Monikerrosarkkitehtuuria noudattavan Java-sovelluksen toteutus

Matti Lindell

13.6.2008

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

Tiivistelmä

Ohjelmistoarkkitehtuuri tarjoaa korkean abstraktiotason näkymän monimutkaisten järjestelmien tarkasteluun ja keinon hahmottaa niiden rakenteita. Ohjelmistoarkkitehtuurin avulla voidaan kuvata järjestelmää, sen osia ja osien välisiä suhteita sekä mahdollistaa sidosryhmien kommunikointi järjestelmän rakenteesta yhteisellä käsitteellisellä tasolla. Näkökulmaan, josta järjestelmän arkkitehtuuria tarkastellaan, liittyy usein sääntöjä ja periaatteita, joiden kautta järjestelmän kokonaisrakenne määräytyy tietyn arkkitehtuurityylin mukaisesti.

Tässä tutkielmassa perehdytään sekä teorian että käytännän tasolla kerrosarkkitehtuuriin ohjelmistojen rakennetta jäsentävänä arkkitehtuurityylinä. Tutkielman aluksi käsitellään ohjelmistoarkkitehtuuria, suunnittelumalleja ja kerrosarkkitehtuuria käsitteellisellä tasolla. Tämän jälkeen perehdytään yksityiskohtaisemin kerrosarkkitehtuurin soveltamiseen käytännössä esittelemällä esimerkkien kautta tutkielmaa varten toteutettua monikerrosarkkitehtuuriin perustuvaa Java-sovellusta.

ACM-luokat (ACM Computing Classification System, 1998 version): D1.5, D.2.2

Avainsanat: ohjelmistoarkkitehtuuri, kerrosarkkitehtuuri, suunnittelumallit, Java

Sisällys

1 Johdanto

Nykypäivän järjestelmien kasvavan koon ja lisääntyvän monimutkaisuuden myötä laskennalliset algoritmit ja tietorakenteet eivät enää muodosta merkittävintä osaa ohjelmistojen suunnitteluun liittyvästä ongelma-alueesta. Koko järjestelmän kattavan rakenteen suunnittelu ja määrittely sekä osakokonaisuuksien koostaminen yhdeksi organisoiduksi kokonaisuudeksi esittävät joukon uudenlaisia suunnitteluongelmia. Tällä tasolla tapahtuva organisointi tarvitsee tuekseen abstraktioita, joilla voidaan ilmentää laajojen osajärjestelmien ominaisuuksia ja vuorovaikutussuhteita (Garlan & Shaw, 1994; Shaw, 1989).

Koskimiehen & Mikkosen (2005) mukaan ohjelmistoarkkitehtuuri tarjoaa korkean abstraktiotason näkymän ohjelmistoihin, joka mahdollistaa monimutkaisten järjestelmien tarkastelun ja keinon hahmottaa niiden rakenteita. Ohjelmistoarkkitehtuuri mahdollistaa myös erilaisten sidosryhmien välisen kommunikoinnin järjestelmästä tarjoamalla vaaditun abstraktiotason sanaston ja käsitteistön. Ohjelmistoarkkitehtuurilla vaikutetaan ohjelmistojen suunnitteluun, toteutukseen, ylläpidettävyyteen, skaalautuvuuteen ja siirrettävyyteen sekä pyritään hallitsemaan niiden kehityksen elinkaarta.

Tässä tutkielmassa perehdytään ohjelmistoarkkitehtuurin ja varsinaisesti kerrosarkkitehtuurin osa-alueeseen keinona selventää, suunnitella ja toteuttaa monimutkaisia tietojärjestelmiä, sekä tuottaa niistä uudelleenkäytettäviä ja paremmin ylläpidettäviä. Ohjelmistoarkkitehtuurin osalta käydään läpi keskeisiä kirjallisuudessa esitettyjä määritelmiä, joiden perusteella pyritään jäsentämään mistä ohjelmistoarkkitehtuurissa perimiltään on kyse. Tutkielmassa perehdytään lisäksi suunnittelumallin käsitteeseen, käytännön hyötyihin ja soveltamiseen ohjelmistoissa. Suunnittelumalleista tuodaan esille myös niiden yhteys ja kytkökset ohjelmistoarkkitehtuurin. Arkkitehtuurityyleistä esitellään kerrosarkkitehtuuri ja asiakas-palvelin-arkkitehtuuri. Kerrosarkkitehtuurin osalta perehdytään sen keskeisiin periaatteisiin, kerrosten jaottelun liittyviin käytäntöihin ja kerrosten keskinäiseen vuorovaikutukseen liittyviin poikkeamiin. Asiakas-palvelin-arkkitehtuuria esitellään osana kerrosarkkitehtuurin käytännön ilmenemismuotoja.

Tutkielman pääpaino on kerrosarkkitehtuurin mukaisen Java-sovelluksen toteutuksessa. Esimerkkisovelluksen avulla teoria ilmennetään käytännön esimerkein ja esitetyt asiat

pyritään yhdistämään johdonmukaiseksi kokonaisuudeksi. Lisäksi kuvataan valittuun kerrosjakoon johtaneet periaatteet, eri kerrosten abstraktiotason sisältö, sekä yksittäisen kerroksen suunnitteluperiaatteet ja käytännön toteutus. Toteutuksen osalta käydään läpi vain eri kerrosten ja sovelluksen toiminnallisuuden kannalta keskeisimmät osa-alueet. Lisäksi kerrosten toteutuksessa hyödynnetään kirjallisuudessa hyväksi havaittuja suunnittelumalleja ja tekniikoita. Esimerkkisovelluksen rakenteita kuvataan UML 1.4.2 -notaation mukaisesti (OMG, 2005).

Käytetyissä koodiesimerkeissä hyödynnetään paikoittain Java 5.0 -versiossa kieleen lisättyä geneeristä tyypitystä (Java Community Process, 2001), joka asettaa lukijalle vaatimuksen geneeristen tyyppien ymmärtämisestä käsitteenä ja käytännön tasolla. Koodiesimerkkejä on osittain yksinkertaistettu selkeyden vuoksi esittämällä vain asiasisällön kannalta olennaisimmat asiat.

Tutkielman rakenne on seuraava. Luvussa 2 esitellään ohjelmistoarkkitehtuuri ja suunnittelumallin käsite. Luvussa 3 käydään läpi kerrosarkkitehtuuri, sen keskeiset periaatteet ja ominaisuudet. Luvussa 4 käsitellään monikerrosarkkitehtuuria noudattavan Java-sovelluksen toteutusta. Tutkielman päättää lyhyt yhteenveto luvussa 5.

2 Ohjelmistoarkkitehtuuri

Tässä luvussa perehdytään ohjelmistoarkkitehtuuriin ja sen merkitykseen tietojenkäsittelytieteen osa-alueena. Aluksi esitellään ohjelmistoarkkitehtuurin taustaa, tämän jälkeen läpikäydään kirjallisuudessa julkaistujen ohjelmistoarkkitehtuurin määritelmiä ja niiden perimmäisiä merkityksiä ja näkökulmia. Lopuksi käsitellään suunnittelumalleja ohjelmistoarkkitehtuuria tukevana käsitteenä.

2.1 Ohjelmistoarkkitehtuurin taustaa

Ohjelmistoarkkitehtuurit ja niiden tutkimus tietojenkäsittelytieteen osa-alueena vakiintui 1990-luvun aikana ohjelmistotekniikan omaksi alueekseen. Tätä ennen ohjelmistoarkkitehtuureja pidettiin enemmin korkeamman tason suunnitteluna (Koskimies & Mikkonen, 2005).

1980-luvulla ja sitä aiemmin käytännössä tunnetut abstrahointitekniikat, kuten korkean tason ohjelmointikielet ja abstraktit tietotyypit, olivat kehittäneet valmiuksia ja uusia lähestymistapoja määritellä sekä kehittää ohjelmistoja. Tästä huolimatta tietojärjestelmien jatkuvasti kasvava koko ja lisääntyvä monimutkaisuus toivat esiin jatkuvasti uudenlaisia ongelmia, joiden ratkaisemiseen perinteiset abstrahointitekniikat eivät sopineet. Tämä Shaw'n (1989) kuvaus sen ajan ohjelmistokehityksessä vallinneista käsityksistä sisältää historiallisen näkökulman ja kuvaa tarvetta, jonka kannustamana pyrittiin löytämään uusia malleja edellä kuvattujen ongelmien ratkaisemiseksi.

Shaw (1989) esittää ratkaisuksi järjestelmätason ohjelmistosuunnittelua, jossa tärkeät päätökset koskevat moduulien ja osajärjestelmien hyödyntämistä sekä organisointia. Tällä tasolla tapahtuva organisointi tarvitsee tuekseen uudenlaisia abstraktioita, joilla voidaan ilmentää laajojen osajärjestelmien ominaisuuksia ja vuorovaikutussuhteita. Kyse on *ohjelmistoarkkitehtuurin tasosta*.

2.2 Ohjelmistoarkkitehtuurin määritelmiä

Ohjelmistoarkkitehtuuri (software architecture) -käsitteelle ja sen perimmäiselle merkitykselle on kirjallisuudessa julkaistu useita eri määritelmiä. Yleisesti hyväksyttyä määritelmää ei käsitteelle kuitenkaan ole olemassa (Fowler, 2002), joten seuraavaksi luetellaan kirjallisuudessa esiteltyjä määritelmiä:

- Booch (1991) määrittää ohjelmistoarkkitehtuurin järjestelmän loogiseksi ja fyysiseksi rakenteeksi, joka rakentuu kehityksen aikana tehdyistä strategisista ja taktisista suunnittelupäätöksistä.

- Perry *& al.* (1992) kuvaavat ohjelmistoarkkitehtuurin muodostuvan erilaisten tiedonvälitykseen osallistuvien arkkitehtonisten osatekijöiden (architectural elements) kautta, jotka muodostavat arkkitehtuurin eri osia yhdessäpitävän koostumuksen.

- Garlanin & Perryn (1995) mukaan ohjelmistoarkkitehtuuri on järjestelmän tai sovelluksen komponenttien ja niiden keskinäisten suhteiden muodostama rakenne, sekä periaatteita (principles) ja ohjeistuksia (guidelines), jotka ohjaavat komponenttien suunnittelua ja kehitystä.

- Buschmann *& al.* (1996) määrittävät ohjelmistoarkkitehtuurin olevan osajärjestelmien (subsystem), komponenttien, sekä niiden välisten yhteyksien kuvaus.

- Bass *& al.* (1998) kuvaavat ohjelmistoarkkitehtuurin järjestelmän rakenteina tai rakenteena, joka koostuu komponenteista, niiden ulkoisista ominaisuuksista ja keskinäisistä suhteista.

Fowler (2002) perustaa oman näkemyksensä toisten aiemmin esittämiin määritelmiin ja nostaa esille kaksi ohjelmistoarkkitehtuurista yleisesti esiintyvää kuvausta: *järjestelmän jakamista osiin hyvin korkean tason näkökulmasta* ja *päätöksiä, joita on vaikea myöhemmin muuttaa*.

Suomenkielisessä kirjallisuudessa ohjelmistoarkkitehtuureja on viime aikoina tutkittu ainakin Koskimiehen ja Mikkosen osalta (2005), jotka määrittävät ohjelmistoarkkitehtuurin olevan

järjestelmän ydin, joka pysyy olennaisesti samana koko järjestelmän elinkaaren ajan, sekä *”ohjelmiston keskeisten osien ja niiden välisten suhteiden kuvaus korkealla abstraktiotasolla”*. Ohjelmistoarkkitehtuuriin nähdään kuuluvan myös tiettyjen ratkaisujen perustelu, ei pelkästään niiden kuvaus.

Edellisten määritelmien perusteella yksi keskeisimmistä piirteistä ohjelmistoarkkitehtuurille vaikuttaa olevan dekompositio- eli ositusperustainen lähestymistapa, jossa järjestelmä pilkotaan osiin, kuten komponentteihin tai vastaaviin osatekijöihin, tietyn periaatteen tai periaatteiden mukaisesti. Myös eri osien välisten yhteyksien kuvaus ja koko järjestelmää koskevien päätösten teko ovat olennaisia piirteitä. Koskimies & Mikkonen (2005) esittävät ohjelmistoarkkitehtuurien kuvaamista koskevan IEEE standardin 1471-2000 (IEEE, 2000) pohjalta, että ohjelmistoarkkitehtuuri ei kuvaa vain järjestelmän jakamista osiin, vaan lisäksi eri osien välisiä suhteita ja suhteiden kehittymistä. Suhteet liittyvät usein järjestelmän ajonaikaiseen käyttäytymiseen, jolloin ohjelmistoarkkitehtuuri kuvaa rakenteen lisäksi myös käyttäytymistä.

Kun ohjelmistoarkkitehtuuria määrittävien käsitteiden kehitystä tarkastellaan historiallisesta näkökulmasta, voidaan havaita jo olemassa olevien vakiintuneiden käsitteiden tarkentuneen uusien käsitteiden kautta, tarjoten aiempaa laajempia tulkintoja.

2.3 Näkökulmia ohjelmistoarkkitehtuuriin

Ohjelmiston arkkitehtuuria tarkastellaan tavallisesti jostain tietystä *näkökulmasta* (viewpoint), kuten prosessirakenteen tai loogisen rakenteen näkökulmista (Koskimies & Mikkonen, 2005). Näkökulmia voi olla samaan aikaan myös useita, kuten Kruchtenin (1995) esittämässä 4+1 näkymän mallissa, jossa arkkitehtuuria tarkastellaan rinnakkaisesti viidestä eri näkökulmasta (kuva 1):

- *Looginen näkymä* (logical view) kuvaa toiminnallisuuden jakautumista järjestelmän osien kesken sekä eri komponenttien velvollisuudet. Looginen näkymä voi kuvata sekä ohjelmakoodin rakenteita (luokat) että ajonaikaisia rakenteita (oliot) ja se toimii pohjana yksityiskohtaiselle suunnittelulle.

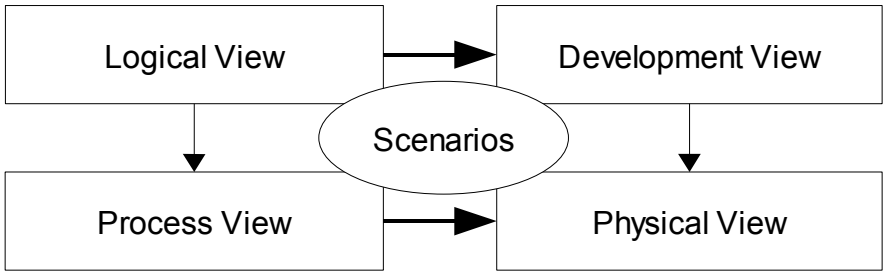
- *Prosessinäkymä* (process view) kuvaa järjestelmän toiminnan jakautumista loogisiin

prosesseihin ja prosessien keskinäistä vuorovaikutusta. Prosessinäkymä on keskeinen rinnakkaisille- ja hajautetuille järjestelmille, sillä sen avulla voidaan arvioida mm. suorituskykyä ja skaalautuvuutta.

