Pääkirjoitus

Komponenttipohjainen sovelluskehitys on tätä päivää. Sunin luotsaama EJB-palvelinkomponenttimalli pyörii lähes jokaisen sovelluspalvelintoimittajan palvelimessa. Luulisi kehitysvauhdin hidastuvan pikkuhiljaa, mutta vaikka J2EE:n perusspesifikaatiot pysyvät nimiltään samana, uusia versioita tulee J2EE:stä hyvään tahtiin. Tämä on iloinen uutinen siinä mielessä, että uusien piirteiden saaminen suosituksiin ei kestä vuosia. Toisaalta samalla muokataan olemassa olevia piirteitä jonkin verran, mikä saattaa aiheuttaa muutoksia kohtalaisen tuoreeseenkin ohjelmistoon. Joku viisas onkin todennut, että ”tämän päiväinen on huomenna ”legacya””.

Kävin tässä taannoin viime keväänä eräässä seminaarissa kuuntelemassa Bud Lawsonin luentoa sovelluskehityksen monimutkaistumisesta ja ohjelmakoodin määrän kasvamisesta. Tämä herätti minussa mielikuvan, jossa ohjelmakoodi alkoi muistuttaa roskavuorta, joka jossain vaiheessa aiheuttaa varsinaisen ohjelmakoodin roskaräjähdyksen, ja maailman täytyttyä ohjelmistokoodin roskalla on aloitettava uljas, uusi maailma, jossa palataan taas takaisin vanhaan, jossa kaunista on pienet ja yksinkertaiset asiat. Vielä emme taida kuitenkaan olla tässä pisteessä.

Java-maailma elää muiden mukana, ja vaikka suoritusympäristöjen määrä onkin muutaman vuoden aikana vähentynyt entisestä neljästä nykyiseen kolmeen, Javan ohjelmointirajapinnat ovat lisääntyneet melkoista vauhtia. Samoin ohjelmointikielen piirteet ovat lisääntyneet, joka näkyy mm. Javan työkalupakin (JDK) koon kasvuna. Vanhoina hyvinä aikoina, kun Java oli vielä nuori, JDK 1.0.2:n asennuspaketin koko oli n. 3,5 MB. Nyt on menossa versio 1.4, ja sen asennuspaketin koko on n. 37 MB.

Java vastaa sovelluskehitysmaailman haasteisiin J2EE-, J2SE- ja J2ME-ympäristöillään, joita yhdessä käyttämällä saadaan aikaan monipuolisia ohjelmistoratkaisuja: JVM voi pyöriä muutaman kilon kokoisena kännykässä ja kännykän sovellus voi keskustella yrityksen suurten palvelinten kanssa muodostaen erilaisia monikanavaratkaisuja. Javalla voi rakentaa tänä päivänä yrityksen tai organisaation operatiivisen järjestelmän. Järjestelmiä voidaan Java-tekniikoilla liittää toisiinsa sekä tarjota WWW-käyttöliittymiä olemassa oleviin järjestelmiin.

Kehityksessä mukana pysyminen vaatii jatkuvaa markkinoiden seurantaa, uusien asioiden tekemistä ja kokeilemista ja opiskelua. On mahdollista sertifioida työntekijöitä erilaisilla tutkinnoilla, jotka takaavat henkilön osaavan tietyt asiat. Microsoft ja Sun tarjoavat kumpikin omiin tekniikoihinsa omat tutkintonsa. Yhteistä näillä tutkinnoilla on kuitenkin se, että tutkinnon suorittanut henkilö on joutunut panostamaan tutkintoon kohtalaisen paljon. Yritysten osaamispääoma kasvaa, ja mittarit osaamisesta voidaan ehkä joiltain osilta yhtenäistää.

Liiketoimintamallit, välineet ja tekniikat kehittyvät, näin tapahtuu myös Javalle. Kukaan ei saa jäädä tuleen makaamaan.

Simo Vuorinen, TietoEnator Oyj

J2EE-arkkitehdin työkalupakki

Simo Vuorinen,
TietoEnator Oyj

Ohjelmistoarkkitehti on projektissa se henkilö, joka laatii ohjelmiston rakennussuunnitelman ja kuvauksen, ja viestii sen sitten kaikille sidosryhmille. Arkkitehdin työ ei pääty siihen, että hän laatii arkkitehtuurin, vaan ohjelmistoarkkitehti myös valvoo suunnitelmien toteuttamista. Arkkitehti siis on olemassa projektissa niin kauan, kuin projekti jatkuu. Java-maailmassa pätee ohjelmistoarkkitehdin tehtäviin sama kuin muissakin ohjelmistoympäristöissä, mutta Javan J2EE-ympäristöön löytyy erittäin kattava määrä siihen varta vasten sovitettuja työkaluja. Tässä artikkelissa käsitellään ohjelmistoarkkitehdin tehtäviä ja työvälineitä J2EE-ympäristön näkökulmasta.

Arkkitehtuuri

Spesifikaatio

Järjestelmän tai ohjelmiston arkkitehtuuri ei ole pelkkä paperinpala, jossa on kuva ympäristöstä ja teknisistä osista sijoitettuna ympäristöön. Arkkitehtuurispesifikaatio ja arkkitehtuuri on kommunikaatioväline, jonka välityksellä ohjelmistoarkkitehti kommunikoi sidosryhmien kanssa, joilla voi olla hyvinkin erilaiset tarpeet, vaatimukset ja näkymä ko. sovelluksesta tai järjestelmästä. Ohjelmistoarkkitehti ei myöskään poistu projektista saatuaan spesifikaation valmiiksi, vaan hän myös valvoo sen toteuttamista sovitulla tavalla.

Arkkitehtuuri terminä

Yleisesti käytettyjä ohjelmistoarkkitehtuurin määritelmiä on useita. Itse olen tyytynyt seuraavaan määritelmään: ”Arkkitehtuurilla tarkoitetaan järjestelmän tai sovelluksen rakennetta tai rakenteita, jotka koostuvat ohjelmistokomponenteista, niiden näkyvistä attribuuteista, ja niiden välisistä suhteista. ” [Bass et al. s.6]

Samoin ohjelmistokomponenteille löytyy useita määritelmiä, ja käytän ohjelmistokomponenttitermiä tässä yhteydessä kuvaamaan ” itsenäisesti tuotettuja tai hankittuja ja jaeltavissa olevia binäärisiä yksiköitä, jotka toimivat ja kommunikoivat keskenään muodostaen toimivan järjestelmän ” [Szyperski].

Arkkitehtuurispesifikaation tulisi siis minimissään kuvata järjestelmän tai sovelluksen komponentit ja niiden keskinäiset riippuvuudet, kertoa jotakin järjestelmälle esitetyistä vaatimuksista, ja periaatteet, joilla näihin vaatimuksiin vastataan. Tämä yleensä tarkoittaa niitä toiminnallisia ja laadullisia attribuutteja, jotka ovat järjestelmälle tai sovellukselle tärkeitä, ja niitä arkkitehtuuri- ja suunnittelumalleja, joilla näihin vaatimuksiin pyritään vastaamaan.

Arkkitehtuurin merkitys

Vaikka ohjelmistoarkkitehtuureilla ja niiden tutkimuksella ja kehityksellä on varsin pitkä historia, ehkä vasta viime aikoina on alettu ymmärtää arkkitehtuurin merkitys ohjelmistoprojektissa. Ohjelmistot, joissa on huolellisesti mietitty, suunniteltu, dokumentoitu ja evaluoitu arkkitehtuuri, näyttäisivät menestyvän paremmin, palvelevan tarkoitustaan paremmin, ja tulevan pitkällä tähtäimellä edullisemmiksi. Arkkitehtuurin tarkoitus on kuvata kohteena oleva ohjelmisto, ja sen sekä kommunikointivälineroolinsa lisäksi parantaa laatua. Laatua on mm. uudelleenkäytettävyys, muokattavuus ja ylläpidettävyys. Jokaisen ohjelmiston laatuattribuutit vaihtelevat riippuen ohjelmiston luonteesta. Nämä attribuutit ovat kuitenkin mitattavia määreitä, joita arvioidaan sovitulla mittausmenetelmällä ja mittaristolla. Mittausmenetelmiä on mm. SEI:n ATAM [SEI].

Arkkitehtuurityö ei pääty projektinkaan päättyessä, vaan arkkitehtuuria ylläpidetään koko ohjelmiston elinkaaren ajan. On sydäntä särkevää nähdä, kuinka hyvin suunniteltu ja toteutettu ohjelmistoarkkitehtuuri rapautetaan nopeastikin reikäjuustoa muistuttavaksi rakennelmaksi unohtamalla arkkitehdin tehtävät ohjelmistoprojektien siirtyessä ylläpitoon ja alkuperäisen ohjelmistoarkkitehdin poistuessa projektista. Tähän voi vaikuttaa myös osaltaan se, että yritykset ja organisaatiot usein kilpailuttavat erikseen järjestelmän tuottamisen ja ylläpidon, jolloin kilpailun ollessa ohjelmistoyritysten välillä kireää, unohdetaan vastuu asiakkaan järjestelmän tulevaisuudesta. Tätä ei voi mielestäni kuitenkaan kokonaan sälyttää pelkästään ohjelmistotalon harteille, vaan myös ostajan on ymmärrettävä, mitä ostaa. Tulevaisuudessa – tai miksei nykyäänkin - ostaja voi törmätä tilanteeseen, jossa ohjelmistotalo A tekee määrittelyn, talo B suunnittelee ja toteuttaa, ja ohjelmistotalo C ylläpitää. Tällöin laadunvalvonta on entistä tärkeämpää.

J2EE-ohjelmistoarkkitehdin kirjahylly

Peruskirjat

Kirjahylly on ohjelmistoarkkitehdin ”sielun peili”; se usein kertoo, millaisiin asioihin arkkitehti on syventynyt, ja mitkä asiat ovat lähellä arkki-

tehdin sydäntä. Ohjelmistoarkkitehdin hyllystä löytyvät yleensä yrityksen menetelmien ja standardien lisäksi useimmiten perustavaa laatua olevat ohjelmistoarkkitehtuureja koskevat kirjat, joissa kuvataan arkkitehtuurityyppejä ja –malleja, ohjelmistotuotantoa käsittelevät kirjat, ja suunnittelumalleja (design patterns) sekä idiomeja koskevaa kirjallisuutta.

J2EE-kirjallisuus

J2EE-ratkaisujen osalta kannattaa pitää huoli siitä, että hyllystä löytyy sekä hyvä EJB- että Web-säiliön toimintaa koskeva kirja, XML-kirja, Javan tietoturvaa käsittelevä kirja ja J2EE-suunnittelumalleja koskeva kirja. Muu kirjallisuus vaihtelee riippuen mm.J2EE-ohjelmistoarkkitehdin asiakassovellusympäristöstä, joka voi vaihdella monikanavaympäristöstä vaikkapa kämmentietokoneeseen, matkapuhelimeen, selaimeen tai toiseen sovellukseen. Lisäksi J2EE:hen kohtalaisen uutena piirteenä tuotu sanomavälitteisyyden mahdollisuus voi vaikuttaa hankittavaan kirjallisuuteen.

J2EE-suunnittelumallit

J2EE-suunnittelumallit [Sun] ovat tarkoitettu J2EE-sovellusten peruselementeiksi. Näistä alkaa pian olla jo runsaudenpulaa, ja käytettävät suunnittelumallit kannattaa valita huolella. On alkanut esiintyä merkkejä jo sellaisestakin, että suunnittelumalli-nimitystä käytetään varsin heppoisin perustein. Mielestäni suunnittelumallin edellytys on, että se vastaa johonkin tiettyyn ongelmaan, mikä yleisesti esiintyy. Lisäksi suunnittelumalli on käytännössä useassa hankkeessa koeteltu ratkaisu ja toimivaksi todettu. Niitä ei välttämättä kannata kaikkia ympätä samaan projektiin. Itse käytän säännöllisesti kourallista perusmalleja, ja hankkeen piirteistä ja laatuattribuuteista riippuen tarpeen mukaan pohdinnan jälkeen lisää. Olen pääasiassa pitäytynyt alkuperäisissä Sunin Java-sivustolta löytyvissä suunnittelumalleissa.

Spesifikaatiot, tutoriaalit ja blueprintit

Lisäksi kannattaa käydä tiuhaan tahtiin seuraamassa ja lataamassa itselleen tarpeelliset J2EE-spesifikaatiot. Näistä löytyy usein tarkempaa tietoa kuin kirjoista. Itselläni on hyllyssä versioittain J2EE-spesifikaation lisäksi EJB-, Servlet-, JSP-, Connector- ja XML-spesifikaatiot.

Javan WWW-sivustolta löytyvät lisäksi tutoriaalit J2EE-materiaaliin, ja mielestäni ne kannattaa käydä läpi, koska niissä kuvataan usein perustavaa laatua olevat ratkaisutavat ja opetetaan yksinkertaisesti tutoriaalin otsikon aiheen perusteet. Kun perusteet on opittu, on helppo lähteä suunnittelemaan uudelleenkäytettäviä ratkaisuja. Blueprints-tyyppiset oppaat pyrkivät kuvaavaan tavan kehittää sovelluksia, joissa on otettu huomioon siirrettävyyteen ja skaalautuvuuteen liittyviin seikkoihin. Blueprints-oppaat löytyvät Javan WWW-sivustolta ilmaisina.

Anti-suunnittelumallit

Antisuunnittelumallit kuvaavat tilanteita, joissa jokin on mennyt pieleen. Näistä antimalleista tunnistaa osan kohtalaisen helposti, ja ne voivat olla todellisia ongelmia projekteissa kohtalaisen useinkin. On hyvä olla olemassa kättä pidempää tällaisia tilanteita varten. Helpoimmin tunnistettavia ongelmatilanteita ovat esim. ”Analysis Paralysis” ja ” Spaghetti Code” [Brown et al.].

Ohjelmistokehitysprosessi ja mallinusmenetelmä

Riippumatta siitä, mitä ohjelmistokehitysmenetelmää tai –prosessia yrityksessä käytetään, on ohjelmistoarkkitehdin osattava se alusta loppuun. Usein tilanne voi olla esim. se, että yrityksellä on oma ohjelmistokehitysprosessi, joka voi pohjautua esim. RUP:iin. Tällöin ohjelmistoarkkitehdin olisi hyvä tuntea molemmat, tai ainakin tietää erot itse käyttämänsä, vähemmän tunnetun menettelyn, ja yleisesti maailmalla tunnetun menettelyn välillä. Tämä ei kuitenkaan vielä riitä. Ohjelmistoarkkitehdillä on oltava kohtalainen käsitys tunnetuista menetelmistä, jotta osaa tarvittaessa soveltaa tilanteiden muuttuessa muita tapoja tehdä sovelluskehitystä. Prosessissa yleensä käytetään tiettyä mallinnuskieltä, joka ohjelmistoarkkitehdin on tunnettava. UML alkaa olla pikkuhiljaa de facto-standardi. mallinnuskieli.

Välineet

Mallinnusväline

J2EE-ohjelmistoarkkitehdin perustyökalut ovat dokumentaation ja suunnitelmien lisäksi vähimmilläänkin melkoiset: jokin mallinnusväline, ohjelmakoodieditori, erilaiset testaustyökalut, joukko emulaattoreita asiakassovelluksen emulointiin, ja joukko sovelluspalvelimia testausympäristöinä.

Mallinnusvälineistä pidän itse sellaisista, joissa käyttöliittymä on intuitiivinen ja mahdollistaa nopean käyttöönoton. Mallinnusvälineen ei mielestäni tule pakottaa tiettyyn menetelmään tai ohjelmistokehitysprosessiin, mutta ko. piirre on lisäarvona hyvä. Mallinnusvälineissä alkaa olla kohtalaisen hyvin otettu huomioon jo se, millä teknologialla projekti lopulta toteutetaan. Tämä mahdollistaa mm. sopivien suunnittelumallien käytön, sopivien komponenttityyppien valinnan. Kun mallista sitten luodaan ohjelmakoodia, monet asiat ovat jo valmiiksi kohdallaan. Mallinnusvälineessä on myös hyvä olla mahdollisuus kahden suuntaiseen työskentelyyn, eli

reverse-engineering on erittäin käyttökelpoinen piirre.

