Ville Vainio

ENTERPRISE JAVABEANS -KOMPONENTTI-MALLI JA SEN VAIKUTUS OHJELMISTON-KEHITYKSEEN

Tietojärjestelmätieteen
pro gradu –tutkielma
1.2.2001

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Vainio, Ville Veli
Enterprise JavaBeans -komponenttimalli ja sen vaikutus ohjelmistonkehitykseen / Ville Vainio.
Jyväskylä: Jyväskylän yliopisto, 2000.
118 s.
Tietojärjestelmätieteen pro gradu –tutkielma

Monitasoiset arkkitehtuurit ja ohjelmistokomponentit mahdollistavat laajojen, tietoverkkoja hyväksi käyttävien ja vahvasti liiketoimintaa tukevien tietojärjestelmien toteuttamisen yhä helpommin. Tässä tutkielmassa pyritään esittämään ohjelmistokomponentille kattava määritelmä kirjallisuudessa esiintyvien määritelmien pohjalta sekä esitetään komponenttipohjaiseen ohjelmistonkehitykseen liittyviä huomioita. Keskeisimmän osan tutkielmasta muodostaa Sun Microsystemsin Java-ohjelmointikieleen perustuvan Enterprise JavaBeans –komponenttimallin (*EJB*) käsittely, joka kuuluu osana suurempaan Java 2 Enterprise Edition –ohjelmointialustan (*J2EE*) kokonaisuuteen. J2EE-ohjelmointialustan ja EJB-komponenttimallin esittelyn lisäksi tutkimuksen tarkoituksena on esittää huomioita EJB-komponenttimalliin pohjautuvasta ohjelmistonkehityksestä sekä käytännön ohjeita sen soveltamiseksi.

Tutkielma on tutkimusotteeltaan käsitteellis-teoreettinen, ja sen keskeisin tuotos on selkeä määritys ohjelmistokomponentille ja komponenttimallille sekä esittely J2EE-ohjelmointialustasta ja EJB-komponenttimallista käytännön soveltamisohjeineen.

Esitettyjen komponenttien ja komponenttipohjaisen ohjelmistonkehityksen asettamien vaatimuksien sekä J2EE-ohjelmointialustan ja EJB-komponenttimallin käsittelyn tuloksena voidaan todeta, että J2EE-ohjelmointialusta tarjoaa hyvät mahdollisuudet monitasoisten komponentteja hyväksikäyttävien järjestelmien toteuttamiseen.

AVAINSANAT: monitasoarkkitehtuuri, ohjelmistokomponentti, komponenttimalli, Enterprise JavaBeans, Java 2 Enterprise Edition

SISÄLLYSLUETTELO

KUVIOIDEN LUETTELO

TAULUKKOJEN LUETTELO

ESIMERKKIEN LUETTELO

1 JOHDANTO

Monitasoarkkitehtuurien ja komponenttipohjaisen ohjelmistonkehityksen yhdessä luoma kehitys on johtamassa siihen, että ohjelmistokomponenttien avulla voidaan yhä enenevässä määrin ratkaista ohjelmistonkehityksessä usein toistuvia ja samankaltaisia ongelmia. Lisäksi jatkuvan kehitystyön tuloksena tarjolla on yhä käyttökelpoisempia ratkaisuja yritystason järjestelmien kehitystyön tueksi. Jatkuva uusien teknologioiden esiinmarssi on kuitenkin hämärtämässä todellisuutta, sillä uusienkin teknologioiden on sovelluttava johonkin tarkoitukseen ollakseen käyttökelpoisia. Tässä tutkielmassa esitetään eräs uusimmista ratkaisuvaihtoehdoista palvelimella tapahtuvaan komponenttipohjaiseen ohjelmistonkehitykseen.

Ohjelmistokomponentille on olemassa runsaasti erilaisia määritelmiä. Liiketoimintakomponenttina pidetään yleisesti sellaista ohjelmistokomponenttia, joka on itsenäinen jonkin liiketoimintakäsitteen tai -prosessin kuvaus. Tässä tutkielmassa on tarkoitus esittää ohjelmistokomponentille sellainen määritelmä, jota voidaan käyttää pohjana kehitettäessä liiketoimintakomponentteja hyväksi käyttävää, arkkitehtuuriltaan monitasoista, yritystasoista tietojärjestelmää. Lisäksi tarkoituksena on pohtia markkinoilla vallalla olevien komponenttimallien soveltumista liiketoimintakomponenttien käyttöön sekä sitä, miten komponentit vaikuttavat ohjelmistonkehitystyöhön.

Merkittävän osan tutkielmasta muodostaa Java 2 Enterprise Edition –ohjelmointialustan (*J2EE*) sekä sen sisältämien määritysten esittely. Tämän tutkimuksen kannalta merkittävin näistä määrityksistä on Enterprise JavaBeans –komponenttimallin (*EJB*) määritys, jonka tarkoituksena on helpottaa palvelimilla toimivien ohjelmistokomponenttien kehitystyötä ja mahdollistaa ohjelmistonkehittäjien aiempaa parempi keskittyminen liiketoimintalogiikan toteuttamiseen esimerkiksi tapahtumanhallinnan tai komponenttien elinkaaren hallinnan sijaan. J2EE-ohjelmointialustasta on EJB-komponenttimallin lisäksi esitelty myös joitain muita EJB-komponenttimallin hyödyntämisen kannalta keskeisimpiä määrityksiä. Omana kokonaisuutenaan käsiteltävän EJB-komponenttimallin käsittelyn tarkoituksena on esittää perusteet komponenttimallin hyötykäytölle.

Tutkielman loppu on omistettu EJB-komponenttimallin mukaisen ohjelmistonkehityksen käsittelyyn. Tarkoituksena ei ole esittää ainoaa ja oikeaa tapaa toteuttaa EJB-komponenttimallia hyödyntäviä järjestelmiä vaan pikemminkin avartaa näkökulmaa komponenttien avulla toteutettavaan ohjelmistonkehitykseen. Lisäksi on esitetty sekä yleisiä että vain EJB:lle ominaisia suunnittelu- ja toteutusperiaatteita, joiden hyödyntäminen on järkevää EJB-komponenttimalliin pohjautuvaa järjestelmää toteutettaessa.

Tutkielma pohjautuu kirjallisuuteen ja sen keskeisimpinä tuotoksina ovat selkeät määritykset ohjelmistokomponentille ja komponenttimallille. Lisäksi J2EE-ohjelmointialustan ja EJB-komponenttimallin esittely sekä käytännön soveltamisohjeet EJB-komponenttimallille ovat tutkielman keskeisiä tuotoksia.

Tutkielmassa käsiteltyjen määritysten sekä kirjallisuuden pohjalta esitettyjen huomioiden valossa EJB-komponenttimallin ja koko J2EE-ohjelmointialustan tulevaisuus näyttää valoisalta, sillä tällä hetkellä ei ole tarjolla mitään vastaavaa yhtä helposti siirrettävää ja valmiita toteutuksia tarjoavaa ratkaisua.

Tutkielma on rakennettu niin, että luvussa 2 käsitellään yleisiä monitasoarkkitehtuureihin liittyviä kysymyksiä ja luvussa 3 yleisiä komponentteihin ja niiden määritelmiin sekä komponenttimalleihin liittyviä asioita. Luvussa 4 esitellään J2EE-ohjelmointialusta ja sen keskeisimmät määritykset ja vastaavasti luvussa 5 pureudutaan tarkemmin EJB-komponenttimallin määrittelyn käsittelyyn. Luvussa 6 esitetään EJB-komponenttimalliin pohjautuvaan ohjelmistonkehitykseen liittyviä huomiota sekä esitetään joitain käytännön suunnittelu- ja toteutusperiaatteita. Lisäksi luvussa 6 esitetään lyhyt esimerkkisovellus. Tutkielman päättää yhteenveto, joka on esitetty luvussa 7.

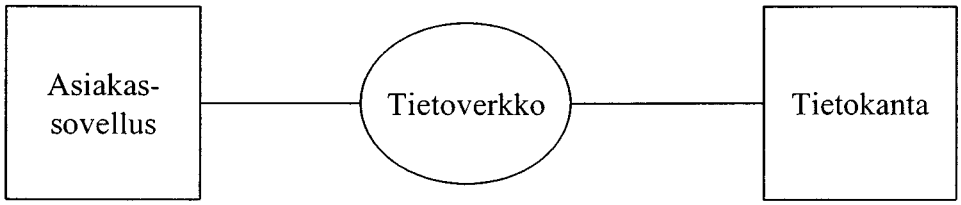
2 MONITASOARKKITEHTUURIT

Tämän luvun tarkoituksena on kartoittaa ja esitellä olemassa olevia monitasoarkkitehtuureja sekä esittää joitain esiteltyihin arkkitehtuureihin liittyviä etuja ja ongelmia. Lisäksi pohditaan lyhyesti, kuinka monitasoiset arkkitehtuurit vaikuttavat ohjelmistonkehitystyöhön.

Keskustietokonepainotteisesta ohjelmien suoritusympäristöstä on yhä enenevässä määrin siirrytty monitasoisten arkkitehtuurien käyttöön. Monitasoisella arkkitehtuurilla viitataan yleensä tietojärjestelmän jakamiseen toisistaan mahdollisimman vähän riippuviin loogisiin tasoihin. Tasojen fyysinen sijoituspaikka voi näin ollen vaihdella, loogisen jaon pysyessä ennallaan.

2.1 Kaksitasomalli

Kaksitasomallilla tarkoitetaan yleisesti sellaista ohjelmiston suoritusympäristöä, jossa asiakkaana toimivan sovelluksen kutsuihin vastaa jollain toisella tietokoneella toimiva palvelinsovellus (Orfali, Harkey & Edwards 1996a, 19). Tyypillinen esimerkki tällaisesta tietojärjestelmästä on esimerkiksi graafisen käyttöliittymän avulla käytettävä tietokantasovellus. Asiakasohjelma lähettää kyselyjä tietokantapalvelimelle, joka vastaa lähetettyihin kutsuihin ja palauttaa pyydetyt tietokannan tiedot. Kuviossa 1 on kuvattu edellisen kaltainen kaksitasoinen arkkitehtuuri, jossa asiakassovellus käyttää tietokantaa tietoverkon välityksellä.



KUVIO 1 Kaksitasomallin mukainen arkkitehtuuri

Edellä kuvatun kaltainen toimintamalli toteutetaan usein niin, että sovelluslogiikka ohjelmoidaan käyttöliittymään. Tällaista asiakasta kutsutaan kirjallisuudessa yleisesti lihavaksi asiakkaaksi (*fat client*), sen sisältämän runsaan sovelluslogiikan vuoksi (ks. esim. Orfali ym. 1996a). Sovelluslogiikkaa voidaan myös siirtää palvelimelle tai sitä voidaan pyrkiä jakamaan molempien tasojen kesken. Vähintään toinen tasoista kuormittuu kuitenkin sovelluslogiikan ohjelmoimisesta, ja näin pienetkin muutokset esimerkiksi tietokannan rakenteessa vaikuttavat kaikkien asiakkaana toimivien ohjelmien toimivuuteen.

Suurimmaksi eduksi kaksitasoiselle arkkitehtuurille voidaan lukea sen yksinkertaisuus, sillä useat työkaluohjelmistot tukevat tällaisen graafiseen käyttöliittymään perustuvan ja runsaasti sovelluslogiikkaa sisältävän sovelluksen rakentamista. Esimerkiksi pienimuotoiset päätöksenteontukijärjestelmät ja WWW-pohjaiset järjestelmät ovat usein tämän mallin mukaisia. (Edwards 1997, 6.)

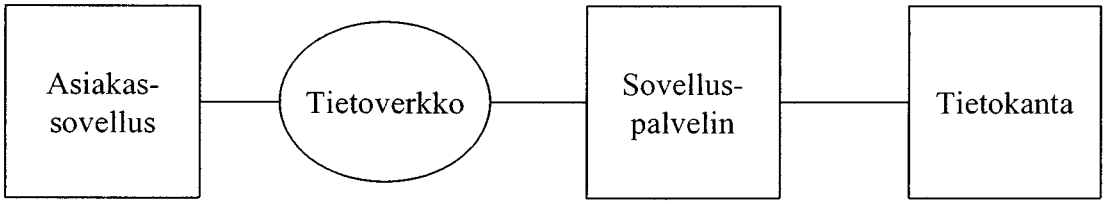
2.2 Kolmitasomalli

Kolmitasomalli on yritys vastata kaksitasomallissa esiintyneisiin suorituskyky- ja skaalautuvuusongelmiin. Kolmitasomallin keskeinen idea on jakaa koko sovelluksen arkkitehtuuri kolmeen osaan, jolloin sovelluslogiikan ohjelmoimista voidaan keskittää tietylle kerrokselle ja vastaavasti tehdä osasta tasoja mahdollisimman riippumattomia muiden tasojen toiminnasta ja näin vähentää eri tasojen välisiä kytkentöjä.

Kolmitasomallin eri tasot jaetaan useimmiten käyttöliittymän, liiketoimintalogiikan ja jaetut tiedot sisältäviin loogisiin tasoihin. Ensimmäisellä tasolla toimiva asiakaskäyttöliittymä ei sisällä sovelluslogiikkaa, eikä se tiedä alemmalla tasoilla olevista kerroksista muuta kuin ne rajapinnat, joiden kanssa sen on välttämätöntä olla yhteydessä. (Edwards 1997, 5; Orfali ym. 1996a, 30.) Liiketoimintalogiikan sisältävä keskikerros voi olla esimerkiksi sovelluspalvelin (*application server*), joka vastaanottaa kaikki asiakkaan pyynnöt ja huolehtii niiden asianmukaisesta toteuttamisesta. Näitä keskikerroksella toimivia ohjelmistoja kutsutaan yleisesti väliohjelmistoiksi (*middleware*) (ks. esim. Orfali ym. 1996a, 16-18). Jaettua tietoa sisältävällä kolmannella kerroksella tarkoitetaan vastaavasti useimmiten esimerkiksi tietokantaa tai tiedostopalvelinta, jossa sijaitsee kaikki

pitkäaikaisesti varastoitava, useiden asiakasohjelmien kesken jaettu tieto (Edwards 1997, 5; Orfali ym. 1996a, 30-31). Kolmitasomallin yhteydessä kirjallisuudessa asiakkaasta käytetään usein nimitystä laiha asiakas (*thin client*), asiakkaan sisältämän vähäisen sovelluslogiikan vuoksi. Vastaavasti, kun sovelluslogiikkaa siirretään toiselle tai kolmannelle tasolle, kutsutaan palvelimia lihaviksi palvelimiksi (*fat server*) niiden sisältämän sovelluslogiikan vuoksi. (ks. esim. Orfali ym. 1996a.)

Esimerkiksi edellä mainitussa kaksitasoisessa arkkitehtuurissa käyttöliittymän sisältämä sovelluslogiikka voitaisiin lähes kokonaisuudessaan siirtää liiketoimintalogiikan sisältävälle sovelluspalvelimelle, joka sijaitsisi keskikerroksella. Tämä keskimmäinen kerros suorittaisi tarvittavat asiakkaan vaatimat toimenpiteet yhteistyössä tietokantapalvelimen kanssa, joka sijaitsisi asiakkaan näkökulmasta kolmannella tasolla. Tietokantapalvelin ei kolmitasomallin mukaisesti sisältäisi itse tietokannan lisäksi mitään sovelluslogiikkaa. Kuviossa 2 on esitetty kolmitasoista arkkitehtuuria hyödyntävä järjestelmä, jossa asiakassovellus ei käytä tietokantaa suoraan vaan sovelluspalvelin huolehtii liikennöinnistä tietokannan kanssa. Liikennöinti asiakassovelluksen ja sovelluspalvelimen välillä tapahtuu tietoverkon välityksellä.



KUVIO 2 Kolmitasomallin mukainen arkkitehtuuri

Kolmitasomallin yleistyksenä voidaan pitää n-tasomallia, jossa loogisia tasoja on enemmän kuin kolme. Tällainen järjestelmä voi koostua esimerkiksi graafisesta käyttöliittymäkerroksesta, sovelluspalvelinkerroksesta sekä niin sanotusta näkymäkerroksesta, joka yhdistää useiden alemmilla tasoilla olevien tietokantojen sisältämiä tietoja palauttaa ne sovelluspalvelinkerrokselle jatkokäsittelyä varten. (Lewandowski 1998, 7.) Lisäksi esimerkiksi vanhojen järjestelmien integrointi uuden monitasoisen arkkitehtuurin osaksi saattaa aiheuttaa loogisten kerrosten lukumäärän kasvamisen kolmea suuremmaksi (Edwards 1997, 11-12).

2.3 Kaksi- ja kolmitasomallin vertailua

Kaksi- ja kolmitasomalliin perustuvien tietojärjestelmien luonne poikkeaa olennaisesti toisistaan. Kaksitasomallilla toteutettavat järjestelmät ovat melko yksinkertaisia, melko pienen käyttäjäjoukon tarvitsemia ja usein esimerkiksi jonkin organisaation yksikön tarpeita varten luotuja järjestelmiä.

Orfali ym. (1996a) vertailevat palvelimien avulla toimivia tietojärjestelmiä nimen omaan sen mukaisesti, mikä niiden käyttöympäristö ja käyttäjämäärä on. Heidän mukaansa muun muassa palvelinten lukumäärä ja niiden välinen kommunikaatio sekä palvelimien maantieteellisten sijaintien erot ja asiakkaan sisältämän sovelluslogiikan määrä vaikuttavat olennaisesti siihen, millainen luonne asiakas-palvelin –arkkitehtuuriin perustuvalla tietojärjestelmällä on. Orfali ym. (1996a) kutsuvat kaksitasomallin mukaisia, korkeintaan kaksi palvelinta sisältäviä järjestelmiä organisaatiossa osastokohtaisiksi järjestelmiksi. Vastaavasti useita palvelimia ja runsaasti niiden välistä kommunikaatiota sisältävä järjestelmä, jossa asiakkaat eivät sisällä juuri muuta kuin käyttöliittymän, he kutsuvat hieman mahtipontisesti linnunratojen väliseksi järjestelmäksi. (Orfali ym. 1996a, 19-22.) Tässä tutkielmassa edellä kuvatun kaltaisia, monitasoisia järjestelmiä kutsutaan yritystason järjestelmiksi, sillä niiden pääasiallinen käyttötarkoitus on vahvasti yritysten liiketoiminnan edistämisessä. Keskeisintä tässä vertailussa on se, että tietojärjestelmien luonne muuttuu ratkaisevasti, mikäli maantieteellisesti hajallaan olevien palvelimien ja asiakkaiden on voitava kommunikoida keskenään ja lisäksi voitava luottaa siihen, että niiden käsittelemä tieto on ajantasaista.

Edwards (1997) ja Roman (1999) esittävät joitain huomioita kaksi- ja kolmitasomalliin perustuvien järjestelmien välillä. Taulukossa 1 on esitetty mukaillen näissä vertailuissa mukana olleita tekijöitä. Skaalautuvuus paranee kolmitasomallissa, koska palvelinten lukumäärää voidaan joustavasti lisätä ilman asiakaspäässä tehtäviä muutoksia. Lisäksi tietokantoja voi olla käytössä useampia, ja jopa vanhoja järjestelmiä voidaan integroida osaksi uutta järjestelmää. Suorituskykyyn keskeisimmin vaikuttava tekijä on verkossa tapahtuvan liikennöinnin väheneminen. Asiakkaan ei esimerkiksi tarvitse lähettää kokonaisia kyselykielen lauseita palvelimelle ja tulkita mahdollisesti saatua kokonaista

tietokantataulua itse. Keskikerroksella oleva palvelin hoitaa suurimman osan liikennöinnistä tietokannan kanssa ja palauttaa vain asiakkaan tarvitsemat tiedot. Lisäksi koska palvelimet voivat kommunikoida keskenään, kuormantasaus suorituskyvyn parantamiseksi on mahdollista palvelimien lukumäärää nostamalla tai yksinkertaisesti lisäämällä palvelinten laskentatehoa. (Edwards 1997, 9-10; Roman 1999, 18-23.)

TAULUKKO 1 Kaksi- ja kolmitasomallin vertailua (mukailtu lähteistä Edwards 1997, 9-10 ja Roman 1999, 18-23)

Ominaisuus	**Kaksitasomalli**	**Kolmitasomalli**
Skaalautuvuus	Heikko	Erinomainen
Palvelinten välinen kommunikointi	Ei	Kyllä
Suorituskyky	Heikko	Hyvä
Tuki vanhemmille järjestelmille	Ei	Kyllä
Kehitystyön helppous	Korkea	Hyvä
Ylläpitokustannukset	Kohtuulliset	Korkeat
Tietoturva	Heikko	Korkea
Resurssien tehokas käyttö	Ei	Kyllä

Kehitystyön helppous on etu, joka kaksitasomallilla on puolellaan. Vaikka monitasoarkkitehtuurien kehittäminen työkalujen ja eri arkkitehtuurimallien avulla helpottuu koko ajan, on useamman tason kehittämisessä aina kiistämättä enemmän työtä kuin vain kahden toistensa kanssa vahvassa sidoksessa keskenään olevan kerroksen välillä. Ylläpitokustannusten kasvua voidaan pitää seurauksena useiden ohjelmistojen asentamisista ja päivittämisistä. Tietoturvaa voidaan parantaa helpommin monikantamalliin perustuvassa arkkitehtuurissa, sillä osa tasoista voidaan eristää ulkomaailmasta esimerkiksi palomuurien taakse. Kolmitasomalli mahdollistaa lisäksi tehokkaamman resurssien käytön, sillä esimerkiksi osa tarvittavista tietokantayhteyksistä voidaan pitää auki koko ajan eikä aikaa kulu turhiin yhteyksien avaamisiin ja sulkemisiin. Tällaisia niin sanottuja resurssipooleja (*resource pool*) käyttämällä resurssien jako ja käyttö tehostuu, ilman että asiakassovelluksen täytyy tietää siitä mitään. (Edwards 1997, 9-10; Roman 1999, 18-23.)

2.4 Monitasoarkkitehtuurit ohjelmistonkehityksessä

Edwards (1997) esittää Gartner Groupin luomaa ohjeistoa siitä, koska pitäisi käyttää kolmitasoarkkitehtuuria kaksitasoisen ratkaisun sijaan. Mitä monimutkaisempi ja pitkäikäisempi toteutettava järjestelmä on, sitä todennäköisempää on sen toteuttaminen kolmitasoarkkitehtuurin mukaisesti. Mikäli käytössä on vanhoja järjestelmiä, useita tietokantoja tai tapahtumien lukumäärä järjestelmässä on erittäin suuri ja järjestelmä tarjoaa useita eri palveluita, on syytä valita kolmitasoinen arkkitehtuuri järjestelmän pohjaksi. Edwards kuitenkin huomauttaa, että kaksitasoinen ratkaisu ei silti missään tapauksessa ole kuolemassa pois. Usein yksinkertaisen ja melko lyhytikäisen sovelluksen toteuttaminen ja ylläpito on sitä edullisempaa, mitä vähemmän toisistaan riippuvia loogisia tasoja on. (Edwards 1997, 16-17.)

Monitasoisen arkkitehtuurin toteutus vaatii eri tasojen tarvitsemien palvelujen toteuttamisen jollain loogisella tasolla. Nykyisin sovelluspalvelimet toteuttavat suurimman osan monitasoisessa arkkitehtuurissa tarvittavista yleisistä palveluista. Niiden avulla kytkeytyminen toisiin järjestelmiin ja eri valmistajien tietokantoihin on mahdollista ilman erillisten ja tapauskohtaisten ratkaisujen toteuttamista. Sovelluspalvelin sijaitsee arkkitehtuurissa keskimmäisellä tasolla, joten se huolehtii kaikkien asiakkaiden pyyntöjen välittämisistä eteenpäin sekä muun muassa tarvittavista muunnoksista järjestelmään kuuluvien eri sovellusten välillä. Sovelluspalvelimien käyttö ei kuitenkaan ole välttämätöntä, kuten ei useimpien muidenkaan väliohjelmistojen. Oleellista monitasoarkkitehtuuriin pohjautuvan järjestelmän kehitysvaiheessa onkin selvittää, ylittävätkö sovelluspalvelimen tuomat hyödyt sen mukanaan tuoman järjestelmän kehitystyön monimutkaisuuden ja mahdollisen suorituskyvyn laskun (King 1999, 26).

Koska sovelluspalvelimille ei ole olemassa minkäänlaista standardia, joka määrittelisi mitä palveluja niiden täytyy tarjota, on tärkeää selvittää kehitettävän järjestelmän tarpeet. Tärkeitä ominaisuuksia sovelluspalvelimelle ovat muun muassa sen skaalautuvuus, mahdollisuus kytkeytyä erilaisiin tietolähteisiin, tarjottujen palvelujen kattavuus, ainakin jonkin asteinen riippumattomuus valmistajasta sekä tuki vahvoille ja käytössäoleville standardeille (King 1999, 30).

2.5 Yhteenveto

Edellä esitetty jako kaksi- ja kolmitasoisiin arkkitehtuureihin on useimmiten riittävä. Kaksitasoinen ratkaisu on helpompi ja nopeampi toteuttaa, minkä lisäksi sen toteutuskustannukset ovat usein monitasoista ratkaisua pienemmät. Käytännössä pitkäikäisten ja laajennettavuutta vaativien järjestelmien vaihtoehdoksi kaksitasoisesta ratkaisusta ei kuitenkaan ole. Monitasoisen ratkaisun parempi skaalautuvuus ja selvästi tehokkaampi resurssien käyttö sekä mahdollisuus liittää jo olemassaolevat järjestelmät osaksi uutta järjestelmää ovat etuja, joita kaksitasoisessa ratkaisussa ei useinkaan voida saavuttaa.

Monitasoarkkitehtuurien toteuttaminen vaatii väliohjelmistojen käyttöä, joista yleisimmin käytössä ovat niin sanotut sovelluspalvelimet. Sovelluspalvelimet toteuttavat useita tehtäviä ja tarjoavat suoritusympäristön useille eri standardien mukaisille palveluille. Suurin ongelma sovelluspalvelimien käytössä on niiden standardoimattomuus, mikä vaikeuttaa muun muassa eri valmistajien tuotteiden käyttöä samassa järjestelmässä.

Monitasoarkkitehtuurien edut ovat useissa tapauksissa selvästi osoitettavissa. Tämän tutkielman kannalta oleellisinta on kuitenkin ymmärtää kolmi- ja useampitasoisten järjestelmien perusteet, ei niinkään sitä miksi jokin tietty yksittäinen ratkaisu olisi toista käyttökelpoisempi jossain tietyssä tilanteessa. Jatkossa tässä tutkielmassa käytetään termiä monitasoinen arkkitehtuuri kuvaamaan kolmi- tai useampitasoista arkkitehtuuria.

3 KOMPONENTTIPOHJAINEN OHJELMISTONKEHITYS

Edellisessä luvussa esitetty kaksi- ja kolmitasomallin vertailu antaa hyvän pohjan tutkia tarkemmin ohjelmistonkehityksessä käytettäviä rakennusosia. Tässä luvussa keskitytään esittämään ohjelmistokomponentin ja komponenttipohjaisen ohjelmistonkehityksen määritelmä. Ohjelmistokomponentista käytetään tässä tutkielmassa jatkossa myös lyhyempää nimitystä komponentti mutta molemmilla tarkoitetaan samaa asiaa.

Kirjallisuudesta esiintyneet ohjelmistokomponentin määritelmät on jaettu korkeamman ja matalamman tason määritelmiin, niiden abstraktioasteen mukaan. Korkeamman tason määritelminä pidetään sellaisia, jotka eivät ota tarkasti kantaa siihen, millainen komponentin fyysisen toteutuksen tulisi olla, vaan pikemminkin esittävät vaatimuksia, jotka komponentin olisi täytettävä. Alemman tason määritelmille on yhteistä niiden tiukempi kanta toteutusta koskeviin kysymyksiin, jolloin vastaavasti suuri joukko komponenttiehdokkaita rajautuu määrittelyn ulkopuolelle. Kirjallisuudessa esiintyneiden määritelmien avulla johdetaan tässä tutkielmassa käytettävä komponentin määritelmä.

Lisäksi luvussa otetaan kantaa liiketoimintakomponentin käsitteeseen, esitetään komponenttimallin määritelmä sekä esitetään joitain komponenttimallien yleisesti tarjoamia palveluja. Luvussa kuvataan lyhyesti kolme tällä hetkellä käytössä olevista komponenttimallista sekä pohditaan niiden sopivuutta liiketoimintakomponenttien käyttöön. Luvun lopuksi esitetään joitain huomioita komponenttipohjaisesta ohjelmistonkehityksestä.

3.1 Ohjelmistokomponentti

Ohjelmistokomponentille löytyy kirjallisuudesta useita toisistaan hyvinkin paljon poikkeavia määritelmiä. Yhä yleisemmin ohjelmistokomponenttien katsotaan olevan jonkin ohjelmointikielen tai käyttöjärjestelmän ominaisuuksia pikemminkin kuin ohjelmoinnillisia tai ohjelmistotuotannollisia rakenteita. Lisäksi usein läheisten käsitteiden, kuten esimerkiksi suunnittelumallien, olio-ohjelmoinnin tai arkkitehtuurien, käsittelyllä

samoissa yhteyksissä komponenttien kanssa on saatu aikaan vahva kytkentä perinteisten ohjelmistotuotantoon liittyvien ja uusien ohjelmistokomponenttien mukanaan tuomien käsitteiden välille.

Seuraavissa kohdissa on esitetty joitain kirjallisuudesta löytyviä määritelmiä ohjelmistokomponentille.

3.1.1 Korkeamman tason määritelmät

D'Souzan ja Willsin (1998) esittämän yleisen komponentin määritelmän mukaisesti komponentti on kokoelma ohjelmistojen kokoamisessa tarvittavia osia, jotka voidaan liittää muuttamattomana osaksi muita komponentteja ja joiden avulla luodaan jokin suurempi kokonaisuus. Tämän määritelmän mukaisesti esimerkiksi ohjelmointikielen lauserakenteet voidaan katsoa komponenteiksi. (D'Souza & Wills 1998, 386.)

Kozaczynskin (1999) esittämän määritelmän mukaisesti ohjelmistokomponentti on sellainen järjestelmän osa, joka on samanaikaisesti sekä suunnittelu- ja rakenneyksikkö että konfiguraation hallinnan ja vaihtokelpoisuuden yksikkö. Lisäksi ohjelmistokomponentti mukautuu rajapintoihin ja tarjoaa niiden toteutuksia hyvin määritellyssä järjestelmäarkkitehtuurissa. Kozaczynski korostaa mainittujen ominaisuuksien samanaikaisuutta merkittävänä tekijänä, sillä esimerkiksi jonkin olio-ohjelmointikielen luokka voi edustaa suunnitteluyksikköä mutta ei samanaikaisesti ole vaihtokelpoinen yksikkö. Hänen mukaansa ohjelmistokomponentti onkin ainoa yksikkö, jonka voidaan katsoa sisältävän kaikki edellämainitut ominaisuudet samanaikaisesti. Kozaczynski mainitsee lisäksi tämän määritelmän olevan yritys määritellä komponentti niin, että se ottaa huomioon komponentin luonteen käsitteellisenä osana sen suunnittelua, rakentamista ja julkaisua (Kozaczynski 1999, 2-4.)

Parrish, Dixon ja Hale (1999) esittävät ohjelmistokomponentille Kozaczynskin (1999) tapaan melko abstraktin määritelmän. Sen mukaisesti ohjelmistokomponentti on tuotos, joka koostuu kolmesta eri osasta, joita ovat palvelu- ja asiakasrajapinta sekä toteutus. Palvelurajapinta kuvaa niitä palveluja, joita komponentti tarjoaa ulkomaailmalle, kun

taas asiakasrajapinta kuvaa niitä muiden komponenttien tarjoamia palveluja, joita komponentti itse käyttää. Toteutuksella viitataan komponentin ohjelmakoodiin, joka on välttämätöntä, jotta komponentti voi toimia tarkoitetulla tavalla. Edellä kuvatun määritelmän mukaisesti komponentti voi olla esimerkiksi olio-ohjelmointikielen luokka, sillä palvelurajapintana toimii funktioiden prototyypit, asiakasrajapintana luokan käyttämät alemman tason järjestelmäkirjastot ja toteutuksena kaikkien eri funktioiden toteutukset.

Yacoub, Ammar ja Mili (1999) esittävät Parrishin ym. (1999) tapaan korkean abstraktioasteen määritelmän ohjelmistokomponentille. He esittävät joukon vaatimuksia, jotka luonnehtivat ohjelmistokomponentteja, ja jakavat nämä vaatimukset edelleen kolmeen luokkaan. Nämä luokat ovat epäformaali kuvaus sekä ulkoiset ja sisäiset piirteet. Koska komponentteja voidaan pitää ohjelmistojen inhimillisinä rakennusosina ja ohjeina ohjelmiston kehityksessä ilmeneviin ongelmiin, täytyy Yacoubin ym. (1999) mukaan komponenteille myös löytyä inhimillisiä piirteitä sisältäviä kuvauksia. Epäformaaleihin kuvauksiin kuuluvat heidän mukaansa muun muassa komponentin ikä, uudelleenkäytettävyyden mahdollinen vaihe ohjelmistonkehityksessä, käyttöyhteys, tarkoitus sekä muut komponentit, jotka ratkaisevat samankaltaisen ongelman. Ulkoiset piirteet määrittelevät vastaavasti komponentin vuorovaikutuksen muihin ohjelmiston osiin ja alustaan, jolla komponentteja käytetään. Tällaisia piirteitä ovat muun muassa siirrettävyys, käytetty teknologia, komponentin rooli ohjelmistossa, yhteistoiminnallisuus sekä vaihe, jossa komponentti sidotaan käytettävään ohjelmistoon. Sisäisiksi piirteiksi Yacoub ym. (1999) lukevat muun muassa pääsyn ohjelmakoodiin, komponentin luonteen ja rakeisuuden, jolla voidaan viitata esimerkiksi komponentin uudelleenkäytettävyyteen, kokoon tai sovellusalueen laajuuteen. Yacoubin ym. (1999) esittämä määritelmä sallii komponenttien olla lähes mitä tahansa rakenteita, jotka täyttävät esitetyt vaatimukset.

3.1.2 Alemman tason määritelmät

Kirjallisuudessa ehkä laajimmin esiintyvä ohjelmistokomponentin määritelmä on Szyperskin (1998) esittämä. Szyperskin (1998) määritelmän mukaisesti ohjelmistokomponentit ovat binäärimuodossa olevia yksiköitä, joiden itsenäinen tuotanto, hankinta ja julkistaminen muodostavat toimivan järjestelmän. Tämän määritelmän mukaisesti

komponentin on siis aina oltava binäärisessä, suorituskelpoisessa muodossa. Lisäksi Szyperskin mukaan komponentilla on sopimuksenmukaisia rajapintoja ja sillä on vain täsmällisesti määriteltyjä ulkoisia riippuvuuksia. Komponentti voidaan myös julkistaa itsenäisesti ja sitä voidaan käyttää järjestelmien rakennusosana. (Szyperski 1998, 34.)

Szyperski (1998) esittää myös Johannes Sametingerin (1997) määritelmän komponentille. Sen mukaisesti uudelleenkäytettävä ohjelmistokomponentti on itseriittoinen, selvästi tunnistettavissa oleva osa, joka kuvaa tai suorittaa jonkin tietyn tehtävän ja jolla on selkeästi määritelty rajapinta, tarkoituksenmukainen dokumentaatio ja määritelty asema uudelleenkäytön kannalta. Sametingerin näkemys poikkeaa melko vahvasti esimerkiksi Szyperskin itse esittämästä määritelmästä, sillä Sametingerin mukaan esimerkiksi lähdekoodi tai dokumentaatio voivat olla komponentteja. (Szyperski 1998, 168.)