- *Kehitysnäkymä* (development view) kuvaa kuinka järjestelmä on jaoteltu erillisinä yksikköinä toteutettaviin osiin. Kehitysnäkymää voidaan hyödyntää mm. projektisuunnittelussa ja projektin hallinnassa.

- *Fyysinen näkymä* (physical view) kuvaa järjestelmän fyysistä koostumusta eri laiteyksiköistä, kuinka laiteyksiköt ovat yhteydessä toisiinsa ja mitä fyysisiä ohjelmistoartefakteja (esim. komponentteja) yksittäiselle laiteyksikölle sijoitetaan. Fyysinen näkymä on keskeinen hajautetuille järjestelmille.

- *Skenaarionäkymä* (scenarios) kuvaa järjestelmän vuorovaikutusta sen ulkopuolella olevien käyttäjien ja muiden järjestelmien kanssa. Skenaarionäkymä toimii lähtökohtana muiden mallin näkymien muodostamiselle, sillä se koskee yleensä järjestelmän keskeisiä toiminnallisia vaatimuksia.



Kuva 1: Ohjelmistoarkkitehtuurin tarkasteluun soveltuva 4+1 näkymän malli (Kruchten, 1995).

Fowlerin (2002) mukaan näkökulma arkkitehtuurin osalta oleellisista asioista voi muuttua merkittävästi läpi järjestelmän elinkaaren. Tämä on johtanut näkemykseen, jonka mukaan järjestelmän arkkitehtuuria ei voi ilmentää vain yhdellä tavalla, vaan yksittäinen järjestelmä mielletään koostuvan useista eri arkkitehtuureista.

2.4 Suunnittelumallit

Suunnittelumalli (design pattern) -käsitteellä (Gamma *& al*., 1995) tarkoitetaan tunnetun ja käytännössä hyväksi havaitun uudelleenkäytettävän ratkaisun kuvausta ohjelmistojen suunnittelua koskeviin ongelmiin. Suunnittelumallien käytöllä pyritään tukemaan myös arkkitehtuurisuunnittelua, jolloin tietyssä ohjelmistoarkkitehtuurissa voidaan soveltaa juuri kyseiseen arkkitehtuuriin hyväksi havaittuja suunnittelumalleja (Koskimies & Mikkonen, 2005).

Suunnittelumalleja voidaan pitää 1990-luvun ohjelmistosuunnitteluun yhtenä eniten vaikuttaneista tekniikoista. Suunnittelumallien suosion taustalla on niiden riippumattomuus käytettävästä teknologiasta, suunnittelumenetelmästä tai ohjelmointikielestä. Suunnittelumallien hyödyntäminen on osoittautunut toimivaksi myös käytännön tasolla, jolloin kyseessä ei ole vain teoriassa hyvältä vaikuttava ajatus (Koskimies & Mikkonen, 2005).

2.4.1 Suunnittelumallin määritelmä

Yleisesti hyväksyttyä määritelmää suunnittelumallien käsitteelle ei ole olemassa (Fowler, 2002), mutta suunnittelumalleja tutkineet (Buschmann *& al.*, 1996; Fowler, 2002; Gamma *& al.*, 1995) viittaavat hyväksi lähtökohdaksi Alexanderin *& al.* (1977) esittämän määritelmän: ”jokainen suunnittelumalli kuvaa jonkin ongelman, joka esiintyy ympäristössämme toistuvasti ja lisäksi ydinratkaisun kyseiseen ongelmaan tavalla, jota voidaan hyödyntää lukuisia kertoja ilman että ydinratkaisua toteutetaan koskaan juuri samalla tavalla”.

Vaikka Alexander viittasikin määritelmällään suunnittelumalleihin rakennusarkkitehtuurissa, sama määritelmä soveltuu kuvaamaan myös ohjelmistojen suunnittelussa käytettäviä suunnittelumalleja. Perusta molempien tieteenalojen suunnittelumalleille on ongelmanratkaisu ongelmalle ominaisessa kontekstissa (Gamma *& al*., 1995; Fowler, 2002).

Gamma *& al*. (1995) kuvaavat suunnittelumallin koostuvan neljästä keskeisestä osasta:

- *Nimi* (pattern name) toimii yhdistävänä käsitteenä ongelmalle, sen ratkaisulle ja seurauksille. Nimi mahdollistaa keskeisten asioiden kuvaamisen muutamin sanoin ja

luo perustan sanastolle, jolla sidosryhmät voivat kommunikoida suunnittelumallin sisällöstä korkeammalla abstraktiotasolla.

- *Ongelma* (problem) kuvaa tilanteen, jolloin suunnittelumallia voidaan soveltaa, sekä selittää varsinaisen ongelman ja sen kontekstin. Ongelma voi kuvata suunnitteluun liittyviä keskeisiä ongelmia, kuten kuinka algoritmeja tai olioita tulisi mallintaa. Ongelma saattaa lisäksi sisältää joukon ehtoja, jotka on täytettävä ennen kuin suunnittelumallia voidaan käytännössä soveltaa.

- *Ratkaisu* (solution) kuvaa mallin muodostavat osatekijät, niiden suhteet, vastuut ja vuorovaikutukset. Ratkaisu ei kuvaa konkreettista mallia tai toteutusta, sillä suunnittelumalli toimii perustana, jota voidaan hyödyntää useissa eri tilanteissa. Sen sijaan ratkaisu tarjoaa abstraktin kuvauksen suunnitteluun sisältyvästä ongelmasta ja kuinka osatekijöiden (luokkien ja olioiden) avulla ongelma ratkaistaan.

- *Seuraukset* (consequences) ovat suunnittelumallin soveltamisesta johtuvia seurauksia ja kompromisseja. Seuraukset ovat kriittisessä asemassa, kun suunnitteluvaihtoehtoja arvioidaan ja verrataan keskenään, sekä oleellisia suunnittelumallin hyötyjen ja haittojen ymmärtämisessä. Seuraukset sisältävät yleensä vaikutuksia järjestelmän joustavuuteen, laajennettavuuteen tai siirrettävyyteen. Vaikutusten ilmaiseminen auttaa suunnittelumallien ymmärtämisessä ja arvioinnissa.

2.4.2 Suunnittelumallien hyödyntäminen ohjelmistoissa

Suunnittelumalleista kirjoittaneet ovat kategorisoineet mallejaan ongelma-alueen tai vastaavan kriteerin perusteella, jolloin vastaavanlaisia ongelma-alueita käsiteltäessä hyödynnettävissä on ennalta toimivaksi todettuja ratkaisumalleja selityksineen ja seurauksineen (Buschmann *& al.*, 1996; Fowler, 2002; Gamma *& al.*, 1995).

Suunnittelumallien yleisyys ja uudelleenkäytettävyys mahdollistavat niiden soveltamisen mitä erilaisimmissa ohjelmistoissa. Myös ohjelmistoalustakohtaisia suositeltavia suunnitteluratkaisuja on mahdollista esittää suunnittelumalleina (Koskimies & Mikkonen, 2005). Esimerkkinä tällaisesta on Java BluePrints (Sun Microsystems**,** 1994), joka kokoaa yhteen Java-ohjelmistoalustalla hyödynnettäviksi suositeltuja suunnitteluratkaisuja.

Suunnittelumallin hyödyntäminen ei lisää järjestelmään uutta toiminnallisuutta, vaan se parantaa ohjelmiston laatuominaisuuksia, kuten siirrettävyyttä, muunneltavuutta, ylläpidettävyyttä, tai suorituskykyä (Koskimies & Mikkonen, 2005). On kuitenkin syytä huomioida, että suunnittelumallissa on kyseessä vain ratkaisumalli. Niitä ei voida sellaisenaan ottaa käyttöön, vaan ideana on soveltaa niitä käyttötarkoitukseen ja kontekstiin sopiviksi. Ne tarjoavat kuitenkin suurissa määrin lisäarvoa pelkän ongelma-alueen ymmärtämiseksi, vaikka mallia ei suoranaisesti hyödynnettäisikään varsinaisen ongelman ratkaisussa (Fowler, 2002).

3 Kerrosarkkitehtuuri

Edellä käsiteltiin ohjelmistoarkkitehtuuria yleisellä tasolla ja seuraavaksi perehdytään yhteen sen esitystyyliin, kerrosarkkitehtuuriin. Aluksi esitellään arkkitehtuurityylin käsite ja yhteydet kerrosarkkitehtuuriin. Seuraavaksi jälkeen esitellään kerrosarkkitehtuurin kuvaamista graafisesti. Tämän jälkeen esitellään kerrosarkkitehtuurin ominaisuuksia ja periaatteita, sekä perehdytään sen moniulotteisuuteen. Lopuksi tarkastellaan esimerkkien kautta kuinka kerrosarkkitehtuuria hyödynnetään yhdessä toisen arkkitehtuurityylin kanssa ryhmittelemään ohjelmistojen rakennetta.

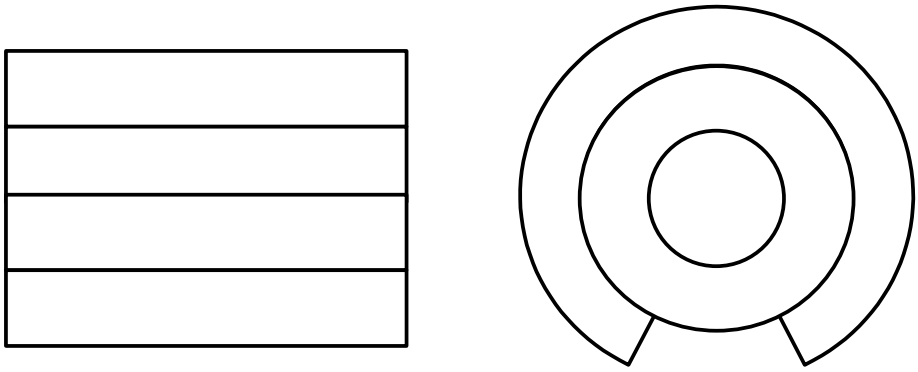
3.1 Kerrosarkkitehtuuri arkkitehtuurityylinä

Kerrosarkkitehtuuri on yksi *arkkitehtuurityylin* (architectural pattern) ilmentymä. Buschmannin *& al.* (1996) mukaan arkkitehtuurityyli on kaikkein korkeimmalla tasolla sovellettava, koko järjestelmää luonnehtiva suunnitteluratkaisu. Koskimies & Mikkonen (2005) perustavat näkemyksensä Shaw'n & Garlanin (1996) esittämään määritelmään ja kuvaavat arkkitehtuurityylejä suunnittelumallien idean yleistyksenä koko järjestelmän arkkitehtuurin kantavaksi periaatteeksi, joita käytetään apuna varsinaisen järjestelmän hahmottamisessa ja sen toteutuksen rakenteen selityksessä. Arkkitehtuurityylistä käytetään myös nimitystä *arkkitehtuurimalli*.

Koskimies & Mikkonen (2005) mainitsevat kerrosarkkitehtuurin arkkitehtuurityylinä, joka soveltuu lähes minkä tahansa järjestelmän kuvaamisessa. Samassa yhteydessä mainitaan, että eri arkkitehtuurityylit eivät ole toisiaan poissulkevia, joten kerrosarkkitehtuuria voidaan hyödyntää järjestelmän arkkitehtuurin kuvaamisessa yhdessä muiden arkkitehtuurityylien kanssa.

3.2 Kerrosarkkitehtuurin kuvaaminen

Kerrosarkkitehtuurin kuvaamiseen (kuva 2) käytetään yleisesti ”kerrosvoileipä” -muotoista esitystapaa (Buschmann *& al.*, 1996; Fowler, 2002; Koskimies & Mikkonen, 2005) tai vastaavasti ympyränmuotoista esitystapaa, jolloin kerrosvoilevän päät on taivutettu yhteen (Garlan & Shaw, 1994; Koskimies & Mikkonen, 2005).



Kuva 2: Kerrosarkkitehtuurin graafisia esityksiä (Garlan & Shaw, 1994; Koskimies & Mikkonen, 2005).

3.3 Kerrosarkkitehtuurin ominaisuuksia

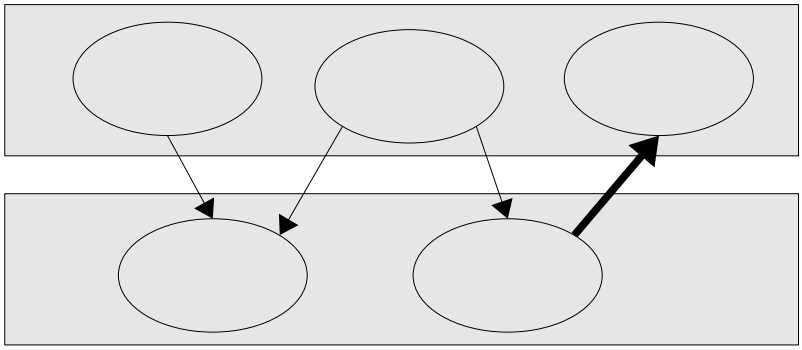
Buschmann *& al.* (1996) mukaan *kerrostus* (layering) on arkkitehtoninen suunnittelumalli, jonka avulla ohjelmistoja voidaan jäsentää jakamalla niitä alitehtävien (subtask) ryhmiin eli kerroksiin, joista jokainen kerros kuvaa tietyn tason abstraktiota. Fowler (2002) kuvaa kerrostuksen yhdeksi tavallisimmista tekniikoista, joita ohjelmistosuunnittelussa käytetään selkeyttämään monimutkaisia järjestelmiä.

Kerrokset järjestellään jonkin abstrahointiperiaatteen mukaan nousevaan järjestykseen usein tasolla laite/ihminen: laite-päässä olevat kerrokset ovat matalammalla abstraktiotasolla, tarjoten laitetta tai käyttöjärjestelmää lähellä olevia toimintoja. Vastaavasti ihmistä lähellä olevat kerrokset tarjoavat korkeamman abstraktiotason palveluja, kuten graafisen käyttöliittymän sovellusalueeseen liittyviä palveluja (Garlan & Shaw, 1994; Koskimies & Mikkonen, 2005). Perusteina kerrosten osittamiselle voivat olla esim. toiminto, tehtävä, laitteisto tai toiminnan kohde (Fowler, 2000).

Kerrosten teknisistä ominaisuuksista johtuen abstraktiotasojen tunnistaminen käytännössä voi osoittautua ongelmalliseksi, jolloin kerrostuksen periaate ilmenee siten, että ylemmän kerroksen palvelut käyttävät hyväkseen alemman kerroksen palveluja: *”abstraktiotasoltaan korkeammat palvelut toteutetaan käyttäen abstraktiotasoltaan matalampia palveluja”*. Tämä on kerrosarkkitehtuurin perusajatus (Koskimies & Mikkonen, 2005). Kun ylempi kerros käyttää lisäksi vain välittömästi alemman kerroksen palveluita, on kyse *suljetusta* tai *puhtaasta kerrosarkkitehtuurista* (Laine & Paakki, 2001).

