämän tutkielman kannalta kiinnostavia. Ainoastaan paikallisten rajapintojen tuki liittyy välillisesti tämän tutkielman aihepiiriin ja sekin vain lopussa tehtävän tulosten arvioinnin osalta. Tutkielman testauksessa käytetty versio palvelinohjelmistosta ei vielä tue paikallisia rajapintoja, joten tämän ominaisuuden vaikutukset tutkielman testituloksiin jää arvailujen varaan. Näin ollen tämän tutkielman kannalta keskeisin EJB-määrityksen versio on 1.1, vaikka lähdeaineistona käytetäänkin version 2.0 määritystä. (DeMichiel ym. 2001, 29-31)

Tässä luvussa esitellään yllä mainittu EJB-komponenttimallin määritys niiltä osin kuin se tämän tutkielman kannalta on mielekästä. EJB-komponenttimallin tarkoituksen on määrittää malli, jonka avulla voidaan toteuttaa hajautettuja järjestelmiä uudelleenkäytettäviä ja siirrettäviä komponentteja käyttäen (Vainio 2000, 45). Mallin esittelyssä pyritään keskittymään niihin piirteisiin, jotka vaikuttavat tutkielman konstruktio-osuudessa tehtäviin valintoihin ja ratkaisuihin. Erityisesti kiinnitetään huomiota siihen, että konstruktiossa tapahtuvan testauksen tulokset ovat keskenään vertailukelpoisia.

EJB-komponenttimallin määrityksessä esitellään kuusi erilaista roolia. Nämä roolit edustavat niitä näkökulmia, joista määrityksen aihealuetta lähestytään. Tuottajalla (Bean Provider) tarkoitetaan EJB-komponenttien toteuttajaa. Hän on vastuussa komponenttien ominaisuuksista, liiketoimintalogiikan toteuttamisesta sekä komponenttien yhteyksistä muihin komponentteihin ja palveluihin. Kokoaja (Assembler) on vastuussa komponenttien yhteenliittämisestä. Tässä lopputuloksena

on toimiva järjestelmä. Sijoittaja (Deployer) vastaa EJB-komponenttien sijoittamisesta suoritusympäristöönsä. Palvelimen tuottaja (Server Provider) ja säiliön tuottaja (Container provider) ajatellaan olevan yksi toimija, koska määritys ei määrittele palvelimen ja säiliön välistä rajapintaa. Säiliön tuottajan vastuulla on toteuttaa EJB-komponettien sijoitustyökalut sekä ajonaikainen ympäristö niiden hallintaan. Palvelimen tuottajan vastuulla on ympäristö, jossa säiliö ja siihen liittyvät työkalut toimivat. Kuudes rooli on varattu ylläpitäjälle (System Administrator), jonka vastuulla on huolehtia järjestelmän konfiguroinnista sekä ajonaikaisesta toiminnasta. (DeMichiel ym. 2001, 33-36.)

Yllä kuvattujen roolien kautta EJB-komponenttimallin määrittelyssä määritellään EJB-tyypit, tapahtumanhallinta, poikkeusten käsittely, hajautustuki, EJB-ympäristö, tietoturvan hallinta, komponenttien sijoittamisen työkalut sekä ajonaikainen ympäristö. EJB-tyyppejä ovat istuntopohjainen (session), kohdepohjainen (entity) ja viestipohjainen (message-driven) komponentti (DeMichiel ym. 2001, 47). Istuntopohjainen ja kohdepohjainen komponenttityyppi esitellään kohdassa 2.2. Viestipohjainen komponentti liittyy asynkroniseen viestin välitykseen, eikä se ole tämän tutkielman kannalta kiinnostava. Tämän tutkielman puitteissa tapahtuva tietokannan käsittely kohdistuu yhteen tietokantatauluun ja käsittelyä tapahtuu yhdestä asiakkaasta kerrallaan. Tapahtumanhallinnan palveluita käytetään testausohjelmistossa siten, että jokainen testaustapahtuma on oma tapahtumansa. Siksi EJB-komponenttimallin määritystä tapahtumanhallinnasta ei tässä käsitellä. Poikkeusten hallinnan määritys sisältää hyvin yksityiskohtaisen selvityksen poikkeusten käsittelystä eri EJB-mallin tasoilla (DeMichiel ym. 2001, 371). Lyhyehkö katsaus poikkeusten hallinnan määritykseen esitetään kohdassa 2.3. Hajautuksen arkkitehtuurin ja toiminnallisuuden määritelmä kuvaa hajautukseen liittyvien modulien yhteistoimintaa sekä vaatimuksia (DeMichiel ym. 2001, 385) ja se esitellään kohdassa 2.1. EJB-ympäristön määritelmä määrittelee rajapinnat, joiden kautta ympäristöön ollaan yhteydessä (DeMichiel ym. 2001, 409). Näiden rajapintojen määritelmä esitellään kohdassa 2.1. Tutkielman puitteissa rakennettava konstruktio on yhden käyttäjän järjestelmä, joten tietoturvan määritelmän esitteleminen on tässä turhaa. Komponenttien sijoittamiseen kohdistuvat vaatimukset ja sijoittamisen työkalut esitellään kohdassa 2.3. Ajonaikaisen ympäristön määritelmä esitellään kohdassa 2.1. (DeMichiel ym. 2001, 33-36.)

2.1 Suoritusympäristö

Java-virtuaalikone (Java Virtual Machine (JVM)) on eräänlainen abstrakti tietokone, jossa Java-ohjelmia ajetaan. Se on Java-teknologian keskeisimpiä, ellei keskeisin kulmakivi, joka mahdollistaa Java-ohjelmien laitteisto- ja käyttöjärjestelmäriippumattomuuden. Tässä tutkielmassa käytetään Java virtuaalikoneesta lyhennettä JVM. EJB-ympäristöllä puolestaan tarkoitetaan EJB-komponenttimallin toteuttavaa palvelinohjelmistoa.

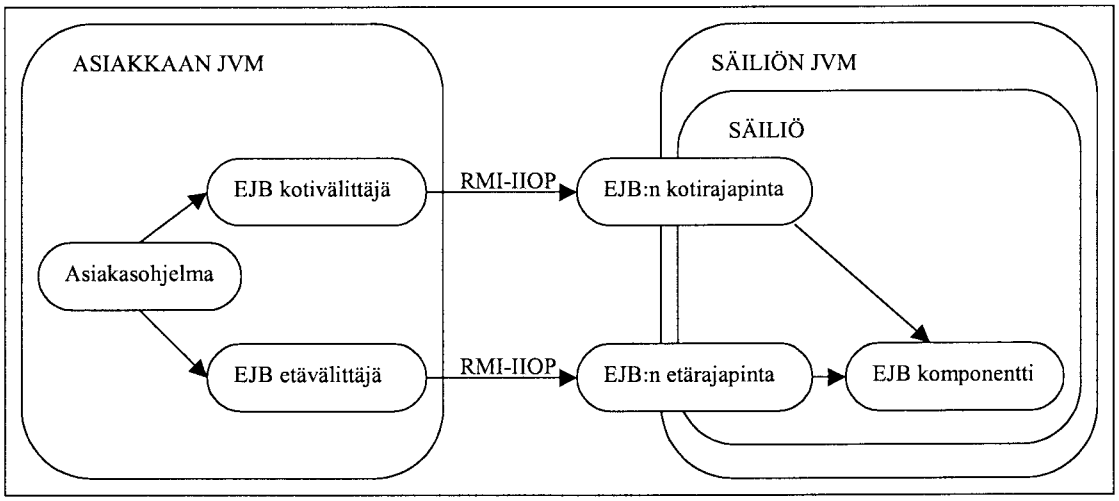
2.1.1 Palvelimen ja säiliön tehtävät

EJB-komponenttimalli ei määrittele säiliön ja palvelimen välistä rajapintaa. Näin ollen on vaikea päätellä, mitkä tehtävät kuuluvat säiliölle ja mitkä palvelimelle. Edelleen tämä on johtanut siihen, että säiliön ja palvelimen tuottaa käytännössä sama taho. Tästä johtuen palvelimen ja säiliön tehtävät esitellään tässä ikäänkuin palvelin ja säiliö olisivat yksi komponentti. Termejä palvelin ja säiliö käytetään tässä varsin vapaasti siten, että eri tehtävien yhteydessä käytetään aina sitä termiä mille kyseisen tehtävän parhaiten arvellaan sopivan. (Monson-Haefel 2001, 49-50.)

Säiliöt tarjoavat suoritusympäristön sisältämilleen komponenteille. Niiden tehtävänä on huolehtia resurssien hallinnasta ja välittää viestejä eri komponenttien välillä. Säiliö ei voi koskaan olla suoraan metodikutsun kohteena. Sen sijaan se tarjoaa erilaisia sovellusliittymiä komponenttien ja asiakassovelluksen käyttöön (ks. alakohta 2.1.4). Säiliön tuottajan tulee tarjota sijoitustyökalut komponenttien sijoittamiseen (deployment) sekä toteuttaa ajonaikainen ympäristö (Runtime Environment) (Vainio 2001, 48). Lisäksi palvelimen tehtävänä on huolehtia käyttöoikeuksista sekä komponenttien pysyvyyden ja tilan hallinnasta. Palvelimen tehtävänä on huolehtia matalan tason palveluista, kuten hajautuksen protokollasta (ks. alakohta 2.1.2) (Monson-Haefel 2001, 67-77).

2.1.2 Hajautus

EJB-komponenttimallin määrityksen mukaisesti EJB:itä voidaan käyttää hajautetusti tietoverkon ylitse. Tällöin EJB:itä käyttävä asiakasohjelma voi sijaita fyysisesti eri koneessa kuin EJB. EJB-mallin versiossa 1.1 tämä on myöskin ainut tapa käyttää EJB:tä. Versiossa 2.0 on esitelty myös paikallinen rajapinta EJB-komponenteille. Hajautuksen perustana on Java Remote Method Invocation (RMI), joka käyttää Internet InterORB Protocol:aa (IIOP). EJB-komponentin kotirajapinnan (home interface) ja etärajapinnan (remote interface) tulee molempien olla määritelty Java RMI rajapinnoiksi. Tämä tarkoittaa sitä, että yllä mainittujen rajapintojen tulee suoraan tai välillisesti vähintään toteuttaa (implements) *java.rmi*-paketissa määritelty *Remote*-rajapinta. *Remote*-rajapinta määrittelee sellaiset metodit, joiden avulla rajapinnan toteutavaa oliota päästään käsittelemään muualla sijaitsevasta Java Virtual Machine:sta (JVM) (Sun 2001, 6). Tällöin säiliön tuottaja voi käyttää RMI-IIOP:ta olioiden hajautusprotokollana (distribution protocol). Ollessaan yhteydessä EJB-komponenttiin, asiakasohjelma käyttää välittäjiä (stub). Asiakasohjelman ja EJB-komponentin kommunikoinnin arkkitehtuuria sekä keskeisiä olioita on esitelty kuvassa 2.1. (DeMichiel ym. 2001, 385-386.)



Kuva 2.1: Asiakasohjelman ja EJB-komponentin välinen kommunikointi

(mukailtu lähteestä DeMichiel ym. 2001, 386).

Kuvassa 2.1 esitellään asiakassovelluksen ja EJB-komponentin välinen kommunikointi yleisellä tasolla. Jokaista etärajapinnan toteuttavaa EJB-komponenttia

kohti toteutetaan EJB:n kotirajapinta (EJB Home Interface). Sen metodien kautta voidaan luoda, etsiä ja tuhota EJB-komponentteja. EJB koti välittäjä (EJB Home Stub) toimii vastakappaleena asiakassovelluksessa EJB:n kotirajapinnalle ja se toteuttaa myös samat metodit. EJB kotivälittäjä ei kuitenkaan toteuta näiden metodien sovelluslogiikkaa, vaan se ainoastaan välittää metodikutsut RMI-IIOP-protokollaa käyttäen EJB-kotirajapinnalle. EJB etävälittäjän ja EJB:n etärajapinnan keskinäinen suhde on samanlainen kuin EJB kotivälittäjän ja EJB:n kotirajapinnan. EJB:n etärajapinnan on toteutettava samat metodit kuin EJB-komponentin ja sen tehtävänä on välittää asiakassovelluksesta tehdyt metodikutsut EJB-komponentille. EJB-komponenttimallin versio 2.0 määrittelee myös paikalliset koti- ja etärajapintoja vastaavat rajapinnat. Niiden tehtävänä on palvella EJB-komponentin paikallista, palvelimen sisäistä käyttöä. Tässä tutkielmassa paikallisia rajapintoja ei käytetä, joten niitä ei myöskään tässä tarkemmin esitellä. (DeMichiel ym. 2001, 58-59 ja 194.)

Edellisen kappaleen ylimalkaisen katsauksen yleiseen EJB-komponenttien toimintaperiaatteeseen on tarkoitus toimia johdantona tulevissa alakohdissa esiteltäviin asioihin. Myöhemmin keskitytään lähinnä EJB-komponentin määrityksen esittelyyn ja edellä mainittujen rajapintojen määrityksiin viitataan ainoastaan niiltä osin, kuin niillä on EJB-komponentteja käyttävän ohjelmiston suorituskyvyn kannalta merkitystä.

2.1.3 Rajapinnat

EJB-komponenttimalli määrittelee neljä rajapintaa, joiden avulla päästään käsiksi EJB-ympäristöön. Java Naming and Directory Interface (JNDI) tarjoaa rajapinnan, jonka avulla voidaan käsitellä ympäristöön sijoitettujen EJB-komponenttien liiketoimintalogiikkaa tarvitsematta koskea niiden lähdekoodiin. JNDI tarjoaa myös rajapinnan, jonka avulla voidaan luoda yhteys EJB-komponentin kotirajapintaan. Näistä JNDI:n rajapinnoista ensin mainittu ei ole tutkielman kannalta kiinnostava, koska EJB:iden liiketoimintalogiikan muuttaminen ei kohdistu EJB-komponenttien olioiden ilmentymiin vaan niitä ohjaaviin ympäristömuuttujiin. Näin ollen se ei vaikuta suoranaisesti EJB-komponenttien suorituskykyyn ajonaikaisessa tilanteessa. Kolmas määritelmä esittelee rajapinnan EJB-ympäristön resurssien hallintaan

(resource manager), jonne käyttäjä voi määritellä tietolähteitä (data source), esimerkiksi yhteyden tietokantaan. Neljäs määritelmä esittelee rajapinnan, jonka kautta voidaan käsitellä edellä mainittuihin tietolähteisiin liittyviä olioita. Kahden viimeksimainitun rajapinnan tarkempi käsittely ei tässä ole kovin kiinnostavaa, koska niilläkään ei ole vaikutusta EJB-komponenttien suorituskykyyn. (DeMichiel ym. 2001, 409-428.)

JNDI:n palvelu yhteyksien luomiseksi EJB-komponentin kotirajapintaan on kaksijakoinen. Komponentin ominaisuudet on määriteltävä sijoituksen kuvaustiedostoon (deployment descriptor) sijoittamista varten (ks. alakohta 2.3.1). Kun sijoittaminen on onnistuneesti suoritettu, voidaan JNDI:n tarjoamaa rajapintaa käyttäen luoda yhteys määrityksen mukaisen EJB-komponentin kotirajapintaan. Yhteys voidaan luoda sekä asiakasohjelmasta että toisesta EJB-komponentista. Yhteyden luonti tapahtuu JNDI:n konteksti-rajapinnan avulla, jonka oletusluokka on *javax.naming.InitialContext*. Luokan *lookup* –metodi palauttaa viitteen haluttuun objektiin säiliön sisällä. (DeMichiel ym. 2001, 415-420.)

2.1.4 Ajonaikainen ympäristö

EJB-komponenttimallin ajonaikaisen ympäristön määritys esittelee joukon sovellusliittymiä (Application Programming Interface (API)), jotka on totetutettava hyväksyttävän EJB-ympäristön aikaansaamiseksi. Lisäksi se ottaa kantaa palautusarvon välityksen semantiikkaan. Tässä esiteltävät API:t ja niiden versiot vastaavat nimenomaan EJB:n version 2.0 vaatimuksia. Tutkielmassa käytettävän ympäristön mahdolliset poikkeukset tässä esiteltyihin API:hin esitellään neljännessä luvussa.

EJB-mallin version 2.0 mukaisen säiliön on toteutettava täysi kokoelma Java 2 Standard Edition (J2SE) version 1.3 API:ja. Tosin säiliön tuottajalle on annettu vapaus rajata pois tiettyjä toimintoja, jotka vaikuttavat tietoturvan hallintaan (ks. taulukko 19, DeMichiel 2001, 498). EJB 2.0 API on luonnollisesti toteutettava säiliössä. JNDI 1.2 API:sta on vähintään toteutettava EJB-komponenteille varattu nimiavaruus (name space). Nimiavaruuden on oltava käytettävissä asiakkaalle kun

tämä kutsuu *javax.naming.InitialContext* –luokan oletuskonstruktoria ja siitä on vähintään toteutettava resurssien hallintaa sekä EJB-komponenttien kotirajapintaa koskevat osat. Säiliöön sijoitettuja EJB-komponentteja ei välttämättä tarvitse esitellä samassa JNDI:n nimiavaruudessa, mutta yhden komponentin ilmentymät tarvitsee. Java Transaction API:sta (JTA) on toteutettava laajennos versio 1.0.1. Sen täytyy sisältää *javax.transaction.UserTransaction* –rajapinta, johon pääsee käsiksi *javax.ejb.EJBContext* –rajapinnan kautta. Tämä rajapinta on tarkoitettu EJB-komponentin itsensä ohjaamaan tapahtumanhallintaan. JDBC 2.0 API:sta on säiliön toteutettava EJB-komponenttien kaikki muu toiminnallisuus, paitsi alatason XA- ja yhteyksien hallintarajapinnat (connection pooling). Java Messaging Service (JMS) API:sta säiliön on toteutettava osa laajennoksen 1.0.2 toiminnallisuudesta EJB-komponenttien käyttöön. EJB-komponentin koti- ja etärajapinnat toteuttavat Java RMI:n *java.rmi.Remote* –rajapinnan (ks. alakohta 2.1.2). Säiliön tehtävänä on varmistaa, että kaikki viitetyyppisenä palautettavat objektit toteuttavat tämän rajapinnan. Kaikki muut objektit on palautettava arvotyyppisenä (pass by value) ja niiden on joko oltava perustietotyyppejä tai niiden on toteutettava *java.io.Serializable* –rajapinta, jotta ne voidaan siirtää Java RMI:n välittämänä. (DeMichiel 2001, 496-500 ja Monson-Haefel 2001, 120-121.)

2.2 EJB-komponentit

EJB-komponenttimalli määrittelee kolme eri tyyppistä EJB-komponenttia. Viestipohjainen EJB-komponentti liittyy asynkroniseen viestin välitykseen ja sitä käytetään normaalisti Java Messaging Service (JMS) –rajapinnan kautta. Istuntopohjainen EJB-komponentti edustaa jotain rajattua liiketoimintaprosessia ja sisältää siihen liittyvän liiketoimintalogiikan, -säännöt sekä tietovuon. Kohdepohjainen EJB-komponentti edustaa pysyvää tietoa, esimerkiksi tietokannan riviä. Asynkroninen viestin välitys ei ole tutkielman kannalta kiinnostava, joten viestipohjaisen komponentin määritys jätetään tässä esittelemättä. Istuntopohjaisen ja kohdepohjaisen määritykset esitellään alla siinä laajudessaan kun se tutkielman kannalta on mielekästä.

2.2.1 Istuntopohjainen komponentti

Istuntopohjainen EJB-komponentti toteuttaa aina *javax.ejb.SessionBean* –rajapinnan, jonka metodien avulla säiliö kommunikoi komponentin kanssa. Myös asiakassovellus käyttää välillisesti osaa näistä metodeista istuntopohjaisen EJB-komponentin luonnin, aktivoinnin tai tuhoamisen yhteydessä. *javax.ejb.SessionBean* –rajapinta on kuvattu taulukossa 2.1.

Rajapinnan metodi	**Metodin kuvaus**
ejbRemove( )	Metodin välityksellä ilmoitetaan komponentin ilmentymän poistamisesta säiliöstä.
ejbPassivate( )	Metodin välityksellä ilmoitetaan, että komponenttia ollaan siirtämässä passiiviseen tilaan.
ejbActivate( )	Metodin välityksellä ilmoitetaan, että komponenttia ollaan siirtämässä aktiiviseen tilaan.
setSessionContext(SessionContext ctx)	Komponentille välitetään tieto omasta kontekstistaan (context) luonnin yhteydessä tällä metodilla.

Taulukko 2.1: *javax.ejb.SessionBean* -rajapinnan metodit

(DeMichiel ym. 2001, 73).

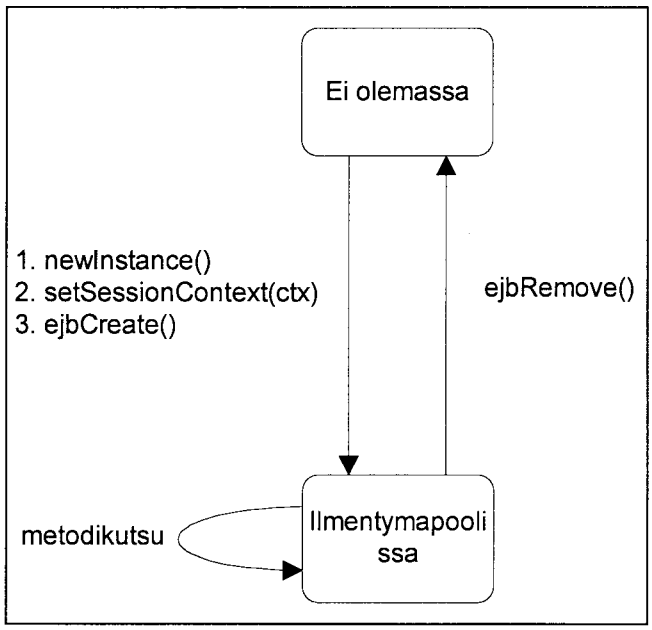
Taulukossa 2.1 esitellään *javax.ejb.SessionBean* –rajapinnan metodit. Metodien esittely jätetään tässä sille tasolle, kuin ne on taulukossa esitelty. Niiden aiheuttamia tilasiirtymiä käsitellään tarkemmin kuvien 2.2 ja 2.3 yhteydessä.

Istuntopohjaista EJB-komponenttia käytetään jonkin rajatun toiminnon toteuttamiseen. Tyypillinen tehtävä EJB-komponentilla voisi olla esimerkiksi XML-dokumentin purkaminen, koron laskeminen tms. toimenpide, joka suoritetaan aina jonkin tietyn asiakassovelluksen puolesta. Istuntopohjaisen komponentin lyhin mahdollinen elinikä muodostuu näin asiakassovelluksen vaatiman toiminnan pituuden mukaan.

Istuntopohjaiset EJB-komponentit jaetaan kahteen alityyppiin. Tilallinen istuntopohjainen EJB-komponentti edustaa ikäänkuin asiakassovelluksen laajennosta tarjoten tilasiirtymät palvelun suorittamisen edetessä. Tilallisesta komponentista puhuttaessa tarkoitetaan tässä tutkielmassa aina istuntopohjaista tilallista EJB-komponenttia. Tilaton istuntopohjainen EJB-komponentti on tarkoitettu hyvin rajatun yksittäisen toiminnon suorittamiseen, eikä se sisällä mitään asiakkaalle näkyvää tilaa. Tilattomasta komponentista puhuttaessa tarkoitetaan tässä tutkielmassa aina istuntopohjaista tilatonta EJB-komponenttia.

2.2.1.1 Tilaton istuntopohjainen komponentti

Tilaton istuntopohjainen EJB-komponentti on tutkielmassa esitetyistä komponenttityypeistä tehokkain. Sen asettamat vaatimukset palvelimen resursseille jäävät vähäisiksi, koska ne eivät ole pysyviä eikä niitä ole sidottu mihinkään tiettyyn asiakassovellukseen. Koska tilaton komponentti ei ylläpidä tilaansa, vapautuu sen resurssit seuraavan asiakkaan käyttöön välittömästi yhden metodikutsun aiheuttaman toiminnon suorittamisen jälkeen. Tämä tekee tilattomista istuntopohjaisista komponenteista kevytrakenteisia ja verrattain nopeita. Tyypillinen tilattoman komponentin tarjoama toiminto on rajattu, yleinen ja uudelleenkäytettävä. Rajauksen yhtenä vaatimuksena on, että toiminto ei saa olla riippuvainen muista komponenteista. Toisin sanoen tilattoman komponentin toimintaan tarvittava tieto on kyettävä välittämään metodin parametreissä. Poikkeuksena tähän on *SessionContext*:in välittäminen komponentille (ks. Taulukko 2.1). Tästä aiheutuu huomionarvoinen rajoitus tilattoman komponentin käytölle. Tilaton komponentti ei saa tallettaa toimintaa suorittaessaan mitään sellaista tietoa, joka vaikuttaisi seuraavan toiminnan suoritukseen. Tämä sen vuoksi, että eri asiakassovellusten kutsujen yhteydessä ei voida ennustaa, mille komponentin ilmentymälle kutsu välitetään. Tämä ei kuitenkaan sulje pois mahdollisuutta esimerkiksi lokitiedoston ja sitä kautta eräänlaisen sisäisen tilatiedon ylläpitämiseen. Tämänkaltainen tilatieto ei kuitenkaan saa näkyä asiakassovellukseen. (Monson-Haefel 2000, 220-221.)



Kuva 2.2: Tilattoman istuntopohjaisen komponentin elinkaari.

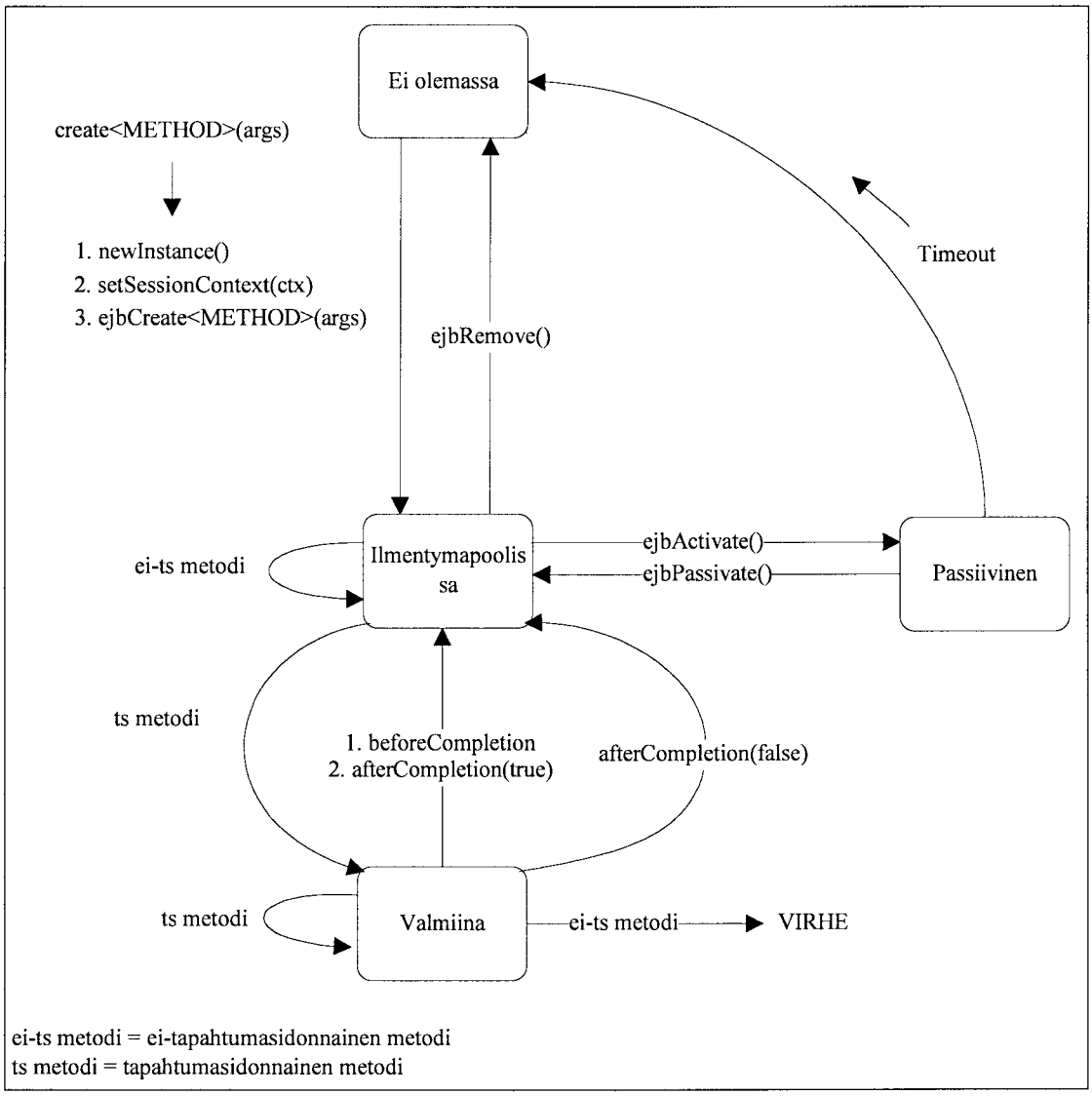
(Mukaellen lähteestä DeMichiel ym. 2001, 89.)

Kuva 2.2 esittää tilattoman komponentin elinkaarta eri kutsujen yhteydessä. Komponentin luominen ja tuhoaminen ovat täysin säiliön hallinnoimia. Asiakassovelluksen tekemät metodikutsut aiheuttavat aina komponentin palautumisen samaan "ilmentymäpoolissa" –tilaan (Method-Ready Pool). Asiakassovelluksen pyytäessä viitettä tilattoman komponentin etärajapintaan säiliö päättää luodaanko komponentista uusi ilmentymä vai käytetäänkö poolissa vapaana olevaa valmista ilmentymää. Säiliön luodessa tilattoman komponentin ilmentymää se kutsuu *ejbCreate-* ja *setSessionContext* –metodeja. *ejbRemove* –metodi puolestaan tuhoaa komponentin ilmentymän ja vapauttaa sen käyttämän muistitilan. Tilattoman komponentin tapauksessa *ejbActivate-* ja *ejbPassivate* –metodit eivät sisällä mitään toiminnallisuutta. (Vainio 2001, 51-54.)