D'Souza ja Wills (1998) esittävät aiemmin kohdassa 3.1.1 esitetyn komponentin määritelmän lisäksi komponentille myös alemman tason määritelmän. Sen mukaisesti komponentti on toteutus kokoelmalle ohjelmistojen kokoamisessa tarvittavia osia, joka voidaan tuottaa ja levittää itsenäisesti, jolla on hyvin määritellyt ja selvät rajapinnat tarjoamistaan ja muilta komponenteilta odottamistaan palveluista. Lisäksi komponentti tulee voida yhdistää muiden komponenttien kanssa ja sen sisäinen toiminnan muokkaamisen ei tule olla mahdollista. (D'Souza & Wills 1998, 387.)

Brown ja Wallnau (1998) esittävät joitain ICSE:n (*International Conference on Software Engineering*) työpajatoiminnassa esille nousseita määritelmiä komponentille. Phillippe Kruchtenin työpajatoiminnassa esittämän määritelmän mukaan komponentti on ei-triviaali, lähes itsenäinen ja korvattavissa oleva osa järjestelmää, joka toteuttaa jonkin selkeän tehtävän hyvin määritellyssä arkkitehtuurissa. Lisäksi komponentti on yhdenmukainen esittämiensä rajapintojen kanssa tarjoten näiden esitettyjen rajapintojen toteutuksen.

Lewandowski (1998) määrittelee komponentin pienimmäksi itsenäiseksi ja hyödylliseksi rakenneosaksi järjestelmässä, jonka tulee toimia moninaisissa ympäris-

töissä. Esimerkkinä hän mainitsee ActiveX-kontrollin, joten olioperustaisuus ei ole vaatimuksena komponentille. Lewandowski myös korostaa rajapintojen merkitystä komponentin toiminnallisuuden selvittämisessä ja hyväksikäytössä.

Orfali, Harkey ja Edwards (1996a) johtavat määritelmän komponentille pääasiassa hajautettujen olioiden avulla. Heidän mukaansa hajautetut oliot ovat väistämättä komponentteja, koska ne sisältävät yhdessä levitettävässä yksikössä toiminnallisuuden, jota hajautettu olio edustaa. Orfali ym. (1996a) esittävät joukon vaatimuksia, jotka minimalistisen komponentin on täytettävä. Komponentin on vaatimusten mukaan oltava avoimesti myytävissä oleva kokonaisuus, sitä on voitava käyttää ennustamattomissa yhteyksissä ja sillä on oltava hyvin määritelty rajapinta. Lisäksi komponentin tulee olla yhteistoiminnallinen muiden komponenttien kanssa ja olio-ohjelmoinnin mukainen olio siinä mielessä, että sen on tuettava muun muassa perintää ja polymorfismia. Komponentti ei myöskään saa olla sovellus. (Orfali ym. 1996a, 389-390.)

Orfali ym. (1996a) laajentavat komponentin määritelmää esittelemällä niin sanotun superkomponentin. Tämä superkomponentti on luonnollisesti minimalistinen komponentti, mutta sen lisäksi sen tulee täyttää myös muita vaatimuksia. Näitä vaatimuksia ovat muun muassa turvallisuus, lisensointi, versiointi, komponentin elinkaaren hallinta, tapahtumapohjainen tiedonvälitys, konfiguroinnin hallinta, tuki rajapintaa muokkaavalle skriptikielelle, metadatan säilytys, tapahtumien hallinta, pysyvyys ja helppokäyttöisyys. Orfalin ym. (1996a) mukaan komponentin ei itse tarvitse tarjota kaikkia edellä mainittuja palveluja vaan osa niistä voidaan toteuttaa yleisinä palveluina, joita kaikki tähän palveluun kytkeytyneet komponentit voivat hyödyntää. (Orfali ym. 1996a, 391-392.)

3.1.3 Liiketoimintakomponentti

Orfali ym. (1996a) määrittelevät liiketoimintakomponentin älykkääksi komponentiksi, joka mallintaa jotain tosielämän vastinetta jollakin sovellusalueella. Tyypillisesti tällaiset komponentit suorittavat joitain erityisiä liiketoimintatehtäviä (Orfali ym. 1996a, 393.) Liiketoimintakomponentti voi olla esimerkiksi hinnoittelukomponentti, joka osaa kulloinkin kyseessä olevan tuotteen perushinnan lisäksi laskea erilaisten alennusten sekä

muiden kulujen osuuden mukaan tuotteen kokonaishintaan. Tällöin samaa komponenttia voidaan käyttää muokattuna esimerkiksi verkossa toimivassa vähittäismyyntiliikkeessä ja autonvalmistajan verkkosivuilla tehtävien tarjouspyyntöjen laskennassa. (Roman 1999, 4.)

Brown ja Wallnau (1998) esittävät Wojtek Kozaczynskin määritelmän liiketoimintakomponentille. Sen mukaisesti liiketoimintakomponentti on itsenäinen liiketoimintakäsitteen tai -prosessin kuvaus. Komponentti koostuu kaikista niistä tarpeellisista osista, joita tarvitaan ilmaisemaan, toteuttamaan ja julkistamaan tämä käsite uudelleenkäytettävänä osana laajempaa liiketoimintajärjestelmää. (Brown & Wallnau 1998, 39.) Wallnau (1999) pitää Kozaczynskin määritelmää liiketoimintakomponentille sinänsä riittävänä mutta huomauttaa kuitenkin, että hänen mielestään varsinaisia markkinoita uudelleenkäytettäville liiketoimintakomponenteille ei voi syntyä. Syynä tähän on liiketoimintaprosessien ainutlaatuisuus ja se, etteivät markkinat hänen mukaansa tule koskaan hyväksymään täysin siirrettäviä komponentteja. Tämä näkökulma poikkeaa täysin muun muassa Szyperskin (1998), Romanin (1999) ja Lewandowskin (1998) esityksistä.

3.1.4 Esitetyistä komponenttien määritelmistä

Aiemmissa kohdissa kuvatuissa, kirjallisuudessa esiintyneissä komponenttien määritelmissä on runsaasti yhteisiä piirteitä, mutta myös useita toisistaan hyvinkin paljon poikkeavia piirteitä voidaan löytää.

Esimerkiksi Orfali ym. (1996a) esittävät, että komponentti itsessään ei voi olla sovellus. Szyperskin (1998) esittämä määritelmä vastaavasti rajoittaa komponentin koskemaan ainoastaan binääristä komponenttia, jolloin komponenttien hyödyntäminen on väistämättä niiden käyttöä ilman niiden muokkaamista. Lisäksi eroavaisuuksia löytyy siinä, pitääkö komponentin noudattaa oliomaailman vaatimuksia ja tukea esimerkiksi perintää, sekä siitä, mitä eroa on komponentilla ja olioilla.

Brown ja Wallnau (1998) esittävät olioparadigman olevan sekä tarpeeton että riittämätön komponenttipohjaisen ohjelmistotuotannon tarpeisiin. Tätä he perustelevat sillä, että olioteknologia ei itsessään ole kykenevä kuvaamaan kaikkia tarvittavia abstraktioita, joita komponenttipohjaisessa ohjelmistotuotannossa tarvitaan. Lisäksi komponenttipohjaista ohjelmistotuotantoa voidaan toteuttaa ilman olioparadigmaa. Tämä ei luonnollisestikaan tarkoita sitä, ettei olioparadigmaa voitaisi hyödyntää komponentteja kehitettäessä.

Parrish ym. (1999) toteavat, että Szyperskin (1998) esittämä määritelmä komponentille on riittämätön komponenttipohjaisen ohjelmistotuotannon pohjaksi, koska sen mukaisesti koko kyseinen ohjelmistotuotannon suuntaus olisi saanut alkunsa binääristen komponenttiteknologioiden kehityksestä. Samalla aiempi kehitys esimerkiksi olio-ohjelmoinnissa ja lähdekooditasolla tehdyssä komponenttikehityksessä olisi merkityksetöntä.

Vayda (1999) vertaa olioita ja komponentteja toisiinsa todeten muun muassa komponenttien olevan usein tilattomia toisin kuin olioiden. Yhtä lailla hänen mukaansa komponenteilla on löysemmät sidokset toisiin komponentteihin kuin olioilla. D'Souza ja Wills (1998) esittävät olioiden ja komponenttien eroavan siinä mielessä toisistaan, että olio on aina jonkin toimivan järjestelmän osa, jolla on oma identiteetti. Komponentti sitä vastoin on ihmisen ja hänen käyttämiensä välineiden aikaansaama tuotos, joka suoritusaikana koostuu olioiden ohjelmakoodista. Tässä mielessä olio ei siis ole komponentti. D'Souza ja Wills kuitenkin huomauttavat, että komponentin määritelmää on mahdollista käyttää löyhemmässä merkityksessä, sillä komponentti ilmenee ajonaikaisesti usein kokoelmana olioita. (D'Souza & Wills 1998, 390.)

Useasti esiintyviä vaatimuksia komponentille ovat siirrettävyys eri ympäristöihin, uudelleenkäytön mahdollistaminen ja komponentin vaihtokelpoisuus korvaavaan komponenttiin. Lisäksi useimmat määritelmät mainitsevat komponentin rajapinnan olevan eräs merkityksellisimmistä tekijöistä komponentin uudelleenkäytettävyyttä arvioitaessa.

Tässä tutkielmassa komponentin katsotaan olevan itsenäinen, suorituskelpoinen yksikkö, jonka tarjoamat palvelut ja ulkoiset riippuvuudet on täsmällisesti määritelty. Lisäksi

komponentti tulee voida korvata toisella samat palvelut toteuttavalla komponentilla ja sen yhdistämisen muihin komponentteihin tulee olla mahdollista. Komponentin oletetaan myös useimmiten olevan liiketoimintakomponentti siinä mielessä, että se on jonkin itsenäisen liiketoimintakäsitteen tai prosessin kuvaus. Edellä kuvatun mukaisesti määritelmässä on otteita ja linjauksia muun muassa Szyperskin (1998), D'Souzan ym. (1998) sekä Brownin ja Wallnaun (1998) esittämästä Wojtek Kozaczynskin määritelmistä.

3.2 Komponenttimalli

Komponenttimallilla (*component model*) tarkoitetaan perusrakennetta, joka tarjoamiensa palvelujen avulla mahdollistaa komponenttien välisen kommunikoinnin ja tarjoaa komponenteille niiden tarvitsemia palveluja. Komponenttimalli voi olla pelkkä määritys tai sen fyysinen toteutus.

Komponenttimallia kuvaamaan on kirjallisuudessa esitetty muitakin termejä. Näitä ovat muun muassa olioväylä (*Object Bus)* (Orfali ym. 1996a) ja komponenttimaailma (*Component World*) (Szyperski 1998). Szyperski (1998) käyttää termiä kuvaamaan lähinnä osittain kilpailevia, päällekkäisiä ja ristiriitaisia komponenttimalleja. Orfali ym. (1996a) vastaavasti käyttävät olioväylä-termiä suoraan sen teknisessä merkityksessä eli alimman tason palveluna sitä tarvitseville komponenteille.

Tässä tutkielmassa komponenttimallin katsotaan tarkoittavan pääosin Orfalin ym. (1996a) näkemyksen mukaan lähinnä komponenttien tarvitsemien palvelujen toteuttamista ja vastaavasti vaatimuksien asettamista komponenttien kehitystyöhön ja käyttöönottoon. Lisäksi komponenttimallia pidetään ennemminkin määrityksenä kuin sen yksiselitteisenä toteutuksena.

3.2.1 Komponenttimallien tarjoamat palvelut

Szyperski (1998) pohtii komponenttimallien tarjoamien palvelujen tarvetta aikaisempien binääritasolla toimivien kutsujen pohjalta. Aikaisemmin käytössä olleet tavat kuten esimerkiksi RPC-kutsut (*Remote Procedure Call*) ja käyttöjärjestelmän tarjoamat rajapinnat ovat riittäneet. Komponenttien ja niiden käyttötarkoituksien kehittyessä on kuitenkin väistämättä mietittävä myös muun muassa sitä, kuinka rajapinnat määritellään, kuinka viitteitä komponentteihin hallitaan sekä miten komponentti paikallistaa komponenttimallin tarjoamat palvelut. (Szyperski 1998, 173.)

Komponenttimallin komponenteille tarjoamia tyypillisiä palveluja ovat myös muun muassa komponenttien paikallistaminen, komponenttiviitteiden hakeminen, tapahtumanhallintapalvelut sekä tietoturvapalvelut (Brown & Wallnau 1998, 43). Lisäksi elinkaaren hallintaan, pysyvyyteen ja komponentin tilanhallintaan liittyviä palveluja voidaan toteuttaa komponenttimallilla (Thomas 1998, 11). Eri komponenttimallit tarjoavat erilaisia palveluja, sillä esimerkiksi tiettyyn käyttöjärjestelmään tai ohjelmointikieleen sidotut komponenttimallit asettavat väistämättä rajoituksia myös toteutettaville ja käytettäville komponenteille.

Komponenttien suoritusympäristön merkitys korostuu pohdittaessa komponenttien riippuvuuksia muista komponenteista tai suoritusympäristönä toimivan komponenttimallin toteutuksen eri palveluista. Szyperskin (1998) esittämän määritelmän mukaisesti komponentilla tulee olla vain täsmällisesti määriteltyjä ulkoisia riippuvuuksia, joten sidokset ulkoisiin rajapintoihin tulisi aina olla saatavilla. Kuitenkin Szyperski huomauttaa, että useimmiten pääpaino on komponentin toteuttamissa ja tarjoamissa rajapinnoissa, eikä tietoa kaikista komponentin itse tarvitsemista rajapinnoista ole aina edes tarjolla (Szyperski 1998, 35).

3.2.2 Esimerkkejä komponenttimalleista

Seuraavassa on esitelty joitain komponenttimalleja sekä niiden keskeisimpiä piirteitä. Tämän tutkimuksen kannalta komponenttimallien yksityiskohtainen selostus ei ole

mielekästä ja niinpä seuraavassa kuvattujen kolmen komponenttimallin esitys pohjautuukin pitkälti Szyperskin (1998) esittämään yleiseen esittelyyn.

OMG (*Object Management Group*) on esitellyt oman avoimen, hajautettujen järjestelmien toteuttamiseen tarkoitetun komponenttimallin määrittelyn, CORBAn (*Common Object Request Broker Architecture*). CORBA-komponenttimallia ei ole sidottu mihinkään ohjelmointikieleen, vaan esimerkiksi asiakas ja palvelin voivat olla toteutettuina millä tahansa kielellä, eikä tämä näy kummallekaan tapahtumaan osallistuvalle oliolle. CORBAssa komponentin määritelmä on melko yleisellä tasolla, sillä mikä tahansa suoritettava ohjelma kelpaa komponentiksi. Niinpä CORBAn tapauksessa käytetäänkin yleisemmin olio-termiä kuvaamaan asiakas-palvelin –suhteen osapuolia. Rajapintojen kuvaamista varten on kehitetty oma kuvauskieli OMG IDL (*Interface Definition Language*). (Szyperski 1998, 178-181.) IDL-kielellä tehdyt rajapintojen kuvaukset käännetään lähdekoodimuotoon kunkin ohjelmointikielen mukaisella kääntäjällä. Ainoa rajoite, joka käytettävää ohjelmointikieltä koskee on näin ollen se, että sille on olemassa IDL-kääntäjä.

COM (*Component Object Model*) on Microsoftin perusta kaikelle komponenttipohjaiselle ohjelmistonkehitykselle. COM on binäärinen standardi, joten mikä tahansa ohjelmointikieli, joka on käännettävissä Windows-ympäristöissä suoritettavaksi, kelpaa toteutuskieleksi. COM ei myöskään ota mitään kantaa komponenttiin käsitteenä, joten mikä tahansa suorituskelpoinen yksikkö voi olla komponentti. DCOM (*DistributedCOM*) on vastaavasti COMin laajennus, joka mahdollistaa sijaintiläpinäkyvyyden toteuttamisen asiakkaiden ja palvelinten välillä. (Szyperski 1998, 194-207.) COM/DCOM-ratkaisujen suurimpana heikkoutena on niiden vahva sidos Microsoft Windows-ympäristöön, vaikkakin jotkin valmistajat ovat tuottaneet toteutuksia myös muille alustoille (Orfali, Harkey & Edwards 1996b, 537).

JavaBeans on Sun Microsystemsin kehittämä Java-ohjelmointikieleen sidottu komponenttimalli, joka on esitelty ensimmäistä kertaa Java Development Kitin (*JDK*) versiossa 1.1. JavaBeans-komponentteja käytetään pääasiassa käyttöliittymäkomponentteina, ja niiden mukauttaminen on mahdollista komponentin

sisältämien ominaisuuksien (*property*) arvoja muuttamalla. (Szyperski 1998, 229-232). Vahvaa sidosta käyttöliittymiin tukee myös Sun Microsystemsin esittämä määritelmä JavaBeans-komponentille. Sen mukaisesti JavaBeans-komponentti on sellainen uudelleenkäytettävä rakenne, jota voidaan muokata visuaalisesti jossain sovelluskehittimessä (ks. esim. Horstmann & Cornell 1998). Esimerkiksi useat Javaa tukevat ohjelmointityökalut sisältävät mahdollisuuden käyttää JavaBeans-komponentteja käyttöliittymän rakentamisessa. JavaBeans-komponenttien käyttö ei kuitenkaan ole rajattu ainoastaan käyttöliittymään, vaan niitä voidaan käyttää ohjelmoinnissa kuten mitä tahansa Javan luokkaa. JavaBeans myös tukee hajauttamista sarjallistamisen ja etäkutsujen (*RMI, Remote Method Invocation*) avulla (Szyperski 1998, 241-242). Sarjallistaminen ja etäkutsut ovat kuitenkin myös muiden Java-olioiden käytössä, joten näiden hajauttamista tukevien ominaisuuksien ei voida katsoa kuuluvan itse JavaBeans-komponenttimalliin.

Taulukossa 2 on esitetty joitakin edellä esitettyjen komponenttimallien ominaisuuksia. Eri komponenttimalleja vertailtaessa COM/DCOM ratkaisujen vahvimmiksi puoliksi jäävät riippumattomuus ohjelmointikielestä ja vahvojen kaupallisten toteutusten olemassaolo. Toisaalta taas JavaBeans –komponenttimallin mukaiset ratkaisut ovat siirrettäviä ja täysin suoritusalustasta riippumattomia, mutta samalla sidottuja Java-ohjelmointikieleen. CORBAn mukainen ratkaisu vastaavasti mahdollistaa useiden eri ohjelmointikielien käyttämisen ja sallii samanaikaisesti lähes minkä tahansa ohjelmointikielellisen rakenteen toimia komponenttina järjestelmässä.

Käytännössä esimerkiksi CORBAn avulla toteutettavassa järjestelmässä asiakkaana toimiva käyttöliittymä voi olla toteutettu JavaBeans-komponenttimallin mukaisesti ja varsinaiset sovelluslogiikkaa sisältävät tasot esimerkiksi C++:lla tai C:llä.

TAULUKKO 2 Eri komponenttimallien ominaisuuksia

Ominaisuus	**CORBA**	**COM / DCOM**	**JavaBeans**
Riippumattomuus ohjelmointikielestä	Kyllä	Kyllä	Ei
Riippumattomuus suoritusalustasta	Ei*1	Ei	Kyllä
Komponentin määritelmä	Ei	Ei	Kyllä
Riippumaton komponenttimallin toteutus	Kyllä	Ei*2	Kyllä
Vahvoja kaupallisia toteutuksia	Kyllä	Kyllä	Kyllä

*1 Periaatteessa myös CORBAn voidaan katsoa olevan suoritusalustasta riippumaton, sillä toteutuksia on olemassa useille alustoille.

*2 Muillekin alustoille on joitain toteutuksia olemassa.

Mikään edellä kuvatuista komponenttimalleista ei kuitenkaan ota yksiselitteisesti kantaa esimerkiksi liiketoimintakomponenttien käyttämiseen ohjelmiston osana. CORBAlla toteutettavassa järjestelmässä komponentit ovat olioita ja niiden toteuttama liiketoimintalogiikka on mahdollista toteuttaa kulloinkin käytettävästä ohjelmointikielestä riippuvien sääntöjen ja mahdollisuuksien mukaisesti. JavaBeans-komponenttimalli on puolestaan suunnattu lähes yksinomaan käyttöliittymäkomponenttien tuottamiseen, eikä liiketoimintalogiikan toteuttaminen käyttöliittymässä ole mielekästä. DCOM on komponenttimallina lähinnä sellaista rakennetta, joka tukee liiketoimintakomponenttien käyttöä. DCOM on kuitenkin täysin alustariippuvainen ja sen komponentin määritelmä on puutteellinen.

3.3 Komponentit ohjelmistonkehityksessä

Komponenttipohjaiselle ohjelmistotuotannolle (*CBSE – Component Based Software Engineering*) löytyy kirjallisuudessa muitakin nimityksiä. Jotkut puhuvat komponenttipohjaisista ohjelmistoista (*CBS - Component Based Software*) tai komponenttipohjaisesta kehityksestä (*CBD - Component Based Development*). Lisäksi käytetään englanninkielistä termiä *Componentware*, jolla myös tarkoitetaan samaa asiaa.

Komponenttipohjaisella ohjelmistotuotannolla tarkoitetaan yleisesti ohjelmistojen kehittämistä käyttäen komponentteja ohjelmiston olennaisina rakennusosasina (Bergner ym. 1999). D'Souza ja Wills (1998) esittävät lisäksi huomattavasti laajemman määritelmän. Sen mukaisesti komponenttipohjainen ohjelmistotuotanto on lähestymistapa, jossa kaikki ohjelmistonkehitykseen liittyvät tekijät suoritettavasta ohjelmasta liiketoimintamalleihin asti täytyy voida luoda kokoamalla ja soveltamalla olemassaolevia komponentteja. (D'Souza & Wills 1998, 385.) Komponenttien olemassaololla on näin ollen ainoastaan yksi tarkoitus: niiden avulla ja niitä yhdistelemällä on voitava luoda kokonaisia toimivia järjestelmiä, jotka saavuttavat niille asetetut päämäärät.

Tämän tutkielman kannalta ei ole mielekästä puhua ohjelmistotuotannosta, sillä sen katsotaan yleisesti kattavan kaikki ohjelmiston tuotantoprosessiin liittyvät osa-alueet, kuten esimerkiksi laatujärjestelmän, projektinhallinnan ja dokumentoinnin (ks. esim. Haikala & Märijärvi 1998). Tutkielmassa käytetäänkin termiä komponenttipohjainen ohjelmistonkehitys kuvaamaan sitä prosessia ja niitä tehtäviä, jotka liittyvät läheisesti komponenttien suunnitteluun ja tuottamiseen, erotuksena ohjelmistotuotannosta, joka kattaa huomattavasti laajemman osan koko ohjelmistonkehitysprosessia.

Seuraavissa kohdissa on esitetty huomioita komponenteista uudelleenkäytettävinä ohjelmiston osina, komponenttien vaatimasta tuesta ohjelmistonkehityksessä, komponenttipohjaisen ohjelmistotuotannon kehityksestä ja käyttöönotosta sekä komponenttimarkkinoista.

3.3.1 Komponentit uudelleenkäytettävinä ohjelmiston osina

Komponenttipohjaisen ohjelmistonkehityksen ympärillä leijuu paljon erilaisia termejä ja markkinointi on myös melko aggressiivista. Komponenttipohjaisuuden luvataan tuovan runsaasti hyötyjä ohjelmistonkehitysprojekteihin ja erityisesti uudelleenkäytön nimeen vannotaan useissa yhteyksissä.

Uudelleenkäyttö on todennäköisesti komponenttien useimmiten esitetty ja niiltä myös vaadittu ominaisuus. Uudelleenkäyttöön liittyy kuitenkin samalla myös lukuisia ongelmia, joiden olemassaolo ohjelmistoja kehitettäessä tulee huomioida. Erityisesti tiettyjen komponenttimallien tai komponenttien valmistajakohtaisten erikoisominaisuuksien käyttö saattaa vaikeuttaa tai jopa estää uudelleenkäyttöä. Lisäksi on huomattava, että uudelleenkäytön historialla on juuret huomattavasti komponenttiajattelua syvemmällä, joten uudelleenkäytön saavuttaminen komponenttien kohdalla ei välttämättä ole yhtään helpompaa kuin esimerkiksi olio-ohjelmoinnin tapauksessa. Esimerkiksi Taivalsaari (1993) esittää kattavan kuvauksen uudelleenkäytön ongelmista olio-ohjelmoinnissa, ja useiden hänen esittämiensä huomioiden soveltaminen komponenttipohjaiseen ohjelmistonkehitykseen on varmasti hyvinkin mahdollista (ks. Taivalsaari 1993, 128-136). Myös Szyperski (1998) esittää joitain huomioita komponenttien uudelleenkäytöstä. Jotta komponentti olisi elinkykyinen, sillä tulee olla riittävän monta käyttötapaa ja erilaista käyttömahdollisuutta. Komponentilla tulee olla jokin ominaisuus, jonka vuoksi sen käyttäminen on mielekästä. Pelkkä tekninen edistyksellisyys ei Szyperskin mukaan välttämättä riitä, mutta esimerkiksi ensimmäinen ratkaisu tiettyyn ratkaisemattomaan ongelmaan, hyvä tuotetuki ja valmistajan maine voivat olla riittäviä perusteita olemassaolevan komponentin lukuisille hyödyntämiskerroille. (Szyperski 1998, 11.) D'Souza ja Wills (1998) huomauttavat, että komponentin vaihtaminen korvaavaan komponenttiin on mahdollista ainoastaan hyvin määriteltyjen rajapintojen ja komponentin sisäisen koskemattomuuden avulla. Koskemattomuudella ei kuitenkaan tarkoiteta joustamatonta toiminnallisuutta, sillä komponenttien toiminnan muokkaaminen voi olla mahdollista komponentin ominaisuuksien (*property*) muuttamisen avulla. (D'Souza & Wills 1998, 395.)

Komponenttipohjaisella ohjelmistonkehityksellä on siis edellä esitettyyn läheisesti liittyen vahva sidos komponenttien uudelleenkäyttöön. Esimerkiksi Szyperski (1998), joka pitää komponentteja aina binäärisinä yksiköinä, rajaa näin ollen lähdekoodin tutkimisen komponenttien uudelleenkäytössä pois. Tällainen niin sanottu kokoava uudelleenkäyttö (*blackbox*) nojaa vahvasti komponentista olemassa olevaan dokumentaatioon ja rajapintojen määrityksiin. Huomioitavaa on kuitenkin, ettei kokoava uudelleenkäyttö tarkoita pelkästään binääristen komponenttien

uudelleenkäyttöä vaan yksinkertaisesti sitä, että lähdekoodia ei tutkita, vaikka se olisi saatavillakin. Vastakohta kokoavalle uudelleenkäytölle on muuntava uudelleenkäyttö (*whitebox*), jossa ohjelmiston tai komponentin toiminnan tunteminen perustuu pitkälti sen lähdekoodin tutkimiseen. (Szyperski 1998, 33-34; Koskimies 1997, 155.) Komponenttien uudelleenkäyttöä ja sen kannattavuutta tutkittaessa pitää myös ottaa huomioon se, ettei läheskään aina ole mielekästä tehdä käytetystä komponentista uudelleenkäytettävää. Uudelleenkäyttö vaatii aina komponentin laajamittaista yleistämistä, mikä ei välttämättä ole mielekästä mikäli komponentin mahdolliset sovellusalueet eivät ole lähellä toisiaan (D'Souza & Wills 1998, 456).

3.3.2 Komponenttien vaatima tuki ohjelmistonkehityksessä

Komponenttimallien määritysten ja niiden toteutuksien määrä kasvaa koko ajan. Varsinainen päähuomio on siirtynyt itse komponenttien suunnittelusta ja kehittämisestä komponenttien suunnittelua ja tuottamista tukevien rakenteiden suunnitteluun ja kehittämiseen.

Komponenttipohjainen ohjelmistonkehitys tarvitsee kuitenkin tukea muun muassa menetelmistä, välineistä, toteutustekniikoista sekä komponenttien hallintaan ja ohjelmistojen kokoamiseen liittyvistä asioista. Menetelmillä tarkoitetaan ratkaisuja, joilla suunnitteluvaiheessa organisaatiota helpotetaan keskittymään olennaisiin järjestelmän osiin sekä niiden väliseen vuorovaikutukseen. Välineiden tulisi tukea komponenttien määrittelyä siten, että se voidaan tehdä toteutustekniikasta riippumatta. Toteutustekniikoiden pitää vastaavasti olla muun muassa tehokkuudeltaan ja turvallisuudeltaan riittäviä, jotta yrityksen asettamat tavoitteet voidaan saavuttaa. Komponenttien hallintaan ja ohjelmistojen kokoamiseen tarvitaan apua muun muassa siksi, että komponentteja ovat saattaneet toteuttaa useat eri ihmiset ja jopa eri teknologioihin pohjautuen. (Brown 1998, 1.)

Bergner, Rausch, Sihling ja Vilbig (1999) esittävät komponenttipohjaisessa ohjelmistonkehityksessä käytettävän menetelmän sisältävän ainakin hyvin määritellyn käsitteellisen viitekehyksen, komponenteille sopivat kuvaustekniikat, komponenttipohjaiselle ohjelmistotuotannolle räätälöidyn prosessimallin ja kuvaustekniikoita sekä prosessimal-

lia tukevat välineet. Hyvin määritellyn käsitteellisen viitekehyksen määritteleminen on välttämätöntä, jotta peruskäsitteet ja –määritelmät voidaan ilmaista selkeästi ja yksiselitteisesti. Kuvaustekniikoiden merkitys korostuu heidän mielestään erityisesti sovelluskehittäjien välisessä kommunikoinnissa. Prosessimallin sopivuus erityisesti komponenttipohjaiseen ohjelmistotuotantoon on myös välttämätöntä, sillä muun muassa Bergner ym. (1999), Vayda (1999) ja Szyperski (1998, 335) esittävät komponenttipohjaisuuden tuovan mukanaan runsaasti uusia tehtäviä ja rooleja ohjelmistonkehitykseen. Tämän lisäksi välineiden tulisi tarjota tuki toteuttamisen lisäksi dokumentaation tuottamiselle ja mahdollisesti jopa kriittisten järjestelmän osien varmentamiselle.

Vayda (1999) kiinnittää huomiota käytettäviin välineisiin ja menetelmään. Välineiden on oltava hänen mukaansa sellaisia, että niissä on tuki muun muassa mallintamiselle ja ohjelmakoodin generoimiselle. Esimerkkeinä näistä CASE-välineistä (*Computer Aided Software Engineering*) hän mainitsee Rational Softwaren ROSE:n, Select Softwaren SELECT:in ja Platiniumin Enterprise Modeling Suiten. Menetelmästä esimerkkinä Vayda mainitsee Catalysis-menetelmän (ks. esim. D´Souza & Wills 1998), joka tukee UML-notaatiota (*Unified Modeling Language*) laajennettuna komponenttipohjaista mallintamista koskevalla käsitteistöllä. Ongelma menetelmissä on useimmiten se, etteivät ne mahdollista komponenttien välisten riippuvuuksien mallintamista riittävän tarkasti vaan mahdollistavat pikemminkin komponenttien sisäisten tapahtumien määrittelemisen ja kuvaamisen (Szyperski 1998, 306). Vaydan (1999) mukaan Catalysis-menetelmässä on kuitenkin aikaisempaa parempia keinoja esimerkiksi komponenttien vaatimien ja tarjoamien palvelujen kuvaamiseen sekä esi- ja jälkiehtojen sekä invarianttien ja muiden rajoitteiden kuvaamiseen Object Constraint Languagen (*OCL*) avulla.

3.3.3 Komponenttipohjaisen ohjelmistonkehityksen käyttöönotto ja kehitys

Komponenttipohjaista ohjelmistonkehitystä voidaan toteuttaa useilla eri tavoilla, mutta Brown ja Wallnau (1998) esittävät jaottelun, jonka mukaisesti voidaan puhua joko abstraktista komponenttipohjaisesta ohjelmistonkehityksestä tai valmiiden

komponenttien avulla tapahtuvasta ohjelmistonkehityksestä. Ensiksi mainitulla tavalla aikaisemmin tuotettuja komponentteja ei huomioida vaan komponenttien tuottaminen aloitetaan tyhjästä. Tällöin vaatimukset käytettävää menetelmää ja prosessimallia kohtaan kasvavat. Vastaavasti valmiiden komponenttien avulla tehtävä ohjelmistonkehitys siirtää pääpainon komponenttien kehittämisestä niiden käyttöönottoon. Tällöin korostuvat muun muassa lähdekoodin saatavuuden, dokumentoinnin oikeellisuuden ja kattavuuden sekä komponentin toimivuuden merkitys. (Brown ja Wallnau 1998, 45.)

Brown ja Wallnau (1998) esittävät myös joitain syitä, jotka ovat johtaneet komponenttipohjaisen ohjelmistotuotannon kehitykseen ja käyttöönottoon. Yhä kehittyneemmät komponentteja tukevat teknologiat ovat heidän mukaansa suurelta osalta edistämässä komponenttipohjaisen ohjelmistonkehityksen suosiota. Kun teknologiat kehittyvät, niin väistämättä myös osaaminen, jossa näitä teknologioita hyödynnetään, kasvaa. Komponentteja tukevilla teknologioilla Brown ja Wallnau (1998) viittaavat nimenomaan eri komponenttimallien toteutuksiin, joiden tarjoamien palvelujen avulla komponenteista voidaan koota toimivia sovelluksia.

Brownin ja Wallnaun (1998) mukaan arkkitehtuurien kehittyminen ja siirtyminen keskustietokoneympäristöistä hajautettuun asiakas-palvelin –tyyppiseen työskentelyyn ovat muuttaneet organisaatioiden ohjelmistonkehitysprojektien luonnetta. Muutamien isojen projektien sijaan useita pienempiä projekteja on käynnissä samanaikaisesti, ja niiden sekä aikaisempien projektien tuotokset on järkevää jakaa koko organisaation käyttöön. Myös organisaatioiden suuret panostukset järjestelmiin aikaisempien vuosien aikana ovat vähentäneet kokonaan uusien järjestelmien kehittämiseen suunnattavaa rahamäärää. Niinpä uusien järjestelmien luominen tulee yhä enemmän olemaan aikaisemmin tehtyjen järjestelmän osien ja uusien vasta kehitettyjen osien yhdistämistä uudeksi järjestelmäksi.

Lisäksi Brown ja Wallnau (1998) esittävät organisaatioiden yhä enenevässä määrin tajunneen ohjelmistojen strategisen vaikutuksen liiketoimintaansa. Jos esimerkiksi koko organisaation kaikki tietojärjestelmät on toteutettu itsenäisesti, saattaa riskinä olla

muuntautumiskyvyn menettäminen sekä asiakkaiden ja markkinoiden vaatimuksiin vastaamisen vaikeutuminen. Vastaavasti, jos organisaatio on ostanut kaikki järjestelmät tietyltä toimittajalta, on vapaiden markkinoiden tarjoama valinnan vapaus menetetty ja organisaatiota koskevat päätökset tietojärjestelmien osalta tekee jokin oraganisaation ulkopuolinen osapuoli. Myös organisaatioiden toimintaympäristö on muuttunut huomattavasti, sillä muutoksia tapahtuu koko ajan ja organisaatioiden täytyy olla valmiita mukautumaan uuteen markkinatilanteeseen nopeasti.