Käytännössä suljetusta kerrosarkkitehtuurista on tarvetta toisinaan poiketa, jolloin kyseessä on *avoin kerrosarkkitehtuuri* (Laine & Paakki, 2001). Poikkeamia on kahdentyyppisiä: palvelukutsun kulkeminen alemmasta kerroksesta ylempään (hierarchy breach) tai kerrosten ohitus palvelukutsun kulkiessa ylemmistä kerroksista alempiin (bridging). Kerrosten ohittaminen voi olla tarpeen esim. tehokkuussyistä, jolloin tarvittava palvelu on tarjolla tehokkaammassa muodossa alemmilla kerroksilla tai jos lähin alempi kerros ei tarjoa tarvittavaa palvelua (Koskimies & Mikkonen, 2005).

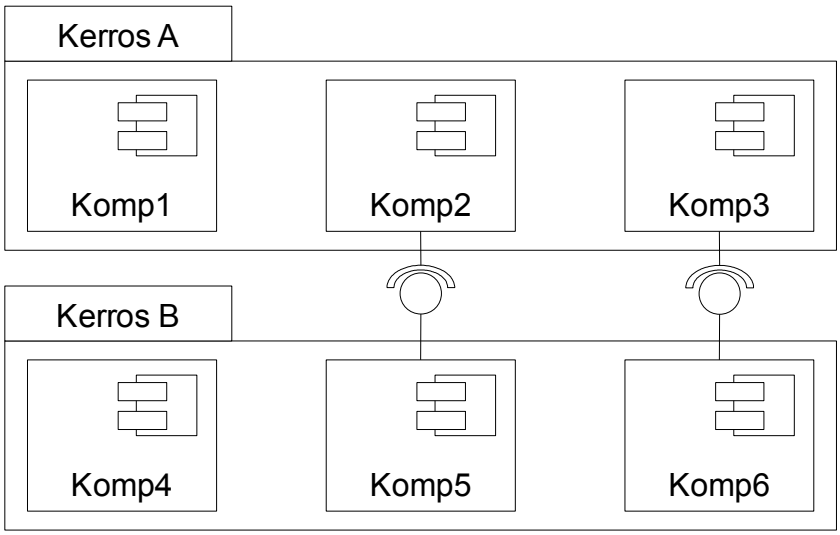
Palvelukutsun kulkeminen alemmasta kerroksesta ylempään on vakava ongelma kerrosarkkitehtuurissa, jos seurauksena on alemman kerroksen riippuvuus ylemmästä kerroksesta. Kyseinen tilanne voi olla välttämätön, jos alemman kerroksen on sovitettava oma palvelunsa ylemmän kerroksen mukaan ja kutsuttava ohjelmakoodia, joka kuuluu ylempään kerrokseen (Koskimies & Mikkonen, 2005). Ongelma voidaan ratkaista toteuttamalla kutsu soveltaen *takaisinkutsu* (callback) -periaatetta (kuva 3), jolloin alemman kerroksen riippuvuus ylemmästä vältetään (Buschmann *& al.*, 1996). Takaisinkutsu on mekanismi, jonka avulla palvelun kutsuja voi saada kontrollin palvelun aikana. Takaisinkutsua varten alempi kerros tarjoaa jonkin rekisteröintioperaation, jolla ylempi kerros rekisteröi ohjelmakoodinsa alemman kerroksen käyttöön (Koskimies & Mikkonen, 2005).



Kuva 3: Takaisinkutsu kerrosten välillä (Laine, 2000).

Avoimen kerrosarkkitehtuurin käytöllä tavoitellaan usein parempaa joustavuutta ja suorituskykyä, joka johtaa samalla huonompaan ylläpidettävyyteen, sillä kerrosten väliset riippuvuussuhteet lisääntyvät. Suljettu kerrosarkkitehtuuri on tavoitetullumpi muoto sen paremman selkeyden ja ylläpidettävyyden takia, mutta voi johtaa vastaavasti järjestelmän heikompaan suorituskykyyn (Laine & Paakki, 2001).

Yksittäisellä kerroksella on joukko rajapintoja, jotka se toteuttaa, ja vastaavasti joukko rajapintoja, jotka kerros tarvitsee (kuva 4). Palvelukutsut kerroksesta toiseen kulkevat rajapintojen kautta ja niiden avulla kerroksen komponentit tehdään riippumattomiksi toisten kerrosten komponenttien toteutuksista, jolloin eri kerrokset ovat tarvittaessa vaihdettavissa ilman vaikutusta muuhun järjestelmään. Oikein toteutettuna rajapintojen käyttö parantaa kerrosten ja koko järjestelmän ylläpidettävyyttä, siirrettävyyttä ja uudelleenkäytettävyyttä (Buschmann *& al.*, 1996; Koskimies & Mikkonen, 2005).



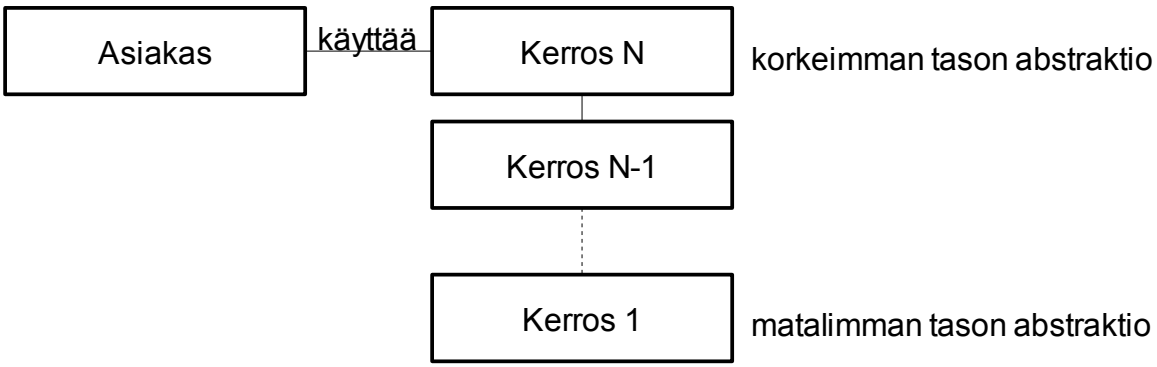
Kuva 4: Kerrosten välisen rajapinnan käyttö- ja tarjontasuhde (Koskimies & Mikkonen, 2005).

3.4 Kerrostuksen periaate

Buschmann *& al*. (1996) esittävät kerrostuksen periaatteen käytön seuraavasti: Jäsennetään järjestelmä kerroksiksi aloittaen alimman abstraktiotason kerroksesta - kerroksesta 1, joka on samalla järjestelmän pohjakerros. Kerroksia lisätään aina abstraktiotason vaihtuessa siten, että kerros J lisätään kerroksen J-1 päälle, kunnes saavutetaan ylimmän tason toiminnallisuus - kerros N. Periaatetta on havainnollistettu kuvassa 5.

On huomioitavaa, että kyseinen malli ei määrää missä järjestyksessä kerrokset varsinaisesti suunnitellaan; se tarjoaa vain käsitteellisen näkymän. Malli ei myöskään määrää tulisiko yksittäisen kerroksen J olla kompleksinen osajärjestelmä, joka tarvitsee edelleen hajauttaa tai tulisiko kerroksen J vain välittää palvelupyyntöjä kerrokselta J+1 kerrokselle J-1

osallistumatta pyynnön sisältöön. Oleellista on, että samassa kerroksessa sijaitsevat komponentit toimivat samalla abstraktiotasolla.

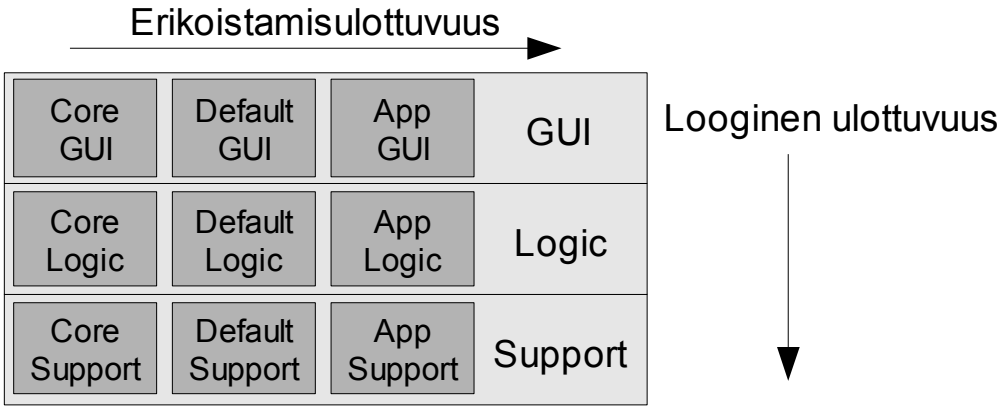


Kuva 5: Kerrostamisen periaate (Buschmann & al., 1996).

Suurin osa palveluista, jotka kerros J tarjoaa, ovat koostettu kerroksen J-1 tarjoamista palveluista, eli toisin sanoen ylemmän kerroksen palvelut toteutetaan alemman kerroksen palveluiden avulla. Lisäksi, kerroksen J palvelut voivat olla riippuvaisia saman kerroksen muista palveluista.

3.5 Kerrosarkkitehtuurin moniulotteisuus

Kerrosarkkitehtuuri voidaan toisinaan mieltää moniulotteiseksi, sillä kerrostus on mahdollista hahmottaa useampaan kuin yhteen suuntaan (kuva 6). Kaksiulotteisuus on havaittavissa erityisesti ohjelmistoalustojen tapauksessa (esim. Java-ohjelmistoalusta), joiden ideana on tarjota yleiskäyttöisiä komponentteja ja valmiita toteutuksia ohjelmistokehitykseen. Tällöin ulottuvuudet ovat tyypillisesti ”looginen” ja ”yleisyys”. Looginen ulottuvuus käsittää perinteisen sovelluksen kerrosjaon jonkin ositusperiaatteen mukaisesti. Yleisyys kuvaa kuinka yleiskäyttöisiä tai vastaavasti tuotekohtaisia komponentit ovat (Koskimies & Mikkonen, 2005).

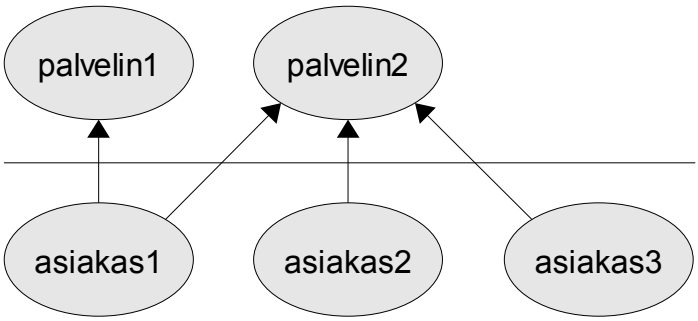


Kuva 6: Kaksiulotteinen kerrosarkkitehtuuri (Koskimies & Mikkonen, 2005).

Fowler (2002) huomauttaa, että kerrosarkkitehtuurin kerroksia kuvaavien englanninkielisten termien *layer* ja *tier* välillä on joskus eroavaisuuksia riippuen asiayhteydestä ja tarkastelun näkökulmasta. Yleensä termejä käytetään synonyymeina, mutta usein termiä *tier* käytetään kuvaamaan kerrosten fyysistä erottelua. Tällöin pyritään ilmentämään, että kerrosten jakoperusteina toimivat fyysiset laitteistorajat eli kerrokset ovat hajautettu fyysisesti eri laitteille. Termiä *layer* käytetään vuorostaan kuvaamaan kerrosten loogista erottelua, jolloin pyritään ilmaisemaan eroavaisuuksia enemmänkin kerrosten käsitteellisessä sisällössä kuin niiden fyysisessä sijainnissa. Riippumatta käytettävästä termistä, kerrosarkkitehtuuri mahdollistaa kerrosten sijoittelun ja tarkastelun sekä loogisessa että fyysisessä ulottuvuudessa: kaikki kerrokset voivat sijaita yhdellä laitteella tai vastaavasti hajautettuna useille eri laitteille.

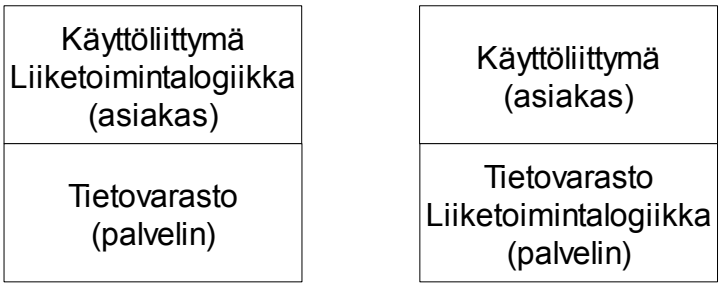
3.6 Kerrosarkkitehtuuri ja asiakas-palvelin-arkkitehtuuri

Yksi tunnetuimpia esimerkkejä, jossa kerrosarkkitehtuurin mallia voidaan hyödyntää, on *asiakas-palvelin-arkkitehtuuri* (kuva 7). Palvelin tarjoaa käytettäviä palveluja asiakkaille, jotka suorittavat palvelupyyntöjä tietyn palvelun käyttämiseksi. Arkkitehtuuritason näkökulmasta palvelin kapseloi resurssien hallinnan, jolloin asiakkaiden ei tarvitse huolehtia tietyn resurssin käyttöön liittyvistä teknisistä ongelmista ja ovat näin ollen riippumattomia toisista asiakkaista. Asiakas-palvelin-arkkitehtuuria ajatellaan yleisesti hajautettuna järjestelmänä (Fowler, 2002; Koskimies & Mikkonen, 2005).



Kuva 7: Asiakas-palvelin-arkkitehtuuri (Koskimies & Mikkonen, 2005).

Asiakas-palvelin-arkkitehtuurin perinteisin ilmenemismuoto tunnetaan 1990-luvulla yleistyneenä *kaksikerrosarkkitehtuurina* (two-tier architecture), jossa järjestelmän komponentit ovat hajautettu asiakkaan ja palvelimen kerroksiin. Asiakas sisältää käyttöliittymän ja muun ohjelmakoodin, palvelin koostuu yleensä tietokantapalvelimesta tai vastaavasta tietovarastosta. Kaksikerrosarkkitehtuuri soveltuu hyvin yksinkertaisten tietorakenteiden esittämiseen ja ylläpitämiseen, mutta ongelmaksi muodostuu käyttöliittymäkomponentteihin upotettu liiketoimintalogiikka ja sen ylläpito. Kyseinen malli ei ole joustava uusien muutosten osalta ja johtaa käyttöliittymäkomponenttien duplikointiin jos liiketoimintalogiikka on tarvetta uudelleenkäyttää. Lisäksi liiketoimintalogiikan kompleksisuuden kasvaminen johtaa käyttöliittymäkomponenttien monimutkaisiin ja hankalasti ymmärrettäviin rakenteisiin. Vaihtoehtoisesti liiketoimintalogiikka voidaan toteuttaa tietokantaan tallennetuilla proseduureilla (stored procedures), mutta seurauksina ovat tietokantatoteutuksesta riippuvainen liiketoimintalogiikka ja rajoitetut mekanismit tiedon rakenteistamiseen (Fowler, 2002; Schussel, 1995). Kaksikerrosarkkitehtuurin variaatiot on esitetty kuvassa 8.