Mallinnusvälineessä on hyvä olla myös valmiina perustavaa laatua olevat suunnittelumallit, ettei niitä tarvitse kaikkia rakentaa tyhjästä. J2EE:n ollessa kyseessä, erittäin hyvä lisä on ns. J2EE-suunnittelumallit. Käytettävät kaaviot ja notaatio riippuvat ohjelmistokehitysprosessista tai –menetelmästä. UML-notaation käyttäjänä peruskaaviot, joilla itse tulen jo toimeen, ovat liiketoimintaprosessikaavio, käyttötapauskaavio, luokkakaavio, vuorovaikutuskaavio, tilakaavio, komponenttikaavio ja sijoituskaavio. Kaupalliset mallinnusvälineet eivät ole kovinkaan edullisia, ja ilmaisvälineitäkin löytyy, joilla tulee toimeen vallan mainiosti.

IDE

Ohjelmakoodieditoreista ollaan montaa mieltä – jotkut koodarit ovat sitä mieltä, että debuggerit ja hienot IDE:t (Integrated Development Environment) ovat turhia tai ei välttämättömiä. Olen vahvasti eri mieltä. Mielestäni on kuitenkin sallittavaa, että arkkitehti rakentaa ensimmäisen kerran kaiken rivieditorilla ymmärtääkseen mitä taustalla tapahtuu - mutta täta toimenpidettä ei tarvitse toistaa. Hyvä IDE nopeuttaa ohjelmakoodin tuottamista, koska koodarin ei tarvitse muistaa kaikkia API:n metodeja ja niille välitettäviä parametreja ulkoa. Lisäksi näissä on usein makroja, joilla voi tuottaa bulkkikoodinpätkiä, vähentäen näin rutiinikoodausta. IDE:ssä voi olla myös dynaaminen tekstin tarkistus, jolloin kirjoitusvirheet harvenevat. Samoin API-dokumentaatio on usein painikkeen päässä, eikä sitä tarvitse erikseen hakea esiin levynnurkasta. Yritystasoiset IDE:t mahdollistavat vielä eri valmistajien sovelluspalvelinten liittämisen IDE:iin ja sovellusten debuggauksen sovelluspalvelinten sisällä.

IDE:ssä on oltava ehdottomasti hyvä debuggeri, ja sitä kannattaa käyttää. On tietenkin tilanteita, jolloin debuggerista ei ole hyötyä, esim. silloin, kun testataan säikeiden toimintaa. Debuggeri ei yleensä kykene monisäikeistetyssä ohjelmassa jäljittämään kaikkea tapahtumia niiden varsinaisessa tapahtumajärjestyksessä, eikä tähän auta muu kuin debug-tekstien kirjoittaminen ulostulovirtaan. Kuitenkin peruskoodin debuggaus auttaa löytämään virheet helpommin kuin mikään muu vastaava menettely. Mielenkiintoinen huomio on se, että Javan WWW-sivuilta löytyy linkki Bil Lewisin uuteen Omniscient Debuggeriin, johon kannattaa käydä tutustumassa.

Testauksesta muutama sana

Minimivaatimuksena mielestäni jokainen ohjelmistoarkkitehti on velvollinen huolehtimaan siitä, että ohjelmistolle ajetaan ainakin yksikkötestit, profilointitestit , suorituskykytestit ja ohjelmakoodin analyysi. Jos kyse on sitten vielä isosta järjestelmästä, voidaan käyttää kaupallisia välineitä, joista löytyy vääntöä joka lähtöön. Useimmat tällä hetkellä rakennettavista uusista yritysten ja yhteisöjen järjestelmistä perustuvat Internetin käyttöön, ja suuri osa näistä on WWW-järjestelmiä. Tällöin on huolehdittava myös riittävän kattavasta tietoturvatestauksesta.

Jos IDE:ssä ei itsessään ole integroituna yksikkötestausvälinettä, kannattaa sellainen ladata verkosta. JUnit-testauskehys on varsin hyvä tällainen, ja se löytyy osoitteesta www.junit.org. JUnit-välineen sivustolla on hyvät ohjeet välineen käytöstä, ja linkkejä sitä koskeviin artikkeleihin. Itse käytän sitä komponenttien toiminnallisten ja laadullisten attribuuttien todentamiseen. Varsinkin isompien järjestelmien testauksessa tällaisesta on valtava hyöty; testit ajetaan aina muutosten jälkeen, jolloin paljastuu karu totuus siitä, minne muualle tehty muutos vaikutti, kuin minne sen piti.

Profilointivälineet auttavat löytämään Javan vastineet muistivuodoille. Javassa on itsessäänkin suorituspoptio, jolla saa suorituksen aikaista profilointidataa. Se ei kuitenkaan itsessään ole kovinkaan helppoa luettavaa, ellei todella tiedä, mitä etsii, ja silloinkin dataa tulee melkoinen määrä parsittavaksi. Hyvä profilointiväline maksaa itsensä takaisin varsin nopeasti – heti, kun toiminta on monimutkaista ja tapahtumia on paljon. Käytettynä yhdessä suorituskykytestivälineen kanssa profilointiväline on erittäin hyvä yhdistelmä.

Itse käytän usein suorituskykyrobottia ja profilointivälinettä sekä yhdessä että erikseen, jolloin näen yhtäaikaisesti monisäikeisen ympäristön toiminnan ja suhteellisen suorituskyvyn. Profilointivälineet esittävät usein varsin intuitiivisesti pullonkaulakohdat ja mahdolliset säikeistykseen liittyvät ongelmat. Säikeistykseen liittyviä ongelmia ei useimmiten pysty havaitsemaan tavallisella debuggerilla. Joissakin profilointivälineissä on myös coverage-tyyppinen testausmahdollisuus, joka kertoo jos jokin kohta koodista jää aina suorittamatta.

Ohjelmakoodin auditointi ja analysointi on tärkeä apuväline ohjelmistoarkkitehdille. Sillä on mahdollisuus selvittää ja raakata pois suuri määrä perustavaa laatua olevia virheitä, jotka myöhemmässä vaiheessa aiheuttaisivat mahdollisesti päänvaivaa. Lähdekoodin auditointiin ja analysointiin löytyy myös automatisoivia apuvälineitä, joiden käyttöä suosittelen. Ne säästävät valtavasti aikaa, ja varmistavat sen, että ohjelmakoodi on jotenkuten kuosissaan. Auditoinnin lisäksi on mittausvälineitä, jotka muodostavat nopeasti arkkitehdille kuvan koodin laadusta. Jos ohjelmistoarkkitehdilla ei ole käytössään apuvälinettä, perinteisen

koodirivien lukumäärän lisäksi kannattaa kiinnittää huomiota ainakin luokkien välisen sidonnan määrään, perintähierarkian syvyyteen, luokan attribuuttien ja metodien määrään ja koheesion puuttumiseen.

Ohjelmakoodin refaktorointi

Refaktorointi on uudelleenjäsennystä, jonka avulla lopputuloksena oleva tuotos, tässä tapauksessa ohjelmakoodi, on siistiä ja optimaalista siten, että koodin monikerta vähenee ja sitä ei löydy replikoituna sinne sun tänne. IDE:issä alkaa olla jo enemmän sääntö kuin poikkeus, että niistä löytyy jonkinlaiset ohjelmakoodin refaktorointityökalut. Refaktorointia kannattaa harjoittaa koko ohjelmistoprojektin ajan ja ehdottomasti ainakin koodauksen alusta lähtien.

J2EE-ympäristöistä

Itse pidän ajatuksesta, jossa softan kehitys tapahtuu J2EE referenssi-implementaation avulla, jolloin voisi ainakin periaatteessa olettaa että rakennelma toimisi muillakin samaa J2EE:n versiotasolla olevien sovelluspalvelinten kanssa. Käytännössä näyttää kuitenkin siltä, että eri valmistajien sovelluspalvelimien komponenttien jakelukuvausten (deployment descriptor) verifioijat ovat varsin eritasoisia ja hyväksyvät ja hylkäävät eri asioita. Tämä vaikeuttaa ajatusta standardista sovelluskehityksestä. Itse olen päätynyt siihen, että levitän sovelluksen 3-4 sovelluspalvelimelle ja testaan sen toimivuutta testihaarniskan, suorituskykyvälineen ja debuggerin avulla. Tällainen paljastaa usein mielenkiintoisia eroja sovelluspalvelinten välillä. Ensialkuun tällainen testaaminen voi tuntua työläältä ja jopa turhalta, mutta totuttelun jälkeen vauhti nopeutuu niin, että mielestäni se kannattaa tehdä. Aina ei tietenkään tarvitse testata kaikkea jokaisella verkosta löytyvällä sovelluspalvelimella, vaan parikin palvelinta riittää.

Kokemus ja jatkuva opiskelu

Ohjelmistoarkkitehti on kokenut henkilö, joka on oppinut tekemällä, näkemällä ja opiskelemalla. Sisäinen intuitio tai ”gut feeling” kasvaa kun kokemus kertyy, ja siihen oppii luottamaan enemmän. Pelkästään sisäisen intuition varaan ei tietenkään lasketa mitään, vaan kokenut henkilö myös sparraa muiden kanssa ajatuksiaan ja pyytää apua silloin kun ei itse tunne asiaa riittävän hyvin. Huolenaiheet kannattaa tuoda esiin ajoissa, sillä ratkaisu voi olla nenän edessä mutta sitä ei välttämättä itse hahmota heti. Kannattaa jo varhaisessa vaiheessa oppia arvostamaan muiden ammattilaisten näkemystä ja kokemusta. Tämän päivän opettaja voi olla huomenna oppilas.

Itse lisäisin vielä ohjelmistoarkkitehdin piirteisiin kyvyn jakaa tietoa ja mentoroida muita. Mentorointia on monen tasoista, ja mentorin on oltava kohtalaisen tunneälykäs jotta viesti menee perille; hän – tässä tapauksessa ohjelmistoarkkitehti - joutuu viestimään arkkitehtuuria eri sidosryhmille kunkin ryhmän ymmärtämällä kielellä. Viestiin liittyy aina tulkinta, ja viesti tulisi tulkita viestijän kannalta positiivisesti.

Lähdeluettelo ja linkkejä

[Bass et al]. Software Architecture in Practice. Bass L., Clements P., Kazman R. Addison-Wesley, 1998.

[Brown et al.] Antipatterns. Refactoring Software, Architectures, and Projects in Crisis. Brown W., Malveau R., McCormick H., Mowbray T. Wiley Computer Publishing, 1998.

[SEI]. The Architecture Tradeoff Analysis Method. Technical report CMU/SEI-98-TR-008 ESC-TR-98-008, 1998.

[Szyperski]. Component Software: Beyond Object-Oriented Programming. Szyperski, C. Addison-Wesley, 1998.

Software Architect Bootcamp. Malveau R., Mowbray T. Prentice-Hall, 2001.

Sunin J2EE-sivut: http://java.sun.com/j2ee/

J2EE-suunnittelumallit: http://java.sun.com/blueprints/patterns/index.html

JUnit: http://www.junit.org/index.htm

Omniscient Debugger: http://java.sun.com/features/2002/08/omnidebug.html

Simo Vuorinen,
Software Architect,
TietoEnator Oyj

J2EE-suunnittelumallit

Heikki Vesalainen ja
Matti Kokkola,
Mermit Business Applications Oy

Java 2 Enterprise Edition (J2EE) on Javan laajennus, joka määrittelee joukon standardeja, joilla toteutetaan monikerrosmallin mukaisia sovelluksia. J2EE tukee sovellusten rakentamista määrittelemällä niille standardin komponenttirakenteen ja tarjoamalla näille komponenteille joukon palveluita.

J2EE:n haasteena on sen monimutkaisuus ja mahdollisuus toteuttaa asioita usealla eri tavalla. Tämä johtaa siihen, että kokematon arkkitehti saa aikaan laatuattribuuteiltaan (esim. tehokkuus, muokattavuus, ylläpidättävyys) huonon arkkitehtuurin.

Hyvä arkkitehtuuri syntyy paitsi huolellisen suunnittelun, myös kokemuksen pohjalta. Kokemusta saa joko tekemällä asioita itse, tai oppimalla muilta. Suunnittelumallit (eng. Design Patterns) ovat tiivistettyä kokemusta hyvin strukturoidussa muodossa. Suunnittelumallit eivät siis ole mitään valmiita komponentteja tai koodia, vaan strategioita ongelmien ratkaisemiseksi.

Tässä artikkelissa annetaan yleiskatsaus J2EE-arkkitehtuureihin liittyviin suunnittelumalleihin esimerkkitapauksen avulla. Tavoitteena ei ole esitellä kaikkia suunnittelumalleja, vaan motivoida käyttämään niitä. Lukijalta oletetaan J2EE:n tuntemista ja UML-mallinnuskielen osaamista.

Artikkelissa käsitellään viisi eri suunnittelumallia esimerkkisovelluksen avulla. Esimerkissä keskitytään ainoastaan liiketoimintakerrokseen (EJB-kerros) liittyviin suunnittelumalleihin, koska useimmiten siellä tehdään sovelluksen laatuattribuuttien kannalta kriittisimmät suunnittelupäätökset.

Artikkelissa käytetään suunnittelumallien englanninkielisiä nimiä. Tämän siksi, että suunnittelumallien englanninkieliset nimet ovat vakiintuneet käytössä ja sunnittelumallien yksi päätarkoituksista on antaa yhteinen nimitys yleisesti tunnetuille asioille.

Kaikilla suunnittelumalleilla on useita toteutustapoja. Tässä artikkelissa esitellään kustakin suunnitelumallista vain yksi toteutustapa. Lisää toteutusesimerkkejä löytyy lopussa listatuista
lähteistä.

Taustaa

Suunnittelmallit (eng. Design Patterns) ovat alkujaan jo 1970-luvulla rakennusarkkitehtuurissa käytettyjä ratkaisuja erilaisiin suunnitteluongelmiin. Karkeimmillaan ne ovat ottaneet kantaa esim. ikkunoiden sijoitteluun rakennuksessa.

Ohjelmistoarkkitehtuureihin suunnittelumallit toi legendaarinen Gang of Fourin (Erich Gamma, Richard Helm, Ralph Johnson ja John Vlissides) kirja Design Patterns: Elements of Reusable Object-Oriented Software. Kirjassa annettiin määritelmä suunnittelumallille ja dokumentoitiin joukko yleisesti olio-ohjelmointiin liittyviä suunnitelumalleja. Jokaisesta mallista dokumentoitiin sen rakenne, vaikutukset ja annettiin esimerkkitoteutus.

Gang of Fourin (GoF) kirjan jälkeen on julkaistu lukuisia muita kirjoja, joissa esitellään suunnittelumalleja tiettyihin ongelma-alueisiin, esim. verkko-ohjelmointiin, rinnakkaisuuden hallintaan tai reaaliaikasovelluksiin.

Suunnittelumalli on määritelmän mukaan käytännössä hyväksi havaittu ratkaisu tiettyyn usein esiintyvään ongelmaan. Tästä johtuen uusille teknologioille ei heti voi olla olemassa malleja, vaan ne syntyvät yhteisen kokemuksen karttuessa. J2EE on nyt saavuttamassa tätä tasoa.

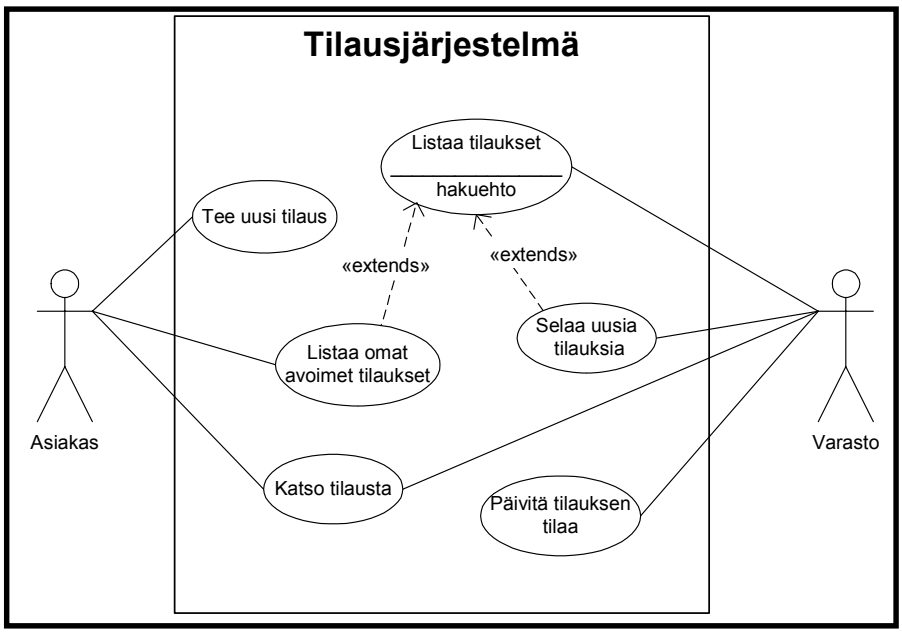
Samasta syystä suunnittelumallit ovat usein itsestäänselvyyksiä kokeneelle arkkitehdille. Tästä huolimatta parhaankin arkkitehdin kannattaa nöyrtyä opiskelemaan eri suunnittelumallit ja niille annetut nimet, sillä asioiden yhteinen nimeäminen helpottaa kommunikointia; monimutkaisen selityksen sijaan voidaan viitata yleisesti tunnettuun malliin sen nimellä.