Szyperski (1998) esittää näkemyksen, jonka mukaisesti epätäydellinenkin teknologia saattaa selviytyä, mikäli markkinat ovat olemassa. Vastaavasti täydellinenkään teknologia ei tule selviämään ilman markkinoita. (Szyperski 1998, 15.) Szyperskin mukaan komponentit tulevat väistämättä saavuttamaan tarpeeksi suuret markkinat, sillä komponenttimarkkinoilla kehitystyötä tekevät useat toistensa kanssa kilpailevat komponenttitoimittajat. Niinpä komponentteja hyväksikäyttävä ohjelmistonkehittäjä voi hyödyntää tämän useiden eri toimittajien tuottaman tuottavuuden ja innovaation kasvun omissa järjestelmissään. (Szyperski 1998, 7.) Tämän perusteella Szyperski esittää komponenttien käyttämisen olevan tulevaisuudessa väistämätöntä.

D'Souza ja Wills (1998) kuvaavat komponenttien mukanaan tuomien etujen olevan muun muassa rajapintakeskeinen suunnittelu, joka mahdollista skaalautuvien ja laajennettavien järjestelmien kehityksen, sekä standardien hyödyntäminen, joka luo pohjan valmiiden palvelujen käyttämiselle. Lisäksi komponentti muodostaa ylläpidettävän yksikön. Sen sijaan, että kokonaista järjestelmää tulisi ylläpitää, voidaan keskittyä yksittäisten komponenttien korvaamiseen ja ylläpitoon. Komponenttien käyttö tuo lisäksi mahdollisuuden kehittää ohjelmistoja yhdenaikaisesti, sillä pienempien kokonaisuuksien tekeminen on mahdollista, mikäli komponenttien suunnittelun alkuvaiheessa keskitytään rajapintojen määrittelyyn ja järkevän kokoisten ohjelmanosien erittelyyn komponenteiksi. (D'Souza ja Wills 1998, 397-398.)

Vayda (1999) esittää joitain huomioita organisaatioiden komponenttipohjaisen ohjelmistotuotannon omaksumisesta. Esimerkiksi uudelleenkäytön oletetaan olevan suora seuraus komponenttien käyttöönotosta, vaikka tärkeämpää olisi keskittyä

pohtimaan aiotaanko tuotettavia komponentteja todella käyttää uudelleen. Mikäli uudelleenkäyttöä aiotaan todella tukea, on sen edistämiseksi tehtävä huomattavia ponnistuksia, muun muassa komponenttien varastoinnin ja dokumentoinnin osalta. Vayda (1999) ehdottaakin kokeiluprojektin käynnistämistä sellaisella kohdealueella, jolla on odotettavissa mahdollisimman hyvä tuotto pääomalle ja mahdollisimman pieni riski epäonnistua. Lisäksi organisaation johdon sitouttaminen tähän projektiin on ensiarvoisen tärkeää vaikkakaan suurta määrää uudelleenkäytettäviä komponentteja tai tuottavuuden kasvua ei voidakaan pitää saavutettavissa olevana tavoitteena.

Jaaksi ja Laitkorpi (1999) esittävät oman komponentit huomioivan laajennuksensa OMT++ -menetelmään (*Object Modeling Technigue*). Heidän mukaansa ainakin ensimmäisten komponenttien tuottaminen pitäisi tehdä osana normaalia ohjelmistonkehitysprosessia ja tämän jälkeen komponenttien kehitys ja ylläpito toteutettaisiin erillään, omien yksiköidensä johdolla (Jaaksi & Laitkorpi 1999, 43).

3.3.4 Komponenttimarkkinat

Kirjallisuudessa puhutaan usein komponenttimarkkinoiden synnystä. Esimerkiksi Orfali ym. (1996b) esittää Gartner Groupin raportin ennustaneen peräti kolmen uuden markkina-alueen esiinnousua. Nämä alueet ovat komponenttimarkkinat, komponenttien tuottamistyökalujen markkinat sekä komponenteista koottujen ohjelmistojen markkinat. Szyperski (1998) jakaa komponenttipohjaisen ohjelmistotuotannon luomat markkinat osittain edelliseen perustuen komponentti-, komponenttiväline- ja komponenttipalvelumarkkinoihin. Erityisesti hän korostaa palvelujen merkitystä markkinoinnissa ja muun muassa silloin, jos pelkkien komponenttien markkinointi on vaikeaa. Komponenttimarkkinoiden synnyssä Wallnau (1999) esittää suurimmaksi ongelmaksi kahden eri markkinatahon vastakkaisten tavoitteiden yhteensovittamisen.

Komponenteille usein esitetty tavoite on niiden helppo korvattavuus ja vaihtokelpoisuus. Vastaavasti kaupallisten tahojen tapauksessa tämä tarkoittaisi sitä, että jonkin valmistajan komponentin tai komponenttimallin toteutus voitaisiin korvata jonkin toisen valmistajan tuotteella. Tällöin markkinoita voitaisiin pitää täysin hyödykemarkkinoina,

joilla vapaa hintakilpailu ohjaisi kysyntää ja tarjontaa. Tätä ei voida kuitenkaan pitää tavoiteltavana kaupallisten tuottajien tapauksessa, sillä kaupallisten tuotteiden tuottaminen on yhtä lailla ohjelmistotuotantoa riskeineen ja kustannuksineen kuin komponenttien käyttäminen itse sovellusten kehittämiseen. Niinpä jokainen kaupallisten tuotteiden tuottaja haluaa sitoa asiakkaansa jollakin tavalla omiin tuotteisiinsa, mikä taas vastaavasti heikentää komponenttien korvattavuutta ja vaihtokelpoisuutta. (Wallnau 1999, 1.) Kuten kohdassa 3.1.3. jo todettiin, tämä näkemys poikkeaa vahvasti siitä, mitä muualla on esitetty komponenttimarkkinoiden synnystä. Erityisesti Szyperski (1998) pitää komponenttien käyttämistä ja komponenttimarkkinoiden syntymistä täysin väistämättömänä (Szyperski 1998, 6).

3.4 Yhteenveto

Kirjallisuuden perusteella komponentille on mahdotonta löytää yhtä ainoaa ja oikeaa määritelmää, sillä jotkin määritelmät korostavat eri asioita kuin toiset. Lisäksi määritelmien erilaisista painotuksista johtuen saattaa jokin määritelmä jättää ottamatta kantaa joihinkin asioihin kokonaan. Lisäksi komponenttien määritelmien abstraktioaste vaihtelee, jolloin komponenteiksi laskettavien ohjelmistonosien määrä kasvaa tai vähenee vastaavasti.

Komponenttimallien kehittyminen ja käyttöönotto aiheuttaa myös ongelmia yrityksissä, jotka suunnittelevat käyttävänsä komponentteja ohjelmistonkehityksessä. Valmistajakohtaiset erikoisominaisuudet ja sitoutuminen tietyn valmistajan tuotteisiin ovat asioita, jotka täytyy ottaa huomioon ostopäätöksiä tehtäessä. Lisäksi valinta eri komponenttimallien välillä ja valinnan aiheuttamat vaikutukset organisaation ohjelmistonkehitykseen tulee huomioida. Komponenttimallien suuri kehitysvauhti aiheuttaa myös sen, että eri versiot vanhentuvat määritelmien kehittyessä nopeasti.

Komponenttipohjaisen ohjelmistotuotannon kehityksestä ja kehityksen taustoista ollaan kirjallisuudessa jonkin verran erimielisiä. Esimerkiksi Szyperskin (1998) komponenttien binäärisyyttä korostava lähestymistapa on ollut kritiikin kohteena, sillä

sen voidaan katsoa hylkäävän aikaisempi komponentteja lähellä oleva kehitys, mitä esimerkiksi Parrish ym. (1998) voimakkaasti kritisoivat.

Myös komponenttimarkkinoiden synnystä ja kaupallisten, valmiiden ja vaihtokelpoisten komponenttien käytöstä ja käyttömahdollisuuksista ollaan erimielisiä. Osa on vahvasti sitä mieltä, että komponenttimarkkinat synnyttävät kokonaisen uuden tavan rakentaa uusia järjestelmiä valmiiden komponenttien avulla. Vastaavasti osalle komponenttimarkkinoiden synty on mahdotonta eri markkinatahojen ristiriitaisten tavoitteiden vuoksi.

Tässä luvussa esitettyjen komponentin ja komponenttimallin määritelmien sekä komponenttipohjaisesta ohjelmistonkehityksestä esitettyjen huomioiden perusteella, komponenttien käyttöä ohjelmistokehityksessä yhä enenevässä määrin voidaan pitää todennäköisenä. Komponenttien käytöstä mahdollisesti saatavat edut eivät kuitenkaan synny itsestään, sillä esimerkiksi uudelleenkäyttö vaatii lisäresursseja ohjelmistonkehitysprojekteihin.

Mikään tässä luvussa esitetyistä komponenttimalleista ei myöskään tue erityisen vahvasti liiketoimintakomponenttien käyttöä. Komponenttimallien olisi myös tuettava paremmin valmiiden komponenttimarkkinoilla tarjolla olevien komponenttien käyttöönottoa.

4 JAVA 2 ENTERPRISE EDITION

Luvuissa kaksi ja kolme esitetyt asiat luovat pohjan tämän luvun keskeisimmälle annille. Luvussa kaksi esiteltiin monitasoarkkitehtuurien peruskäsitteistöä sekä verrattiin lyhyesti kaksi- ja kolmitasomallia toisiinsa. Luvussa kolme vastaavasti käsiteltiin yleisiä ohjelmistokomponentteihin ja komponenttipohjaiseen ohjelmistonkehitykseen liittyviä asioita. Tämän luvun tarkoituksena on esitellä eräs uusimmista ratkaisuehdotuksista, jonka tarkoituksena on pyrkiä ratkaisemaan monitasoarkkitehtuureihin pohjautuvien, vahvasti liiketoimintaa tukevien järjestelmien ongelmia komponenttiajattelua hyväksikäyttäen.

Luvussa esitellään Sun Microsystemsin Java 2 Enterprise Edition –ohjelmointialusta (*J2EE*) ja siihen kuuluvat olennaisimmat määritykset. Tässä luvussa myös esitellään tarkemmin joitain J2EE:hen kuuluvien teknologioiden määrityksiä, joiden hyödyntäminen monitasoisia liiketoimintajärjestelmiä toteutettaessa on todennäköistä. Suurimman ja tämän tutkielman kannalta kiinnostavimman osan J2EE:stä muodostaa Enterprise JavaBeans (*EJB*)-komponenttimallin määritys, joka on esitetty omana kokonaisuutenaan luvussa viisi.

4.1 Java ohjelmointikielenä

Java muistuttaa syntaksinsa puolesta hyvin paljon C++:aa, ja esimerkiksi ohjaus- ja luokkarakenteet on lainattu lähes sellaisinaan (ks. Stroustrup 1998). Osa C++:n virhealttiista osista on kuitenkin pyritty poistamaan ja näin luomaan turvallisempi ja helpommin omaksuttava ohjelmointikieli (Koskimies 1997, 232.)

Javan suoritusaikainen toiminta toteutetaan niin sanotun virtuaalikoneen (*virtual machine*) avulla, joka tulkitsee suorituksen aikana lähdekoodista tavukoodiksi (*byte code*) käännettyjä tiedostoja. Tarvittaessa tehokkuuden lisäämiseksi lähdekoodi voidaan

myös kääntää kulloisenkin laiteympäristön konekoodiksi (Koskimies 1997, 233). Tällöin kuitenkin Javan perusperiaatteisiin kuuluva siirrettävyys menetetään.

Siirrettävyyden saavuttamiseksi kaikkien kerran lähdekoodista tavukoodiksi käännettyjen sovellusten suorittamisen tulee olla mahdollista kaikissa ympäristöissä, joissa on käytössä samaan Javan määrityksen versioon perustuva virtuaalikone (Koskimies 1997, 233). Tämän tutkielman puitteissa ei oteta tarkemmin kantaa Javan suoritusaikaiseen toimintaan, lähdekoodin kääntämiseen tavukoodiksi tai virtuaalikoneiden yhteensopivuusongelmiin.

Tämän tutkielman tarkoituksena ei ole selvittää Java-ohjelmointikieleen liittyviä piirteitä kuin niiltä osin, joilta se on välttämätöntä tutkielmassa esitettyjen asioiden ymmärtämisen takia. Myöskään mitään ohjelmointikielen perusrakenteita ei tulla selvittämään yksityiskohtaisesti. Luvuissa viisi ja kuusi esitetään myös Java-ohjelmointikielen syntaksin mukaisia ohjelmaesimerkkejä, jotka noudattelevat pääpiirteissään Sun Microsystemsin nimeämis- ja ohjelmoimiskäytäntöjä. Esimerkkejä on kuitenkin tilanpuutteen vuoksi joissain kohdin voimakkaasti yksinkertaistettu, eikä niiden hyödyntäminen suoraan ole välttämättä mahdollista. Esimerkkien tarkoituksena on selventää esitettävää asiaa, ei osoittaa yksikäsitteisesti toimivaa toteutusta. Tarkemmin Javan toiminnasta ja ohjelmointikielen perusrakenteista on kerrottu muun muassa teoksessa Arnold ja Gosling (1997).

4.2 Javan ohjelmointialustat

Ohjelmointialustalla (*platform*) tarkoitetaan tässä tutkielmassa sellaista ohjelmiston kehitys- ja suoritusympäristön määritelmää, jonka määrittelemien palvelujen avulla sovelluksia voidaan kehittää ja suorittaa. Sun Microsystems on vuoden 1999 lopussa julkistamansa uuden Java 2 -määrityksen myötä ottanut käyttöön uuden kolmitasoisen jaon eri ohjelmointialustoilleen. Tunnetuin ja ehkäpä käytetyin näistä on Java 2 Standard Edition (*J2SE*), joka sisältää kaikki tavallisten Java-ohjelmien ja –sovelmien toteuttamiseen tarvittavat sovellusliittymät. Java 2 Micro Edition (*J2ME*) on vastaavasti erilaisiin tietimiin ja kynämikroihin sekä esimerkiksi kelloihin tarkoitettu Java-ohjelmointialustan

määritelmä, joka sisältää vain erittäin rajoitetun osajoukon kaikista Javan ominaisuuksista (Roman 1999, 28). Tämän tutkielman kannalta merkityksellisin näistä ohjelmointialustoista on Java 2 Enterprise Edition (*J2EE*), joka sisältää kaikki hajautettujen liiketoimintaa tukevien järjestelmien toteuttamiseen tarvittavat sovellusliittymät. J2EE:n tarkoituksena on nimensä mukaisesti tarkoitus pyrkiä vastaamaan yritystason järjestelmien asettamiin vaatimuksiin. Sun Microsystems tarjoaa veloituksetta käyttöön kaikkien edellä lueteltujen ohjelmointialustojen määritysten mukaiset toteutukset Windows- ja Solaris-käyttöjärjestelmäympäristöihin.

J2EE:n voidaan katsoa koostuvan neljästä itsenäisestä osasta: sovellusohjelmointimallista (*application programming model*), yhteensopivuuden todentamistesteistä (*compatibility test suite*), esimerkkitoteutuksesta (*reference implementation*) ja itse J2EE sovellusliittymistä (*Application Programming Interface, API*). Sovellusohjelmointimallin tarkoituksena on tarjota neuvoja ja ohjeita yritystason järjestelmien toteuttamiseen. Sovellusohjelmointimalli ei ole täydellinen kuvaus todellisesta maailmasta vaan pikemmin abstrakti kuvaus J2EE:n eri palvelujen ja teknologioiden käytöstä. Lisäksi keskeinen tehtävä sovellusohjelmointimallilla on jakaa ohjelmistonkehityksessä suoritettavat toteutusta koskevat tehtävät sovellusohjelmoijan tekemiin tehtäviin ja J2EE:n valmiina tarjoamiin palveluihin. Yhteensopivuuden todentamistesteillä tarkoitetaan joukkoa erilaisia testejä, joilla voidaan varmistua siitä, että jokin tietty määritelmän toteutus on varmasti J2EE-määrityksien mukainen. Vastaavasti esimerkkitoteutus tarjoaa mallin yhdestä mahdollisesta järjestelmätoteutuksesta, jonka avulla voidaan todentaa J2EE-alustalla toimivien sovellusten toimivuus ja ominaisuudet. (Shannon 1999, 1-1 - 1-2; Sun 1999b, 1-3.) Tämä esimerkkitoteutus on kuvitteellinen WWW-ympäristössä toimiva elektroninen kauppa, joka esittelee kattavasti J2EE:n mahdollisuuksia ja eri teknologioiden käyttöä. Esimerkkisovellus on Sun Microsystemsin periaatteiden mukaisesti saatavissa lähdekoodeineen veloituksetta.

Sun Microsystemsin tuottaman J2EE:n määritelmien mukaisen ohjelmointialustan toteutuksen tarkoituksena on tarjota esimerkki esitetyn määrityksen mukaisena ohjelmointialustan toteutuksena. Tämän lisäksi tätä ohjelmointialustan toteutusta käyte-

tään yhteensopivuuden todentamistestivälineiden suoritusalustana ja sitä voidaan vapaasti käyttää sovellusten kehittämiseen. (Sun 2000, 24.) Lisäksi ohjelmointialustan toteutusta voidaan luonnollisesti käyttää esimerkkisovelluksen suorittamiseen, sillä ohjelmointialusta sisältää kaikki ne peruspalvelut, jotka J2EE:n mukaisen ympäristön tulee sisältää.

4.3 Sovellusliittymät

J2EE sisältää useita sovellusliittymiä, joita käyttäen sovelluksia voidaan kehittää ja joiden tarjoamia palveluja sovellukset voivat käyttää. Jo J2SE sisältää kaksi yritystason järjestelmien sovellusliittymää, JavaIDL:n ja JDBC (*Java DataBase Connectivity*) Core APIn. JavaIDL on Sun Microsystemsin Javaan pohjautuva CORBAn toteutus, jonka avulla Java-ohjelmien on mahdollista kommunikoida myös muilla ohjelmointikielillä tehtyjen CORBA-sovellusten kanssa. JDBC Core API on vastaavasti asiakassovelluksen rajapinta kommunikointiin erilaisten tietokantojen kanssa.

J2EE:n sisältämät eri sovellusliittymät lyhyine kuvauksineen on esitetty taulukossa 3. Seuraavissa alakohdissa on lisäksi kuvattu tarkemmin yritystason järjestelmien kannalta neljä oleellisinta sovellusliittymää. Muista sovellusliittymistä esitetään yksityiskohtaisempaa tietoa ainoastaan, mikäli se on välttämätöntä. Lisätietoa eri sovellusliittymistä on saatavilla runsaasti esimerkiksi Sun Microsystemsin eri teknologioita koskevasta dokumentaatiosta (ks. esim. Shannon 1999).

TAULUKKO 3 J2EE:n sisältämät sovellusliittymät (mukailtu lähteestä Shannon 1999, 6-12 – 6-19)

Laajennus	**Kuvaus**
JDBC 2.0	*Java DataBase Connectivity* tarjoaa yhdenmukaisen sovellusliittymän relaatiotietokantojen käsittelyyn.
RMI-IIOP 1.0	*Remote Method Invocation* on yksinkertainen hajautettujen sovellusten kommunikointiprotokolla. IIOP (*Internet Inter-ORB Protocol*) mahdollistaa kommunikoinnin CORBA-sovellusten kanssa.
EJB 1.1	*Enterprise JavaBeans* on komponenttimallin määritys palvelintasolla toimiville liiketoimintakomponenteille.
Servlets 2.2	Palvelinsovelmat eli *servletit* ovat Java-sovelmien vastinpari palvelimella ja ne tarjoavat CGI-ohjelmien kaltaisia palveluja asiakassovellukselle.
JSP 1.1	*Java Server Pages* on tapa upottaa Java koodia HTML-tiedostoon. Suoritettaessa JSP-sivu käännetään palvelinsovelmaksi.
JMS 1.0	*Java Messaging Service* mahdollistaa asynkronisen tiedonvälityksen Java-olioiden välillä.
JNDI 1.2	*Java Naming and Directory Interface* mahdollistaa yksittäisten komponenttien ja muiden resurssien paikallistamisen.
JTA 1.0	*Java Transaction API* yhdessä Java Transaction Servicen (*JTS*) kanssa mahdollistaa luotettavan tapahtumapohjaisen tehtävienhallinnan.
JavaMail 1.1	*JavaMail* mahdollistaa sähköpostiviestien lähettämisen suoritusalustasta riippumattomalla tavalla.
JAF 1.0	JavaMail käyttää *JavaBeans Activation Frameworkia* MIME-sähköpostiviestien tietojen käsittelyssä.

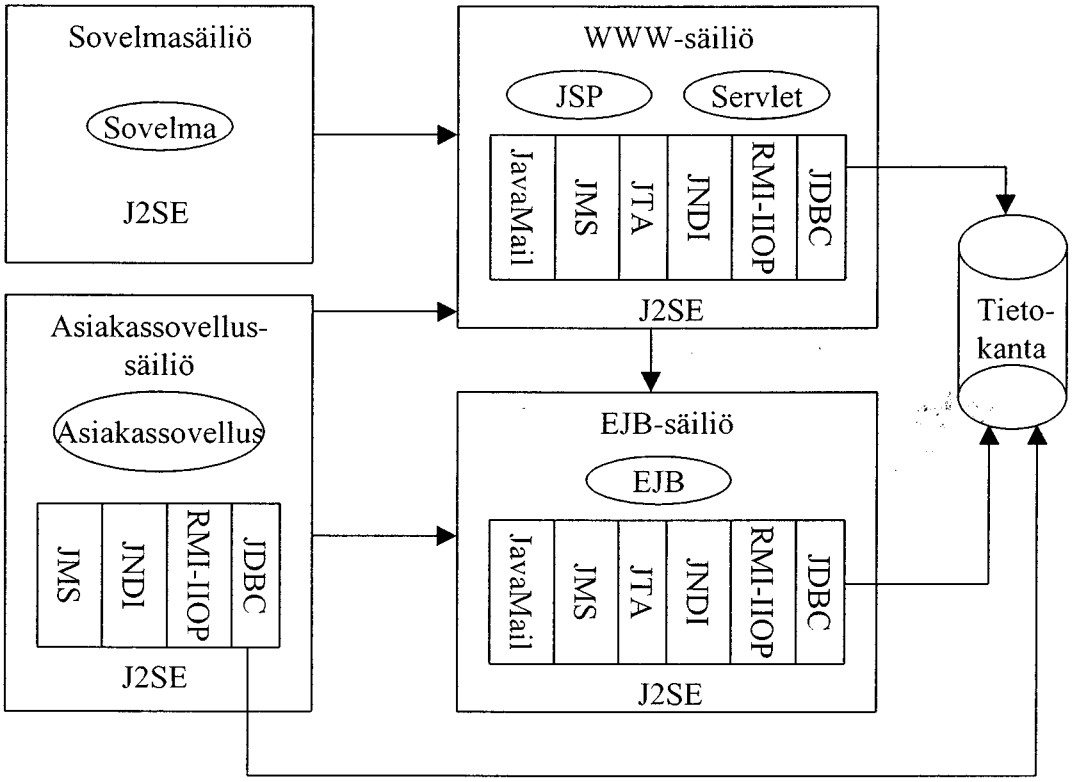
J2EE:n keskeinen idea on tarjota suoritusympäristön määritys erityyppisille ohjelmille. Tätä suoritusympäristöä nimitetään suoritettavasta ohjelmasta riippumatta aina säiliöksi (*container*). J2EE määrityksen mukaan on olemassa neljä erilaista säiliötä, jotka kaikki palvelevat eri tarkoituksia ja niiden on myös tarjottava tarkoituksiensa mukaisia palveluja. Nämä säiliöt ovat asiakassovellus-, sovelma-, WWW- ja EJB-säiliöt.

Asiakassovellus- ja sovelmasäiliöt toimivat asiakastasolla, kun taas vastaavasti WWW- ja EJB-säiliöt ovat palvelintason säiliöitä. Taulukossa 4 on esitetty mitä sovellusliittymiä näiden eri säiliöiden on tarjottava.

TAULUKKO 4 Säiliöiden tarjoamat sovellusliittymät (Shannon 1999, 6-2)

	Asiakastaso		**Palvelintaso**	
Laajennus	**Asiakassovellus**	**Sovelma**	**WWW**	**EJB**
JDBC 2.0	Kyllä	Ei	Kyllä	Kyllä
RMI-IIOP 1.0	Kyllä	Ei	Kyllä	Kyllä
EJB 1.1	Kyllä	Ei	Kyllä	Kyllä
Servlets 2.2	Ei	Ei	Kyllä	Ei
JSP 1.1	Ei	Ei	Kyllä	Ei
JMS 1.0	Kyllä	Ei	Kyllä	Kyllä
JNDI 1.2	Kyllä	Ei	Kyllä	Kyllä
JTA 1.0	Ei	Ei	Kyllä	Kyllä
JavaMail 1.1	Ei	Ei	Kyllä	Kyllä
JAF 1.0	Ei	Ei	Kyllä	Kyllä

Merkittävin ero kahden palvelintasolla toimivan säiliön, WWW- ja EJB-säiliön välillä on niiden tarjoama tuki servleteille ja JSP:lle. EJB-säiliön ei tarvitse näitä toteuttaa, mutta mikäli kommunikointi asiakassovelluksen ja sovelluspalvelimen välillä halutaan toteuttaa esimerkiksi servletien avulla, on sovelluspalvelimen toteutettava WWW-säiliön toiminnallisuus. Kuviossa 3 on esitetty eri säiliöiden loogiset sijainnit ja niiden väliset suhteet sekä eri sovellusliittymät kullakin tasolla. Kuviossa 3 esitetty jako ei kuvaa säiliöiden fyysistä jakoa eri palvelimille, prosesseihin, osoiteavaruuksiin tai virtuaalikoneisiin (Shannon 1999, 2-3).



KUVIO 3 Säiliöiden sijainnit, suhteet ja sovellusliittymät (mukailtu lähteestä Shannon 1999, 2-3)

Kuviosta 3 voidaan todeta erot eri säiliöiden tarjoamien sovellusliittymien määrässä. Asiakastasolla yksinkertaisin säiliö on sovelmasäiliö, jonka ei tarvitse toteuttaa mitään sovellusliittymiä. Vastaavasti asiakassovellussäiliön tulee toteuttaa sovellusliittymiä, joiden avulla toteutetaan muun muassa nimipalvelun käyttö ja kommunikointi tietovarastojen kanssa. Asiakassovellussäiliö kykeneekin huomattavasti monipuolisempaan kommunikointiin muiden säiliöiden kanssa kuin sovelmasäiliö. Vastaavasti palvelintasolla varsinaisen liiketoimintalogiikan toteuttaa EJB-säiliö. Asiakassovellussäiliö voi kommunikoida suoraan EJB-säiliön kanssa, mutta sovelmasäiliön on pakko suorittaa kommunikointi EJB-säiliön kanssa WWW-säiliön välityksellä, jonka servletit tai JSP-sivut voivat toteuttaa varsinaisen kommunikoinnin EJB-säiliön kanssa.

4.3.1 Java Naming and Directory Interface

Java Naming and Directory Interface (*JNDI*) on sovellusliittymä, joka tarjoaa yhdenmukaisen rajapinnan erilaisten nimi- ja hakemistopalvelujen käyttöön. JNDI:n toteuttaman

sovellusliittymän avulla on mahdollista paikallistaa tietoverkkoon kytkettyjä resursseja kuten esimerkiksi tulostimia ja muita tietokoneita. (Sun 1999a, 1.) EJB-komponenttimalli käyttää JNDI:tä laajasti komponenttien paikallistamiseen, ja niinpä JNDI on elintärkeä tukipalvelu koko EJB-komponenttimallille (Roman 1999, 561).

JNDI:n eduksi voidaan laskea sen riippumattomuus varsinaisen nimi- tai hakemistopalvelun fyysisestä toteutuksesta. Tästä johtuen kaikkien eri palvelujen käyttö tapahtuu Java-sovelluksessa saman sovellusliittymän välityksellä. Lisäksi JNDI on suunniteltu laajennettavaksi sovellusliittymäksi, joten tulevaisuudessa mahdollisesti JNDI:n välityksellä käytettäväksi liitettävät uudet nimi- tai hakemistopalvelut eivät aiheuta muutoksia ohjelmakooditasolla lainkaan. (Roman 1999, 561-565.)

JNDI koostuu kahdesta osasta, itse sovellusten käyttämästä JNDI-sovellusliittymästä sekä palveluntarjoajan sovellusliittymästä (*service provider interface, SPI*). Palveluntarjoajan sovellusliittymän toteutuksia ovat kaikki sellaiset toteutukset, jotka mahdollistavat pääsyn johonkin tiettyyn nimi- tai hakemistopalveluun. (Sun 1999a, 4.)

JNDI:lle on olemassa useita toteutuksia palveluntarjoajan sovellusliittymästä eri nimi- ja hakemistopalveluille (ks. esim. Roman 1999 tai Sun 1999a). Muun muassa Sun Microsystems toimittaa veloituksetta LDAP-hakemistopalvelun (*Lightweight Directory Access Protocol*) käyttöön soveltuvan palveluntarjoajan sovellusliittymän toteutuksen. (Roman 1999, 567.)

4.3.2 Java Database Connectivity

Java Database Connectivity (*JDBC*) on sovellusliittymä, jonka avulla Java-ohjelmat voivat kommunikoida erilaisten tietokantojen kanssa. JDBC:n perusperiaatteena on tarjota sovellusohjelmoijalle yhtenäinen ja tarkan määrityksen mukainen rajapinta, jota käyttämällä kaikki tarvittava toiminta tietokantojen kanssa voidaan suorittaa. Eri tietokantatuotteiden valmistajat toteuttavat tämän sovellusliittymän rajapinnan omissa ajureissaan tietokannan vaatimalla tapauskohtaisella tavalla.

Tietokantatuotteen valmistajan toteuttama ajuri suorittaa kaiken kommunikoinnin sovelluksien ja tietokannan resurssinhallitsimen (*resource manager*) välillä. Resurssinhallitsimella tarkoitetaan tässä yhteydessä ulkoisen resurssin eli tietokannan tai muun tallennuspaikan yhteydessä toimivaa hallitsinta, joka huolehtii kaikkien tietokannan tapahtumien suorittamisesta ja valvonnasta (ks. myös 4.3.4). Kaikki ajurit liittyvät J2EE-ympäristöön erillisen J2EE SPI –rajapinnan (*Service Provider Interface*) avulla, ja niiden toiminta on näin taattu kaikissa J2EE-määrityksen mukaisissa ympäristöissä. (Roman 1999, 268; Shannon 1999, 2-3.)

Keskeisimpänä JDBC:n etuna voidaan pitää sitä, että sovellusohjelmoijan ei tarvitse tietää millaisen tietokannan kanssa ollaan tekemisissä, vaan riittää, että hän tietää kuinka JDBC-rajapintaa käytetään.

JDBC määritys on jaettu kahteen eri osaan, joista toinen on JDBC Core Api 2.1, joka mahdollistaa tavanomaisten tietokantatehtävien suorittamisen. Sen rajapinnat on esitetty paketissa *java.sql*. Toinen määritys, JDBC Optional Package 2.0 sisältää laajennuksen perustoimintoihin ja mahdollistaa muun muassa hajautetut tapahtumat sekä yhteyspoolien käytön (*connection pool*). Optional Packagen rajapinnat on määritelty paketissa *javax.sql*. (White & Hapner 1999, 9.)

4.3.3 Java Remote Method Invocation

Java Remote Method Invocation (*RMI*) on Java-ohjelmointikieleen toteutettu tapa hajautettuun olioiden väliseen kommunikointiin etäkutsujen avulla. On syytä huomioida, että RMI ei ole erillinen J2EE-ohjelmointialustan määritys vaan RMI on toteutettuna jokaisessa Java-virtuaalikoneessa ja on näin ollen tiivis osa Java-ohjelmointikieltä.
RMI:n päätavoitteiksi on nimetty yksinkertaisuus ja sopivuus Java-ohjelmointikieleen. Näiden tavoitteiden saavuttamiseksi RMI muun muassa tukee hajautettuja olioita kuten mitä tahansa paikallisiakin Java-olioita, mahdollistaa kokonaisten olioiden välittämisen parametreina ja peittää hajautukseen tarvittavat toimenpiteet ohjelmoijalta. (Sun 1998, 2-4.)

RMI hyödyntää voimakkaasti rajapinta-ajattelua ja kaikkien RMI:n avulla käytettävien luokkien on esiteltävä metodinsa erillisessä rajapinnassa, joka perii *java.rmi.Remote*-luokan. Varsinainen toteutusluokka perii vastaavasti *java.rmi.server.UnicastRemoteObject*-luokan. Mikäli toteutusluokka kuitenkin jo perii jonkin toisen luokan, on Javan yksinperinnästä johtuen mahdotonta periä *java.rmi.server.UnicastRemoteObject*-luokkaa. Tällöin voidaan käyttää metodia *java.rmi.server.UnicastRemoteObject.exportObject()*-metodia. (Sun 1998, 29-31; Roman 1999, 509-523.) Kaikkien metodien tulee lisäksi aiheuttaa *java.rmi.RemoteException* tyyppinen poikkeus, mikäli tietoliikenteessä ilmenee ongelmia (Sun 1998, 22). Lisäksi jokainen metodi voi luonnollisesti aiheuttaa haluamiaan poikkeuksia edellä vaaditun poikkeuksen lisäksi.

Kun toteutusluokka ja sen rajapinta ovat valmiita täytyy niiden avulla vielä luoda tynkä- (*stub*) ja runkoluokat (*skeleton*), joiden avulla hajautus asiakassovelluksen ja palvelimen välillä toteutetaan (Roman 1999, 528). Tynkien ja runkojen avulla varmistetaan täydellinen sijaintiläpinäkymättömyys.

Varsinainen olioiden löytäminen toteutuu RMI-rekisterin (*RMI Registry*) avulla, jollainen jokaisella olioita sisältävällä palvelimella tulee olla. Tähän rekisteriin rekisteröidään kaikki oliot jollain tietyllä nimellä, jonka perusteella viite varsinaiseen olioon löydetään. (Sun 1998, 45.) Jotta vältyttäisiin olioiden sijaintien ja nimien kiinnittämiseltä jo sovelluksia toteutettaessa, on eräs järkevä vaihtoehto käyttää JNDI:tä (Roman 1999, 514).

RMI:n käyttö on mahdollista myös CORBAa hyväksikäyttäen. Tällöin liikennöinti asiakas- ja palvelinsovelluksen välillä toteutetaan IIOP-protokollan (*Internet Inter-ORB Protocol*) avulla. Kahden Javalla toteutetun sovelluksen välillä liikennöinti toteutetaan vastaavasti Java-ohjelmointikieleen sidotulla JRMP:llä (*Java Remote Method Protocol*). Tässä tutkielmassa ei kuitenkaan oteta kantaa RMI:n ja IIOP:n käyttöön tai siinä mahdollisesti esiintyviin ongelmiin (ks. esim. Roman 1999, 303-341).

4.3.4 Java Transaction API ja Java Transaction Service

Java Transaction API (*JTA*) ja Java Transaction Service (*JTS*) ovat J2EE:n tapa toteuttaa sovellusten tapahtumanhallinta. Tapahtumanhallinnalla tarkoitetaan menetelmää, jossa tapahtumien hallinta toteutetaan niin, että sovellusohjelmoijan ei tarvitse huolehtia esimerkiksi monimutkaisista tietokantojen tahdistuksista. Sovellusohjelmoija jakaa suorittamansa tehtävät tapahtumiksi. Jokainen tapahtuma suoritetaan aina kokonaan ja vahvistetaan (*commit*), tai sitten sitä ei suoriteta ollenkaan ja kaikki mahdollisesti tapahtuman sisällä jo tehdyt toiminnot peruutetaan (*rollback*) (Elmasri & Navathe 1994, 534.) Jokaisella tapahtumaan osallistuvalla Java-oliolla on siis mahdollisuus "äänestää" tapahtuman onnistumisen tai sen peruuttamisen puolesta. Tapahtumanhallinnan keskeisin vaatimus on se, että sovelluspalvelin osaa palauttaa peruutettua tapahtumaa edeltävän tilan palvelimelle ja tapahtumaan osallistuneisiin tietokantoihin (Morgenthal 1999, 36).