2.2.1.2 Tilallinen istuntopohjainen komponentti

Tilallinen istuntopohjainen EJB-komponentti tarjoaa vaihtoehdon tilattoman komponentin ja kohdepohjaisen komponentin välimaastosta. Tilallinen komponentti on sidottu johonkin tiettyyn asiakassovellukseen koko elikaarensa ajaksi. Tarjoamansa toiminnon suorituksessa tilallinen komponentti voi tallettaa tietoa, joka vaikuttaa

toimintoihin metodikutsujen välillä. Nämä tiedot voidaan tallettaa väliaikaisesti pysyvään tietovarastoon passivoinnin yhteydessä (ks. Taulukko 2.1). Tilallinen komponentti ei kuitenkaan suoraan edusta tietoa pysyvässä tietovarastossa kuten kohdepohjainen komponentti. Palvelimen tarjoajalla on mahdollisuus passivoinnin yhteydessä vapauttaa komponentin käyttämä muistitila. Aktivoinnin seurauksena komponentin tila on kuitenkin palautettava ja uuden kutsun yhteydessä tiedetään ainakin käsitteellisesti, mille komponentin ilmentymälle kutsu välitetään. Näin ollen komponentin tila on ennustettavissa kutsusta toiseen. (Monson-Haefel 2000, 242-243.)



Kuva 2.3: Tilallisen istuntopohjaisen komponentin elinkaari.

(Mukaellen lähteestä DeMichiel ym. 2001, 77.)

Kuva 2.3 esittää tilallisen komponentin elinkaaren sekä tilasiirtymät eri metodikutsujen yhteydessä. Tässä yhteydessä on tarpeen erottaa kaksi metodityyppiä. Ei-tapahtumasidonnaisilla (Non-Transaction Context Method) metodeilla tarkoitetaan *javax.ejb.SessionBean* –rajapinnan metodeja. Niitä kutsutaan säiliön toimesta. Tapahtumasidonnaiset metodit (Transaction Context Method) puolestaan ovat sidoksissa komponentin tarjoamaan palveluun ja sisältävät varsinaisen liiketoimintalogiikan. Näiden metodien kutsut välitetään aina asiakassovelluksesta. Siirtyminen "ei olemassa" –tilasta "ilmentymäpoolissa" -tilaan tapahtuu tilattoman komponentin tavoin kuitenkin siten, että komponentista luodaan aina uusi ilmentymä. Siirtyminen "ilmentymäpoolissa" –tilasta "valmiina" –tilaan tapahtuu aina tapahtumasidonnaista metodia kutsumalla. "Valmiina" –tilassa ei-tapahtumasidonnaisen metodin kutsu aiheuttaa virheen ja tapahtumasidonnaisen metodin kutsu tilasiirtymän "valmiina" –tilaan tai "ilmentymäpoolissa" –tilaan. "Ilmentymäpoolissa" –tilasta siirtyminen "passiivinen" –tilaan tapahtuu ei-tapahtumasidonnaista *ejbPassivate* –metodia kutsumalla ja paluu "ilmentymäpoolissa" –tilaan *ejbActivate* –metodilla. "Passiivinen" –tilasta on myös mahdollista siirtyä "ei olemassa" –tilaan tietyn aikarajan ylityttyä. (DeMichiel 2001, 76-77.)

2.2.2 Kohdepohjainen komponentti

Kohdepohjainen komponentti edustaa EJB-komponenttimallissa pysyvässä tietovarastossa, normaalisti tietokannassa olevaa tietoa. Muutokset kohdepohjaiseen komponenttiin aiheuttavat siten myös muutoksen tietovarastoon. Kohdepohjaisen komponentin tarkoituksena on helpottaa tietovarastossa olevan tiedon muokkausta. Lisäksi tiedon esittäminen omissa komponenteissaan, sen sijaan että käyttäisi esimerkiksi liiketoimintakomponenteissa tietorakenteita, edesauttaa komponenttien uudelleenkäytettävyyttä. Luotaessa uutta kohdepohjaisen komponentin ilmentymää, luodaan samalla tietovarastoon uusi tietue. Komponentin tilan muuttuessa, siirtyy muutos tietovaraston tietueeseen synkronisesti. Tämä tietovaraston ja komponentin synkronisointi antaa kohdepohjaiselle komponentille ominaisuuden, jota kutsutaan pysyvyydeksi (persistence). Pysyvyyden hallintaan on EJB-komponenttimallissa kaksi tapaa. Näiden tapojen mukaisesti erotetaan myös kaksi pysyvyyden muotoa.

Säiliön hallinnoimalla pysyvyydellä (Container-Managed Persistence) tarkoitetaan kohdepohjaisen komponentin ominaisuutta, jossa synkronisointi komponentin ja tietokannan tietojen välillä hoituu automaattisesti säiliön toimesta. Samalla säiliö voi huolehtia myös tapahtumanhallinnasta. Komponentin hallinnoimassa pysyvyydessä (Bean-Managed Persistence) synkronisointi sekä tapahtumanhallinta tapahtuu komponentin sisällä. (Monson-Haefel 2000, 153-154.)

Kohdepohjaisen komponentin tiedon lähteenä voi olla jokin muukin tietovarasto kuin tietokanta (Vainio 2001, 55). Tutkimuksen kohteena on kuitenkin tiedon siirtäminen tietokantaan, joten tässä tutkielmassa tullaan jatkossa käyttämään termiä tietokanta viitattaessa komponenttien taustalla olevaan tietovarastoon. Edelleen jokainen kohdepohjainen komponentti edustaa tässä tutkielmassa aina yhtä tietokannan taulun riviä. Komponenttia, jonka pysyvyyden hallinnasta huolehtii säiliö, kutsutaan tässä tutkielmassa CMP-komponentiksi ja vastaavasti komponenttia, jonka pysyvyyden hallinnasta huolehtii komponentti itse, kutsutaan BMP-komponentiksi. Nämä lyhenteet tulevat edellisessä kappaleessa mainituista pysyvyyden muotojen englanninkielisistä termeistä.

Alakohdan 2.2.1 alussa mainittiin, että istuntopohjaisen komponentin on aina toteutettava *javax.ejb.SessionBean* rajapinta. Samalla tavalla kohdepohjaisen komponentin on toteutettava *javax.ejb.EntityBean* rajapinta kommunikoidakseen säiliön ja asiakassovelluksen kanssa. Tämä rajapinta on kuvattu taulukossa 2.2.

Rajapinnan metodi	**Metodin kuvaus**
setEntityContext(EntityContext ctx)	Komponentille välitetään tieto omasta kontekstistaan (context) luonnin yhteydessä tällä metodilla.
unsetEntityContext( )	Metodi poistaa komponentin ilmentymän ilmentymäpoolista.
ejbRemove( )	Metodi passivoi komponentin siirtämällä sen ilmentymäpoolissa-tilaan. Samalla se poistaa komponentin entiteetin ja tekee peruutuksen (roll back) mahdolliselle vahvistamattomalle tietovaraston käsittelylle kyseisen komponentin osalta.
ejbActivate( )	Metodi aktivoi komponentin siirtämällä sen valmiina-

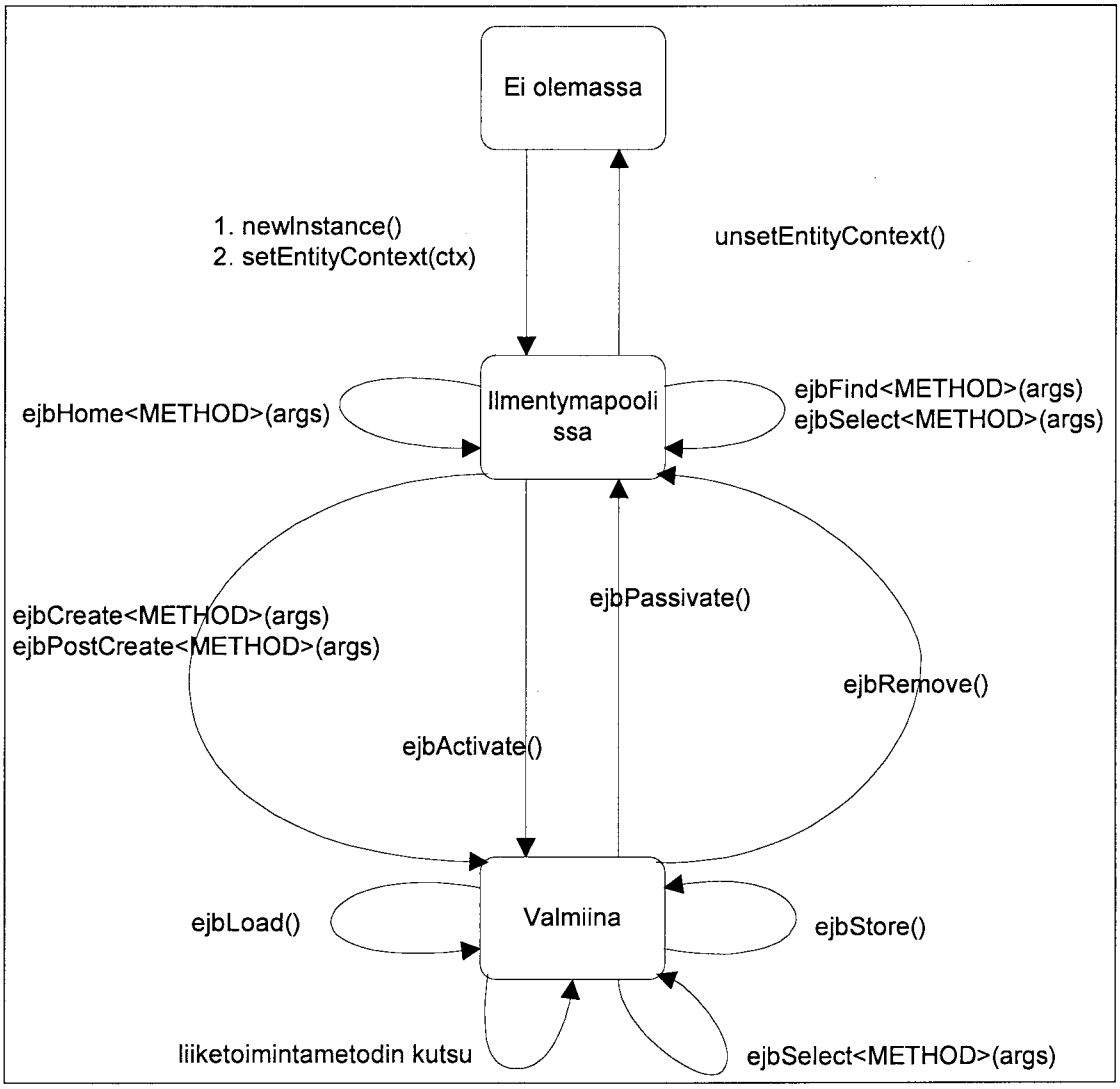
Rajapinnan metodi	Metodin kuvaus
	tilaan
ejbPassivate( )	Metodi passivoi komponentin siirtämällä sen ilmentymäpoolissa-tilaan.
ejbLoad( )	Metodin avulla säiliö synkronisoi komponentin tilan tietovaraston tietojen kanssa.
ejbStore( )	Metodin avulla säiliö synkronisoi tietovaraston tiedot komponentin tilan kanssa.

Taulukko 2.2: *javax.ejb.EntityBean* rajapinnan metodit.

(DeMichiel ym. 2001, 170-173).

Taulukossa 2.2 esitellään *javax.ejb.EntityBean* –rajapinnan metodit. Metodien esittely jätetään tässä sille tasolle, kuin ne on taulukossa esitelty. Niiden aiheuttamia tilasiirtymiä käsitellään tarkemmin kuvan 2.4 yhteydessä.

Kohdepohjaisen EJB-komponentin elinkaari seuraa samankaltaista tilakaaviota huolimatta siitä, onko kyseessä CMP- vai BMP-komponentti. Ainoastaan tilasiirtymiä on CMP-komponentilla hieman enemmän. Kuvassa 2.4 on esitelty kuvattu kohdepohjaisen EJB-komponentin ilmentymän elinkaaren mahdolliset tilamuutokset.



Kuva 2.4: Kohdepohjaisen EJB-komponentin elinkaari.

(Mukaellen lähteestä DeMichiel ym. 2001, 168.)

Kuvassa 2.4 on esitelty kohdepohjaiselle EJB-komponentille kolme mahdollista tilaa. "Ei olemassa" –tilassa komponenttia ei tilan nimen mukaisesti ole luotu. Komponentin luominen tapahtuu säiliön hallinnoimana (newInstance) ja samalla komponentille luovutetaan tieto omasta kontekstistaan (context) (setEntityContext). Tällöin siirrytään "ilmentymäpoolissa" –tilaan, jolloin komponentti on olemassa ilman tarkempaa identiteettiä eikä se esitä mitään tietokannan tietoa. Kun komponentille määritellään uusi identiteetti (ejbCreate) tai kun se aktivoidaan (ejbActivate), siirrytään valmiustilaan jossa komponentti on aktiivinen ja osoittaa jotain tietoa tietokannassa. Vastaavasti paluu "ilmentymäpoolissa" –tilaan tapahtuu ilmentymän passivoinnin (ejbPassivate) tai poistamisen (ejbRemove) yhteydessä. Huomattavaa

tässä on, että poistaminen ei poista itse komponentin ilmentymää, vaan sen esittämän tiedon tietokannasta. Paluu "ilmentymäpoolissa" –tilasta "ei olemassa" –tilaan taas tapahtuu konteksti tuhoamalla (unsetEntityContext), jolloin myös komponentin ilmentymä tuhotaan. Kuten edellä mainittiin, on tilasiirtymiä CMP-komponentissa hieman BMP-komponenttia enemmän. Kuvassa 2.4 esitetyt tilasiirtymät ejbSelect<METHOD>(args) tilasta "valmiina" samaan tilaan ja ejbSelect<METHOD>(args) tilasta "ilmentymäpoolissa" samaan tilaan ovat käytössä ainoastaan CMP-komponentin ollessa kyseessä. (DeMichiel ym. 2001, 168-170.)

Kohdepohjaisten komponenttien yhteydessä käytetään EJB-komponenttimallin mukaisesti avainkenttä (PrimaryKey)–luokkaa. Sen avulla voidaan luoda yksikäsitteisesti kohdepohjainen komponentti (DeMichiel ym. 2001, 120). Avainkenttä –luokkaa käytetään pääasiassa komponentin hakemiseen kotirajapinnan kautta ja komponenttien vertailuun, joten sitä ei tässä tarkemmin käsitellä.

2.3 Komponentit suoritusympäristössä

Toimivassa EJB-komponenttimallin mukaisessa suoritusympäristössä täytyy tietyt elementit olla kunnossa. Tarvitaan sovelluspalvelin, joka sisältää yhden tai useampia säiliöitä. Komponenttien on oltava sijoitettu (deploy) onnistuneesti säiliöön. Lisäksi tarvitaan asiakassovellus, joka käyttää sijoitettuja komponentteja. Komponenttien sijoittaminen ei toimenpiteenä vaikuta EJB-komponentteja käyttävän sovelluksen suorituskykyyn. Sijoittamisen kuvauksessa tyydytäänkin hyvin yleisluontoiseen lyhyeen esitykseen.

2.3.1 Sijoittaminen ympäristöön

EJB-komponenttimalli määrittelee sijoituksen kuvaustiedoston (Deployment Descriptor), joka toimii keskeisessä roolissa sijoitettaessa komponentteja suoritusympäristöön. Kuvaustiedoston sisältää tietoa komponenteista sekä niiden sijoittamisesta. Kuvaustiedoston määritys esittelee kaksi tiedon tyyppiä. Ensimmäinen tyyppi on pakollinen, komponentin rakennetta (Enterprise beans' structural

information) kuvaava tieto ja se esittelee komponentin rakenteen lisäksi tämän ulkoiset riippuvuudet sekä sen, miten komponenttia ajonaikaisesti hallitaan. Toinen tiedon tyyppi kuvaa sijoittamista (Application assembly information) laajemmasta näkökulmasta. Siinä voidaan esimerkiksi määritellä käyttöoikeuksia komponentille ja riippuvuuksia komponenttien välillä. (DiMichiel ym. 2001, 455-456.)

2.3.2 Asiakassovelluksen näkökulma

Asiakassovellus ei koskaan käsittele EJB-komponentteja suoraan. Sen sijaan käytetään rajapintaluokkia, joiden välityksellä päästään käyttämään komponenttien liiketoimintalogiikkaa (ks. alakohta 2.1.2). Viite rajapintaluokkaan saadaan käyttämällä JNDI API:n rajapintaa (ks. alakohta 2.1.2), jonka käyttö mahdollistaa viitteen saamisen riippumatta sen sijainnista verkkoympäristössä. Rajapintaluokat käyttävät EJB:n palvelin API:a (EJB client-side API), jonka välityksellä komponentin käyttö tapahtuu. EJB-komponenttimallin versiossa 2.0 on myös määritelty EJB:n paikallinen palvelin API, jonka kautta komponenttia voi käyttää tehostetusti yhden JVM:n sisällä ilman RMI-IIOP:ta ja sen mukanaan tuomaa rasitusta suorituskykyyn. Tämän tutkielman kannalta tuo paikallinen API olisi hyvinkin kiinnostava tutkimuksen kohde, mutta tutkielman teon ajankohtana ei käytetystä palvelinohjelmistosta ollut saatavilla paikallista rajapintaa tukevaa versiota. Asiakassovellus näkee komponentin objekteista ainoastaan EJB kotivälittäjän sekä EJB etävälittäjän peittävät rajapinnat (ks. kuva 2.1), jotka toteuttavat samat rajapinnat kuin EJB:n koti- ja etärajapinta. (Monson-Haefel 2001, 117-119.)

2.3.3 Poikkeusten käsittely

Poikkeusten (Exception) käsittely tapahtuu EJB-komponenttimallissa samalla tavalla kuin Javassa yleensäkin. Yhden *try* –lohkon sisältä mahdollisesti aiheutettavat (throw) poikkeukset on vastaanotettava (*catch*) välittömästi lohkon jälkeen. Sovelluspoikkeuksella (application exception) tarkoitetaan EJB-komponenttimallin yhteydessä kaikkia niitä poikkeuksia *java.rmi.RemoteException*:ia lukuunottamatta, joita EJB-komponenttien rajapintaluokat voivat nostaa. EJB-komponentit käyttävät

näitä poikkeuksia välittäessään tietoa asiakasohjelmalle epänormaaleista tapahtumista komponentin toiminnassa. Tyypillisesti järjestelmä kykenee toipumaan sovelluspoikkeuksen aiheuttamasta virheestä, eikä niitä olekaan tarkoitettu systeemitason (system-level) ongelmien raportointiin. Systeemitason poikkeukset hoidetaan EJB-komponentissa nostamalla kyseisen metodin esittelyn mukainen poikkeus, joka ei ole sovelluspoikkeus. Tarkemmin poikkeusten käsittely EJB-komponentissa määritellään lähteessä DiMichiel ym. (2001) luvussa 18. Tutkielman kannalta tarkempi määrittely tässä ei ole mielekästä, koska tutkielmassa tehtävässä testissä poikkeuksen aiheuttaneita testitapauksia ei hyväksytä testin tuloksiin. (DeMichiel ym. 2001, 371-374.)

3 TESTAUKSEN VIITEKEHYS

Toisessa luvussa esiteltiin EJB-komponenttimallin määritys tutkielman kannalta mielekkäiltä osin. Luvun keskeisin anti oli toimintaympäristön sekä siinä käytettävien komponenttien kuvaaminen. Tässä luvussa keskitytään kuvaamaan EJB-komponenttimallissa esiteltyjen komponenttien käyttöön liittyviä tekijöitä testauksessa käytettävän konstruktion näkökulmasta. Tässä luvussa myös määritellään testauksen viitekehys ja esitellään ne EJB:n käyttötavat, joita voidaan pitää merkittävinä siirrettäessä tietoa tietokantaan. Lisäksi tässä luvussa tehdään perusteltu valinta niistä tavoista joita testauksessa käytetään. Luvun tarkoituksena on selvittää, mihin EJB:n käyttötapojen ominaisuuksiin testauksessa kiinnitetään huomiota sekä otetaan kantaa testauksessa käytettävään tekniikkaan ja materiaaliin. Pääpaino viitekehyksen määrittelyssä on testauksessa saatavien tulosten vertailukelpoisuudessa. Jonkin verran otetaan myös kantaa siihen, miten vaihtoehdot vaikuttavat ohjelmiston laatuun, lähinnä modulaarisuuteen ja ylläpidettävyyteen.

3.1 Komponentti testauksen viitekehyksessä

Tutkielmassa tapahtuvan testauksen kohteena ovat EJB-komponentit. Näin ollen testauksen viitekehystä määriteltäessä on mielekästä ajatella testausohjelmiston muidenkin osien koostuvan komponenteista. Ohjelmistokomponentille on alan kirjallisuudessa esitetty lukuisia, toisistaan osin merkittävästikin poikkeavia määritelmiä (Vainio 2001, 10). Testauksen viitekehyksen kannalta on mielekästä esittää ohjelmistokomponentille melko korkean abstraktiotason määritelmä, joka antaa riittävästi liikkumavaraa testausohjelmiston komponenttien esittelyyn ja sitä kautta testauksen viitekehyksen määrittelyyn. Tässä luvussa keskitytään tämän kaltaisiin määritelmiin, eikä muunlaisia määritelmiä esitellä. Tässä luvussa käytetyillä termeillä komponentti ja ohjelmistokomponentti tarkoitetaan samaa asiaa.

D'Souza ja Wills (1998) esittävät komponentille määritelmän, jossa komponentti nähdään ohjelmiston kokoamisessa tarvittavana osana. Määritelmän mukaan komponentti tulee voida muuttamattomana liittää muiden komponenttien kanssa

osaksi jotain suurempaa kokonaisuutta. Edelleen komponentilla tulee olla hyvin määritellyt ja selkeät rajapinnat tarjoamistaan palveluista. Komponentin tulee olla näin liitettävissä muihin komponentteihin, eikä sen sisäinen muokkaaminen saa olla mahdollista. (D'Souza & Wills 1998, 386-387.)

Kozaczynski (1999) esittää komponentille tutkielman kannalta mielenkiintoisen määritelmän. Sen mukaan komponentti on samanaikaisesti sekä suunnittelu- ja rakenneyksikkö että konfiguraation hallinnan ja vaihtokelpoisuuden yksikkö, joka mukautuu rajapintoihin ja tarjoaa niiden toteutuksia. Edelleen määritelmän mukaan komponentti on ainoa yksikkö, joka sisältää edellämainitut ominaisuudet samanaikaisesti. Määritelmän tavoitteena on huomioida komponentin luonne käsitteellisenä osana sen suunnittelua, toteuttamista sekä julkaisua. (Kozaczynski 1999, 2-4.)

Yacoubin ym. (1999) esittämä määritys ohjelmistokomponentille sisältää joukon vaatimuksia komponenttia esitteleville kuvauksille. Nämä kuvaukset kuvaavat komponenttien inhimillistettyjä piirteitä ja ne on jaettu kolmeen luokkaan. Epäformaaleja kuvauksia ovat muun muassa komponentin ikä, käyttöyhteys ja tarkoitus. Ulkoiset kuvaukset esittelevät komponentin ulospäin näkyvät ominaisuudet, kuten rajapinnat ja vuorovaikutuksen komponentin suoritusalustaan. Sisäiset kuvaukset käsittelevät komponentin toteutusta ja luonnetta ja niiden tavoitteena on antaa vastauksia uudelleenkäytettävyyteen ja sovellusalueen laajuuteen liittyviin kysymyksiin. Yacoubin ym. (1999) mukaan siis komponentti on yksikkö, josta edellämainitut piirteet ovat löydettävissä ja kuvattavissa. (Yacoub ym. 1999.)

Kuten edellä mainittiin, on testauksen viitekehyksen määrittelyn kannalta mielekästä, että testausohjelmiston kaikkien osien voidaan ajatella olevan komponentteja. Näin ollen komponentille on löydettävä riittävän korkean abstraktiotason määritelmä, jotta tämä mahdollistuisi. D'Souzan ja Willsin (1998) määritelmä ei sellaisenaan täytä halutun kaltaisen määritelmän vaatimuksia. Esimerkiksi sen mukainen komponenttien sisäisen muokkaamisen kieltäminen estää tämän määritelmän käyttämisen sellaisenaan testausohjelmistossa käytettävien komponenttien yhteydessä. Määritelmässä on kuitenkin kiinnostava korkean tason vaatimus, jonka mukaan komponentti tulee voida muuttamattomana liittää osaksi muita komponentteja. Tämä

vaatii tarkkaa ja selkeää rajapintojen määrittelyä. Kozaczynskin (1999) määritelmän lähtökohta, komponentin luonteen huomioiminen käsitteellisenä osana suunnittelussa ja toteuttamisessa, on myös kiinnostava näkökulma lähestyä aihetta. Tutkielman testausohjelmistossa käytettävät komponentit todellakin ovat hyvin eri luonteisia, joten tätä näkökulmaa ei voida ohittaa testauksen viitekehystä määriteltäessä. Erityisesti Kozaczynskin (1999) esittämä kaksijakoinen vaatimus komponentin ominaisuuksille olla samanaikaisesti sekä suunnittelu- ja rakenneyksikkö että konfiguraation hallinnan ja vaihtokelpoisuuden yksikkö, soveltuu mainiosti testausohjelman komponenttien kuvaamiseen. Määritelmä vaatii kuitenkin hieman tarkennuksia ainakin rajapintojen vaatimuksiin sopiakseen sellaisenaan komponentin määritelmäksi tässä tutkielmassa. Yacoub ym. (1999) keskittyy määritelmässään komponenttien piirteiden esittämiseen kolmesta eri näkökulmasta. Poikkeuksena tässä määritelmässä kahteen edellä esitettyyn nähden on epäformaalien piirteiden kuvaamisen vaatimus. Se tuo testauksen viitekehyksen esittämisen kannalta mielenkiintoisia välineitä erityisesti testausohjelmiston komponenttien luonteen kuvaamiseen.

Tutkielman määritys komponentille on eräänlainen sekoitus edellä esitetyistä määrityksistä. Komponentilla tarkoitetaan tässä tutkielmassa kokonaisuutta, joka voi koostua pienemmistä komponenteista. Komponentilla on oltava tarkasti ja selkeästi määritelty rajapinta muihin komponentteihin. Komponentin tehtävän, käyttöyhteyden ja sovellusalueen tulee niinikään olla selkeästi määriteltyjä. Lisäksi komponentti tulee olla vaihdettavissa toiseen komponenttiin sekä liitettävissä muuttamattomana toisiin komponentteihin. Tämän määritelmän mukaisesti esimerkiksi tietokannan voidaan ajatella olevan ohjelmiston yksi komponentti ja toisaalta yksi luokka, vaikkapa EJB, voi niinikään olla komponentti.

3.2 Viitekehyksen määritys

Testauksen keskeisimpänä mielenkiinnon kohteena on suorituskyvyn tehokkuus. Käytännössä tämä tarkoittaa ajan mittaamista siirrettäessä saman laatuista ja samaa määrää tietoa eri tavoilla tietokantaan. Jotta eri EJB:n käyttötapojen suorituskyvyn mittaamisessa saadut tulokset ovat keskenään vertailukelpoisia, on poistettava kaikki

ne tiedon siirtonopeuteen vaikuttavat tekijät, jotka eivät kuulu kunkin tavan ominaispiirteisiin. Tällä tarkoitetaan sitä, että pyritään vakioimaan kaikkien muiden komponenttien tuoma lisä suorituksen kokonaisaikaan, paitsi itse vertailtavien komponenttien. Komponenteilla tässä tarkoitetaan kaikkia edellisen luvun määrityksen mukaisia komponentteja. Myös tietokannan sekä asiakasohjelman ajatellaan tässä olevan yksi komponentti. Niiden tekijöiden vaikutus, joiden vaikutusta kokonaissuoritukseen ei voida täysin vakioida, pyritään minimoimaan. Tässä poikkeuksena on lähiverkko, jonka vaikutusta ei tutkielman testauksessa kyetä minimoimaan.

3.2.1 Asiakasohjelma

Asiakasohjelma on testausohjelmiston komponentti, jonka vaikutus testausprosessin suoritusaikaan pyritään vakioimaan. Testausprosessin suoritusajalla tarkoitetaan sitä aikaväliä, joka alkaa kun asiakasohjemassa käynnistetään ajanotto ja päättyy kun ajanotto pysäytetään. Asiakasohjelmassa tehtävistä toimenpiteistä kaikissa testitapauksissa vain yksi tapahtuu edellä mainitulla aikavälillä. Siinä asiakasohjelma kutsuu EJB-komponentin etärajapinnan välittäjän metodia siirtääkseen testausmateriaalia tietokantaan ja välittää metodikutsun parametrina tuon testimateriaalin. Asiakasohjelman kannalta metodin kutsun käyttämä aika riippuu lähiverkosta. Koska verkon vaikutusta ei kyetä minimoimaan, todetaan tässä asiakasohjelman sinänsä kuluttavan vakiomääräisen ajan, mutta lähiverkko tuo oman ennustamattoman lisänsä testausohjelmiston mitattavaan toimintaan. Tämä ennustamaton ajankulutus hyväksytään tutkielman testauksessa, jos sen aiheuttamat heilahtelut testin tuloksissa pysyvät testauksen analyysin viitekehyksessä määritellyissä rajoissa.

3.2.2 EJB-komponenttien käyttötavat

Tutkimuksessa tapahtuvan testauksen tavoitteena on saada vertailukelpoisia tuloksia sovellettaessa kahta EJB:n käyttötapaa suurien tietomäärien siirrossa tietokantaan. Nämä kaksi eri tapaa voidaan kumpikin toteuttaa usein eri keinoin. Toinen tapa on

käyttää tiedon siirrossa istuntopohjaista EJB-komponenttia, jossa tietokannan käsittely tapahtuu suoraan Structured Query Language (SQL)–komentoja käyttäen. Tästä tavasta käytetään myöhemmin termiä suoraviivainen tapa. Toinen tapa on käyttää istuntopohjaisen komponentin kautta kohdepohjaisia komponentteja, jotka edustavat suoraan tietokannan tietoa (ks. kohta 2.2). Tästä tavasta käytetään myöhemmin termiä oppikirjamainen tapa. Tutkielma pohjautuu hypoteesiin, jonka mukaan oppikirjamainen tapa käyttää EJB:tä on suorituskyvyltään merkittävästi heikompi verrattuna suoraviivaiseen tapaan suuria tietomääriä käsiteltäessä. Kiinnostuksen kohteena on, kuinka merkittävä tuo ero on ja missä tietomäärissä tiedon määrän kasvaessa.