Kuva 8: Kaksikerrosarkkitehtuurin variaatioita.

Asiakas-palvelin-arkkitehtuurin yleisin ilmenemismuoto tunnetaan *kolmikerrosarkkitehtuurina* (three-tier architecture), jota kutsutaan myös

monikerrosarkkitehtuuriksi (multi-tier architecture). Kolmikerrosarkkitehtuuri pyrkii ratkaisemaan kaksikerrosarkkitehtuurissa ilmenneet ongelmat erottamalla liiketoimintalogiikan täysin omaksi kerroksekseen (Fowler, 2002). Brownin *& al*. (2001) esittämän nimeämiskäytännön mukaisesti kolmikerrosarkkitehtuurin kerrokset (kuva 9) koostuvat: *esityskerroksesta* (presentation), *liiketoimintalogiikan kerroksesta* (domain) ja *tietolähdekerroksesta* (data source). Kolmikerrosarkkitehtuurin mukainen ositus mahdollistaa liiketoimintalogiikan uudelleenkäytettävyyden ja paremman ylläpidettävyyden verrattuna kaksikerrosarkkitehtuurin malliin. Kerrosarkkitehtuurille ominaisesti jokainen kerros voi käytännössä sijaita hajautettuna fyysisesti erillisille laitteille, mutta vastaavasti kaikki kerrokset voivat sijaita fyysisesti yhdellä laitteella. Käytännön tasolla kolmikerrosarkkitehtuuria hyödynnetään tyypillisesti sovelluspalvelinperustaisissa sovelluksissa, jossa esityskerros on loppukäyttäjän www-selain, liiketoimintalogiikan kerroksen muodostaa sovelluspalvelin komponentteineen ja tietolähdekerroksena on relaatiotietokanta (Fowler, 2002; Schussel, 1995).

Kuva 9: Kolmikerrosarkkitehtuurin kerrokset (Brown & al., 2001).

Kolmikerrosarkkitehtuurin pohjalta on kehitetty asiakas-palvelin-arkkitehtuurin malleja, joissa liiketoimintalogiikan kerrosta on ositettu abstraktiotasoltaan täsmällisempiin kerroksiin (Bennett *& al.*, 2006; Fowler, 2002). Esimerkiksi nelikerrosarkkitehtuurissa (kuva 10) liiketoimintalogiikan kerros ositetaan kahdeksi kerrokseksi: *sovellusaluelogiikka* toteuttaa varsinaisen sovellusalueen logiikan ja käsitteet, *sovelluslogiikka* sisältää yksittäisen sovelluksen logiikan. Jos mallia sovellettaisiin esimerkiksi matkanvarausjärjestelmässä, sisältäisi sovellusaluelogiikka jokaiselle matkalle yhteiset asiat, kuten asiakas, laskutus, kohde, jne., ja vastaavasti sovelluslogiikka sisältäisi matkanvarauslogiikan. Kyseinen kerrosarkkitehtuurin malli tarjoaa mahdollisuuden uudelleenkäyttää kahta alinta kerrosta, kun toteutetaan uusi sovellus samalle sovellusalueelle (Bennett *& al.*, 2006; Koskimies & Mikkonen, 2005).

Kuva 10: Nelikerrosarkkitehtuurin kerrokset (Bennet & al., 2006).

4 Monikerrosarkkitehtuuria noudattavan Java-sovelluksen toteutus

Seuraavaksi perehdytään monikerrosarkkitehtuuria noudattavan Java-sovelluksen toteutukseen käytännön näkökulmasta soveltamalla tutkielmassa edellä esiteltyä kerrosarkkitehtuuria, sekä kirjallisuudessa esiteltyjä suunnittelumalleja ja tekniikoita. Aluksi esitellään esimerkkisovelluksen taustatietoja, sen kuvaus ja ongelman asettelu. Tämän jälkeen esitellään esimerkkisovelluksen kerrosjakoon johtaneet periaatteet, jonka jälkeen yksittäiset kerrokset käydään läpi suunnittelun ja toteutuksen tasolla.

4.1 Järjestelmän kuvaus

Tutkielman taustatietoja varten toteutettiin asiakas-palvelin-arkkitehtuuriin perustuva esimerkkisovellus, jossa noudatetaan samalla kerrosarkkitehtuurin mukaista arkkitehtuurityyliä. Ideana on esitellä esimerkkisovelluksen kautta kerrosarkkitehtuurin toteuttamista käytännön tasolla, perehtyen eri kerrosten suunnitteluun ja toteutukseen sekä keskinäiseen vuorovaikutukseen. Esiintyviin ongelmatilanteisiin pyritään löytämään ratkaisuja hyödyntämällä suunnittelumalleja ja Java-ohjelmistoalustalle tarjolla olevia tekniikoita.

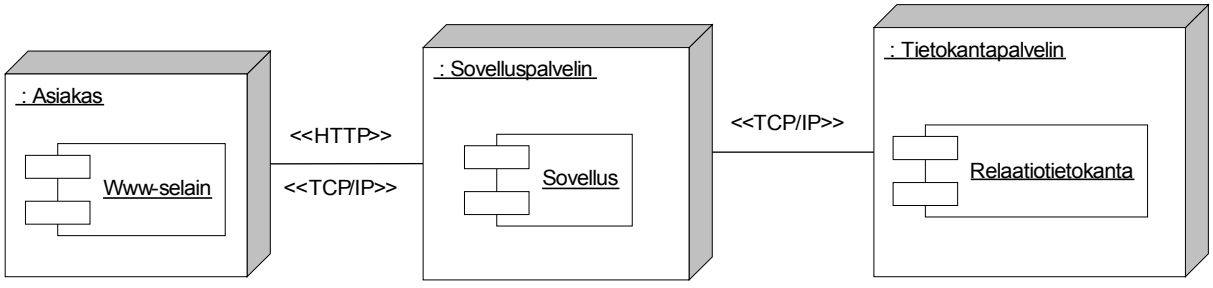
Esimerkkisovelluksen käyttötarkoitus on esittää ja ylläpitää tietoa kursseista, oppilaista ja opettajista sekä tarjota graafinen käyttöliittymä käyttäjille tiedon operointia varten. Sovellus suunnataan käyttöympäristöön, jossa vaatimuksina ovat reaaliaikaisuus, usean käyttäjän rinnakkainen tiedon operointi sekä käsiteltävän tiedon pysyvyys. Käyttäjiä ei ole tarvetta ryhmitellä tai muuten erotella toisistaan, eikä autentikoida. Sovelluksen toiminnot on kuvan 1 mukaisesti kuvattu skenaarionäkymästä liitteessä 1.

Esimerkkisovelluksen käyttöympäristöksi valittiin Apache Tomcat -sovelluspalvelin (The Apache Software Foundation, 2008a) ja HSQLDB -tietokantapalvelin (HSQLDB, 2008). Sovelluspalvelin valittiin osaksi esimerkkisovelluksen ympäristöä, sillä se mahdollistaa sovelluksen käytön usean käyttäjän toimesta rinnakkaisesti ja reaaliaikaisesti. Lisäksi ympäristö tukee standardoituja tekniikoita dynaamisen sisällön tuottamisen ja sovelluksesta voidaan tehdä www-selainkäyttöinen, jolloin ei ole tarvetta toteuttaa käyttäjille erillistä

asiakasohjelmistoa. Standardeja noudattava sovelluspalvelin tukee vähintään HTTP-protokollaa, joka mahdollistaa sovelluksen käytön tietoverkossa (The Apache Software Foundation, 2008a; Java Community Process, 2003a; Java Community Process, 2003c; Java Community Process, 2004).

Esimerkkisovelluksen käyttöympäristön vaatimuksissa esitetyllä tiedon *pysyvyydellä* (persistence) viitataan tilanteeseen, jossa järjestelmän käsittelemän tiedon elinkaari on riippumaton varsinaisen järjestelmän tilasta. Ilman pysyvyyttä sovelluksen uudelleenkäynnistys johtaisi olemassa olevan tiedon häviämiseen (Bauer & King, 2007). Esimerkkisovelluksen osalta tiedon pysyvyyden varmistamiseen käytetään tietolähteenä relaatiotietokantaa. Käytetty tietokantarakenne on havainnollistettu liitteessä 3. Vuorovaikutukseen sovelluksen ja tietokannan välillä käytetään standardoitua JDBC (Java Database Connectivity) -rajapintaa, joka mahdollistaa tiedon välityksen Java-ohjelmiston ja relaatiotietokannan välillä (Ellis & Ho, 2001).

Kun edellä kuvatun esimerkkisovelluksen arkkitehtuuria tarkastellaan kuvan 1 mukaisesti fyysisestä näkymästä, niin sen rakenteen (kuva 11) voidaan havaita noudattavan kolmikerrosarkkitehtuurille tyypillistä jakoa: tietokantapalvelin muodostaa tietolähdekerroksen, sovelluspalvelin liiketoimintalogiikan kerroksen ja loppukäyttäjä asiakkaan roolissa ilmentää www-selaimella esityskerroksen. Tämän tutkielman osalta keskitytään jatkossa liiketoimintalogiikan kerrokseen ja sen vuorovaikutukseen muiden kerrosten kanssa, esityskerroksen tai tietolähdekerroksen toteutukseen ei perehdytä tarkemmin.



Kuva 11: Esimerkkisovelluksen arkkitehtuurin fyysinen rakenne.

4.2 Kerrosten jaottelun periaatteet

Esimerkkisovelluksen suunnittelun alkuvaiheessa valittiin kerrosarkkitehtuurin malleista aiemmin tutkielmassa esitelty kolmikerrosarkkitehtuuri (kuva 9), sillä järjestelmän arkkitehtuurin fyysinen rakenne (kuva 11) tuki valintaa ja kolmikerrosarkkitehtuuri on todettu soveltuvaksi asiakas-palvelin -ympäristöön (Fowler, 2002; Schussel, 1995). Liiketoimintalogiikka haluttiin eriyttää omaksi kerroksekseen, jotta se olisi helpommin ylläpidettävissä ja mahdollista hajauttaa sovelluspalvelinympäristöön. Tästä syystä kaksikerrosarkkitehtuurin mukainen malli ei soveltunut käytettäväksi.

Liiketoimintalogiikan kerroksen suunnittelussa hyödynnettiin kirjallisuudessa esiteltyjä suunnitteluperiaatteita ja lähestymistapoja määrittämään kerroksen eri komponentit, niiden vastuualueet ja keskinäiset vuorovaikutussuhteet (Bauer & King, 2007; Fowler, 2002; Walls & Breidenbach, 2005). Käytettyihin suunnitteluperiaatteisiin ja lähestymistapoihin perehdytään myöhemmin kyseisen luvun alakohdissa**.** Määriteltyjä komponentteja ja näiden vastuualueita lähemmin tarkasteltaessa havaittiin, että kerros olisi mahdollista osittaa tarkemmin kuvamaan eri komponenttien abstraktiotasoja. Tämän seurauksena esimerkkisovelluksen käyttämä kerrosarkkitehtuurin malli muotoutui vastaamaan Marinescun (2002) kuvaamaa kerrosjakoa, joka koostuu viidestä kerroksesta (kuva 12). Verrattaessa kerrosjakoa aiemmin esiteltyyn nelikerrosarkkitehtuurin malliin (kuva 10), liiketoimintalogiikan kerros on ositettu sovellusaluelogiikan ja sovelluslogiikan kerrosten lisäksi myös palvelukerrokseen (service layer).

Kuva 12: Esimerkkisovelluksen käyttämä kerrosjako.

Käytännön tasolla esimerkkisovelluksen toteuttama kerrosjako ei ole suljettu kerrosarkkitehtuuri. Kerrosten välillä tapahtuu ohituksia, sillä sovellusaluelogiikan kerroksen sisältöä hyödynnetään sen suoranaisesti ylemmän kerroksen lisäksi myös muissa ylemmissä kerroksissa. Tämä on perusteltua, kun sovellusaluelogiikan olioita voidaan käyttää sellaisenaan käsiteltävän tiedon mallintamiseen muissa kerroksissa (Fowler, 2002; Bauer & King, 2007). Jos kyseisiin olioihin tarvitaan lisätoiminnallisuutta, voidaan hyödyntää esimerkiksi *rakennetta mukauttavia suunnittelumalleja* (structural patterns), kuten *koristelijaa* (decorator), joiden avulla olion ominaisuuksia tai toiminnallisuutta voidaan muuttaa vaikuttamatta sen alkuperäisen toteutukseen (Gamma *& al.*, 1995). Esimerkkisovelluksen käyttämässä kerrosarkkitehtuurin toteutuksessa ei yksikään alempi kerros ole riippuvainen ylemmästä, eli palvelukutsut kulkevat aina ylhäältä alas.

4.3 Sovellusaluelogiikan kerros

Seuraavaksi esitellään esimerkkisovelluksen kerroksista sovellusaluelogiikan kerros. Aluksi perehdytään kerroksen rakenteeseen käsitteellisellä tasolla, jonka jälkeen kerroksen sisältö ilmennetään osittaisena käytännön toteutuksena.

4.3.1 Kerroksen kuvaus

Sovellusalueen malli (domain model) on käsitteellinen malli, jota olio-ohjelmointikielissä käytetään kuvamaan järjestelmän kohdeympäristön käsitteitä ja käyttäytymistä yhteenkuuluvien olioiden muodostamana oliomallina (Fowler, 2002; Marinescu, 2002). Sovellusalueen malli voidaan johtaa käyttötapausten perusteella sekä konsultoimalla sidosryhmiä, joilla on asiantuntemusta kohdeympäristöstä ja sen ongelmista (Marinescu, 2002). Sovellusalueen malli on perusteltua eriyttää omaksi sovellusaluelogiikan kerrokseksi, jolloin se on hyödynnettävissä muilta sovelluksen kerroksilta sekä uudelleenkäytettävissä samaa sovellusaluetta käsittelevissä ohjelmistoissa (Marinescu, 2002). Sovellusalueen malli on keskeinen osa koko sovellusta. Se tarjoaa yhteisen käsitteistön ohjelmistoprojektiin osallistuville sidosryhmille ja mahdollistaa kommunikoinnin sovelluksesta samalla käsitteellisellä tasolla (Evans, 2003). Sovellusalueen mallin virheellinen mallinnus johtaa kohdeympäristön todellisuuden vääristymiseen, josta seurauksena voi olla käyttökelvoton sovellus. Vaikutukset heijastuvat myös muiden kerrosten suunnitteluun ja toteutukseen, sillä olettamukset tiedosta ja käyttäytymisestä perustetaan sovellusalueen mallin kuvaamaan ympäristöön.

Esimerkkisovelluksen yksi vaatimuksista oli toteuttaa käsiteltävän tiedon pysyvyys, jota varten sovellusaluelogiikan kerroksen ja tietolähdekerroksen välille on muodostettava yhteys ja toteutettava logiikka tiedon välityksestä tietolähteeseen. Sovellusaluelogiikan kerros tarjoaa tiedon käytettäväksi myös ylemmille kerroksille, jota varten toteutetaan tarvittavat palvelurajapinnat kerrosarkkitehtuurin periaatteiden mukaisesti (kuva 4).