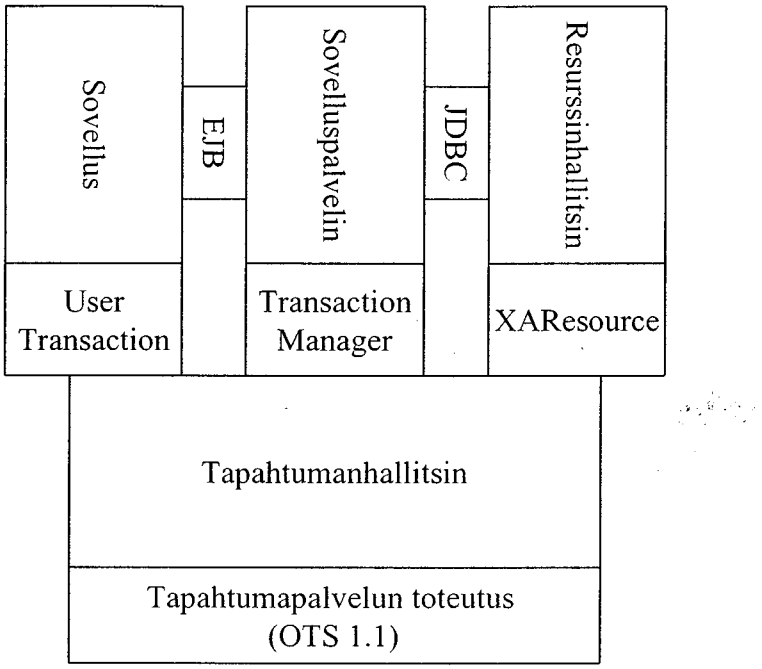
JTA on korkean tason sovellusliittymä, jonka avulla sovellusohjelmoijat ja säiliöiden toteuttajat voivat käyttää tapahtumanhallintapalveluja. JTS on vastaavasti määritys, jonka tarkoituksena on määritellä tapahtumanhallitsimen (*transaction manager*) toiminta, joka tukee korkeammalla tasolla JTA:n määrittelemiä metodeja ja lisäksi määrittelee alemmalla tasolla, kuinka yhteistoiminta OMG:n CORBA Object Transaction Servicen (*OTS*) kanssa toteutetaan (Cheung 1999, 6). JTS käyttää siis OMG:n CORBA OTS:ää varsinaiseen tapahtumanhallintaan ja määrittelee ainoastaan rajapinnat, jonka avulla sitä voidaan Javassa käyttää ja jonka avulla se kommunikoi CORBA OTS:n kanssa. Tapahtumanhallitsin voi olla yhteydessä myös muihin tapahtumanhallitsijoihin, jotka toteuttavat toimintansa käyttäen CORBA OTS:n rajapintoja (Cheung 1999, 9).

JTA:n voidaan katsoa koostuvan kolmesta osasta. Ensimmäinen näistä osista on korkean tason sovellusliittymä, jonka tarjoamien palvelujen avulla sovellusohjelmat voivat osallistua tapahtumaan. Toinen osa on korkean tason sovellusliittymä, jonka avulla sovelluspalvelin voi käyttää tapahtumanhallitsimen palveluja kontrolloidessaan tapahtumia. Kolmannen osan JTA:sta muodostaa standardoidun X/Open XA -protokol-

lan mukainen toteutus, jonka avulla resurssinhallitsimien (*resource manager*) ja ulkoisten tapahtumanhallitsimien välinen kommunikointi toteutetaan. Resurssinhallitsin tarjoaa sovelluksille pääsyn tapahtumanhallitsimen kautta ulkoisiin resursseihin, jotka osallistuvat tapahtumaan. (Cheung & Matena 1999, 4.) Tällainen ulkoinen resurssinhallitsimen hallitsema resurssi voi olla esimerkiksi tietokanta. Nämä kolme osaa on esitelty rajapinnoissa *javax.transaction.UserTransaction, javax.transaction.TransactionManager* ja *javax.transaction.xa.XAResource*. (Cheung & Matena 1999, 8-14.) JTA:n omaksuma tapa käyttää X/Open XA –protokollaa kommunikointiin on laajasti hyväksytty tapa eri tapahtumanhallitsimien ja resurssinhallitsimien toteuttamaan tapahtumienhallintaan (Slama, Garbis & Russell 1999, 171-172).

Kuviossa 4 on esitetty tapahtumanhallitsimen rajapinnat, joiden avulla se kommunikoi sovellusten, sovelluspalvelimien ja resurssinhallitsimien kanssa. Lisäksi kolmen tapahtumanhallitsimen kanssa kommunikoivan yksikön välille on piirretty esimerkit määrityksistä, joiden mukaisesti ne voivat kommunikoida keskenään. Tapahtumanhallitsimen ei tarvitse paljastaa sitä käyttäville yksiköille yksityiskohtaista tietoa toteuttamastaan tapahtumanhallinnasta.

Kuviossa 4 esitetyn perusteella sovellukset voivat siis kommunikoida tapahtumanhallitsimen kanssa UserTransaction-rajapintaa hyväksi käyttäen. Vastaavasti sovelluspalvelin käyttää TransactionManager-rajapintaa ja yksittäiset resurssinhallitsimet standardoidun X/Open XA -protokollan mukaista rajapinnan toteutusta kommunikointiin tapahtumanhallitsimen kanssa. Sovellustasolla toimivat ohjelmat voivat kommunikoida sovelluspalvelimen kanssa esimerkiksi EJB-määrityksen mukaisesti ja sovelluspalvelin eri resurssienhallitsimien kanssa esimerkiksi JDBC:n välityksellä.



KUVIO 4 Tapahtumanhallitsimen rajapinnat (mukailtu lähteestä Cheung 1999, 7)

4.4 Yhteenveto

Tämän luvun aluksi esiteltiin lyhyesti Javaa olio-ohjelmointikielenä sekä käsiteltiin joitain Java-ohjelmointikieleen liittyviä periaatteita. Suurimman osan luvusta muodostaa kuitenkin kuvaus Sun Microsystemsin tuottaman J2EE-ohjelmointialustan määritelmästä. Suoritusalustasta riippumaton J2EE-ohjelmointialusta on yritys pyrkiä ratkaisemaan monitasoarkkitehtuureihin pohjautuvien vahvasti liiketoimintaa tukevien järjestelmien ongelmia komponenttiajattelua hyväksikäyttäen.

J2EE muodostuu suuresta joukosta eri teknologioiden määrityksiä, ja eri J2EE-komponenttien suoritusympäristöjen eli säiliöiden on tarjottava erilaisia sovellusliittymiä ja tuettava siten eri teknologioita säiliöstä riippuen. Säiliöitä on määritelmässä esitetty neljä. Nämä säiliöt ovat sovelma-, asiakassovellus,- EJB- ja WWW-säiliöt, kahden viime mainitun ollessa palvelimella toimivia ja kahden ensiksi mainitun asiakassovelluksen päässä toimivia säiliöitä.

Luvussa esiteltiin lyhyesti kaikki eri J2EE-ohjelmointialustan sisältämät teknologioiden määritykset mutta tarkemmalla tasolla niistä esiteltiin hajautettujen järjestelmien kannalta keskeisimmät. Keskeisen osan J2EE-ohjelmointialustasta muodostaa nimipalvelu Java Naming and Directory Interface (*JNDI*), jonka avulla erilaisten resurssien paikallistaminen on mahdollista hajautetuissa järjestelmissä. Lisäksi tietovarastojen hallinta ja niiden sisältämän tiedon käsittely muodostaa suuren osan järjestelmässä tehtävän työn määrästä, mitä varten J2EE tarjoaa sovellusliittymän joustavaan ja riippumattomaan tietovarastojen käsittelyyn Java Database Connectivityn (*JDBC*) avulla.

Varsinaisen hajautuksen eri sovellusten ja komponenttien välillä J2EE toteuttaa Remote Method Invocationin (*RMI*) avulla, joka on Java-ohjelmointikielestä riippuva tapa toteuttaa hajautettujen sovellusten kommunikointi. RMI tarjoaa kuitenkin mahdollisuuden kommunikoida myös CORBAn IIOP-protokollan (*Internet Inter-ORB Protocol*) välityksellä. Viimeisenä J2EE:n keskeisimmistä määrityksistä esiteltiin tapahtumanhallinta, joka on toteutettu korkean tason sovellusliittymän Java Transaction API:n (*JTA*) sekä Java Transaction Servicen (*JTS*) avulla. JTS:n määritys esittää kuinka yhteistoiminta alemman tason palvelun eli OMG:n Object Transaction Servicen kanssa toteutetaan. Vastaavasti JTA määrittelee korkean tason sovellusliittymän, jonka avulla sovellusohjelmat voivat käyttää tapahtumanhallitsimen palveluja hyväkseen.

Tässä luvussa tarkemmin esitellyt neljä teknologiaa muodostavat vahvan perustan seuraavassa luvussa esiteltävälle EJB-komponenttimallille, ja kaikkien niiden käyttö on todennäköistä lähes aina, mikäli toteutettavaa järjestelmää käytetään liiketoimintaa todella tukevana järjestelmänä.

5 ENTERPRISE JAVABEANS –KOMPONENTTIMALLI

Tämä tutkielman kannalta oleellisin J2EE:n tarjoamista määrityksistä on Enterprise JavaBeans -komponenttimallin (*EJB*) määritys, joka tukeutuu voimakkaasti edellisessä luvussa esitettyihin muihin J2EE:n sisältämiin määrityksiin. Tässä luvussa kuvataan EJB-komponenttimallin perusperiaatteet, eri komponenttityypit sekä esitetään huomioita EJB-komponenttimallista. Kaikki EJB-komponenttimallia koskevat huomiot perustuvat, ellei erikseen ole muuta mainittu, uusimpaan komponenttimallin määrityksen versioon 1.1.

EJB-komponenttimallin tarkoituksena on määritellä komponenttimalli, jonka avulla palvelimilla toimivien komponenttien tuottaminen Java-ohjelmointikieltä käyttäen olisi helpompaa ja komponenttien kehittämisessä voitaisiin keskittyä liiketoimintalogiikan ohjelmoimiseen monimutkaisen ja usein tapauskohtaisen komponenttien hallinnan ohjelmoinnin sijaan. Koska EJB-komponenttimalli on pelkkä määritys, voi kuka tahansa sovelluspalvelimen valmistaja lisätä tuotteeseensa tuen EJB-komponenteille, toteuttamalla EJB-komponenttimallin määrityksen mukaisen komponenttien suoritusympäristön.

EJB-komponenttimallin määrityksen tarkoituksena on tuottaa Java-ohjelmointikieltä käyttävä yritystason järjestelmiä tukeva komponenttimallin kuvaus, jonka avulla voidaan toteuttaa hajautettuja järjestelmiä uudelleenkäytettäviä ja siirrettäviä komponentteja hyväksi käyttäen. Lisäksi sovellusten toteuttaminen on yksinkertaista, sillä sovelluspalvelinten valmistajien on huolehdittava komponenttien tarvitsemien palvelujen toteuttamisesta, komponenttien kehittäjien käyttäessä ainoastaan tarkoin määriteltyjä sovellusliittymiä. EJB-komponentit ovat lisäksi yhteensopivia esimerkiksi muiden Java- ja CORBA-sovellusten kanssa. Uudelleenkäyttöä ja komponenttien sovittamista kulloiseenkin sovellusympäristöön sopivaksi edistää myös EJB-komponenttimallin mahdollistama tapa, jossa osa komponentin tarvitsemista määrityksistä voidaan tehdä komponentin ominaisuuksien (*property*) arvoja muuttamal-

la komponenttia käyttöönotettaessa. Tämä tapa on samankaltainen JavaBeans –komponenttimallin kanssa.

Ensimmäisen määrityksen (1.0) tavoitteena oli lisäksi muun muassa määritellä eri roolit EJB-sovelluskehityksessä sekä varsinaisen sovelluspalvelimella toimivan ajoympäristön määritteleminen. Tällä hetkellä uusimman määrityksen (1.1) suurimpina tavoitteina on tarjota parempi tuki sovellusten kokoamiselle ja levittämiselle sekä määritellä tarkemmin eri roolien vastuualueet. (Matena & Hapner 1999, 19-20.)

5.1 Roolit

EJB-komponenttimalli määrittelee kuusi eri roolia, jotka osallistuvat komponenttipohjaiseen ohjelmistonkehitykseen EJB-järjestelmää suunniteltaessa, toteutettaessa ja ylläpidettäessä. Näistä ensimmäinen on tuottaja (*provider*), jonka tuottamana syntyy kokoelma EJB-komponentteja. Tuottaja on siis vastuussa yksittäisen komponentin sisältämän liiketoimintalogiikan ohjelmoimisesta, minkä lisäksi hän tarjoaa yksityiskohtaista tietoa muun muassa komponenttien ominaisuuksista ja niiden sidoksista muihin komponentteihin ja palveluihin. Kokoaja (*assembler*) toimii tuottajan tai tuottajien toteuttamien komponenttien yhteenliittäjänä sekä yhdistää komponentit ja niiden sisältämän tiedon toimivaksi järjestelmäksi. Kokoajan täytyy tuntea kohdealue hyvin, mutta hänen ei tarvitse tuntea komponenttien toteutustapaa rajapintoja lukuunottamatta. Sijoittaja (*deployer*) huolehtii vastaavasti kokoajan tuottaman järjestelmän käyttöönotosta jossain tietyssä ympäristössä. Sijoittajan vastuulla on huolehtia siitä, että kaikki tuottajan asettamat resurssit ovat käytössä siinä tietyssä ympäristössä, jossa komponentteja kulloinkin ollaan ottamassa käyttöön. Sijoittaja käyttää tehtävään sovelluspalvelimen tarjoamia välineitä, joiden avulla tarvittavien luokkien generointi ja varsinaisten komponenttien käyttöönotto voidaan tehdä. (Matena & Hapner 1999, 21-24.) Tuottajan ja kokoajan tehtävien suurimpana erona on se, että tuottajan tehtävänä on toteuttaa alimman tason liiketoimintalogiikka yksittäisiin komponentteihin. Vastaavasti kokoajan vastuulla on korkeamman tason liiketoimintalogiikan soveltaminen komponentteja yhdistelemällä.

Sovelluspalvelimen valmistajan tehtävänä on huolehtia tarvittavien palvelujen luomisesta palvelimelle. EJB-komponenttimallin roolimääritysten mukaisesti EJB-palvelimen ja EJB-säiliön toteuttaminen ovat eri rooleja, sillä periaatteessa palvelimen ja säiliön voi toteuttaa esimerkiksi kokonaan eri valmistaja. Palvelimen toteuttajan (*server provider*) vastuulla on huolehtia muun muassa tapahtumienhallinnasta ja alempien tasojen palvelujen käyttämisestä esimerkiksi käyttöjärjestelmätasolla sekä ajonaikaisen ympäristön luomisesta säiliöitä varten, joita voi olla samalla palvelimella useampiakin. Säiliön toteuttajan (*container provider*) vastuulla on vastaavasti varsinaisen ajoympäristön tarjoaminen komponenteille sekä sijoittajan tarvitsemien työkalujen toteuttaminen. (Matena & Hapner 1999, 21-24.) EJB-komponenttimallin määrityksestä ei kuitenkaan löydy minkäänlaista rajapinnan määritystä palvelimen ja säiliön välille. Käytännössä säiliön ja palvelimen tehtävät ovat yhteisiä vaatimuksia sovelluspalvelimen valmistajalle, jonka on toteutettava molemmat roolit. Tämän hetkisen määrityksen mukaan palvelimen ja säiliön toimittajina toimii aina sama taho (Matena & Hapner 1999, 23). Koska rajapintaa ei ole määritelty, on kahden eri toimittajan säiliön ja palvelimen yhdistäminen keskenään lähes mahdotonta, sillä kaikki valmistajat ovat luoneet omat tapansa kommunikoida säiliön ja palvelimen välillä (Roman 1999, 47).

Kuudes rooli on varattu ylläpitäjälle (*system administrator*), jonka tehtävänä on huolehtia muun muassa järjestelmän valvonnasta (Matena & Hapner 1999, 24). Palvelimet ja säiliöt voivat myös tarjota valmistajakohtaisia valvontatyökaluja, vaikka EJB-komponenttimalli ei näitä määrittelekään (Roman 1999, 50).

5.2 Suoritusympäristö

EJB-komponentit tarvitsevat toimiakseen tarkasti määritellyn suoritusympäristön, jonka keskeisen osan muodostavat palvelin ja sen sisältämät säiliöt, joita voi yhdellä palvelimella olla yksi tai useampia. Koska EJB-komponenttimallin määrityksestä puuttuu säiliön ja palvelimen välisen rajapinnan tarkka määritys, käsitellään seuraavassa yhtäläisesti molemmille kuuluvia tehtäviä ja vaatimuksia.

Säiliöiden tehtävänä on tarjota suoritusympäristö sisältämilleen komponenteille ja huolehtia muun muassa sovelluspalvelimen lähettämien tapahtumien välittämisestä kaikille säiliön komponenteille. EJB-säiliön on määrityksen mukaisesti myös tarjottava komponenteille erilaisia sovellusliittymiä. Kaikki eri säiliöiden tarjoamat sovellusliittymät on esitetty taulukossa 3 sivulla 35 (ks. myös Matena & Hapner 1999, 272). Säiliö ei koskaan toimi suoraan varsinaisena asiakkaan tai EJB-komponentin kutsun kohteena, vaan se ainoastaan välittää pyyntöjä sekä huolehtii tarvittavien palvelujen saatavuudesta ja näin ollen kantaa vastuun hajautetun komponenttiarkkitehtuurin mukanaantuomista ylimääräisistä tehtävistä. (Roman 1999, 60-62.) Säiliö toteuttaa näin EJB-komponenttimallin mukaisen toiminnan, jonka mukaisesti asiakassovellus ei koskaan kommunikoi suoraan itse komponentin kanssa vaan kommunikointi tapahtuu aina säiliön välityksellä.

Palvelimen toteuttajan erityisalueena on hajautettujen tapahtumien hallinta sekä alemman tason järjestelmäasiantuntemus, ja tyypillisesti palvelimen toteuttaja on esimerkiksi käyttöjärjestelmä- tai tietokantatuotteiden valmistaja. Vastaavasti säiliön toteuttajan vastuulle jäävät erilaisten hallintatyökalujen ja ajoympäristön toteuttaminen. (Matena & Hapner 1999, 23.) Roman (1999) ja Valesky (1999) ovat esittäneet EJB-komponenttimallin määrityksen pohjalta palveluja, joita säiliön ja palvelimen tulee toteuttaa. Näitä palveluja ovat muun muassa resurssien ja komponenttien elinkaarien hallinta, tilan ja tapahtumien hallinta, tietoturva, sijaintinäkymättömyys sekä käyttöönottosovellukset. Tehokkaalla resurssien ja komponenttien elinkaaren hallinnalla komponentteja voidaan luoda ja tuhota tarpeen mukaan sekä säilyttää niin sanotuissa ilmentymäpooleissa (*instance pool*). Tilanhallinnalla voidaan taata tilallisten komponenttien (ks. 5.3.1) tilojen pysyvyys esimerkiksi eri asiakaskutsujen välillä ja tapahtumien hallinnalla turvata tietokannan eheys päivitysten yhteydessä. Nämä edellä mainitut ominaisuudet ovat komponenteille täysin näkymättömiä EJB-komponenttimallin tarjoamia ominaisuuksia. Tietoturvasta huolehditaan sallimalla toimintojen käyttö vain niille, joilla on oikeus kyseinen toiminto suorittaa. Tietoturvan takaaminen perustuu Java 2 –alustan tapaan käyttää pääsynvalvontalistoja (*access control list, ACL*) käyttäjän tunnistamiseen (*authentication*) ja valtuuksien todentamiseen (*authorization*). Sijaintiläpinäkyvyys on oleellinen osa hajautettua järjestelmää, sillä tällöin asiakassovellusten ei tarvitse tietää missä kulloinkin tehtävä toiminto todellisuudessa suoritetaan. Osa järjestelmästä

voidaan esimerkiksi poistaa käytöstä ja siirtää poistetut komponentit tilapäisesti toiselle palvelimelle. Asiakassovellus suorittaa näin tarvitun toiminnon siellä, missä sen toteuttava komponentti sijaitsee. Käyttöönoton yhteydessä säiliön tarvitsemat ohjelmakoodit generoidaan automaattisesti sen mukaan, mitä käyttöönotettavassa järjestelmässä tarvitaan. Niinpä kaikki edellä mainitut palvelut toteutuvat vasta käyttöönoton yhteydessä kunkin säiliön toteuttajan välineiden avulla. (Roman 1999, 62-65; Valesky 1999, 21-25.) EJB-komponenttimallin määritys ei sido säiliön tai palvelimen toteuttajaa mihinkään tiettyyn ohjelmointikieleen, vaan palvelut voidaan toteuttaa millä tahansa ohjelmointikielellä.

Tämän tutkielman kannalta ei ole merkityksellistä erotella palvelimen ja säiliön hoitamia tehtäviä. Tämän vuoksi tutkielmassa viitataan suoritusympäristöön, jolla tarkoitetaan jonkin väliohjelmiston toteuttamaa EJB-komponenttimallin määrityksen mukaista komponenttien suoritusympäristöä.

5.3 Komponenttityypit

EJB-komponentit ovat binäärimuodossa olevia suoritettavia Java-ohjelmia, jotka hyödyntävät tarkasti määritellyn suoritusympäristön palveluja. Tässä mielessä Szyperskin (1998) esittämä määritelmä ohjelmistokomponentille on käyttökelpoinen EJB-komponenttien tapauksessa. Lisäksi Orfalin ym. (1996) sekä Brownin ja Wallnaun (1998) esittämä Wojtek Kozaczynskin määritelmä liiketoimintakomponentille ovat sellaisenaan melko hyviä kuvauksia siitä, millainen EJB-komponentti liiketoimintakomponenttina on. Nämä Szyperskin, Orfalin ym. (1996) ja Kozaczynskin määritelmien eri näkökulmat on liitetty yhdeksi tässä tutkielmassa käytettäväksi määritelmäksi kohdassa 3.1.4.

EJB-komponentit voidaan jakaa kahteen eri päätyyppiin, kohdepohjaisiin (*entity bean*) ja istuntopohjaisiin (*session bean*). Lisäksi istuntopohjaisia komponentteja on kahdenlaisia, tilallisia (*stateful*) ja tilattomia (*stateless*). (Matena & Hapner 1999, 34-35.) EJB-komponenttimallin määrityksen versiossa 1.0, kohdepohjaisten komponenttien tukeminen ei ollut pakollista, mutta uuden 1.1 version myötä kaikkien sovelluspalveli-

mien on tuettava myös niitä (Matena & Hapner 1999, 15). Seuraavissa kappaleissa on kuvattu istunto- ja kohdepohjaisten komponenttien keskeisimpiä piirteitä.

5.3.1 Istuntopohjaiset komponentit

Istuntopohjaiset komponentit edustavat jotain rajattua liiketoimintaprosessia, sisältäen liiketoimintalogiikan ja –säännöt sekä tietovuon, joiden mukaisesti kulloinkin suoritettavana oleva tehtävä toteutetaan. Istuntopohjaisen komponentin tyypillisiä tehtäviä ovat esimerkiksi hinnan laskeminen, tietokantaoperaatioiden suoritus ja videokuvan pakkaaminen (Roman 1999, 51.) Istuntopohjaisen komponentin ominaisuuksiin kuuluu lisäksi se, että tehtävä suoritetaan kulloinkin aina jonkin tietyn asiakassovelluksen puolesta, joten komponentin ilmentymän elinikä on suhteellisen lyhyt, rajoittuen asiakkaan vaatiman toiminnan pituuteen (Matena & Hapner 1999, 34-35).

Tilattoman istuntopohjaisen komponentin tapauksessa yksittäisen asiakkaan tekemien toimintojen historiatieto ei ole saatavilla, eikä komponentilla ole näin ollen mahdollisuutta tunnistaa sitä kutsuvaa asiakassovellusta. Tilallisen istuntopohjaisen komponentin tapauksessa tilanne on päinvastainen, sillä se voi säilyttää tietoa tilastaan ja näin ollen säilyttää tilatiedon esimerkiksi useiden metodikutsujen välillä. Esimerkiksi verkossa toimivan kaupan ostoskorina toimiva komponentti voi säilyttää tilansa niin kauan, kuin asiakassovellus sitä tarvitsee, ja tuhota tilatiedot vasta, kun niitä ei enää tarvita. (Roman 1999, 51-52.)

Istuntopohjaisen komponentin toteuttaminen perustuu *javax.ejb.SessionBean*-rajapinnan toteuttamiseen, jonka metodit jokaisen istuntopohjaisen komponentin on toteutettava (Matena & Hapner 1999, 53). Esimerkissä 1 on kuvattu kuvitteellisen tilallisen istuntopohjaisen komponentin ohjelmakoodi. Tilallinen ja tilaton istuntopohjainen komponentti poikkeavat toisistaan ainoastaan muodostimen eli *ejbCreate*-metodin osalta. Tilattomalla komponentilla saa olla määrityksen mukaan ainoastaan yksi *ejbCreate*-metodi, joka ei saa ottaa parametreja (Matena & Hapner 1999, 67). Tämä rajoite johtuu yksinkertaisesti siitä, että säiliö voi käsitellä tilattomien komponenttien ilmentymiä mielivaltaisesti muun muassa tuhoamalla ja luomalla niitä tarvittaessa. Mikäli tilattomalle

komponentille olisi mahdollista antaa alustusarvoja komponentin luonnin yhteydessä, ei säiliö tietäisi näitä alustusarvoja eikä osaisi valita oikeaa muodostinta komponentin luomiseksi. Esimerkissä 1 kuvattu komponentti olisi edellä kuvatuin muodostimeen kohdistuvin muutoksin täysin kelvollinen tilaton istuntopohjainen komponentti.

```
import javax.ejb.*;

public class HelloWorldBean implements SessionBean
{
    public String myText = null;
    public void ejbCreate()
    {}
    public void ejbCreate(String textToSay)
    {
        myText = textToSay;
    }
    public void ejbRemove()
    {}
    public void ejbActivate()
    {}
    public void ejbPassivate()
    {}
    public void setSessionContext(SessionContext ctx)
    {}
    public void sayHello()
    {
        System.out.println(myText);
    }
}
```

ESIMERKKI 1 Tilallisen istuntopohjaisen komponentin toteutus

Esimerkissä 1 on kuvattu ainoastaan yksi määrityksen ulkopuolinen liiketoimintametodi, joka on tarkoitettu asiakassovellusten kutsuttavaksi. Tämä metodi on *sayHello()*, joka tulostaa yksinkertaisen tekstin, joka on tallennettu komponentin jäsenmuuttujaksi sen luonnin yhteydessä. Kaikki muut metodit ovat *javax.ejb.SessionBean*-rajapinnan pakottamia metodeja, joiden avulla säiliö ja asiakassovellukset kommunikoivat komponentin kanssa. e*jbRemove*-metodin avulla asiakassovellus ilmoittaa komponentille, että se voidaan tarpeettomana poistaa ilmentymäpoolista ja tuhota. Tässä yhteydessä komponentin tulee vapauttaa hallitusti kaikki tarvitsemansa resurssit, kuten esimerkiksi tietoliikenneyhteydet.

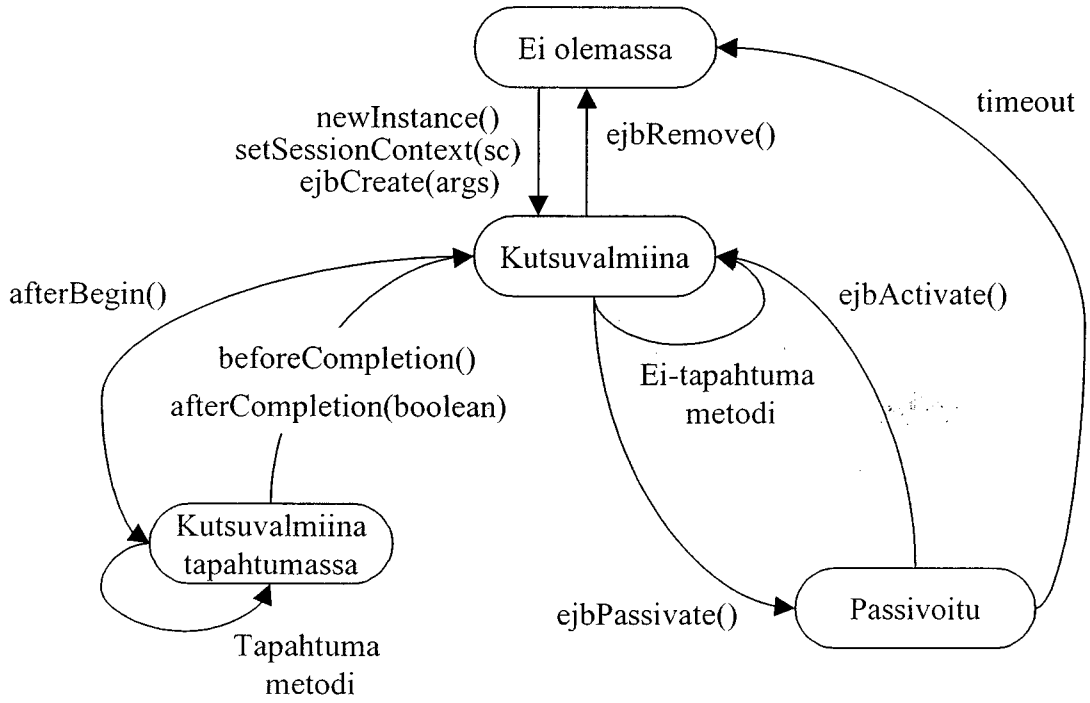
Vastaavasti *ejbPassivate*-metodin avulla säiliö voi ilmoittaa komponentille, että se aiotaan siirtää pois ilmentymäpoolista ja tallentaa johonkin pysyvään tietovarastoon.

ejbPassivate-metodikutsun yhteydessä komponentti tekee tyypillisesti samat lopetustoimenpiteet kuin *ejbRemove*-metodikutsunkin yhteydessä. *ejbActivate*-metodi toimii vastakohtana *ejbPassivate*-metodille ja se palauttaa komponentin tilan pysyvästä tietovarastosta. Palautuksen yhteydessä komponentin tulee alustaa itsensä samaan tilaan, kuin se oli ennen *ejbPassivate*-metodin kutsumista. *ejbActivate*- ja *ejbPassivate*-metodit eivät sisällä mitään toiminnallisuutta, mikäli kysymyksessä on tilaton istuntopohjainen komponentti. (Matena & Hapner 1999, 53.) Edellä todetusta seuraa, että metodien *ejbActivate*- ja *ejbPassivate* toteutukset voidaan tilattoman komponentin tapauksessa yksinkertaisesti jättää tyhjiksi.

setSessionContext-metodin avulla säiliö mahdollistaa komponentille pääsyn komponentin ja säiliön väliseen ympäristöön, kontekstiin (*context*). Säiliö kutsuu tätä metodia aina komponentin luomisen yhteydessä, mutta komponentin ei ole pakko tallettaa tätä parametrina annettua kontekstia mihinkään jäsenmuuttujaan. Konteksti perustuu *javax.ejb.SessionContext*-rajapintaan, ja säiliö tarjoaa toteutuksen tälle rajapinnalle jokaisen luodun komponentin yhteydessä. (Matena & Hapner 1999, 54.) Mikäli komponentti esimerkiksi haluaa tietää menossa olevan tapahtuman tilan, se voi tiedustella sitä säiliöltä kontekstin avulla. Kontekstin käsite on määritelty tarkemmin kohdassa 5.5.2.

Tilallinen istuntopohjainen komponentti voi lisäksi vielä halutessaan toteuttaa *javax.ejb.SessionSynchronization*-rajapinnan. Tämä rajapinta sisältää metodit, joita säiliö kutsuu tapahtumien alkaessa sekä juuri ennen tapahtuman loppumista ja sen loputtua. Komponentti voi käyttää näiden metodikutsujen avulla saatua tietoa esimerkiksi tapahtuman keskeyttämiseen tarvittaessa juuri ennen kuin tapahtuma on loppunut. (Matena & Hapner 1999, 54-55.)

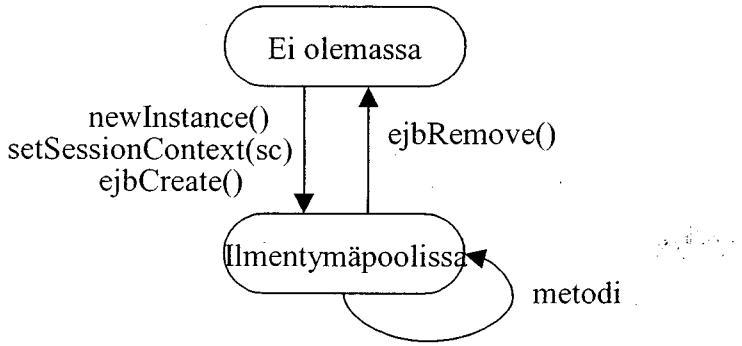
Kuviossa 5 on esitetty tilallisen istuntopohjaisen komponentin ilmentymän elinkaari, jossa on kuvattu kunkin metodikutsun aikaansaama tilanmuutos komponentissa. Tilallinen istuntopohjainen komponentti voi olla neljässä eri tilassa. Ilmentymää ei ole olemassa (*does not exist*), ilmentymä on kutsuvalmiina (*ready*), ilmentymä on passivoitu ja sen tila on tallennettu (*passivated*) tai ilmentymä on kutsuvalmiina osallisena jossain tapahtumassa (*ready in transaction*) (Matena & Hapner 1999, 102.)



KUVIO 5 Tilallisen istuntopohjaisen komponentin elinkaari (mukailtu lähteestä Matena & Hapner 1999, 57)

Kun komponentti on kutsuvalmiina, sitä voidaan kutsua suorittamaan jokin tehtävä. Mikäli metodin kutsuminen ei pakota komponenttia osallistumaan johonkin tapahtumaan (ks. 5.5.1), se voidaan suorittaa kutsuvalmiina-tilassa. Muussa tapauksessa komponentti siirtyy kutsuvalmiina tapahtumassa –tilaan ja on osallisena jossain tarkasti määritellyssä tapahtumassa. Komponentti voi siirtyä passivoitu-tilaan, mikäli säiliö haluaa poistaa komponentin ilmentymän tilapäisesti muististaan. Tapahtumaan osallista ilmentymää ei kuitenkaan koskaan voida passivoida ennen kuin tapahtuma on vahvistettu (*commit*) tai peruutettu (*rollback*). Säiliö voi lisäksi poistaa komponentin ilmentymän kokonaan mikäli ilmentymä on jonkin tietyn ennaltamääritellyn ajan passivoitu- tai kutsuvalmis-tilassa (*timeout*). Tällöin kaikki viittaukset kyseiseen komponentin ilmentymään ovat virheellisiä, koska komponentin ilmentymä on tuhottu. (Matena & Hapner 1999, 57-58.) Säiliön toteuttaja voi vapaasti valita menetelmän, jonka perusteella päätetään, mitä komponentin ilmentymiä kulloinkin poistetaan ilmentymäaltaasta. Perustapauksena voidaan pitää menetelmää, jossa pisimmän aikaa käyttämättömänä ollut komponentti poistetaan ilmentymäaltaasta (*Least Recently Used, LRU*) (ks. esim. Roman 1999, 116-117).