Käytännön näkökulma tuo mielenkiintoisen tavan lähestyä testauskohteiden määrittelyä. Monson-Haefel (2001, 24, 42-43) kertoo kirjassaan istuntopohjaisten komponenttien olevan asiakassovelluksen laajennoksia. Hän mainitsee niiden toimivan eräänlaisina agentteina käsiteltäessä kohdepohjaisia komponentteja, ja istuntopohjaisten komponenttien luonne houkuttelee käyttämään niitä liiketoimintalogiikan toteuttamiseen, ohentamaan asiakassovellusta. Käytännön sovelluksissa tyypillinen tapa käyttää istuntopohjaisia komponentteja onkin juuri toteuttaa niillä yleisiä, mutta selkeästi rajattuja tehtäviä. Tämä luo intressin tehdä poikkileikkaus edellisessä kappaleessa esitettyihin komponenttien käyttötapoihin. Poikkileikkauksella tässä tarkoitetaan molempien komponenttien käyttötapojen testaamista kahdella eri tavalla. Toisessa tavassa tieto viedään tietokantaan mahdollisimman häiriöttömästi siten, että testausprosessin kaikissa vaiheissa pyritään komponenttien vaikutus testaustapahtuman suoritusaikaan minimoimaan. Tästä tavasta käytetään myöhemmin termiä puhdas tapa. Toisessa tavassa tietoa käsitellään ennen tietokantaan siirtämistä ja aiheutetaan näin koko prosessiin viivästyttävä tekijä, jonka kautta prosessi muistuttaa paremmin käytännön mallia EJB-komponentteja hyödyntävästä sovelluksesta. Tästä tavasta käytetään myöhemmin termiä käytännön tapa. Tällä jaolla pyritään selvittämään, onko käytännön tavalla käyttää EJB:tä suorituskykyjen eroa vähentävää tai lisäävää vaikutusta.

Tiedon välittäminen EJB-komponenttimallin mukaisesti asiakassovelluksen ja EJB-komponenttien välillä tapahtuu Java Remote Method Invocation (RMI):n avulla. Java RMI:n asettamat rajoitukset tiedon välitykseen on esitelty alakohdassa 2.1.4. Nämä

rajoitukset asettavat vaatimuksia testausaineiston käytölle testausprosessin tietyissä vaiheissa. Myös tietokannan tietotyyppien ja Javan luokkien keskinäinen skaalautuvuus tai skaalautumattomuus aiheuttaa omat rajoituksensa tietotyyppien ja luokkien valinnalle. Tässä tarkastellaan vaatimuksia asiakassovelluksen ja EJB-komponenttien välisessä kommunikaatiossa. Tietokannan vaikutuksiin keskitytään seuraavassa alakohdassa. Koska Java RMI mahdollistaa EJB-komponenttien hajautuksen ja koska hajautus on koko EJB-komponenttimallin keskeisimpiä piirteitä, käytetään tutkielman testauksessa EJB-komponentteja aina lähiverkon yli.

Asiakassovelluksen ja EJB-komponentin välisessä kommunikoinnissa välitettävän tiedon muodon on oltava joko jokin perustietotyypeistä tai luokka, joka toteuttaa *java.io.Serializable* –rajapinnan. Tämä sääntö koskee myös EJB-komponenttimallin versiossa 2.0 esiteltyjen paikallisten rajapintojen käyttöä komponenttien yhteydessä, vaikka ne eivät käytäkään Java RMI:tä tiedon välityksessä. Paikallisia rajapintoja ei tarkemmin esitellä tässä tutkielmassa, koska tutkielman teon ajankohtana testauksessa käytetty palvelinohjelmisto ei niitä tukenut. Paikallisten rajapintojen määritelmä löytyy lähteestä DeMichiel ym. (2001). Perustietotyyppien käyttö asiakassovelluksen ja EJB-komponentin välisessä kommunikoinnissa on kätevää tilanteissa, joissa välitettävän tiedon määrä ei ole kovin suuri ja tietotyyppien määrä pysyy vähäisenä. Tutkielman testausmateriaali ei kuitenkaan täytä näitä vaatimuksia, joten tiedon välityksessä käytetään ensisijaisesti *java.io.Serializable* –rajapinnan toteuttavia luokkia. Asiakasohjelman ja EJB-komponenttien välisessä kommunikoinnissa käytettävät luokat esitellään tarkemmin neljännessä luvussa.

3.2.3 Tietokanta ja tietotyypit

Tutkielman konstruktiossa käytetään Oracle-tietokantaa. Edellä mainittiin, että tietokannan ajatellaan tässä olevan yksi konstruktion komponentti. Tällainen ajattelutapa on mielekästä, koska tietokanta on konstruktion näkökulmasta yksi selkeän, rajatun tehtävän toteuttava osa. Tietokannan tehtävänä on varastoida testauksessa käytettävä tieto jokaisesta testaustapahtumasta ja siten toimia päätepisteenä ja tarkistuskohteena testaustapahtumien onnistumisen varmentamiseksi. Tietokanta kuuluu edelleen aiemmin esitetyn jaon mukaisesti niihin komponentteihin,

joiden vaikutus testausprosessiin pyritään vakioimaan (ks. kohta 3.2). Käytännössä tämä tarkoittaa sitä, että testausprosessissa tietokannassa tapahtuvan prosessoinnin määrä minimoidaan. Yksi toimenpide tämän varmistamiseksi on, että testauksessa käytettävä tieto viedään aina yhteen ja samaan tietokannan tauluun. Näin useamman tietokantataulun käytön vaikutusta suoritusaikaan ei tutkielmassa tarvinnut edes selvittää. Jotta testaustapahtumat eivät vaikuttaisi toisiinsa, tuo taulu tyhjennetään aina jokaisen testitapauksen välillä. Taulun tyhjentämättä jättäminen aiheuttaisi tietokannan aiheuttaman ylimääräisen viiveen tiedon määrän kasvaessa, koska tietokanta tarkastaa aina tauluun lisättävän rivin avaimen pätevyyden. Toinen tilanne, jossa tietokannan toiminta tuo oman viiveensä testausprosessiin, on tapa viedä tieto tietokantaan. Tässä tietokannan vaikutuksen vakioimiseksi olisi tiedon vieminen tietokantaan toteutettava aina samanlaisella SQL-lauseella. Kun vertailemme edellä esiteltyä oppikirjamaista tapaa ja suoraviivaista tapaa viedä tieto tietokantaan ja samalla tarkastelemme alakohdassa 2.2.2 esiteltyä kohdepohjaisen komponentin määritelmää, voimme havaita, että tiedon vientiä ei voida näin vakioida. Suoraviivaisessa tavassa on käytettävä SQL-lausetta, kun taas oppikirjamaisessa tavassa kohdepohjaisen komponentin ominaisuudet synkronisoidaan tietokannan tietojen kanssa säiliön toimesta. Tietokannan vaikutusta ei siis voida tältä osin vakioida testauksessa. Testauksen viitekehyksen määrittelyn kannalta tämä ei kuitenkaan ole ongelma, sillä erityyppisen tiedonviennin vaikutukset johtuvat itse asiassa vertailun kohteena olevien EJB-komponenttien, eivät tietokannan, ominaisuuksista.

Kuten edellä mainittiin, tutkielman konstruktiossa käytetään Oraclen tietokantaa. Oraclen tietokannassa käytetyt tietotyypit eivät ole täysin Javan tietotyyppien ja yleisten luokkien kanssa yhteensopivia. Esimerkiksi Javassa ei ole yhtä luokkaa, joka kuvaisi sekä päivämäärää että kellonaikaa. Sen sijaan niiden esittäminen on toteutettu erillisissä luokissa. Oraclessa puolestaan on perustietotyyppi *DATE*, joka voi esittää päivämäärää ja kellonaikaa samanaikaisesti (Koch & Loney 1997, 1060-1061). Tutkielman konstruktiossa tietoa siirretään EJB-komponentista Oraclen tietokantaan, joten käytettävien tietotyyppien esittely on tässä tarpeen. Kuten edellä mainittiin, tietokanta ei ole tutkielmassa vertailun kohteena oleva komponentti. Niinpä sen vaikutus testausprosessiin pyritään vakioimaan. Näin ollen on mielekästä pitää

lähtökohtana tietotyyppien valinnalle, että konstruktiossa käytetään tietokannan perustietotyyppien kanssa mahdollisimman yhteensopivia tietotyyppejä tai luokkia.

Oraclen perustietotyypeistä varteenotettavimmat ovat *VARCHAR2*, *NUMBER* ja *DATE*. Muut Oraclen tietotyypit ovat liian yksinkertaisia tai liian tietokantakeskeisitä tyyppejä, eikä niiden käyttö sen vuoksi tunnu mielekkäältä tässä tutkielmassa. Edellä mainitusta yhteensopimattomuudesta Javan kanssa johtuen *DATE* –tietotyyppiä ei tutkielmassa käytetä. *DATE* –tietotyypin käyttö johtaisi tiedon muokkaukseen ennen tiedon siirtoa tietokantaan ja tämä rikkoisi aiemmin edellä esiteltyä puhtaan EJB:n käyttötavan periaatetta testausprosessissa. *VARCHAR2* –tietotyyppi edustaa Oraclen järjestelmässä merkkijonoa ja se konvertoituu hyvin Javan *java.lang.String* –luokan kanssa. Koska tiedon välittäminen merkkijonomuodossa on varsin yleistä hajautetuissa järjestelmissä, on myös *VARCHAR2* –tietotyypin käyttäminen tässä luontevaa. Oraclen *NUMBER* –tietotyypillä esitetään kokonais- ja desimaalilukumuotoista tietoa ja sen käyttömahdollisuudet ovat erittäin moninaiset. Tämä tietotyyppi konvertoituu vähintäänkin kohtuullisesti Javan kaikkien numeerista tietoa esittävien perustietotyyppien kanssa. Koska *NUMBER* –tietotyyppiä käytetään erittäin yleisesti Oraclen tietokannoissa, on se myös luonteva valinta tähän tutkielmaan käytettäväksi. *VARCHAR2* –tietotyyppiä vastaavaksi tietotyypiksi EJB-komponentin puolella *java.lang.String* on edellä mainituista syistä paras valinta. *NUMBER* –tietotyyppiä vastaavaksi tietotyypiksi valitaan tässä *int*, joka on Javassa 32-bittinen (Eckel 2000, 105).

3.2.4 Viitekehyksen arviointia

Viitekehyksen määrittelyä varten on esitelty tutkielmassa käytettävän termin komponentti määritelmä. Tuon määritelmän mukaisesti testausohjelmistossa käytettävät komponentin on esitelty ja jaoteltu testauksen kohteena oleviin komponentteihin sekä muihin komponentteihin. Viitekehyksen määrityksen keskeisimmät rajaukset koskevat testauksen kohteena olevien komponenttien käyttötapoja sekä toimenpiteitä, joilla muiden komponenttien vaikutukset testaustapahtumiin pyritään vakioimaan. Lisäksi on otettu kantaa tietotyyppien ja luokkien käyttöön testauksen kannalta kriittisillä rajapinnoilla.

Edellä esiteltiin puhdas tapa sekä käytännön tapa siirtää tietoa tietokantaan. Käytännön tavassa tietoa käsitellään ennen siirtoa tietokantaan. Tästä vaatimuksesta löydetään tarve vielä yhdelle testausohjelmiston komponentille, joka ei ole testauksen kohteena. Koska tätä komponenttia on voitava käyttää toisen EJB-komponentin toimesta, on EJB luonnollinen valinta tämän komponentin tyypiksi. Jotta komponentti täyttäisi tarkoituksensa eli toisi käytännön kaltaisen mallin EJB-komponentteja käyttävästä sovelluksesta ja kuitenkin samalla vaikuttaisi testausprosessiin mahdollisimman neutraalisti, on komponentissa tapahtuvaan prosessointiin kuluvan ajan oltava suhteessa testauksessa kulloinkin käytettävän materiaalin määrään.

Muita kuin testauksen kohteena olevia komponentteja testausohjelmistossa ovat siis asiakasohjelma, tietokanta sekä yksi EJB-komponentti. Asiakasohjelmalle määriteltiin sellainen käyttömalli, jota käyttäen sen vaikutukset testausprosessiin voidaan katsoa vakioiduiksi. Tietokannan vaikutuksen vakioiminen oli hieman monimutkaisempaa, mutta senkin käytölle kyettiin löytämään malli, jossa tietokannan vaikutukset testausprosessiin on kyetty minimoimaan niin vähäisiksi, ettei niillä ole merkittävää vaikutusta testaustulosten oikeellisuuteen. EJB-komponentille myös määriteltiin malli, joka rajaa komponentin käyttämän prosessointiajan olevan suhteessa testausmateriaalin määrään. Kun myös testauksen kohteena olevien komponenttien käyttö sekä testauksen kannalta merkittävien rajapintojen sekä niissä käytettävien tietotyyppien ja luokkien käyttö on rajattu, voidaan testauksen viitekehyksen katsoa kattavan koko testausohjelmiston.

3.3 EJB:n käyttötapojen valinta

Edellä on esitelty suoraviivainen tapa sekä oppikirjamainen tapa käyttää EJB:tä tiedonsiirrossa tietokantaan. Edelleen on mainittu, että nuo molemmat tavat voidaan toteuttaa eri tavoin. Tässä luvussa selvitetään nuo tavat ja valitaan niistä sopivimmat tutkielmassa tehtävään konstruktioon. Lisäksi vielä vertaillaan näitä käyttötapoja valittujen toteutustapojen suhteen ja otetaan kantaa tietokantayhteyden hallintaan komponentin sisäisessä toteutuksessa.

3.3.1 Tilallinen ja tilaton istuntopohjainen komponentti

Suoraviivaisessa tavassa käyttää EJB:tä tieto viedään tietokantaan suoraan istuntopohjaisesta komponentista SQL-lauseen välityksellä. Istuntopohjaisia komponentteja on olemassa kahden tyyppisiä, tilallisia ja tilattomia. Tilattomat istuntopohjaiset komponentit ovat EJB-komponenttimallin tehokkaimpia komponentteja. Niiden resurssit vapautetaan ja ne passivoidaan aina jokaisen käyttötapahtuman jälkeen ja teho perustuu sitä kautta tapahtuvaan tehokkaaseen komponenttien kierrätykseen (ks. 2.2.1.1). Tilalliset komponentit on aina sidottu johonkin asiakassovellukseen. Myös tilallinen komponentti voidaan passivoida, mutta passivoinnin yhteydessä niiden tila talletetaan aina johonkin pysyvään tietovarastoon. Komponentin aktivoinnin yhteydessä tämä tila palautetaan ja vaikka komponentin ilmentymä ei olisikaan sama kuin edellisen kutsun yhteydessä, on komponentti kuitenkin täsmälleen samassa tilassa (ks. 2.2.1.2). Näin siis myös tilallisia komponentteja kierrätetään, mutta se on huomattavasti raskaampi prosessi, kuin tilattomien komponenttien yhteydessä. Molemmissa tapauksissa komponenttien kierrättäminen tapahtuu säiliön toimesta.

Säiliö siis kierrättää molempia istuntopohjaisia komponenttityyppejä itsenäisesti siten, että kierrätys ei ole testausohjelman käyttäjän kontrolloitavissa. Testaustapahtumissa on varmistuttava komponenttien suhteen tasa-arvoisesta lähtötilanteesta. Testaustapahtumien välillä ei voida tietää, onko säiliön muistissa kutsuttavan komponentin ilmentymä valmiina käytettäväksi, ja edelleen voidaan olettaa, että valmiina olevan komponentin käyttöönotto on tehokkaampaa kuin uuden komponentin luonti. Tästä johtuen tasa-arvoisen lähtötilanteen varmistamiseksi testaustapahtumien välillä on säiliöohjelma käynnistettävä jokaisen testaustapahtuman välillä uudestaan. Tästä aiheutuu edelleen se, että tilattoman komponentin tehokkuusetu suhteessa tilalliseen komponenttiin häviää. Näin ollen testauksen kannalta on merkityksetöntä, käytetäänkö tilallista vai tilatonta istuntopohjaista komponenttia. Testausohjelmistossa käytettävän istuntopohjaisen komponentin täytyy kuitenkin sisältää sellaisia muutettavissa olevia ominaisuuksia, jotka ovat tyypillisiä tilalliselle istuntopohjaiselle komponenttille. Näihin ominaisuuksiin kuuluu ainakin tietokantayhteyden hallinta. Niinpä testausohjelmistossa käytetään sekä

oppikirjamaisen että suoraviivaisen tavan yhteydessä aina tilallista istuntopohjaista komponenttia, vaikka sitä käytettäisiinkin kuten tilatonta komponenttia.

3.3.2 CMP- ja BMP-komponentti

Oppikirjamaisessa tavassa siirtää tietoa tietokantaan käytetään istuntopohjaisen komponentin kautta kohdepohjaista komponenttia. Kohdepohjaisia komponentteja on kahden tyyppisiä (ks. alakohta 2.2.2), CMP-komponentti ja BMP-komponentti. BMP-komponentissa pysyvyyden ja tapahtuman hallinnasta huolehtiminen on komponentin toteuttajan vastuulla. CMP-komponentissa taas pysyvyydestä ja tapahtuman hallinnasta huolehtii säiliö automaattisesti. Etuja BMP-komponentin käytölle verrattuna CMP-komponenttiin tuntuu olevan vaikea löytää. Monson-Haefelin (2001) mukaan BMP:n käyttö CMP:n sijaan on perusteltua tilanteessa, jossa saman kohdepohjaisen komponentin tulee hallita tieto erityyppisistä tietokannoista ja näin tilan hallinta tietokannan ja komponentin välillä on paremmin kontrolloitavissa (Monson-Haefel 2001, 265). Perustelu tuntuu hieman ontuvalta, sillä saman tilanteen voi hoitaa useammalla CMP-komponentilla ilman merkittäviä komponenttien määrän aiheuttamia tappioita tehokkuudessa. Vaikka komponentin ja tietokannan synkronisoinnin parempi hallinta tässä tapauksessa onkin löydettävissä, on tapaus niin erikoislaatuinen ja harvinainen, että sen käyttö perusteluksi BMP-komponentin olemassaololle vaikuttaa teennäiseltä. Yleensä ottaen alan kirjallisuudesta jää sellainen mielikuva, että BMP-komponentti on historiallinen jäänne EJB-komponenttimallin version 1.0 ajoilta. Koska suorituskyvyn suhteen kohdepohjaisen komponenttityypin valinnalla ei näytä olevan suurempaa merkitystä ja koska testausohjelmiston tapauksessa BMP-tyyppisen komponentin käytölle ei ole perusteita, käytetään tutkielman testauksessa CMP-tyyppistä kohdepohjaista EJB-komponenttia.

3.3.3 Käyttötapojen vertailua

Testauksen viitekehyksen rajauksen lähtökohtana on kahden eri tiedonsiirtotavan vertailu. Tämän vertailun pohjaksi määriteltiin suoraviivainen ja oppikirjamainen tapa

käyttää EJB-komponentteja tiedonsiirrossa tietokantaan. Lisäarvoa tutkimuksesta haettiin lisäämällä vielä käytännön näkökulma ja tehtiin poikittainen jako puhtaaseen ja käytännön tapaan. Tällä tavalla saatiin määriteltyä neljä erilaista tapaa siirtää tietoa tietokantaan.

Oppikirjamainen tapa on yleisesti suositeltava tapa käyttää EJB:tä. Sen suurimpia etuja ovat muun muassa puhtaasti komponenttimaisen rakenteen säilyminen, automaattinen tapahtumanhallinta ja SQL-lauseiden välttäminen istuntopohjaisen komponentin toteutuksessa. Suoraviivaisessa tavassa komponenttimainen rakenne rikkoontuu siinä mielessä, että rakenteellisesti yhteenkuuluvat tietojoukot, esimerkiksi tietueet, eivät ole paketoituna komponenttiin, vaan ne hajoavat toteuttajan valitsemiin tietorakenteisiin. SQL-lauseiden välttäminen istuntopohjaisen komponentin toteutuksessa voidaan ajatella olevan oppikirjamaisen tavan etu siinä mielessä, että SQL-lauseiden käyttö on huomattavasti epäformaalimpaa, ja sitä kautta vaikeampaa ylläpitää, kuin kohdepohjaisen komponentin kotirajapinnan hakumetodien käyttö. Toisaalta tämän kotirajapinnan hakumetodien mahdollisuudet tiedon haussa tietokannasta, etenkin useammasta taulusta tehtävien yhdisteiden luominen, ovat rajoittuneempia kuin SQL-lauseiden käytössä. Tämä ongelma on vielä olemassa EJB-komponenttimallin versiossa 1.1, mutta versio 2.0 tuo mukanaan jo oman EJB Query Language (EQL)-kyselykielen, jolla kyetään toteuttamaan monimutkaisempia hakuja. Vaikka tässä tutkielmassa ei EQL-kyselykieltä käytetäkään, voidaan todeta, että SQL-lauseiden välttäminen istuntopohjaisessa komponentissa on joka tapauksessa etu ohjelmiston ylläpidettävyyden kannalta. Näin ollen suoraviivaisen tavan ainoaksi eduksi jää tutkielman pohjana olevan hypoteesin mukaisesti nopeus.

Kuten edellä mainittiin, tutkielmalle on haettu lisäarvoa tuomalla tutkimukseen mukaan käytännön tapa käyttää EJB:tä. Käytännön tavassa testaustapahtumaan lisätty keinotekoinen viive on lähes suoraan suhteessa käytettävän testausmateriaalin määrään. Viive on toteutettu kutsumalla istuntopohjaisesta komponentista toista EJB-komponenttia, joka tarjoaa palvelunaan XML-dokumentin jäsentämisen. Tuo suhde testausmateriaalin määrään aiheutuu siitä, että tietokantaan vietävien rivien määrä kasvattaa lineaarisesti XML-dokumentin kokoa ja tämä taas näkyy lisääntyneenä viiveenä jäsentämiseen käytettävän komponentin toiminnassa. Sitä, kuinka hyvin tai huonosti tämä malli kuvaa käytännön tapaa käyttää EJB:tä, on vaikea arvioida.

Voitaisiin esimerkiksi ajatella mallin, jossa viive olisi vakiomittainen, mutta sidoksissa testausmateriaalin määrän sijaan jokaiseen tietokantakutsuun, olevan paremmin käytännön tapaa kuvaava. Edelleen voitaisiin ajatella molempien edellä kuvatun kaltaisten viiveiden olevan välttämättömiä, jotta voitaisiin sanoa tavan kuvaavan käytännön tapaa käyttää EJB:tä. Käytännön toteutuksista on kuitenkin löydettävissä malli, jossa EJB:tä käytetään samankaltaisesti kuin tässä tutkielmassa. Koska puhtaan tavan ja käytännön tavan vertailu ei ole tutkimuksen keskeisintä aluetta, tyydytään tässä olemaan tarkastelematta edellä kuvattua käytännön tapaa käyttää EJB:tä turhan kriittisesti ja hyväksytään se tapa yhdeksi mahdolliseksi käytännön toteutukseksi.

3.3.4 Tietokantayhteyden hallinta komponentissa

Suoraviivaisessa tavassa siis tieto viedään tietokantaan suoraan istuntopohjaisesta EJB-komponentista. Tämä tarkoittaa myös sitä, että istuntopohjainen komponentti huolehtii tietokantayhteydestä. Koska edellä valittiin testauksessa käytettäväksi nimenomaan tilallistä istuntopohjaista komponenttia, voidaan yhteys tietokantaan testaustapahtumissa hoitaa testauksen kannalta kahdella eri tavalla. Yhteys tietokantaan voidaan ottaa komponentin luonnin yhteydessä, jolloin samaa yhteyttä käytetään jokaisessa suoraviivaisessa testaustapahtumassa. Toinen vaihtoehto on ottaa yhteys tietokantaan aina jokaisen suoraviivaisen tavan testauksessa tulevan kutsun alussa, jolloin istuntopohjaisen komponentin käyttö luonnollisesti hieman hidastuu. Vaikka tuo hidastuminen ei etenkään suurissa tietomäärissä ole kovin merkittävä, tehdään tässä valinta tietokantayhteyden käytöstä. Oracle8i-tietokannassa samanaikaisten JDBC-yhteyksien määrä on määriteltävissä asennusvaiheessa, mutta se on kuitenkin rajoitettu. Jos jokainen istuntopohjainen komponentti luo muodostimessaan yhteyden tietokantaan, loppuvat mahdolliset yhteydet tietokantaan kesken ohjelmiston kasvaessa riittävän suureksi. Jos tietokantayhteydet loppuvat, syntyy riskialtis tilanne, jossa voi tapahtua ohjelmiston merkittävää hidastumista. Olisi siis melkoisen lyhytnäköistä olla vapauttamatta tietokannan resursseja jokaisen käyttökerran jälkeen, vaikka sillä ei testausohjelman tapauksessa merkitystä resurssien kannalta olekaan. Näin ollen testausohjelmistossa käytetään tietokantayhteyttä

säästeliäästi siten, että yhteys luodaan jokaisen suoraviivaisen testaustapahtuman alussa ja vapautetaan testaustapahtuman lopussa.

4 TESTAUSYMPÄRISTÖ

Toisessa luvussa esiteltiin EJB-komponenttimallin määritys tutkielman kannalta olennaisilta osin. Kolmannessa luvussa määriteltiin komponentti käsitteenä sekä rajattiin testauksen viitekehys. Tässä luvussa esitellään tuon viitekehyksen mukaisesti toteutetun testausohjelmiston arkkitehtuuri sekä komponenttikohtaisesti ne piirteet, jotka ovat tutkielman kannalta kiinnostavia.

Testausohjelmiston rakenteen lähtökohtana on järjestelmä, jossa energiamittareista etäluettua mittaustietoa siirretään tietokantaan. Tämä näkyy testauksessa käytettävässä materiaalissa, jossa XML-tiedoston rakenne ja tiedot voisivat pienin muutoksin sellaisenaan olla mittaustietoa. Testauksessa käytettävässä materiaalissa on myös sellaista tietoa, jota ei ole tarkoitus siirtää tietokantaan, vaan sitä on käytetty esimerkiksi kuvitellussa tiedon siirrossa ennen tiedon saapumista tietokantarajapinnalle. Tämä käytännön lähtökohta testausohjelmistolle näkyy myös joissain attribuuttien sekä tietokannan nimeämiskäytännöissä. Lähtökohta ja sen mukanaan tuomat ominaisuudet sekä ratkaisut testausohjelmistossa on pyritty ratkaisemaan siten, etteivät ne vaikuta testauksen tuloksiin.

4.1 Testausohjelmiston vaatimuksista

Edellä on esitelty testausohjelmistolle korkean tason vaatimuksia, jotka toimivat lähtökohtana suunnittelulle. Tässä tehdään tiivistetty katsaus näihin vaatimuksiin ja esitellään, mihin tarkemman tason määrityksiin ja ratkaisuihin ne ovat testausohjelmiston suunnittelussa johtaneet. Keskeisiä korkean tason vaatimuksia testausohjelmistolle löytyy neljä kappaletta. Testausohjelmistolla on kyettävä viemään tietoa tietokantaan EJB-komponentteja eri tavoin käyttäen. Näistä tiedonsiirroista on kyettävä mittaamaan aika ja ohjelmiston käyttäjälle on välitettävä tieto mittauksen tuloksista. Lisäksi ei-toiminnallisena vaatimuksena on, että samoilla tietomäärillä tehdyistä testitapauksista saatavien tulosten on oltava keskenään vertailukelpoisia. Ei-pakollisena vaatimuksena voidaan pitää asiakasohjelman käyttöä lähiverkon yli. Testauksen viitekehyksen määrityksen tarkentuessa on päädytty EJB-komponenttien

käyttötavoista kahteen merkittävimpään, oppikirjamaiseen ja suoraviivaiseen tapaan. Edelleen näiden molempien tapojen yhteydessä käytetään puhdasta ja käytännön tapaa siirtää tietoa tietokantaan. EJB:n käyttötavoista saadaan johdettua testausohjelmistolle vaatimus, jonka mukaan vähintään yhden tiedonsiirtoon käytettävän EJB-komponentin on tarjottava edellä kuvatut neljä palvelua asiakasohjelmalle. Lisäksi käytännön tapa siirtää tietoa tietokantaan tuo testausohjelmistolle vaatimuksen tietoa käsittelevästä komponentista.

Edellä mainitut neljä palvelua asiakasohjelmalle on mahdollista toteuttaa useilla eri tavoilla. Laajemmassa ja monimutkaisemmassa ohjelmistossa nämä palvelut kannattaisi todennäköisesti toteuttaa kukin omaan komponenttiinsa. Testausohjelmiston toteutus pyritään kuitenkin pitämään mahdollisimman yksinkertaisena, jotta hallittavuus säilyy hyvänä ja sitä kautta vältetään riskitekijöitä testituloksien oikeellisuuden suhteen. Kaikki testaustapahtumat aloitetaan aina käynnistämällä säiliö uudelleen (ks. alakohta 3.3.1), joten käytetyn komponentin identiteetillä ei ole testaukseen vaikutusta. Komponenttien koko voi vaikuttaa suorituksen nopeuteen. Koska testausohjelmistossa vertailun kohteena olevien komponenttien liiketoimintalogiikka on erittäin vähäistä, pysyy niiden koko verrattain pienenä. Komponenttien koon ei siten voida katsoa vaikuttavan testitulosten oikeellisuuteen, vaikka kaikki neljä palvelua toteutettaisiin yhteen komponenttiin. Edellä mainituista syistä johtuen on kyseiset neljä palvelua päädytty toteuttamaan yhteen EJB-komponenttiin, *sEJBInsert*:iin.