Tällä hetkellä julkaistut J2EE-suunnittelumallit on jaettu kolmeen eri ryhmään: esityskerrokseen, liiketoimintakerrokseen ja integrointikerrokseen.

Esityskerroksen suunnittelumallit liittyvät JSP-sivujen rakenteeseen, tiedon välittämiseen servlettien ja JSP-sivujen välillä sekä servlettien rakenteeseen. Liiketoimintakerroksen suunnittelumallit liittyvät EJB-komponenttien käyttöön ja integrointikerroksen suunnittelumallit vastaavasti J2EE-sovellusten integrointiin taustajärjestelmien kanssa.

Esimerkkijärjestelmä

Tässä artikkelissa suunnittelumallien esittelemiseksi käytetään äärimmilleen yksinkertaistettua tilausjärjestelmää ja sen EJB-arkkitehtuuria. Esimerkin tilausjärjestelmällä yritysasiakas voi tilata tuotteita extranet-jär-



Kuva 1: Esimerkkijärjestelmän käyttötapaukset

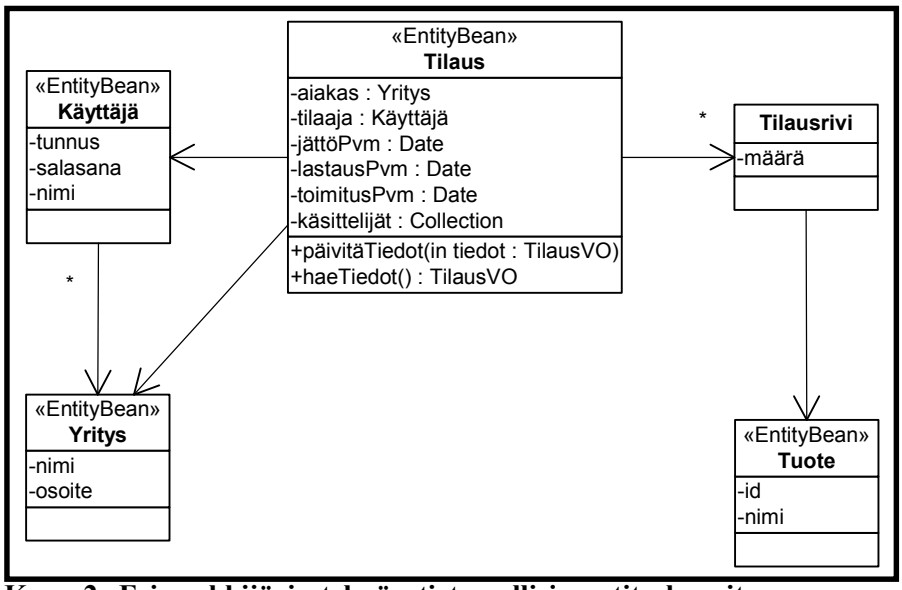
jestelmän kautta. Asiakkaan lisäksi järjestelmää käyttää varastotyöntekijä, joka käsittelee asiakkaiden tekemät tilaukset. Järjestelmän käyttöliittymäkerroksen (Servlet, JSP) arkkitehtuuria ei tässä esitetä.

Järjestelmässä on seuraavat käyttötapaukset (suluissa käyttäjärooli): Tee uusi tilaus (asiakas), selaa avoimia tilauksia (asiakas), selaa uusia tilauksia (varastotyöntekijä), merkitse tilaus käsitellyksi (varastotyöntekijä). Kts. kuva 1.

Käsitemalliin kuuluu seuraavat käsitteet: tilaus, tuote, yritys (asiakasyritys) ja käyttäjä (asiakasyrityksen edustaja). Tämän lisäksi tietomallissa on tilausrivi, joka yhdistää tuotteet tilauksiin.

Esimerkin suunnittelumallit

Seuraavassa esitellään esimerkissä käytetyt suunnittelumallit. Jokaisesta mallista kerrotaan lyhyesti sen vaikutuksista.



Kuva 2: Esimerkkijärjestelmän tietomalli ja entity beanit

Aggregate Entity (tai Composite Entity)

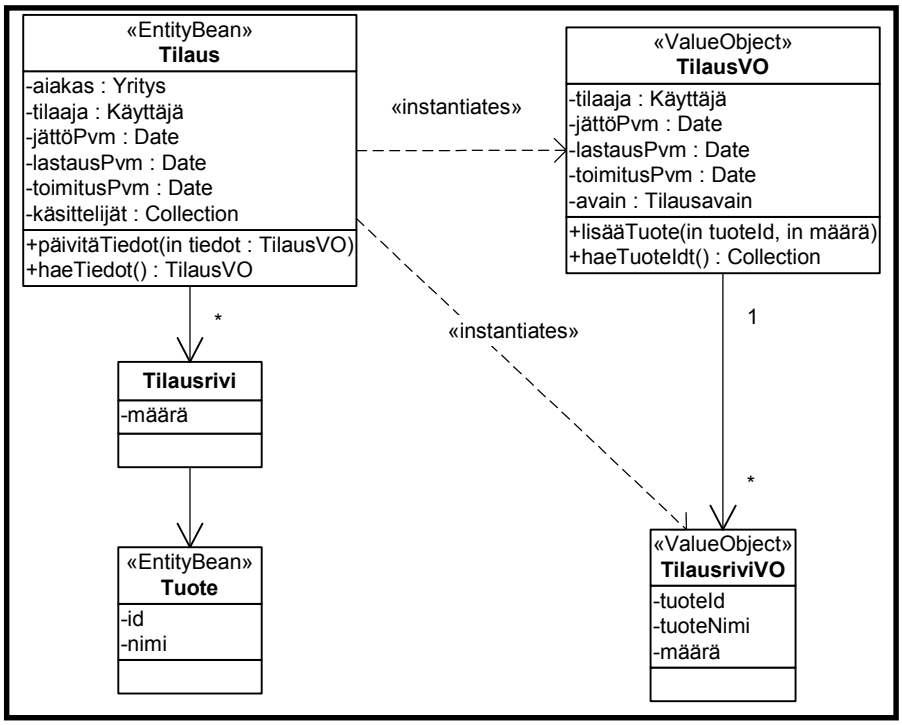
Yksi vaikeimmista arkkitehtuurillisista päätöksistä EJB-kerroksessa on päättää mitkä asiat kuvataan entity beaneina. Ongelma syntyy siitä, että entity bean voi kuvata hyvin eri tasoisia asioita käsitemallin asioista aina relaatiotietokannan taulun riveihin.

Nyrkkisääntönä on, että entity benin tulisi olla liiketoimintakomponentteja, eli kuvata käsitemallin tasolla olevia asioita. Nämä voivat toisinaan koostua useammista olioista tai tietokannan taulusta. Esimerkissämme tällainen on Tilaus ja sen sisältämät tilausrivit (Kuva 2).

Tilausrivit ja niiden elinkaari on erottamaton osa tilausta. Tilausrivi liittyy aina vain yhteen tilaukseen ja kun tilaus poistetaan, myös tilausrivit poistetaan. Lisäksi tilausrivejä käsitellään Tilaus-komponentin kautta, eikä koskaan suoraan.

Aggregate entity -mallin tarkoituksena on piilottaa tietomallin käsitteet, vähentää tarvittavien etämetodikutsujen ja ylläpidettävien entity beanien määrää.

Aggregate entity -mallissa yksi entity bean käsittelee useampia tietomallin olioita. Nämä lapsioliot eivät ole tavallisia entity beaneja, vaan niiden tallettaminen ja lataaminen tapahtuu pääkomponentissa. EJB 2.0:n myötä lapsioliot voidaan kuvata myös entity beanina, jolla on vain paikallinen rajapinta (Local Interface), eli sitä ei voida käyttää EJB-palvelimen ulkopuolelta.



Kuva 3: Value object -esimerkki

Value Object

Yksi yleisimmistä syistä huonosti suunnitellun J2EE-sovelluksen hitauteen on etämetodikutsujen suuri määrä. Etämetodikutsuja käytetään käyttöliittymä- ja EJB-kerroksen välisessä kommunikoinnissa, mutta J2EE:n hajautuvan arkkitehtuurin takia myös EJB-kerroksen sisällä. Etämetodikutsut ovat niiden aiheuttaman tietoliikenteen takia hitaampia, kuin vastaavat paikalliset kutsut.

Perustapauksessa entity beanin attribuutteja käsitellään sen get/set -metodeilla. Yleensä esityskerroksessa tarvitaan kuitenkin useampia attribuutteja kerralla, joten jokainen niistä pitäisi hakea omalla etämetodikutsulla.

Value object -mallin tavoitteena on vähentää etämetodikutsuja. Value objectin avulla yhden tai useamman entity beanin tiedot voidaan siirtää yhdellä kertaa, minkä jälkeen tiedon käsittely voidaan tehdä paikallisesti.

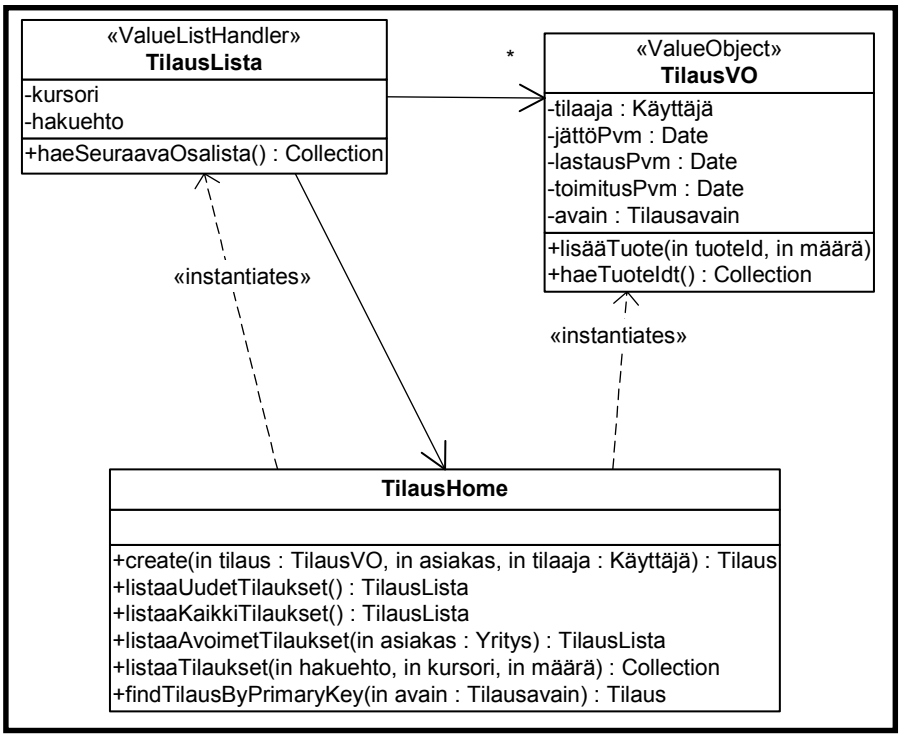
Jos tietoja päivitetään, voidaan value object välittää takaisin entity beanille, joka päivittää itsensä value objectin arvoilla.

Kts. kuva 3.

Esimerkissä value object -mallia käytetään siirtämään tilauksen ja tuotteen tiedot sovellukselle. Jos esimerkiksi tilauksen tietoja halutaan muuttaa, kutsuu sovellus ensin Tilauksen haeTiedot-metodia. Tämän tuloksena Tilaus alustaa uuden TilausVO-olion, joka pitää sisällään halutun tilauksen tiedot tilausriveineen (TilausriviVO). Alustettu TilausVO-olio sarjallistetaan ja siirretään sovellukselle, joka voi muokata sitä tarpeidensa mukaan. Muokattu TilausVO-olio kirjoitetaan takaisin tietokantaan kutsumalla Tilauksen päivitäTiedot-metodia.

Value List Handler

Value list handler -suunnittelumallin tarkoitus on tehdä erilaisten listausten teko nopeaksi ja joustavaksi. Varsinkin selainkäyttöliittymissä erilaiset listat esitetään sivu kerrallaan, jolloin ei ole järkevää siirtää koko listaa yhdellä kertaa EJB-kerroksesta Servlet-kerrokseen. Myöskään entity beanien käyttö ei ole perusteltua silloin, kun kyseessä on suuri lista, joka aihe-



Kuva 4: Value list handler -esimerkki

uttaisi suuren määrän entity beanien instantiointeja. Entity beanien sijaan listan sisältämät oliot ovat suoraan tietokantahaun perusteella instantioituja value objecteja. Nämä value objectit eivät välttämättä sisällä kaikkia vastaavan entity beanilta pyydetyn value objektin tietoja.

Esimerkissä value list handler -mallin ytimenä on TilausLista-luokka, jolta sovellus voi pyytää seuraavaa näytettävää listan osajoukkoa. TilausLista ylläpitää kursoria, joka osoittaa viimeiseksi näytettyyn value objectiin. TilausListalla on myös tiedossa alkuperäinen hakuehto, jotta se voisi hakea samoilla ehdoille seuraavan osajoukon.

TilausListalta saadut TilausVO-oliot poikkeavat Tilaus entity beanilta saaduista TilausVO:ista siinä, että ne eivät sisällä viittauksia TilausriviVO:ihin. Tämä johtuu siitä, että tilauslistauksissa ei tarvita tietoja tilausriveistä ja siksi tämän tiedon siirtäminen käyttöliittymäkerrokseen olisi turhaa.

Kts. kuva 4 ed. sivu.

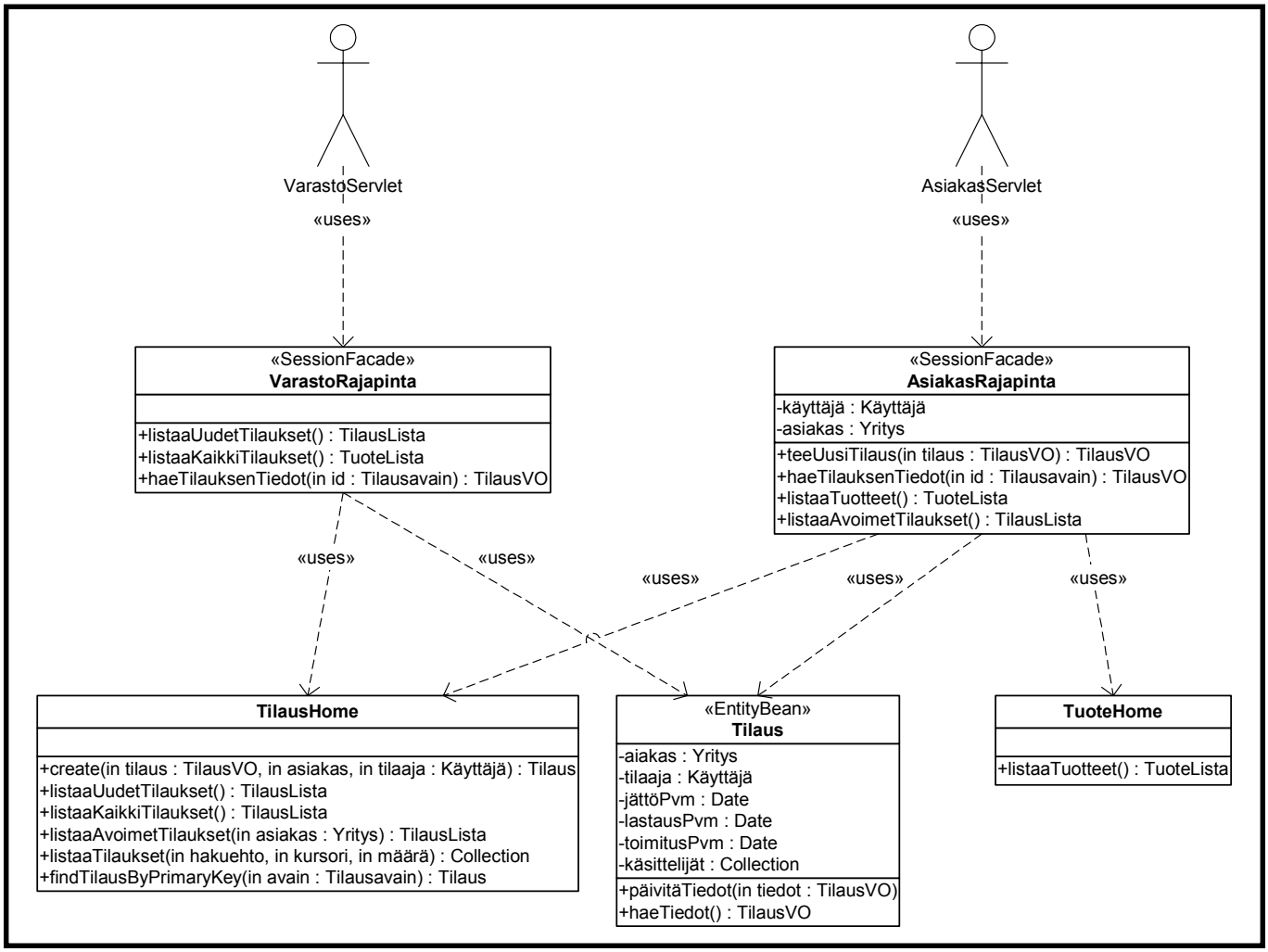
Value list handler -mallin käyttö nopeuttaa suurten hakujen tekemistä, koska koko hakutulosta ei välitetä käyttöliittymäkerrokselle, vaan tulosjoukon rajaus voidaan tehdä joko EJB-kerroksessa tai suoraan tietokannassa.