Kuviossa 6 on esitetty Kuvion 5 kaltainen komponentin elinkaarta kuvaavat siirtymät tilattoman istuntopohjaisen komponentin ilmentymälle.



KUVIO 6 Tilattoman istuntopohjaisen komponentin elinkaari (mukailtu lähteestä Matena & Hapner 1999, 69)

Kuviosta 6 voidaan havaita, että tilattomalla istuntopohjaisella komponentilla on oleellisesti vähemmän tiloja ja siirtymiä kuin tilallisella istuntopohjaisella komponentilla. Asiakassovelluksen halutessa luoda ilmentymän tietystä komponentista säiliö luo tarvittaessa uuden ilmentymän tai käyttää ilmentymäpoolissa jo olemassaolevaa ilmentymää. Säiliö myös poistaa tarvittaessa ilmentymiä ilmentymäaltaasta. (Matena & Hapner 1999, 69.) Tärkeimpänä erona tilalliseen istuntopohjaiseen komponenttiin on se, että asiakassovelluksen muodostimen kutsu ei todellisuudessa välttämättä aiheuta uuden ilmentymän luomista säiliössä. Niinpä tilasiirtymä ei olemassa –tilasta ilmentymäpoolissa-tilaan on täysin riippumaton asiakassovelluksen suorittamasta *ejbCreate*-metodikutsusta.

5.3.2 Kohdepohjaiset komponentit

Kohdepohjaiset komponentit edustavat EJB-komponenttimallissa pysyvää tietoa. Kohdepohjaisen komponentin tärkein ominaisuus on sen pitkäikäisyys, sillä jokainen kohdepohjainen komponentti edustaa näkymää johonkin pysyvään tietovarastoon tietoon, kuten esimerkiksi tietokannan taulun riviin. Samaa komponenttia hyödyntävät useat asiakkaat, ja päivitysten tekemisestä sekä siihen liittyvistä samanaikaisuusongelmista huolehditaan tapahtumanhallintamekanismien avulla. Lisäksi kohdepohjaiset komponentit selviytyvät esimerkiksi palvelimella tapahtuvista virheistä, sillä komponentin tila

palautetaan esimerkiksi koko palvelinohjelman kaatuessa automaattisesti viimeiseen tietokantaan vahvistettuun tilaan. (Matena & Hapner 1999, 35.) Koska kohdepohjaiset komponentit edustavat ainoastaan olioperustaista näkymää pysyvään tietoon, ne täytyy aika ajoin tahdistaa varsinaisen tietovaraston kanssa (Matena & Hapner 1999, 85). Huomattavaa on myös, että mikään ei pakota kohdepohjaisia komponentteja kommunikoimaan ainoastaan tietokantojen kanssa. Kohdepohjaisten komponenttien tapauksessa viitattava tietovarasto saattaa olla esimerkiksi jokin toiminnanohjausjärjestelmä tai keskustietokoneella toimiva sovellus. (Matena & Hapner 1999, 99).

Kommunikointi muiden elementtien kuin tietokantojen kanssa tullaan toteuttamaan liittimien (*connector*) avulla, jotka mahdollistavat komponentin kommunikoinnin esimerkiksi toiminnanohjausjärjestelmien kanssa. Kytkimien määrittely tullaan kuitenkin toteuttamaan vasta J2EE-ohjelmointialustan tulevissa versioissa. (ks. Shannon 1999, 11-2.) Tässä tutkielmassa ei oteta tarkemmin kantaa kohdepohjaisen komponentin viittaamaan tietoon ja viitattavasta kohteesta käytetään yleisesti tietovarasto-termiä.

Kohdepohjaiset komponentit eivät, toisin kuin istuntopohjaiset komponentit, sisällä lainkaan liiketoimintalogiikkaa, vaan niiden sovelluslogiikka rajoittuu niiden kykyyn hakea ja päivittää tietovaraston tietoja tarvittaessa. Toisin kuin istuntopohjaisten komponenttien tapauksessa kohdepohjaisille komponenteille ei ole välttämätöntä määritellä muodostinta, mikäli asiakassovelluksille ei haluta antaa mahdollisuutta luoda uutta tietoa tietovarastoon. Koska kohdepohjainen komponentti edustaa ainoastaan näkymää pysyvään tietoon, voidaan tietoa päivittää tietovarastoon komponentin sijaan esimerkiksi jonkin täysin järjestelmän ulkopuolisen sovelluksen avulla. (Matena & Hapner 1999, 85; Roman 1999, 192-193.) Vastaavasti kohdepohjaiselle komponentille voidaan tilallisen istuntopohjaisen komponentin tapaan määritellä useita muodostimia, mikäli tietovarastoon halutaan luoda uutta tietoa komponentin välityksellä (Matena & Hapner 1999, 89.) Tällöin muodostimien on luonnollisesti erottava parametrilistojensa perusteella toisistaan.

Kohdepohjaisten komponenttien sisältämä ohjelmakoodi on monimutkaisempaa kuin yksinkertaisten istuntopohjaisten komponenttien, sillä niiden on väistämättä kommunikoitava tietovaraston kanssa, jossa niiden sisältämä tieto varastoidaan pysyvästi. Esimerkissä 2 on esitetty *javax.ejb.EntityBean*-rajapinta, joka jokaisen kohdepohjaisen komponentin on toteutettava. Esimerkki 2 ei siis sellaisenaan ole vertailukelpoinen Esimerkissä 1 esitetyn istuntopohjaisen komponentin varsinaisen ohjelmakoodin kanssa.

```
public interface javax.ejb.EntityBean
      implements javax.ejb.EnterPriseBean
{
    public abstract void setEntityContext(javax.ejb.EntityContext);
    public abstract void unsetEntityContext();
    public abstract void ejbRemove();
    public abstract void ejbActivate();
    public abstract void ejbPassivate();
    public abstract void ejbLoad();
    public abstract void ejbStore();
}
```

ESIMERKKI 2 javax.ejb.EntityBean-rajapinta

setEntityContext-metodi on samankaltainen kohdassa 5.3.1 esitetyn istuntopohjaisen komponentin *setSessionContext*-metodin kanssa. Säiliö kutsuu *setEntityContext*-metodia lisätessään komponentin ilmentymäpooliin, mutta komponentin ei ole pakko talletttaa tätä kontekstia mihinkään jäsenmuuttujaan. Konteksti perustuu *javax.ejb.EntityContext*-rajapintaan ja säiliö tarjoaa toteutuksen tälle rajapinnalle. (Matena & Hapner 1999, 102, 147-148.) Komponentti voi tiedustella esimerkiksi suoritusympäristöään koskevia tietoja tämän kontekstin avulla. Vastaavasti un*setEntityContext*-metodin avulla säiliö ilmoittaa komponentille poistavansa sen ilmentymäpoolista, ja komponentti voi vapauttaa mahdollisesti varaamiaan resursseja ennen automaattisesti tapahtuvaa roskienkeruuta. Resurssit ovat tyypillisesti sellaisia, jotka on varattu *setEntityContext*-metodikutsun yhteydessä. (Matena & Hapner 1999, 105.)

ejbRemove-metodin avulla asiakassovellus voi poistaa komponentin esittämän tiedon tietovarastosta. Tämä metodi toteuttaa varsinaisen tiedon tuhoamisen tietovarastosta esimerkiksi JDBC-rajapinnan avulla. Tämän jälkeen komponentti siirretään ilmentymäpooliin, tilaan jossa sille ei ole osoitettu mitään tietovaraston tietoa esitettäväksi. Tässä suhteessa komponentin tila on samankaltainen *ejbPassivate*-metodikutsua seuraavaan

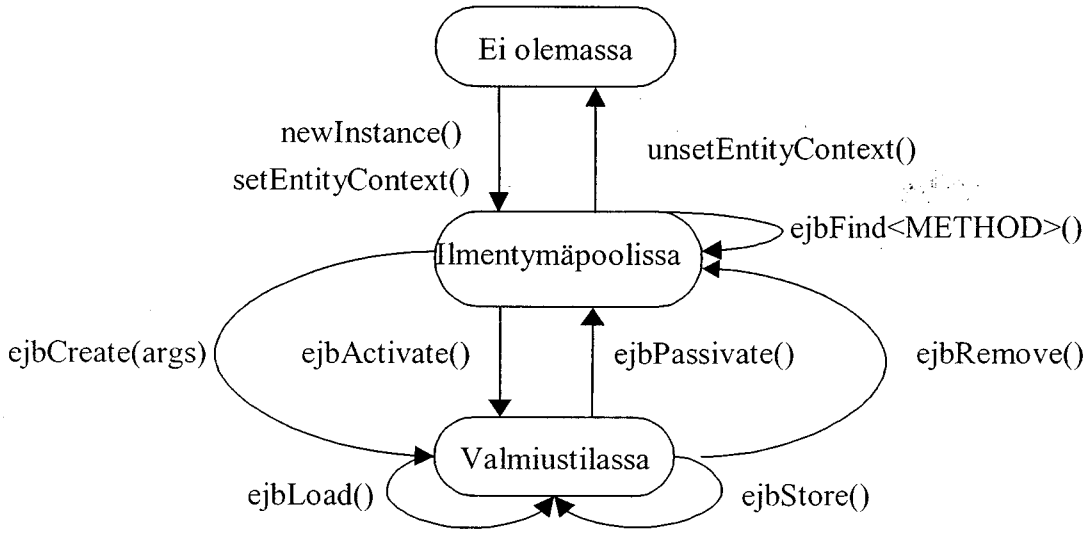
tilaan. *ejbPassivate*-metodilla säiliö voi tarvittaessa siirtää komponentin pois valmiustilasta ilmentymäpooliin. Samalla komponentin tulee vapauttaa kaikki varaamansa resurssit ja tahdistaa tilansa tietovaraston kanssa *ejbStore*-metodin avulla. *ejbActivate*-metodi toimii päinvastoin kuin *ejbPassivate*-metodi ja se palauttaa komponentin tilan passivointia edeltävään tilaan muun muassa varaamalla käyttöönsä kaikki komponentin tarvitsemat resurssit. (Matena & Hapner 1999, 106.)

ejbLoad-metodia kutsutaan aina heti *ejbActivate*-metodin kutsumisen jälkeen palauttamaan tietovaraston tiedot komponentin tietorakenteeseen. Vastaavasti *ejbStore*-metodia kutsutaan tallentamaan komponentin sisältämät tiedot tietovarastoon. *ejbStore*-metodia kutsutaan tapahtumienhallinnan osalta aina, kun on tarpeellista synkronoida komponentin esittämä tieto tietovaraston kanssa. Lisäksi *ejbStore*-metodia kutsutaan aina ennen *ejbPassivate*-metodin kutsua, jotta mitään tietoa ei menetetä. (Matena & Hapner 1999, 106-107.) *ejbStore*- ja *ejbLoad*-metodi voivat toteuttaa tiedon tallennuksen ja haun esimerkiksi JDBC-rajapinnan avulla.

Edellä kuvattujen metodien lisäksi kohdepohjaisen komponentin on toteutettava ainakin yksi ilmentymiensä etsintää tukeva metodi. Yhden näistä etsintämetodeista on aina oltava nimeltään *ejbFindByPrimaryKey*, jonka avulla komponenttia voidaan etsiä tietovarastosta sen avaimen arvon perusteella. Lisäksi komponentille voidaan määritellä rajaton määrä muita etsintää tukevia metodeja. Näiden etsintämetodien tulee aina palauttaa yksi tai useampia avainluokan (*primary key class*) ilmentymiä. Jokaiselle kohdepohjaiselle komponentille on määriteltävä oma avainluokkansa, joka sisältää yksikäsitteisen tunnuksen kullekin kohdepohjaisen komponentin ilmentymälle. (Matena & Hapner 1999, 109.)

Kuviossa 7 on esitetty kohdepohjaisen komponentin ilmentymän elinkaari, jossa on kuvattu kunkin metodikutsun aikaansaama tilanmuutos komponentissa. Kohdepohjaisen komponentin ilmentymä voi määrityksen mukaisesti olla kolmessa eri tilassa. Ilmentymää ei ole olemassa (*does not exist*), ilmentymä on ilmentymäpoolissa ilman tarkempaa identiteettiä (*pooled*), eli sille ei ole osoitettu mitään tietovaraston tietoa esitettäväksi mutta se on olemassa ilmentymäpoolissa, tai ilmentymä on valmiustilassa

(*ready*), jolloin se sisältää jotain tiettyä tietovaraston tietoa ja on näin ollen valmiina käytettäväksi johonkin ennaltamääritettyyn tarkoitukseen. (Matena & Hapner 1999, 102.)



KUVIO 7 Kohdepohjaisen komponentin elinkaari (mukailtu lähteestä Matena & Hapner 1999, 102)

Kuviota 7 tutkimalla voidaan todeta säiliön käyttämä menetelmä pitää kohdepohjaisia komponentteja ilmentymäpoolissa ilman, että niille on vielä osoitettu mitään tietovaraston tietoa esitettäväksi. Koska komponentin ilmentymien luominen on resursseja kuluttava operaatio, on järkevää, että säiliö pitää ilmentymiä ilmentymäpoolissa valmiina odottamassa muodostimen kutsua (Roman 1999, 185-186). Kun komponentin ilmentymä on luotu, se voidaan tarvittaessa passivoida ja palauttaa ilmentymäaltaaseen. Huomioitavaa on myös se, ettei *ejbRemove*-metodikutsun suorittaminen tuhoa itse komponentin ilmentymää vaan ilmentymän osoittaman tiedon tietovarastosta.

5.3.3 Komponenttien tilanhallinta

Komponenttien tilanhallinnalla viitataan komponenttien ilmentymien kykyyn säilyttää oma tilansa eli tallentaa pysyvästi sisältämiensä muuttujien ja tietorakenteiden arvot. Istuntopohjaisten komponenttien tapauksessa tilanhallinta tulee kysymykseen ainoastaan

tilallisten komponenttien kohdalla, sillä tilattomien komponenttien tilaa ei komponentin luonteen vuoksi ole mahdollista tallentaa pysyvästi.

Jokaisella komponentilla voi lisäksi olla käytössään staattisia *static*-avainsanalla varustettuja jäsenmuuttujia, jotka muodostavat komponentin tilan. EJB-määrityksen mukaisesti staattisten arvojen lukeminen on sallittua mutta niiden arvon muuttaminen ei. Tämän vuoksi kaikki staattiset jäsenmuuttujat tulisi EJB-määrityksen suositusten mukaisesti varustaa *final*-avainsanalla (ks. Matena & Hapner 1999, 272).

Tilallisen istuntopohjaisen komponentin tilanhallinta toteutuu kutsuttaessa *ejbPassivate*-metodia, jonka kutsumisen jälkeen säiliö tallentaa komponentin tilan asianmukaisesti. Säiliö voi kutsua *ejbPassivate*-metodia, mikäli esimerkiksi säiliölle määrätty yhtäaikaa muistissa olevien komponenttien lukumäärä ylitetään. Vastaavasti, kun asiakassovellus kutsuu komponenttia, joka on poistettu muistista kutsutaan, *ejbActivate*-metodia, jolloin komponentin tila palautetaan passivointia edeltävään tilaan. Huomattavaa on kuitenkin, että istuntopohjaisten komponenttien luonteesta johtuen ne eivät osaa palauttaa aikaisempaa tilaansa esimerkiksi sovelluspalvelimen uudelleenkäynnistyksen tai sovelluspalvelimella tapahtuvan peruuttamattoman virheen yhteydessä.

Kohdepohjaisten komponenttien tilanhallinta toteutuu samankaltaisesti istuntopohjaisten komponenttien kanssa. *ejbPassivate*-metodikutsulla säiliö voi ilmoittaa komponentille siirtävänsä sen ilmentymäpooliin. Vastaavasti *ejbActivate*-metodikutsulla säiliö ilmoittaa komponentille siirtävänsä sen takaisin käyttöön ja yhdistävänsä sen johonkin tiettyyn avainarvoon ja tietovaraston tietoon. Lisäksi kohdepohjaista komponenttia aktivoitaessa ja passivoitaessa komponentin tila tulee tahdistaa tietovaraston kanssa *ejbStore*- ja *ejbLoad*-metodien avulla.

Komponentin tilan tallentamisen tekee mahdolliseksi Javan sarjallistuvuusrajapinta (*java.io.Serializable*), jonka jokainen EJB-komponentti toteuttaa välillisesti. Tämä rajapinnan välillinen toteuttaminen johtuu siitä, että rajapinta *javax.ejb.EnterpriseBean*, josta kaikki komponenttityypit periytyvät, perii tämän sarjallistuvuusrajapinnan. (Roman 1999, 117-118.)

Uusimman EJB-komponenttimallin määrityksen mukaisesti säiliön tulee kyetä tallentamaan perustietotyyppien lisäksi kaikki sarjallistuvat tietotyypit sekä viitteet EJB-olioihin ja kotiolioihin (ks. 5.4) (Roman 1999, 652-653.) Sovelluspalvelimen toteuttaja voi myös vapaasti valita, kuinka komponenttien tilat tallennetaan. Tila voidaan tallentaa tiedostoon tai tarvittaessa tietokantaan, mutta komponentin tuottajan, kokoajan tai sijoittajan ei tarvitse olla tietoinen tästä tilanhallinnan toteutustavasta.

5.3.4 Komponenttien pysyvyydenhallinta

Pysyvyydellä (*persistence*) tarkoitetaan EJB-komponenttien tapauksessa kohdepohjaisten komponenttien kykyä pitää sisältämänsä tieto ajantasaisena verrattuna siihen tietovaraston tietoon, jota komponentti edustaa (Matena & Hapner 1999, 99).

Kohdepohjaisen komponentin esittämä tieto on tallennettuna johonkin pysyvään tietovarastoon ja kohdepohjainen komponentti esittää tätä tietoa oliomuodossa. Kohdepohjaisten komponenttien tapauksessa on mahdollista valita, suoritetaanko pysyvyydenhallinta automaattisesti säiliön toimesta vai suoritetaanko komponentin pysyvyydenhallinta komponentin tuottajan ohjelmoimien haku- ja tallennusmetodien avulla säiliön niin pyytäessä.

Mikäli pysyvyydenhallinta suoritetaan automaattisesti, säiliö tuottaa toteutuksen kaikille tarvittaville kohdepohjaisen komponentin metodeille. Komponentin tuottajan ei siis tarvitse huolehtia tiedon hakemisesta ja tallentamisesta lainkaan. Käytännössä säiliön toteuttamassa automaattisessa pysyvyydenhallinnassa *ejbCreate*-, *ejbLoad*-, *ejbStore*-, *ejbRemove*- ja *ejbFind*-metodit pidetään sellaisina, että niissä ei käsitellä tietovarastoa lainkaan. Kaikki edellä lueteltujen metodien tarvitsemat tietovarastoon kohdistuvat kutsut luodaan käyttöönoton yhteydessä säiliön toteuttajan tarjoamilla välineillä. (Matena & Hapner 1999, 100-101.) Edellä lueteltujen metodien täytyy kuitenkin luonnollisesti toteuttaa kaikki muu niiltä odotettu toiminnallisuus. Lisäksi kuvaustiedostossa (ks. 5.4.3) on kerrottava säiliölle, mitkä kentät komponentissa ovat sellaisia, jotka tulee tallentaa pysyvästi.

Vastaavasti komponentin itse suorittamassa pysyvyydenhallinnassa kaikkien tietovarastoa käsittelevien metodien toteutus on komponentin tuottajan vastuulla. Suurin ero kahden eri kohdepohjaisen komponentin tilanhallinnassa se, että toisessa tapauksessa kaikki ohjelmakoodi on kiinteästi osana komponenttia, kun taas toisessa ohjelmakoodi generoidaan vasta käyttöönoton yhteydessä (Matena & Hapner 1999, 101). Komponentin itse suorittaman pysyvyydenhallinnan ongelmat liittyvät siihen, että tapa jolla komponentin tila tallennetaan, on ohjelmoitu kiinteästi osaksi komponenttia. Toisaalta säiliön ei tällöin tarvitse luoda näitä tietovarastoa käyttäviä kutsuja lainkaan. Säiliön toteuttamassa pysyvyydenhallinnassa tarvittava ohjelmakoodi generoidaan käyttöönoton yhteydessä ja se täytyy myös tuottaa kaikille tietovarastoille erikseen. Samalla kuitenkin varmistutaan siirrettävyydestä, sillä komponentti itse ei sisällä mitään pysyvyydenhallintaan liittyvää ohjelmakoodia. Toisaalta esimerkiksi Hill ja Salo (1999) esittävät säiliön toteuttaman tilanhallinnan ongelmaksi sen, että Javan perustietotyyppien lisäksi ei voida olla varmoja eri säiliöiden kyvystä tallentaa muuta komponentin sisältämää tietoa. Mikäli tallennettavat rakenteet ovat monimutkaisia, on ainoa ratkaisu turvautua komponentin itse toteuttamaan pysyvyydenhallintaan, jolloin voidaan varmistua siitä, että kaikkien komponentin sisältämien tietojen tallennus onnistuu ainakin toteutushetkellä käytössä olevassa säiliössä. (Hill & Salo 1999, 6-7.) EJB-komponenttinallin määrityksen versiossa 1.1 on kuitenkin korjattu puutteita, joihin esimerkiksi Hill ja Salo (1999) puuttuvat. Kuten aiemmin jo todettiin, uusimman määrityksen mukaisesti säiliön tulee osata tallentaa perustietotyyppien lisäksi kaikki sarjallistuvat tietotyypit sekä viitteet EJB-olioihin ja kotiolioihin (Roman 1999, 652-653.)

Tässä yhteydessä ei oteta tarkemmin kantaa kohdepohjaisten komponenttien pysyvyydenhallinnan eroihin tai niiden käyttötarkoituksiin.

5.4 Komponentti kokonaisuutena

Valmis, suoritus- ja levityskelpoinen EJB-komponentti on paljon muutakin kuin vain itse komponentin ohjelmakoodi. Seuraavassa on lyhyesti esitelty kaikki toimivan EJB-komponentin sisältämät tiedostot ja sen tarvitsemat palvelut.
Komponentin tuottajan on itse komponentin ohjelmakoodin lisäksi tuotettava muita tiedostoja, jotka sisältävät tietoa komponentista ja auttavat sen käyttöönotossa. Kotirajapinta (*home interface*) on rajapinta, jonka avulla säiliö tuottaa toteutuksen kotioliolle (*home object*). Tämän olion avulla asiakassovellus saa viitteen haluamaansa komponenttiin. Kotioliolla on aina jokin tietty yksikäsitteinen nimi, jolla se on rekisteröity nimipalveluun ja jolla sitä voidaan sieltä etsiä. Kotirajapinta sisältää esittelyn kaikista komponentin mahdollista luontipalveluista parametreineen. Kun asiakassovellus on saanut viitteen kotiolioon, se voi kutsua kotiolion luontimetodia luodakseen uuden ilmentymän EJB-oliosta (*EJB-object*). Tämä EJB-olio on vastaavasti säiliön tuottama toteutus etärajapinnalle (*remote interface*), joka sisältää esittelyn kaikista komponentin sisältämistä palveluista, jotka se tarjoaa. Sekä kotirajapinta että etärajapinta ovat RMI:n mukaisia rajapintojen määrittelyjä, mikä mahdollistaa säiliön toteuttaman olioiden hajautuksen. Lisäksi kaikkien tietotyyppien on oltava RMI-IIOP - kommunikointiin sopivia, sillä säiliön toteuttaja voi käyttää kommunikointiin RMI:n käyttämän JRMP:m sijaan myös IIOP:tä. (Matena & Hapner 1999, 199.)

Kaikki asiakkaan tekemät palvelupyynnöt kulkevat EJB-olion kautta, eikä asiakkaalla ole koskaan suoraa viitettä varsinaiseen komponentin ilmentymään (*instance*), joita hallitsee ja joiden luomisesta ja tuhoamisesta säiliö vastaa täysin itsenäisesti. Niinpä esimerkiksi kotirajapinnan kautta tehty luomispyyntö uudelle EJB-oliolle ei välttämättä saa aikaan uuden ilmentymän luomista säiliössä, sillä säiliöllä voi olla valmiina ilmentymäpoolissaan ilmentymä, jonka viite EJB-oliolle luonnin yhteydessä palautetaan. Kohdepohjaisille komponenteille on lisäksi aina määriteltävä oma avainluokkansa, joka sisältää yksikäsitteisen tunnuksen kullekin kohdepohjaisen komponentin ilmentymälle.

Komponentin ohjelmakoodin, koti- ja etärajapinnan lisäksi tuottajan tulee luoda myös kuvaustiedosto (*deployment descriptor*), joka sisältää tiedon komponentin säiliöltä

vaatimista ominaisuuksista, kuten esimerkiksi tarvittavasta tapahtuman-, pysyvyyden- tai tietoturvanhallinnasta. Tämä tiedosto tuotetaan usein myös automaattisesti sovelluskehittimien avulla. Lisäksi komponentin ominaisuuksia voidaan muokata niin sanotun ominaisuustiedoston (*properties file*) avulla, joka voi sisältää tietoa esimerkiksi hinnoittelukomponentin käyttämästä alennusprosentista. Näiden kahden edelläkuvatun tiedoston oleellisin ero on siinä, että kuvaustiedosto on ohjeellinen kuvaus siitä, mitä komponentti vaatii suoritusympäristöltä, ja ominaisuustiedosto vastaavasti määrittelee komponentin sisäisesti käyttämiä tietoja.

Kun kaikki tuottajan velvollisuuksiin kuuluvat tiedostot on luotu, on komponentti vielä muutettava levityskelpoiseen ja helposti käyttöönotettavaan muotoon. EJB-komponentit käyttävät samaa ZIP-pakkausformaattiin perustuvaa JAR-tiedostomuotoa (*Java archive*) levityspakettien pakkaamiseen kuin muutkin Java-sovellukset. Tämä ejb-jar -tiedosto sisältää itse komponentin luokkatiedoston lisäksi etä- ja kotirajapinnat sekä ominaisuus- ja kuvaustiedostot. Mikäli kehitystyötä tehdään jollain sovelluskehittimellä tämän tiedoston luominen voidaan automatisoida. Muussa tapauksessa se voidaan tuottaa *jar*-työkaluohjelman avulla, joka on osa J2EE-ohjelmointialustaa (Roman 1999, 109-110.)

Seuraavissa alakohdissa on kuvattu tarkemmin osa edellä kuvatuista komponentin kannalta tärkeimmistä tiedostoista. Esiteltyjen tiedostojen osalta on myös esitetty niiden keskeisimmät erot istunto- ja kohdepohjaisten komponenttien tapauksessa.

5.4.1 Kotirajapinta

Kotirajapinta on pienin eroavaisuuksin samanlainen istunto- ja kohdepohjaisilla komponenteilla. Molempien komponenttien tapauksessa kotirajapinnan tulee tarjota esittelyt *ejbCreate*- ja *ejbRemove*-metodeille. Kuten aiemmin on todettu, tilattomalla istuntopohjaisella komponentilla muodostin on aina parametriton ja vastaavasti kohdepohjaiselle komponentille muodostinta ei ole välttämätöntä esitellä lainkaan. Lisäksi kotirajapinnan avulla voidaan noutaa viite kotirajapintaan sekä metatietoa komponentista, mutta nämä toiminnat eivät kuitenkaan vaikuta kotirajapinnan esittelyihin mitenkään. (Matena & Hapner 1999, 89, 42.) Kohdepohjaisen komponentin

on lisäksi esiteltävä kotirajapinnassaan kaikki etsintämetodit. Määrityksen vaatimusten mukaisesti kotirajapinnassa on tällöin esiteltävä ainakin *findByPrimaryKey*-metodi. (Matena & Hapner 1999, 89.)

Kotirajapinnan toteutus perustuu komponenttityypistä riippumatta aina *javax.ejb.EJBHome*-rajapinnan toteuttamiseen. Esimerkissä 3 on esitetty kotirajapinnan määrittely kuvitteelliselle kohdepohjaiselle komponentille, joka pitää yllä laskuria, jonka arvoa voidaan kasvattaa. Laskurikomponentteja voidaan myös etsiä niiden omistajan tai suurimman laskuriarvon mukaan.

```
import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;

public interface CounterHome implements EJBHome
{
    Counter create(String counterID, String owner)
        throws CreateException, RemoteException;
    Public Counter findByPrimaryKey(CounterPK key)
        throws FinderException, RemoteException;
    Public Counter findBiggestCounter()
        throws FinderException, RemoteException;
    Public Counter findByOwner(String owner)
        throws FinderException, RemoteException;
}
```

ESIMERKKI 3 Kohdepohjaisen komponentin kotirajapinnan määrittely

5.4.2 Etärajapinta

Etärajapinnan tarkoituksena on esitellä kaikki komponentit tarjoamat liiketoimintametodit, joita asiakassovelluksen on tarkoitus käyttää. Säiliö toteuttaa etärajapinnassa esitellyt metodit ejb-oliossa, jonka avulla varsinaiset metodikutsut välitetään komponentin ilmentymälle. Etärajapinta toteuttaa molempien komponenttityyppien tapauksessa *javax.ejb.EJBObject*-rajapinnan. Esimerkissä 4 on esitetty etärajapinnan määrittely esimerkissä 3 esitetylle kuvitteelliselle kohdepohjaiselle laskurikomponentille. Etärajapinnassa on esitelty metodit laskurin arvon kasvattamista ja vähentämistä varten sekä metodi laskurin omistajan nimen palauttamista varten.

```
import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Counter implements EJBObject
{
    public void increaseCounter();
    public void decreaseCounter();
    public String getOwnerName();
}
```

ESIMERKKI 4 Kohdepohjaisen komponentin etärajapinnan määrittely

5.4.3 Avainluokka

Kuten aiemmin on todettu, jokaiselle kohdepohjaiselle komponentille on määriteltävä oma avainluokkansa, joka sisältää yksikäsitteisen tunnuksen kullekin kohdepohjaiselle komponentille. EJB-komponenttimallin määrityksen mukaisesti kahden kohdepohjaisen komponentin ilmentymän oletetaan olevan identtisiä, jos niiden kotirajapinnan ja avainluokan ilmentymät ovat identtisiä.

Avainluokan ilmentymä toimii siis yksikäsitteisenä tunnisteena kohdepohjaiselle komponentille. EJB-komponenttimallin mukaisesti avainluokka voi sisältää mielivaltaisia kenttiä, joiden avulla tunnistaminen voidaan tehdä. Ainoa ehto avainluokkaa koskien on se, että sen on toteutettava *java.io.Serializable*-rajapinta, jotta se voidaan välittää oliona tietoverkossa. Esimerkissä 5 on esitetty kuvitteellisen avainluokan ohjelmakoodi.

```
import java.io.Serializable;

public class CounterPK implements java.io.Serializable
{
    public String counterID;
    public counterPK()
    {}
    public CounterPK(String id)
    {
        this.counterID = id;
    }
    public String toString()
    {
        return counterID;
    }
   public int hashCode()
    {
        return counterID.hashCode();
    }
    public boolean equals(Object counterObjPK)
    {
        return ((CounterPK)counterObjPK).
               counterID.equals(counterID);
    }
}
```

ESIMERKKI 5 Avainluokan ohjelmakoodi

Esimerkissä 5 esitetty avainluokan kuvitteellinen toteutus liittyy aiemmissa esimerkeissä käytettyyn laskurikomponenttiin. Varsinaista laskurikomponentin ohjelmakoodia ei kuitenkaan ole esitetty, joten esimerkissä 5 esitetyn ohjelmakoodin oikeellisuutta ei voi yksiselitteisesti todeta. Mikäli oletetaan edellä esitetyn avainluokan ohjelmakoodin olevan oikeellinen, tulee laskurikomponentin toteutuksessa esitellä julkinen (*public*) jäsenmuuttuja *counterID*, joka on tietotyypiltään sama esimerkissä 5 esitetyn *counterID* jäsenmuuttujan kanssa. Avainluokan ohjelmakoodissa on lisäksi toteutettava *hashCode*- ja *equals*-metodit, joilla helpotetaan asiakassovellusten toimintaa (Matena & Hapner 1999, 127).

Edellä esimerkissä 5 kuvatussa avainluokassa komponentin yksikäsitteisenä tunnisteena käytettiin yhtä merkkijonoa, jonka on siis oltava jokaiselle laskurikomponentille yksikäsitteinen. Avainluokan voi kuitenkin muodostaa esimerkistä 5 poiketen useampikin komponentin julkinen jäsenmuuttuja. Tässä tutkielmassa ei oteta kantaa siihen, kuinka

yksikäsitteinen avain komponentille luodaan. Yksikäsitteisten avaimien generoimiseksi on kuitenkin olemassa useita eri tapoja (ks. esim. Roman 1999).

5.4.4 Kuvaustiedosto

Kuvaustiedoston tärkein tehtävä on toimia eräänlaisena sopimuksena kokoajan tuottaman ejb-jar -tiedoston ja sen käyttöönottavan sijoittajan välillä. Käytännössä tuottajan luoma ejb-jar -tiedosto ei sisällä käyttöönoton kannalta oleellista tietoa, vaan vasta kokoajan luoma ejb-jar -tiedosto sisältää useiden komponenttien lisäksi kuvaustiedoston, jonka perusteella käyttöönotto voidaan tehdä. Kuvaustiedosto sisältää siis tiedon siitä, kuinka ejb-jar -tiedoston sisältö voidaan muuttaa toimivaksi järjestelmäksi tai sen osaksi. (Matena & Hapner 1999, 239.) EJB-komponenttimallin määrittelyn 1.1 versiossa kuvaustiedosto on muutettu aiemmasta poikkeavasti sellaiseksi, että se voi sisältää tietoa yhdestä tai useammasta komponentista. Jokaiselle komponentille ei siis näin ollen tarvitse enää määritellä omaa kuvaustiedostoaan (Roman 1999, 636.)

Kuvaustiedosto on fyysiseltä muodoltaan XML-tiedosto (*eXtensible Markup Language*), jonka oikeellisuuden tarkistus toteutetaan DTD-tiedoston (*Data Type Definition*) avulla. (Ks. Matena & Hapner 1999, 244 ja Goldfarb & Prescod 1998). Kuvaustiedosto sisältää muun muassa tiedon jokaisen komponentin toteuttavan luokan nimestä, sen koti- ja etärajapinnoista sekä komponentin tyypistä. Tämän tutkielman kannalta ei ole merkityksellistä esitellä laajasti kuvaustiedoston sisältämiä tietoja, joten tarkempi esittely jätetään tekemättä.

5.4.5 Ominaisuustiedosto

Ominaisuustiedosto poikkeaa kuvaustiedostosta siinä mielessä, että se määrittelee aina joitain komponentin sisäisesti käyttämiä tietoja. Ominaisuustiedostossa voidaan määritellä sellaisia tietoja, joiden voidaan olettaa muuttuvan tai joita todennäköisesti tullaan lisäämään tai poistamaan komponenttia käytettäessä. Lisäksi ominaisuustiedos-

ton käyttäminen mahdollistaa sen, ettei komponentin ohjelmakoodissa tarvitse kiinteästi määritellä kaikkia muuttujien arvoja.