Testissä käytettävä materiaali generoidaan aina XML-muotoon. Tiedon siirtäminen tietokantaan suoraan XML-muodosta on kuitenkin hankalaa ilman kunnollista XML-jäsentäjää (parser). XML-jäsentäjällä tarkoitetaan tässä komponenttia, joka muokkaa tiedon XML-muodosta johonkin toiseen muotoon. Toisen edellä esitellyn matalan tason vaatimuksen mukaisesti testausohjelmiston on sisällettävä käytännön tavan testitapauksissa tiedon käsittelyä. Koska testiaineisto on joka tapauksessa järkevää muuntaa XML-muodosta johonkin toiseen muotoon, on testiohjelmistossa päädytty toteuttamaan EJB-komponentti, joka purkaa halutut tiedot testiaineistosta *java.util.Properties*-muotoon. Tämän tiedon muodon muuttamisen katsotaan testiohjelmistossa kattavan käytännön tavan vaatimukset tiedon käsittelystä ennen vientiä tietokantaan. Testauksessa siis tiedon muoto muutetaan kaikissa

testitapauksissa käyttäen *sEJBParser*-komponentin palvelua, puhtaassa tavassa ennen ajanoton aloittamista ja käytännön tavassa ajanoton aloittamisen jälkeen. XML-muodon valinta perustuu testausmateriaalin puumaiseen rakenteeseen, jonka esittämisen muodoksi XML on luonnollinen valinta. Sinänsä XML ei suoranaisesti liity tutkielmassa tehtävään testaukseen, se on vain yksi monista mahdollisista testauksessa käytettävän tiedon muodoista.

4.2 Kokonaisarkkitehtuuri

Kolmannessa luvussa määriteltiin neljä eri tapaa siirtää tietoa tietokantaan EJB-komponentteja käyttäen. Oppikirjamainen tapa siirtää tietoa on käyttää istuntopohjaisen EJB-komponentin kautta kohdepohjaista komponenttia. Suoraviivaisessa tavassa tieto siirretään tietokantaan suoraan istuntopohjaisesta komponentista JDBC-rajapintaa käyttäen. Testausohjelmistossa näitä molempia tapoja on kolmannen luvun määrityksen mukaisesti voitava käyttää kahdella eri tavalla. Puhtaassa tavassa tieto pyritään siirtämään mahdollisimman häiriöttömästi ja käytännön tavassa tietoa käsitellään ennen siirtoa tietokantaan. Testausohjelmiston keskeisin anti on esittää toteutukset näille tiedonsiirtotavoille. Tässä luvussa esitetään arkkitehtuuri testausohjelmistolle, jossa nämä neljä eri käyttötapaa on toteutettu komponentin palveluina asiakasohjelmalle. Lisäksi esitellään toteutus tiedonsiirrossa käytettyjen tapojen mittaamiselle sekä mittaustulosten esittämiselle.

Jaaksi ym. (1999) esittävät kirjassaan 3+1-mallin arkkitehtuurien kuvaamiseen. Malli käyttää Unified Modelling Language (UML)–notaatiota kuvaamaan neljää eri näkökulmaa arkkitehtuuriin. Malli on kohdistettu erityisesti suurten järjestelmien kuvaamiseen, mutta sitä voi soveltaen käyttää myös pienempien järjestelmien kuvaamiseen. 3+1-mallin looginen näkymä (Logical View) kuvaa korkean tason näkökulmaa, jossa ohjelmisto jaetaan aliohjelmistoihin (application product) ja ohjelmiin (application). Näkymän tarkoituksena on esittää järjestelmän rakenne suuressa mittakaavassa ja muodostaa eräänlainen perusta tarkemman tason arkkitehtuurin kuvaamiselle. Ajonaikainen näkymä (Run-Time View) kuvaa järjestelmän ajonaikaista rakennetta luokkakaavioiden (class diagram) ja sijoituskaavioiden (deployment diagram) avulla. Siinä kuvataan lähinnä järjestelmän

sisältämät prosessit ja niiden väliset riippuvuudet. Kehittämisnäkymä (Development View) kuvaa järjestelmän jakautumisen komponenttitasolla. Skenaarionäkymä (Scenario View) kuvaa tiedon virtaa eri komponenttien välillä ajon aikana. Skenaarionäkymä kuvataan herätekaavioiden (sequence diagram) avulla. (Jaaksi ym. 1999, 212-224.)

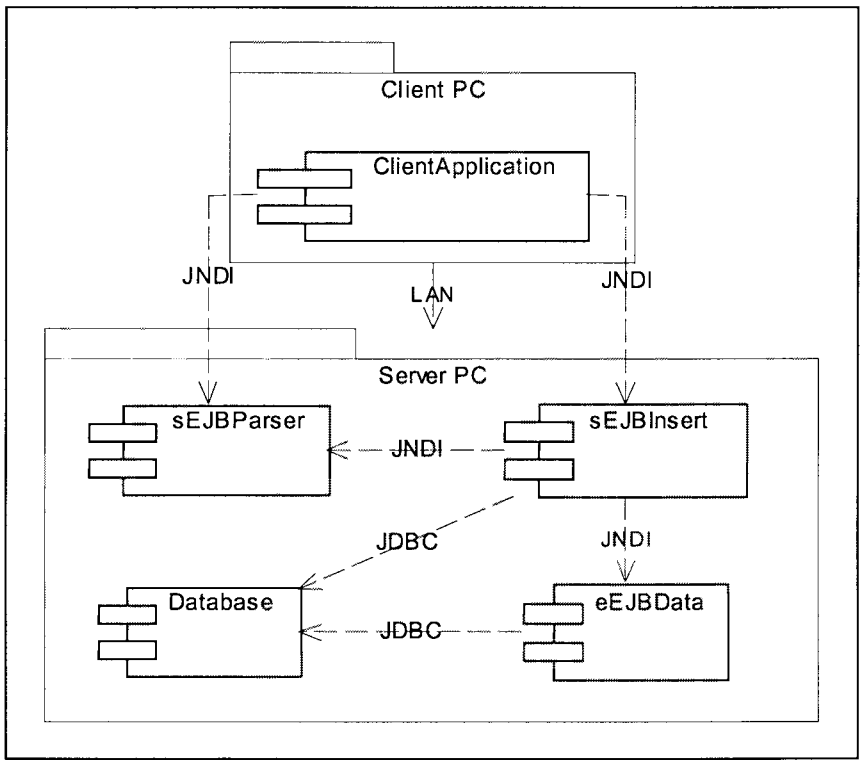
Jaaksin ym. (1999) esittämä malli tukee erityisesti komponenttikeskeisen ohjelmiston kuvaamista, joten se sopii mainiosti tämän tutkielman välineeksi arkkitehtuurin kuvaamiseen. Koska kyseinen malli kuitenkin ensisijaisesti on tarkoitettu palvelemaan suurten järjestelmien kuvaamista, käytetään sitä tässä soveltaen. Testausohjelmiston kuvauksen ei tässä tarvitse olla kattava muuten kuin niiltä osin, joilla on merkitystä testauksen oikeellisuuden toteamiselle. Edellä kuvatun 3+1-mallin looginen näkymä on korkeatasoinen kuvaus, ja se palvelee enemmän isojen järjestelmien arkkitehtuurien jäsentämistä kuin pieniä järjestelmiä. Vaikka testausohjelmistosta on erotettavissa tyypillisen kolmikerroksisen arkkitehtuurin kerrokset, on se kuitenkin pieni järjestelmä. Loogisen näkymän esittäminen tässä yhteydessä ei näin ollen toisi arkkitehtuurin kuvaamisen kannalta merkittävää etua, eikä sitä tässä käytetä. Ajonaikaisen näkymän esittämät prosessien keskinäiset riippuvuudet niin ikään edustavat verrattain korkean tason näkökulmaa järjestelmän arkkitehtuurista. Testausohjelmiston kannalta ajonaikaisen näkymän mielekkäin piirre on ohjelman osien laitteistotason sijoittumisen esittäminen. Tämä voidaan kuitenkin esittää myös kehittämisnäkymän yhteydessä, joten ajonaikaista näkymää ei käytetä testausohjelmiston arkkitehtuurin kuvaamiseen. Kehittämisnäkymän esitys ohjelmiston komponenttitasoisesta rakenteesta puolestaan sopii hyvin testausohjelmiston kuvaamiseen, koska siinä liikutaan riittävän matalalla abstraktiotasolla. Myöskin skenaarionäkymän edustaman ajonaikaisen dynamiikan esittäminen on tässä tarpeen testausohjelmiston toiminnan ymmärtämiseksi.

Edellä mainittiin Jaaksin ym. (1999) esittämän mallin käyttävän UML-notaatiota. Malli ei kuitenkaan sido käyttämään mitään tiettyä notaatiota, tekijät ovat vain valinneet yhden mallin kannalta kelvollisen notaation (Jaaksi ym. 1999, 4). Tässä luvussa ei keskitytä siihen ovatko kuvaukset syntaksiltaan jonkin tietyn notaation mukaisia. Sen sijaan keskitytään kuvausten semantiikkaan. Tässä luvussa esitetyt mallit kuitenkin pyritään esittämään UML-notaation kaltaisesti siten, että loogisen

päättelyn kautta yhteneväisyydet vastaaviin UML-notaation kuvauksiin on helposti löydettävissä.

4.2.1 Kehittämisnäkymä

Kehittämisnäkymän tarkoituksena on esittää testausohjelmiston staattinen rakenne ajon aikana. Näkymässä kuvataan ohjelmiston komponentit sekä niiden väliset riippuvuudet.



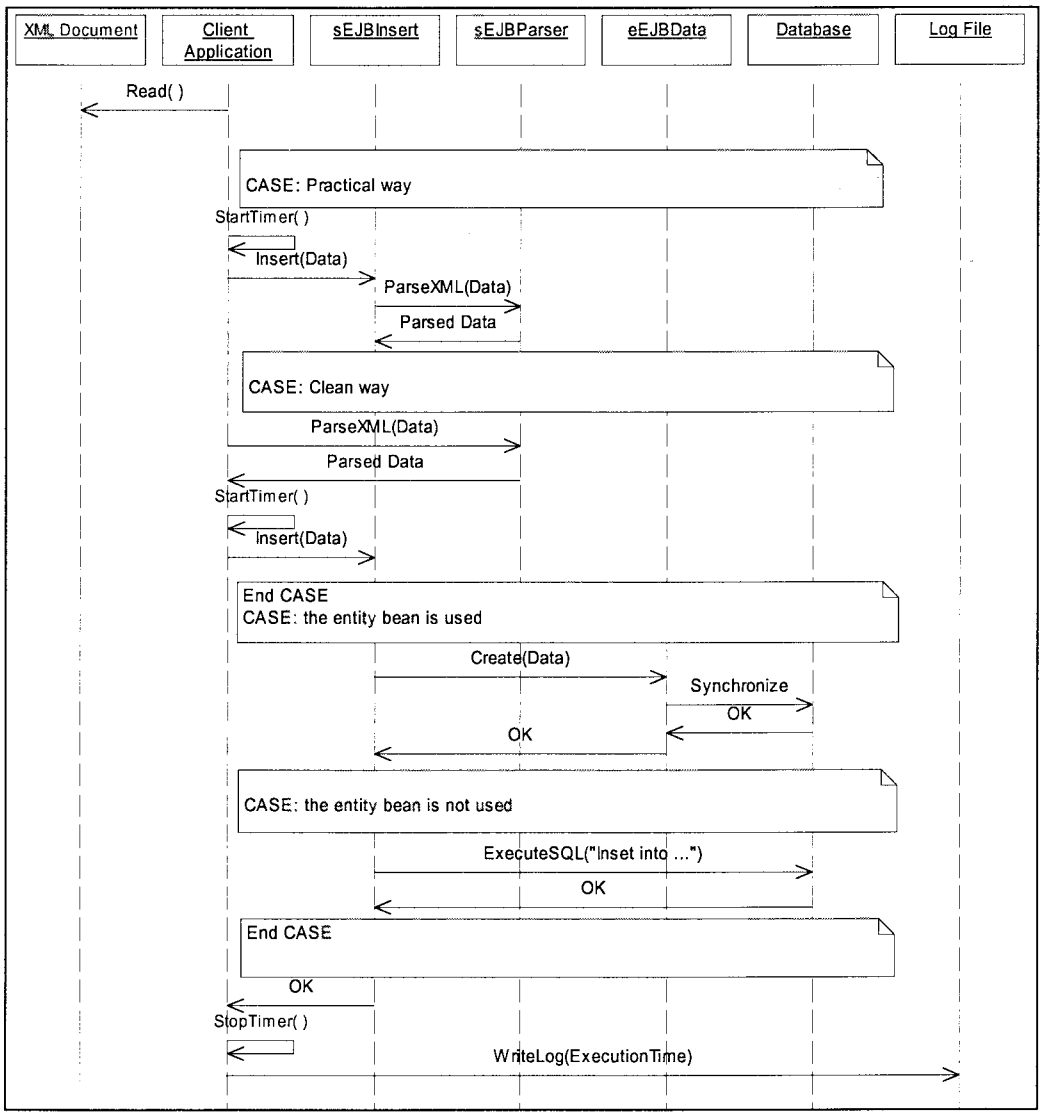
Kuva 4.1: Testausohjelmiston komponentit ja niiden riippuvuudet.

Kuvassa 4.1 on esitetty testausohjelmiston komponenttitasoinen rakenne. Poikkeuksena UML-notaatioon, siinä on esitetty myös komponenttien jakautuminen ajon aikana eri fyysisissä laitteissa. Myöskään rajapintoja ei ole kuvattu UML-notaation mukaisesti, on vain esitelty komponenttien väliset riippuvuudet. Asiakaskoneessa (Client PC) on yksi komponentti, asiakasohjelma. Sen tehtävänä on lähettää tietokantaan siirrettävää tietoa palvelinkoneelle (Server PC) ja mitata siirtoon kuluva aika. Palvelinkoneessa rajapintana asiakkaalle toimii *sEJBInsert*-komponentti.

Se tarjoaa asiakkaalle neljä palvelua, jotka ovat toteutuksia edellä esitellyille tavoille siirtää tietoa tietokantaan. *eEJBData*-komponentti on testausohjelman ainoa kohdepohjainen komponentti ja se edustaa siten suoraan tietokannan tietoa (ks. alakohta 2.2.2). *sEJBParser*-komponentin tehtävänä on datan käsittely ja sitä käytetään käytännön tavassa siirtään tietoa tietokantaan. Nämä kolme EJB-komponenttia ja niiden tarjoamat palvelut on esitelty tarkemmalla tasolla myöhemmin. Asiakasohjelman ja EJB-komponenttien, sekä komponenttien keskinäinen kommunikointi tapahtuu JNDI:n kautta. Yhteydet tietokantaan hoidetaan JDBC-rajapinnan kautta. Vaikka Oracle9iAS Containers for J2EE (OC4J) –säiliö täyttääkin kolmannessa luvussa määritellyn komponentti –käsitteen vaatimukset, ei sitä ole tässä yhteydessä esitelty komponenttina. Sen ajatellaan tässä pikemminkin olevan suoritusalusta EJB-komponenteille.

4.2.2 Skenaarionäkymä

Skenaarionäkymä kuvaa testausohjelmistossa tapahtuvaa kontrollin vaihtoa sekä tiedon virtaa yhden testaustapahtuman aikana. Näkymän tarkoituksena on herätekaavion avulla esitellä ohjelmiston komponenttien ajonaikainen yhteistoiminta sekä ohjelmistoon liittyvien tiedostojen käyttö. Alla esitetyn kuvan 4.2 herätekaaviossa on esitelty kaksi testausohjelmistolle keskeistä tiedostoa. XML-tiedosto (XML Document) sisältää testauksessa käytettävän materiaalin, ja sen sisältämästä tiedosta osa siirretään tietokantaan. Loki-tiedostoon (Log File) kirjoitetaan kustakin testaustapahtumasta yksi rivi, joka sisältää testaustapahtuman ajankohdan, tiedonsiirtoon kuluneen ajan sekä siirretyn tiedon kuvauksen. Lokin kirjoittaminen ei näy testauksen suoritusajoissa, koska se suoritetaan vasta ajanoton lopettamisen jälkeen.



Kuva 4.2: Testausohjelmiston ajonaikainen toiminta.

Kuvassa 4.2 on esitelty testausohjelmiston ajonaikainen toiminta. Ennen asiakasohjelman käynnistämistä on tietokantaprosessien oltava käynnissä palvelinkoneessa. Niinikään säiliö-ohjelman on oltava palvelinkoneessa käynnissä ja EJB-komponenttien on oltava onnistuneesti sijoitettuna säiliöön. Asiakasohjelma lukee käynnistyttyään XML-tiedoston, joka sisältää testauksessa käytettävän materiaalin. Yksi XML-tiedosto sisältää aina tietokannan tauluun siirrettävien rivien lukumäärän verran tietueita, jotka koostuvat otsikko- ja luentatiedoista. Jos testaustapahtumassa on kyseessä käytännön tavan (CASE: Practical way) testaaminen, käynnistää asiakasohjelma ajanoton ja lähettää XML-tiedoston *sEJBInsert*-komponentille. *sEJBInsert* komponentti puolestaan käyttää *sEJBParser* komponentin palvelua tiedon hakuun dokumentista. Jos taas kyseessä on puhdas tapa (CASE: Clean

way) siirtää tietoa tietokantaan, käyttää asiakasohjelma itse *sEJBParser*-komponentin palvelua ja käynnistää ajanoton sen jälkeen. Sitten asiakaohjelma lähettää testausmateriaalin *sEJBInsert*-komponentille. Oppikirjamaisessa tavassa (CASE: Entity bean is used) *sEJBInsert*-komponentti luo jokaista testausmateriaalin sisältämää tietuetta vastaavan ilmentymän *eEJBData*-komponentista ja tämä puolestaan synkronisoidaan tietokannan taulun kanssa säiliön toimesta. Suoraviivaisen tavan (CASE: Entity bean is not used) tapauksessa *sEJBInsert*-komponentti siirtää tietueet tietokantaan suoraan SQL-lausetta käyttäen. Tämän jälkeen *sEJBInsert* palauttaa tiedon onnistumisesta asiakasohjelmalle, joka pysäyttää ajanoton ja kirjoittaa tiedot tapahtumasta lokitiedostoon.

4.3 Asiakasohjelma

Testausohjelmiston asiakasohjelma koostuu kahdesta luokasta. *Main*-luokassa on ainoastaan *main*-funktio, josta koko ohjelma käynnistyy. Jokaisessa testaustapahtumassa asiakasohjelma käynnistetään aina uudelleen ja sille välitetään parametrina sen XML-tiedoston nimi, jota kulloinkin käytetään testin materiaalina. Tiedoston hakemistopolku on määritelty testausohjelman ohjelmakoodissa. Toisena parametrina asiakasohjelmalle välitetään tieto siitä testaustavasta, mikä kulloinkin on kyseessä. *main*-funktion tehtävänä on lukea XML-tiedosto ja muuntaa se *org.w3c.dom.Document* –muotoon käyttäen *oracle.xml.parser.v2.DOMParser* – luokkaa. Tämän jälkeen *main*-funktiosta kutsutaan asiakasohjelman toista luokkaa, *DBInterface*:a. *DBInterface*-luokan pääasiallisina tehtävinä on yhteyksien hallinta palvelinkoneen EJB-komponentteihin, EJB-komponenttien käyttö käyttäjän haluamalla tavalla sekä ajanotto testaustapahtumista. Lisäksi *DBInterface*-luokka kirjoittaa tiedon jokaisesta testaustapahtumasta lokitiedostoon. Asiakasohjelman käytössä on myös kaksi verrattain pientä luokkaa: *ConstValues* ja *eDocument*. *ConstValues*-luokka sisältää joukon vakioita (final-määreellä varustettuja muuttujia), joita käytetään yleisesti testausohjelmistossa. *eDocument*-luokkaa käytetään *org.w3c.dom.Document*:n välittämiseen EJB-komponentille. *org.w3c.dom.Document* –oliota ei voida sellaisenaan välittää, koska se ei toteuta *java.io.Serializable* – rajapintaa (ks. alakohta 2.1.4). Myös osa EJB-komponenteista käyttää näitä pieniä luokkia omissa toiminnoissaan. Asiakasohjelman ohjelmakoodi löytyy liitteestä 1.

4.4 EJB-komponentit

Testausohjelmistossa käytetään kolmea EJB-komponenttia. *sEJBInsert*- ja *eEJBData*-komponentit ovat testauksen varsinaisia kohteita. *sEJBInsert* on tilallinen istuntopohjainen komponentti. S-kirjain nimen edessä tulee sanasta session ja ilmaisee komponentin istuntopohjaisuutta. *eEJBData*-komponentti on testausohjelmiston ainoa kohdepohjainen komponentti ja e-kirjain nimen edessä ilmaisee tätä kohdepohjaisuutta (entity). *eEJBData* on vielä tarkennettuna CMP-komponentti (ks. alakohta 2.2.2). Lisäksi testausohjelmistossa käytetään *sEJBParser*-komponenttia, joka on tilaton istuntopohjainen komponentti. Sitä käytetään kaikissa testitapauksissa, puhtaassa tavassa *DBInterface*-luokan toimesta ja käytännön tavassa *sEJBInsert*-komponentin toimesta.

4.4.1 sEJBInsert

Monipuolisin testausohjelmiston EJB-komponenteista on *sEJBInsert*. Se huolehtii tietokantayhteydestä suoraviivaisen testauksen yhteydessä ja kohdepohjaisista komponenteista oppikirjamaisen testauksen yhteydessä. *sEJBInsert*-komponentti käyttää myös *sEJBParser*-komponentin palveluita aina käytännön tavan yhteydessä. *DBInterface*-luokalle *sEJBInsert*-luokka tarjoaa etärajapintansa kautta neljä palvelua, yhden kutakin testaustapaa kohden. *sEJBInsert*-komponentin etärajapinta on esitelty taulukossa 4.1.

Metodi	**Parametrit**	**Kuvaus**
insert	*eDocument doc*	Metodia käytetään suoraviivaisessa käytännön tavassa siirtää tietoa tietokantaan. Parametri sisältää testausmateriaalin XML-dokumentin muodossa.
insert	*Properties header, Vector data*	Metodia käytetään suoraviivaisessa puhtaassa tavassa siirtää tietoa tietokantaan. Parametrit sisältävät testausmateriaalin otsikkotiedot sekä varsinaisen tietosisällön.

Metodi	Parametrit	Kuvaus
insert_e	*eDocument doc*	Metodia käytetään oppikirjamaisessa käytännön tavassa siirtää tietoa tietokantaan. Parametri sisältää testausmateriaalin XML-dokumentin muodossa.
insert_e	*Properties header, Vector data*	Metodia käytetään oppikirjamaisessa puhtaassa tavassa siirtää tietoa tietokantaan. Parametrit sisältävät testausmateriaalin otsikkotiedot sekä varsinaisen tietosisällön.

Taulukko 4.1: *sEJBInsert*-komponentin etärajapinta

Taulukossa 4.1 esitellään *sEJBInsert*-komponentin etärajapinta. Siitä ilmenee neljä komponentin tarjoamaa palvelua *DBInterface*-luokalle. *Insert*-metodia, jossa parametrina välitetään *eDocument*-luokan ilmentymä, käytetään suoraviivaisen käytännön tavan testauksessa. Siinä *sEJBInsert*-luokka käyttää *sEJBParser*-luokkaa aluksi parametrina saamansa XML-dokumentin purkamiseen. Sen jälkeen tieto siirretään tietokantaan SQL-lausetta käyttäen. *Insert*-metodia, jossa parametrina välitetään luokkien *Properties* ja *Vector* ilmentymät, käytetään suoraviivaisen puhtaan tavan testauksessa. Siinä parametrina saadut tiedot siirretään suoraan tietokantaan SQL-lausetta käyttäen. *Insert_e*-metodin nimen perässä oleva e-kirjain tulee sanasta entity ja se ilmaisee kohdepohjaisen komponentin käyttöä metodia kutsuttaessa. Siinä tieto puretaan XML-muodosta *sEJBParser* komponentin palveluita käyttäen ja viedään tietokantaan *eEJBData*-komponenttien avulla, kun parametrina välitetään *eDokument*-luokan ilmentymä. Kun parametrina välitetään luokkien *Properties* ja *Vector* ilmentymät, viedään tieto kantaan puhtaan menetelmän mukaisesti käyttäen *eEJBData*-komponentteja. *sEJBParser*-komponentin ohjelmakoodi on liitteessä 1.

Taulukossa 4.1 esitellään kolme erityyppistä parametrina välitettävää luokkaa. *eDocument*-luokka on esitelty edellä kohdassa 4.2. *Properties*-luokka on *java.util* – paketin luokka. Se on eräänlainen säiliöluokka, johon voi tallettaa ja josta voi hakea merkkijono-muotoista (*java.lang.String*) tietoa avainsanoja (key attribute) käyttäen. *Properties*-luokkaa käyttäen on kätevää paketoida suuriakin määriä tietoa, joten sitä käytetään tässä tiedon välitykseen. *Vector*-luokka puolestaan toteuttaa muunneltavan kokoisen taulukon, johon se säilöö *java.lang.Object* –tyyppisiä olioita. Koska Javan kaikki luokat ovat periytyneet kyseisestä luokasta, voi *Vector*-luokan ilmentymä

tallettaa minkä tahansa luokan olioita. Testausohjelman tapauksessa *Vector* sisältää *Properties*-luokan oliota, joista kukin sisältää yhden tietueen vaatimat tiedot.

4.4.2 eEJBData

Toinen testausohjelmiston testauksen kohteena oleva EJB-komponentti on *eEJBData*. Kuten edellä mainittiin, *eEJBData*-komponentti on kohdepohjainen CMP-komponentti. Se edustaa pysyvää, tietokannassa säilytettävää tietoa ja tiedon synkronisointi tietokannan taulun kanssa tapahtuu säiliön toimesta. Koska *eEJBData* on kohdepohjainen komponentti, se ei sisällä mitään liiketoimintalogiikkaa (ks. alakohta 2.2.2). *eEJBData*-komponentin ainoa rajapinta-metodi, jota testausohjelmassa aktiivisesti käytetään, on komponentin muodostin (constructor). Aktiivisella käytöllä tässä tarkoitetaan ohjelmoijan kooditasolla tekemiä toimenpiteitä, jotka eivät tapahdu automaattisesti. Toisin sanoen testausohjelmistossa ei ohjelmoijan toimesta kutsuta mitään muuta *eEJBData*-komponentin metodia.

eEJBData-komponentin konstruktori vastaanottaa kuusi parametria. *MPID* (lyhenne sanoista Metering Point ID) on *int* –tyyppinen parametri, jonka arvo viittaa yhteen mittauspisteeseen (Metering Point). *MPID* on *eEJBData*-komponenttia vastaavan tietokantataulun toinen avain ja sen tyyppi on Javan *int*-perustietotyyppi. Toinen avain on *TIMESTAMP*, joka sisältää päivämäärän ja kellonajan mittaustapahtumasta. Vaikka kysessä on ajankohdan ilmoittava tieto, on parametri tyyppiä *java.lang.String*. Tämä johtuu luvussa aiemmin esitetystä yhteensopivuusongelmasta Javan ja Oraclen tietotyyppien ja luokkien välillä (ks. alakohta 3.2.2). Muut muodostimen parametrit ovat *int*–tyyppinen *VALUE* sekä *java.lang.String* –tyyppiset *SENDTIME*, *STATUS* ja *PRODUCT*. *VALUE* parametri sisältää mittauksen arvon ja *SENDTIME* tiedon lähetysajan. *STATUS* ja *PRODUCT* sisältävät tiedot mitatun arvon tilasta tietokannassa sekä kyseessä olevan mittaustuotteen. Kyseisten parametrien ymmärtäminen ei ole tutkielman kannalta tarpeellista, joten niitä ei tässä tarkemmin käsitellä. Testausohjelmiston tietokannan taulussa näiden parametrien nimet esiintyvät taulun kenttinä, niillä on vain eri tietotyyppi (ks. kohta 4.4). *eEJBData*-komponentin ohjelmakoodi on liitteessä 1.

4.4.3 sEJBParser

Alakohdassa 3.2.1 esiteltiin tiedonsiirrolle käytännön tapa, jossa prosessille aiheutetaan viivästyttävä tekijä käsittelemällä tietoa ennen sen siirtämistä tietokantaan. *sEJBParser*-komponentin pääasiallinen käyttötarkoitus on liiketoimintalogiikan toteuttaminen tälle tiedon käsittelylle. Tässä tapauksessa tiedon käsittely sisältää XML-muotoisen tiedon muuttamisen käytännöllisempään muotoon. *sEJBParser*-komponenttia siis käytetään ensisijaisesti *sEJBInsert*-komponentin toimesta testattaessa käytännön tapaa siirtää tietoa tietokantaan. Hajautustuntumattomuuden ansiosta *sEJBParser*-komponentille on myös toinen käyttötarkoitus. Sitä käytetään asiakasohjelman toimesta puhtaan tavan testauksessa ennen ajanoton aloittamista (ks. kuva 4.2). Toteutukseltaan *sEJBParser*-komponentti on hyvin karkea, eikä se kykene toipumaan useimmista virhetilanteista. Se kykenee muokkaamaan vain tässä tutkielmassa käytettyä testausaineistoa, eikä esimerkiksi XML-dokumentin muotoa tarkisteta.

sEJBParser-komponentti tarjoaa yksinkertaisen rajapinnan, jossa on vain yksi metodi nimeltään *parse*. Se vastaanottaa parametrinaan *eDocument*-luokan ilmentymän ja palauttaa *java.util.Vector* –olion. *Vector*-olio sisältää aina yhden *java.util.Properties* – olion sekä *java.util.Vector* –olion. *Properties*-olio sisältää testausaineiston otsikkotiedot ja *Vector*-olio joukon *Properties*-olioita, joista kukin sisältää yhden mittauspisteen mittaustiedon. *sEJBParser*-komponentin ohjelmakoodi löytyy liitteestä 1.

4.4.4 Kuvaustiedosto

Alakohdassa 2.3.1 on lyhyesti esitelty kuvaustiedoston määritys EJB-komponenttimallissa. Mallin määritys on verrattain laaja, eikä sitä ole mielekästä kovin tarkalla tasolla käydä läpi tässä tutkielmassa. EJB-komponenttimallin mukaisesti kuvaustiedosto on aina XML-muotoinen ja sen nimi on aina ejb-jar. Tutkielman kannalta keskeisimmät kuvaustiedoston piirteet ovat komponenttien kuvaukset sekä komponenttien välisten riippuvuuksien kuvaukset. Komponenttien kuvaukset sisältävät säiliölle tarkoitettuja ohjeita komponentin hallinnoinnista.