Malli mahdollistaa myös välimuistin käyttämisen hakujen yhteydessä. Jos tiettyä hakua suoritetaan peräjälkeen ja voidaan luottaa siihen, ettei tieto ole muuttunut, voidaan hakutulos palauttaa value list handleriin rakennetusta välimuistista.

Esimerkissä myös TuoteLista-luokka noudattaa value list hander -mallia.

Session Facade

Uudelleenkäytettävyyden edistämiseksi sovelluslogiikka jaetaan usein pieniin yksittäisiin metodeihin, joista kukin tekee jonkin selkeän pienen toiminnon. Tämä johtaa siihen, että yhden käyttötapauksen toteuttaminen



Kuva 5: Session facade -esimerkki

vaatii useita metodikutsuja useisiin eri EJB-komponentteihin.

Session facade -suunnittelumallin tarkoituksena on vähentää sovelluksen ja EJB-kerroksen välisiä kytköksiä ja siten edistää sovelluksen ylläpidettävyyttä ja muokattavuutta, sekä vähentää etämetodikutsujen määrää. Session facade -malli muistuttaa läheisesti GoF:n facade-mallia, mutta on tässä tapauksessa kohdistettu J2EE-ympäristöön.

Session facade -mallissa yhteen käyttötapaukseen liittyvät toiminnot kätketään yhden session beanissa olevan metodin taakse. Näin käyttöliittymäkerroksesta ei ole viittauksia EJB-kerroksen sisäisiin komponentteihin, vaan vain yhteen session facade -komponenttiin. Session facade -komponentteja voi olla useita ja niitä voi käyttää ryhmittelemään käyttötapaustason mukaisia metodikutsuja. Tällöin sovellukselle näkyvä rajapinta yksinkertaistuu. Ryhmittely voi tapahtua usealla eri tavalla, mm. käyttäjärooleittain tai aihealueittain.

Jos käytettävä EJB-alusta osaa tehdä saman alustan sisällä suoritetut kutsut lokaaleina (esim. EJB 2.0), myös tehokkuus kasvaa, koska useamman etämetodikutsun sijaan suoritetaan vain yksi kutsu (sovellukselta session facade -komponentille).

Esimerkissä on kaksi session facadea: VarastoRajapinta ja AsiakasRajapinta. Kumpikin facade kätkee taakseen yhteen käyttäjärajapintaan liittyvät toiminnot.

Kts. kuva 5 ed. sivu.

Service Locator

Service locator on tässä käsitellyistä suunnittelumalleista yksinkertaisin. Sen tavoitteena on kätkeä JNDI-hakuihin liittyvä monimutkaisuus ja enkapsuloida siihen liittyvät operaatiot yhden luokan sisälle. JNDI-viittauksien hakeminen on yksi J2EE-sovellusten kulmakiviä: viittaukset EJB-komponentteihin ja JMS-palveluun haetaan JNDI:n kautta.

Jokaiseen JNDI-hakuun liittyy joukko operaatioita, jotka ovat täysin samat riippumatta siitä mitä haetaan. Esimerkiksi EJB-komponentin hakuun liittyy seuraavat operaatiot: 1. JNDI-ympäristön luominen, 2. home-rajapinnan hakeminen, 3. home-rajapinnalta varsinaisen pavun pyytäminen. Tämän päälle tietysti tulee mahdollisista virheistä elpyminen tai ylipäänsä niiden yhdenmukainen käsittely.

Kaikkien em. operaatioiden sijoittaminen keskitettyyn paikkaan vähentää sovelluksen kompleksisuutta. Javan reflect-pakkauksesta löytyvien toimintojen avulla service locatatorin toteutukseta saa tehtyä melko geneerisen ja siron.

Service locatoriin voidaan myös sisäänrakentaa välimuistiominaisuuksia. Jos esimerkiksi tiettyyn papuun ei ole syytä hakea aina tuoretta viittausta, voi service locator -toteutus pitää vanhaa viittausta tallessa ja palauttaa sen sovellukselle.

Yhteenveto

Monien muiden teknologioiden rinnalla J2EE on vielä hyvin nuori. Tämän takia yleisesti tunnettujen suunnittelumallien valikoima on melko suppea. Nyt tunnetut suunnittelumallit kuitenkin kattavat jo yleisimmät J2EE-sovelluksen arkkitehtuurin suunnitteluun liittyvät ongelmat ja niiden tunteminen on hyödyksi sekä suunnittelun, että kommunikoinnin kannalta.

Suunnittelumallien sokea noudattaminen ei ole järkevää, eikä niitä kannata väkipakolla tunkea arkkitehtuuriinsa; kyseessä ei ole palapeli, jossa jokainen suunnittelumalli pitäisi saada mahtumaan mukaan, vaan työkalusetti, jota käytetään tarpeen mukaan.

Jos tuntee, että nykyiseen mallivalikoimaan olisi jotain annettavaa, kannattaa liittyä Sunin J2EE-patterns -keskustelulistalle. Listalla keskustellaan nykyisten mallien kehittämisestä ja täysin uusista malleista. Keskustelulistan osoite löytyy lähdeluettelosta.

Heikki Vesalainen ja
Matti Kokkola,
Mermit Business Applications Oy,
{vesalainen, kokkola}@mermit.fi

Lähteitä

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns, Elements of Reusable Object-Oriented Software, Addison-Wesley 1995

Deepak Alur, John Crupi, Dan Malks: Core J2EE Patterns, Best Practices and Design Strategies, Prentice Hall PTR, 2001

Douglas Schmidt, Michael Stal, Hans Rohnert, Frank Buschmann: Pattern Oriented Software Architecture vol 2: Patterns for Concurrent and Networked Objects, John Wiley & Sons 2001.

Sun Java Center J2EE Patterns, http://developer.java.sun.com/developer/technicalArticles/J2EE/patterns/

An interest list for Sun Java Center J2EE Pattern Catalog, http://archives.java.sun.com/archives/j2eepatterns-interest.html

J2ME: Konfiguraatiot ja profiilit

Juha Tamminen,
Sun Microsystems Oy,

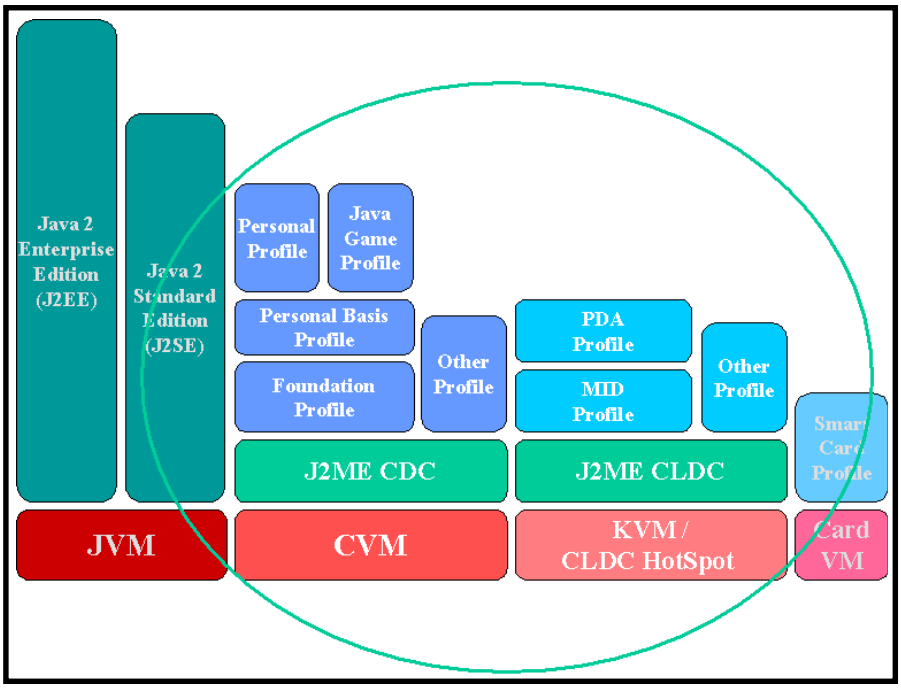
Java™ 2 Platform, Micro Edition (J2ME™) tarkoittaa Javaa pieniin laitteisiin. Se ei ole Java-ohjelmistopaketti tai spesifikaatio Java-ohjelmointikielelle matkapuhelimiin, taskutietokoneisiin (PDA), television set-top-laatikoihin tai muihin pieniin laitteisiin, vaan yleisnimi Java-kielelle kyseisiin laitteisiin.

J2ME jaetaan kahteen osa-alueeseen: konfiguraatioon (configuration) ja profiiliin (profile). Karkeasti ottaen konfiguraatio määrittelee vaatimukset Javan asentamiseksi laitteeseen ja profiili laitteen tarjoamat palvelut Java-sovellukselle. Tässä artikkelissa selvitellään tarkemmin, mitä konfiguraatiot ja profiilit ovat ja mistä ne koostuvat. Artikkelissa selvitetään myös mitä ovat lisäpalveluja tarjoavat Optional Package ja Building Block -luokkakirjastot.

Konfiguraatiot

Konfiguraatio määrittelee minimivaatimukset laitteen käytettävissä olevalle muistille ja suoritinteholle. Samalla se määrittelee spesifikaation laitteeseen asennettavalle Java-virtuaalikoneelle (JVM, Java Virtual Machine). Kuvassa 1 on esitetty ”Java-kartta”, jossa laitteistovaatimus pienenee siirryttäessä vasemmalta oikealle. Tarkastelun kohteena oleva J2ME sijaitsee kartan keskivaiheilla, kyseisessä kuvassa mittakaavaltaan suurennettuna.

Konfiguraatio määrittelee myös laitteelle jonkin koosteen käytössä olevista Java 2 Platform, Standard Edition (J2SE™) -luokkakirjastoista sekä muista tarvittavista luokkakirjastoista. Konfiguraatioita on tällä hetkellä kaksi, Connected Device Configuration (CDC) ja Connected Limited Device Configuration (CLDC). Konfiguraatioiden kehitys tapahtuu Java-yhteisössä eli JCP:n (Java Community Process) alaisuudessa.



Kuva 1. J2ME:n sijoittuminen “Java-kartalla”

Connected Device Configuration

CDC-konfiguraatio on tarkoitettu laitteille, joissa on vähintään 512 kilotavua ROM-muistia ja 256 kilotavua RAM-muistia ja jollakin tavalla toteutettu verkkoyhteys. Tyypillisesti CDC-konfiguraation toteuttavissa laitteissa on muistia yli 2 megatavua jaettavissa virtuaalikoneelle ja Java-sovellukselle. Tällaisia laitteita ovat esimerkiksi television set-top-laatikot, autojen navigointijärjestelmät ja korkeatasoiset PDA-laitteet. CDC:n määrittelemän Java-virtuaalikoneen on tuettava Java Virtual Machine Specification, 2nd Edition –spesifikaatiota täydellisesti. CDC:n määrittämää Java-virtuaalikonetta kutsutaan CVM:ksi (C Virtual Machine).

Connected Limited Device Configuration

CLDC-konfiguraatio on tarkoitettu CDC-konfiguraation ehdot täyttäviä laitteita pienemmille laitteille. Minimi muistimäärä on 160 kilotavua yhteensä sekä ROM- että RAM-muistille. Tyypillisesti CLDC-konfiguraation toteuttavissa laitteissa on muistia enemmän kuin 160 kilotavua, mutta alle 2 megatavua. CLDC-konfiguraation toteuttavien laitteiden verkkoyhteys on myös yleensä rajoitetumpi tai hitaampi kuin CDC:ssä. CLDC-konfiguraation ehdot täyttäville laitteille tarkoitettua Java-virtuaalikonetta kutsutaan KVM:ksi (Kilobyte Virtual Machine).

Toinenkin CLDC-konfiguraation määritykset toteuttava virtuaalikone

CLDC HotSpot on juuri saapunut markkinoille. Se on tarkoitettu laitteille, joissa on muistia yli 512 kilotavua. Tämä virtuaalikone tarjoaa huomattavasti KVM:ää paremman suorituskyvyn.

Profiilit

Profiili määrittelee laiteympäristön. Se perustuu tiettyyn konfiguraatioon ja lisää siihen sovellusten suorittamista varten välttämättömiä laitekohtaisia ohjelmistorajapintoja, kuten esimerkiksi käyttöliittymän (GUI, Graphical User Interface), sovelluksliittymän (API, Application Programming Interface) tai sovellusliittymän tiedon tallennukselle.

Foundation-profiili

Foundation-profiili (Java Specification Requests: JSR-46) on nimensä mukaisesti perusprofiili CDC-konfiguraation omaavalle laitteelle. Se laajentaa CDC-konfiguraation tarjoamia palveluja, mutta ei sisällä esimerkiksi palveluja käyttöliittymän tekoon. Foundation-profiilia käytetään pohjana muille CDC:n profiileille, ja siitä on julkaistu versio 1.0.1. Foundation-profiili sisältää TCP/IP-protokollan mukaiset Socket- ja ServerSocket-luokat verkkoyhteyden luomiseksi.

Personal-profiili

Ennen J2ME:tä julkaistua pieniin laitteisiin soveltuva Java-ympäristöä kutsuttiin PersonalJavaksi. PersonalJava soveltui muun muassa television set-top-laatikoihin, korkeatasoisiin PDA-laitteisiin sekä kommunikaattoreihin. J2ME:n julkaisun jälkeen PersonalJava on tarkoitus korvata yhdistelmällä, jossa on CDC-konfiguraatio, Foundation-profiili ja Personal-profiili (JSR-62).

Personal-profiili on tarkoitettu käytettäväksi yhdessä CDC-konfiguraation kanssa. Personal-profiili tarjoaa kehittäjälle applet-tuen selaimelle, AWT-luokkakirjaston (Abstract Window Toolkit) käyttöliittymien tekoon sekä tuen JavaBeans™-spesifikaation toteuttaville JavaBeans-komponenteille. PersonalJava-ympäristössä tehdyt sovellukset toimivat Personal-profiilin omaavassa ympäristössä. Personal-profiilista on julkaistu ehdotus ensimmäiseksi versioksi kesällä 2002.

Personal Basis -profiili

Personal Basis -profiili (JSR-129) on karsittu joukko Personal-profiilin tarjoamista luokkakirjastoista käytettäväksi yhdessä CDC-konfiguraation kanssa. Personal Basis -profiili sisältää supistetun käyttöliittymäkirjaston yksinkertaisten käyttöliittymien tekoon. Applet-tuki ei myöskään sisälly Personal Basis -profiiliin. Personal Basis -profiili sijoittuu Foundation- ja Personal-profiilien väliin. Personal Basis -profiilista on julkaistu versio 1.0 kesällä 2002.

Java Game-profiili

Lähitulevaisuudessa esiteltävä, CDC-konfiguraation päälle aseteltava Java Game -profiili (JSR-134) tarjoaa Java-ympäristön pelien kehittämiselle pieniin laitteisiin. Java Game -profiili perustuu Personal Basis -profiiliin, johon on lisätty muun muassa Mobile Multimedia API, Java 2D API ja Java 3D API.