4.3.2 Kerroksen sovellusalueen mallin toteutus käytännössä

Esimerkkisovelluksen tapauksessa sovellusalueen mallin muodostavat oliot (liite 2) johdettiin sovelluksen käyttötapausten perusteella (liite 1) Marinescun (2002) kuvaaman lähestymistavan mukaisesti hahmottamalla ongelma-alueen subjektit: ihmiset, asiat ja käsitteet. Mallia tarkennettiin lopuksi kuvaamalla olioiden ominaisuudet ja keskinäiset suhteet.

Käytännössä sovellusalueen malli toteutettiin käyttäen POJO (Plain Old Java Object) -tyyppisiä olioita. POJO-olioiden perusajatuksena ja samalla hyötynä on yksinkertaisuus (Fowler, 2000). Ne ovat lisäksi riippumattomia käytettävästä ohjelmistoalustasta verrattaessa EJB:n (Enterprise JavaBeans) tarjoamaan komponenttimalliin, jossa sovellusalueen mallin olioilta vaaditaan ohjelmistoalustakohtaisten rajapintojen toteutusta (Java Community Process, 2003b; Java Community Process, 2005). Esimerkkinä POJO-olion toteutuksesta esitetään sovellusalueen malliin kuuluva kurssia (course) mallintava olio (kuva 13), jota on yksinkertaistettu selkeyden vuoksi. POJO-olion vaatimuksia ovat `java.io.Serializable` -rajapinnan toteutus, parametriton konstruktori sekä lisäksi olion ominaisuuksia ja suhteita tulee voida operoida vastaavien get/set -metodien kautta (Bauer & King, 2007).

Sovellusalueen mallin olioihin voidaan tarvittaessa sisällyttää myös liiketoimintalogiikkaa (Fowler, 2002). Tässä tutkielmassa noudatetaan lähestymistapaa, jossa liiketoimintalogiikka toteutetaan sovelluksen ylemmillä kerroksilla ja POJO-olioiden toteutukset pyritään pitämään mahdollisimman yksinkertaisina. Fowler (2003) nimeää tällaisen lähestymistavan *antisuunnittelumalliksi*, joka on vastoin olioperustaisen suunnittelun periaatteita, ja esittää sen johtavan proseduraaliseen ohjelmointiin, kun logiikka erotetaan sovellusalueen käsitettä mallintavasta oliosta. Kirjallisuudesta löytyy kuitenkin näkemyksiä myös lähestymistavan puolesta (Marinescu, 2002; Walls & Breidenbach, 2005).

```
public class Course implements Serializable {
	private String name;
	private String code;
	private Teacher teacher;
	private List<Student> students;

	public Course() {
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public String getCode() {
		return code;
	}

	public void setCode(String code) {
		this.code = code;
	}

	public Teacher getTeacher() {
		return teacher;
	}

	public void setTeacher(Teacher teacher) {
		this.teacher = teacher;
	}

	public List<Student> getStudents() {
		return students;
	}

	public void setStudents(List<Student> students) {
		this.students = students;
	}
}
```

Kuva 13: Esimerkki kurssia mallintavasta POJO-oliosta.

4.3.3 Pysyvyys osana sovellusaluelogiikan kerrosta

Java-ohjelmistoalustalle on kehitetty tekniikoita, jotka toteuttavat olioiden mallintaman tiedon automaattisen muuntamisen ohjelmiston ja relaatiotietokannan välillä. Käsitteellisellä tasolla tekniikoissa on kyse *olio-relaatio-mallinnuksesta* (ORM, Object Relational Mapping). Olio-relaatio-mallinnusta käytetään ratkaisemaan *olioparadigman ja relaatioparadigman yhteensopimattomuus* (object-relational impedance mismatch), jolla tarkoitetaan paradigmojen välisiä käsitteellisiä ja teknisiä eroavaisuuksia. Käytännön tasolla olio-relaatio-mallinnus -tekniikan hyödyntäminen mahdollistaa pysyvän tiedon operoinnin oliolähtöisesti

siten, että sovellusalueen mallin luokat ja luokkien väliset yhteydet vastaavat relaatiotietokannan tauluja ja taulujen välisiä relaatioita (Bauer & King, 2007).

JSR 220 -spesifikaation (Java Community Process, 2005) oheistuotoksena kehittynyt JPA (Java Persistence API) määrittää joukon pysyvyyden hallintaan liittyviä rajapintoja, sekä yksinkertaisen JPQ (Java Persistence Query Language) -kyselykielen. JPQ:lla mahdollistetaan relaatioalgebran projektio (project), liitos (join) ja valinta (select) -operaatiot. Toteutuksen JPA:lle tarjoavat olio-relaatio-mallinnus -tekniikat (Bauer & King, 2007). JPA julkaistiin osana Java EE 5.0 -standardia (Java Community Process, 2006), jolloin siitä tuli virallinen osa Java-ohjelmointikieltä. Julkaisun myötä olio-relaatio-mallinnus -tekniikan toteutuksille on käytettävissä yhteiset rajapinnat ja kyselykieli, jotka mahdollistavat ohjelmakomponenttien riippumattomuuden varsinaisesta olio-relaatio-mallinnus -toteutuksesta sekä käytetystä relaatiotietokannan toteutuksesta (Bauer & King, 2007). Kerrosarkkitehtuurin näkökulmasta JPA määrittelee ja toteuttaa rajapinnan sovellusaluelogiikan kerroksen ja tietolähdekerroksen välille.

JPA sisältää määritellyn joukon *annotaatioita* (annotation), joilla pysyvät luokat, niiden attribuutit ja yhteydet kuvataan tietokantaan (Bauer & King, 2007). Kuvassa 14 esitellään edellä kuvattu kurssia mallintava POJO-olio (kuva 13), johon on lisätty JPA:n mukaiset annotaatiot. `@Entity` -annotaatiolla ilmennetään kyseessä olevan pysyvä luokka. `@Table` -annotaatio määrittää käytettävän relaatiotietokantataulun nimen. Luokkaan on lisätty myös uusi luokkamuuttuja `id` ja sille get/set -metodit, joiden tarkoitus on kuvata tietokantataulun primääriavainta. Annotaatiolla `@Id` kuvataan tietokannassa primääriavainta vastaavaa kenttää ja `@GeneratedValue` -annotaatio parametreineen kertoo strategian, jolla primääriavain luodaan. `@Column` -annotaation avulla voidaan asettaa taulurajoitteita, kuten `nullable="false"`, joka estää NULL-arvojen tallentamisen tietokantaan. `@ManyToOne` ja `@ManyToMany` -annotaatioilla kuvataan relaatioita muihin tietokantatauluihin ja määritellään lisäksi missä tilanteissa hyödynnetään vyörytystä (cascade), jolla tässä yhteydessä tarkoitetaan olion ja sen viittauksista koostuvaan rakenteen eli oliopuun vyörytystä. Liitteessä 4 esitetään kokonaisuudessaan esimerkkisovelluksen sovellusalueen mallin luokat, jotka muodostavat JPA annotaatioiden kautta liitteessä 3 esitetyn tietokantarakenteen.

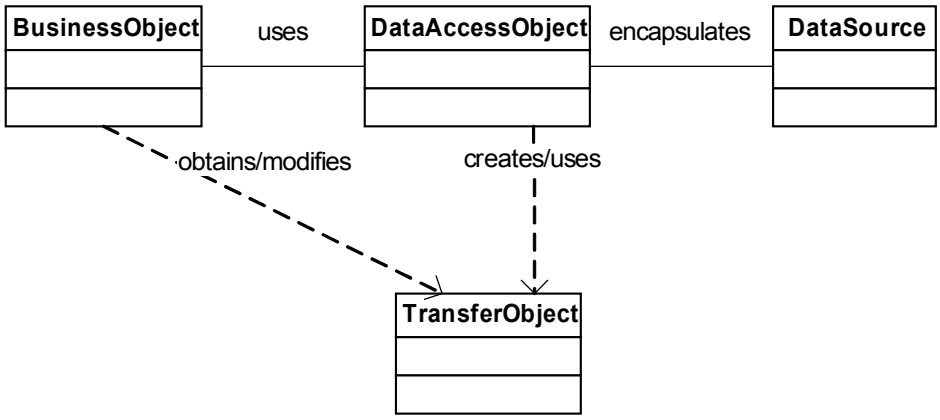
```
@Entity
@Table(name="course")
public class Course implements Serializable {
	private Long id;
	private String name;
	private String code;
	private Teacher teacher;
	private List<Student> students;

	public Course() {
	}

	@Id @GeneratedValue(strategy=GenerationType.AUTO)
	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	@Column(nullable=false)
	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	@Column(nullable=false, unique=true)
	public String getCode() {
		return code;
	}

	public void setCode(String code) {
		this.code = code;
	}

	@ManyToOne(cascade={CascadeType.PERSIST, CascadeType.MERGE})
	public Teacher getTeacher() {
		return teacher;
	}

	public void setTeacher(Teacher teacher) {
		this.teacher = teacher;
	}

	@ManyToMany(mappedBy="courses",
			cascade={CascadeType.PERSIST, CascadeType.MERGE})
	public List<Student> getStudents() {
		return students;
	}

	public void setStudents(List<Student> students) {
		this.students = students;
	}
}
```

Kuva 14: Esimerkki JPA-standardin mukaisesti annotoidusta kurssia mallintavasta oliosta.

4.3.4 Yhteys tietolähdekerrokseen

Ennen kuin edellä kuvattu kurssia mallintava olio (kuva 14) saadaan pysyväksi tietolähdekerrokseen, on sovellusaluelogiikan kerrokseen toteutettava tarvittava logiikka. Sovellusalueen mallin luokat on suositeltava pitää erillään pysyvyyteen liittyvästä logiikasta, jotta niiden tehtävä ja käyttötarkoitus on vain sovellusalueen mallintaminen (Bauer & King, 2007).

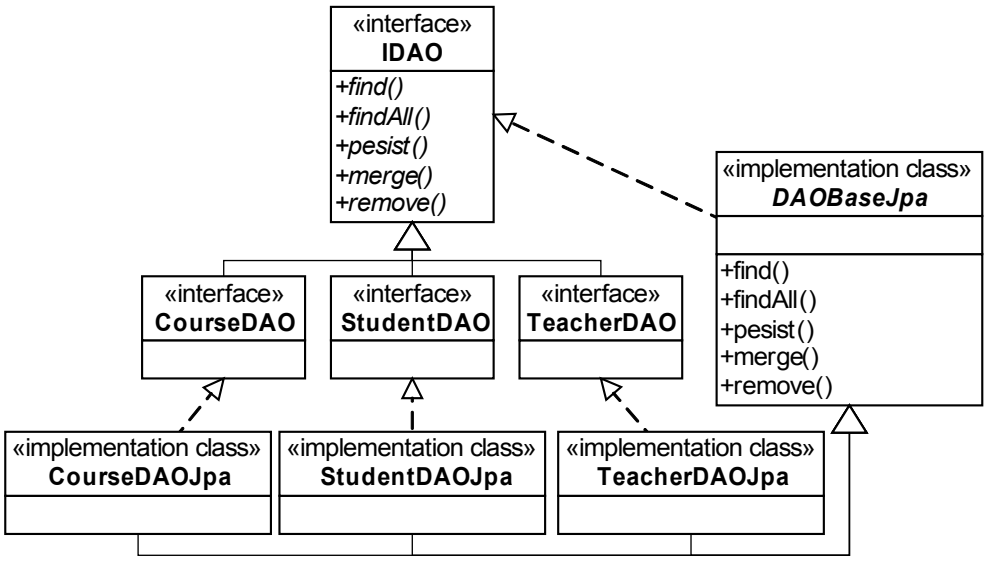
DAO (Data Access Object) -suunnittelumalli (Sun Microsystems, 2002a) on yleisesti hyödynnetty ja käytännössä hyväksi todettu suunnittelumalli tietolähteen ja pysyvyyteen liittyvän logiikan abstrahointiin (Bauer & King, 2007; Walls & Breidenbach, 2005). Kerrosarkkitehtuurin näkökulmasta DAO-suunnittelumallin toteuttavat oliot tarjoavat palvelurajapinnat ylemmille kerroksille tietolähdekerroksen käyttöä varten ja piilottavat mahdolliset alemmat kerrokset. Oleellista on luoda erikseen DAO-rajapinnat ja rajapintoja vastaavat toteutukset, jolloin ylemmät kerrokset rekisteröivät käyttöönsä pelkän rajapinnan eivätkä ole riippuvaisia sen varsinaisesta toteutuksesta. DAO-olion käyttötarkoitusta ja yhteyksiä ympäristöön on havainnollistettu kuvassa 15, jossa liiketoimintaolio (BusinessObject) käyttää DAO-oliota (DataAccessObject) tiedon pysyvyyteen liittyvissä toiminnoissa. DAO-olio kapseloi käyttämänsä tietolähteen (DataSource) liiketoimintaoliolta. Tiedon välitykseen liiketoimintaolion ja DAO-olion välillä käytetään tietoa mallintavaa oliota (TransferObject), jonka liiketoimintaolio voi tarjota DAO-oliolle, tai DAO-olio voi luoda sen itse. Esimerkkisovelluksen tapauksessa tietoa mallintavat oliot ovat ilmentymiä sovellusalueen mallin luokista (liite 2) ja liiketoimintaoliot koostuvat palvelukerroksen olioista (kuva 22), jotka kuvataan myöhemmin palvelukerroksen yhteydessä.



Kuva 15: DAO-olion yhteydet ympäristöön (Sun Microsystems, 2002a).

4.3.5 Kerroksen DAO-toteutus käytännössä

Lähestymistapa DAO-suunnittelumallin toteutukselle on alun perin omaksuttu Bauerin & Kingin (2007) kuvaamasta geneerisestä esimerkistä. DAO-suunnittelumallin toteutus esimerkkisovelluksen ympäristöön sovitettuna esitellään kuvassa 16, jota on selkeyden vuoksi yksinkertaistettu ilmentämällä vain tärkeimmät operaatiot. Toteutus koostuu rajapintaluokista `IDAO`, `CourseDAO`, `StudentDAO` ja `TeacherDAO`. `IDAO` on geneerinen rajapinta, joka sisältää perusoperaatiot: luonti, haku, päivitys ja poistaminen, jotka yleisemmin tunnetaan CRUD (Create-Read-Update-Delete) -toiminnallisuutena (Fowler, 2000; Bauer & King, 2007). `DAOBaseJpa` on abstrakti kantaluokka, joka toteuttaa `IDAO`-rajapinnan JPA:n avulla sekä tarjoaa lisäoperaatioita sitä laajentaville luokille. `CourseDAOJpa`, `StudentDAOJpa` ja `TeacherDAOJpa` luokat toteuttavat jokainen oman DAO-rajapintansa sekä laajentavat `DAOBaseJpa` kantaluokkaa, jolloin ne perivät myös toteutuksen `IDAO`-rajapinnan metodeille.



Kuva 16: Esimerkkisovelluksen hyödyntämä DAO-suunnittelumallin toteutus.