Komponenttien välisillä riippuvuuksilla tarkoitetaan komponenttien oikeuksia käyttää jonkin toisen komponentin tai ulkoisten resurssien palveluita. CMP-tyyppisten komponenttien tapauksessa kuvaustiedostossa vielä määritellään ne attribuutit, joiden arvot on tarkoitus synkronisoida tietokannan taulun arvojen kanssa. (Monson-Haefel 2001, 473-479 ja 486-488.)

Testausohjelmistossa on toteutettu EJB-komponenteille yhteinen ejb-jar.xml – tiedosto, jossa esitellään komponenttien tyypit sekä komponenttien väliset riippuvuudet. Riippuvuudella tässä tarkoitetaan komponentin oikeutta käyttää toteutuksessaan hyödykseen toisen komponentin palveluita. Tiedostoon on määritelty *sEJBInsert*-komponentille oikeus käyttää toteutuksessaan sekä *sEJBParser*- että *eEJBData*-komponentteja. *sEJBParser*-komponentille ei ole määritelty tiedostossa käyttöoikeuksia mihinkään muuhun komponenttiin. *eEJBData*-komponentille on määritelty kuusi alakohdassa 4.3.2 esiteltyä attribuuttia, sekä oikeus käyttää säiliön kannalta ulkoista resurssia, tietokantaa. Attribuuttien esittäminen on tiedostossa tarpeellista, jotta säiliö osaa synkronisoida komponentin tilan tietokannan tietojen kanssa. Tietokanta on resurssina määritelty säiliön sisäiseen ohjaustiedostoon, jonka nimi on data-sources.xml. Tiedoston käyttöohjeet löytyvät lähteestä Orion (2001).

4.5 Tietokanta

Kolmannessa luvussa mainittiin tietokannan kuuluvan testausohjelmistossa niihin komponentteihin, joiden vaikutus testausprosessiin pyritään vakioimaan ja prosessointi tietokannassa minimoimaan. Siinä mainittiin myös tiedonsiirtotavan vaikuttavan viiveeseen, jonka tietokanta aiheuttaa testausprosessille. Eri tiedonsiirtotapojen vaikutusten todettiin olevan testattavien komponenttien, ei tietokannan, ominaisuuksia. Lisäksi todettiin vielä, että tietokannan taulu tyhjennetään testaustapahtumien välillä, jotta taulun koko ei pääsisi vaikuttamaan testausprosessiin. Testausohjelmistossa käytetään Oracle8i tietokannan versiota 8.1.7 ja siinä käytetään ainoastaan yhtä taulua. Tällä tavalla on rajattu tietokannassa tapahtuva prosessointi ainoastaan avainkenttien oikeellisuuden tarkastamiseen. Koska testaustapahtumissa keskenään vertailtavat tulokset on saatu aina samanmääräisen datan käsittelystä ja koska tietokannan kuluttaman ajan avaimen tarkistamiseen voidaan arvioida riippuvan

lineaarisesti testiaineiston määrästä, voidaan todeta, ettei avaimen tarkistaminen vaikuta testaustulosten vertailukelpoisuuteen.

Tietokantataulun kenttä	**Kentän tyyppi**	**EJB-komponentin attribuutti**	**Attribuutin tyyppi**
MPID	*NUMBER(3)*	MPID	*int*
TIMESTAMP	*VARCHAR2(25)*	TIMESTAMP	*java.lang.string*
PRODUCT	*VARCHAR2(12)*	PRODUCT	*java.lang.string*
VALUE	*NUMBER(8)*	VALUE	*int*
SENDTIME	*VARCHAR2(25)*	SENDTIME	*java.lang.string*
STATUS	*VARCHAR2(4)*	STATUS	*java.lang.string*

Taulukko 4.2: *metering_data* -taulun kentät ja niiden tietotyypit sekä niitä vastaavat attribuutit ja attribuuttien tietotyypit EJB-komponentissa.

Taulukossa 4.2 esitellään *metering_data* –tietokantataulun kentät tietotyyppeineen sekä niitä vastaavat attribuutit ja niiden tietotyypit *eEJBData*-komponentissa. Kenttien tietotyyppien määrityksessä on noudatettu alakohdassa 3.2.2 esiteltyä linjaa, jonka mukaan Javan *int*-tietotyyppiä vastaava tietotyyppi tietokannassa on *NUMBER* ja *java.lang.String* –tietotyyppiä vastaava tietotyyppi tietokannassa on *VARCHAR2*. *metering_data*-taulun kenttien koot ovat: MPID-kentässä 3-numeroinen luku, TIMESTAMP-kentässä 25:n merkin merkkijono, PRODUCT-kentässä 12:n merkin merkkijono, VALUE-kentässä 8-numeroinen luku, SENDTIME-kentässä 25:n merkin merkkijono ja STATUS-kentässä neljän merkin merkkijono.

4.6 Ympäristö

4.6.1 Oracle9iAS palvelin

Oracle9i Application Server (Oracle9iAS) on laaja tuoteperhe, joka tarjoa palvelinalustan muun muassa sähköisen kaupankäynnin toteutuksille. Oracle9iAS Container for J2EE (OC4J) on yksi osa tätä tuoteperhettä ja se sisältää säiliön palvelut

EJB-komponenteille. OC4J on tyypillinen EJB-komponenttien säiliöratkaisu, jossa säiliön ja palvelimen tehtäviä ei ole eritelty (ks. alakohta 2.1.1). OC4J perustuu Orionin tekemään palvelimeen (ks. Orion 2001). Tutkielman testauksessa käytetty OC4J:n versio 1.0.2.2 tukee täysin EJB-komponenttimallin version 1.1 määritystä sekä joitain osia version 2.0 määrityksestä. Tutkielman kannalta merkittäviä OC4J:n toimintoja ovat JTA:n versio 1.0.1, JNDI:n versio 1.2 sekä JDBC:n versio 2.0. (Oracle 2001, 2.)

OC4J on yksi monista säiliön sekä palvelimen palvelujen tarjoavista ohjelmistoista EJB-komponenteille. Palvelimien toteutuksissa on merkittäviä eroja ainakin vaadittavan muistitilan suhteen. Periaatteessa tässä tutkielmassa olisi voitu käyttää mitä tahansa palvelinta, joka toteuttaa EJB-komponenttimallin version 1.1 määrityksen. Tutkielman liittyminen tekijän nykyisiin työtehtäviin kuitenkin vaatii juuri tämän palvelimen käyttöä testauksessa.

4.6.2 Laitteisto

Testausohjelmiston asiakasohjelma ajetaan 400 megahertsin prosessorilla varustetulla pc-koneella (personal computer), jossa keskusmuistia on 384 megatavua. Käyttöjärjestelmänä koneessa on Microsoftin valmistaman Windows NT:n versio 4.0, johon on päivitetty Service Pack versio 6. Tästä koneesta käytetään myöhemmin nimitystä asiakaskone. Toinen testauksessa käytettävä pc-kone on varustettu 733 megahertsin prosessorilla ja siinä on keskusmuistia 256 megatavua. Siinä on myös käyttöjärjestelmänä Microsoft:n Window NT:n versio 4.0, johon on päivitetty Service Pack versio 6. Tässä koneessa ajetaan tietokannan prosessit sekä palvelinohjelmisto säiliöineen ja siitä käytetään myöhemmin nimitystä palvelinkone. Lähiverkkona testauksessa käytetään sadan megabitin sekuntinopeuden mahdollistavaa lähiverkkoa. Sekä asiakas- että palvelinkoneessa verkkokortti toimii sadan megabitin sekuntinopeudella.

5 TESTITAPAUKSET JA TESTAAMINEN

Kolmannessa luvussa esiteltiin testauksen viitekehys ja neljännessä luvussa testausympäristön arkkitehtuuri. Tässä luvussa esitellään perusteet testimateriaalin valinnalle ja kuvataan testaustapahtumien kulku. Lopussa kerrotaan testauksessa tehdyistä havainnoista sekä esitellään testin tulokset.

5.1 Testitapaukset

Kohdassa 3.2 määriteltiin testauksen viitekehys. Siinä määriteltiin muun muassa käytettävät tietotyypit testausohjelmistossa ja tietokannassa. Edelleen siinä määriteltiin, että mittauksen yksikkönä testauksessa käytetään yhtä tietokannan riviä. Määritelmän mukaisesti testausohjelman puolella kaksi *int*-tyyppistä ja neljä *java.lang.String* –tyyppistä arvoa muodostavat yhden rivin, joka tietokannan puolella konvertoituu kahdeksi *NUMBER*- ja neljäksi *VARCHAR2*-tyyppiseksi arvoksi. Tässä luvussa esitellään lähtökohdat testausmateriaalin rivimäärien valinnalle. Lisäksi tässä määritellään edellä kuvatun määritelmän mukaisten rivien määrät, joita testauksessa tietokantaan viedään, sekä kuvataan millaisella kaavalla kukin testaustapahtuma viedään läpi.

5.1.1 Testausmateriaalin valinnan lähtökohdat

Testauksen keskeisimpänä mielenkiinnon kohteena ovat EJB-komponenttien suorituskykyerot eri tietomäärissä, erityisesti suurissa tietomäärissä. Koska tiedon määrän yksiköksi on jo määritelty yksi rivi, on tässä luvussa tarkoitus määritellä millaisia rivimääriä testataan. Vastaavan kaltaista testiä ei tutkielman kirjoittamisen ajankohtana ollut kohtuullisessa ajassa löydettävissä, joten sopivien rivimäärien löytäminen on ollut arvaamisen ja kokeilun varassa. Kuitenkin erilaisia testattavia rivimääriä on saatava mukaan riittävän kattava kokoelma, jotta voidaan tuloksien perusteella tehdä tulkinta, millaisissa tietomäärissä alakohdassa 3.2.2 esiteltyjen tiedonsiirtotapojen ero käy merkittäväksi. Näin ollen tietomäärien valinnassa on

päädytty ratkaisuun, jossa tietokantarivien määrää kasvatetaan, kunnes edellä mainitun kaltaisen tiedonsiirtotapojen suorituskykyeron voidaan tulkita olevan merkittävä.

Mitta-asteikkona testauksessa käytetään suhdeasteikkoa, jossa mittaustulos on minimissään nolla ja maksimissaan n sekunnin sadasosaa (s/100). Mitta-asteikko on tässä tapauksessa suhdeasteikollinen, koska sen minimikohta on asteikkoon nähden absoluuttinen nollapiste ja koska saatujen arvojen suhteellinen muutos on määriteltävissä (Riukulehto & Huhtala, 1992, 7). Koska kyseessä on suurten tietomassojen testaus, on katsottu pienimmäksi tiedon määrän kasvatusväliksi kahdenkymmenen tietokantarivin olevan sopiva määrä testauksen alussa. Siirryttäessä suurempiin tietomääriin nostetaan tiedon määrän kasvatusväliksi sata tietokantariviä. Tällä tavalla saadaan tietoa suorituskykyerojen lisäksi siitä, miten suorituskykyjen ero muodostuu ja onko samankaltaisia suorituskykyjen muutoksia nähtävissä myös pienemmillä testausmäärillä. Keskeisin mielenkiinnon kohde on ison kasvatusvälin testaus. Siitä testauksessa hankitaan riittävästi materiaalia, jotta voidaan nähdä millaisen kaavan mukaisesti kussakin testaustavassa suorituskyky hidastuu tiedon määrän kasvaessa.

Edellä mainittiin, että testauksessa tietokantarivien määrää kasvatetaan, kunnes suorituskykyjen eron voidaan tulkita olevan merkittävä. Suorituskykyjen eron merkittävyyden tulkinta on kuitenkin ongelmallista. Tämän eron merkittävyyden määrä on aina suhteessa siihen, kuinka paljon kulloinkin arvostetaan suorituskyvyn nopeutta ja kuinka paljon hitaamman tavan tuomia muita hyötyjä. Näin ollen saman suorituskykyjen eron voidaan tulkita näkökulmasta riippuen olevan esimerkiksi hyvin merkittävä tai ei lainkaan merkittävä. Tässä tutkielmassa EJB:n eri käyttötapojen eron katsotaan olevan merkittävä siinä vaiheessa, kun testituloksista on tilastollisesti ennustettavissa sen muodostuvan mistä tahansa näkökulmasta katsottuna merkittäväksi siirrettävien tietomäärien kasvaessa riittävän suureksi. Tässä ei siis pyritä löytämään jotain tiettyä tietokannan rivimäärää, jonka jälkeen käyttötapojen suorituskykyjen ero muodostuisi merkittäväksi. Sen sijaan EJB:n eri käyttötapojen suorituskykyjen eron merkittävyydeksi riittää todistus siitä, että tämä ero kasvaa siirryttäessä isompiin tietomääriin.

5.1.2 Testausmateriaalin esittely

Kuten edellä mainittiin, kohdassa 3.2 määriteltiin mittauksen yksikkönä käytettävän yhtä tietokannan riviä, joistä kukin sisältää kuusi kenttää. Tyypillisesti käytännössä arvot kenttiin tämänkaltaisessa tiedonsiirrossa haetaan isommasta tietomassasta, joka sisältää myös sellaista tietoa, jota ei ole tarkoitus siirtää tietokantaan (ks. luku 4). Perusteena tällaisen näennäisesti ylimääräisen tiedon olemassaololle on käytännön näkökulma tiedon siirrossa. Siinä pyritään saamaan aikaan tyypillinen käytännön malli siirrettäessä tietoa tietokantaan (alakohta 3.2.2). Luvun neljä alussa mainittiin testausohjelmiston rakenteen lähtökohtana olevan järjestelmä, jossa energiamittareista etäluettua mittaustietoa siirretään tietokantaan. Tässä tutkielmassa käytetty testausmateriaali voisi muotonsa puolesta olla muuttamattomana tällaista mittaustietoa ja tämä näkyy erityisesti kenttien nimeämiskäytännöissä. Testimateriaalia siirretään eri komponenttien välillä testausohjelmiston sisällä pääasiassa XML-muodossa. Alla olevassa taulukossa esitellään testissä käytetyn materiaalin mahdolliset tietokentät ja niiden arvot. Esimerkki XML-muotoisesta testausmateriaalista löytyy liitteestä 2.

Kentän nimi	**Mahdolliset arvot**	**Kuvaus**
meteringdata	-	Koko mittaustieto yhden konsentraattorin alaisuudessa olevista mittareista.
header	-	Mittaustiedon otsikkotiedot.
msgtype	mass data | ...	Viestin tunniste
sendtime	**Pvm & aika**	**Lähetysaika konsentraattorilta**
concentrator	Numero	Konsentraattorin osoite
commserver	Numero	Kommunikaatiopalvelimen osoite
mp	-	Mittauspisteen tiedot
meteringpoint	**Numero**	**Mittauspisteen osoite**
product	**series | single value**	**Mittauksen tyyppi**
measuretype	electricity | gas | heat	Mittaustiedon tyyppi
unit	var | kWh | ...	Mittaustiedon tietotyyppi
pulse_constant	Numero	Pulssivakio
transformer	Numero	Mittamuuntajakerroin
measurement	-	Yhden mittauspisteen mittauksista saadut

Kentän nimi	Mahdolliset arvot	Kuvaus
		tiedot
mvalue	**Numero**	**Mitattu arvo**
timestamp	**Pvm & aika**	**Mittauksen ajankohta**
status	**OK | NOK**	**Mittaustiedon validiteetin hyväksyntä**

Taulukko 5.1: Testausmateriaalissa käytetyt kentät.

Taulukossa 5.1 on esitelty testausmateriaalin kentät siten, että tietokannan rivissä käytetyt kentät materiaalista on esitetty tummennettuina. Tummentamattomina esitellyt kentät eivät ole tutkielman kannalta kiinnostavia ja niiden esittely jätetään sille tasolle, kuin ne on taulukossa esitelty. Myöskään tummennettuina esiteltyjen kenttien tarkempi sisältö ei ole tutkielman kannalta kiinnostavaa, joten niitäkään ei tässä tarkemmin esitellä. Alakohdassa 3.2.3 määriteltiin tutkielmassa käytettävät tietotyypit. Taulukon 5.1 kentistä *meteringpoint-* ja *mvalue*-kentät ovat kolmannen luvun määritelmän mukaisesti testausohjelmiston Java-kielellä ohjelmoidulla puolella tietotyyppiä *int* ja tietokannan puolella tietotyyppiä *NUMBER*. Edelleen *sendtime-*, *product-*, *timestamp-* ja *status*-kentät ovat Java-kielellä ohjelmoidulla puolella tietotyyppiä *java.lang.String* ja tietokannan puolella tietotyyppiä *VARCHAR2*. Tietokannan avainkenttinä käytetään testausmateriaalin kenttiä *meteringpoint* sekä *timestamp*. Vaikka testausmateriaali sisältää muutamia testauksen kannalta ylimääräisiä kenttiä, ei tällä ole merkittävää vaikutusta testauksen tulosten vertailukelpoisuuteen.

5.1.3 Testitapausten esittely

Alakohdassa 5.1.1 mainittiin, että testauksen alussa tietokantaan siirrettävien rivimäärien kasvatusväli on alussa kaksikymmentä ja myöhemmin sata tietokantariviä. Kahdenkymmenen rivin kasvatusväliä käytetään testauksessa sataan tietokantariviin saakka. Siitä eteenpäin tietokantarivien määrää kasvatetaan sadalla aina kahdeksaan sataan tietokantariviin saakka. Tutkielman testauksessa jonkin tietyn rivimäärän testausta kutsutaan testaustasoksi. Näin ollen testaustasoja testauksessa on kaiken kaikkiaan kaksitoista. Kullakin testaustasolla tapahtuu testaus neljällä eri

tavalla: suoraviivainen puhtaan-, suoraviivainen käytännön-, oppikirjamainen puhtaan- sekä oppikirjamainen käytännön tavan testaus (ks. alakohta 3.2.2). Yhden tavan testausta tietyllä testaustasolla kutsutaan tässä tutkielmassa testitapaukseksi. Testitapauksia tutkielman testauksessa on siten kaiken kaikkiaan neljäkymmentäkahdeksan. Myöhemmin tutkielmassa käytetään termiä pienen mittakaavan testaus viitattaessa testitapauksiin testaustasoilla kahdestakymmenestä tietokantarivistä sataan tietokantariviin ja termiä ison mittakaavan testaus viitattaessa testitapauksiin testaustasoilla sadasta tietokantarivistä kahdeksaansataan tietokantariviin.

5.2 Testaustapahtumat

Edellisessä alakohdassa määriteltiin testitapauksen käsite. Tässä alakohdassa esitellään testaustapahtuma, vieraat havainnot, vieraiden havaintojen tunnistamiseen käytettävät keinot sekä niiden aiheuttamat toimenpiteet.

5.2.1 Testaustapahtuman käsite

Yhden testitapauksen testaamista kutsutaan tässä tutkielmassa testaustapahtumaksi. Tutkielman testauksessa käytettäviä työkaluja ovat testausohjelmisto, OC4J sekä T.O.A.D.-tietokannan hallintatyökalu. Lisäksi käytetään MS Excel –taulukko-ohjelmaa tulosten taulukointiin. Jokainen testaustapahtuma aloitetaan kohteena olevan tietokantataulun tyhjentämisellä ja tapahtuman vahvistamisella T.O.A.D.-ohjelmasta käsin. Tietokantataulun tyhjentämisellä vakioidaan, tai ainakin minimoidaan, tietokannan vaikutus testausprosessiin (ks. alakohta 3.2.3). Sen jälkeen OC4J pysäytetään ja käynnistetään uudelleen. Tämä tapahtuu kahdesta syystä. OC4J varastoi dynaamiseen muistiin tiettyjä EJB-komponentteja, jotta niiden käyttöönotto olisi komponentin seuraaville käyttäjille nopeampaa. Tämä aiheuttaisi todennäköisesti vääristymiä testaustuloksiin. Toinen syy on tietokannan ja testissä käytettävien kohdepohjaisten EJB-komponenttien synkronisointi. Kun tietokannan taulu tyhjätään, voi OC4J:n hallinnoimaan muistiin jäädä sellaisia kohdepohjaisia komponentteja, jotka edustavat poistettuja tietokannan rivejä. Tämä voi aiheuttaa ennustamattomia

virheitä testauksessa. Kolmas toimenpide testaustapahtumassa on testausohjelmiston ajaminen. Lopuksi vielä T.O.A.D.-ohjelmasta käsin tarkistetaan tiedonsiirron onnistuminen ja rivimäärän oikeellisuus.

5.2.2 Vieraat havainnot

Tilastotieteessä on yleisesti käytössä vieraan havainnon käsite. Sillä tarkoitetaan tutkittavasta aineistosta havaittavia tulkinnan kannalta epäilyttäviä tuloksia. Syy tällaisiin tuloksiin voi löytyä mittausvirheistä, ilmiön aidosta erikoislaatuisuudesta tai aineistoon vahingossa joutuneesta arvosta tai havainnosta. Toimenpiteenä tällainen tulos aiheuttaa havainnon syyn etsimisen ja virheen korjaamisen. Jos syytä ei löydetä toimitaan tilanteen mukaan, esimerkiksi jättäen tulos huomioimatta. Havainnon tunnistamisessa on keskeisimmässä asemassa tutkittavan ilmiön tuntemus. (Kankainen & Högmander, 2001, 26-27.)

Alakohdassa 5.2.1 esitelty kuvaus sisältää testaustapahtumassa tehtävät toimenpiteet. Kokonaisuudessaan jokainen testaustapahtuma sisältää viisi alakohdassa 5.2.1 kuvattua neljän toimenpiteen iteraatiota. Näin jokaista testitapausta kohden saadaan viisi arvoa, joista kyseessä olevan testitapauksen arvoksi lasketaan keskiarvo. Ennen keskiarvon laskemista kuitenkin tarkastetaan, etteivät nämä viisi arvoa poikkea merkittävästi toisistaan. Jos viidestä saadusta arvosta suurimman ja pienimmän välinen erotus on pienen mittakaavan testauksessa enemmän kuin kaksikymmentä prosenttia kaikkien viiden arvon keskiarvosta, katsotaan tällaisen poikkeaman olevan vieras havainto. Vastaava prosenttimäärä ison mittakaavan testauksessa on viisitoista prosenttia. Tällaisissa tapauksissa testaustapahtuma uusitaan. Tässä tutkielmassa käytetään termiä poikkeama yhteydessä, jossa viitataan testaustapahtuman uusimisen aiheuttavaan poikkeamaan Tällöin poikkeamalla tarkoitetaan vierasta havaintoa.

Kuten aiemmin mainittiin, kaikkien ei-vertailtavien komponenttien suoritusaika pyritään testausohjelmistossa vakioimaan. Edelleen mainittiin, että jos tällaisen komponentin vaikutusta ei voida vakioida, pyritään sen vaikutus minimoimaan (ks. kohta 3.2). Edellä mainitun mahdollisen poikkeaman arvioitiin aiheutuvan niiden testausohjelmiston ei-vertailtavien komponenttien toiminnasta, joiden toiminta

kyettiin ainoastaan minimoimaan. Edelleen tällaisten ei-vertailtavien komponenttien toiminnasta syntyy testausohjelmistolle vähimmäissuoritusaika, joka kuluu testausohjelmiston ajamiseen, vaikka tietokantaan ei tietoa vietäisikään. Suhteessa vertailtavien komponenttien suoritusaikaan on tämä vähimmäissuoritusaika luonnollisestikin suurempi pienillä testausmäärillä, kuin suurilla. Tästä johtuen tutkielman testauksessa sallitaan pienen mittakaavan testauksessa hieman suurempi poikkeama kuin suuren mittakaavan testauksessa.

5.3 Testin tulokset

Kohdassa 5.1 määriteltiin testissä käytettävä aineisto ja kohdassa 5.2 testaustapahtumat toimenpiteineen. Tässä kohdassa kuvataan testauksen eteneminen, siinä tehdyt havainnot sekä testin tulokset.

5.3.1 Havaintoja testauksessa

Testausohjelmisto ylläpitää tiedostoa, josta testistä saadut tulokset on luettavissa (ks. kohta 4.3). Tulokset saadaan millisekunnin (ms) tarkkuudella. Testauksen edetessä kuitenkin havaittiin, että saadun aika-arvon viimeinen, millisekuntia osoittava numero ei ollut luotettava. Se oli alussa yksi, ja kasvoi lähes säännönmukaisesti testaustasojen kasvaessa. Sen sijaan muissa aika-arvon numeroissa ei havaittu vastaavan kaltaista tai muutakaan epäluotettavuuteen viittaavaa säännönmukaisuutta. Koska testauksen kohteena on EJB-komponenttien suorituskyvyn testaus nimenomaan suurilla testausmäärillä, todettiin millisekunnin tarkkuuden sijaan sekunnin sadasosan tarkkuuskin riittäväksi. Näin ollen testauksen tarkkuustasoksi muutettiin sekunnin sadasosa ja millisekuntia osoittava numero jätettiin kokonaan noteeraamatta. Sitä ei siis käytetty edes pyöristyksiin. Tarkkaa selitystä moiselle ilmiölle on hankala löytää, eikä se tutkielman kannalta ole kovin merkittävääkään. Yksi mahdollinen selitys voisi olla virhe J2SE:n *System*-luokan metodissa *currentTimeMillis*, jota testauksen ajanotossa käytettiin.

Toinen mielenkiintoinen havainto liittyy tulosten poikkeavuuteen toisistaan. Usein merkittävän mittaisen tauon jälkeen aloitettaessa testaaminen uudelleen, oli ensimmäisen testitapauksen ensimmäinen viidestä arvosta merkittävästi toisia suurempi. Merkittävän mittainen tauko oli testaajan saaman tuntuman mukaan muutamia minuutteja. Tämä aiheutti aina testaustapahtuman hylkäämisen ja uusimisen. Kuten edelliseen, myöskin tähän ilmiöön on vaikea löytää perusteltavissa olevaa syytä. Mahdollisesti jokin järjestelmän osa testausohjelmiston asiakasohjelman ja tietokannan välillä ylläpitää tietyn ajan parempaa toimintavalmiutta, jonka loppumisen jälkeen tietoliikenteen ja/tai jonkin palvelun aloittaminen uudelleen on hieman hitaampaa. Tälläkään havainnolla ei kuitenkaan arvioitu olevan testauksen lopullisten tulosten kannalta merkitystä, koska virheelliset tulokset saatiin aina korjattua testaustapahtumia uusimalla.

5.3.2 Testauksen eteneminen

Edellä mainittiin testauksen sisältävän neljä erityyppistä testitapaa, joista kukin testataan kahdellatoista eri testaustasolla (ks. alakohta 5.1.3). Tutkielman testauksessa edettiin testaustaso kerrallaan siten, että käytiin läpi aina neljä testitapausta aloittaen suoraviivaisen puhtaan tavan testauksesta päättyen oppikirjamaisen käytännön tavan testaukseen. Tämän jälkeen testausohjelmiston ylläpitämästä lokitiedostosta haettiin testin tulokset ja siirrettiin ne MS Excel –taulukko-ohjelmaan. Testaustasoissa aloitettiin pienimmästä ja edettiin järjestyksessä suurimpaan testaustasoon saman kaavan mukaisesti. Lopuksi MS Excel –taulukko-ohjelmassa laskettiin jokaiselle testitapaukselle viiden saadun arvon keskiarvo sekä suurimman ja pienimmän arvon erotuksen prosentuaalinen poikkeama keskiarvosta.

5.3.3 Tulokset

Edellä mainittiin, että testitulosten poikkeavuuksissa havaittiin tiettyä säännönmukaisuutta testitapauksen aloittamisen ajankohtaan liittyen (ks. alakohta 5.3.1). Edelleen myöhemmin mainittiin, että testaus suoritettiin neljän testitapauksen sykleissä siten, että aina välillä saadut arvot siirrettiin MS Excel –taulukko-ohjelmaan

(ks. alakohta 5.3.2). Testitapausten ensimmäisen läpikäynnin jälkeen voitiin havaita testauksen etenemiskäytännön vaikuttaneen testaustuloksiin. Melko säännönmukaisesti suoraviivaisen puhtaan tavan testitapauksissa ensimmäinen viidestä mitatusta arvosta aiheutti tuloksen hylkäämisen aiheuttavan poikkeaman. Lisäksi suoraviivaisen puhtaan tavan testaustulosten poikkeamien keskiarvo oli yli viisitoista prosenttia, kun muissa testaustavoissa sama keskiarvo oli kuuden prosentin paikkeilla. Testitapausten ensimmäisen läpikäynnin tulokset on esitelty kuvassa 5.1.

	data20	data40	data60	data80	data100	data200	data300	data400	data500	data600	data700	data800	
Suora-	95	98	128	129	118	145	189	217	274	260	271	380	
viivainen -	94	99	102	115	119	141	168	206	229	256	261	306	
Puhdas	98	101	103	108	120	137	152	231	221	220	261	306	
Tapa	94	87	102	110	126	125	171	206	255	228	260	316	
	95	103	117	109	114	147	151	209	219	236	262	309	
Keskiarvo	**95,2**	**97,6**	**110,4**	**114,2**	**119,4**	**139**	**166,2**	**213,8**	**239,6**	**240**	**263**	**323,4**	
Poikkeama	**0,04**	**0,16**	**0,24**	**0,18**	**0,10**	**0,16**	**0,23**	**0,12**	**0,23**	**0,17**	**0,04**	**0,23**	0,16
			x			x	x		x	x		x	
Oppi-	104	151	172	168	203	266	352	420	481	572	611	704	
kirjamainen -	105	152	172	175	189	267	354	443	503	570	611	707	
Puhdas	105	151	177	172	198	270	360	426	494	567	601	677	
Tapa	105	151	163	167	195	252	326	431	486	535	662	688	
	109	150	164	175	205	273	341	422	481	537	619	703	
Keskiarvo	**105,6**	**151**	**169,6**	**171,4**	**198**	**265,6**	**346,6**	**428,4**	**489**	**556,2**	**620,8**	**695,8**	
Poikkeama	**0,05**	**0,01**	**0,08**	**0,05**	**0,08**	**0,08**	**0,10**	**0,05**	**0,04**	**0,07**	**0,10**	**0,04**	0,06
Suora-	130	154	170	183	207	252	286	358	435	444	546	584	
viivainen -	128	135	176	187	199	243	282	384	414	452	542	577	
Käytännön	128	137	169	187	196	241	277	354	414	489	553	579	
Tapa	127	137	176	186	199	241	295	347	418	485	564	605	
	128	135	167	187	207	259	296	359	418	495	542	594	
Keskiarvo	**128,2**	**139,6**	**171,6**	**186**	**201,6**	**247,2**	**287,2**	**360,4**	**419,8**	**473**	**549,4**	**587,8**	
Poikkeama	**0,02**	**0,14**	**0,05**	**0,02**	**0,05**	**0,07**	**0,07**	**0,10**	**0,05**	**0,11**	**0,04**	**0,05**	0,06
Oppi-	152	183	228	258	269	375	467	569	677	786	881	1012	
kirjamainen -	154	172	223	245	271	382	471	589	676	780	899	989	
Käytännön	152	172	229	245	265	400	467	558	686	754	879	956	
Tapa	151	171	232	250	262	373	482	579	717	771	891	986	
	152	172	208	231	270	391	468	557	679	783	891	977	
Keskiarvo	**152,2**	**174**	**224**	**245,8**	**267,4**	**384,2**	**471**	**570,4**	**687**	**774,8**	**888,2**	**984**	
Poikkeama	**0,02**	**0,07**	**0,11**	**0,11**	**0,03**	**0,07**	**0,03**	**0,06**	**0,06**	**0,04**	**0,02**	**0,06**	0,06

Kuva 5.1: Testitapausten ensimmäisen läpikäynnin tulokset.