Mobile Information Device -profiili

Tällä hetkellä suuren kiinnostuksen kohteena ovat erilaiset mobiilipalvelut ja niiden tuottaminen. Mobiililaitteissa käytettävä Java perustuu CLDC-konfiguraatioon ja MID-profiiliin (JSR-37, JRS-118). MID-profiilin perusvaatimuksena on 128 kilotavua muistia Java-sovellukselle, 32 kilotavua ajonaikaista muistia ja 8 kilotavua muistia tiedon tallentamiselle. Käyttöliittymä perustuu LCD-tekniikkaan ja on vähintään 96*54 pikselin kokoinen. Profiili vaatii myöskin kaksisuuntaisen verkkoyhteyden sekä mahdollisuuden tiedon syöttöön, esimerkiksi näppäimistön tai näytön välityksellä. Profiili tukee vain HTTP-protokollan mukaista verkkoyhteyttä. CLDC-ympäristöön on tällä hetkellä saatavissa esimerkiksi Enhydran kSOAP (Simple Object Access Protocol) niminen SOAP-toteutus, joka käyttää HTTP:tä kutsujen toteuttamisessa. Profiilin toteuttavassa ympäristössä ajettavaa Java-sovellusta kutsutaan MIDletiksi. Sama Java-ohjelma toimii eri valmistajien MID-profiilin omaavissa laitteissa. MID-profiilista on saatavissa versio 1.0.3.

PDA-profiili

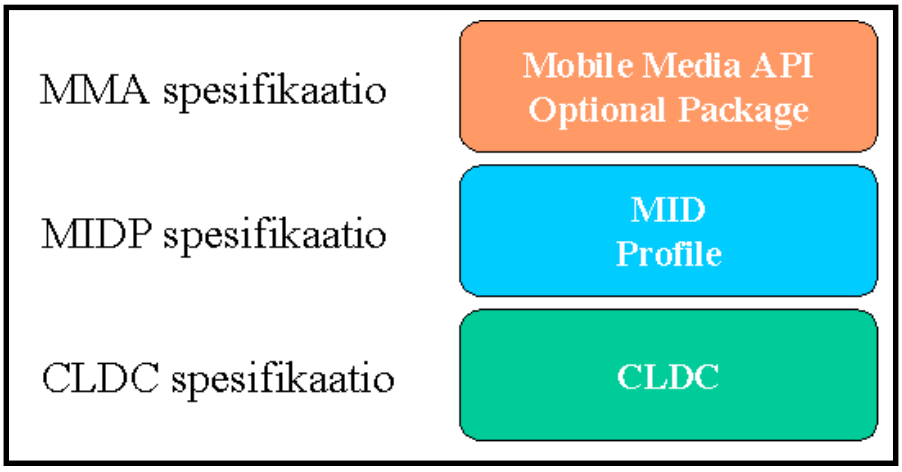
PDA-profiili (JSR-75) on tarkoitettu PDA-laitteiden Java-ympäristöksi CLDC-konfiguraation päälle ja vastaa toiminnoiltaan CDC-konfiguraation Personal-profiilia. Profiili sisältää kokonaan MID-profiilin. Lisänä MID-profiiliin on PDA-profiilissa AWT-käyttöliittymä, Personal Information Management -sovellusliittymä (PIM), sarjaporttiyhteys infrapunan tai kaapelin avulla sekä tiedostopalvelinyhteys hakemistoon tai muistikorttiin. PIM:n avulla Java-sovellus pääsee käsiksi PDA-laitteen osoite-, tehtävä- ja kalenterisovelluksiin. PDA-profiilin toteuttavassa ympäristössä ajettavaa Java-sovellusta kutsutaan PDAletiksi. PDA-profiilin virallisen version odotetaan tulevan valmiiksi syksyn 2002 aikana.

Optional Package

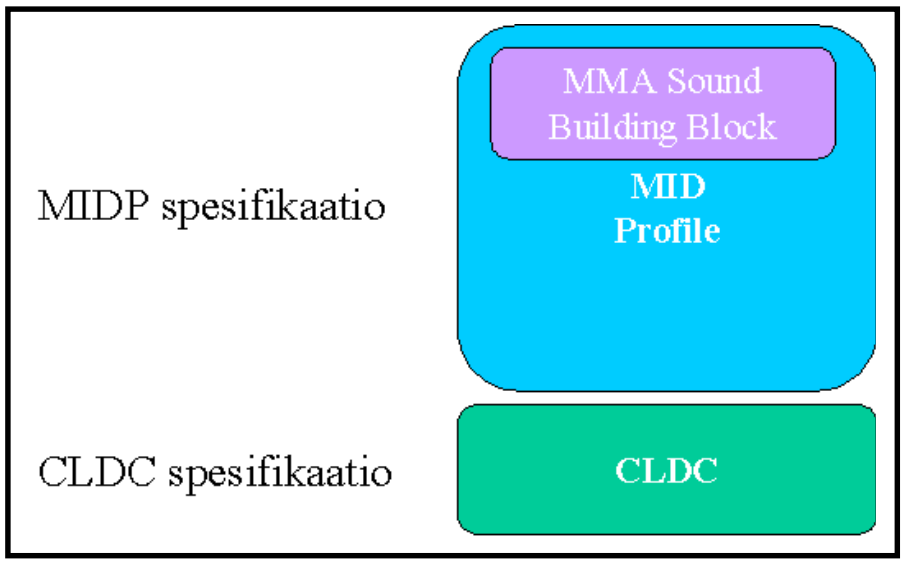
Optional Package -luokkakirjastot ovat profiilin päälle lisättäviä ohjelmistorajapintoja, joilla mahdollistetaan jokin nimenomainen, sovelluksen vaatima toiminta. Optional Packagen lisääminen laitteen Java-ympäristöön on laitteen valmistajan määriteltävissä. Optional Packagen spesifikaatiossa on määriteltävä kaikki mahdolliset riippuvuudet muihin ohjelmistorajapintoihin sekä riippuvuudet konfiguraatioihin ja profiileihin. Kuvassa 2 on kuvattu Optional Packagen sijoittuminen Java-karttaan. Kuvassa on Mobile Media API lisätty MID-profiilin päälle tarjoamaan kuvan ja äänen käsittelyyn tarkoitettuja Java-luokkia. Optional Packagen tarjoama toiminta voi olla esimerkiksi sellainen, että puhelin saadaan tärisemään, kun pelissä tapahtuu jotain sopivaa. Valmiita tai valmistumassa olevia Optional Packageja ovat

- Java Technology for the Wireless Industry (JSR-185)
- Mobile Media API (JSR-135)
- Location API for J2ME (JSR-179)
- SIP API for J2ME (JSR-180)
- Wireless Messaging API (JSR-120)
- Security and Trust Services API for J2ME (JSR-177)
- J2ME Web Services Specification (JSR-177)
- Mobile 3D Graphics API for J2ME (JSR-184)
- Bluetooth API (JSR-82)
  J2ME RMI (JSR-66)

RMI esiintyi aluksi omana profiilinaan, mutta on nyt julkaistu Optional Packagena. CDC-konfiguraation, Foundation-profiilin ja RMI Optional Packagen yhdistelmä on pienin kokoonpano, joka tukee Javan hajautettua oliotekniikkaa (Remote Method Invocation).



Kuva 2. Optional Packagen lisäys Java-ympäristöön.



Kuva 3. Mobile Multimedia API:n äänenkäsittelyyn tarkoitetuista luokista muodostettu Building Block lisättynä MID-profiiliin.

Building Block

Building Block on osajoukko olemassaolevasta ohjelmistorajapinnasta. Building Block sisältyy aina joko johonkin olemassaolevaan profiiliin tai Optional Packageen ja sen spesifikaatioon. Kuvassa 3 on lisätty MID-profiiliin Mobile Multimedia API:sta äänen käsittelyyn tarvittavat Java-luokat. Building Blockin riippuvuudet on esitettävä specifikaatiossa. J2ME Architecture Expert Group pitää listaa luoduista Building Blockeista ja karsii tulevia Building Block -pyyntöjä.

Juha Tamminen,
Java-arkkitehti,
Sun Microsystems Oy,
juha.tamminen@sun.com

Pullonkaulojen poistaminen J2EE-sovelluksista

Antti Tulisalo,
Sun Microsystems Oy

Muutaman vuoden takainen IT-nostatus sai aikaan monessa suhteessa sekä paljon hyvää että pahaa. Suuren yleisön kiinnostus IT-alasta toi mukanaan monta hyvää tuoteideaa ja uutta yritystä sekä edisti perinteistenkin alojen visioita kaikista niistä eduista, mitä uudenlainen kaupankäynti saattaisi tuoda mukanaan.

Kaikessa mukana ollut kiire, kasvu ja kansainvälistyminen toivat alalle kouluttamatonta ja kokematonta työvoimaa sekä suunnittelemattomia, dokumentoimattomia ja testaamattomia sovelluksia, joiden ongelmia yritettiin paikata palkkaamalla lisää työvoimaa ja ostamalla tehokkaampia laitteita.

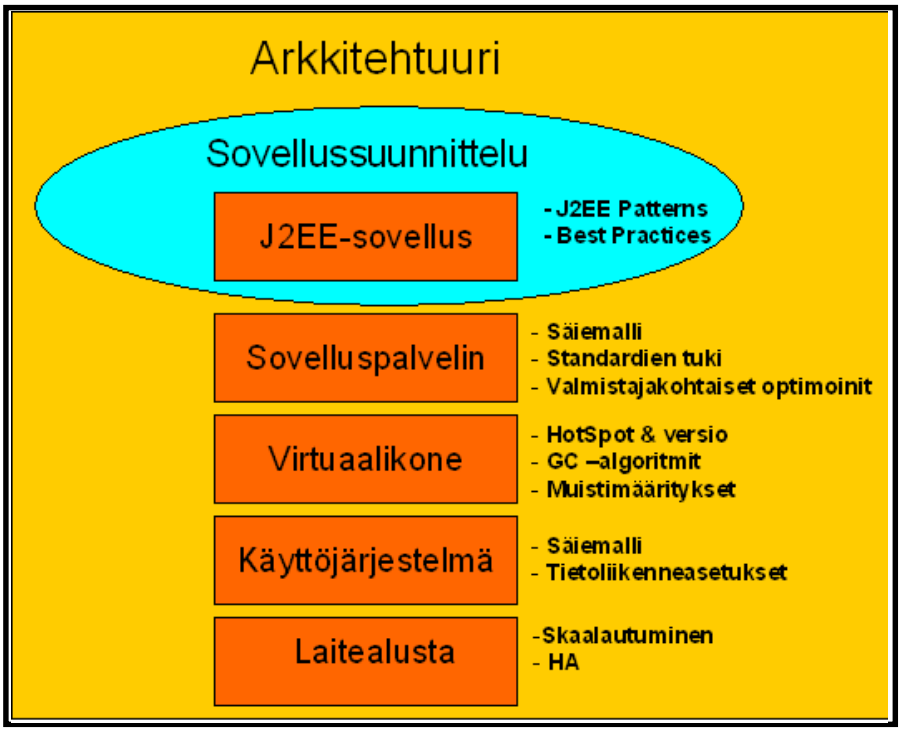
IT-alan viime aikojen ongelmat ovat kuitenkin kaikessa kurjuudessaan onnistuneet tuomaan jotain hyvääkin: järjestelmäkehityksen laatu on parantunut huomattavasti ainakin peilatessani nykytilaa omiin kokemuksiini arkkitehtuurien evaluoinnista sekä tuotantotulipalojen sammuttamisesta. Asiakkaat ovat huomattavasti entistä tarkempia siitä mitä heille toimitetaan, ja kustannussäästöt ovat johtaneet siihen, että jo olemassaolevista sekä työn alla olevista sovelluksista halutaan saada kaikki mahdollinen teho irti ilman kompromissejä laadun suhteen.

Olen useammin kuin kerran törmännyt tilanteeseen, jossa epävakaasti tai hitaasti toimivan sovelluksen ongelmat on laitettu jonkun uuden ajan kansantarun piikkiin: Java/EJB/J2EE on hidas, sovelluspalvelin/käyttöjärjestelmä on hidas, Java/käyttöjärjestelmä on rikki, jne. Useimmissa tapauksissa jäljet kuitenkin johtavat lopulta aivan muualle. Tarkoituksenani on tässä artikkelissa pyrkiä valoittamaan hieman sitä, miten J2EE-arkkitehtuurit tulisi nähdä useasta eri komponentista koottuna kokonaisuutena ja miten tämän kokonaisuuden osien suunnitelmallisella säätämisellä päästään helpoimmin tehokkuuden kannalta haluttuihin tuloksiin (Kuva 1.)

Arkkitehtuurin ja suunnittelun merkitys

Täysin suunnitelmaton työ johtaa usein katastrofiin. Poikkeuksiakin toki on, ja haluaisin vilpittömästi uskoa, että tällä hetkellä muodissa oleva, perinteistä suunnittelutyötä halveksiva extreme programming toimii kokeneiden koodaajien käytössä. Väittäisin kuitenkin, että perinteinen, tarkkaa sovelluskehitysprosessia kunnioittava tapa on silti oikea tie onnistuneiden raskaan sarjan J2EE-arkkitehtuurien toteuttamiseksi.

Jostain syystä varsinainen arkkitehtuurisuunnittelu tapaa yleensä jäädä hieman vajaaksi (tai sitten se puuttuu kokonaan), vaikka itse sovelluksen suunnittelu olisikin kunnossa. Arkkitehtuurisuunnittelun tarkoituksena on luoda pohja koko järjestelmälle sekä sille asetetuille arvoille (skaalautuva, turvallinen, hallittava jne).



Kuva 1. J2EE-alustan säädettävät osat

J2EE-arkkitehtuurisuunnitelman tulisi mielestäni sisältää mm. seuraavanlaisia asioita: mitä laitteistoa tarvitaan ja minkälaisessa konfiguraatiossa (vertikaalinen vai horisontaalinen skaalautuminen, turvallisuusseikat, korkea käytettävyys jne.) se toteuttaa asetetut arvot; laitteistossa ajettava käyttöjärjestelmä, virtuaalikone, sovelluspalvelin sekä karkean runko osista, joita J2EE:stä aiotaan käyttää; mahdollisesti myös karkean tason suunnittelumallirunko (kooste suunnittelumallien vuorovaikutuksista ja suhteista), jonka tarkoituksena on antaa suunnittelijoille lähtökohta ja näkemys siitä, miten tuleva sovellus sovitetaan arkkitehtuuriin.

J2EE:ssa on mahdollista tehdä asiat täydellisen pieleen silloinkin, kun spesifikaation sääntöjä noudatetaan sanatarkasti. Kokeneet J2EE-arkkitehdit ja -suunnittelijat ovatkin tutustuneet alalla vallitseviin J2EE:n parhaisiin toimintatapoihin ja valmiisiin suunnittelumalleihin (patterns), jotka helpottavat huomattavasti onnistuneen toteutuksen syntymistä sekä tulevia jatkokehitys- ja ylläpitotöitä.

Vaikka kaiken tekisikin kaikkien taiteen sääntöjen mukaan, jää sovellukseen aina parantamista: bugeja, uudelleensuunnittelua kaipaavia kohtia, arkkitehtuurin yksittäisten komponenttien virittelyä sekä muita vastaavia asioita, jotka saattavat kaivata työtä vaatimusmäärittelyjen tavoitteiden saavuttamiseksi.

Mistä aloittaa?

Järjestelmän kehityksen saavuttaessa pisteen, jossa ruvetaan testaamaan järjestelmän kykyä kestää vaatimusmäärittelyssä asetettuja suorituskykyarvoja, alkaa usein ihmettely siitä, miten koko järjestelmää pitäisi oikein tutkia. Koska ainakin suurin osa sovelluslogiikan virheistä on tässä vaiheessa jo (toivottavasti) korjattu, alkaa mielenkiinto kohdistua liiketoimintaprosesseja suorittavien kutsujen vasteaikoihin sekä maksimaaliseen käyttäjämäärään.

Ongelmana on yleensä tässä vaiheessa se, miten koko hommasta saisi fiksusti otteen. Ongelman ratkaisemiseksi on olemassa useitakin keinoja, lähinnä kaupallisten, sovellusten käytöstä profiloivien tuotteiden muodossa. Hyvä paketti on sellainen, jossa testattavaa alustaa voidaan kuormittaa yhdellä ohjelmalla, tarkistella toisella laitealustan ja virtuaalikoneen käytöstä (muistinkulutusta, roskienkeruuta ja synkronoitua koodia) ja profiloida kolmannella varsinaista J2EE-sovelluspalvelinta metodi- ja transaktiotasolla tarkkaillen vasteaikoja ja sisäistä työnkulkua.