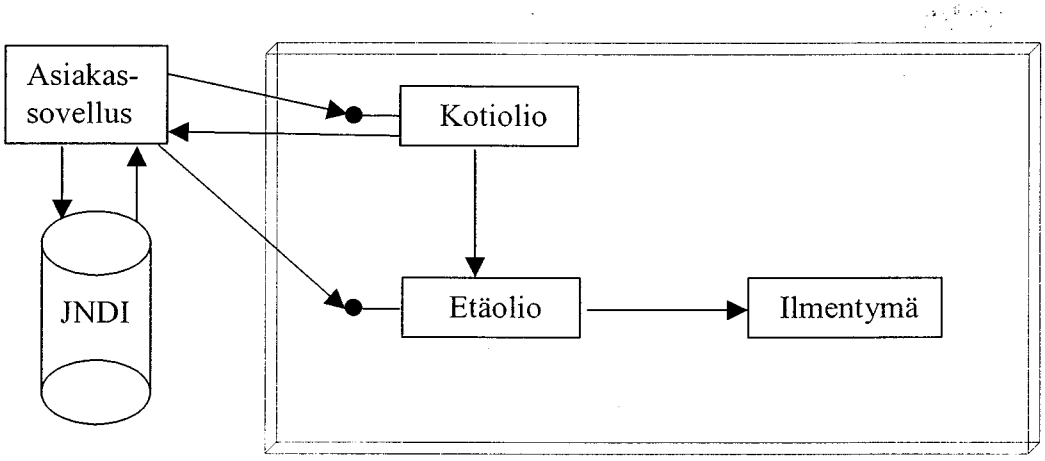
Ominaisuustiedosto sisältää tietoja, joiden komponentti olettaa olevan saatavilla suoritusympäristössä suorituksen aikana. Ominaisuustiedosto sisältää siis ohjeita sijoittajaa varten, joka ottaa komponentin käyttöön jossain tietyssä ympäristössä. Komponentti ei kuitenkaan voi lukea ominaisuustiedoston sisältämiä arvoja suorituksen aikana, vaan kaikkien ominaisuustiedoston sisältämien tietojen tulee olla saatavilla suorituksen aikana suoritusympäristössä. Tämä rajoitus johtuu siitä, että EJB-komponentti ei voi käyttää hyväkseen *java.io*-paketin tarjoamia palveluja esimerkiksi hakemistojen ja tiedostojen käsittelyyn suorituksen aikana (Matena & Hapner 1999, 273).

Komponentin käyttämän alennusprosentin kiinnittäminen esimerkiksi asiakkaan mukaan voidaan toteuttaa helposti ominaisuustiedoston avulla, yksinkertaisesti vain lisäämällä ominaisuustiedostoon käyttäjien tunnistetiedot ja alennusprosentti (ks. esim. Roman 1999, 437). Näitä tietoja voidaan myös helposti päivittää, mikäli esimerkiksi asiakkaiden määrä kasvaa.

Huomattavaa on kuitenkin, että laajemmissa järjestelmissä ominaisuustiedoston käyttö saattaa johtaa hallitsemattomaan tietojen päivitykseen. Lisäksi ominaisuustiedoston muokkaaminen on mahdollista tavallisella tekstieditorilla, joten kuka tahansa voi muokata sen sisältämiä tietoja. Suositeltavaa suurempien tietomäärien ja liiketoiminnan kannalta kriittisempien tietojen kohdalla onkin käyttää esimerkiksi tarkoitusta varten erikseen toteutettua ohjelmaa, jolla komponentin käyttämiä ulkoisia tietoja voidaan muokata ja mahdollisesti tallentaa esimerkiksi tietokantaan. Tietokannassa tiedot ovat muokattavissa ainoastaan asianmukaisten käyttöoikeuksien omaavien henkilöiden ja sovellusten avulla. Lisäksi tietokannassa säilytettävien tietojen tapauksessa komponentti voi ajonaikaisestikin hakea tietoja ja saman komponentin eri ilmentymät voivat käyttää tarvittaessa jopa eri tietoja.

5.5 Komponenttien ja suoritusympäristön yhteistoiminta

Komponenttien löytäminen ja niiden palvelujen käyttäminen on oleellista EJB-komponenttimalliin pohjautuvan järjestelmän käytössä. Seuraavassa on esitetty yksinkertaistettu esimerkki siitä, kuinka asiakassovellus saa viitteen haluamaansa komponenttiin ja kuinka komponentin palvelujen käyttäminen käytännössä tapahtuu. Kuviossa 8 on esitetty toimivan EJB-komponentin suoritusympäristö.



KUVIO 8 EJB-komponentin suoritusympäristö (mukailtu lähteestä Roman 1999, 93)

Kuvion 8 esittämä suoritusympäristö on voimakkaasti yksinkertaistettu esitys siitä, kuinka mahdollisimman yksinkertainen EJB-komponenttimallin mukainen toteutus voisi toimia. Lisäksi kuvio ei ota mitään kantaa asiakassovelluksen toimintaan eikä nimeä erikseen, minkälainen komponentti on kyseessä.

Asiakassovelluksen on ensiksi saatava viite kotiolioon, jonka asiakassovellus noutaa määritellyltä nimipalvelulta. Asiakassovellus tietää tässä vaiheessä ainoastaan käyttämänsä nimipalvelun sijainnin ja etsimänsä komponentin nimen nimipalvelussa. Kun viite kotiolioon on saatu, asiakassovellus voi sen avulla luoda uuden EJB-olion, jonka kautta liikennöinti itse komponenttiin tapahtuu. Samalla säiliö komponenttityypistä riippuen joko luo uuden komponentti-ilmentymän tai, mikäli ilmentymäpoolissa sillä hetkellä on jo olemassa ilmentymä kyseiselle komponentille, palauttaa viitteen tähän jo olemassaolevaan komponentin ilmentymään. Tämän jälkeen

komponentin tarjoamat palvelut ovat käytettävissä asiakassovelluksessa saadun etäolion viitteen avulla.

Periaatteiltaan kaikki toiminta komponenttien ja niitä käyttävien asiakassovellusten välillä on samankaltaista kuin edellä ja kuviossa 8 kuvattiin. Kuviosta 8 poiketen ja edellä esitettyyn pohjautuen komponentteja on toiminnaltaan ja komponenttityypiltään erilaisia, sillä pysyvän tiedon varastointi ja itse liiketoimintalogiikan toteuttaminen toteutetaan eri komponenttityyppien avulla. Lisäksi suuremmissa järjestelmissä asiakassovelluksia on samanaikaisesti useampia ja ne voivat käyttää osin samoja resursseja. Asiakassovellus ei myöskään välttämättä toteuta liikennöintiä komponenttien kanssa suoraan etäolion avulla, vaan esimerkiksi HTML-sivun (*HyperText Markup Language*) tai sovelman välityksellä voidaan olla yhteydessä servletiin, joka hakee viitteet ja käyttää etäolioita kutsujen suorittamiseen.

5.5.1 Tapahtumanhallinta

EJB-komponenttimallin käyttämä tapahtumanhallinta perustuu kohdassa 4.3.4 esiteltyihin JTA:han ja JTS:ään. Erityisen oleelliseksi tapahtumanhallinnan EJB-komponenttimallin tapauksessa tekee se, että asiakassovelluksen suorittamat toimenpiteet saattavat aiheuttaa toimintaa hajautetusti useissa eri palvelimissa ja tietokannoissa. (Matena & Hapner 1999, 153).

EJB-komponenttimalli tarjoaa kaksi mahdollisuutta toteuttaa tapahtumanhallinta komponentissa. Komponentti voi toteuttaa tapahtumanhallinnan itse (*bean-managed transaction demarcation*) tai se voidaan jättää säiliön toteutettavaksi (*container-managed transaction demarcation*). Säiliön toteuttaman tapahtumanhallinnan tapauksessa sovelluksen kokoajan tehtävänä on määritellä kuvaustiedostossa kunkin komponentin metodin tarvitsema tapahtumanhallintatapa. (Matena & Hapner 1999, 153-154.) Komponentin itsensä toteuttaman tapahtumanhallinnan tapauksessa sovelluksen kokoajan ei tule määritellä mitään tapahtumanhallintaan liittyviä parametreja. Lisäksi komponentin itsensä toteuttama tapahtumanhallinta on mahdollista ainoastaan istuntopohjaisten komponenttien tapauksessa (Matena & Hapner 1999, 173).

Säiliön toteuttamassa tapahtumanhallinnassa voidaan kullekin metodille määritellä jokin kuudesta tapahtumanhallintaan liittyvästä parametrista. Nämä parametrit lyhyine kuvauksineen on esitetty taulukossa 5. Mikäli komponentti toteuttaa *javax.ejb.SessionSynchronization*-rajapinnan se voi käyttää ainoastaan *Required, RequiresNew* ja *Mandatory* parametreja, sillä muuten säiliö ei pystyisi ilmoittamaan tapahtumista säiliölle rajapinnassa esiteltyjen metodien avulla (Matena & Hapner 1999, 170).

TAULUKKO 5 Säiliön toteuttaman tapahtumanhallinnan parametrin mahdolliset arvot

Parametrin arvo	**Kuvaus**
NotSupported	Metodi ei tue tapahtumanhallintaa. Kaikki menossa olevat tapahtumat asetetaan odottamaan tämän metodikutsun ajaksi.
Required	Metodi vaatii tapahtumanhallinnan. Mikäli metodia kutsutaan ilman tapahtumaa, luodaan uusi tapahtuma metodia varten.
Supports	Mikäli metodia kutsutaan ilman tapahtumaa, toimii kuten NotSupported, jos kutsutaan jonkin tapahtuman aikana, toimii kuten Required.
RequiresNew	Mikäli metodia kutsutaan ilman tapahtumaa, luodaan uusi tapahtuma. Mikäli metodia kutsutaan tapahtuman aikana, luodaan uusi tapahtuma ja alkuperäinen tapahtuma jää odottamaan.
Mandatory	Metodia on kutsuttava jonkin tapahtuman sisällä. Toimii kuten Required. Ilman tapahtumaa tehtävä kutsu aiheuttaa poikkeuksen.
Never	Metodin kutsu tapahtuman sisällä aiheuttaa poikkeuksen. Ilman tapahtumaa tehty kutsuu toimii kuten NotSupported.

Esimerkissä 6 on esitetty komponentin toteuttaman tapahtumanhallinnan sisältävän istuntopohjaisen komponentin osittainen ohjelmakoodi. Ohjelmakoodi ei sellaisenaan ole toimintakelpoinen, sillä se sisältää vain kuvitteellisen transactionalMethod-metodin määrittelyn. Lisäksi komponentti käyttää täysin kuvitteellista nimeä tietokantayhteyden avaamiseen ja suorittaa kutsuja tietokantaan ilman mitään järkeviä parametreja.

Komponentin toteuttamassa tapahtumanhallinnassa komponentti käyttää hyväkseen säiliön toteuttamaa *javax.transaction.UserTransaction*-rajapintaa, jonka avulla se ilmoittaa säiliölle tapahtuman alkamisesta ja päättymisestä. Tilattoman istuntopohjaisen komponentin on aina vahvistettava tapahtuma ennen metodikutsun paluuta. Vastaavasti tilallinen komponentti säilyttää tilansa metodikutsujen välissä, joten se voi mutta sen ei ole pakko vahvistaa tapahtumaa jokaisen metodikutsun yhteydessä. Tilallisen komponentin tapauksessa tapahtuma voi siis jatkua useiden metodikutsujen ajan. (Matena & Hapner 1999, 161.) Esimerkissä 6 esitetty ohjelmakoodi olisi vähäisin muutoksin muutettavissa vastaamaan säiliön toteuttaman tapahtumanhallinnan vaatimuksia, poistamalla kaikki *javax.transaction.UserTransaction* -rajapintaan kohdistuvat metodikutsut.

```
public class MySessionTransactionExample implements SessionBean
{
    EJBContext ejbContext;
    public void transactionalMethod()
    {
        javax.transaction.UserTransaction ut;
        javax.sql.DataSource ds;
        java.sql.Connection c;
        java.sql.Statement s;
        InitialContext ic = new InitialContext();
        ds = (javax.sql.DataSource)
            ic.lookup("java:comp/env/jdbcDB");
        c = ds.getConnection();
        s = c.createStatement();
        ut = ejbContext.getUserTransaction();
        ut.begin();
        s.executeQuery(...);
        s.executeUpdate(...);
        ut.commit();
        s.close();
        c.close();
    }
}
```

ESIMERKKI 6 Istuntopohjaisen komponentin toteuttama tapahtumanhallinta

Esimerkissä 6 esitetty ohjelmakoodi toteuttaa tapahtumanhallinnan siten, että se antaa vahvistamisen tapahtumanhallitsimen tehtäväksi. Jos vahvistaminen haluttaisiin toteuttaa resurssinhallitsinkohtaisesti tulisi esimerkissä 6 esitetystä ohjelmakoodista poistaa kaikki kohdat, joissa viitataan tapahtuman aloittamiseen ja vahvistamiseen. getUserTransaction-metodin kutsuminen olisi näin ollen tarpeetonta ja *ut.begin*- ja *ut.commit* – metodikutsuja ei suoritettaisi. Sen sijaan luodulle yhteydelle ja sen resurssinhallitsijalle

annettaisiin ennen yhteyden sulkemista käsky vahvistaa tehty tapahtuma metodikutsulla *c.commit*. Mikäli kuitenkin toimitaan esimerkissä 6 esitetyllä tavalla, näiden resurssihallitsinkohtaisten *java.sql.Connection*-rajapinnassa esiteltyjen *commit*- ja *rollback*-metodien kutsuminen ei ole sallittua. (Matena & Hapner 1999, 161).

EJB-komponenttimalli mahdollistaa säiliön ja komponentin toteuttaman tapahtumanhallinnan lisäksi myös kolmannen tavan tapahtumanhallintaan. Asiakassovellus voi toteuttaa tapahtumanhallinnan saman edellä kuvatun *javax.transaction.UserTransaction*-rajapinnan avulla, johon asiakassovellus voi hakea viitteen JNDI-sovellusliittymän avulla (ks. Shannon 1999, 6-15 ja Matena & Hapner 1999, 156-157). Asiakassovelluksen toteuttamaa tapahtumanhallintaa voi olla syytä käyttää esimerkiksi, jos asiakassovellus kutsuu useita pieniä komponentteja suorittamaan jonkin isomman kokonaisen tehtävän (Roman 1999, 298).

Itse tapahtumanhallinnan lisäksi on mahdollista määritellä, kuinka tiukasti samanaikaiset tapahtumat ovat toisistaan eristettyjä. EJB-komponenttimalli tukee neljää eri eristystasoa (*isolation level*), joilla tapahtuman eristyvyysaste voidaan määritellä (ks. esim. Matena & Hapner 1999, 160-161 ja Roman 1999, 284-285). Eristyvyysasteet ovat kuitenkin resurssinhallitsinkohtaisia ja niiden tarkempi käsittely on rajattu tämän tutkielman ulkopuolelle.

5.5.2 Konteksti

Kaikki istuntopohjaiset komponentit saavat ilmentymän luonnin yhteydessä säiliöltä viitteen sen toteuttamaan *javax.ejb.SessionContext*-rajapintaan.Vastaavasti kaikki kohdepohjaiset komponentit saavat viitteen säiliön toteuttamaan *javax.ejb.EntityContext*-rajapintaan. Näiden rajapintojen avulla komponentti voi tiedustella säiliöltä suoritusaikaisia tietoja muun muassa parhaillaan menossa olevan tapahtuman tilasta.

javax.ejb.SessionContext-rajapinnan *getEJBObject*-metodi palauttaa viitteen istuntopohjaisen komponentin EJB-olioon ja *getEJBHome*-metodi viitteen kotirajapin-

taan. Näitä metodeja voivat käyttää kaikki istuntopohjaiset komponentit tapahtumanhallintatavasta riippumatta. Mikäli istuntopohjaisen komponentin tapahtumia hallitaan säiliön toimesta, komponentti voi kutsua *setRollbackOnly*- ja *getRollbackOnly*-metodeja. Ensimmäinen näistä metodeista merkitsee suorituksessa olevan tapahtuman niin, että tapahtuman lopputulos tulee olemaan vahvistamaton ja peruttu tapahtuma. Vastaavasti jälkimmäisen metodin avulla komponentti voi tiedustella säiliöltä, onko menossa oleva tapahtuma merkitty peruutettavaksi. Jos istuntopohjainen komponentti huolehti itse tapahtumanhallinnastaan, se voi käyttää *getUserTransaction*-metodia saadakseen viitteen säiliön toteuttamaan *javax.ejb.UserTransaction*-rajapintaan (ks. 5.5.1).

javax.ejb.SessionContext-rajapinta sisältää myös kaksi tietoturvan takaamiseen liittyvää metodia, joiden avulla voidaan todeta metodin kutsujan identiteetti (ks. esim. Matena & Hapner 1999, 54). Tietoturvapiirteiden käsittely on kuitenkin rajattu tutkielman ulkopuolelle, joten näitä ominaisuuksia ei käsitellä tarkemmin.

javax.ejb.SessionContext- ja *javax.ejb.EntityContext*-rajapinnat toteuttavat kumpikin yleisemmän tason *javax.ejb.EJBContext*-rajapinnan. Tästä syystä niiden metodit ovat samankaltaisia ja niinpä *javax.ejb.EntityContext*-rajapinta sisältää ainoastaan yhden *javax.ejb.SessionContext*-rajapinnasta poikkeavan metodin, *getPrimaryKey*-metodin. *getPrimaryKey*-metodilla saadaan viite avainarvoon, joka kohdepohjaisella komponentin ilmentymällä kutsuhetkellä on. Roman (1999) esittää *getPrimaryKey*-metodille komponentin itse toteuttamassa tilanhallinnassa useita hyödyllisiä käyttötarkoituksia, sillä esimerkiksi *ejbRemove*- ja *ejbLoad*-metodien yhteydessä ei välttämättä tiedetä tarkkaan, minkä avaimen mukaista tietoa ollaan poistamassa tai hakemassa, ellei avaimen arvoa noudeta ennen metodikutsujen varsinaista suorittamista (ks. Roman 1999, 199-201). getPrimaryKey-metodin kutsuminen istuntopohjaiselle komponentille aiheuttaa *java.rmi.RemoteException* tyyppisen poikkeuksen (ks. Matena & Hapner 1999, 47).

5.6 Yhteenveto

Tämän luvun keskeisimmän sisällön muodosti Enterprise JavaBeans –komponenttimallin (*EJB*) esittely. EJB-määrityksen keskeisin tavoite on määritellä komponenttimalli,

jota hyödyntämällä Java-ohjelmointikielellä toteutettujen siirrettävien palvelinkomponenttien toteuttaminen olisi helpompaa verrattuna valmistaja- ja ohjelmointikielikohtaisiin ratkaisuihin. Lisäksi tarkan määrityksen avulla komponenttien suunnittelijat ja toteuttajat voivat keskittyä liiketoimintalogiikan toteuttamiseen komponenttien hallinnan toteuttamisen sijaan.

Luvun aluksi esiteltiin eri roolit, jotka osallistuvat komponenttipohjaiseen ohjelmistonkehitykseen EJB-järjestelmää suunniteltaessa, toteutettaessa ja ylläpidettäessä. Nämä roolit ovat ohjeellisia ja niiden määrittelyjä on parannettu uusimmassa määrityksessä (1.1). Tämän jälkeen esiteltiin perustietoa EJB-komponenttien suoritusympäristöstä eli säiliöstä ja palvelimesta, jolla säiliöt fyysisesti sijaitsevat.

Eri komponenttityyppien esittely ja niiden tilan- ja pysyvyydenhallinnan esittely muodosti suuren osan luvun sisällöstä. EJB-komponenttimalli määrittelee kolme eri komponenttityyppiä: istuntopohjaiset tilattomat ja tilalliset komponentit sekä kohdepohjaisen komponentin. Kaikkien komponenttien olemassaoloon on selkeä syy, sillä niiden käyttötarkoitukset ovat erilaiset. Kohdepohjaiset komponentit edustavat näkymää pysyvään tietovarastoon, kun taas istuntopohjaiset komponentit sisältävät varsinaisen liiketoimintalogiikan. Komponenttien tilanhallinnalla viitataan tässä yhteydessä niiden kykyyn säilyttää tilansa pysyvästi, esimerkiksi metodikutsujen välillä. Vastaavasti pysyvyydenhallinta on kohdepohjaisiin komponentteihin liittyvä menetelmä, jonka avulla voidaan varmistua siitä, että kohdepohjaisen komponentin esittämä tieto on ajantasaista verrattuna tietovaraston sisältämään varsinaiseen tietoon. Edellä esitettyjen EJB-komponenttimallin määrittelemien istunto- ja kohdepohjaisten komponenttien voidaan katsoa edustavan melko hyvin sitä määritelmää komponentille, joka esitettiin tutkielman kohdassa 3.1.4.

Eri komponenttityyppien esittelyn jälkeen käsiteltiin tarkemmin komponenttia kokonaisuutena, sillä jokainen komponentti koostuu useista osista. Jokaisella komponentilla on itse komponentin ohjelmakoodin lisäksi oltava liiketoimintametodit esittelevä etärajapinta sekä luomismetodit esittelevä ja komponentin luomisen mahdol-

listava kotirajapinta. Usean komponentin muodostamalla kokonaisuudella tulee myös olla kuvaustiedosto, jossa määritellään muun muassa komponenttien tarvitsemat palvelut ja niiden vaatima tapahtumanhallinta. Jokaisella komponentilla voi myös olla ominaisuustiedosto, jolla komponentin toiminnallisuutta voidaan muokata komponentin sisäistä rakennetta muuttamatta. Lisäksi kohdepohjaisilla komponenteilla on avainluokka, jonka avulla ne voidaan tunnistaa yksiselitteisesti.

Luvun lopuksi käsiteltiin komponenttien ja suoritusympäristön yhteistoimintaa. Tässä yhteydessä esitettiin hyvin yksinkertaistettu esimerkki siitä, kuinka asiakassovellus saa viitteen palvelimella olevaan komponenttiin. Samalla esiteltiin komponenttien tapahtumanhallinnan periaatteet, joiden mukaisesti komponenttitasolla tapahtumanhallinta voidaan joko toteuttaa säiliön toimesta täysin automaattisesti tai sitten komponentti voi itse hallita tapahtumansa. Lisäksi kolmas tapa on antaa asiakassovelluksen hoitaa kaikki tapahtumanhallintaan liittyvät toimenpiteet. Tapahtumanhallintatavan valinta on täysin tilannekohtainen asia, eikä tämän tutkielman puitteissa otettu kantaa eri tapahtumanhallintatapojen eroihin tai käyttötarkoituksiin.

6 EJB-KOMPONENTTIMALLIIN POHJAUTUVA OHJELMISTONKEHITYS

Edellisissä luvuissa esitellyt J2EE-ohjelmointialusta ja EJB-komponenttimalli luovat pohjan tämän luvun keskeisimmälle annille, EJB-komponenttimalliin pohjautuvalle ohjelmistonkehitykselle.

Tässä luvussa esitetään joitain yleisiä komponenttien toteutusperiaatteita sekä EJB-komponenttimallin ja J2EE:n asettamat vaatimukset ohjelmistonkehitykselle. Lisäksi pyritään löytämään keskeiset tehtävät, jotka EJB-komponenttimallia menestyksellisesti sovellettaessa tulee suorittaa. Esimerkkien avulla pyritään havainnollistamaan näiden tehtävien tarpeellisuutta sekä osaltaan avartamaan näkökulmaa komponenttipohjaiseen EJB-määritykseen pohjautuvaan ohjelmistonkehitykseen.

6.1 J2EE:n ja EJB-komponenttimallin asettamat vaatimukset ohjelmistonkehitykselle

EJB-komponenttimallin määritys asettaa joitain erittäin tarkkoja vaatimuksia siitä, kuinka Javan ohjelmointikielellisiä rakenteita voidaan käyttää EJB-komponentteja toteutettaessa. Tässä yhteydessä ei oteta tarkemmin kantaa säiliön toteuttajan vastuuseen eri rajapintojen toteuttamisessa.

Erittäin vahva ja samalla melko rajaava vaatimus on se, että EJB-komponentit eivät saa käynnistää uusia säikeitä. Tämä rajoitus säikeiden käytöstä perustuu siihen, että säiliön toteutus voi käyttää yhden virtuaalikoneen sijaan useita eri virtuaalikoneita hallitsemaan komponenttien ilmentymiä ja näin ollen tahdistus eri virtuaalikoneiden välillä olisi mahdotonta. Samasta syystä staattisten *static*-määreellä varustettujen attribuuttien arvojen lukeminen on sallittua mutta niiden arvon muuttaminen ei. (Matena & Hapner 1999, 272.) Muun muassa edellä kuvattujen rajoitusten vuoksi J2EE:n eri komponentit eivät siis voi käyttää osaa J2SE:hen kuuluvista palveluista. Sun Microsystems ei kuiten-

kaan halua horjuttaa J2SE:n asemaa itsenäisenä tuotteena, joten sen tarjoamia palveluja ei tule muokata J2EE:n rajoitusten vuoksi. Eri palvelujen käyttörajoitukset perustuvatkin Shannonin (1999) esittämiin J2SE:n tietoturvamäärityksiin, joiden avulla määritellään ne minimaaliset vaatimukset, joiden kanssa jokaisen J2EE-tuotteen tulee olla yhteensopiva. (Shannon 1999, 6-3 – 6-5.)

Vahva rajoitus koskee myös EJB-komponenttien kommunikointia ulkomaailmaan erilaisten näyttö- tai syöttölaitteiden avulla. EJB-komponenttien ei tule käyttää AWT-toiminnallisuutta (*Abstract Window Toolkit,* ks. esim. Horstmann & Cornell 1998) esimerkiksi arvojen näyttämiseen näytöllä, sillä useat palvelimet asettavat rajoituksia sovellusohjelman ja kytkettyjen oheislaitteiden käytön välille. Lisäksi suora kommunikointi palvelimen tiedostojärjestelmän kanssa *java.io*-pakettia käyttäen on kiellettyä (ks. 5.4.5). Kommunikointiin ja tietojen talletukseen tulee käyttää siihen parhaiten soveltuvaa sovellusliittymää, kuten esimerkiksi JDBC:tä. (Matena & Hapner 1999, 272-273.)

Matena ja Hapner (1999) esittävät runsaasti muitakin rajoituksia, jotka parantavat erityisesti tietoturvaa. Tälläinen tietoturvaa parantava ja toteutusta rajaava sääntö on esimerkiksi se, ettei komponentti saa antaa *this*-viitettä parametrina mihinkään komponentin ulkopuoliseen metodiin. *This*-viitteen antamisen sijaan tulee käyttää tapauksesta riippuen joko *SessionContext.getEJBObject-* tai *EntityContext.getEJBObject*-metodeja. (Matena & Hapner 1999, 273-274.)

6.2 Yleisiä komponenttien toteutusperiaatteita

Kohdassa 3.1.4 esitettiin tämän tutkielman puitteissa käytettävä komponentin määritelmä. Sen mukaisesti komponentin katsotaan olevan itsenäinen suorituskelpoinen yksikkö, jonka tarjoamat palvelut ja ulkoiset riippuvuudet on täsmällisesti määritelty. Lisäksi komponentti tulee voida korvata toisella samat palvelut toteuttavalla komponentilla ja sen yhdistämisen muihin komponentteihin tulee olla mahdollista. Komponentin oletetaan myös useimmiten olevan liiketoimintakomponentti siinä mielessä, että se on jonkin itsenäisen liiketoimintakäsitteen tai -prosessin kuvaus.

Seuraavassa on esitetty joitain yleisiä ainakin jossain määrin komponentin tyypistä tai käytettävästä komponenttimallista riippumattomia suunnittelu- ja toteutusperiaatteita sekä joitain komponenttien määritelmissä kirjallisuudessa usein esiintyviä vaatimuksia. Erityisesti tarkoituksena on tarkastella näitä periaatteita ja vaatimuksia EJB-komponenttimallin hyödyntämisen kannalta.

6.2.1 Komponenttien väliset kytkennät

Eräs sovelias periaate on pyrkiä vähentämään kytkentöjä komponenttien kesken (D'Souza & Wills 1998, 641). Mitä enemmän komponentit ovat suorassa viittaussuhteessa toisiinsa, sitä vaikeampi niitä on erottaa toisistaan ja esimerkiksi käyttää uudelleen toisessa ympäristössä. Kytkentä ilmenee välittömästi, kun jokin komponentti käyttää suoraan toista komponenttia, jonkin metodikutsun kohteena tai mikäli jokin komponentti saa tarpeettomasti viitteitä toisiin komponentteihin esimerkiksi metodin parametrien välityksellä. EJB-komponenttimallin mukaisessa ohjelmistonkehityksessä asiakassovellus voi käyttää useita komponentteja hyödykseen eräänlaisen julkisivun (*façade*) avulla. Julkisivun käsitteenä ja sen käytön suunnittelumallina on esittänyt ainakin Gamma ym. (1995). Julkisivuna on komponentti, joka toimii kutsujen välittäjänä muille komponenteille ja pitää sisällään tarvittavat viitteet niihin. Asiakassovelluksen kannalta riippuvuuksia ei kuitenkaan ole kuin muutamiin komponentteihin sen sijaan, että jokaista komponenttia käsiteltäisiin suoraan ilman julkisivukomponentin käyttöä.

6.2.2 Rajapinnat

Luvussa kolme esitetyistä kirjallisuudessa esiintyneistä komponenttien määritelmistä useat korostavat rajapintojen merkitystä. Myös tämän tutkielman puitteissa käytettävä komponentin määritelmä velvoittaa komponentin määrittelemään rajapintansa. Rajapintojen merkitystä voidaankin pitää suurena sekä komponentin sisäisen toiminnallisuuden, että ulkoisten riippuvuuksien määrittelyssä.

EJB-komponenttimallissa komponentti tarjoaa tietoa komponentin suoritusympäristöltään vaatimista ominaisuuksista ominaisuustiedostossa, joka määrittelee kaikki sellaiset omainaisuudet, jotka säiliön on komponentille tarjottava. Vastaavasti EJB-komponentin etärajapinta kuvaa kaikki ne palvelut, joita komponentti tarjoaa. Lisäksi EJB-komponenttimallia hyödynnettäessä rajapinnat ovat aina standardimuotoisia, sillä niiden on periydyttävä tietyistä olemassaolevista rajapinnoista. Nämä rajapinnat pakottavat esittelemään jokaiselle komponenttityypille pakolliset metodit, mikä tekee rajapinnoista hyvin samankaltaisia.

6.2.3 Käyttöliittymän erottaminen liiketoimintalogiikasta

Käyttöliittymän ja liiketoimintalogiikan erottaminen on eräs monitasoarkkitehtuureihin pohjautuvan ohjelmistonkehityksen tavoitteista, ja myös D'Souza ja Wills (1998) esittävät käyttöliittymän ja liiketoimintalogiikan eriyttämistä toisistaan. EJB-komponenttimalli tarjoaa mahdollisuuden erittäin laihan asiakassovelluksen toteuttamiseen, sillä kaikki liiketoimintalogiikka on komponenttimallin mukaisesti mahdollista sijoittaa yhdelle tai useammalle palvelimelle.

D'Souza ja Wills (1998) esittävät lisäksi riippumattomuuden alustasta olevan komponentille tavoiteltava piirre. Tällä he tarkoittavat riippumattomuutta ohjelmointikielestä, tilanhallinnasta ja hajautusmekanismeista (D'Souza & Wills 1998, 649). Riippumattomuus ohjelmointikielestä ei luonnollisestikaan toteudu EJB-komponenttimallin tapauksessa, sillä toteutuskielenä on aina Java. Tilanhallinnan ja hajautusmekanismien riippumattomuutta voidaan kuitenkin pitää melko suurena. Erityisesti säiliön toteuttaman tilanhallinnan tapauksessa (ks. 5.3.3) komponentin ei tarvitse itse huolehtia mistään tilanhallinnan toimenpiteistä. Riippuvuus alustaan tilanhallinnassa kasvaa kuitenkin mikäli käytetään komponentin itsensä toteuttamaa tilanhallintaa, sillä jokaisen komponentin tilanhallinta joudutaan ohjelmoimaan kiinteäksi osaksi komponenttia. Hajautuksen suhteen riippumattomuus toteutuu hyvin, sillä EJB-komponenttimalli on suunniteltu tukemaan hajautusta eikä ohjelmoijan tarvitse puuttua hajautukseen liittyviin toimenpiteisiin juuri lainkaan.

6.2.4 Muita yleisiä komponenttien toteutusperiaatteita

Booch (1987) esittää suuntaviivoja uudelleenkäytölle. Uudelleenkäytettävän ohjelmistonosan tulee olla muun muassa ylläpidettävä, tehokas, luotettava ja ymmärrettävä. Näiden periaatteiden hyödyntäminen EJB-komponenttimallin mukaisessa ohjelmistonkehityksessä on osittain mahdollista. Ylläpidettävyys ei välttämättä toteudu ilman lähdekoodia, eikä lähdekoodin olemassaoloa EJB-komponentin tapauksessa voida pitää välttämättömänä. Toisaalta hyvin suunnitellun EJB-komponentin ylläpidettävyys toteutuu ainakin osittain ominaisuustiedoston avulla (ks. 5.4.5), jonka sisältämiä arvoja muuttamalla komponentin sisäistä toimintaa voidaan muuttaa ilman lähdekoodin muuttamistakin. Tehokkuus ja luotettavuus ovat melko vaikeita tekijöitä todeta objektiivisesti, eikä niiden mittaamista voi komponenttimallin avulla juurikaan helpottaa. Ymmärrettävyyden toteaminen on kuitenkin helpompaa, ja EJB-komponenttien tapauksessa ymmärrettävyyttä edistävät tarkat määritykset, joita komponenttien on noudatettava.

Szyperski (1998) ottaa kantaa säikeiden käyttöön ohjelmistotuotannossa. Säikeiden käyttö väistämättä monimutkaistaa ohjelmiston toimintaa ja kasvattaa riskiä samanaikaisuusongelmien hallinnassa esimerkiksi tietokannoissa. Järkevimpänä keinona samanaikaisuusongelmien välttämiseksi Szyperski pitää tapahtumien käyttöä (Szyperski 1998, 307-308.) EJB-komponenttimalli tarjoaa ratkaisun niin tapahtumien kuin säikeidenkin käytön osalta. EJB-komponentit eivät itse saa käynnistää säikeitä eivätkä käyttää mitään Javan säikeiden tahdistusmetodeja, vaan säiliö käyttää säikeitä tarpeen mukaan ja huolehtii niiden toiminnasta (ks. myös 6.1.) Lisäksi EJB-komponenttimalli tukee tapahtumanhallintaa. (ks. 5.5.1).

6.3 EJB-komponenttimallille sopivia suunnittelu- ja toteutusperiaatteita

Seuraavissa kappaleissa on esitetty joitain EJB-komponenttimallin mukaiseen ohjelmistonkehitykseen sopivia komponenttien suunnittelu- ja toteutusperiaatteita. Periaatteet ovat osittain melko yleisiä, mutta paikoin myös täysin EJB-komponentteihin

keskittyviä. Näkökulmaksi on kuitenkin tietoisesti valittu esitettävien periaatteiden soveltuvuus EJB-komponenttien kehittämiseen.

6.3.1 Komponenttityypin valinta

Keskeisimpiä tehtäviä EJB-komponenttimallia hyödyntävässä ohjelmistonkehityksessä on valinta eri komponenttityyppien käyttökohteiden välillä. EJB-komponenttimallin määrityksen tarkoituksena on ollut esitellä kaikille kolmelle komponenttityypille järkevät ja todelliset käyttötarkoitukset.

Kohdepohjaisten komponenttien valintaan johtaa aina tarve esittää näkymä pysyvän tietovaraston tietoon. Lisäksi kohdepohjaiset komponentit mahdollistavat eri asiakassovellusten samanaikaisen kommunikoinnin saman komponentin ilmentymän kanssa. Samanaikaisessa kommunikoinnissa komponentin ei tule huolehtia tilasta eri asiakassovelluksissa vaan siitä, että tieto itse komponentissa on oikeellista eri asiakassovellusten kutsujen aikana. Myös kaiken sellaisen tiedon, jonka tulee säilyä useiden eri asiakkaiden elinkaaren ajan tai jonka tulee selviytyä esimerkiksi palvelinohjelmiston virheistä, pitää olla toteutettuna kohdepohjaisena komponenttina. (Sun 2000, 131-133.)