Kuvassa 5.1 on taulukoitu testitapausten ensimmäisen läpikäynnin tulokset. Kaavion tulokset on saatu suorittamalla testaus alakohdassa 5.3.2 kuvatulla tavalla.Ylimmällä rivillä ilmoitetaan testaustaso tietokannan rivimäärinä. Jokaisen testaustyypin tulokset ovat omina seitsemän rivin ryhminään siten, että viivan yläpuolella on viisi arvoa kukin omalla rivillään. Viivan alapuolella on näiden viiden arvon keskiarvo sekä suurimman ja pienimmän arvon erotuksen suhteellinen poikkeama keskiarvosta. Poikkeama on ilmoitettu desimaalilukuna välillä [0,1] ja se on muutettavissa prosenttimääräiseksi kertomalla sadalla. Kaavion oikeassa reunassa on omassa sarakkeessaan ilmoitettu kunkin testaustavan poikkeamien keskiarvo myöskin

desimaalilukuna välillä [0,1]. Lisäksi kaavioon on merkattu x-kirjain jokaisen sellaisen testitapauksen alle, jonka poikkeama aiheuttaa hylkäämisen. Kaaviosta on helposti havaittavissa edellä mainittu säännönmukainen poikkeaminen suoraviivaisen puhtaan tavan testaustapauksissa. Kaaviosta löytyy kuusi liian suuren poikkeaman vuoksi hylättävää testitulosta ja ne kaikki ovat suoraviivaisen puhtaan testaustavan testitapauksia. Myöskin alle viiden prosenttiyksikön päähän hylkäämisestä jääneitä testitapauksia kyseisestä testaustavasta löytyy neljä kappaletta. Koska poikkeamien säännönmukaisuus sekä poikkeamien keskiarvojen ero oli näin merkittävä, tehtiin päätös pudottaa poikkeuksellisesti hylkäysrajaa viidellä prosenttiyksiköllä suoraviivaisen puhtaan tavan tapauksessa sekä pienen että ison mittakaavan testauksessa. Hylkäysrajan tiukentaminen on tässä kuitenkin väliaikaista ja aiemmin mainitut hylkäysrajat pysyvät edelleen voimassa myöhemmin mahdollisesti ilmenevien satunnaisten poikkeamien varalta. Näin ollen uudelleen testattavia testitapauksia on kaikkiaan kymmenen kappaletta.

Uusintatesti käytiin läpi aiemmasta hieman poiketen. Testin alussa tehtiin neljänkymmenen tietokantarivin testaustasolla kolme ylimääräistä tiedon syöttöä, jotta edellä kuvattu poikkeama ei toistuisi tässäkin testissä. Näistä ylimääräisistä tiedon syötöistä saatuja tuloksia ei noteerattu lopullisissa tuloksissa. Mainittakoon kuitenkin, että niistä ensimmäinen olisi jälleen aiheuttanut testitapauksen hylkäämisen, joten alakohdassa 5.3.1 esitetty hypoteesi poikkeaman syystä sai hieman vahvistusta. Uusittavat testitapaukset käytiin läpi järjestyksessä aloittaen neljänkymmenen rivin testaustasolta ja päättyen kahdeksansadan rivin testaustasoon. Testin välissä ei tietoja viety MS Excel –taulukko-ohjelman taulukkoon, vaan testi suoritettiin yhtäjaksoisena. Tulokset siirrettiin MS Excel –taulukko-ohjelmaan testin päätyttyä. Uusitun testin tulokset eivät poikkeamien puolesta antaneet enää aihetta uusiin testeihin. Lopulliset tulokset on listattu kuvassa 5.2.

	data20	data40	data60	data80	data100	data200	data300	data400	data500	data600	data700	data800	
Suora-	95	117	122	129	132	142	152	188	204	216	271	307	
viivainen -	94	114	112	121	124	141	171	195	206	212	261	287	
Puhdas	98	114	116	129	125	128	169	179	229	218	261	281	
Tapa	94	113	114	122	127	127	151	178	203	240	260	303	
	95	113	114	129	126	138	152	185	209	241	262	304	
Keskiarvo	**95,2**	**114,2**	**115,6**	**126**	**126,8**	**135,2**	**159**	**185**	**210,2**	**225,4**	**263**	**296,4**	
Poikkeama	**0,04**	**0,04**	**0,09**	**0,06**	**0,06**	**0,11**	**0,13**	**0,09**	**0,12**	**0,13**	**0,04**	**0,09**	0,08
Oppi-	104	151	172	168	203	266	352	420	481	572	611	704	
kirjamainen -	105	152	172	175	189	267	354	443	503	570	611	707	
Puhdas	105	151	177	172	198	270	360	426	494	567	601	677	
Tapa	105	151	163	167	195	252	326	431	486	535	662	688	
	109	150	164	175	205	273	341	422	481	537	619	703	
Keskiarvo	**105,6**	**151**	**169,6**	**171,4**	**198**	**265,6**	**346,6**	**428,4**	**489**	**556,2**	**620,8**	**695,8**	
Poikkeama	**0,05**	**0,01**	**0,08**	**0,05**	**0,08**	**0,08**	**0,10**	**0,05**	**0,04**	**0,07**	**0,10**	**0,04**	0,07
Keskiarvo													
Suora-	130	154	170	183	207	252	286	358	435	444	546	584	
viivainen -	128	135	176	187	199	243	282	384	414	452	542	577	
Käytännön	128	137	169	187	196	241	277	354	414	489	553	579	
Tapa	127	137	176	186	199	241	295	347	418	485	564	605	
	128	135	167	187	207	259	296	359	418	495	542	594	
Keskiarvo	**128,2**	**139,6**	**171,6**	**186**	**201,6**	**247,2**	**287,2**	**360,4**	**419,8**	**473**	**549,4**	**587,8**	
Poikkeama	**0,02**	**0,14**	**0,05**	**0,02**	**0,05**	**0,07**	**0,07**	**0,10**	**0,05**	**0,11**	**0,04**	**0,05**	0,06
Oppi-	152	183	228	258	269	375	467	569	677	786	881	1012	
kirjamainen -	154	172	223	245	271	382	471	589	676	780	899	989	
Käytännön	152	172	229	245	265	400	467	558	686	754	879	956	
Tapa	151	171	232	250	262	373	482	579	717	771	891	986	
	152	172	208	231	270	391	468	557	679	783	891	977	
Keskiarvo	**152,2**	**174**	**224**	**245,8**	**267,4**	**384,2**	**471**	**570,4**	**687**	**774,8**	**888,2**	**984**	
Poikkeama	**0,02**	**0,07**	**0,11**	**0,11**	**0,03**	**0,07**	**0,03**	**0,06**	**0,06**	**0,04**	**0,02**	**0,06**	0,06

Kuva 5.2: Testin lopulliset tulokset.

Kuvassa 5.2 esitellään tutkielman testauksessa saadut lopulliset testitulokset. Kuvaa 5.2 tulkitaan samalla tavalla kuin edellä esiteltyä kuvaa 5.1. Edellä hylätyt testitapaukset uusittiin suorittamalla testaus yhdellä kertaa pitämättä taukoja testitapausten välillä. Uusituissa testitapauksissa oli kaikissa kyse yhdestä testitavasta. Verrattuna testin ensimmäiseen läpikäyntiin, oli uusintatestin suorituskaava siis hieman toisenlainen. Ensimmäisessä läpikäynnissä tuloksia saatiin kuvan 5.2 tulosten esitysmuodon mukaisesti sarake kerrallaan. Tällöin löydettiin runsaasti uusinnan aiheuttavia poikkeamia testituloksissa juuri ensimmäiseltä riviltä. Tästä syntyy epäilys siitä, onko tämä poikkeamia aiheuttanut ilmiö heijastanut vaikutuksiaan myöskin alempien rivien tuloksiin siten, että niitä olisi vain taulukosta vaikea havaita. Uusitussa testissä tuloksia saatiin vain ensimmäiselle riville. Uusituista tuloksista ei ollut löydettävissä mitään merkkejä alakohdassa 5.3.1 esitellystä ilmiöstä. Näin ollen voidaan todeta ilmiön aiheuttaneen vaikutuksensa ainoastaan ensimmäisiin mitattuihin arvoihin ja uusimatta jääneiden testitapausten tulokset voidaan hyväksyä sellaisenaan.

6 TULOSTEN ARVIOINTI

Luvussa viisi esiteltiin tutkielman testauksesta saadut tulokset. Tässä luvussa määritellään näiden tulosten analyysin ja vertailun viitekehykset sekä suoritetaan tulosten analyysi ja vertailu. Lopussa esitellään analyysistä tehdyt johtopäätökset.

6.1 Analyysin viitekehys

Tämän kohdan keskeisin tavoite on määritellä testauksesta saatujen aineistojen testitapakohtainen analyysin viitekehys sekä aineistojen vertailun viitekehys. Aineistolla tarkoitetaan tässä sellaista rajattua testauksesta saatujen tulosten joukkoa, jossa kaikki yksittäisten testitapausten testitulokset on saatu samalla testitavalla ja samassa mittakaavassa. Aineistoja on siis käytettävissä kahdeksan kappaletta. Koko aineistolla tarkoitetaan näitä kahdeksaa aineistoa yhdessä. Testitapakohtainen analyysin viitekehys esitellään alakohdassa 6.1.1 ja testitapakohtainen analyysi kohdassa 6.2. Aineiston vertailun viitekehys esitellään alakohdassa 6.1.2 ja vertailu suoritetaan kohdassa 6.3.

Analyysin viitekehyksen keskeisin menetelmä on regressioanalyysi ja tarkennettuna yhden selittävän muuttujan regressiomalli. Siinä selitetään kahden muuttujan välistä korrelaatiota määrittelemällä regressiosuora. Regressiosuoralla tarkoitetaan sellaista korrelaatiodiagrammiin sijoitettua suoraa, joka mahdollisimman hyvin kuvaa diagrammin pistejoukkoa. Kun selittäviä muuttujia on vain yksi, on regressiosuoran selitysaste korrelaatiokertoimen neliö. Selitysaste on suhdeluku siitä, kuinka hyvin korrelaatiodiagrammin pisteet noudattavat regressiosuoraa. Toisin sanoen selitysaste kuvaa tutkittavan ilmiön korrelaation lineaarisuutta. Regressiosuoraa käytetään selitettävän muuttujan arvon ennustamiseen käyttäen selittävää muuttujaa. Mitä korkeampi selitysaste regressiosuoralla on, sen luotettavampi on ennustettu arvo. Regressiosuoran muuttujien laskukaavat löytyvät lähteestä Riukulehto & Huhtala (1992) sivulta 86. (Riukulehto & Huhtala, 1992, 86-87.)

6.1.1 Testitapakohtainen analyysin viitekehys

Alakohdassa 5.1.1 esiteltiin testissä käytettävä mitta-asteikko. Sen todettiin olevan suhdeasteikollinen, koska asteikon minimikohta on asteikkoon nähden absoluuttinen nollapiste ja koska arvojen suhteellinen muutos on määriteltävissä. Suhdeasteikollinen mitta-asteikko mahdollistaa myös tutkittavan aineiston tulosten jatkuvuuden tunnistamisen. Tutkielman tutkimuksesta saadut tulokset sisältävät kahdeksan aineistoa, kaksi kutakin testitapaa kohden (pienen ja suuren mittakaavan testit). Näistä aineistoista kukin analysoidaan erikseen. Koska suoraviivaisen tavan testauksessa on tutkimuksen ainoa tutkittava muuttuja ollut testausmateriaalin määrä kasvatettuna vakiomäärällä, voidaan näiden aineistojen kuvaajien olettaa olevan nousevia suoria. Käytännön tavan testauksissa mukana on myös toinen muuttuja: ylimääräisen käsittelyn tuoma viive testausohjelmiston suoritukseen. Tämän suorituksen arvioitiin kuitenkin aiheuttavan viiveensä lineaarisesti, koska ylimääräisen käsittelyn määrä testausohjelmistossa on suoraan verrannollinen testattavaan rivimäärään ja siten myös käytännön tavan testaustulosten kuvaajien arveltiin olevan nousevia suoria. Tämän luvun tehtävänä on luoda testitapakohtaisille aineistoille viitekehys, jolla nämä ominaisuudet voidaan todeta olevan tai ei olevan olemassa. Lisäksi viitekehyksessä muodostetaan kullekin aineistolle kuvaaja sekä määritellään sen kuvaama funktio.

Edellä mainittiin tutkittavan aineiston kuvaajalle oletettavan kaksi ominaisuutta, nouseva ja suora. Aineiston kuvaajan voidaan todeta olevan nouseva aina, kun testaustason X_i arvo on pienempi kuin testaustason X_{i+1} arvo. Tässä i on testaustason järjestysluku testaustasojen ollessa suuruusjärjestyksessä pienimmästä suurimpaan. Kuvaajan lineaarisuuden toteaminen on hieman monimutkaisempaa. Tässä lineaarisuutta tutkitaan määrittelemällä kullekin aineistolle regressiosuora, jonka selitysasteen perusteella määritellään lineaarisuus. Selitysasteen tulkintarajaksi määritellään tässä 0,95. Tulkintaraja on melkoisen ankara yhden selittävän muuttujan regressiosuoralle, mutta siinä on huomioitu suhteellisen pienet testausmäärät. Jos aineiston regressiosuoran selitysaste ei ylitä asetettua tulkintarajaa, tarkastellaan aineistoa etsien siitä yksittäisiä suoraan nähden erityisen suuria tai pieniä arvoja. Jos tulkintarajan alitukselle ei tällaista selittävää tekijää löydy, harkitaan kyseiselle aineistolle uusintatestiä. Jos tulkintaraja alittuu yli kymmenyksellä, uusitaan testi joka

tapauksessa. Pienen mittakaavan aineistoille testiä ei kuitenkaan uusita, koska siinä on kyseessä toissijainen mielenkiinnon kohde. Pienessä mittakaavassa regressiosuorien selitysasteiden odotusarvot ovat muutenkin isoa mittakaavaa pienemmät johtuen testausohjelmiston kuluttaman minimiajan ja siinä tapahtuvan satunnaisen vaihtelun suuremmasta suhteellisesta vaikutuksesta suoritusaikoihin.

Testitapakohtaisessa analyysissä esitellään myös aineistojen regressiosuorien funktiot, jotka ovat muotoa $f(x) = kx + m$, missä k on suoran regressiokerroin (kulmakerroin), m on vakio ja x selittävä muuttuja. Tässä tutkittavan ilmiön tapauksessa m on testausohjelmiston kuluttama minimiaika ja x on tietokantaan siirrettävä rivimäärä. Funktio antaa tuloksena testausohjelmiston kuluttaman ajan sekunnin sadasosina kullakin rivimäärällä kyseisellä testaustavalla. Myöhemmin käytettäessä termiä aineiston kuvaajan funktio, tarkoitetaan regressiosuoran funktiota.

6.1.2 Testitapojen vertailun viitekehys

Testitapojen vertailu jaetaan kolmeen eri lähestymistapaan. Ensimmäisessä osassa vertaillaan pienen ja ison mittakaavan aineistoja etsien niistä odotetun kaltaista samanlaisuutta. Vertailussa keskitytään tutkimaan kulmakertoimia ja toteamaan, että isossa mittakaavassa havaitut suuntaukset suorituskykyjen eroille joko ovat tai eivät ole havaittavissa jo pienissä tietomäärissä. Toisessa osassa vertaillaan puhtaan ja käytännön tavan tuloksia etsien käytännön tavan tuloksista suorituskykyerojen muutoksia puhtaaseen tapaan verrattuna. Siinä vertaillaan lähinnä kulmakertoimien suhteellista muutosta siirryttäessä suoraviivaisesta tavasta oppikirjamaiseen tapaan niin käytännön kuin puhtaankin tavan testaustuloksissa. Kolmannessa lähestymistavassa vertaillaan tutkielman kannalta keskeisintä mielenkiinnon kohdetta, suoraviivaisen ja oppikirjamaisen tavan suorituskykyeroja. Siinä pyritään puhtaan tavan ison mittakaavan aineistoja käyttäen löytämään kaava, joka kuvaa suoraviivaisen ja oppikirjamaisen tavan suorituskykyerojen muutosta.

Kuten edellä mainittiin, ison ja pienen mittakaavan vertailussa etsitään aineistoista odotetun kaltaista samanlaisuutta. Vertailussa tehdään lyhyt katsaus kulmakerrointen absoluuttisiin eroihin pienen ja ison mittakaavan välillä. Tästä tarkastelusta

mahdollisesti löytyviä ilmiöitä ei kuitenkaan lähdetä tarkemmin analysoimaan, vaan niiden lähinnä todetaan olevan olemassa. Vertailun keskeisimpänä kohteena ovat kulmakertoimien suhteet testaustapojen välillä siirryttäessä suoraviivaisen tavan testauksesta oppikirjamaiseen testaukseen. Kulmakertoimien suhteita vertaillaan niiden prosentuaalisina muutoksina suoraviivaisen ja oppikirjamaisen tavan välillä. Kiinnostuksen kohteena on, löytyykö isossa mittakaavassa havaittuja ominaisuuksia jo pienissä tietomäärissä. Jos näissä prosenttiluvuissa on nähtävissä samankaltaisuutta, voidaan todeta mittaustulosten olevan samansuuntaisia molemmissa mittakaavoissa. Samankaltaisuuden tulkinta jätetään tässä hieman epämääräiselle tasolle, koska kyseessä eivät kuitenkaan ole tutkielman kannalta kaikkein keskeisimmät vertailun kohteet. Pienen ja ison mittakaavan vertailun lopussa vielä arvioidaan aineistojen kuvaajien lineaarisuutta, ja sitä kautta mittaustulosten luotettavuutta, visuaalisesti tarkastellen. Samalla tutkitaan, ovatko aineistojen kuvaajat kulmakertoimen mukaisesti samassa järjestyksessä molemmissa mittakaavoissa. Jos näiden kahden lähestymistavan avulla ei voida osoittaa pienen ja ison mittakaavan aineistojen samankaltaisuutta, tarkastellaan vielä mittakaavojen aineistoja omina kokonaisuuksinaan testitapakohtaisen analyysin valossa.

Puhtaan ja käytännön tavan vertailussa etsitään mahdollisia suorituskykyerojen muutoksia. Kuten pienen ja ison mittakaavan vertailussa, on tässäkin tarkastelun kohteena kulmakertoimien prosentuaaliset muutokset siirryttäessä suoraviivaisen tavan aineistosta oppikirjamaisen tavan aineistoon. Tässä kuitenkin keskitytään vertailemaan näitä muutoksia nimenomaan puhtaan ja käytännön tavan välillä. Tavoitteena on osoittaa, että käytännön tapa käyttää EJB:tä joko vaikuttaa tai ei vaikuta suorituskykyeroihin suoraviivaisen ja oppikirjamaisen tavan välillä. Lisäksi esitellään mahdollisen vaikutuksen suunta, eli käytännön tapa joko vähentää tai lisää suorituskykyeroja.

Suoraviivaisen ja oppikirjamaisen tavan tulosten vertailussa keskitytään lähinnä puhtaan testaustavan ison mittakaavan tuloksiin. Tämä sen vuoksi, että kaikki muut vaihtoehdot tuovat vertailuun mukaan sellaista epämääräisyyttä, joka estää tai ainakin merkittävästi vaikeuttaa vertailun yleistettävyyttä. Tapojen vertailussa keskitytään tarkastelemaan aineistojen regressiosuorien funktioita ja niistä muodostetaan kaava, jonka mukaisesti suoraviivaisen ja oppikirjamaisen tavan suorituskykyerot

muodostuvat. Kaavan on tarkoitus edustaa todistusta siitä, että suoraviivaisen ja oppikirjamaisen tavan suorituskykyero kasvaa tiedon määrän kasvaessa ja muuttuu siten näkökulmasta riippumatta jossain vaiheessa merkittäväksi (ks. alakohta 5.1.1). Edelleen kaavasta esitellään yleinen muoto, jota voidaan tietyin rajoituksin käyttää myös tämän tutkimuksen ulkopuolella. Kyseessä on tutkielman keskeisin vertailun kohde, jossa huomio kiinnitetään nimenomaan suorituskykyerojen tarkasteluun. Vertailussa käytettävät funktiot ovat hyvin tapauskohtaista materiaalia ja niiden yleistettävyys sellaisenaan on hyvin vähäistä. Koska funktiot on saatu varsin kattavan ja yhdenmukaisen testin tuloksena ja koska testin tulosten on osoitettu olevan vertailukelpoisia, on niiden osoittama suorituskykyero kuitenkin tapauskohtaisesti validi ja jossain määrin yleistettävissä.

6.2 Testitapakohtainen analyysi

Alakohdassa 6.1.1 esiteltiin testitapakohtaisen analyysin viitekehys. Tässä alakohdassa analysoidaan testiaineistot testitapakohtaisesti tuon viitekehyksen mukaisesti. Lopussa tarkastellaan ja arvioidaan tulosten reliabiliteettiä ja validiteettia. Alakohdan keskeisimpänä tavoitteena on osoittaa testistä saatujen tulosten olevan uskottavia ja keskenään vertailukelpoisia.

6.2.1 Suoraviivainen puhdas tapa

Sekä pienen että ison mittakaavan suoraviivaisen puhtaan tavan aineistoissa jokaisen testaustason testitulos on pienempi kuin järjestyksessään seuraavan testaustason tulos. Näin ollen voimme todeta molempien aineistojen kuvaajien olevan nousevia.

SUORAVIIVAINEN-PUHDAS ISO MITTAKAAVA		PIENI MITTAKAAVA	
Regression Statistics		*Regression Statistics*	
Multiple R	0,981	Multiple R	0,903
R Square	0,962	R Square	0,816
Observations	40	Observations	25
	Coefficients		*Coefficients*
Intercept	90,3	Intercept	93,1
X Variable 1	0,24	X Variable 1	0,38

Kuva 6.1: Suoraviivaisen puhtaan tavan regressioanalyysin tulokset.

Kuvassa 6.1 on kuvattu suoraviivaisen puhtaan tavan regressioanalyysin tulokset sekä ison että pienen mittakaavan testaustuloksista. "Multiple R" ilmoittaa aineiston korrelaatiokertoimen ja "R square" on korrelaatiokertoimen neliö, eli regressiosuoran selitysaste. "Observations" –rivillä on ilmoitettu korrelaatiopisteiden eli testaustapahtumien lukumäärä. "Coefficients" –sarakkeessa on "Intercept" –rivillä regressiosuoran funktion vakio ja "X Variable 1" –rivillä on regressio- eli kulmakerroin.

Ison mittakaavan aineiston regressiosuoran selitysaste on kuvan 6.1 mukaisesti 0,962 ja pienen mittakaavan selitysaste 0,816. Ison mittakaavan aineistossa regressiosuoran voidaan siten tulkita alakohdan 6.1.1 tulkintarajan mukaisesti olevan lineaarinen. Pienessä mittakaavassa selitysaste osoittaa selvää hajontaa tuloksissa tapahtuneen. Pienen mittakaavan korrelaatio on kuitenkin melko hyvä, kun otetaan huomioon kyseessä olevan yhden selittävän muuttujan regressiomalli. Ison mittakaavan selitysaste ei anna aihetta testin uusimiselle. Ison mittakaavan aineiston kuvaajan funktioksi saadaan $f(x) = 0,24x + 90,3$. Pienen mittakaavan aineiston kuvaajan funktio on $f(x) = 0,38x + 93,1$. Pienen mittakaavan kuvaaja nousee hieman jyrkemmin kuin ison mittakaavan kuvaaja.

6.2.2 Oppikirjamainen puhdas tapa

Sekä pienen että ison mittakaavan oppikirjamaisen puhtaan tavan aineistoissa jokaisen testaustason testitulos on pienempi kuin järjestyksessään seuraavan testaustason tulos. Näin ollen voimme todeta molempien aineistojen kuvaajien olevan nousevia.

OPPIKIRJAMAINEN-PUHDAS ISO MITTAKAAVA		PIENI MITTAKAAVA	
Regression Statistics		*Regression Statistics*	
Multiple R	0,996	Multiple R	0,939
R Square	0,993	R Square	0,881
Observations	40	Observations	25
	Coefficients		*Coefficients*
Intercept	131,3	Intercept	97,6
X Variable 1	0,71	X Variable 1	1,03

Kuva 6.2: Oppikirjamaisen puhtaan tavan regressioanalyysit.

Kuvassa 6.2 on kuvattu oppikirjamaisen puhtaan tavan regressioanalyysin tulokset sekä ison että pienen mittakaavan testauksessa. Kuvaa tulkitaan samalla tavalla kuin kuvaa 6.1.

Ison mittakaavan aineiston regressiosuoran selitysaste on kuvan 6.2 mukaisesti 0,993 ja pienen mittakaavan selitysaste 0,881. Ison mittakaavan aineiston kuvaajan voidaan näin ollen tulkita olevan selvästi lineaarinen ja erittäin uskottava. Pienen mittakaavan aineiston lineaarisuudessa puolestaan on jonkin verran hajontaa. Ison mittakaavan aineiston kuvaajan funktioksi saadaan $f(x) = 0,71x + 131,3$. Pienen mittakaavan aineiston kuvaajan funktio on $f(x) = 1,03x + 97,6$. Pienen mittakaavan kuvaaja nousee lähes viisikymmentä prosenttia jyrkemmin kuin ison mittakaavan kuvaaja.

6.2.3 Suoraviivainen käytännön tapa

Sekä pienen että ison mittakaavan suoraviivaisen käytännön tavan aineistoissa jokaisen testaustason testitulos on pienempi kuin järjestyksessään seuraavan testaustason tulos. Näin ollen voimme todeta molempien aineistojen kuvaajien olevan nousevia.

SUORAVIIVAINEN-KÄYTÄNNÖN ISO MITTAKAAVA		PIENI MITTAKAAVA	
Regression Statistics		*Regression Statistics*	
Multiple R	0,994	Multiple R	0,977
R Square	0,988	R Square	0,954
Observations	40	Observations	25
	Coefficients		*Coefficients*
Intercept	132,0	Intercept	107,4
X Variable 1	0,58	X Variable 1	0,97

Kuva 6.3: Suoraviivaisen käytännön tavan regressioanalyysin tulokset.

Kuvassa 6.3 on kuvattu suoraviivaisen käytännön tavan regressioanalyysin tulokset sekä ison että pienen mittakaavan testauksessa. Kuvaa tulkitaan samalla tavalla kuin kuvaa 6.1.

Ison mittakaavan aineiston regressiosuoran selitysaste on kuvan 6.3 mukaisesti 0,988 ja pienen mittakaavan selitysaste 0,954. Molempien mittakaavojen aineistojen kuvaajien voidaan siten tulkita olevan selvästi lineaarisia. Ison mittakaavan aineiston kuvaajan funktioksi saadaan $f(x) = 0{,}58x + 132{,}0$. Pienen mittakaavan aineiston kuvaajan funktio on $f(x) = 0{,}97x + 107{,}4$. Pienen mittakaavan kulmakerroin on lähes kaksinkertainen ison mittakaavan kulmakertoimeen verrattuna.

6.2.4 Oppikirjamainen käytännön tapa

Sekä pienen, että ison mittakaavan oppikirjamaisen käytännön tavan aineistoissa jokaisen testaustason testitulos on pienempi kuin järjestyksessään seuraavan testaustason tulos. Näin ollen voimme todeta molempien aineistojen kuvaajien olevan nousevia.

OPPIKIRJAMAINEN-KÄYTÄNNÖN ISO MITTAKAAVA		PIENI MITTAKAAVA	
Regression Statistics		*Regression Statistics*	
Multiple R	0,998	Multiple R	0,978
R Square	0,997	R Square	0,956
Observations	40	Observations	25
	Coefficients		*Coefficients*
Intercept	169,6	Intercept	122,0
X Variable 1	1,02	X Variable 1	1,51

Kuva 6.4: Oppikirjamaisen käytännön tavan regressioanalyysin tulokset.

Kuvassa 6.4 on kuvattu oppikirjamaisen käytännön tavan regressioanalyysin tulokset sekä ison että pienen mittakaavan testauksessa. Kuvaa tulkitaan samalla tavalla kuin kuvaa 6.1.

Ison mittakaavan aineiston regressiosuoran selitysaste on kuvan 6.4 mukaisesti 0,997 ja pienen mittakaavan selitysaste 0,956. Molemmissa mittakaavoissa aineistojen kuvaajien voidaan siten tulkita olevan selvästi lineaarisia. Ison mittakaavan aineiston kuvaajan funktioksi saadaan $f(x) = 1{,}02x + 169{,}6$. Pienen mittakaavan aineiston kuvaajan funktio on $f(x) = 1{,}51x + 122{,}0$. Pienen mittakaavan kulmakerroin on lähes puolitoistakertainen ison mittakaavan kulmakertoimeen verrattuna.