Tässä vaiheessa tehdyt havainnot realisoidaan hyvin suunnitellulla korjauksilla testattavaan sovellukseen. Tämän jälkeen samaa testiä ajetaan uudelleen ja uudelleen, kunnes mahdolliset muistivuodot sekä pullonkaulat on poistettu ja järjestelmä on saavuttanut vaatimusmäärittelyjen suorituskykyarvot.

Laitealusta

Laitealusta on iso osa järjestelmän tehokkuutta. Tässä kerroksessa tehokkuuteen sekä moneen muuhun lisäarvoa järjestelmään tuovaan ominaisuuteen voidaan panostaa monin keinoin. Perinteisessä vertikaalisessa skaalauksessa alustaa voidaan tehostaa esimerkiksi muistin ja lisäprosessorien avulla. Vertikaalisessa skaalauksessa voidaan myös asettaa useampia koneita klusteriksi, jolloin skaalautuvuuden lisäksi myös saatavuus paranee. Pullonkauloja saattaa syntyä järjestelmän vaatimuksiin sopimattomasta tietoliikenneverkosta sekä riittämättömästä massamuistista.

Täytyy muistaa, että oikein suunniteltu laitearkkitehtuuri on aivan yhtä tärkeä kuin sovellusarkkitehtuurikin. Molempien pitäisi aina olla osana järkevää arkkitehtuurisuunnittelua. Laitealustan puutteiden korjaamisessa saa kuitenkin nopeasti paljon aikaan pelkällä rahalla, mikä ei usein pidä paikkaansa huonosti suunnitellun sovelluksen kanssa.

Käyttöjärjestelmä

Käyttöjärjestelmätason toimillakin voidaan melko helposti saada aikaan lisätehoa J2EE-sovelluksiin. Erilaisten vaihtoehtoisten säiekirjastojen käyttö saattaa saada aikaan huomattavaakin tehonlisäystä. Täytyy muistaa että Java ei välttämättä ole tehokkaimmillaan jokaisessa ympäristössä suoraan ”paketista ulos otettuna”.

Käyttöjärjestelmätasolla on olemassa myös useita eri komentoja käyttöjärjestelmästä riippuen, jotka helpottavat varsinaista ongelmanselvitystä. Esimerkkejä näistä ovat Unix-alustan truss- ja vmstat-komennot, joiden avulla voidaan tarkastella muun muassa käyttöjärjestelmäkutsuihin kuluvaa aikaa sekä prosessien run/block-jonoja.

Virtuaalikone

Eri laitealustoille ja niiden käyttöjärjestelmille on olemassa erilaisia Java-virtuaalikoneita useammilta eri valmistajilta. Nämä eroavat toisistaan Javan versioiden, muistinhallinnan,

roskienkeruualgoritmien sekä muiden erilaisiin ympäristöihin tarkoitettuihin optimointimenetelmien sekä ajomoodien puolesta. Yksi suosituimmista virtuaalikoneista on varmasti Sunin HotSpot JVM (Java Virtual Machine), jossa on omat ajomoodinsa asiakas- ja palvelinohjelmakäyttöön. HotSpotissa on myös lukuisia erilaisia virtuaalikoneen sisäisiä optimointimenetelmiä, muun muassa metodien inline-muunnoksia ja koodin ongelmakohtien natiivikoodikäännöksiä.

Tulevat JDK:t ja JVM:t sisältävät joukon profilointivälineitä sekä uuden rajapinnan, jonka avulla esimerkiksi sovelluksen muistinkulutusta on helppo seurata valmiin graafisen työkalun ansioista. Tulevat virtuaalikoneet sisältävät myös uusia yhtäaikaisia useamman säikeen roskienkeruumenetelmiä, jotka osaltaan auttavat sovittamaan Javaa mitä erilaisimpiin arkkitehtuureihin.

Virtuaalikoneen mustinkäyttöä säätelemällä voidaan vaikuttaa yllättävän paljon sovelluksen käyttäytymiseen. HotSpot-virtuaalikoneen muistimallia voi säätää erilaisilla parametreilla, mikä osaltaan vaikuttaa siihen miten roskienkeruu (turhien olioiden poisto muistista) käyttäytyy sovellusta ajettaessa. Esimerkiksi säätämällä niin sanottua eden-aluetta (muistialue, johon Javan oliot ”synnytetään”) suuremmaksi pystytään helposti pidentämään väliaikaisten olioiden roskien keruun välejä.

Virtuaalikoneen säätäminen vaatii kuitenkin asiantuntemusta itse virtuaalikoneesta, käytössä olevasta ympäristöstä sekä sovelluksen käyttäytymisestä.

Sovelluspalvelin

Sovelluspalvelinta säätämällä saadaan yleensä aikaan isojakin tehoeroja J2EE-ympäristöissä. Sovelluspalvelin saattaa mahdollistaa vaihtoehtoisen säiemallin käyttöönoton (multipleksaus), erilaisia tärkeiden olioiden poolausmahdollisuuksia sekä erilaisia J2EE:n rajoitteita kiertäviä, valmistajakohtaisia lisäarvoa tuottavia optioita ja komponentteja. Esimerkiksi Enterprise JavaBeans -komponenttien (EJB) valmistajakohtaisia asennuskuvaimia käyttämällä pystytään vaihtamaan Javan parametrinvälitysmallia, mahdollistamaan erilaisia batch update -menetelmiä, vaihtamaan isolation level -moodeja tilannekohtaisiksi, mahdollisuuden lokaaleihin rajapintoihin EJB 1.1:ssä sekä tehokkaamman EJB:n kyselykielen.

Kaiken avain on deklaratiivisuus. Jos esimerkiksi EJB-järjestelmä on tehty CMP-tekniikan (Container Managed Persistence) avulla, on järjestelmän koodauksen jälkeinen optimointi huomattavasti helpompaa kuin BMP- tekniikan (Bean Managed Persistence) avulla koodatuissa beaneissa. Syy tähän on se, että BMP-mallissa sovelluspalvelimen containerin on pakko noudattaa beanin elinkaarta, koska ohjelmoija on niiden sisällön koodannut itse. CMP-mallissa container huolehtii optimoinnista, ja silloin esimerkiksi EJB 2.0:ssa EJB 1.1:n perinteiset ”Dirty Marker” ja ”Lazy Loading” –strategiat toteutuvat automaattisesti olettaen tietenkin, että container on koodattu hyvin.

Sovellus

Viimeisenä muttei vähäisimpänä tällä listalla on varsinainen J2EE-sovellus. Sovelluskerroksessa tehdyt virheet voivat pilata kaiken muun työn tavalla, joka saattaa pahimmassa tapauksessa johtaa koko sovelluksen uudelleensuunnitteluun ja koodaamiseen. Koko homma perustuu lopulta onnistuneeseen sovellusarkkitehtuuriin.

Pienissä ja yksinkertaisissa sovelluksissa sovellusarkkitehtuurin virheet voidaan hoitaa raa’alla voimalla, mutta monimutkainen sovellus vaatii pätevien ihmisten tekemän huolellisen sovellussuunnittelun. J2EE-yhteisö on valtavan kokoinen, joten arkkitehdit ja suunnittelijat löytävät useampaan ongelmaan monenlaisia apukeinoja. Hyvänä esimerkkinä mainittakoon Sunin J2EE Patterns and Best Practices, joka sisältää valmiita suunnittelumalleja ja neuvoja olemassaolevien sovellusten ongelmapisteiden korjaamiseksi.

Suunnitelkaa, toteuttakaa, testatkaa ja korjatkaa. Onnistunut arkkitehtuuri on monen ihmisen yhteistyön tulos.

Antti Tulisalo,
Java-arkkitehti,
Sun Microsystems Oy

EJB ja EntityBean -komponentit

eli systeemityön käsitteellisestä kauneudesta ja käytännön raadollisuudesta

Jarmo Reunanen,
ReSolution Oy

Parin viimeisen vuoden aikana java on voimakkaasti vallannut alaa hajautettujen, laajojen sovellusarkkitehtuureiden toteutusalustana. Saavutus on merkittävämpi kuin ensi ajattelemalta tulee havainneeksi: tätä maailmankolkkaa ovat pitkään hallinneet mahtavat hirmuliskot, pitkälle kehittyneet sovelluspalvelimet, jotka – niin kuin kaikki liike-elämän voimakkaat ja laadukkaat toimijat – ovat pyrkineet suojaamaan markkina-asemaansa kehittymällä jatkuvasti monipuolisemmiksi, tehokkaimmiksi, skaalautuvimmiksi ja vikasietoisimmiksi. Tehokkaasti, laadukkaasti ja nopeasti tuotettujen järjestelmien hintana on kuitenkin usein ollut lukittuminen valittuun sovelluspalvelimeen. Sovelluspalvelimien toimittajat ovat markkinatoimijoita ja sellaisina harkitusti toteuttaneet klassista liiketalouden kielioppia: alalle tulon kynnystä on pyritty nostamaan palveluja monipuolistamalla ja omaa neuvotteluvoimaa lisäämään kasvattamalla asiakkaan tuotevaihtokustannuksia.

Tähän maailmaan putosi meteori kun Sun Microsystems julkisti EJB (Enterprise JavaBeans) arkkitehtuurinsa: EJB on arkkitehtuuri hajautetussa transaktioympäristössä ajettavien järjestelmien luomiseksi. Kuten perinteiset sovelluspalvelimet, EJB:kin lupaa vapauttaa sovelluskehittäjän transaktiojärjestelmien kanssa käytävästä mutapainista, hajautukseen ja koodin yhtäaikaiskäyttöön liittyvistä päänsäryistä, tietoturvan kiinteään koodaamiseen kompastumiselta jne.

Näiden lisäksi EJB-malli lupaa jotain ennen kuulumatonta: mahdollisuuden kirjoittaa sovelluksen ytimenä oleva liiketoimintalogiikka joukoksi uudelleen käytettäviä, yhdisteltäviä ja tuoteriippumattomia sovelluskomponentteja. Täydellisesti täyttyessään tämä lupaus muuttaa systeemityöteollisuuden rakenteita oleellisesti: lisäarvon tuottamisen painopiste siirtyy palvelinteknologian kehittämisestä sovelluspalveluiden tuottamiseen. On kuin autot ja polttoaine halpenisivat niin, että mökille olisi viimeinkin varaa hankkia kunnon laituri.

Tämän artikkelin tarkoituksena on tutkia, missä määrin EJB:n lupaama systeemityötaivas käytännössä voi toteutua. Erityisesti paneudun EntityBean-komponentteihin, jotka kokemusteni perusteella aihettavat samankaltaisia yllätyksiä hyvinkin erikaltaisissa projekteissa.

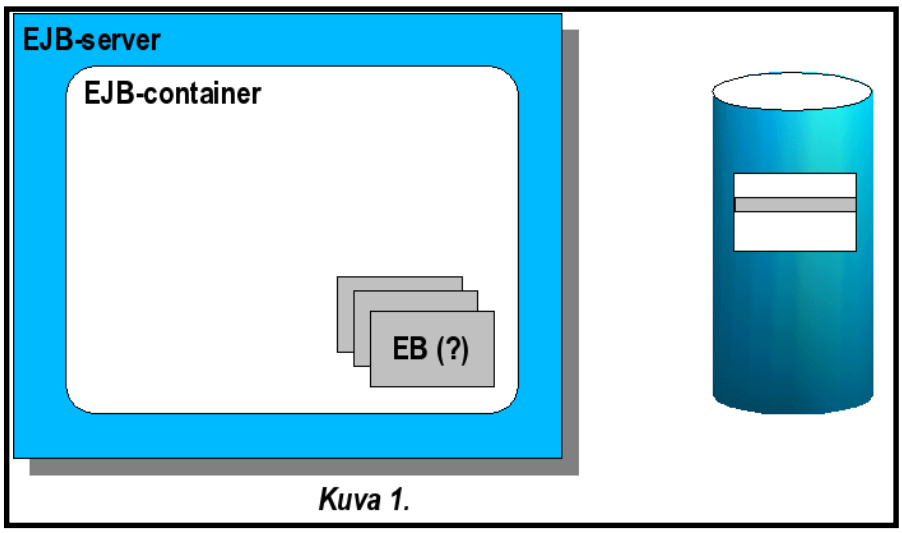
(Ennen varsinaista käsittelyä muutama tarkentava huomio: EJB on osa laajempaa J2EE-arkkitehtuuria, johon tässä yhteydessä ei ole tarkoituksenmukaista kajota. EJB itsessäänkin sisältää lukuisia määrittelyjä ja palveluja, joihin en viittaa ja joiden merkityksen omavaltaisesti ohitan; tarkoitus on riittävästi asioita yksinkertaistamalla tuoda esiin jotakin oleellista EJB:n käytöstä laajojen käytännön projektien toteutusalustana. Vältän myös EJB-terminologian suomentamista, vakiintuneita suomenkielisiä termejä kun ei ole. Muuta osaa artikkelista tällainen kielellisesti löysä attitude ei haperra.)

EJB-arkkitehtuurista

Koska EJB:n tehtävä on samankaltainen kuin sovelluspalvelimilla, yksi sen tasoista on mutkaton ja suoraviivainen: se toimii perinteisenä sovelluspalvelimena. Nämä palvelut tuottaa *EJB-server* -taso. Se huolehtii käyttöjärjestelmien, tietokantojen, transaktiojärjestelmien, viestinvälitysjärjestelmien ja muiden alustapalvelujen kanssa tehtävästä yhteistyöstä.

Varsinaiset sovelluskomponentit eivät kuitenkaan näe EJB-serverin tekemää, toteutusympäristöön ja -tuotteisiin voimakkaasti sopeutettua työtä. Ne elävät todellisuudesta vieraantunutta elämää, ja niin niiden tuleekin tehdä: jotta komponentit voisivat olla uudelleen käytettävissä, niiden ei tule ottaa kantaa transaktiotuotteisiin, hajautusprotokolliin, tietoturvapalveluihin jne. Sovelluskomponenttien täytyy pystyä luottamaan siihen, että niiden näkemä ja kokema maailma on aina samanlainen, riippumatta ympärillä olevasta todellisuudesta.

Tällaisen digitaalisen hampurilaisravintolan tarjoaa EJB-serverin päällä toimiva *EJB-container* -taso. EJB-container on sovelluskomponenttien ajoalusta, joka huolehtii sovelluskomponenttien elinkaaresta kehdosta hautaan - käynnistysarkistosta roskankerääjään. Se tarjoaa myös vakioidun rajapinnan EJB-serverin hallitsemien palveluiden käyttämiseen. EJB-container toteuttaa nurkumatta ja orjallisesti ideaalimaailman vaatiman illuusion: sovelluskomponentti voi keskittyä oman tehtävänsä hoitamiseen siinä uskossa, että se on ainoa mer-



Kuva 1.

kittävä älyllinen olento koko universumissa.

EJB-containerin huostaan voidaan asentaa eri tyyppisiä komponentteja, joista tässä tarkastelen EJB1.1 määrittelyn mukaisia EntityBean-komponentteja. EntityBean edustaa pysyvää, yleensä tietokantaan talletettavaa sovelluskomponenttia, ja luonteensa mukaisesti se on tärkeä kulmakivi useimmissa EJB-sovelluksissa. EJB-containerin idealismin vuoksi se joutuu myös kohtaamaan tosielämän raadollisuuden, valitettavasti, kuten alla olevasta huomaamme.

EntityBean-komponenttien käsittely

Jotta saisimme selkeämmän kuvan EntityBeanien käyttöön liittyvistä erityistekijöistä ja riskeistä, vuokratkaamme EJB-toimintavideo ja seuratkaamme hidastetuin kuvin EntityBean-komponentin elinkaarta ja vaiheita.

Lähdetään liikkeelle alkuteksteistä: oletetaan, että esimerkkijärjestelmämme on suunniteltu siten, että tietokannassa olevaa taulua vastaa yksittäinen EntityBean-luokka. Tämä on yksinkertaistettu, joskaan ei kohtuuton oletus. Hyvin normalisoidussa tietokantamallissa taulut edustavat itsenäisiä entiteettejä, eikä hyppäys oliomallin itsenäisiin käsitteisiin ole suuren suuri. Valitsemassamme tilanteessa jokainen taulussa oleva rivi vastaa yksittäistä EntityBean-objektia, joka sovellukseen saatetaan aktivoida.