Istuntopohjaisia komponentteja käytetään vastaavasti aina tapauksissa, joissa halutaan toteuttaa liiketoimintalogiikkaa jonkin tietyn asiakassovelluksen puolesta. Kaikki kommunikointi on sidottu aina tiettyyn asiakassovellukseen, eikä yhtä komponenttia jaeta samanaikaisesti useiden eri asiakassovellusten kesken. Istuntopohjaiset komponentit eivät myöskään koskaan esitä näkymää pysyvään tietoon eivätkä näin ollen ole kykeneviä pysyvyydenhallintaankaan. (Sun 2000, 135-137.) Käytännössä istuntopohjaisten komponenttien voidaan katsoa edustavan sellaista toiminnallisuutta, joka voitaisiin toteuttaa myös asiakassovelluksessa ja joiden kehittäminen on yhtä yksinkertaista kuin vastaavan sovelluslogiikan toteuttaminen asiakassovelluksessakin.

Tilallisia istuntopohjaisia komponentteja käytetään ylläpitämään tilatietoa jonkin tietyn asiakassovelluksen puolesta. Kuitenkaan komponentin tilaa ei voida palauttaa ennalleen esimerkiksi jo kerran lopetetun istunnon jälkeen, jossa istuntopohjainen komponentti on

menettänyt tilatietonsa asiakassovelluksesta. (Sun 2000, 135-137; Roman 1999, 56-57.) Tilattomat istuntopohjaiset komponentit ovat anonyymeja yleiseen käyttöön tarkoitettuja komponentteja, jotka eivät säilytä tilatietoa minkään asiakassovelluksen puolesta. Sama komponentin ilmentymä voi siis palvella useita eri asiakkaita elinkaarensa aikana. Lisäksi tilattomien istuntopohjaisten komponenttien suorituskyky on usein parempi kuin tilallisten, sillä niiden ei tarvitse ylläpitää mitään tilatietoa asiakassovelluksistaan. Tosin tilattomien istuntopohjaisten komponenttien käyttäjän täytyy tarvittaessa huolehtia itse tilanhallinnasta, mikä vastaavasti lisää asiakassovelluksessa tarvittavan liiketoimintalogiikan määrää. (Sun 2000, 138-139.)

6.3.2 Liiketoimintametodit

Etärajapinnan esittelemät liiketoimintametodit ovat sellaisia, joille EJB-komponentin tulee tarjota toteutus. Käännösaikana mikään tarkistus ei varmista sitä, onko näin todella tapahtunut, ja näin jokin etärajapinnan esittelemä metodi on saattanut jäädä määrittelemättä komponentin toteutuksessa. Helpoin ratkaisu olisi toteuttaa etärajapinta komponentissa, sillä se pakottaisi komponentin toteuttamaan kaikki etärajapinnassa esitellyt metodit. Tämä onkin täysin mahdollista mutta tässä toimintatavassa on suuria riskejä.

Koska etärajapinta perii *javax.ejb.EJBObject*-rajapinnan, jouduttaisiin tässä rajapinnassa esitellyt metodit määrittelemään myös komponentin toteutukseen. Näillä metodeilla ei kuitenkaan olisi mitään käyttöä komponentissa. Vaarallisin seikka liittyy kuitenkin siihen, että mikäli etärajapinta perittäisiin komponentin toteutukseen, olisi olemassa mahdollisuus, että jokin toinen komponentti tai asiakassovellus voisi saada viitteen suoraan itse komponentin toteutukseen. Oikea tapa toimia välitettäessä viitettä komponenttiin on välittää EJB-olion viite komponentin suoran viitteen sijaan *SessionContext.getEJBObject*- tai *EntityContext.getEJBObject*-metodin avulla (Matena & Hapner 1999, 274).

Ratkaisuksi liiketoimintametodien pakolliseen toteuttamiseen etärajapinnan mukaisesti voidaan luoda ylimääräinen rajapinta, joka sisältää ainoastaan etärajapinnan esittelemät

liiketoimintametodit. Kun tämä pelkät liiketoimintametodit sisältämä rajapinta peritään komponentin toteuttamaan luokkaan, voidaan jo käännösaikana varmistua siitä, että kaikille liiketoimintametodeille on myös määritelty toteutus komponentissa. Edellä kuvatun toimintamallin ovat esittäneet ainakin Roman (1999), Valesky (1999) ja Sauer (2000), joskaan kukaan heistä ei ole idean alkuperäinen esittäjä.

Liiketoimintametodien suunnitteluun ja toteuttamiseen liittyy myös itse metodien mielekkyyden todentaminen. Liian suppeiden ja triviaalien metodien toteuttaminen saattaa olla turhaa, kun taas vastaavasti liian laajat ja paljon toiminnallisuutta sisältävät metodit saattavat vaikeuttaa komponentin siirrettävyyttä ja uudelleenkäyttöä. Lisäksi useiden triviaalien metodien kutsuminen yhden metodin sijaan aiheuttaa turhaa verkkoliikennettä (ks. 6.3.2).

6.3.3 Verkkoliikenne

Verkkoliikenteen vähentäminen on erittäin tärkeä tekijä pohdittaessa kokonaisen järjestelmän suorituskykyä, sillä jokaisen tietoverkon välityksellä tehdyn metodikutsun suoritus on hitaampaa kuin paikallisesti tehdyn kutsun. Verkkoliikenteen määrää on kuitenkin mahdollista karsia järkevällä suunnittelulla ja toteutuksella. Valesky (1999) ja Roman (1999) molemmat esittävät suunnitteluperiaatteen, jonka mukaisesti kohdepohjaisten komponenttien käyttöä tulisi kapseloida istuntokohtaisten komponenttien sisään. Tämän periaatteen mukaisesti jokaisen kohdepohjaisesta komponentista haetun tai sinne tallennetun tiedon ei tarvitse aiheuttaa uutta tietoverkon välityksellä toteutettavaa metodikutsua. Istuntopohjaisen komponentin metodille voidaan välittää tarvittavat tiedot, ja se voi toteuttaa liikennöinnin kohdepohjaisen komponentin kanssa itsenäisesti. Tällöin asiakkaan ja säiliön välillä tarvitaan ainoastaan yksi metodikutsu aikaisemmin tehtyjen useamman kutsun sijaan. Lisäksi Roman (1999) esittää tämän kapseloinnin tuovan mukanaan muun muassa sen edun, että asiakassovelluksen ei tarvitse toteuttaa minkäänlaista tapahtumanhallintaa, mikä yksinkertaistaa asiakassovelluksen toteutustyötä. Samalla Roman kuitenkin toteaa, että aina kohdepohjaisten komponenttien kapseloiminen istuntopohjaisten komponenttien sisään ei ole järkevää. Näin on tilanne esimerkiksi silloin, jos samalla sovelluspalvelimella toimivat servletit ja kohdepohjaiset komponentit

kommunikoivat keskenään. Koska verkkoliikennettä ei synny lainkaan, ei sitä voi vähentää. (Roman 1999, 402.) Lisäksi mikäli kommunikointi kohdepohjaisten komponenttien kanssa on vähäistä ja kohdepohjaisia komponentteja on vähän, on kohdepohjaisten komponenttien kapselointi turhaa (Sun 2000, 145-146). Edellä esitetyllä suunnitteluperiaatteella on läheinen yhteys myös Gamman ym. (1995) esittämän julkisivuajattelun kanssa.

Brown, Eskelin ja Pryce (1999) menevät verkkoliikennettä karsivassa ajattelussaan vielä pidemmälle ja esittävät mallin, joka perustuu toisteisiin komponentteihin. Tämän mallin mukaisesti kohdepohjaiset komponentit, joiden tietoja tarvitaan asiakassovelluksessa, siirretään sinne kokonaisuudessaan. Tämän siirrettävyyden toteuttaminen perustuu *java.io.Serializable*-rajapinnan toteuttamiseen, sillä RMI:n avulla voidaan siirtää mitä tahansa Java-olioita, kunhan edellä mainittu rajapinta on toteutettu. Tämän ajattelun heikkoutena voidaan pitää sitä, että kohdepohjaisten komponenttien muuttaminen aiheuttaa tarpeen päivittää tämän asiakassovelluksessa toisteisena olevan komponentin tiedot tietovarastoon. Brown, Eskelin ja Pryce (1999) esittävät myös joitain muita tämän mallin mukanaan tuomia ongelmia, mutta niitä ei käsitellä tässä yhteydessä niiden monimutkaisuuden ja suunnitteluperiaatteen äärimmäisyyden vuoksi.

Brownin ym (1999) esittämää mallia mukailee Sun Microsystemsin esittämä ajatus, jonka mukaisesti pelkästään sisältämiensä attribuuttiensa arvoja palauttavan EJB-komponentin toteuttaminen ei välttämättä ole mielekästä aiheutuvan ylimääräisen verkkoliikenteen vuoksi. Jos komponentin elinkaari on täysin riippuvainen toisesta oliosta ja sen attribuuttien arvoja ei voi muokata kuin olion avulla johon riippuvuussuhde on olemassa, voi olla järkevää luopua EJB-komponentin käytöstä kokonaan. EJB-komponentin sijaan voidaan käyttää tavallista Java-oliota, joka toteuttaa *java.io.Serializable*-rajapinnan ja joka näin ollen voidaan siirtää verkossa RMI:n avulla. Kun olio on siirretty, kaikki attribuuttien arvojen kyselyt toteutetaan paikallisen olion avulla. Vastaavasti mikäli tämän olion sisältämien attribuuttien arvoja halutaan muuttaa, tuhotaan koko olio ja luodaan uusi olio, joka sisältää uudet tiedot. Tämä uusi olio voidaan sitten välittää RMI:n avulla takaisin tallennettavaksi. (Sun 2000, 143-145.) Eri-

tyisesti tietojen päivityksen suhteen tämä toimintamalli poikkeaa olennaisesti Brownin ym. (1999) esittämästä.

Romanin (1999) näkemyksen mukaan kohdepohjaisten komponenttien *ejbLoad*-metodien tulisi käyttää hyväkseen niin sanottua laiskan lataamisen periaatetta (*lazy loading*). Mikäli kaikki kohdepohjaisen komponentin viittaamat toiset komponentit haetaan muistiin *ejbLoad*-metodissa, voi käydä niin, että osaa muistiin haetuista komponenteista ei koskaan tule käytetyksi. Tästä syystä voi olla järkevää luoda ylimääräinen komponentti kohdepohjaisen komponentin ja sen viittaamien komponenttien väliin, joka huolehtii kunkin viitatun komponentin hakemisesta muistiin vasta sitten kun kutakin komponenttia todella tarvitaan. Tämä kohdepohjaisen komponentin ja sen viittaamien komponenttien välissä toimiva komponentti pitää vain kirjaa kulloinkin muistiin haetuista komponenteista ja vastaisi tarvittaessa muistissa olemattoman komponentin hakemisesta. (Roman 1999, 498.)

6.3.4 Siirrettävyys

Valesky (1999) esittää periaatteen, jonka mukaisesti esimerkiksi kaikki sovelluspalvelimen vaatima valmistajakohtainen ohjelmakoodi kapseloitaisiin helpomman siirrettävyyden takaamiseksi (Valesky 1999, 185). Kapseloiminen helpottaa käyttöönottoa toisella sovelluspalvelimella, mutta mahdollistaa myös sen, että samankin sovelluspalvelimen valmistajan tekemät muutokset vaikuttavat vain mahdollisimman pieneen osaan komponentteja. Osittain tämänkin periaatteen voidaan katsoa noudattelevan Gamman ym. (1995) esittämää julkisivuajattelua.

Brown (2000) esittää mielenkiintoisen näkemyksen EJB-komponenttien muodostimien, *ejbCreate*-metodien parametreista. Riippuvuuksien vähentämiseksi olisi Brownin (2000) suotavaa, että nämä metodit hyväksyvät parametreinaan ainoastaan sellaisia arvoja, joita niillä on jäsenmuuttujinaan, tai sellaisia primitiivisiä esityksiä, joiden avulla EJB-komponentti voidaan luoda. Brownin (2000) mukaan tällainen primitiivinen esitys voi olla esimerkiksi XML-merkkijono, jonka sisältämän tiedon avulla EJB-komponentti luodaan. Brownin (2000) esittämä periaate kuulostaa varsin perustellulta, sillä turhien

viitteiden tallentaminen ja riippuvuuksien lisääminen vaikeuttaa komponenttien uudelleenkäyttöä.

Roman (1999) esittää liiketoimintalogiikan jakamisen kohdepohjaisten ja istuntopohjaisten komponenttien välillä olevan erittäin merkityksellinen tekijä, erityisesti mikäli komponenttien välisiä kytkentöjä halutaan vähentää siirrettävyyden takaamiseksi. Kohdepohjaisten komponenttien tulee sisältää mahdollisimman vähän liiketoimintalogiikkaa ja kommunikoida eri komponenttien kanssa ainoastaan pakottavan tarpeen niin vaatiessa. Vastaavasti istuntopohjaisten komponenttien tehtävänä on huolehtia tarpeellisen liiketoimintalogiikan toteuttamisesta ja käyttää kohdepohjaisia komponentteja ainoastaan tiedon hakuun, ei liiketoimintaan liittyvien tehtävien suorittamiseen. Kohdepohjaisten komponenttien kommunikointia muiden komponenttien kanssa voidaan pitää hyväksyttävänä ainoastaan silloin, kun se tapahtuu muiden kohdepohjaisten komponenttien kanssa ja on ehdottoman välttämätöntä. (Roman 1999, 499-500.)

6.3.5 Muita toteutusperiaatteita

Valesky (1999) esittää tapahtumien määrän olevan myös merkityksellinen tekijä järjestelmän suorituskykyä arvioitaessa. Tapahtumanhallinnan käyttäminen aiheuttaa aina jonkin verran ylimäärää, ja on järkevää pyrkiä käyttämään tapahtumanhallintaa vain silloin kun se todella on välttämätöntä. Erityisesti Valesky (1999) pitää järkevänä sellaisten toimintojen jättämistä tapahtumanhallinnan ulkopuolelle, jotka vain lukevat tietoa. Tällöin riski sellaisen tiedon lukemiselle, joka muuttuu jonkin toisen tapahtuman sisällä, luonnollisesti kasvaa mutta toisaalta säästetään tapahtumanhallinnan mukanaan tuoma ylimäärä. (Valesky 1999, 187.)

6.4 EJB-komponenttimallin soveltamisen keskeiset tehtävät

Seuraavissa alakohdissa esitettäviksi tehtäviksi on pyritty löytämään sellaisia EJB-komponenttimallin hyödyntämisessä oleellisia tehtäviä, joiden voidaan katsoa olevan

melko riippumattomia kohdealueesta ja käytettävistä ohjelmistoista. Seuraavassa on esitetty jako järjestelmän suunnittelussa ja toteutuksessa huomioitaviin seikkoihin, pääpainon ollessa toteutukseen liittyvissä tehtävissä.

6.4.1 Suunnittelun keskeiset tehtävät

Järjestelmän suunnittelussa keskeisin huomio tulee kiinnittää järkevään vastuiden jakoon komponenttien välillä. Komponentin tulee tehdä vain sellaisia tehtäviä, jotka sille luonnollisesti kuuluvat ja joiden suorittamiseksi sen ei tarvitse kommunikoida kohtuuttoman paljon muiden komponenttien kanssa. Toisaalta liian massiivisten ja paljon toimintaa sisältävien komponenttien suunnittelua tulee välttää muun muassa paremman uudelleenkäytön ja tehokkuuden takaamiseksi. Lisäksi julkisivukomponenttien käyttöä tulee suosia, mikäli suoria viitteitä useisiin komponentteihin muuten jouduttaisiin ylläpitämään esimerkiksi asiakassovelluksessa.

Komponenttityypin valinta kolmen eri EJB-komponenttimallin tukeman komponenttityypin kesken on myös keskeinen vaikutin järjestelmän järkevyyttä ja suorituskykyä arvioitaessa. Komponenttityyppi tulee valita sen käyttötarkoituksen mukaan huomioiden muun muassa asiakassovelluksen mahdollisuudet suorittaa esimerkiksi tilanhallintaa tarvittaessa komponenttien puolesta.

Liiketoimintalogiikan siirtäminen pois asiakassovelluksesta täytyy myös huomioida suunnittelussa. Tasapaino asiakassovelluksen ja EJB-komponenttien sisältämän liiketoimintalogiikan välillä voidaan saavuttaa ainakin osittain tutkimalla järjestelmässä esiintyvää verkkoliikennettä. Mikäli kaikki asiakassovelluksen toiminta aiheuttaa kutsuja sovelluspalvelimelle, voi suorituskyky heiketä. Toisaalta, mitä enemmän liiketoimintalogiikkaa siirretään asiakassovellukseen, sitä tiukemmaksi kytkentä asiakassovelluksen ja komponenttien välillä tulee. Pienetkin muutokset liiketoimintalogiikassa on myös vaikeampaa toteuttaa päivitysten avulla jokaiseen asiakassovellukseen sen sijaan, että voitaisiin vain muuttaa esimerkiksi yhden komponentin ominaisuuksia.

Tämän tutkielman puitteissa ei oteta tarkemmin kantaa asiakassovelluksen tai kokonaisen järjestelmän suunnitteluun vaan tarkoituksena on keskittyä kuvaamaan niitä toimenpiteitä, joilla säiliössä toimivien komponenttien suunnittelu ja erityisesti toteutus voidaan tehdä järkevällä tavalla. Tämän vuoksi edellä ja kappaleessa 6.1 esitetyt esimerkit komponenttien ja järjestelmän suunnitteluperiaatteista katsotaan riittäviksi tämän tutkielman tarkoituksiin.

6.4.2 Toteutuksen keskeiset tehtävät

Kohdepohjaisten komponenttien ominaispiirre on niihin kohdistuvien muutostarpeiden vähäisyys, sillä niissä ei juurikaan ole liiketoimintalogiikkaa joka muuttuisi. Vastaavasti istuntopohjaisia komponentteja joudutaan muokkaamaan mahdollisesti ainakin ominaisuustiedoston avulla tarpeen mukaan, vaikka lähdekooditasolla tehtäviä muutoksia ei toteutettaisi lainkaan. Uudelleenkäytön mahdollisuudet ovat siis käytännössä kohdepohjaisilla komponenteilla paremmat niihin kohdistuvien muutostarpeiden vähäisyyden vuoksi (Roman 1999, 56).

Molempien komponenttityyppien tapauksessa on järkevää aloittaa toteutus etärajapinnan toteuttamisella, sillä sen toteuttamisen yhteydessä esiteltävät metodit määräävät kaikki ne liiketoimintametodit, jotka komponentin on toteutettava. Mikäli halutaan varmistua siitä, että kaikki esitellyt metodit varmasti saavat toteutuksen on syytä käyttää kohdassa 6.3.1 esitettyä tapaa luoda ylimääräinen rajapinta, joka pakottaa ohjelmakoodia käännettäessä toteutuksen kaikille etärajapinnassa esitellyille metodeille. Lisäksi etärajapintaa toteutettaessa on syytä varmistua sen sisältämien liiketoimintametodien järkevyydestä. Triviaalit hakumetodit on mahdollisuuksien mukaan kapseloitava yhden metodin sisään verkkoliikenteen vähentämiseksi.

Etärajapinnan toteuttamisen jälkeen luonnollinen askel on toteuttaa itse komponentin ohjelmakoodi, jossa kaikki etärajapinnan esittelemät metodit saavat varsinaisen toteutuksen. Hyvän suunnittelun pohjalta tämän vaiheen tulisi olla melko suoraviivainen toimenpide eikä muutoksia komponenttien vastuujakoihin tai niiden määrään tulisi enää tehdä. Kohdepohjaisten komponenttien tapauksessa on toteutettava ainakin yksi

komponenttien etsintään tarkoitettu metodi ja vastaavasti myös luontimetodien määrä vaihtelee komponenttityypistä riippuen. *ejbCreate*-metodeille välitettäviä parametreja on myös syytä tarkkailla. Mahdollisuuksien mukaan kannattaa noudattaa Brownin (2000) esittämää tapaa käyttää vain sellaisia parametreja, joita komponentilla on jäsenmuuttujinaan, tai sellaisia primitiivisiä esityksiä, joiden avulla komponentti voidaan luoda. Lisäksi kaiken valmistajakohtaisen ohjelmakoodin kapselointi on järkevää suorittaa komponentin toteutusvaiheessa, etenkin jos komponentille halutaan taata hyvät uudelleenkäytön mahdollisuudet. Liiketoimintalogiikan toteuttamista on myös syytä tarkkailla, jotta kohdepohjaisiin komponentteihin ei tarpeettomasti ohjelmoida siirrettävyyttä ja uudelleenkäyttöä vaikeuttavaa liiketoimintalogiikkaa.

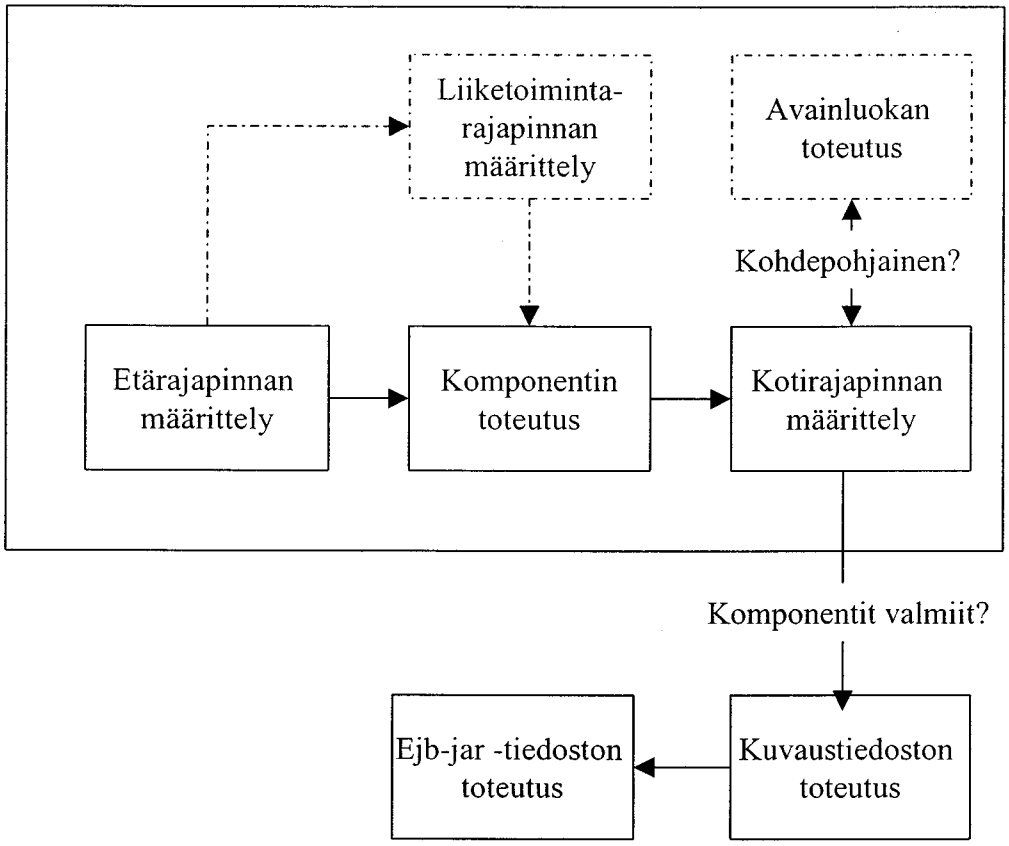
Kotirajapinnan luonti on tehtävä, joka on luonnollista suorittaa seuraavaksi. Kotirajapintaan kerätään kaikki komponentin luomiseen liittyvät metodit, jotka komponentin ohjelmakoodissa on toteutettu. Lisäksi kohdepohjaisten komponenttien kotirajapinnassa esitellään komponentin toteuttamat etsintämetodit. Kotirajapinnan luominen on melko suoraviivainen tehtävä, sillä kaikki sen sisältämät luontimetodit ovat jo saaneet varsinaisen toteutuksensa itse komponentin ohjelmakoodissa ja kohdepohjaisten komponenttien tapauksessa myös kaikki etsintämetodit on jo toteutettu.

Kohdepohjaisille komponenteille on määriteltävä kolmen edellä esitetyn luokan lisäksi erillinen avainluokka. Avainluokan toteuttamisessa keskeinen tehtävä on löytää ainakin yksi sellainen komponentin julkinen jäsenmuuttuja, jonka avulla jokainen komponentin ilmentymä voidaan erottaa muista saman komponentin ilmentymistä.

Kun kaikki edellä esitetyt luokat on toteutettu, tulee komponentille luoda kuvaustiedosto. Kuvaustiedoston luominen on suoraviivainen prosessi, jossa määritellään komponentin tarvitsemat palvelut ja komponentin säiliöltä odottama toiminnallisuus. Kuvaustiedoston muoto luonnollisesti vaihtelee eri komponenttien mukaan, mutta sen sisältö on pääpiirteissään samankaltainen kaikilla komponenteilla. EJB-komponenttimallin määrityksen version 1.1 mukaisesti jokaiselle komponentille ei enää tarvitse luoda omaa kuvaustiedostoa vaan jokaista ejb-jar -tiedostoa ja sen sisältämiä komponentteja kohti luodaan vain yksi kuvaustiedosto. Kuvaustiedoston luo-

misen jälkeen yksi tai useampia komponentteja paketoidaan levityskelpoiseksi kokonaisuudeksi ejb-jar -tiedostoon.

Kuviossa 9 on esitetty EJB-komponenttien kehittämisen keskeiset tehtävät. Katkoviivoituksella merkityillä tehtävillä ja siirtymillä niiden välillä kuvataan tehtävien vaihtoehtoisuutta komponenttityypistä ja valitusta toteutustavasta riippuen. Lisäksi kehyksen sisään on ositettu ne tehtävät, joita voidaan tehdä useampia kertoja, mikäli komponentteja tuotetaan levitettäväksi samalla kertaa useampia.



KUVIO 9 EJB-komponenttien toteuttamisen keskeiset tehtävät

Kuvion 9 esittämän tehtäväjaon mukaisesti kaikki kehyksen sisällä olevat tehtävät ovat sellaisia, joiden automatisointi ei ole mahdollista, sillä luotavat tiedostot sisältävät liiketoimintalogiikkaan ja komponenttien sisäiseen toimintaan liittyvää tietoa, joiden päättteleminen ei ole mahdollista. Kehyksen ulkopuolella olevat tehtävät, kuvaustiedoston- ja ejb-jar –tiedoston toteutus ovat vastaavasti sellaisia, jotka todellisuudessa automatisoidaan kulloinkin käytössä olevien komponenttien käyttöönottovälineiden avulla.

Esimerkiksi Sun Microsystemsin J2EE-ohjelmointialustan toteutuksessa on mukana *deploytool*-sovellus, joka avulla komponentteja voidaan koostaa. Tämä sovellus automatisoi täysin kuvaustiedoston ja ejb-jar –tiedoston tuottamisen.

6.5 Esimerkkisovellus

Seuraavissa alakohdissa on esitetty suppeahko esimerkkisovellus, jossa on pyritty huomioimaan tutkielmassa aikaisemmin esitettyjä huomioita EJB-komponenttien suunnittelusta ja toteuttamisesta. Esimerkin tarkoituksena on esittää sovellettuna joitain tässä luvussa aiemmin esitettyjä periaatteita, ja esitetyn esimerkin sisältämät komponenttien ohjelmakoodit ovat sellaisenaan suorituskelpoisia.

Esimerkin käyttöönottaminen sovelluspalvelimella on kuitenkin tehtävä, joka vaatii parametrien asettamista kulloisenkin suoritusympäristön mukaisesti, ja tämän vuoksi esimerkissä esitettyjen ohjelmakoodien lisäksi näitä eri parametreja ei ole käsitelty. Muun muassa kuvaustiedoston luominen suoritetaan usein automaattisesti käyttöönoton yhteydessä, joten sen sisältämiin tietoihin ei oteta kantaa. Myöskään kohdepohjaisen komponentin vaatimien tietokantaan liittyvien määritysten esittäminen jätetään tekemättä, sillä tämän tutkimuksen puitteissa ei ole mahdollista toteuttaa täysipainoista komponenttien suoritusympäristöä.

6.5.1 Vaatimusmäärittely ja suunnittelu

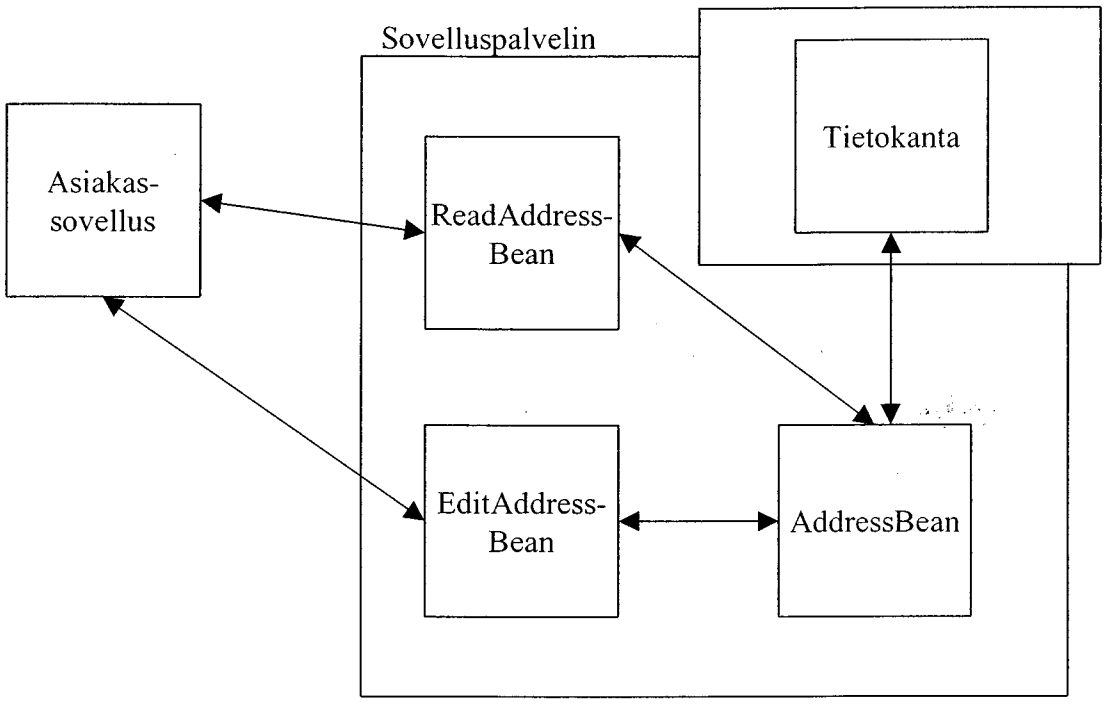
Toteutettavan esimerkkisovelluksen tarkoituksena on toteuttaa yksinkertainen osoitehakemisto, josta henkilöitä voidaan hakea nimen perusteella ja jossa jo olevien henkilöiden tietoja voidaan muuttaa. Asiakassovelluksen on lisäksi sisällettävä mahdollisimman vähän liiketoimintalogiikkaa ja sitä on voitava käyttää fyysiseltä sijainniltaan eri paikoista.

Vaatimusmäärittelyn mukaisesti sovelluksen käsittelemät tiedot on tallennettava johonkin pysyvään tietovarastoon. Lisäksi, koska sovellusta on voitava käyttää useasta

paikasta, on tiedot tallennettava jonnekin jaettuun tietovarastoon, jotta kaikki tieto on aina saatavilla sovelluksen käyttäjän sijainnista riippumatta.

Sovelluksessa on oltava jokin pysyvää tietoa esittävä komponentti, joka edustaa aina kulloinkin haluttua osoitetietoa. Tällaiseksi komponentiksi sopii hyvin kohdepohjainen komponentti, joka edustaa näkymää tietovarastoon, johon varsinaiset osoitetiedot on tallennettu. Verkkoliikennettä karsivan ajattelun mukaisesti silloin, kun tietoja ainoastaan luetaan eikä muuteta, voidaan kaikki tiedot siirtää asiakassovellukselle, jotta vältytään turhalta verkkoliikenteeltä. Tämä voidaan toteuttaa helposti julkisivukomponentin avulla, jonka kautta tietoja ei ole mahdollista muuttaa. Vastaavasti tietojen muuttamista varten voidaan luoda toinen julkisivukomponentti, joka käsittelee tarvittavat muutokset ja toteuttaa ne kohdepohjaisen komponentin avustuksella tietovarastoon. Asiakassovellus ei siis koskaan ole tekemisissä itse tietoa esittävän kohdepohjaisen komponentin kanssa.

Järjestelmän arkkitehtuuri ja eri komponentit on esitetty kuviossa 10. *ReadAddressBean*-komponentti on tilaton istuntopohjainen komponentti, joka tarjoaa metodit tietyn osoitetiedon hakemiseksi. Tilallinen istuntopohjainen *EditAddressBean*-komponentti vastaa osoitetietojen päivityksistä ja kohdepohjainen *AddressBean*-komponentti esittää näkymää tietovarastoon ja huolehtii pyyntöjen mukaisesti tietovaraston päivittämisestä.



KUVIO 10 Esimerkkijärjestelmän arkkitehtuuri

6.5.2 Toteutus

Toteutus suoritetaan, kuten kohdassa 6.4.2 esitettiin. Taulukossa 6 on esitetty kaikki esimerkkijärjestelmän komponentit ja niiden tarvitsemat luokat. Kaikkien komponenttien yhteydessä on käytetty hyväksi liiketoimintarajapinnan periaatetta ja nämä rajapinnat on nimetty komponenttiluokan mukaisesti, minkä lisäksi tarkentavana päätteenä on käytetty *BusinessInterface*-sanaa.

ReadAddress-komponenttia käytettäessä halutaan vain lukea jonkin henkilön osoitetieto, joten muokattavuus ei ole tärkeää. Tämän vuoksi *ReadAddress*-komponentin hakumetodit palauttavat erillisen *AddressObject*-olion, joka sisältää kaikki osoitteeseen liittyvät tiedot erillisinä merkkijonoina. Näin toimittaessa jokaista erillistä osoitetietoa varten ei tarvitse tehdä kutsua verkon välityksellä. Vastaavasti *EditAddress*-komponentin tapauksessa henkilön osoitetietoja halutaan muokata, joten on huolehdittava tapahtumanhallinnasta ja kohdepohjaisen komponentin sisältämän informaation tallentamisesta tietovarastoon. Tällöin ei tule kyseeseen *ReadAddress*-

komponentin kaltainen *AddressObject*-komponentin avulla toteutettava toiminnallisuus, sillä komponentin tietojen tahdistaminen muutosten jälkeen olisi vaikeaa ja useat asiakassovellukset voisivat samanaikaisella toiminnallaan aiheuttaa ongelmia tietovaraston eheydessä.

TAULUKKO 6 Esimerkkisovelluksen komponentit

Komponentti	**Kuvaus**
ReadAddressBean	Tilaton istuntopohjaisen osoitteenhakukomponentin toteutus
ReadAddressHome	ReadAddressBeanin kotirajapinta
ReadAddress	ReadAddressBeanin etärajapinta
ReadAddressBusinessInterface	ReadAddressBeanin liiketoimintametodien esittelyt sisältävä rajapinta
EditAddressBean	Tilallinen istuntopohjaisen osoitteenmuokkauskomponentin toteutus
EditAddressHome	EditAddressBeanin kotirajapinta
EditAddress	EditAddressBeanin etärajapinta
EditAddressBusinessInterface	EditAddressBeanin liiketoimintametodien esittelyt sisältävä rajapinta
AddressBean	Kohdepohjaisen osoitekomponentin toteutus
AddressBusinessInterface	AddressBeanin liiketoimintametodien esittelyt sisältävä rajapinta
AddressHome	AddressBeanin kotirajapinta
Address	AddressBeanin etärajapinta
AddressPK	AddressBeanin avainluokka
AddressObject	Luokka jota käytetään paluuarvona haettaessa jonkin henkilön osoitetiedot vain lukemista varten.