6.2.5 Mittausten reliabiliteetti ja validiteetti

Mittarin reliabiliteetilla tarkoitetaan satunnaisvirheiden osuutta mittaustuloksissa. Mittarin validiteetilla puolestaan tarkoitetaan mittarin kykyä mitata haluttua ominaisuutta. Toisin sanoen, jos mittaustuloksissa on satunnaisvirheitä vähän, on reliabiliteetti hyvä ja jos mittari mittaa sitä mitä sen odotetaankin mittaavan, on validiteetti hyvä. (Kankainen & Högmander, 2001, 6.)

Tutkittavan aineiston kuvaajien oletettiin etukäteen olevan lineaarisia, koska testauksessa käytettiin vain yhtä tuloksiin vaikuttanutta muuttujaa kasvatettuna vakiomäärällä. Mittauksen reliabiliteettia voidaan siten arvioida tarkastelemalla

tulosten lineaarisuutta. Edellä lineaarisuuden tarkastelua varten on tehty regressioanalyysi, jonka selitysaste toimii mittayksiköstä riippumattomana suhdelukuna korrelaatiopisteiden lineaarisuudelle (Ks. alakohta 6.1.1.). Ison mittakaavan aineistojen todettiin olevan selkeästi lineaarisia ja virhemäärien erittäin vähäisiä. Pienessä mittakaavassa puolestaan kahden regressiosuoran selitysasteet osoittivat aineiston kuvaajissa olevan jonkin verran hajontaa ja kahdessa tapauksessa lineaarisuus selkeästi havaittavissa. Tulkinnat tehtiin alakohdassa 6.1.1 määritellyllä tavalla siten, että isojen mittakaavojen tapauksessa lineaarisuuden tulkintaraja selitysasteen arvossa oli 0,95. Pienessä mittakaavassa funktion kulmakerroin oli säännönmukaisesti suurempi kuin isossa mittakaavassa, mutta tulokset olivat kuitenkin hyvin samansuuntaiset: testitapojen kulmakertoimet kasvoivat testitapauksittain samassa järjestyksessä. Koska tulokset olivat hyvin odotetun kaltaisia, isossa mittakaavassa reliabiliteetti voidaan tulkita hyväksi ja pienessäkin mittakaavassa kohtuulliseksi. Reliabiliteetin olisi voinut odottaa olevan parempi, jos testaus olisi suoritettu ilman lähiverkon tuomaa heilahtelua testausohjelmiston ei-testattavien osien suoritusaikaan.

Mittaustulosten mittayksikkönä toimii aika ja testauksen tavoitteena on saada vertailukelpoisia tuloksia, ei niinkään absoluuttisia aika-arvoja. Eri testitapauksista saatujen tulosten voidaan todeta olevan keskenään vertailukelpoisia, joten mittauksen validiteetti on osoittautunut hyväksi.

6.3 Testitapojen vertailua

Kohdassa 6.2 analysoitiin kunkin testaustavan tulokset erikseen. Kohdassa osoitettiin testistä saatujen tulosten olevan uskottavia ja vertailukelpoisia. Tässä luvussa vertaillaan näitä tuloksia keskenään alakohdan 6.1.2 viitekehyksen mukaisesti. Aluksi vertaillaan ison ja pienen mittakaavan testaustuloksia tarkastellen niitä lähinnä odotetun samankaltaisuuden näkökulmasta. Sitten vertaillaan puhtaan ja käytännön tavan tuloksia keskenään. Siinä näkökulmana on lähinnä tarkastelu siitä, onko käytännön tavan testauksessa löydettävissä jotain yllättäviä havaintoja suoraviivaisen ja oppikirjamaisen tavan kesken. Lopuksi suoritetaan suoraviivaisen ja

oppikirjamaisen tavan tuloksien vertailua. Viimeksi mainittu on tutkielman keskeisin mielenkiinnon kohde.

6.3.1 Pienen ja ison mittakaavan vertailua

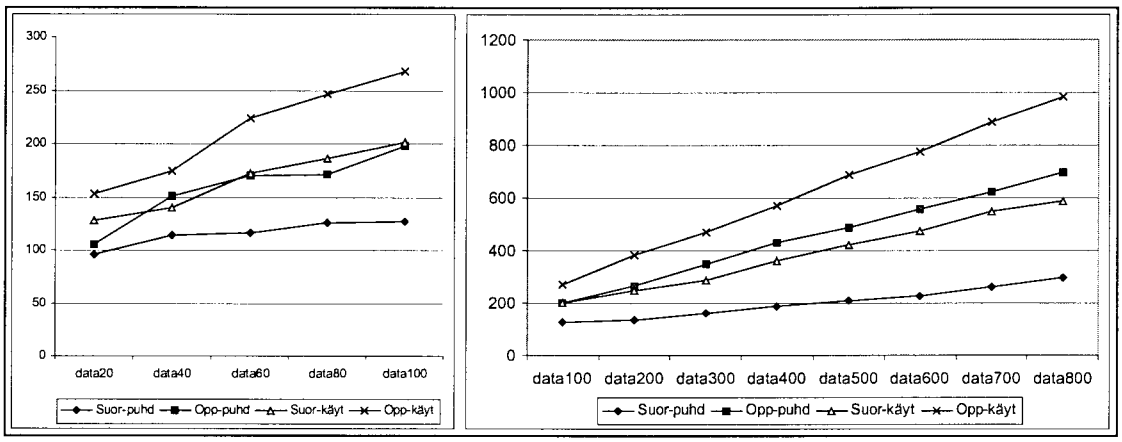
Alakohdassa 5.1.1 perusteltiin pienen mittakaavan testauksen olemassaoloa sillä, että halutaan nähdä, onko testauksessa löydettävät suorituskykyjen erot nähtävissä jo pienemmissä testausmäärissä. Tässä alakohdassa tehdään vertailua, jonka tavoitteena on löytää odotettu yhtäläisyys pienen ja ison mittakaavan suorituskykyeroissa. Aluksi vertaillaan molempien mittakaavojen aineistojen kuvaajien kulmakertoimia keskenään. Sitten tarkastellaan graafia, jossa esitellään kaikkien aineistojen kuvaajat.

Kertoimien keskiarvot	**Pieni mittakaava**	**Iso mittakaava**
Suoraviivainen puhdas	0,38	0,24
Oppikirjamainen puhdas	1,03 (+ 271 %)	0,71 (+ 296 %)
Suoraviivainen käytännön	0,97	0,58
Oppikirjamainen käytännön	1,51 (+ 156 %)	1,02 (+ 176 %)

Taulukko 6.1: Aineistojen kulmakertoimien keskiarvot.

Taulukossa 6.1 on esitelty testauksen kaikkien aineistojen kuvaajien kulmakertoimet. Taulukosta on selvästi havaittavissa, että pienessä mittakaavassa kulmakertoimet ovat säännönmukaisesti noin viisikymmentä prosenttia suurempia kuin isossa mittakaavassa. Kulmakertoimien erot johtunevat testausohjelmiston jostain sellaisesta osasta, joka aiheuttaa voimakkaampaa suoritusajan kasvua pienissä tietomäärissä. Samankaltaista ilmiötä ei ollut kuitenkaan havaittavissa ison mittakaavan aineistojen alkupäässä, joten se näyttäisi esiintyvän ainoastaan erittäin pienissä tietomäärissä. Kulmakertoimien erot pienen ja ison mittakaavan välillä eivät kuitenkaan ole tutkielman kannalta kovin kiinnostavia, joten niitä ei tässä tarkemmin analysoida. Sen sijaan kulmakertoimien keskinäiset suhteet eri testaustapojen välillä ovat kiinnostavia. Taulukon 6.1 oppikirjamaisen tavan riveille on kulmakertoimen viereen sulkujen sisään merkitty prosentuaalinen kulmakertoimen kasvu siirryttäessä suoraviivaisesta tavasta oppikirjamaiseen tapaan. Puhtaassa tavassa siirtymä on pienessä mittakaavassa

271 % ja isossa mittakaavassa 296 %. Käytännön tavassa vastaavat prosenttiluvut ovat pienessä mittakaavassa 156 % ja isossa 176 %. Molemmissa mittakaavoissa siis kulmakertoimen kasvu on siis hyvin samansuuntainen siirryttäessä suoraviivaisesta tavasta oppikirjamaiseen. Edelleen kulmakertoimen kasvun muutos siirryttäessä puhtaasta tavasta käytännön tapaan, on molemmissa mittakaavoissa hyvin samansuuntainen. Tästä voidaan todeta, että molempien mittakaavojen tulokset ovat erittäin samansuuntaisia huolimatta siitä, että kohdassa 6.2 esitetyissä aineistokohtaisissa analyyseissä havaittiin aineistojen regressiosuoran selitysasteissa isompia heilahduksia pienen mittakaavan tuloksissa verrattuna isoon mittakaavaan.



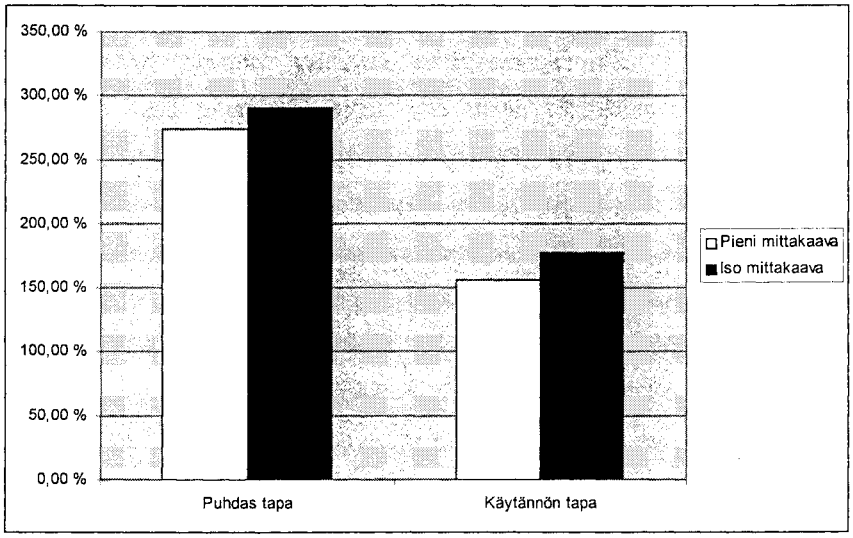
Kuva 6.5: Pienen ja ison mittakaavan aineistojen kuvaajat.

Kuvassa 6.5 esitellään pienen ja ison mittakaavan testauksissa saatujen aineistojen kuvaajat. Kuvaajat on piirretty testaustulosten keskiarvojen mukaan (ks. kuva 5.2). Pienen mittakaavan kuvaajat ovat vasemmanpuoleisessa ja ison mittakaavan kuvaajat oikeanpuoleisessa kaaviossa. Molempien kuvien pystyakselilla olevat arvot ovat sekunnin sadasosia. Pienen mittakaavan kuvaajia tarkasteltaessa voidaan selvästi havaita heilahtelua tapahtuneen mittauksissa. Tämä ilmiö oli arvattavissa jo kohdan 6.2 selitysasteiden tarkastelussa, jossa pienen mittakaavan selitysasteet olivat jonkin verran ison mittakaavan selitysasteita pienemmät. Ison mittakaavan tuloksissa suuret selitysasteet näkyvät selvänä lineaarisuutena kuvaajissa. Kuitenkin käyrät ovat kulmakertoimen mukaan järjestettynä samassa järjestyksessä molemmissa kaavioissa ja kulmakertoimet kasvavat lähes samassa suhteessa, joten suorituskykyjen erojen voidaan todeta olevan havaittavissa jo pienissä tietomäärissä. Johtuen suurista

heilahteluista tuloksissa, eivät pienen mittakaavan tulokset kuitenkaan yksinään olisi riittäneet todentamaan tätä suorituskykyjen eroa edes kohtuullisen luotettavasti.

6.3.2 Puhtaan ja käytännön tavan vertailua

Alakohdassa 3.2.2 esiteltiin puhdas sekä käytännön tapa viedä tietoa tietokantaan. Tällä jaottelulla on tarkoitus selvittää, onko käytännön tavalla käyttää EJB:tä suorituskykyjen eroa vähentävää tai lisäävää vaikutusta. Vastauksen kyseiseen ongelmaan saamme tarkastelemalla puhtaassa ja käytännön tavassa aineistojen kulmakertoimien suhteellista muutosta siirryttäessä suoraviivaisen tavan tuloksista oppikirjamaisen tavan tuloksiin.



Kuva 6.6: Kulmakertoimien prosentuaaliset kasvut siirryttäessä suoraviivaisen tavan aineistoista oppikirjamaisen tavan aineistoihin.

Kuvassa 6.6 on esitelty kulmakertoimien suhteelliset muutokset siirryttäessä suoraviivaisesta tavasta oppikirjamaiseen tapaan. Toisin sanoen kaikissa sadan prosentin ylityksissä oppikirjamainen tapa on suoraviivaista tapaa hitaampi. Pystyakselilla olevat arvot edustavat muutoksen määrää prosenteissa suoraviivaisen ja oppikirjamaisen tavan välillä ja vaaka-akselilla on esitelty kertoimen muutos ensin

puhtaassa tavassa ja sitten käytännön tavassa viedä tietoa tietokantaan. Valkoiset palkit kuvaavat muutosta pienen mittakaavan mittauksissa saaduissa aineistoissa ja mustat palkit ison mittakaavan aineistoissa. Molemmissa mittakaavoissa on selvästi havaittavissa samanlainen suuntaus kulmakertoimen muutoksissa. Puhtaassa tavassa oppikirjamainen tapa käyttää EJB:tä kasvattaa ohjelman suoritukseen kuluvan ajan lähes kolminkertaiseksi suoraviivaiseen tapaan verrattuna. Toisin sanoen, kun ohjelman käyttämästä kokonaisajasta vähennetään ohjelman kuluttama minimiaika, on oppikirjamainen tapa lähes kolme kertaa hitaampi verrattuna suoraviivaiseen tapaan. Vastaasti käytännön tavassa tämä suorituksen hidastuminen on reilusti alle kaksinkertainen molemmissa mittakaavoissa.

Käsittelyn määrä käytännön tavan testauksessa on tutkielman testauksessa ollut lähes suoraan suhteessa kulloinkin testattavaan rivimäärään. Parempi ja monipuolisempi käytännön tavan tiedon käsittelyn määrittely voisi antaa täsmällisempää tietoa siitä, miten tiedon käsittely vaikuttaa suorituskykyeroihin. Tutkielman testauksessa käytännön tapa toteutettiin siten, että tietoa käsiteltiin käyttäen komponentteja samassa EJB-säiliössä kuin missä itse ajallisesti mitattava testaus suoritettiin. Tämä on mahdollisesti aiheuttanut tilanteen, jossa kyseinen säiliö on tiedon käsittelyn johdosta siirtynyt tavalla tai toisella parempaan valmiustilaan juuri ennen varsinaisen mitattavan testauksen aloittamista. Laadullisesti ja määrällisesti erilainen käsittely saattaisi antaa toisenlaisia tuloksia. Joka tapauksessa ylimääräinen tiedon käsittely ohjelman suoritusprosessissa lisää ohjelmiston kuluttamaa minimiaikaa ja vaikuttaa siten tasoittavasti suoraviivaisen ja oppikirjamaisen tavan suorituskykyeroon.

6.3.3 Suoraviivaisen ja oppikirjamaisen tavan vertailua

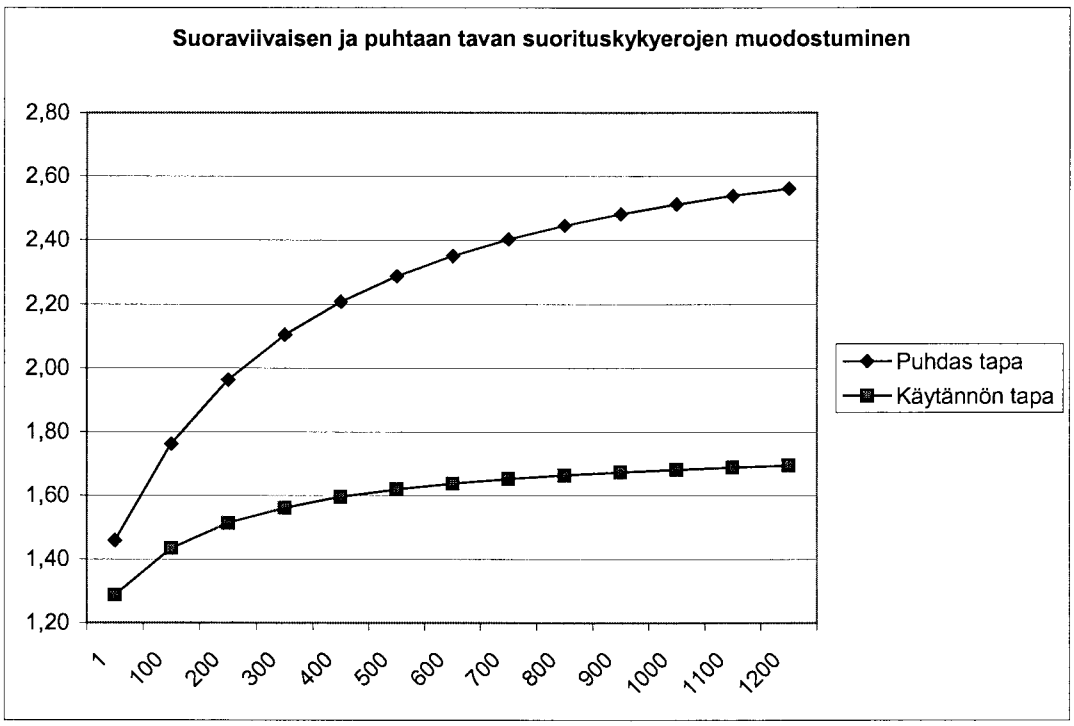
Edellä jo hieman tarkasteltiin suoraviivaisen ja oppikirjamaisen tavan eroja käytännön ja puhtaan tavan vertailussa. Siinä todettiin, että kun testausohjelmiston minimiaika jätetään huomioimatta, on oppikirjamaisen tavan suoritusaika puhtaalla tavalla lähes kolminkertainen ja käytännönkin tavalla yli puolitoista kertainen suoraviivaisen tavan suoritusaikaan verrattuna (ks. alakohta 6.3.2). Testausohjelmiston minimiajan huomioimatta jättäminen johtaa kuitenkin tuloksiin, jotka eivät anna täysin realistista kuvaa suorituskykyjen eron muodostumisesta. Todellisten suorituskykyerojen

muodostumista kuvaava kaava saadaan muodostettua testituloksista johdetuista funktioista. Tässä käytetään pelkästään ison mittakaavan tuloksia niiden suuremman luotettavuuden vuoksi.

Suor-puhd	Opp-puhd	Suor-käyt	Opp-käyt
0,24x + 90,3	0,71x + 131,3	0,58x + 132,0	1,02x + 169,6

Taulukko 6.2: Ison mittakaavan aineistojen kuvaajien funktiot.

Taulukossa 6.2 on esitelty ison mittakaavan testauksessa saatujen aineistojen funktiot. Näiden funktioiden kautta tarkastellaan suoraviivaisen ja oppikirjamaisen tavan suhteellista suorituskykyeroa, kun testausohjelmiston kuluttama minimiaika myös huomioidaan. Suorituskykyjen eroa hyvin kuvaava lukuarvo on kerroin, joka kuvaa oppikirjamaisen tavan kuluttamaa aikaa suhteessa suoraviivaiseen tapaan. Tämä kerroin saadaan jakamalla oppikirjamaisen tavan funktio suoraviivaisen tavan funktiolla. Edelleen sama kerroin kerrottuna sadalla on oppikirjamaisen tavan suorituskyvyn prosenttimäärä suoraviivaisen tavan suorituskyvystä.



Kuva 6.7: Suoraviivaisen ja oppikirjamaisen tavan suorituskykyerojen muodostuminen.

Kuvassa 6.7 on esitelty suoraviivaisen ja oppikirjamaisen tavan suorituskykyjen erojen muodostuminen sekä puhtaassa että käytännön tavassa viedä tietoa tietokantaan. Kaaviota tulkitaan siten, että käyrän jokaisella pisteellä oppikirjamainen tapa on pystyakselilla ilmoitetun lukuarvon verran suoraviivaista tapaa hitaampi vaaka-akselilla ilmoitetulla tietokannan rivimäärällä. Tässä keskitytään tarkastelemaan lähinnä puhtaan tavan käyrää. Käytännön tavan käyrän tutkiminen ei ole tässä kovin hyödyllistä, koska käytännön tavan testauksessa suoritettu tiedon ylimääräisen käsittelyn määrä ei ole yleistettävissä. Käytännön tavan suorituskykyjen eroa kuvaava käyrä on kuvassa 6.7 lähinnä muistuttamassa siitä, että puhdas tapa on vain yksi erikoistapaus tiedon viennissä tietokantaan ja erilaisilla variaatioilla voidaan suorituskykyjen eroa saada vähennettyä. Puhtaan tavan käyrän funktio on, kuten edellä mainittiin, muodostettu jakamalla oppikirjamaisen tavan funktio suoraviivaisen tavan funktiolla. Näin ollen puhtaan tavan kuvaajan funktio on $(0,71x + 131,3) / (0,24x + 90,3)$.

Kuvan 6.7 puhtaan tavan käyrää tarkasteltaessa voidaan havaita, että alussa suorituskykyeron kasvu on hyvin voimakasta. Siirryttäessä suurempiin tietomääriin eron kasvukäyrä loivenee. Alun voimakas kasvu johtuu testausohjelmiston käyttämän minimiajan suhteellisesta vaikutuksesta ohjelman kokonaissuoritusaikaan. Suuremmissa tietomäärissä tämä suhteellinen vaikutus vähenee ja lähestyy vertailtavien funktioiden kulmakertoimista pääteltävissä olevaa suorituskykyeron kasvua. Toisin sanoen suorituskykyeron kasvu tasoittuu, mutta ei koskaan pysähdy, lähestyttäessä ääretöntä tietomäärää. Tosin tässä on huomioitava, että siirryttäessä riittävän suuriin tietomääriin, on odotettavissa ongelmia käytettävän laitteiston kapasiteetissa. Esimerkiksi tietokannan käyttäytyminen voi muuttua arvaamattomaksi, kun tietokantaan viedään yhden transaktion sisällä riittävän paljon tietoa.

Edellä mainittiin, että esitelty puhtaan tavan suorituskykyjen eroa kuvaava funktio kuvaa tätä eroa vain yhdessä erikoistapauksessa. Tässä esitellään funktio yleisemmässä muodossa. Alakohdassa 6.1.1 esiteltiin periaate yksittäisten aineistojen kuvaajien funktioiden laskemiseksi. Sen mukaan jokaisen aineiston kuvaajan funktio on muotoa $f(x) = kx + m$. Tässä käytetään funktiota $f(x)$ kuvaamaan suoraviivaisen tavan aineiston kuvaajaa ja määritellään oppikirjamaiselle tavalle funktio $g(x) = lx +$

n. Kuvassa 6.7 esitellyn suorituskykyjen suhdetta kuvaavan funktion nimeämme tässä $p(x)$:si ja se on muotoa $p(x) = g(x) / f(x)$. Näin saamme yleisen suorituskykyjen eroja kuvaavan funktion $p(x) = (lx + n) / (kx + m)$, missä x on tietokantaan syötettyjen rivien määrä, l on oppikirjamaisen tavan kulmakerroin, k suoraviivaisen on tavan kulmakerroin, n on testausohjelmiston kuluttama minimiaika oppikirjamaisessa tapauksessa ja m testausohjelmiston kuluttama minimiaika suoraviivaisessa tapauksessa.

6.4 Johtopäätökset

Edellä on esitelty tulosten analysointia ja erilaisten käyttötapojen vertailua sekä niistä tehtyjä johtopäätöksiä. Tässä kohdassa tehdään yhteenveto tulosten analysoinnista ja vertailusta. Lisäksi arvioidaan tulosten oikeellisuutta.

6.4.1 Vertailun yhteenveto

Tutkielman lähtökohtana oli kaksi erilaista tapaa käyttää EJB-komponenttimallia vietäessä tietoa tietokantaan. Näille käyttötavoille määriteltiin käsitteet suoraviivainen ja oppikirjamainen tapa käyttää EJB:tä (ks. alakohta 3.2.2). Suoraviivaisessa tavassa tieto viedään tietokantaan käyttäen SQL-komentoa istuntopohjaisessa komponentissa ja oppikirjamaisessa tavassa käytetään vientiin kohdepohjaista komponenttia. Ensimmäisessä luvussa mainittiin, että kohdepohjainen komponentti tuo mukanaan tiettyjä ominaisuuksia verrattuna suoran SQL-komennon käyttöön. Näistä ominaisuuksista tiedon tallennuksen ja käsittelyn välinen synkronisointi lienee tärkein. Edelleen ensimmäisessä luvussa todettiin tutkielman tarkoituksena olevan muun muassa selvittää, kuinka merkittäviä edellä mainittujen EJB:n käyttötapojen suorituskykyerot ovat siirryttäessä yhä suurempiin tietomääriin. Alakohdassa 5.1.1 mainittiin suorituskykyjen erojen merkittävyyden olevan suhteessa siihen, kuinka paljon arvostetaan suorituskyvyn nopeutta ja kuinka paljon hitaamman tavan tuomia muita hyötyjä. Edelleen siinä mainittiin, että tutkielmassa ei pyritä löytämään mitään tiettyä rivimäärää, jolloin suorituskykyjen ero muodostuisi merkittäväksi vaan sen sijaan pyritään hankkimaan riittävästi tietoa tällaisen rivimäärän löytämiseksi.

Suoraviivaisen ja oppikirjamaisen tavan tulosten vertailusta voidaan selvästi havaita, että kohdepohjaisen komponentin käyttö suoran SQL-lauseen sijaan hidastaa selvästi ohjelman toimintaa siirrettäessä suuria määriä tietoa tietokantaan. Siirryttäessä yhä suurempiin tietomääriin, on kohdepohjaisen komponentin käyttö lähes kolme kertaa hitaampaa kuin SQL-lausetta käyttäen, jos tietoa ei käsitellä ennen vientiä. Jos taas tietoa käsitellään ennen tietokantaan vientiä, tasoittuvat nämä suorituskykyerot jonkin verran. Suorituskykyjen erot ovat nähtävissä jo pienissä, alle sadan tietokantarivin tietomäärissä. Niissä erot ovat kuitenkin ajallisesti niin pienet, ettei niillä ole juurikaan merkitystä koko ohjelmiston suorituskyvyn kannalta.

Tutkielman tuloksista voidaan todeta, että pienissä tietomäärissä suorituskykyyn kannattaa kiinnittää huomiota ainoastaan sellaisissa erikoistapauksissa, joissa suorituskyvyn optimointi on erityisen tärkeää. Kaikissa muissa tapauksissa valinta EJB:n käyttötapojen välillä kannattaa tehdä sen perusteella, tarvitaanko alakohdan alussa mainittuja kohdepohjaisen komponentin ominaisuuksia. Kun kyseessä on isot tietokannan rivimäärät, kannattaa suorituskykyyn kiinnittää enemmän huomiota. Tällaisissa tapauksissa kannattaa analysoida suorituskykyeroja kohdassa 6.3 esitetyn funktion $p(x)$ avulla etenkin sellaisissa tapauksissa, joissa tietoa ei käsitellä istuntopohjaisen komponentin toimesta ennen vientiä tietokantaan. Jos tietoa käsitellään ennen tietokantaan vientiä, kannattaa suorittaa vielä perusteellisempaa testausta kohdassa 3.2 esitetyllä tavalla. Pienen ja ison tietomäärän raja on hyvin tapauskohtainen. Testitulosten perusteella voitaneen kuitenkin tehdä karkea arvio, että alle sadan tietokantarivin määriä voidaan pitää kaikissa tapauksissa pieninä ja sellaisissa tapauksissa komponenttien valinta kannattanee perustella ennemmin lisäominaisuuksien tarpeella kuin suorituskyvyllä.

6.4.2 Tulosten merkityksen arviointia

Tutkielman alussa ja sen aikana ilmeni näkökulmia ja muita seikkoja, joiden huomioiminen tutkielmassa olisi tarkentanut tutkimustuloksia, lisännyt tulosten yleistettävyyttä ja/tai parantanut laadullisesti tutkielmaa kokonaisuutena. Tutkielman laajuudesta (tai suppeudesta) ja tiukahkosta aikataulusta johtuen kuitenkin pysyttiin

melko tarkasti alkuperäisissä suunnitelmissa ja rajauksissa. Tässä alakohdassa tehdään lyhyt katsaus keskeisimpiin edellä kuvatun kaltaisiin seikkoihin ja arvioidaan mitä lisäarvoa ne mahdollisesti olisivat tutkimukselle antaneet.

Alakohdassa 3.2.2 todettiin EJB-komponentteja käytettävän testauksessa lähiverkon yli. Tätä perusteltiin sillä, että hajautus on EJB-komponenttimallin keskeisimpiä piirteitä. Sinänsä perustelu on looginen, etenkin kun ottaa huomioon sen, että lähiverkon saattoi alussa olettaa tuovan vain vakiomääräisen tai lähes vakiomääräisen lisän testausohjelmiston suoritusaikaan. Kuitenkin myöhemmin havaittiin, että tauko testauksessa vaikutti lähes säännönmukaisesti uudelleen aloitetun testin ensimmäiseen tulokseen. Olisi ollut mielenkiintoista tietää, johtuiko tämä ilmiö mahdollisesti lähiverkon toiminnasta, ja testaaminen sekä lähiverkon yli että paikallisesti olisi tuonut tähän vastauksen. Tutkielman koko olisi kuitenkin laajentunut merkittävästi tällaisen lisäyksen johdosta, joten lisätestauksia ei lähdetty suorittamaan.