Kun tämän kaltainen EJB-sovellus käynnistyy, on aika irrottaa ote perunalastupussista ja pysäyttää videonauhuri ensimmäistä kertaa. Tilanne on kuvan 1. mukainen:

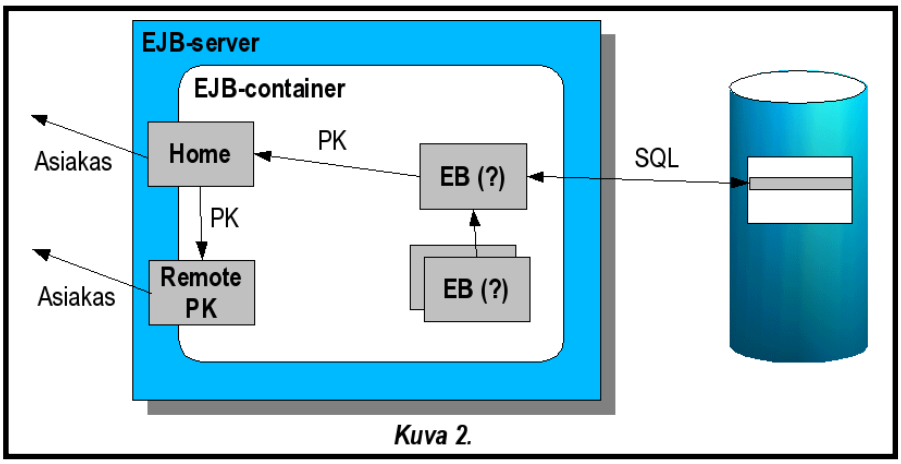
1. Tietokannan taulussa on joukko rivejä, jotka ovat autuaan tietämättömiä siitä, että niillä on erityismerkitys juuri käynnistetylle sovellukselle,

2. EJB-container puolestaan on tietoinen siitä, että jossakin vaiheessa tietokantarivejä vastaavia EntityBean-kompenentteja saatetaan käyttää. Jotta objektien luomiseen ja tuhoamiseen liittyvää työtä voitaisiin nopeuttaa, se luo valmiiksi joukon EntityBean-instansseja pahan päivän varalle. Nämä instanssit ovat kasvottomia, persoonattomia ja toistaiseksi hyödyttömiä, eräänlaisia zombeja, jotka on nostettu maan päälle myöhemmin koittavaa tehtävää varten.

3. EJB-container odottaa (kuten Harrison Ford elokuvan Blade Runner hienon loppurymistelyn alussa), että jotain alkaa tapahtua.

Aikanaan asiakassovellus, esimerkiksi www-palvelimen aluisuudessa ajettava java-servlet -komponentti, tarvitsee tietokannassa olevaa tietuetta. Se ottaa EJB-containeriin yhteyttä ja pyytää tätä luomaan tietuetta vastaavan EntityBean-objektin. Asiakassovellus pysähtyy odottamaan pyytämänsä palvelun täyttymistä. Filmimme alkaa taas pyöriä ja pysähtyy kuvan 2. kohdalle:

1. EJB-container reagoi asiakassovelluksen yhteydenottoon luomalla ns. *Home-objektin*. Home-objektin tehtävä on huolehtia tarvittavien EntityBean-komponenttien elinkaaresta.



Kuva 2.

Home-objektille välitetään asiakassovelluksen käyttämät hakuehdot, joiden perusteella luotava EntityBean-instanssi (tai laajempien hakujen yhteydessä joukko EntityBean-instansseja) voidaan yksilöidä.

2. Home-objekti ei yksin osaa tehdä tarvittavia tietokantahakuja, vaan se valitsee zombikartanosta valmiiksi luodun EntityBean-objektin ja antaa tälle tehtäväksi suorittaa tietokantahauat. Valittu EntityBean-komponentti tekee työtä käskettyä ja paluuarvona se välittää Home-objektille tietokannasta haetun, itsensä yksilöivän avaintiedon (PrimaryKey). EJB-containerin on nyt ratkaistava perustavaa laatua oleva ongelma. Entäpä jos haku on ollut muotoa "SELECT * FROM VERYBIGTABLE" - tulisiko kaikkia haettuja tietueita vastaavat objektit instantioida eläviksi, täysverisiksi objekteiksi. Tämä saattaisi syödä kohtuuttomasti käytettävissä olevia resursseja.

3. EJB-container päättää toimia toisin. Se palauttaa työnsä päättäneen zombin takaisin kartanoonsa odottamaan mahdollista uutta tehtävää ja instantioi EntityBean-komponenttia edustavan ns. *Remote-objektin* antaen sille saamansa avaimen tiedoksi. Asiakassovellukselle lähetetään rajapintaobjekti, jonka avulla Remote-objektin kanssa voidaan kommunikoida. Operaatio on valmis, rymistely hellittää hetkeksi ja EJB-container Remote-objekteineen jää odottamaan uutta tapahtumaa.

Remote-objektilla on kaksi päätehtävää: se toimii asiakassovelluksen rajapintana EntityBeanin palvelujen aktivoimiseksi, sen lisäksi se edustaa EntityBeanin identiteettiä EJB-containerin sisällä. On ikään kuin asiakassovelluksen viittaama EntityBean olisi jaettu kahteen osaan: Remote-objekti tietää luodun komponentin identiteetin (sillähän on hallussaan yksilöivä avain), mutta ei EntityBean-luokkaan koodattuja toimenpiteitä. Zombikartanon yksilöimättömät beankomponentit taas osaavat kaikki toimenpiteet, mutteivat tiedä omaa identiteettiään, jonka perusteella tietokantaoperaatioita voitaisiin tehdä.

On hyvin tärkeää havaita, että näin toimiessaan EJB-container välttää tietokannassa olevan tietosisällön kahdentamisen. Asiakassovelluksen yhteydenoton jälkeenkin tietokanta on ainoa paikka, jossa viitattujen objektien tietosisältö on tallessa. EJB-containeriin on ainoastaan luettu viitatun objektin yksilöivä avain. Ideaalimaailmassa näin tulee ollakin. Jos tietokannan tieto muuttuu EJB-sovelluksen ulkopuolella, tehdyt muutokset heijastuvat automaattisesti sovelluksemme EntityBean-objekteihin.

Aikanaan asiakassovellus päättää käyttää EntityBean-komponentin palveluja. Aikaisemman toimintansa tuloksena sillä on suora yhteys Remote-objektiin, jolla puolestaan on tarjottavinaan kaikki EntityBean-komponentin tarjoamat palvelut. Asiakassovellus antaa tehtävän Remote-objektin suoritettavaksi. Elokuvassamme on jälleen vuorossa toimintakohtaus, jota voimme kotivideoilla katsoa kuva kuvalta, kuten kuva 3. osoittaa:

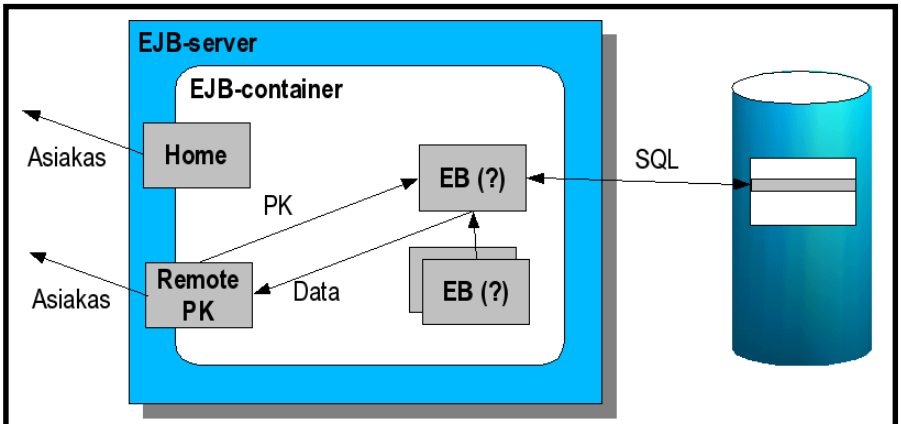
1. Remote-objektin on suoritettava EntityBean-komponentin palvelu ja välitettävä sen tulokset asiakassovellukselle. EJB-containerissa ei kuitenkaan ole palvelun suoritamiseen vaadittua tietoa, tieto on vain tietokannan taulussa.

2. Tiedon hakemiseksi EJB-container aktivoi jälleen beanvarastosta yhden aikaisemmin luoduista objekteista ja välittää tälle Remote-objektin tiedossa olevan avaimen. Avaimen perusteella aktivoitu bean-komponentti tekee tietokantahaun ja päivittää oman tietosisältönsä.

3. EntityBean-komponentti voi nyt suorittaa pyydetyn palvelun, jonka paluuarvo välitetään edelleen avainta hallitsevalle Remote-objektille.

4. Remote-objekti palauttaa tiedon asiakassovellukselle, jonka matka jatkuu koodissa eteen päin - pyydetty palvelu on suoritettu.

5. EJB-container on nyt uuden ongelman edessä: viitattu EntityBean-komponentti on edelleen aktivoituna ja täysivoimaisena ajo-



Kuva 3.

ympäristössä. Sen mahdollisesta tulevasta käytöstä ei kuitenkaan ole varmuutta. Lisäksi tietokanta voi päivittyä sovelluksemme ulkopuolella. EJB-container päätyy tylyyn johtopäätökseen: jotta EntityBean-komponenttimme tietosisällön eheys voidaan varmistaa, se täytyy välttämättä ladata uudestaan seuraavan aktivoinnin yhteydessä.

6. Vaivalla henkiin puhallettu EntityBean-komponenttimme on näin tarpeeton ja se tuhotaan tai palautetaan objektivarastoon, zombiksi zombien joukkoon. EJB-container rauhoittuu odottamaan seuraavaa operaatiota.

Asiakassovellus on tietämätön tästä käyttämänsä objektin masentavasta kohtelusta. Sillä on edelleen yhteys Remote-objektiin, se kuvittelee komponentin olevan hengissä, ja on hyvinkin todennäköistä, että se käyttää yhteyttään uudelleen. Kärjistetysti sanottuna: jokainen asiakassovelluksen tekemä EntityBean-viittaus aiheuttaa fyysisen tietokantahaun, uudelleen ja erikseen, huolimatta siitä, että asiakassovellus saattaa viitata jo kerran aktivoimansa EntityBean-komponentin muuttumattomaan tietosisältöön. (Tämä on kärjistys, se on myönnettävä. Tarkasti saivarrellen elokuvamme lopputeksteissä olisi maininta: "No transactions were harmed during the shooting of this film". Toisin sanoen, jokainen alkava tietokantatransaktio vähintäänkin aiheuttaa datan uudelleen lukemisen tietokannasta. Käytännössä lopputulos on hyvin lähellä yllä kuvattua.)

Johtopäätöksiä

Mikä on lopputulos? Miksi EJB-containerin sisäisen toiminnan tutkimiseen on käytettävä kaikki tämä vaiva, kun EJB-arkkitehtuuri nimenomaan lähtee oletuksesta, että sovelluskehittäjällä ei ole tarvetta tietää pinnan alla olevista yksityiskohdista?

Vastaus on selkeä. EntityBean-komponenttien sielunmaailma lähtee ideaalisen kauneuden ja eheyden ajatuksesta. EJB-container takaa, että kaikki EntityBean-komponentteja käsittelevät operaatiot tehdään hajautettujen transaktioiden maailmassa. Näihin transaktioihin voivat osallistua myös EJB-sovelluksen ulkopuoliset tahot. Sovellusten eheys varmistetaan ilman ohjelmoijan omaa koodia. Kaiken hintana on kuitenkin lukuisten, useimmiten optimoimattomien tietokantahakujen toistuva suorittaminen. Suoraan EJB-arkkitehtuuriin kirjoitetut sovellukset, jotka käyttävät runsaasti EntityBeaneja, eivät skaalaudu ajan vaatimusten mukaisesti – tietokantaoperaatiot ja EJB-containerin objektien aktivointiin kuluva työ syövät suorituskykyä kohtuuttomasti.

Tämä ei ole poikkeuksellinen eikä rienaava väite. Itse asiassa havainto on yleisesti hyväksytty ja kuuluu osana siihen hintaan, joka korkean abstraktiotason sovellusarkkitehtuureista on maksettava. Tilanne ei kuitenkaan ole niin synkeä kuin Harrison Fordin ilme Blade Runnerin loppukohtauksessa (tällä tarkoitan tietenkin Director's Cut -versiota).

EntityBeanin yllä kuvattu problematiikka on niin yleisesti hyväksytty, että sille on jopa selkeä, toimiva ja standardoitu ratkaisumalli. Ratkaisun periaate on yksinkertainen: EntityBeaneja tulisi käyttää silloin, aina silloin, ja vain silloin, kun niiden käytöstä on ratkaisevaa etua. Tällaisia käyttökohteita ovat erityisesti:

1. sovellusobjektit, joiden tila on talletettava pysyvästi, ja joiden toiminta ratkaisevasti riippuu pysyvästi talletetun datan tilasta,
2. sovellusobjektit, jotka voidaan ja jotka tulee kaikkina ajan hetkinä yksilöidä jonkin yksikäsitteisen avaintiedon perusteella,
3. sovellusobjektit, joiden täytyy tukea usean asiakkaan yhtäaikaista, transaktioihin sidottua käyttöä, sekä
4. sovellusobjektit, joiden käsittelyn on onnistuttava korkean tason ad-hoc -kyselyillä (EJB2.0)

Nämä (ja joukko muita) suositeltavia käyttökohteita ja suunnittelumalleja on standardoitu, jotta pyörää ei tarvitsisi keksiä uudelleen. Toimintaelokuvissa seinään törmääminen saattaa olla harkittua, mutta kriittisten sovellusten suunnittelussa siihen ei ole varaa. Sun Microsystemsin laaja J2EE-arkkitehtuurimalli sisältää tarkkaan dokumentoituja suosituksia (J2EE Blueprints), joiden avulla käytännön systeemityölle annetaan eväät riittävän suorituskykyisen lopputuotteen rakentamiseksi.

J2EE on tämän päivän systeemityön evaluoiduimpia malleja, ja mikäli EJB-elokuvamme jatko-osa kiinnostaa, sen traileriin kannattaa tutustua osoitteessa http://java.sun.com/j2ee.

Jarmo Reunanen,
konsultti,
ReSolution Oy

ReSolution on java- ja unix-konsultointiin erikoistunut, omistamani yritys. Tarkempia tietoja yrityksestä ja itsestäni saitilta www.resolution.fi).

Kokemuksia SCEA-tutkintoon valmistautumisesta

Kari Marttila,
Tietoenator Oyj

Kerron tässä artikkelissa hyvän ystäväni ja arvostamani Java-gurun Simo Vuorisen pyynnöstä joitakin kokemuksia SCEA (Sun Certified Enterprise Architect) –tutkintoon valmistautumisesta. En kerro artikkelissa mitään testisalaisuuteen liittyviä yksityiskohtia testistä vaan kerron yleisellä tasolla erilaisia kokemuksia tutkintoon valmistautumisesta (Sunin Suomen toimisto on tarkistanut jutun testiä koskevat osat). Tässä artikkelissa testistä kertomani yksityiskohdat on saatavilla Sunin k.o. tutkinnon kotisivuilta ja Sunin k.o. tutkinnon Certification Success Guide:sta.

Aloin syksyllä 2001 TietoEnator / Public sector Java-tukiryhmän vetäjän ominaisuudessa ideoida muutaman yksikköni asiantuntijan ja esimiesteni kanssa suunnitelmaa, jolla saataisiin yksikköön noin neljä Sunin sertifioimaa arkkitehtia. Yksikön tarpeena oli tietysti saada konkreettista näyttöä osaamisesta tunnustetun ja kansainvälisesti standardoidun sertifikaatin muodossa. Sun itse toteaakin Certification Success Guide:ssaan, että yrityksen kannattaa sertifioida työntekijöitään ammattitutkinnoilla, koska ”taidot ja tiedot, jotka testataan sertifiointiprosessin aikana, ovat samoja, joita työntekijä tarvitsee käytännön työssään, kun pyritään nopeampiin aikatauluihin, parempaan tuottavuuteen, korkeampaan laatuun ja korkeampaan työtyytyväisyyteen”. Ja työntekijän ei kannata unohtaa sitä, että sertifikaatti on kuitenkin henkilökohtainen eli sitä voi käyttää itse konkreettisena todisteena työmarkkinoilla omasta osaamisestaan.