Taulukossa 6 esitetyt esimerkkisovelluksen rajapinnat ja luokat on esitelty kokonaisuudessaan liitteissä seuraavasti: Kohdepohjaisen *AddressBean*-komponentin ja sen tarvitsemien luokkien ohjelmakoodi on esitetty liitteessä 1. Istuntopohjaisten *ReadAddressBean*- ja Edit*AddressBean*-komponenttien ohjelmakoodit on vastaavasti esitetty

liitteissä 2 ja 3. Liitteessä 4 on lisäksi esitetty yksinkertaisen asiakassovelluksen ohjelmakoodi sekä sen lisäksi muutama SQL-lause, joiden avulla komponenttien tarvitseman tietokantataulun luominen tietokantaan onnistuu. Liitteessä 4 esitetty asiakassovellus on täysin puutteellinen siinä mielessä, että se ei sisällä mitään mahdollisuutta käyttäjän kommunikointiin järjestelmän kanssa. Esimerkin asiakassovellus ainoastaan käyttää hyväkseen tietyn ennalta määrätyn periaatteen mukaisesti järjestelmässä olevia komponentteja.
Liitteissä esitettyjä ohjelmakoodeja on kommentoitu jonkin verran ymmärrettävyyden parantamiseksi. Kaikkia metodeja tai muuttujia ei kuitenkaan ole kommentoitu vaan lähinnä kommentteja on lisätty sellaisiin kohtiin, joiden merkitys ei EJB-komponenttimallin määrityksen tai asiayhteyden vuoksi muuten ole selvä.

6.5.3 Huomioita esimerkkisovelluksesta

Kuten aiemmin todettiin, tämän tutkimuksen tarkoituksena ei ole esittää toimintamallia sille, kuinka edellä esitetty esimerkkisovellus otetaan käyttöön sovelluspalvelimella. Käyttöönoton yhteydessä esimerkkisovelluksen komponenteille tulisi antaa ainakin tapahtumanhallintaan liittyviä parametreja sekä antaa tietoa käytössä olevasta tietovarastosta. Lisäksi käytettävä sovelluspalvelin vaatii omat asetuksensa, jotka ovat täysin riippuvaisia käytettävästä tuotteesta.

Tietovaraston toteuttaminen tietokantana vaatii paljon aikaa, ja tämän tutkielman puitteissa ei liitteessä 4 esitettyä laajemmin oteta kantaa esimerkiksi siihen millaisia tietokannan sisältämät taulut ovat, miten ne luodaan tai miten jotain tiettyä tietokantatuotetta tulisi käyttää. Tämän vuoksi ei myöskään ole mielekästä esittää eri parametrien arvoja, sillä niiden arvot voidaan päättää osittain vasta käyttöönoton yhteydessä. Kaikki esimerkissä esiintyvä ohjelmakoodi on kuitenkin käännetty Javan luokkatiedostoiksi ja ne ovat tässä mielessä syntaktisesti oikeellisia.

Esimerkkiä voidaan pitää myös toiminnallisuutensa puolesta puutteellisena, sillä edellä esitettyjen kaltaisilla komponenteilla toteutettu järjestelmä ei toteuta kaikkia osoitehakemistolle asetettuja vaatimuksia. Esimerkiksi tietojen lisäys järjestelmään tu-

lee esimerkkijärjestelmässä tehdä jonkin ulkopuolisen sovelluksen avulla ja tietojen haku onnistuu vain henkilön nimen perusteella. Lisäksi asiakassovelluksen ja komponenttien toteuttama poikkeustilanteiden hallinta on täysin puutteellista, sillä poikkeustilanteissa sovellus esimerkin kaltaisella toteutuksella yksinkertaisesti vain lopettaa toimintansa.

6.6 EJB-komponenttimalliin pohjautuvan ohjelmistonkehityksen tulevaisuus

EJB-komponenttimallin määritys on kehittynyt nykyiseen versioon 1.1 mennessä huomattavasti, ja useat puutteet on jo korjattu nykyisin saatavilla olevassa määrityksessä. Tämän tutkielman tarkoituksena ei ole esitellä eri versioissa olevia eroavaisuuksia tai pohtia tulevaisuudessa mahdollisesti määritykseen lisättäviä ominaisuuksia. Uudelle määritykselle ominaiseen tapaan EJB:stä ja sen käyttökelpoisuudesta tehtävät arviot vaihtelevat suuresti. Seuraavassa on esitetty joitain huomioita EJB:n käytöstä ja tulevaisuudesta.

Teknisessä mielessä EJB-komponenttimallia vertaillaan useimmiten Microsoftin COM/DCOM –ratkaisuihin (ks. esim. Roman & Öberg 1999b). Täysin puolueettoman teknisen vertailun tekeminen on kuitenkin vaikeaa, ja käytettävä ratkaisu riippuu usein muistakin tekijöistä kuin pelkästään teknisestä paremmuudesta. Tämän tutkielman puitteissa ei oteta kantaa EJB-komponenttimallin ja sen kanssa kilpailevien teknologioiden paremmuuteen vaan pikemminkin esitetään huomioita EJB-komponenttimallin käytöstä.

Zona Researchin tekemän tapaustutkimuksen mukaan EJB:n avulla saatavat liiketoiminnalliset hyödyt piilevät muun muassa lyhentyneessä järjestelmien kehitysajassa, ohjelmointihenkilöstön vähäisemmässä tarpeessa, ohjelmakoodin paremmassa hallinnassa ja uudelleenkäytettävyydessä eri laitteistoalustoilla (Zona Research 1999). Roman ja Öberg (1999a) vastaavasti esittävät EJB:n eduiksi muun muassa kustannustehokkuuden sekä sen, että EJB ei pakota sitoutumaan yhteen toimittajaan tai laitteistoalustaan.

Sun Microsystemsin periaatteiden mukaisesti J2EE-ohjelmointialustan määritys ja siihen kuuluvan EJB-komponenttimallin määritykset ovat jatkuvan kehityksen alaisena ja

vapaasti kenen tahansa kommentoitavissa. Niinpä uusien ominaisuuksien lisääminen on usein seurausta informaatioteollisuuden esittämistä tarpeista. Tämän avoimuuden vuoksi EJB-komponenttimallin tulevaisuus näyttää melko valoisalta. Lisäksi useat kaupalliset sovelluspalvelinten valmistajat ovat jo lisänneet tuotteisiinsa tuen uusimpien määrityksien mukaiselle komponenttien suoritusympäristölle.

Java-ohjelmointikielellä toteutetuille järjestelmille on usein esitetty ongelmaksi suorituskyky. Palvelimella toimivat EJB-komponentit eivät kuitenkaan välttämättä vaadi suurta suoritusnopeutta, sillä todennäköisin ongelma ja hidastava tekijä näissä järjestelmissä on verkkoliikenteen pullonkaulat, ei niinkään Java-ohjelmointikielen suorituskyky.

6.7 Yhteenveto

Tämän luvun tarkoituksena oli esittää joitain yleisiä komponenttien toteutusperiaatteita sekä joitain EJB-komponenteille ominaisia toteutusperiaatteita. Lisäksi esitettiin J2EE-ohjelmointialustan ja EJB-komponenttimallin asettamia vaatimuksia, jotka on huomioitava järjestelmiä kehitettäessä. Komponenttien yleisiä toteutusperiaatteita pyrittiin löytämään kirjallisuudesta, huomioiden aikaisemmissa luvuissa esitetyt huomiot komponenteista ja niiden määritelmistä. Tällaisia melko yleiseksi laskettavia komponenttien toteutusperiaatteita olivat muun muassa komponenttien välisien kytkentöjen vähentäminen, rajapintojen eriyttäminen ja käyttöliittymän erottaminen liiketoimintalogiikasta.

EJB-komponenttimallille ominaisia toteutus- ja suunnitteluperiaatteita etsittiin kirjallisuudesta ja Sun Microsystemsin määrityksistä pyrkien löytämään sellaisia yleisiä periaatteita, joiden käyttö olisi mielekästä sovellusalueesta riippumatta. Tällaisia EJB-komponenttimallikohtaisia suunnittelu- ja toteutusperiaatteita olivat muun muassa erillisen liiketoimintametodit sisältävän rajapinnan toteuttaminen, verkkoliikenteen vähentäminen sekä siirrettävyyden turvaaminen.

Toteutus- ja suunnitteluperiaatteiden työstämisen jälkeen luvussa esiteltiin jako suunnittelu- ja toteutusvaiheen tehtäviin, pääpainon ollessa toteutusvaiheen tehtävissä. Luvun lopuksi esitettiin yksinkertainen esimerkki, jossa pyrittiin huomioimaan osa esitetyistä periaatteista sekä kartoitettiin EJB-komponenttimallin tulevaisuutta.

7 YHTEENVETO

Tämän tutkielman tarkoituksena on ollut selvittää, kuinka EJB-komponenttimallia voidaan hyödyntää komponenttipohjaisessa ohjelmistonkehityksessä. EJB-komponenttimallin määrityksen uutuudesta johtuen aikaa on uhrattu runsaasti komponenttimallin ja J2EE-ohjelmointialustan esittelyyn varsinaisen EJB-komponenttimallin mukaisen ohjelmistonkehityksen käsittelyn lisäksi.

Perusta monitasoiselle komponenttipohjaiselle ohjelmistonkehitykselle on luotu luvussa 2 esittelemällä perusteet monitasoarkkitehtuureista, hyvine ja huonoine puolineen. Monitasoarkkitehtuurien käsittelyssä pääpaino on ollut perusasioiden esittelyssä, ei niinkään yksityiskohtaisten ja tapauskohtaisten toteutusratkaisujen arvioinnissa. Monitasoarkkitehtuurien käsittelyn pohjalta kolmi- tai useampitasoisten ratkaisujen voidaan katsoa edustavan niitä periaatteita, joita soveltamalla erityisesti suuret ja laajennettavuutta tarvitsevat järjestelmät tullaan toteuttamaan.

Monitasoarkkitehtuurien käsittelyn jälkeen luvussa 3 on käsitelty komponenttipohjaista ohjelmistonkehitystä kokonaisuutena. Erityisesti tässä luvussa on paneuduttu kirjallisuudessa esiintyneiden komponenttien määritelmien esittämiseen ja analysointiin. Kirjallisuudessa esitetyt määritelmät on jaettu kahteen tasoon niiden abstraktioasteen mukaan, minkä lisäksi liiketoimintakomponentista käsitteenä on esitetty joitain huomioita. Näiden kirjallisuudessa esitettyjen määritelmien pohjalta on johdettu tässä tutkielmassa käytetty määritelmä komponentille, jonka mukaisesti komponentin katsotaan olevan itsenäinen, suorituskelpoinen yksikkö, jonka tarjoamat palvelut ja ulkoiset riippuvuudet on täsmällisesti määritelty. Lisäksi komponentti tulee voida korvata toisella samat palvelut toteuttavalla komponentilla ja sen yhdistäminen muihin komponentteihin tulee olla mahdollista. Komponentin oletetaan myös olevan liiketoimintakomponentti siinä mielessä, että se on jonkin itsenäisen liiketoimintakäsitteen tai prosessin kuvaus.

Luvussa 3 on lisäksi esitetty määritelmä komponenttimallille ja lyhyet kuvaukset nykyisin laajemmassa käytössä olevista JavaBeans-, COM/DCOM- ja CORBA-komponenttimalleista. Luvun 3 lopuksi on esitetty huomioita komponenteista ohjelmistonkehityksessä, niiden luonteesta uudelleenkäytettävinä ohjelmiston osina, niiden vaatimasta tuesta ohjelmistonkehityksessä sekä komponenttimarkkinoista. Komponenteista ja niiden hyötykäytöstä on olemassa runsaasti erilaisia käsityksiä. Yhtä ainoaa ja oikeaa tapaa hyödyntää komponentteja tuskin on olemassa, mutta valitun hyödyntämistavan mukaisesti on valittava oikein ne toimenpiteet, jotka komponentteja ohjelmistonkehityksessä hyödynnettäessä on tehtävä.

Luvussa 4 on pureuduttu tarkemmin erääseen tämän tutkielman kannalta keskeisimmistä osista, J2EE-ohjelmointialustaan. J2EE-ohjelmointialustan käsittely on aloitettu esittelemällä lyhyesti taustaa Java-ohjelmointikielestä ja Javan eri ohjelmointialustoista. Luvun tärkeimmän osan muodostaa kuitenkin J2EE:n sisältämien sovellusliittymien esittely, joista neljä EJB-komponenttimallin kannalta tärkeintä on esitelty yksityiskohtaisemmin. Yksityiskohtaisemmin esiteltäväksi sovellusliittymiksi valittiin nimipalvelut tarjoava JNDI, tietovarastojen kanssa kommunikointia helpottava JDBC sekä hajautuksen määrittelevä RMI ja tapahtumanhallinnan määrittelevät JTA ja JTS. Luvussa on lisäksi esitelty komponenttien suoritusympäristöjen, säiliöiden, perusasiat ja esitetty näiden eri säiliöiden toteuttama toiminnallisuus.

Luvussa 4 esitelty J2EE-ohjelmointialusta muodosti vahvan pohjan luvussa 5 tarkemmin esitellylle EJB-komponenttimallille. Luvun 5 aluksi on esitelty EJB-komponenttimallin soveltamiseen liittyvät roolit ja joitain huomioita säiliöistä komponenttien suoritusympäristönä. Eri komponenttityypit sekä niiden toteuttama tilan- ja pysyvyydenhallinta on esitetty luvussa seuraavaksi esimerkkejä hyväksi käyttäen. Kohde- ja istuntopohjaisten komponenttien ja niiden ominaisuuksien esittelyn jälkeen on esitetty komponentti kokonaisuutena kaikkine tarvittavine osineen. Nämä jokaisen EJB-komponentin tarvitsemat osat itse komponentin ohjelmakoodin lisäksi ovat koti- ja etärajapinta avainluokka sekä kuvaustiedosto. Kaikille komponenteille voidaan myös tuottaa ominaisuustiedosto, jonka avulla komponenttien ominaisuuksien muokkaaminen helpottuu. Lisäksi kohdepohjaiselle komponentille tulee aina toteuttaa avainluokka. Luvun lopuksi

on esitetty joitain huomioita komponentin ja suoritusympäristön yhteistoiminnasta sekä komponenttien käyttämästä tapahtumanhallinnasta, joka EJB-komponenttimallin mukaisesti voidaan toteuttaa joko automaattisesti säiliön toimesta tai komponentti itse voi huolehtia kaikista tapahtumanhallintaan liittyvistä toimenpiteistä.

Tutkielman luvussa 6 on esitetty huomioita EJB-komponenttimalliin pohjautuvasta ohjelmistonkehityksestä. EJB-komponenttimalliin pohjautuvaa ohjelmistonkehitystä on lähestytty tutkimalla J2EE-ohjelmointialustan ja EJB-komponenttimallin asettamia rajoitteita, sillä niiden olemassaoloon ei ohjelmistonkehittäjä voi vaikuttaa. Seuraavaksi on pyritty löytämään yleisiä komponenttien suunnittelu- ja toteutusperiaatteita, joiden soveltaminen myös EJB-komponenttimalliin pohjautuvaa ohjelmistonkehitystä toteutettaessa on mahdollista. Yleisten suunnittelu- ja toteutusperiaatteiden esittelyn jälkeen on esitetty joitain vain EJB-komponenttimallille soveltuvia suunnittelu- ja toteutusperiaatteita. Edellä esitettyjen periaatteiden ja EJB-komponenttimallin määrityksen pohjalta on johdettu ne keskeiset tehtävät, jotka EJB-komponenttimallia hyödynnettäessä tulee tehdä. Luvussa on lisäksi esitetty yksinkertainen esimerkkisovellus, jossa on pyritty huomioimaan osa esitetyistä suunnittelu- ja toteutusperiaatteista. Esimerkki on kuitenkin melko puutteellinen esitys siitä, mitä todellisuudessa tulisi ottaa huomioon EJB-komponenttimallia sovellettaessa.

Tämän tutkimusotteeltaan käsitteellis-teoreettisen tutkimuksen keskeisin tuotos on edellä esitettyyn perustuen selkeä määritys ohjelmistokomponentille ja komponenttimallille sekä esittely J2EE-ohjelmointialustasta ja EJB-komponenttimallista käytännön soveltamisohjeineen. Tutkimuksen ulkopuolelle on rajattu useita tekijöitä, jotka vaikuttavat EJB-komponenttimallin menestyksekkääseen soveltamiseen. Tällaisia tekijöitä ovat muun muassa käytettävän sovelluspalvelimen valinta, erilaiset komponenttien suunnitteluun ja toteutukseen liittyvät vaiheet, välineet ja menetelmät sekä valinta eri teknologioiden välillä järjestelmää suunniteltaessa. Lisäksi tutkimuksen puutteeksi voidaan lukea se, että yhtään EJB-komponenttimallin mukaista toteutusta ei ole koeteltu todellisessa komponenttien suoritusympäristössä.

Jatkotutkimusaiheista hedelmällisimpiä olisivat sellaiset, joissa tutkittaisiin komponenttien siirrettävyyttä eri sovelluspalvelimille, komponenttien tehokkuutta sekä rakennettujen järjestelmien käytön aikaista luotettavuutta ja ylläpitotoimenpiteiden helppoutta. Lisäksi EJB-komponenttimalliin pohjautuvien järjestelmien suunnitteluun keskittyvästä tutkimuksesta olisi varmasti hyötyä.

LÄHDELUETTELO

Arnold, K. & Gosling, J. 1997. The Java™ Programming Language. 4. Painos. Reading, MA: Addison-Wesley.

Bergner, K., Rausch, A., Sihling, M. & Vilbig, A. 1999. Componentware – Methodology and Process. Second International Workshop on CBSE in conjunction with ICSE99. 17.-18.5.1999. Los Angeles, Yhdysvallat.

Booch, G. 1987. Software Components with ADA: Structured, Tools and Subsystems. 2. painos. The Benjamin/Cummings Publishing Company.

Brown, A. W. 1998. From Component Infrastructure To Component-Based Development. International Workshop on CBSE in conjunction with ICSE98. 25.-26.4. 1998. Kyoto, Japani.

Brown, A. W. & Wallnau, K. C. 1998. The Current State of CBSE. IEEE Software 15(5), 37-46.

Brown, K. 1999. A Mini-pattern language for Distributed Component Design. Conference on Pattern Languages of Programming. 15.-18.8.1999. Monticello, Yhdysvallat.

Brown, K. 2000. Limit Parameters For EJB Creates. Saatavilla www-muodossa <URL:http://www.c2.com/cgi/wiki?LimitParametersForEjbCreates. 8.3.2000.

Cheung, S. 1999. Java™ Transaction Service (JTS) version 1.0. Sun Microsystems Inc.

Cheung, S. & Matena, V. 1999. Java™ Transaction API (JTA) version 1.01. Sun Microsystems Inc.

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

Edwards, J. 1997. 3-Tier Client-Server At Work. John Wiley & Sons.

Goldfarb, C. F. & Prescod, P. 1998. The XML Handbook. Prentice Hall.

Elmasri, R. & Navathe, S. B. 1994. Fundamentals of Database Systems. 2. painos. Redwood city, CA: The Benjamin/Cummings Publishing Company, Inc.

Haikala, I. & Märijärvi, J. 1998. Ohjelmistotuotanto. 6. painos. Helsinki: Suomen Atk-kustannus oy.

Hill, J. & Salo, T. 1999. Persistence in Enterprise JavaBeans Applications. Journal of Object-Oriented Programming 12(1), 6-10.

Horstmann, C. F. & Cornell, G. 1998. Core Java 1.1 Volume 2 - Advanced Features. Sun Microsystems Press.

Jaaksi, A. & Laitkorpi, M. 1999. Extending the Object-Oriented Software Development Process with Component-Oriented Design. Journal of Object-Oriented Programming 12(1), 41-50.

King, N. 1999. Application Servers – Behind The Mystery. Intelligent Enterprise 2(3), 24-30.

Koskimies, K. 1997. Pieni oliokirja. Jyväskylä: Gummerus Kirjapaino Oy.

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

Lewandowski, S. M. 1998. Frameworks for Component-Based Client/Server Computing. ACM Computing Surveys 30 (1).

Matena, V. & Hapner, M. 1999. Enterprise JavaBeans™ Specification, v1.1. Sun Microsystems Inc.

Morgenthal J., P. 1999. Understanding Enterprise Java APIs. Component Strategies August 1999.

Orfali, R., Harkey, D. & Edwards, J. 1996a. The Essential Client/Server Survival Guide. 2. painos. John Wiley & Sons.

Orfali, R., Harkey, D. & Edwards, J. 1996b. The Essential Distributed Objects Survival Guide. John Wiley & Sons.

Parrish, A., Dixon, P. & Hale, D. 1999. Component Based Software Engineering: A Broad Based Model is Needed. Second International Workshop on CBSE in conjunction with ICSE99. 17.-18.5.1999. Los Angeles, Yhdysvallat.

Roman E. 1999. Mastering Enterprise JavaBeans™ and the Java™ 2 Platform, Enterprise Edition. John Wiley & Sons.

Roman E. & Öberg, R. 1999a. The Business Benefits of EJB and J2EE Technologies over COM+ and Windows DNA. The Middleware Company.

Roman E. & Öberg, R. 1999b. The Technical Benefits of EJB and J2EE Technologies over COM+ and Windows DNA. The Middleware Company.

Sametinger, J. 1997. Software Engineering with Reusable Components. Springer-Verlag.

Sauer, F. 2000. Business Interface. Saatavilla www-muodossa <URL:http://www.c2.com/cgi/wiki?BusinessInterface. 8.3.2000.

Shannon, B. 1999. Java™ 2 Platform Enterprise Edition Specification, v1.2. Sun Microsystems Inc.

Slama, D., Garbis, J. & Russell, P. 1999. Enterprise CORBA. Prentice-Hall.

Stroustrup, B. 1998. The C++ Programming Language. 3. painos. Reading, MA: Addison-Wesley

Sun Microsystems Inc. 2000. Designing Enterprise Applications with the Java™ 2 Platform, Enterprise Edition, v1.0.

Sun Microsystems Inc. 1999a. Java Naming and Directory Interface™ Application Programmin Interface (JNDI API).

Sun Microsystems Inc. 1999b. Simplified Guide to the Java™ 2 Platform, Enterprise Edition.

Sun Microsystems Inc. 1998. Java™ Remote Method Invocation Specification (Java RMI).

Szyperski, C. 1998. Component Software - Beyond Object-Oriented Programming. Reading, MA: Addison-Wesley.

Taivalsaari, A. 1993. A Critical View of Inheritance and Reusability in Object-Oriented Programming. Jyväskylä: Jyväskylän yliopistopaino ja Sisäsuomi oy.

Thomas, A. 1998. Enterprise JavaBeans™ Technology - Server Component Model for the Java™ Platform. Patricia Seybold Group.

Valesky, T. 1999. Enterprise JavaBeans™ - Developing Component-Based Distributed Applications. Reading, MA: Addison-Wesley.

Vayda, T. 1999. Organizing for Components – Managing risk and maximizing reuse. Component Strategies 1(8), August 1999.

Wallnau, K. C. 1999. On Software Components and Commercial ("COTS") Software. Second International Workshop on CBSE in conjunction with ICSE99. 17.-18.5.1999. Los Angeles, Yhdysvallat.

White, S. & Hapner, M. 1999. JDBC™ 2.1 API. Sun Microsystems Inc.

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.

Zona Research Inc. 1999. Enterprise JavaBeans Technology: A Business Benefits Analysis.

Address-komponentti

Tässä liitteessä on esitelty kohdepohjaisen Address-komponentin tarvitsemat ohjelmakoodit. Komponentin pysyvyydenhallinta toteutetaan säiliön toimesta, sillä yhtään tietokantaa suoraa käsittelevää metodia ei ole toteutettu. Komponentin avulla ei myöskään voi luoda uutta tietoa tietokantaan, sillä ejbCreate-metodien ja sen vastinparien ejbPostCreate-metodien esittelyt ja toteutukset puuttuvat kokonaan.

```
// Etärajapinta kohdepohjaiselle Address-komponentille

import javax.ejb.EJBObject;
import java.rmi.RemoteException;

public interface Address extends EJBObject, AddressBusinessInterface
{}

// Kotirajapinta kohdepohjaiselle Address-komponentille

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;
import javax.ejb.EJBException;

public interface AddressHome extends EJBHome
{
   // Koska uusia osoitetietoja ei voi luoda sovelluksen avulla
   // ei yhtään create-metodia tarvitse esitellä

   // Hakumetodin määrittely
   public Address findByPrimaryKey(AddressPK addrPK)
        throws FinderException, RemoteException;
}

// Liiketoimintametodit sisältävä rajapinta
// kohdepohjaiselle Address-komponentille

import java.rmi.RemoteException;

public interface AddressBusinessInterface
{
    public String getName()
        throws RemoteException;
    public String getAddress()
        throws RemoteException;
    public void setName(String name)
        throws RemoteException;
    public void setAddress(String address)
        throws RemoteException;
}
```

Address-komponentin pysyvyydenhallinta on toteutettu säiliön toimesta, joten avainluokan jäsenmuuttujien tulee olla suoraan itse komponentin toteuttavan luokan julkisia jäsenmuuttujia.

// Kohdepohjaisen Address-komponentin avainluokka

```
public class AddressPK implements java.io.Serializable
{
    public String myName;
    public AddressPK(String name)
    {
        myName = name;
    }
    public AddressPK() {}
    public String toString()
    {
        return myName;
    }
    public int hashCode()
    {
        return myName.hashCode();
    }
    public boolean equals(Object addressPK)
    {
        if ( ((AddressPK)addressPK).myName.
          compareTo(myName) == 0)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

// Kohdepohjaisen Address-komponentin toteutus

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

public class AddressBean implements EntityBean,
   AddressBusinessInterface
{
    protected EntityContext ctx;

    // Molemmat String-tyyppiset jäsenmuuttujat ovat public-määreellä
    // varustettuja, jotta säiliö voi tallentaa niiden arvot.
    public String myAddress;
    // Tätä muuttujaa käytetään avaimena
    public String myName;

    // Palauttaa henkilön nimen
    public String getName()
        throws RemoteException
    {
        return myName;
    }
```

```
    // Palauttaa henkilön osoitteen
    public String getAddress()
        throws RemoteException
    {
        return myAddress;
    }

    // Asettaa henkilölle uuden nimen
    public void setName(String name)
        throws RemoteException
    {
        this.myName = name;
    }

    // Asettaa henkilölle uuden osoitteen
    public void setAddress(String address)
        throws RemoteException
    {
        this.myAddress = address;
    }

    // Säiliön vaatimat metodit alkavat
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void ejbLoad() {}
    public void ejbStore() {}
    public void setEntityContext(EntityContext ctx)
        throws RemoteException
    {
        // Asettaa kontekstin jäsenmuuttujaan
        this.ctx = ctx;
    }
    public void unsetEntityContext()
        throws RemoteException
    {
        ctx = null;
    }
// Säiliön vaatimat metodit loppuvat
}
```

EditAddress-komponentti

Tässä liitteessä on esitelty tilallisen istuntopohjaisen EditAddress-komponentin tarvitsemat ohjelmakoodit.

```
// Etärajapinta istuntopohjaiselle EditAddress-komponentille

import javax.ejb.EJBObject;

public interface EditAddress extends EJBObject,
    EditAddressBusinessInterface
{}

// Kotirajapinta istuntopohjaiselle EditAddress-komponentille

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;
public interface EditAddressHome extends EJBHome
{
    // Luontimetodin määrittely
    EditAddress create()
        throws RemoteException, CreateException;
}

// Liiketoimintametodit sisältävä rajapinta istuntopohjaiselle
// EditAddress-komponentille

import java.rmi.RemoteException;
import javax.ejb.FinderException;

public interface EditAddressBusinessInterface
{
    public String getName(String name)
        throws RemoteException, FinderException;
    public String getAddress(String name)
        throws RemoteException, FinderException;;
    public void setName(String oldName, String newName)
        throws RemoteException, FinderException;;
    public void setAddress(String name, String address)
        throws RemoteException, FinderException;;
}
```

```
// Istuntopohjaisen EditAddress-komponentin toteutus

import javax.ejb.SessionBean;
import javax.ejb.CreateException;
import javax.ejb.FinderException;
import java.rmi.RemoteException;
import javax.ejb.EJBException;
import javax.ejb.SessionContext;


import java.util.Properties;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class EditAddressBean
    implements SessionBean, EditAddressBusinessInterface
{
    private SessionContext ctx;
    // Viite kohdepohjaisen komponentin kotirajapintaan
    private AddressHome aHome;

    // Palauttaa parametrina annetun henkilön nimen
    public String getName(String name)
        throws RemoteException, FinderException
    {
        Address addr = aHome.findByPrimaryKey(new AddressPK(name));
        return addr.getName();
    }
    // Palauttaa parametrina annetun henkilön osoitteen
    public String getAddress(String name)
        throws RemoteException, FinderException
    {
        Address addr = aHome.findByPrimaryKey(new AddressPK(name));
        return addr.getAddress();
    }
    // Asettaa parametrina annetulle henkilölle uuden nimen
    public void setName(String oldName, String newName)
        throws RemoteException, FinderException
    {
        Address addr = aHome.findByPrimaryKey(new AddressPK(oldName));
        addr.setName(newName);
    }
    // Asettaa parametrina annetulle henkilölle uuden osoitteen
    public void setAddress(String name, String address)
        throws RemoteException, FinderException
    {
        Address addr = aHome.findByPrimaryKey(new AddressPK(name));
        addr.setAddress(address);
    }
```

```
    // Säiliön vaatimat metodit alkavat
    public void ejbCreate() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext ctx)
    {
        try
        {
            this.ctx = ctx;
            Context initContext = new InitialContext();

            // Haetaan viite osoitekomponentin kotirajapintaan
            // myöhempää käyttöä varten
            Object objRef = initContext.lookup("MyAddress");
            aHome =(AddressHome)PortableRemoteObject.narrow(
                   objRef,AddressHome.class);
        }
        catch (Exception e) {
            throw new EJBException( e );
        }
    }
    // Säiliön vaatimat metodit loppuvat
}
```

ReadAddress-komponentti

Tässä liitteessä on esitelty tilattoman istuntopohjaisen ReadAddress-komponentin tarvitsemat ohjelmakoodit.

```
// Etärajapinta istuntopohjaiselle ReadAddress-komponentille

import javax.ejb.EJBObject;

public interface ReadAddress extends EJBObject,
   ReadAddressBusinessInterface
{}

// Kotirajapinta istuntopohjaiselle ReadAddress-komponentille

import javax.ejb.EJBHome;
import javax.ejb.CreateException;
import java.rmi.RemoteException;

public interface ReadAddressHome extends EJBHome
{
    // Luontimetodin määrittely, jonka tulee olla parametriton,
    // sillä kyseessä on tilaton istuntopohjainen komponentti
    ReadAddress create()
        throws RemoteException, CreateException;
}

// Liiketoimintametodit sisältävä rajapinta istuntopohjaiselle
// ReadAddress-komponentille

import java.rmi.RemoteException;

public interface ReadAddressBusinessInterface
{
    public AddressObject readAddress(String name)
        throws RemoteException;
}

// Osoitetietojen siirrossa käytettävä AddressObject luokka, joka
// sisältää henkilön nimen ja osoitteen jäsenmuuttujinaan.

public class AddressObject implements java.io.Serializable
{
    private String myName;
    private String myAddress;

    public AddressObject(String name, String address)
    {
        myName = name;
        myAddress = address;
    }
    public String getAddress()
    {
        return myAddress;
```

```
    }
    public String getName()
    {
        return myName;
    }
}

// Istuntopohjaisen ReadAddress-komponentin toteutus

import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.rmi.RemoteException;
import java.util.Properties;
import javax.rmi.PortableRemoteObject;

public class ReadAddressBean
    implements SessionBean, ReadAddressBusinessInterface
{
    private SessionContext ctx;
    // Viite kohdepohjaisen komponentin kotirajapintaan
    private AddressHome aHome;

    public AddressObject readAddress(String name)
        throws RemoteException
    {
        AddressObject addrObj;
        try
        {
            // Haetaan viite osoiteolion kotirajapintaan
            Context initContext = new InitialContext();
            Object objRef = initContext.lookup("MyAddress");
            aHome = (AddressHome)PortableRemoteObject.narrow(
                    objRef, AddressHome.class);
            // Etsitään parametrina annettua henkilöä vastaavat tiedot
            Address addr = aHome.findByPrimaryKey(
                new AddressPK(name));
            // Luodaan osoitetiedot sisältävä olio
            addrObj = new AddressObject(
                addr.getName(), addr.getAddress() );
        }
        // Tämä vastaanottaa kaikki mahdolliset poikkeukset
        catch (Exception e) {
            e.printStackTrace();
        }
        return addrObj;
    }
    // Säiliön vaatimat metodit alkavat
    public void ejbCreate() {}
    public void ejbRemove() {}
    public void ejbActivate() {}
    public void ejbPassivate() {}
    public void setSessionContext(SessionContext ctx)
    {
        this.ctx = ctx;
    }
    // Säiliön vaatimat metodit loppuvat
}
```

Asiakassovellus

Tässä liitteessä on esitelty yksinkertaisen asiakassovelluksen ohjelmakoodi. Asiakassovellus lukee ensin kuvitteellisen henkilön nimi- ja osoitetiedot ja tulostaa ne. Tämän jälkeen henkilön osoitetieto tulostetaan vielä kerran, jonka jälkeen osoitetietoa muokataan ja uusi osoite tulostetaan.

// Asiakassovelluksen ohjelmakoodi

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.rmi.PortableRemoteObject;

public class AddressClient {

    public static void main(String[] args) {
        try
        {

            Object objref = null;
            AddressObject addObj = null;
            Context initial = new InitialContext();

            objref = initial.lookup("MyReadAddress");
            ReadAddressHome readAddHome =
                (ReadAddressHome)PortableRemoteObject.narrow(objref,
                                        ReadAddressHome.class);
            ReadAddress readAdd = readAddHome.create();
            addObj = readAdd.readAddress("Teppo");
            System.out.println(addObj.getAddress());
            System.out.println(addObj.getName());

            objref = initial.lookup("MyEditAddress");
            EditAddressHome EditAddHome =
                (EditAddressHome)PortableRemoteObject.narrow(objref,
                                        EditAddressHome.class);
            EditAddress EditAdd = EditAddHome.create();
            System.out.println(EditAdd.getAddress("Teppo"));
            EditAdd.setAddress("Teppo", "Kotipolku 9");
            System.out.println(EditAdd.getAddress("Teppo"));
            EditAdd.remove();

        }
        // Tämä vastaanottaa kaikki mahdolliset poikkeukset.
        catch (Exception ex) {
               System.err.println("An exception occured." );
               ex.printStackTrace();
        }
    }
}
```

```
// Sovelluksen tarvitseman tietokantataulun luonti- ja tiedonsyöttö-
// lauseet

// Luo uuden address nimisen taulun, johon voidaan lisätä
// henkilön nimi ja osoite 20 merkin mittaisena merkkijonona
create table address (name varchar(20),address varchar(20));
// Luo tietokantatauluun Teppo nimisen henkilön ja sille osoittteen
insert into address values ('Teppo', 'Omakuja 2');
```