Kuvassa 17 esitellään `IDAO`-rajapinta. Rajapinnan määrittelyssä geneerinen tyyppi `T` kuvaa olion luokan, jonka pysyvyyden hallintaan rajapinnan ilmentyvät erikoistuvat. Metodit `persist`, `merge` ja `remove` vaativat parametrina `T` -tyyppisen olion. Metodia `find` käytetään pysyvän `T` -tyyppisen olion hakemiseen ja se vaatii parametrina `java.io.Serializable` -tyyppisen olion, jolla viitataan olion yksilöivään `id` -arvoon, kuten aiemmin ilmennetty kurssi-olion kuvauksessa (kuva 14). Metodilla `findAll` palautetaan kaikki pysyvät `T` -tyyppiset oliot. Metodia `persist` käytetään tekemään `T`

-tyyppisen olion tilasta pysyvä. Metodia `merge` käytetään päivittämään pysyvän `T` -tyyppisen olion tilaa ja metodilla `remove` poistetaan `T` -tyyppisen olion pysyvyys.

```
public interface IDAO<T> {
      T find(Serializable id);
      List<T> findAll();
      void persist(T entity);
      T merge (T entity);
      void remove(T entity);
}
```

Kuva 17: Geneerinen IDAO-rajapinta.

Kuvassa 18 esitetään abstrakti kantaluokka `DAOBaseJpa`, joka toteuttaa edellä esitellyn `IDAO`-rajapinnan (kuva 17) JPA:n avulla. Luokan määrittelyssä geneerinen tyyppi `T` kuvaa olion luokan, jonka pysyvyyden hallintaan ilmentymä erikoistuu. Luokan toteutuksessa käytetty `EntityManager` on rajapinta JPA:n sisältämästä `javax.persistence` paketista, joka tarjoaa valtaosan operaatioista pysyvyyden hallintaan. Luokan konstruktoria on kuormitettu, jotta käytettävä `EntityManager` ilmentymä voidaan määritellä myös muulla keinoin kuin luokan ilmentymän luontihetkellä. `@PersistenceContext` -annotaatiota voidaan käyttää ilmaisemaan riippuvuutta `EntityManager` kontekstista, jota voidaan hyödyntää esim. pysyvyyden hallintaan erikoistuneissa toteutuksissa, kuten Spring-sovelluskehyksessä (Johnson *& al.*, 2008; SpringSource, 2008). Lisäksi metodissa `findAll` hyödynnetään JPQ-kyselykieltä palauttamaan kaikki `T` -tyyppiset pysyvät oliot.

`IDAO`-rajapintaa laajentavat rajapinnat erikoistuvat tietyn sovellusalueen mallin luokan pysyvyyden operointiin. Kuvassa 19 esitetään kurssi-olioiden (kuva 14) operointiin erikoistunut `CourseDAO`-rajapinta, joka määrittää lisäksi metodin `getByCode` kurssin löytämiseen kurssikoodin perusteella.

Kuvassa 20 esitetään `CourseDAOJpa`, joka on `CourseDAO`-rajapinnan toteutus JPA:n avulla. Metodi `remove` korvaa perityn toteutuksen poistamalla kurssilta kaikki sille ilmoittautuneet oppilaat ja sen opettajan ennen kurssin poistoa. Metodi `getByCode` hyödyntää JPQ-kyselykieltä etsimällä kurssin parametrina saadun arvon perusteella. Esitellyn DAO-suunnittelumallin hyödyntämistä käytännössä kuvataan esimerkkisovelluksen palvelukerroksen toteutuksen yhteydessä.

```
public abstract class DAOBaseJpa<T> implements IDAO<T> {
	protected Class<T> entityClass;
	private EntityManager em;

	protected DAOBaseJpa(Class<T> entityClass) {
		this(entityClass, (EntityManager) null);
	}

	protected DAOBaseJpa(Class<T> entityClass, EntityManager em) {
		this.entityClass = entityClass;
		this.em = em;
	}

	public EntityManager getEntityManager() {
		return em;
	}

	@PersistenceContext
	public void setEntityManager(EntityManager em) {
		this.em = em;
	}

	public Class<T> getEntityClass() {
		return entityClass;
	}

	public T find(Serializable id) {
		try {
			return getEntityManager().find(getEntityClass(), id);
		} catch (NoResultException e) {
			return null;
		}
	}

	public List<T> findAll() {
		return getEntityManager().createQuery(
			"select o from "+getEntityClass().getName()+" o")
			.getResultList();
	}

	public void persist(T entity) {
		getEntityManager().persist(entity);
	}

	public T merge(T entity) {
		return getEntityManager().merge(entity);
	}

	public void remove(T entity) {
		getEntityManager().remove(entity);
	}
}
```

Kuva 18: IDAO-rajapinnan JPA:n avulla toteuttava toteuttava DAOBaseJpa.

```
public interface CourseDAO extends IDAO<Course> {
	Course getByCode(String code);
}
```

Kuva 19: Kurssi-olioiden käsittelyyn erikoistunut CourseDAO-rajapinta.

```
public class CourseDAOJpa extends DAOBaseJpa<Course> implements CourseDAO {

	public CourseDAOJpa() {
		super(Course.class);
	}

	public CourseDAOJpa(EntityManager em) {
		super(Course.class, em);
	}

	@Override
	public void remove(Course entity) {
		for (Student student : entity.getStudents()) {
			student.getCourses().remove(entity);
		}
		entity.getStudents().clear();

		if (null != entity.getTeacher()) {
			entity.getTeacher().getCourses().remove(entity);
			entity.setTeacher(null);
		}

		super.remove(entity);
	}

	public Course getByCode(String code) {
		List<Course> results = getEntityManager()
		.createQuery("select c from Course c where c.code = :code")
		.setParameter("code", code)
		.getResultList();

		return (!results.isEmpty()) ? results.get(0) : null;
	}
}
```

Kuva 20: CourseDAO-rajapinnan JPA:n avulla toteuttava CourseDAOJpa.

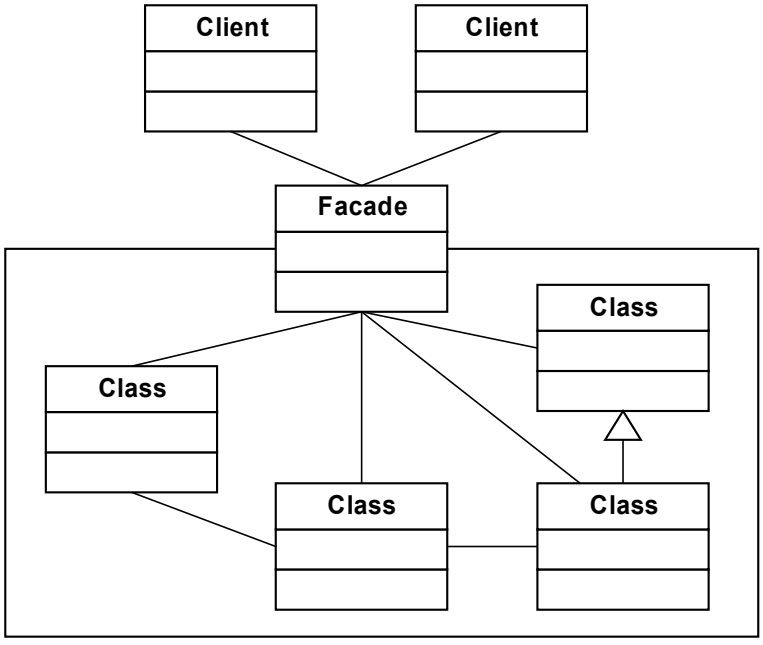
4.4 Palvelukerros

Seuraavaksi esitellään esimerkkisovelluksen kerroksista palvelukerros. Aluksi perehdytään kerroksen rakenteeseen käsitteellisellä tasolla, jonka jälkeen kerroksen sisältö ilmennetään osittaisena käytännön toteutuksena.

4.4.1 Kerroksen kuvaus

Idea palvelukerroksesta ja sen käytöstä on omaksuttu Marinescun (2002) kuvaamasta kerrosarkkitehtuurin mallista sekä Randy Staffordin esittämästä palvelukerros -suunnittelumallista (Fowler, 2002). *Palvelukerros* kapseloi sovelluksen liiketoimintalogiikan operaatioiden muodossa ja tarjoaa ylemmille kerroksille palvelurajapinnat operaatioiden kutsumiseen. Lisäksi palvelukerroksen tehtävä on hallita (tietokanta)transaktioita (Fowler, 2002).

Palvelukerros voidaan toteuttaa erilaisten lähestymistapojen kautta, riippuen missä kerroksessa sovelluksen liiketoimintalogiikka varsinaisesti toteutetaan (Fowler, 2002). Esimerkkisovelluksen osalta noudatetaan lähestymistapaa, jossa liiketoimintalogiikka toteutetaan palvelukerroksessa. Palvelukerros vuorovaikuttaa kerroshierarkiassa alempana olevan sovellusaluelogiikan kerroksen kanssa DAO-rajapintojen välityksellä ja delegoi osan DAO-rajapintojen operaatioista ylempien kerrosten käyttöön. Lisäksi joukosta DAO-rajapintoja koostetaan yksinkertaistetut palvelurajapinnat ylemmille kerroksille. Kyseinen lähestymistapa, jossa rajapintoja koostetaan yksinkertaisempaan muotoon, tunnetaan myös Facade-suunnittelumallina (kuva 21) (Gamma *& al*., 1995).

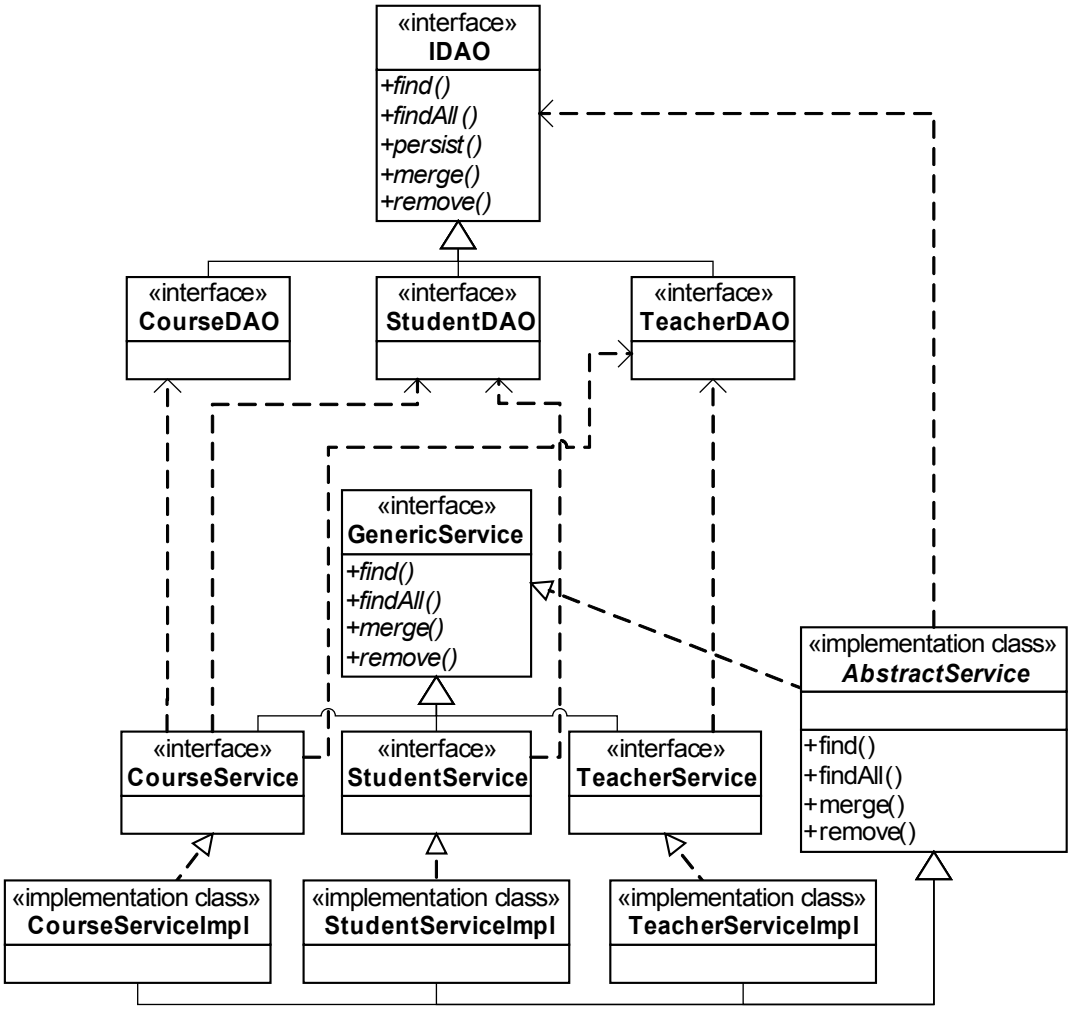


Kuva 21: Facade-suunnittelumallin rakenne (Gamma *& al*., 1995).

4.4.2 Kerroksen toteutus käytännössä

Esimerkkisovelluksen osalta palvelukerros toteuttaa myös tietokantatransaktiot työyksikkö (Unit of Work) -suunnittelumallin mukaisesti. Työyksikkö-sunnittelumallin perusajatus on jäljittää liiketoimintalogiikan olioihin tekemiä muutoksia määritellyn transaktion aikana sekä hallita muutosten vienti käytettyyn tietolähteeseen yhtenä kokonaisuutena (Fowler, 2002). Työyksikkö-suunnittelumallin avulla pyritään varmistamaan tiedon eheys ja oikeellisuus sekä tehostamaan tiedonhallintaa. Käytännön toteutuksessa hyödynnetään Spring-sovelluskehystä, joka tarjoaa valmiin transaktiohallinan toteutuksen sovelluspalvelinympäristöön sekä mahdollistaa metodikohtaisesti määriteltävät transaktiorajat, jolla toteutetaan työyksikkö-suunnittelumallin mukainen transaktionhallinta (Johnson *& al.*, 2008). Transaktionhallinta on mahdollista toteuttaa myös muulla keinoin, kuten JTA:n (Java Transaction API) avulla, mutta kaikki sovelluspalvelimet eivät toteuta tätä rajapintaa (The Apache Software Foundation, 2008a).

Esimerkkisovelluksen palvelukerroksen rakenne esitellään kuvassa 22, jota on selkeyden vuoksi yksinkertaistettu ilmentämällä vain tärkeimmät operaatiot. Kuvassa tuodaan esille myös yhteydet sovellusaluelogiikan rajapintoihin. Kerroksen rakenne muodostuu rajapintaluokista `GenericService`, `CourseService`, `StudentService` ja `TeacherService`. `GenericService` on geneerinen rajapinta, joka määrittää perustoiminnallisuuden kaikille Service-tyyppisille rajapinnoille. `AbstractService` on abstrakti kantaluokka, joka toteuttaa `GenericService`-rajapinnan käyttäen sovellusaluelogiikan kerroksen tarjoamia DAO-rajapintoja (kuva 16). `CourseServiceImpl`, `StudentServiceImpl` ja `TeacherServiceImpl` luokat toteuttavat jokainen oman Service-rajapintansa ja laajentavat `AbstractService` kantaluokkaa, jolloin ne perivät myös toteutuksen `GenericService`-rajapinnan metodeille. Palvelukerroksen rakenne vastaa suurelta osin aiemmin esiteltyä DAO-suunnittelumallin toteutuksen rakennetta, joten kerroksen toteutuksesta esitellään seuraavaksi vain olennaisimmat kohdat. Oleellista on luoda erikseen Service-tyyppiset rajapinnat ja rajapintoja vastaavat toteutukset, jolloin ylemmät kerrokset rekisteröivät käyttöönsä pelkän rajapinnan eivätkä ole riippuvaisia sen varsinaisesta toteutuksesta.