Suunnittelu eteni niin, että yhtiö lupasi maksaa tutkintoon valmentavan kouluttamisen (kurssin, kirjat ja testimaksut). Aikaa tutkinnon suorittamista varten varattiin kurssilla käymiseen, kirjallisuuteen tutustumiseen, testeissä käymiseen ja harjoitustyön tekemiseen. Kun nämä asiat oli sovittu yrityksen kanssa, otimme yhteyttä Sun Microsystems Koulutuspalveluiden koulutusmyyjä Stefan Nygårdiin, jonka kanssa räätälöimme tutkintoon valmistavan kurssin. Kurssi räätälöitiin meitä varten Sunin kursseista SL-425 Architecting and Designing J2EE™ Applications ja SL-500 J2EE™ Patterns. Luennoitsijaksi Stefan Nygård sai Jarmo Reunasen, jolla on useiden vuosien kokemus vaativien arkkitehtuurien suunnittelusta ja jota SUN on paljon käyttänyt vaativien kurssiensa kouluttajana. Koska kurssin organisointia varten oli nähty melkoisesti vaivaa, niin juttelin esimieheni kanssa, että yritämme saada kurssille kaikkiaan noin kymmenen henkilöä liiketoimintaryhmämme eri yksiköistä. Kurssille saatiinkin loppujen lopuksi 11 TE-läistä. Kurssi toteutettiin viisi-päiväisenä kurssina helmikuussa 2002. Kurssi koostui luennoista ja erilaisista harjoitustöistä. Käsittääkseni kaikki kurssilaiset olivat varsin tyytyväisiä kurssiin. Omasta mielestäni kurssi oli parhaimpia kaupallisia alan kursseja, joita olen käynyt. Vauhti oli riittävän nopea ettei ehtinyt pitkästyä, kurssimateriaali oli erinomaista ja kurssin vetäjä oli todella kokenut arkkitehtuuriasioissa ja erittäin hyvä pedagogi. Tässä yhteydessä täytyy samalla kiittää Stefania, joka oli todella aktiivinen ja avulias kurssin organisoimisessa, ja Jarmoa hyvästä opetuksesta.

Kurssin jälkeen minulla oli joitakin kiireellisiä työtehtäviä, minkä takia en ehtinyt heti aloittaa tutkinnon suorittamista. Varasin tutkinnon ensimmäisen osan suorittamista varten ajan 22.3.2002, jolla viikolla kertasin n. 1 ½ päivää kurssin materiaalia. Tässä yhteydessä on ehkä syytä hieman valottaa itse tutkinnon testejä. Tutkinnon varsinainen virallinen nimi on SUN CERTIFIED ENTERPRISE ARCHITECT FOR JAVA[tm] 2 PLATFORM, ENTERPRISE[tm] EDITION ja sen virallinen kotisivu on http://suned.sun.com/US/certification/java/java_archj2ee.html. Sivulla kerrotaan, että tutkinto koostuu kolmesta testistä, jotka pitää suorittaa seuraavassa järjestyksessä:

- **Step 1**: Sun Certified Enterprise Architect for J2EE Technology, Knowledge-based Multiple Choice Exam (310-051) ($150)
- **Step 2**: Sun Certified Enterprise Architect for J2EE Technology, Assignment (CX-310-300A) ($250)
- **Step 3**: Sun Certified Enterprise Architect for J2EE Technology, Essay Exam (310-061) ($150) VA[tm] 2 PLATFORM, ENTERPRISE[tm] EDITION

Ensimmäinen testi on siis monivalintatesti, jossa mitataan tietämyksen teoreettista tasoa. Testejä järjestää Suomessa Auktorisoidut Prometric-testikeskukset (itse kävin Lan Group Finlandin testipaikassa Espoon keskuksessa, joka on yksi tällainen testikeskus). Monivalintatehtäviä on 48 ja niiden suorittamiseen on aikaa 75 minuuttia. Testi on teknisesti saman-

kaltainen tietokoneohjattu testi kuin SCJP-testikin (Sun Certified Java Programmer). Edellä mainitussa Certification Success Guide:ssa mainitaan, että testin objektiiveina on seuraavat aihealueet: 1. Common Architectures, 2. Legacy Connectivity, 3. Enterprise JavaBeans, 4. Enterprise JavaBeans Container Model, 5. Protocols, 6. Applicability of J2EE Techonology, 7. Design Patterns, 8. Messaging, 9. Internationalisation ja 10. Security.

En tietenkään ala kertoa tässä testin yksityiskohtia, mutta voi mainita joitakin yleisiä asioita, joita kannattaa ottaa huomioon testiin valmistautumisessa:

Kannattaa lukea hyvin testin objektiivit (Certification Success Guide).

- Kannattaa valmistautua testiin huolellisesti lukemalla muutama hyvä J2EE-kirja (esim. em. kurssien kurssimateriaali) sekä jokin hyvä kirja UML:sta sekä GoF:n Design Patterns. Lisäksi kannattaa selata Internetistä hyviä SCEA-sivuja, joissa on kerrottu hyviä lähteitä esim. protokollista, legacy connectivitystä jne.
- Testissä kannattaa muistaa kellottaa omaa suoritustaan. Esim. 48 tehtävää / 75 minuuttia -> esim. 48 tehtävää / 60 min + 15 varalle -> n. 25 tehtävää / 30 min -> n. 13 tehtävää / 15 min -> eli hieman yli minuutti per tehtävä.
- Kellotin koko testin ajan omaa etenemistäni ja lopussa jäi 7 minuuttia ylimääräistä aikaa, jonka käytin siihen, että tarkistin tehtävät, ettei tullut huolimattomuusvirheitä.

Olen itse tehnyt useita vuosia Javalla lähes pelkästään server-suunnittelua ja –ohjelmointia, joten useimmat asiat (EJB, JNDI, JMS…) olivat valmiiksi aika tuttuja. Luin kuitenkin varmuuden vuoksi aihealueen melko huolellisesti läpi ja sainkin testistä 97 %:a oikein, mikä olikin huomattavasti paremmin kuin vuotta aikaisemmin suorittamani SCJP-tutkinto, josta sain huomattavasti vähemmän pisteitä lähinnä sen takia, etten laiskuuttani viitsinyt lukea testin graafiseen käyttöliittymäohjelmointiin liittyviä asioita, vaan suoritin testin lähinnä server-ohjelmointikokemukseni perusteella.

Tästä pääsemmekin mielenkiintoiseen ilmiöön siitä, että useimmat ihmiset eivät näytä hahmottavan asioiden todennäköisyyksiä. Esimerkiksi henkilö, joka kommentoi erilaisia monivalintatestejä sanomalla, ”että sehän on vain monivalintatesti, siitähän pääsee arvaamalla läpi”, ei todennäköisesti ole koskaan vaivautunut opiskelemaan todennäköisyyslaskennan alkeita. Laskin huvin vuoksi, että jos meillä on testi, jossa on esim. 48 tehtävää, joiden muoto on ”Valitse seuraavista 7 vaihtoehdosta 3 oikeaa”, niin tällaisen testin suorittaminen puhtaasti arvaamalla on käytännöllinen mahdottomuus. Yhdessä tehtävässä on 35 eri vaihtoehtoa (n!/k!(n-k)!), joten todennäköisyys, että suorittaa yhden tehtävän arvaamalla, on 1/35 eli n. 0.0268 eli n. 3%:n luokkaa. Eli jos kaikissa tehtävissä on odotettavissa sama edellä mainittu kombinaatiovaatimus, niin todennäköisyys, että arvaa kaikki 48 tehtävää oikein on 0.0286^48 = 7.7*10^(-75) (mikä on huikean pieni todennäköisyys). SCEA-tutkinnon 1-osassa läpimenovaatimus on 68% oikein eli ainakin 33 tehtävää 48 tehtävästä pitää olla oikein eli jos kaikki tehtävät olisivat edellisen kombinaatiovaatimuksen kaltaisia, niin todennäköisyys, että saa vaaditut 68%:a tehtävistä arvaamalla oikein olisi laskettavissa kaavasta: P(ainakin k onnistunutta suoritusta) = $S^{n}_{i=k}(^{n}_{i})p^{i}q^{n-i}$, n = toisistaan riippumattomien testien lukumäärä, tässä = 48, k = onnistunut testisuoritus, tässä halutaan todennäköisyydet tilanteista i=[33,48], p = yhden testin onnistumisen todennäköisyys = 1/35, q = yhden testin epäonnistumisen todennäköisyys (= 1-p). Tässä P(33) = n. 8*10^(-40). Todennäköisyys on huomattavasti pienempi kuin arkijärjellä voisi kuvitella. Jotta paremmin hahmottaa em. todennäköisyyttä, voi todeta, että on esimerkiksi huomattavasti todennäköisempää samana päivänä sekä voittaa U.S. State Lotteryn päävoitto että kuolla salamaniskuun (todennäköisyys 1/2^55 = 2.8*10^(-17), Schneier: Applied Cryptography, s. 18) kuin päästä läpi arvaamalla edellä mainitusta koetilanteesta. Matematiikan lisensiaatti Risto Pohjosmäelle kiitokset todennäköisyyslaskujen validoinnista.

Tämän jälkeen meni taas pari kuukautta muissa työkiireissä, minkä takia en ehtinyt aloittaa tutkinnon 2-osan eli harjoitustyön tekemistä. Sain lopulta kesäkuussa kalenteriin aikaa kolme päivää harjoitustyön tekemistä varten. Harjoitustyön saa ladata Sun:n sivuilta kun 1-osa on mennyt hyväksytysti läpi ja sen tekemisellä ei ole aikarajaa. Harjoitustyö on pienehkö arkkitehtuuri- ja suunnittelutyö, kuten mainitaan em. Certification Success Guide:ssa, ja sen tarkoituksena on demonstroida, että osaa käytännössä soveltaa taitojaan suunnittelijana ja arkkitehtina. Harjoitustyössä annetaan hyvät ohjeet, mitä dokumentteja ratkaisussa vaaditaan (diagrammeja: luokkakaavioita, sekvenssikaavioita, deployment-kaavioita… sekä erilaisia selityksiä tehdyistä ratkaisuista). Diagrammit saa piirtää haluamallaan työkalulla kunhan ne ovat UML-notaation mukaisia. Tämän jälkeen harjoitustyöstä pitää tehdä ohjeiden mukaan joukko HTML-sivuja, joihin kuvat pitää linkata, ja lopuksi jarrata koko paketti yhdeksi jar-tiedostoksi. Harjoitustyön ohjeissa on tarkkaan kuvattu tarkistuskriteerit ja näistä kriteereistä pitää 70% täyttää, jotta harjoitustyö menisi läpi.

Harjoitustyön jättämisen jälkeen saa varata ajan tutkinnon 3-osan suorittamista varten, joka on siis esseekoe, jossa erilaisilla kysymyksillä vali-

doidaan, että on suorittanut itse harjoitustyön (pitää osata perustella erilaisia teknisiä ratkaisuja, mitä on työssä tehnyt tai jättänyt tekemättä). Essee-kysymyksiä on 4 kappaletta ja aikaa on 90 minuuttia, joten 1-osaan verrattuna tämä on testin helpoin osa (tietysti edellyttäen, että osaa selvästi englanniksi selvittää tekemiään teknisiä ratkaisuja). Tämän jälkeen jäin kesälomalle. Sunilta kerrottiin, että harjoitustyön tarkistamisessa menee noin neljä viikkoa. Kesälomalta paluuni jälkeen työpaikalla odotti kirje, jossa kerrottiin, että tutkinto on mennyt läpi.

Erilaisilla SCEA-sivuilla saa sen käsityksen, että IT-alan asiantuntijoiden keskuudessa tutkintoa pidetään varsin vaikeana, työläänä ja arvostettuna. Hyväksytyn suorituksen jälkeen Sun lähettää testatulle henkilölle tutkintotodistuksen ja SCEA-tiffin, jonka voi liittää esim. käyntikorttiinsa ja CV:hen sekä kivan pikku pinssin, jota pitämällä voi kiusata työkavereita. :-)

Tutkinnon suorittaminen oli mielenkiintoinen ja antoisa oppimiskokemus. Olen sitä mieltä, että on sekä työntekijöiden että oman yksikön kannalta mielekästä, että erilaisten ammatillisten tutkintojen suorittamista kannustetaan. Työntekijälle tulee hyvä mieli siitä, että saa konkreettisen ja kansainvälisesti standardoidun ja tunnetun todistuksen ammattitaidostaan (henkilökohtainen sertifikaatti), ja oma yksikkö saa meriittiä siitä, että yksikön työntekijöillä on konkreettinen todistus työntekijöidensä ammatillisesta kompetenssista, mitä voi käyttää hyväksi esimerkiksi projekti-tarjouksissa.

Kari Marttila,
DI, SCJP, SCEA,
TietoEnator Oyj

Linkkejä

Kirjoja

Eckel: Thinking in Java (2nd ed.). Hyvä Javan perusoppikirja.

Kassem: Designing Enterprise Architectures with the J2EE. Hyvä ja tiivis kirja J2EE:sta.

Fowler: UML Distilled. Hyvä ja tiivis UML:n perusoppikirja.

GoF: Design Patterns. Se alkuperäinen Design Pattern –kirja.

Alla olevissa SCEA-linkeissä on myös mainittu monia hyviä kirjoja.

Sunin Java-kursseja Suomessa

http://www.sun.fi/edu/desc/java/, joista SCEA-tutkintoa varten kannattaa käydä ainakin:

- SL-425 Architecting and Designing J2EE™ Applications
- SL-500 J2EE™ Patterns

Tekstissä mainittujen henkilöiden yhteystiedot

Stefan Nygård (Sun Microsystems Koulutuspalvelut): mailto:stefan.nygard@sun.com, http://www.sun.fi/edu.

Jarmo Reunanen (Resolution Oy): mailto:jarmo.reunanen@resolution.fi, http://www.resolution.fi.

JDK ja J2EE

JDK: http://java.sun.com/j2se/

J2EE: http://java.sun.com/j2ee/

Java-tutoriaaleja

JDK: http://java.sun.com/docs/books/tutorial/

J2EE: http://java.sun.com/j2ee/tutorial/

SUN:n Java-tutkintoihin liittyviä sivuja

SUN:n Certification-sivu: http://suned.sun.com/US/certification/java/index.html

Success Storyt: http://suned.sun.com/US/certification/java/index.html)

Marcus Greenin hyvät Certification-sivut: http://www.jchq.net/, joissa on hyvä Mock Exam –koe.

Certification Gurun hyvät sertifiointisivut: http://www.certificationguru.com/javaarchitectresources.html

jGurun sertifiointi-sivut: http://www.jguru.com/faq/subtopic.jsp?topicID=993

Prasksin SCEA-sivut: http://prasks.webahn.com/architect/scja.html

Amit Jnagalin SCEA-sivut: http://www.stormpages.com/jnagal/

Michael Thomasin SCEA-sivut: http://www.michael-thomas.com/architect/index.htm

Authorized Prometric testing centers: Kysy näistä Suomen Sunin edustajilta.

JavaSIG

Javan käyttäjät ja harrastajat ovat perustaneet eri puolilla maailmaa intressiryhmiä, joita nykyisin kutsutaan nimellä Java User Group. Suomessa Java-kerho on toiminut vuodesta 1998 alkaen entisen FISUG ry:n (Suomen Sun-käyttäjien kerho) alaisuudessa nimellä JavaSIG (Special Interest Group). Nyt JavaSIG jatkaa Sytyke ry:n Java-kerhona.

JavaSIGin toiminta

Tärkein toimintamuoto on pienimuotoisten seminaarien ja panelien järjestäminen aiheista, jotka kiinnostavat Java-asiantuntijoita ja sellaisiksi aikovia. Näitä ovat mm. olio-ohjelmoinnin ja -suunnitelun menetelmät, välineet ja käytännön sovellukset. Monissa seminaareissa on esiintynyt vilkastakin keskustelua kun koolla on joukko java-ammattilaisia. Seminaarien aiheita ovat olleet mm sovelluspalvelimet, java ja tietokannat, XP-ohjelmointi, vuosittaiset javaone-kuulumiset, suorituskyky, tuotteen rakentaminen javan avulla, patternit jne. Syksyn ensimmäisen seminaarin aihe on uusimmat sovelluskehitysympäristöt ja refactoring.

JavaSig:stä voivat hyötyä information muodossa niin Javaan siirtyvät systeemityön ammattilaiset, javassa jo pidemmälle ehtineet ammattilaiset kuin myöskin opiskelijat. Haemme uusia aktiiveja myöskin kerhotoiminnan pyörittämiseen.

Käypä vilkaisemassa websivujamme http://www.pcuf.fi/sytyke/kerhot/javasig/ ja liity JavaSigin postituslistalle.

JavaSIGin hallituksen kokous Café Ekberg: Juha Mäkeläinen, Hannu Kokko, Simo Vuorinen, poissa kuvasta Rami Talme ja Johannes Verwijnen.