Tutkielma rajoittui tarkastelemaan EJB:n eri käyttötapoja ainoastaan Oracle9iAS-palvelinohjelmistossa. Useamman palvelinohjelmiston käyttäminen testauksessa olisi tuonut huomattavasti yleistettävämmän ja laajemman näkökulman tutkielmalle. Samalla kuitenkin tutkielman luonne ja laajuus olisivat siirtyneet aivan eri mittasuhteisiin, eikä niin laajaan tutkielmaan ollut resursseja. Huomioitavaa tässä kuitenkin on, että testausohjelmisto on ohjelmakoodeja muuttamatta siirrettävissä mihin tahansa EJB-komponenttimallin version 1.1 tai uudemman määrityksen toteuttavaan palvelinohjelmistoon. Näin ollen tässä tutkielmassa toteutettu testaus olisi kohtuullisen helposti toistettavissa jossain toisessa ympäristössä.

Tutkielman testauksessa keskityttiin EJB-komponenttimallin version 1.1 ominaisuuksiin. Mallin versio 2.0 tuo kuitenkin joitakin sellaisia ominaisuuksia, jotka vaikuttavat kohdepohjaisen komponentin suorituskykyyn. Erityisesti alakohdassa 3.2.2 mainitun paikallisen rajapinnan käyttö kohdepohjaisissa komponenteissa olisi mielenkiintoinen lisä tämän tutkielman testauksen viitekehykseen. Tutkielman teon ajankohtana OC4J-säiliö ei kuitenkaan vielä tukenut tätä paikallista rajapintaa, eikä se siten ollut testattavissa.

7 YHTEENVETO

EJB-komponenttimalli on hyvin monipuolinen ja monitahoinen ratkaisu hajautettuihin, komponenttipohjaisiin sovelluksiin. Tutkimuksen kohteena tutkielmassa oli tämän mallin yksi mahdollinen, joskin erikoislaatuinen sovellusalue, suurten tietomäärien vienti tietokantaan. Kyseessä on verrattain erikoislaatuinen tapa käyttää EJB-mallia, joka ei välttämättä tarjoa kovin pitkälle kehitettyjä ratkaisuja tai palveluja tämän tyyppiseen käyttöön. Malli tarjoaa kuitenkin paljon ominaisuuksia yleiseen tietokannan käsittelyyn, ja ottaen huomioon mallin suuren suosion ja laajan käyttökentän on arvattavaa, että suurten tietomäärien vienti tietokantaan tulee joissain mallia käyttävissä sovelluksissa välttämättä tarpeelliseksi. Tällaisissa tapauksissa herää myös kysymyksiä liittyen tiedon viennin tehokkuuteen, koska usein juuri suurten tietomassojen käsittely ja siirtäminen muodostuvat ohjelmiston suorituskyvyn kannalta kriittisiksi alueiksi.

Tutkimuksen taustalla oli siis suurten tietomassojen vienti tietokantaan EJB-komponenttimallin ratkaisuja käyttäen. Vielä tarkennettuna, kohteena oli kahden merkittävimmän mallin tarjoaman tietokannan käsittelytavan tehokkuuden vertailu. Toisen tavan todettiin olevan oppikirjamaisempi, laadullisesti parempia palveluja tarjoava tapa. Toisen tavan todettiin olevan suoraviivaisempi ja ennalta arvaten tehokkaampi. Keskeisimpänä mielenkiinnon kohteena tutkielmassa oli, kuinka merkittävästi tämä suoraviivaisempi tapa on tehokkaampi kuin oppikirjamainen tapa ja millaisissa tietomäärissä. Lisäarvoa tähän vertailuun haettiin kahdesta muusta näkökulmasta. Pienen ja ison mittakaavan testauksissa pyrittiin selvittämään, ovatko nämä suorituskykyerot nähtävissä jo pienissä tietomäärissä. Lisäksi tehtiin käyttötapoihin jako puhtaaseen ja käytännön tapaan, joista jälkimmäisessä pyrittiin löyhästi mallintamaan tyypillistä käytännön mallia käyttää EJB-komponettimallia ja sen kautta arvioimaan käytännön tavan vaikutuksia suorituskykyeroihin.

Jotta edellä kuvattuihin tavoitteisiin pääsy oli mahdollista, oli luotava ympäristö ja viitekehys edellä kuvattujen ominaisuuksien testaamiseen. Näin muodostui tutkielman rakenteellinen kahtiajako, jossa alkuosassa luotiin konstruktio edellä mainitun testauksen viitekehyksen kautta. Tämä tutkimusotteeltaan konstruktiivinen osio

sisältää katsauksen EJB-komponenttimalliin, testauksen viitekehyksen määrityksen ja testausympäristön kuvauksen. Tutkielman toisessa, empiirisessä osiossa suoritettiin testaus sekä analysoitiin ja vertailtiin testin tuloksia. Konstruktiivisen osion tuotokset ovat varsin yleiskäyttöisiä. Sen keskeisimmät osat, testauksen viitekehys ja testausohjelmisto, ovat helposti käytettävissä myös tämän tutkielman ulkopuolella. Myös konstruktiiviseen osioon liittyvät ohjelmakoodit, jotka on esitelty liitteessä yksi, ovat varsin pienin muutoksin, osin muuttamattomanakin, käytettävissä tutkielman ulkopuolella. Empiirisen osion testaustulokset ovat tapauskohtaista materiaalia johtuen tutkimuksen rajoittumisesta yhteen EJB-palvelinohjelmistoon. Kuitenkin empiirisen osion analyysin ja vertailun viitekehykset ovat sovellettavissa muihinkin EJB-ympäristöihin.

Tutkielman alussa määritellyt tavoitteet saavutettiin hyvin. Kun todettiin, miten merkittävyys suorituskykyeroissa oli määriteltävissä, voitiin todeta suorituskykyeroja kuvaavan kaavan olevan riittävä osoitus suorituskykyerojen merkittävyydestä. Tällainen kaava saatiin testaustulosten luotettavuuden toteamisen kautta muodostettua. Lisäksi kaavasta vielä esiteltiin yleinen muoto, jota tietyin rajoituksin voi soveltaa myös tutkielman ulkopuolella. Myöskin tutkielmalle lisäarvoa hakevien näkökulmien huomioiminen tutkielmassa osoittautui melko hyödylliseksi. Pienen ja ison mittakaavan testauksen kautta voitiin suorittaa karkeaa arviointia siitä, millaisissa tietomäärissä kannattaa alkaa kiinnittää huomiota näiden EJB-komponenttien käyttötapojen suorituskykyeroihin. Edelleen kyettiin arvioimaan, miten tämän tutkimuksen tuloksiin on suhtauduttava pienissä tietomäärissä ja jopa luomaan yleistä ohjeistusta EJB-komponenttien käytön valinnalle pienissä tietomäärissä. Käytännön tavan tuominen mukaan testaukseen osoitti lähinnä EJB-mallin käytön käytännössä olevan niin moninaista, että sellaisesta on vaikea muodostaa keskiarvoista käytännön tapaa. Kuitenkin voitiin havaita, että käytännön malli EJB-komponenttien käytöstä vaikuttaa vähentävästi puhtaan ja oppikirjamaisen EJB:n käyttötavan suorituskykyeroihin.

Paitsi konkreettisia tuotoksia ja tuloksia, tutkielma tuotti myös joukon kysymysmerkkejä, mahdollisia jatkotutkimuksen kohteita. Mahdollisen jatkotutkimuksen suuntia on useita. Tutkimuksen valitettava rajoittuminen EJB-komponenttimallin versioon 1.1, jätti avoimeksi version 2.0 tuomien kohdepohjaisen

EJB-komponentin tehokkuutta lisäävien ominaisuuksien vaikutuksen. Toisaalta rajoittuminen vain yhden erikoisen EJB:n käyttötavan tutkimiseen, jätti tekijän mieleen ajatuksen testauksen viitekehyksen mahdollisesta soveltumisesta myös joihinkin muihin EJB-komponenttimallin ominaisuuksien testaamiseen. Kolmas helposti havaittava tutkimuksen laajentamisen mahdollisuus on testata useampia EJB-palvelinohjelmistoja. Osittain näitä mahdollisia laajennoksia olisi voitu huomioida jo tutkielmaa suunniteltaessa ja sisällyttää tutkielmaan siten, ettei tutkielma olisi muodostunut liian laajaksi ollakseen pro gradu –tutkielma. Tämä olisi kuitenkin vaatinut tekijältä huomattavasti laajempaa aihealueen tuntemusta ja asioiden jäsentyneisyyttä tutkielmaa aloitettaessa.

LÄHDELUETTELO

D'Souza, D. F. & Wills, A. C. 1998. Objects, Components and Frameworks with UML – The Catalysis Approach. Reading, MA: Addison-Wesley.

DeMichiel, L. G., Yalcinalp, L. Ü. & Krishnan, S. 2001, Enterprise JavaBeans Specification, v2.0. Sun Microsystems Inc.

Jaaksi A., Aalto J-M., Aalto A.& Vättö, K. 1999. Tried & True Object Development. Cambridge University Press.

Kankainen, A. & Högmander, H. 2001. Tilastotieteen peruskurssi 1. Matematiikan ja tilastotieteen laitos, Jyväskylän yliopisto.

Koch, G. & Loney, K. 1997. Oracle8: The Complete Reference. Osborne/McGraw-Hill Co.

Kozaczynski, W. 1999. Composite Nature of Component. Second International Workshop on CBSE in conjunction with ICSE99. 17.-18.5.1999. Los Angeles, Yhdysvallat.

Monson-Haefel R. 2000. Enterprise JavaBeans, Second Edition. O'reilly & Associates, Inc.

Monson-Haefel R. 2001. Enterprise JavaBeans, Third Edition. O'reilly & Associates, Inc.

Oracle 2001, Oracle9i Application Server. Oracle Co [online]. [viitattu 8.3.2002]. Saatavilla pdf-muodossa <http://technet.oracle.com/tech/java/oc4j/pdf/OC4J_TWP.pdf>

Orion 2001, Documentation for Orion Application Server. Orion [online]. [viitattu 8.3.2002]. Saatavilla www-muodossa <http://www.orionserver.com/docs/index.html>

Riukulehto, T. & Huhtala, K. 1992. Tilastomenetelmien peruskurssi. Tilastotieteen laitos, Jyväskylän yliopisto.

Sun Microsystems Inc. 2001. Java Remote Method Invocation Specification.

Vainio, V. 2001. Enterprise JavaBeans –komponenttimalli ja sen vaikutus ohjelmistonkehitykseen. Jyväskylän yliopisto.

Yacoub, S., Ammar, H. & Mili, A. 1999. Characterizing a Software Component. Second International Workshop on CBSE in conjunction with ICSE99. 17.-18.5.1999. Los Angeles, Yhdysvallat.

Tässä liitteessä esitellään tutkielman testauksessa käytetyn testausohjelmiston ohjelmakoodit. Luokista on esitelty pelkkä ohjelmakoodi, EJB-komponenteista on esitelty myös koti- ja etärajapinta.

//Pääohjelma:

```
package main;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.NodeList;
import java.io.*;
import java.util.*;
import java.lang.*;
import org.xml.sax.SAXException;
import oracle.xml.parser.v2.*;
import main.*;

public class main extends Object
{
  public static void main(String[] args)
  {
    System.out.println("Start\n");

    DBInterface db = new DBInterface(); // EJB's are controlled thru this
    ConstValues con = new ConstValues();
    DOMParser parser = new DOMParser();
    String file = new String();
    file = "d://Proto/testData/" + args[0];
    try
    {
      InputStream str = new FileInputStream(file); // read testing data
      parser.parse(str); // parse testing data
      eDocument doc = new eDocument(parser.getDocument());
      db.insert(doc, new Integer(args[1]).intValue(), args[0]); // start testing
    }//try
    catch(Exception e)
    {
      System.out.println("Exception: " + e.getMessage());
    }

  }//-----------------------------------------------------------------------main

}//-----------------------------------------------------------------class-main
```

//Asiakasohjelman tietokantarajapintaluokka:

```
package main;

import org.w3c.dom.*;
import java.io.*;
import java.util.*;
import java.lang.*;
import java.text.*;
import org.xml.sax.SAXException;
import javax.ejb.*;
import javax.naming.*;
import java.rmi.RemoteException;
import oracle.xml.parser.v2.*;
import beans.*;

public class DBInterface extends Object
{
  ConstValues _const;
  FileOutputStream _logWriter;
  String _fileName;
  sEJBInsertHome _beanHome;
  sEJBParserHome _parserHome;

  public DBInterface()
  {
    _const = new ConstValues();
    _fileName = new String("d://Proto/logFile.txt");
    Hashtable table = getTable();
    try
    {
      Context initContext = new InitialContext(table);
      _beanHome = (sEJBInsertHome) initContext.lookup("sEJBInsert");
      _parserHome = (sEJBParserHome) initContext.lookup("sEJBParser");
      _logWriter = new FileOutputStream(_fileName, true);
    }//try
    catch(Exception e){}//catch
  }//-----------------------------------------------------------------constructor

  public void finalize()
  {
    try
    {
      _logWriter.close();
    }//try
    catch(IOException e){}//catch
  }//-------------------------------------------------------------------finalize

  private Hashtable getTable()
  {
    // Connection properties to JNDI are set here
    Hashtable table = new Hashtable();
    table.put("java.naming.factory.initial",
      "com.evermind.server.ApplicationClientInitialContextFactory");
    table.put("java.naming.provider.url",
      "ormi://sampo-nt/beans");
    table.put("java.naming.security.principal",
      "admin");
    table.put("java.naming.security.credentials",
      "admin");

    return table;
  }//-------------------------------------------------------------------getTable

  public void insert(eDocument doc, int type, String file) throws Exception
  {
    Hashtable header = new Hashtable();
    Vector data = new Vector();
    long start = 0, stop = 0; // Timing attributes
    Date date;
    String logDescription = new String();
    try
    {
      sEJBInsert bean = _beanHome.create();
      if(type == _const.INSERT_E || type == _const.INSERT_S)
      {
        sEJBParser parser = _parserHome.create();
```

```
        data = parser.parse(doc);
        header = (Hashtable) data.firstElement();
        data.removeElementAt(0);
      }//if parsing is done here
      switch(type)
      {
        case _const.INSERT_E : // entitybean insert, no parsing
          start = System.currentTimeMillis();
          bean.insert_e(header, data);
          stop = System.currentTimeMillis();
          logDescription = "Entitybean used, no parsing.";
          break;
        case _const.INSERT_EX : // entitybean insert with parsing
          start = System.currentTimeMillis();
          bean.insert_e(doc);
          stop = System.currentTimeMillis();
          logDescription = "Entitybean used with parsing.";
          break;
        case _const.INSERT_S : // sessionbean insert, no parsing
          start = System.currentTimeMillis();
          bean.insert(header, data);
          stop = System.currentTimeMillis();
          logDescription = "Sessionbean used, no parsing.";
          break;
        case _const.INSERT_SX : // sessionbean insert with parsing
          start = System.currentTimeMillis();
          bean.insert(doc);
          stop = System.currentTimeMillis();
          logDescription = "Sessionbean used with parsing.";
          break;
        default: break;
      }//switch

      date = new Date(System.currentTimeMillis());
      String temp =
        new String("\n" + date.toLocaleString() + "\t" + (stop-start) +
        " ms\t(" + logDescription + " File: " + file +")");
      _logWriter.write(temp.getBytes()); // write log-file
    }//try
    catch(Exception e)//CreateException e)
    {
      throw new Exception(e.getMessage());
    }//catch
    System.out.println("Done.");

  }//-----------------------------------------------------------------------insert

}//-------------------------------------------------------------class-DBInterface
```

//Asiakasohjelmassa käytetty vakioarvoja sisältävä luokka:

```
package main;

import java.io.Serializable;

public class ConstValues implements Serializable
{
  public final static String METERINGDATA = "meteringdata";
  public final static String HEADER = "header";
  public final static String MP = "mp";
  public final static String MSGTYPE = "msgtype";
  public final static String SENDTIME = "sendtime";
  public final static String CONCENTRATOR = "concentrator";
  public final static String COMMSERVER = "commserver";
  public final static String METERINGPOINT = "meteringpoint";
  public final static String PRODUCT = "product";
  public final static String MEASURETYPE = "measuretype";
  public final static String UNIT = "unit";
  public final static String MEASUREMENT = "measurement";
  public final static String PARAMETER = "parameter";
  public final static String PULSE_CONSTANT = "pulse_constant";
  public final static String TRANSFORMER = "transformer";
  public final static String MVALUE = "mvalue";
  public final static String TIMESTAMP = "timestamp";
  public final static String STATUS = "status";
  public final static String AMOUNT = "amount";
  public final static String DESCRIPTION = "description";

  public final static int INSERT_S = 1;
  public final static int INSERT_E = 2;
  public final static int INSERT_SX = 3;
  public final static int INSERT_EX = 4;

}//-----------------------------------------------------------class-ConstValues
```

```
//XML-dokumentin välittäjäluokka:

package main;

import oracle.xml.parser.v2.XMLDocument;
import org.w3c.dom.Document;
import java.io.Serializable;

public class eDocument implements Serializable
{
  XMLDocument _doc;
  public eDocument(XMLDocument doc)
  {
    _doc = doc;
  }//-----------------------------------------------------------------constructor

  public Document getDocument()
  {
    return _doc;
  }//-----------------------------------------------------------------getDocument

}//-----------------------------------------------------------------class-eDocument
```

//XML-dokumentin purkajakomponentti:

```
package beans;

import main.*;
import javax.ejb.*;
import java.util.*;
import org.w3c.dom.*;
import oracle.xml.parser.v2.*;

public class sEJBParserBean implements SessionBean
{
  public ConstValues c;

  public sEJBParserBean(){}

  public void ejbCreate() throws CreateException
  {
    c = new ConstValues();
  }

  public void ejbActivate(){}

  public void ejbPassivate(){}

  public void ejbRemove(){}

  public void setSessionContext(SessionContext ctx){}

  public Vector parse(eDocument data) throws Exception
  {
    Document doc = data.getDocument();
    Hashtable header = new Hashtable();
    Vector mdata = new Vector();
    Vector hdata = new Vector();
    try
    {
      header = getHeader(doc);
      hdata.add(header);
      mdata = getData(doc, hdata);
    }//try
    catch(Exception e){}//catch
    return mdata;
  }//-----------------------------------------------------------------------parse

  private Hashtable getHeader(Document doc) throws Exception
  {
    Hashtable data = new Hashtable();
    // document ->     version ->      meteringdata ->  header
    Node header = doc.getFirstChild().getNextSibling().getFirstChild();
    NamedNodeMap nList = header.getAttributes();
    if(nList.getLength() > 0)
      data.put(c.DESCRIPTION, nList.item(0).getNodeValue());
    header = header.getFirstChild();
    data.put(c.MSGTYPE, header.getFirstChild().getNodeValue());
    header = header.getNextSibling();
    data.put(c.SENDTIME, header.getFirstChild().getNodeValue());
    header = header.getNextSibling();
    data.put(c.COMMSERVER, header.getFirstChild().getNodeValue());
    header = header.getNextSibling();
    data.put(c.CONCENTRATOR, header.getFirstChild().getNodeValue());
    return data;
  }//-----------------------------------------------------------------getHeader

  private Vector getData(Document doc, Vector data) throws Exception
  {
    Vector all = new Vector(data);
    Hashtable p;
    Node node = doc.getFirstChild().getNextSibling().getFirstChild();
    Node sub = node.getNextSibling();
    do
    {
      p = new Hashtable();
      node = sub.getFirstChild();
      p.put(c.METERINGPOINT, node.getFirstChild().getNodeValue());
      node = node.getNextSibling();
      p.put(c.PRODUCT, node.getFirstChild().getNodeValue());
```

```
      node = node.getNextSibling();
      p.put(c.MEASURETYPE, node.getFirstChild().getNodeValue());
      node = node.getNextSibling();
      p.put(c.UNIT, node.getFirstChild().getNodeValue());
      node = node.getNextSibling();
      p.put(c.PULSE_CONSTANT, node.getFirstChild().getNodeValue());
      node = node.getNextSibling();
      p.put(c.TRANSFORMER, node.getFirstChild().getNodeValue());
      Node subsub = node.getNextSibling();
      Vector subVector = new Vector();
      Hashtable subp;
      do
      {
        subp = new Hashtable();
        node = subsub.getFirstChild();
        subp.put(c.MVALUE, node.getFirstChild().getNodeValue());
        node = node.getNextSibling();
        subp.put(c.TIMESTAMP, node.getFirstChild().getNodeValue());
        node = node.getNextSibling();
        subp.put(c.STATUS, node.getFirstChild().getNodeValue());
        subVector.add(subp);
        subsub = subsub.getNextSibling();
      } while(subsub != null); //do-while
      all.add(p);
      all.add(subVector);
      sub = sub.getNextSibling();
    } while(sub != null); //do-while
    return all;
  }//-------------------------------------------------------------------getData

}//-----------------------------------------------------class-sEJBParserBean

package beans;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;
import org.w3c.dom.*;
import main.eDocument;
import oracle.xml.parser.v2.*;

public interface sEJBParser  extends EJBObject
{
  public Vector parse(eDocument data) throws RemoteException, Exception;
}//-------------------------------------------------------interface-sEJBParser

package beans;

import java.rmi.*;
import javax.ejb.*;

public interface sEJBParserHome extends EJBHome
{
  public sEJBParser create() throws RemoteException, CreateException;
}//-----------------------------------------------------interface-sEJBParserHome
```

//Testauksen käyttötapoja hallinnoiva komponentti:

```
package beans;

import javax.ejb.*;
import javax.naming.*;
import javax.sql.*;
import java.rmi.RemoteException;
import java.sql.*;
import main.*;
import org.w3c.dom.Document;
import java.util.*;

public class sEJBInsertBean implements SessionBean
{
  public SessionContext ejbContext;
  public eEJBDataHome _data;
  public sEJBParserHome _parserHome;
  public ConstValues _c;
  public Connection _connection;
  public DataSource _ds;

  public sEJBInsertBean(){}

  public void ejbCreate() throws CreateException
  {
    _c = new ConstValues();
    _connection = null;
    _data = null;
    try
    {
      Context c = new InitialContext();
      _parserHome = (sEJBParserHome) c.lookup("sEJBParser");
      _ds = (DataSource) c.lookup("jdbc/OracleDS");
    }//try
    catch(Exception e){}//catch
  }//-----------------------------------------------------------------ejbCreate

  public void ejbActivate(){}

  public void ejbPassivate(){}

  public void ejbRemove(){}

  public void setSessionContext(SessionContext ctx)
  {
    this.ejbContext = ctx;
  }

  public void insert(Hashtable header, Vector data)
  {
    try
    {
      int i=0;
      if(_connection == null) _connection = _ds.getConnection();
      String sendtime = (String) header.get(_c.SENDTIME);
      String product, timestamp, status, mpid, value;
      Hashtable mpoint, mvalue;
      Vector values;
      PreparedStatement statement = _connection.prepareStatement(
        "insert into metering_data(mpid, timestamp, value, product," +
        " sendtime, status) values(?, ?, ?, ?, ?, ?)");
      do
      {
        mpoint = (Hashtable) data.firstElement();
        data.removeElementAt(0);
        values = (Vector) data.firstElement();
        data.removeElementAt(0);
        mpid = (String) mpoint.get(_c.METERINGPOINT);
        product = (String) mpoint.get(_c.PRODUCT);
        do
        {
          mvalue = (Hashtable) values.firstElement();
          values.removeElementAt(0);
          value = (String) mvalue.get(_c.MVALUE);
          timestamp = (String) mvalue.get(_c.TIMESTAMP);
          status = (String) mvalue.get(_c.STATUS);
```

```
        statement.setInt(1, new Integer(mpid).intValue());
        statement.setString(2, timestamp);
        statement.setInt(3, new Integer(mpid).intValue());
        statement.setString(4, product);
        statement.setString(5, sendtime);
        statement.setString(6, status);
        statement.execute(); // data insertion
        statement.clearParameters();
      } while(values.size() > 0); // do-while
    } while(data.size() > 0); // do-while
    statement.close();
    _connection.close();
    _connection = null;
  }//try
  catch(Exception e){}//catch
}//-----------------------------------------------------------------------insert

public void insert(eDocument doc)
{
  try
  {
    sEJBParser parser = _parserHome.create();
    Vector data = parser.parse(doc);          // First data parsing...
    Hashtable header = (Hashtable) data.firstElement();
    data.removeElementAt(0);
    insert(header, data);                     // ...then data insertion
  }//try
  catch(Exception e){}//catch
}//-----------------------------------------------------------------------insert

public void insert_e(Hashtable header, Vector data)
{
  try
  {
    String sendtime = (String) header.get(_c.SENDTIME);
    String product, timestamp, status, mpid, value;
    Hashtable mpoint, mvalue;
    Vector values;
    do
    {
      mpoint = (Hashtable) data.firstElement();
      data.removeElementAt(0);
      values = (Vector) data.firstElement();
      data.removeElementAt(0);
      mpid = (String) mpoint.get(_c.METERINGPOINT);
      product = (String) mpoint.get(_c.PRODUCT);
      do
      {
        mvalue = (Hashtable) values.firstElement();
        values.removeElementAt(0);
        value = (String) mvalue.get(_c.MVALUE);
        timestamp = (String) mvalue.get(_c.TIMESTAMP);
        status = (String) mvalue.get(_c.STATUS);

        if(_data == null)
        {
          Context c = new InitialContext();
          _data = (eEJBDataHome) c.lookup("eEJBData");
        }//if
        _data.create(new Integer(mpid).intValue(),
          new Integer(value).intValue(), sendtime, timestamp, status,
          product);

      } while(values.size() > 0); // do-while
    } while(data.size() > 0); // do-while
  }//try
  catch(Exception e){}//catch
}//-----------------------------------------------------------------------insert_e

public void insert_e(eDocument doc)
{
  try
  {
    sEJBParser parser = _parserHome.create();
    Vector data = parser.parse(doc);          // First data parsing...
    Hashtable header = (Hashtable) data.firstElement();
    data.removeElementAt(0);
```

```
      insert_e(header, data);                    // ...then data insertion
    }//try
    catch(Exception e){}//catch
  }//-------------------------------------------------------------------insert_e

}//-----------------------------------------------------------class-sEJBInsertBean

package beans;

import java.rmi.*;
import javax.ejb.*;
import java.util.*;
import main.*;

public interface sEJBInsert extends EJBObject
{
public void insert(Hashtable header, Vector data) throws RemoteException;
public void insert_e(Hashtable header, Vector data) throws RemoteException;
public void insert(eDocument doc) throws RemoteException;
public void insert_e(eDocument doc) throws RemoteException;
}//---------------------------------------------------------interface-sEJBInsert

package beans;

import java.rmi.*;
import javax.ejb.*;

public interface sEJBInsertHome extends EJBHome
{
  public sEJBInsert create() throws RemoteException, CreateException;
}//-------------------------------------------------------interface-sEJBInsertHome
```

```
//Testauksen kohdepohjainen komponentti:

package beans;

import javax.ejb.*;
import beans.*;

public class eEJBDataBean implements EntityBean
{
  public int MPID;
  public String TIMESTAMP;
  public String PRODUCT;
  public int VALUE;
  public String STATUS;
  public String SENDTIME;

  public eEJBDataBean(){}

  public eEJBDataPK ejbCreate() throws CreateException
  {
    return new eEJBDataPK();
  }

  public void ejbPostCreate(){}

  public void ejbActivate(){}

  public void ejbLoad(){}

  public void ejbPassivate(){}

  public void ejbRemove(){}

  public void ejbStore(){}

  public void setEntityContext(EntityContext ctx){}

  public void unsetEntityContext(){}

  public eEJBDataPK ejbCreate(int MPID, int VALUE, String SENDTIME,
    String TIMESTAMP, String STATUS, String PRODUCT) throws CreateException
  {
    setMPID(MPID);
    setSENDTIME(SENDTIME);
    setPRODUCT(PRODUCT);
    setSTATUS(STATUS);
    setTIMESTAMP(TIMESTAMP);
    setVALUE(VALUE);
    return new eEJBDataPK();
  }//-------------------------------------------------------------class-ejbCreate

  public void ejbPostCreate(int MPID, int VALUE, String SENDTIME,
    String TIMESTAMP, String STATUS, String PRODUCT){}

  public String getTIMESTAMP()
  {
    return TIMESTAMP;
  }

  public void setTIMESTAMP(String newTIMESTAMP)
  {
    TIMESTAMP = newTIMESTAMP;
  }

  public String getPRODUCT()
  {
    return PRODUCT;
  }

  public void setPRODUCT(String newPRODUCT)
  {
    PRODUCT = newPRODUCT;
  }

  public int getVALUE()
  {
    return VALUE;
```

```
  }

  public void setVALUE(int newVALUE)
  {
    VALUE = newVALUE;
  }

  public String getSENDTIME()
  {
    return SENDTIME;
  }

  public void setSENDTIME(String newSENDTIME)
  {
    SENDTIME = newSENDTIME;
  }

  public int getMPID()
  {
    return MPID;
  }

  public void setMPID(int newMPID)
  {
    MPID = newMPID;
  }

  public String getSTATUS()
  {
    return STATUS;
  }

  public void setSTATUS(String newSTATUS)
  {
    STATUS = newSTATUS;
  }

}//-----------------------------------------------------------class-eEJBDataBean

package beans;

import java.rmi.*;
import javax.ejb.*;
import beans.*;

public interface eEJBData extends EJBObject
{
public void setMPID(int newMPID) throws RemoteException;
public void setPRODUCT(String newPRODUCT) throws RemoteException;
public void setSENDTIME(String newSENDTIME) throws RemoteException;
public void setSTATUS(String newSTATUS) throws RemoteException;
public void setTIMESTAMP(String newTIMESTAMP) throws RemoteException;
public void setVALUE(int newVALUE) throws RemoteException;
public int getMPID() throws RemoteException;
public String getPRODUCT() throws RemoteException;
public String getSENDTIME() throws RemoteException;
public String getSTATUS() throws RemoteException;
public String getTIMESTAMP() throws RemoteException;
public int getVALUE() throws RemoteException;
}//-------------------------------------------------------------interface-eEJBData

package beans;

import java.rmi.*;
import javax.ejb.*;
import beans.*;

public interface eEJBDataHome  extends EJBHome
{
  public eEJBData create() throws RemoteException, CreateException;
  public eEJBData findByPrimaryKey(eEJBDataPK primaryKey)
    throws RemoteException, FinderException;
  public eEJBData create(int MPID, int VALUE, String SENDTIME,
    String TIMESTAMP, String STATUS, String PRODUCT)
    throws CreateException, RemoteException;
}//-----------------------------------------------------------interface-eEJBDataHome
```