Kuva 22: Esimerkkisovelluksen palvelukerroksen rakenne ja yhteydet sovellusaluelogiikan kerrokseen.

Kuvassa 23 esitellään `GenericService`-rajapinta. Rajapinnan määrittelyssä geneerinen tyyppi `T` kuvaa olion luokan, johon liittyvään liiketoimintalogiikan toteutukseen rajapinnan toteutukset erikoistuvat. Rajapinta sisältää `@Transactional` -annotaatiot, jotka määrittävät transaktiorajat ja sääntöjä transaktionhallintaan. Säännöistä `readOnly=true` ja `readOnly=false` määrittävät onko kyseessä muutoksia salliva transaktio, `propagation=Propagation.REQUIRES_NEW` määrittää metodin suorituksen tarvitsevan uuden transaktion ja mahdollisesti jo olemassa oleva transaktio keskeytetään suorituksen ajaksi (Johnson *& al.*, 2008). Metodia `find` käytetään `T` -tyyppisen olion hakemiseen sen yksilöivän arvon perusteella. Metodilla `findAll` palautetaan kaikki `T` -tyyppiset oliot. Metodia `merge` käytetään sekä luomaan että päivittämään `T` -tyyppisen olion tilaa ja metodilla `remove` poistetaan `T` -tyyppinen olio.

```
@Transactional(readOnly=true)
public interface GenericService<T> {

	T find(Serializable id);

	List<T> findAll();

	@Transactional(readOnly=false, propagation=Propagation.REQUIRES_NEW)
	T merge(T entity);

	@Transactional(readOnly=false, propagation=Propagation.REQUIRES_NEW)
	void remove(T entity);
}
```

Kuva 23: GenericService-rajapinta,

Kuvassa 24 esitetään abstrakti kantaluokka `AbstractService`, joka toteuttaa `GenericService`-rajapinnan (kuva 23) määrittämät operaatiot. Luokan määrittelyssä geneerinen tyyppi `T` määrittää olion, jonka liiketoimintalogiikan toteutukseen luokan ilmentymä erikoistuu ja geneerinen tyyppi `DAO` määrittää käytettävän DAO-rajapinnan (kuva 16) tyypin. Metodit edustavat sovelluksen liiketoimintalogiikkaa, jotka kantaluokka `AbstractService` toteuttaa delegoimalla vastuun käyttämälleen DAO-rajapinnalle. Luokka perii myös `GenericService`-rajapinnan määrittämät annotaatiot transaktionhallintaan.

Kuvassa 25 esitetään `GenericService`-rajapintaa laajentava `CourseService`-rajapinta, joka määrittelee kurssi-olioihin (kuva 14) liittyvän liiketoimintalogiikan. Rajapinta määrittelee metodin `getByCode`, jota käytetään tietyn kurssin hakuun kurssikoodin perusteella ja metodin `merge`, jota käytetään uuden tai jo olemassa olevan kurssin tallentamiseen. Rajapinta perii lisäksi `GenericService`-rajapinnan määrittämät annotaatiot transaktionhallintaan.

Kuvassa 26 esitetään `CourseService`-rajapinnnan toteutus `CourseServiceImpl`, joka laajentaa `AbstractService`-kantaluokkaa. Luokan tehtävä on toteuttaa kurssi-olioihin (kuva 14) liittyvä liiketoimintalogiikka sovelluksen osalta. Toteutus hyödyntää sovellusaluelogiikan kerroksen kolmea eri DAO-rajapintaa (kuva 16) tarvittavan tietosisällön hakuun, kuten aiemmin ilmennetty kuvassa 22. Metodin `merge` toteutus on esimerkki monimutkaisemmasta liiketoimintalogiikasta, jossa luodaan uusi tai päivitetään jo olemassa olevaa kurssia.

```
public abstract class AbstractService<T, DAO extends IDAO<T>>
	implements GenericService<T> {

	protected DAO dao;

	public AbstractService() {
	}

	public DAO getDao() {
		return dao;
	}

	public void setDao(DAO dao) {
		this.dao = dao;
	}

	public T find(Serializable id) {
		return dao.find(id);
	}

	public List<T> findAll() {
		return dao.findAll();
	}

	public T merge(T entity) {
		return dao.merge(entity);
	}

	public void remove(T entity) {
		dao.remove(entity);
	}
}
```

Kuva 24: GenericService-rajapinnan toteuttava AbstractService kantaluokka.

```
@Transactional(readOnly=true)
public interface CourseService extends GenericService<Course> {

	Course getByCode(String code);

	@Transactional(readOnly=false, propagation=Propagation.REQUIRES_NEW)
	Course merge(Course entity, Long teacherID, List<Long> studentIDs)
}
```

Kuva 25: CourseService-rajapinta.

```
public class CourseServiceImpl extends AbstractService<Course, CourseDAO>
      implements CourseService {

      private TeacherDAO teacherDAO;
      private StudentDAO studentDAO;

      public CourseServiceImpl() {
      }

      public StudentDAO getStudentDAO() {
            return studentDAO;
      }

      public void setStudentDAO(StudentDAO studentDAO) {
            this.studentDAO = studentDAO;
      }

      public TeacherDAO getTeacherDAO() {
            return teacherDAO;
      }

      public void setTeacherDAO(TeacherDAO teacherDAO) {
            this.teacherDAO = teacherDAO;
      }

      public Course getByCode(String code) {
            return dao.getByCode(code);
      }

      public Course merge(Course entity, Long teacherID, List<Long>
                  studentIDs) {
            if (null != teacherID && teacherID.longValue() != 0) {
                  Teacher teacher = teacherDAO.find(teacherID);
                  entity.setTeacher(teacher);
            } else {
                  entity.setTeacher(null);
            }

            List<Student> students = entity.getStudents();
            if (null == students) { //uusi ilmentymä
                  students = new ArrayList<Student>();
                  entity.setStudents(students);
            } else if (!students.isEmpty()) { //olemassa oleva ilmentymä
                  for (Student student : students) { //poistetaan oppilaat
                        Student proxy = studentDAO.find(student.getId());
                        proxy.getCourses().remove(entity);
                  }
                  students.clear();
            }
            if (null != studentIDs) { //lisätään valitut oppilaat
                  for (Long id : studentIDs) {
                        Student proxy = studentDAO.find(id);
                        proxy.getCourses().add(entity);
                        students.add(proxy);
                  }
            }
            return dao.merge(entity);
      }
}
```

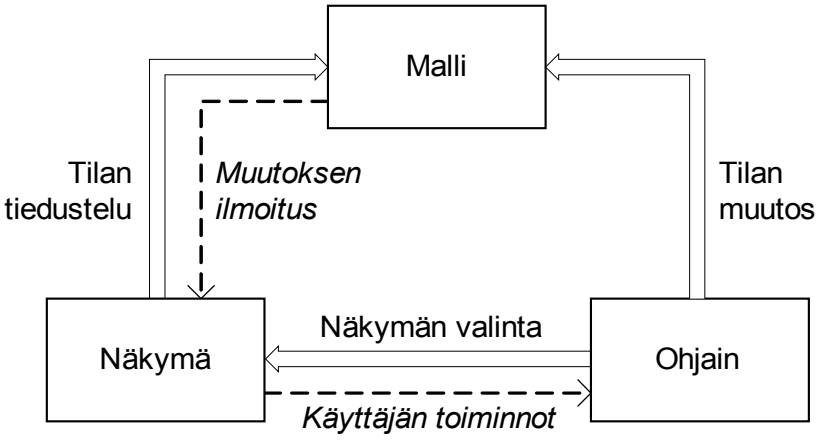
Kuva 26: CourseService-rajapinnan toteuttava CourseServiceImpl.

4.5 Sovelluslogiikan kerros

Seuraavaksi esitellään esimerkkisovelluksen kerroksista sovelluslogiikan kerros. Aluksi perehdytään kerroksen rakenteeseen käsitteellisellä tasolla, jonka jälkeen kerroksen sisältö ilmennetään osittaisena käytännön toteutuksena.

4.5.1 Kerroksen kuvaus

Sovelluslogiikan kerroksen ideana on yhdistää työnkulku (workflow) esityskerroksen ja palvelukerroksen välillä. Kerros on vastuussa sovellusta käyttävien asiakkaiden istuntojen (HttpSession) hallinnasta, esityskerroksesta toimitettujen syötteiden validoinnista sekä palvelukutsujen delegoinnista palvelukerrokseen liiketoimintalogiikkaa varten (Marinescu, 2002). Esimerkkisovelluksessa sovelluslogiikan kerroksen tehtävä on myös mallintaa käyttöliittymää, jolloin se heijastaa sovelluksen tiedon tilaa ja esittää sen erilaisissa näkymissä oikeassa kontekstissa. Sovelluslogiikan kerroksen toteutuksessa on hyödynnetty *malli-näkymä-ohjain-arkkitehtuuria* (MVC, model-view-controller) (kuva 27), jonka avulla erotetaan sovelluksen käyttöliittymä sen varsinaisesta logiikasta ja datasta. Koskimies & Mikkonen (2005) kuvaavat MVC-arkkitehtuurin Buschmannin *& al.* (1996) esittelemän määritelmän perusteella rakentuvan kolmesta osatekijästä: *malleista* (model), jotka edustavat osaa sovelluksen tiedosta tai loogisesta sovelluksen tilasta, *näkymistä* (view), jotka edustavat osaa näkyvästä käyttöliittymästä, sekä *ohjaimista* (controller), jotka toimivat sovittimena mallien ja näkymien välillä varmistaen niiden keskinäisen vastaavuuden.



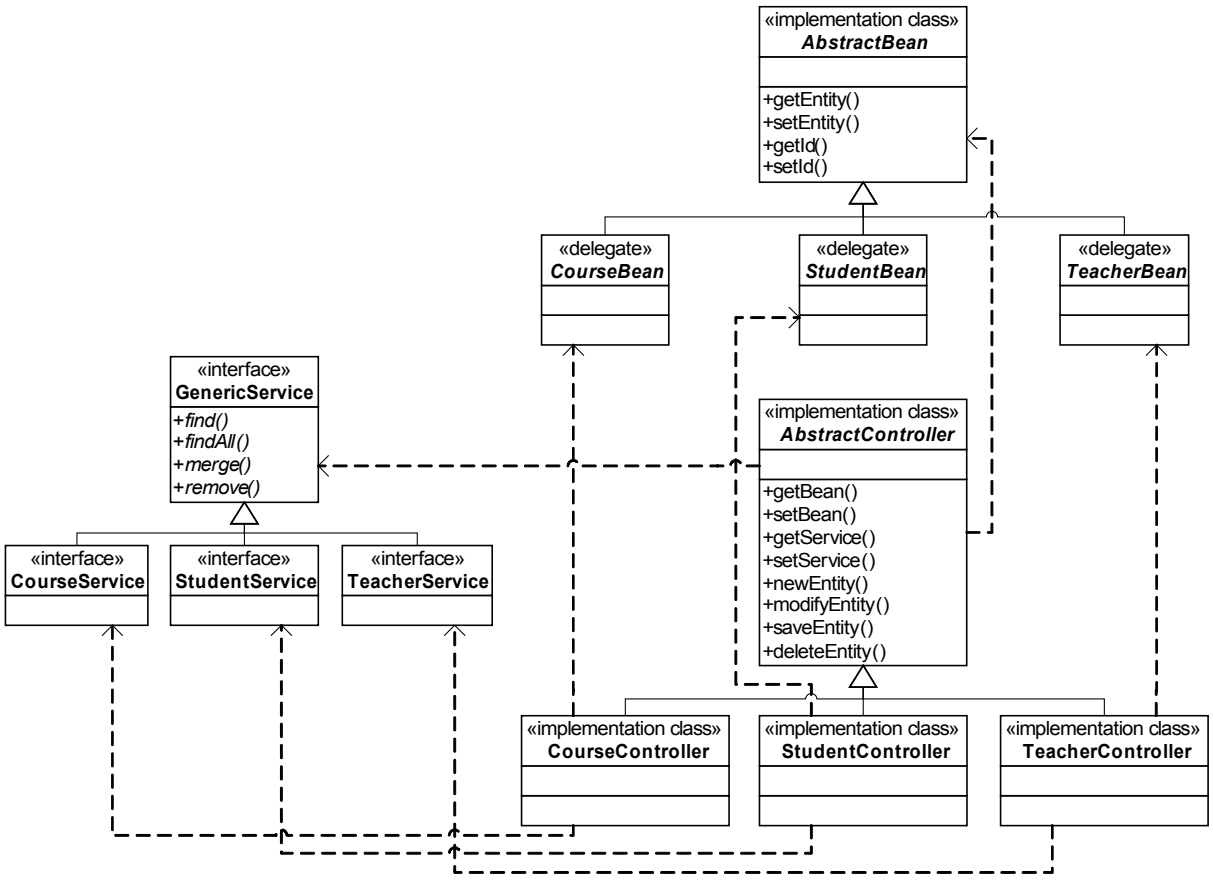
Kuva 27: Malli-näkymä-ohjain-arkkitehtuuri (Sun Microsystems, 2002b).

4.5.2 Kerroksen toteutus käytännössä

Java-ohjelmistoalustalle on toteutettu useita MVC-arkkitehtuurin perustuvia www-käyttöliittymän toteutukseen tarkoitettuja sovelluskehyksiä, joiden ideana on yksinkertaistaa ja tehostaa käyttöliittymän ja siihen liittyvän logiikan toteutusta. Tällaisia sovelluskehyksiä ovat mm. Struts (The Apache Software Foundation, 2008b), JavaServer Faces (Java Community Process, 2004) ja Spring Web MVC (Johnson *& al.*, 2008). Näistä JavaServer Faces on standardoitu, joten se valittiin esimerkkisovelluksen käyttöliittymälogiikan toteutusta varten. Kerrosarkkitehtuurin näkökulmasta JavaServer Faces -sovelluskehys määrittää ja toteuttaa rajapinnan sovelluslogiikan kerroksen ja esityskerroksen välillä (Mann, 2004).

Esimerkkisovelluksen sovelluslogiikan kerroksen rakenne esitellään kuvassa 28, jota on selkeyden vuoksi yksinkertaistettu ilmentämällä vain tärkeimmät operaatiot. Kuvassa ilmennetään myös yhteydet palvelukerroksen tarjoamiin rajapintoihin. Kerroksen rakenne muodostuu tietoa mallintavista luokista `CourseBean`, `StudentBean` ja `TeacherBean`, jotka perivät toteutuksen abstraktilta kantaluokalta `AbstractBean`. Tietoa mallintavat konkreettiset Bean-tyyppiset luokat ovat sisäiseltä toteutukseltaan delegaatteja, jotka käyttävät sovellusalueen mallin luokkia tiedon mallintamiseen ja mukauttavat tarvittaessa tiedon esitystä sovelluslogiikan kerrokselle sopivaksi. Sovelluslogiikka kursseille toteutetaan `CourseController` luokassa, oppilaille `StudentController` luokassa ja opettajille `TeacherController` luokassa, jotka perivät perustoiminnallisuuden abstraktilta kantaluokalta `AbstractController`. MVC-arkkitehtuurin kannalta ajateltuna Bean-tyyppiset luokat toimivat malleina, Controller-tyyppiset luokat yhdessä JavaServer Faces -sovelluskehyksen sisältämän FacesServletin kanssa ohjaimina ja näkymät muodostuvat esim. JSP-tekniikalla (Java Community Process, 2003a) toteutetuista käyttöliittymää kuvaavista tiedostoista (Mann, 2004).

Kuvassa 29 esitetään abstrakti kantaluokka `AbstractBean`, joka määrittää perustoiminnallisuuden kaikille Bean-tyyppisille luokille. Geneerinen tyyppi `T` kuvaa olion luokan, jonka delegointiin kantaluokan ilmentyvät erikoistuvat. `AbstractBean` määrittää lisäksi kaksi abstraktia metodia `getId` ja `setId`.



Kuva 28: Esimerkkisovelluksen sovelluslogiikan kerroksen rakenne ja yhteydet palvelukerrokseen.

Kuvassa 30 esitetään `CourseBean`, joka laajentaa `AbstractBean`-kantaluokkaa. `CourseBean` käärii sovellusalueen mallin sisältämän kurssi-olion (kuva 14) ja käyttää delegointia toistamaan kaikki sen metodit. Ideana on mallintaa lisätietoa, jota kurssi-oliosta puuttuu metodien `getTeacherFullName` ja `getStudentCount` muodossa.

Kuvassa 31 esitetään abstrakti kantaluokka `AbstractController`, joka toteuttaa perustoiminnallisuuden sovelluksen osalta MVC-arkkitehtuurin mukaisesta ohjauksesta. Luokan määrittelyssä geneerisistä tyypeistä `EntityT` määrittää sovellusalueen mallin olion ja `BeanT` määrittää Bean-tyyppisen olion, joilla rajataan millaista Bean-tyyppistä oliota `AbstractController` käyttää tiedon mallintamiseen. Geneerinen tyyppi `ServiceT` määrittää käytettävän Service-tyyppisen olion, joka toimii rajapintana palvelukerrokseen. Metodeja `newBeanEntity` ja `newBean` käytetään yhdessä luomaan ilmentymä käytettävästä Bean-tyyppisestä luokasta. Metodit `newEntity`, `modifyEntity`, `saveEntity` ja `deleteEntity` sisältävät ohjauslogiikan esityskerroksen lähettämille pyynnöille. Metodi

`newEntity` suoritetaan siirryttäessä esityskerroksessa näkymään, jossa lisätään uutta tietoa. Metodi `modifyEntity` suoritetaan siirryttäessä esityskerroksessa näkymään, jossa muokataan olemassa olevaa tietoa. Metodi `saveEntity` suoritetaan, kun esityskerroksesta tulee pyyntö tiedon tallentamiseen ja tarvittava validointi tiedon oikeellisuudelle on suoritettu onnistuneesti. Metodi `deleteEntity` suoritetaan, kun esityskerroksesta tulee pyyntö tiedon poistamiselle. Metodi `getAllAsList` suoritetaan, kun esityskerroksesta tulee pyyntö tietosisällön näyttämisestä kokonaisuudessaan, kuten pyyntö kaikkien kurssien listauksesta.

Kuvassa 32 esitetään `CourseController`, joka toteuttaa `AbstractController` kantaluokan. Kuvaa on selkeyden vuoksi yksinkertaistettu ilmentämällä vain perityt metodit jotka luokan täytyy toteuttaa. Luokka ei määritä lisätoiminnallisuutta ohjauslogiikan osalta.

Kuvassa 33 esitetään palveluketjun kulku esimerkkisovelluksen eri kerrosten välillä, kun esityskerroksesta lisätään uusi kurssi. `UI` kuvaa esityskerroksen ilmentämää näkymää, `CourseController` vastaa kurssi näkymiin liittyvästä ohjauksesta, CourseService on palvelukerroksen rajapinta, joka sisältää liiketoimintalogiikan kurssien käsittelyyn, `StudentDAO`, `TeacherDAO` ja `CourseDAO` ovat sovellusaluelogiikan rajapintoja jotka vastaavat tiedon pysyvyyteen liittyvistä operaatioista.

```
public abstract class AbstractBean<T> implements Serializable {
	protected T entity;

	public AbstractBean(T entity) {
		this.entity = entity;
	}

	public abstract Long getId();
	public abstract void setId(Long id);

	public T getEntity() {
		return entity;
	}

	public void setEntity(T entity) {
		this.entity = entity;
	}
}
```

Kuva 29: Abstrakti kantaluokka AbstractBean.

```
public class CourseBean extends AbstractBean<Course>
	implements Serializable {

	public CourseBean(Course entity) {
		super(entity);
	}

	public Long getId() {
		return entity.getId();
	}

	public void setId(Long id) {
		entity.setId(id);
	}

	public String getName() {
		return entity.getName();
	}

	public void setName(String name) {
		entity.setName(name);
	}

	public String getCode() {
		return entity.getCode();
	}

	public void setCode(String code) {
		entity.setCode(code);
	}

	//..loput delegaatit jätetty ilmentämättä


	public String getTeacherFullName() {
		Teacher teacher = entity.getTeacher();
		return (null != teacher) ? teacher.getFullNameForList() : null;
	}

	public Integer getStudentCount() {
		List<Student> entities = entity.getStudents();
		return Integer.valueOf(null != entities ? entities.size() : 0);
	}
}
```

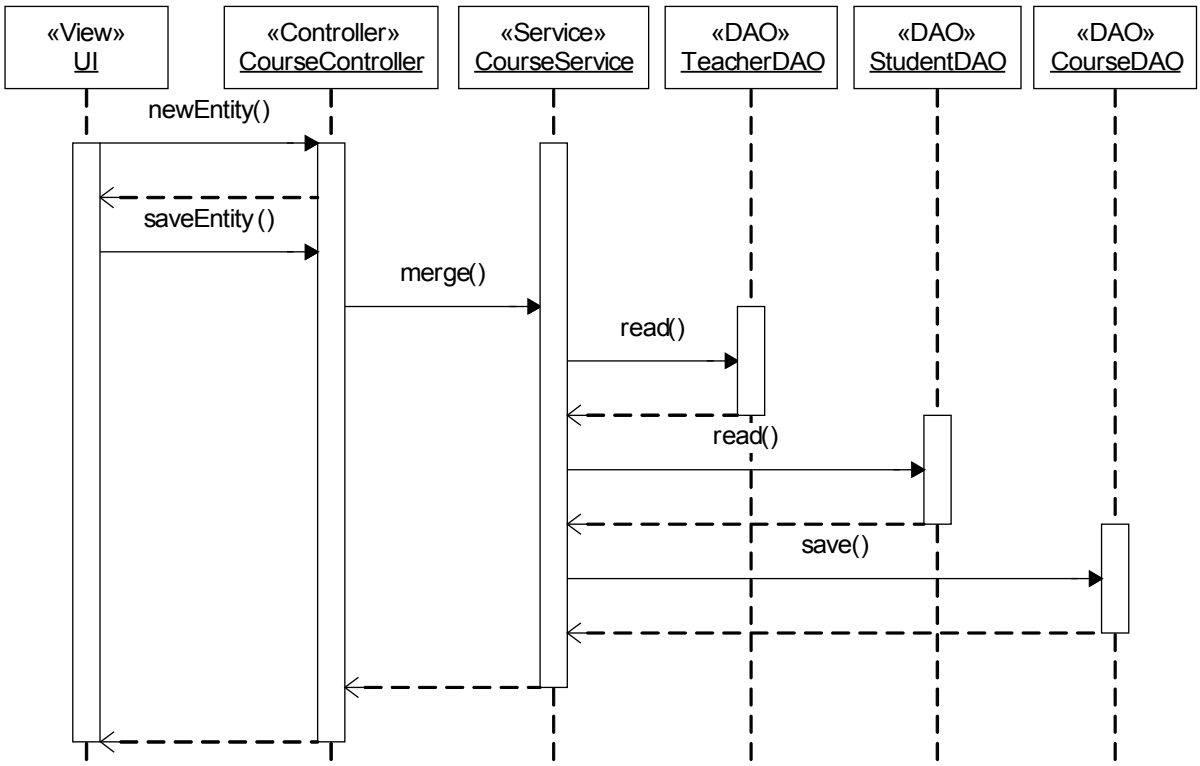
Kuva 30: Kurssi-oliota sovelluslogiikan kerroksessa mallintava CourseBean-luokka.

```
public abstract class AbstractController<EntityT,
            BeanT extends AbstractBean<EntityT>,
            ServiceT extends GenericService<EntityT>>
      implements Serializable {

      protected BeanT bean;
      protected transient ServiceT service;

      public AbstractController(BeanT bean) {
            this.bean = bean;
      }

      protected abstract EntityT newBeanEntity();
      protected abstract BeanT newBean(EntityT entity);

      public BeanT getBean() { return bean; }
      public void setBean(BeanT bean) { this.bean = bean; }

      public ServiceT getService() { return service; }
      public void setService(ServiceT service) { this.service = service; }

      public String newEntity() {
            bean.setEntity(newBeanEntity());
            return Outcome.SUCCESS;
      }

      public String modifyEntity() {
            bean.setEntity(service.find(bean.getId()));
            return Outcome.SUCCESS;
      }

      public String saveEntity() {
            EntityT entity = bean.getEntity();
            bean.setEntity(service.merge(entity));
            return Outcome.SUCCESS;
      }

      public String deleteEntity() {
            EntityT entity = service.find(bean.getId());
            service.remove(entity);
            bean.setEntity(newBeanEntity());
            return Outcome.SUCCESS;
      }

      public List<BeanT> getAllAsList() {
            return asBeans(service.findAll());
      }

      protected List<BeanT> asBeans(List<EntityT> entities) {
            List<BeanT> results = new ArrayList<BeanT>(entities.size());
            for (EntityT entity : entities) {
                  BeanT result = newBean(entity);
                  results.add(result);
            }
            return results;
      }
}
```

Kuva 31: Ohjauslogiikasta vastaava kantaluokka AbstractController.

```
public class CourseController
      extends AbstractController<Course, CourseBean, CourseService>
      implements Serializable {

...

      public CourseController() {
            super(new CourseBean(new Course()));
      }

      @Override
      protected Course newBeanEntity() {
            return new Course();
      }

      @Override
      protected CourseBean newBean(Course entity) {
            return new CourseBean(entity);
      }

...
}
```

Kuva 32: Kurssi-näkymien ohjauksesta vastaa CourseController.



Kuva 33: Palvelukutsun kulkeminen läpi kerrosten.

4.6 Esimerkkisovelluksen kokoonpanon hallinta

Yksi keskeinen ongelma eri osista koostuvissa ohjelmistoissa on osien koostaminen yhdeksi toimivaksi kokonaisuudeksi vastaamaan erilaisia käyttötarkoituksia tai ympäristön vaatimuksia (Fowler, 2004). Kerrosarkkitehtuurin näkökulmasta tilanne ilmenee esim. silloin, kun kerrosten välinen kommunikaatio täytyy toteuttaa eri ympäristössä eri tavoin, joka käytännössä tarkoittaa kerrosten välisen rajapinnan toteutuksen vaihtamista. Jos rajapinnan konkreettinen ilmentymä luodaan ohjelmakoodissa, se ei ole vaihdettavissa ilman muutoksia ohjelmakoodiin.

Esimerkkisovelluksen tapauksessa kyseinen ongelmatilanne ilmenee, jos DAO-rajapintojen (kuva 16) tai Service-rajapintojen (kuva 22) toteutukset on tarvetta vaihtaa toiseksi. Käytännöllinen lähestymistapa olisi määritellä sovelluksen eri rajapintojen toteutukset keskitetysti siten, ettei ohjelmakoodin tarvitse tehdä muutoksia. *Dependency Injection* (DI) on suunnittelumalli, jonka avulla komponenteille voidaan määrittää riippuvuuksia ulkoapäin siten, etteivät yhdistettävät komponentit ole tietoisia toisistaan (Fowler, 2004). Tällä aikaansaadaan alhainen kytkentä, joka parantaa mm. komponenttien vaihdettavuutta. DI-suunnittelumalli mahdollistaa sovelluksen kokoonpanon muuttamisen ilman vaikutuksia ohjelmakoodin, jos esim. tietyn rajapinnan toteutus halutaan vaihtaa toiseksi. Samalla vältetään tilanne, jossa rajapintaa käyttävä osapuoli joutuisi muuten tietämään minkä luokan ilmentymän ja miten se luo rajapintaa vastaan (Johnson *& al.*, 2008; SpringSource, 2008).

Esimerkkisovelluksessa hyödynnetään DI-sunnittelumallin mukaisen toiminnallisuuden toteuttavaa Spring-sovelluskehystä kokoonpanon hallintaan kerrosten välisten rajapintojen osalta. Spring-sovelluskehyksen vastuulla on luoda sovelluksen käyttämästä rajapinnasta tarvittaessa ilmentymä sovelluksen suoritusaikana ja sitoa se määriteltyä rajapintaa vasten. Näin ohjelmakoodissa ei tarvitse luoda ilmentymiä rajapinnoista. Kuvassa 34 esitetään esimerkkisovelluksen kokoonpanon hallintaa Spring-sovelluskehyksellä. Kokoonpanossa määritellään aluksi DAO-olioiden toteutukset, jonka jälkeen toteutukset sidotaan Service-rajapintojen muuttujiin. Määrittelyssä `bean` -elementti viittaa olioon, `property` -elementti viittaa olion metodiin ja attribuutti `name` vastaa metodin nimeä ilman get/set -alkuliitettä. Kyseistä lähestymistapaa hyödyntämällä DAO-rajapintojen ja Service-rajapintojen toteutukset ovat vaihdettavissa ilman muutoksia ohjelmakoodiin.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"

   <bean id="courseDAO" class="fi.joensuu.cs.th.dao.jpa.CourseDAOJpa"/>
   <bean id="studentDAO" class="fi.joensuu.cs.th.dao.jpa.StudentDAOJpa"/>
   <bean id="teacherDAO" class="fi.joensuu.cs.th.dao.jpa.TeacherDAOJpa"/>

   <bean id="courseService"
         class="fi.joensuu.cs.th.service.impl.CourseServiceImpl">
      <property name="dao" ref="courseDAO"/>
      <property name="studentDAO" ref="studentDAO"/>
      <property name="teacherDAO" ref="teacherDAO"/>
   </bean>
   <bean id="studentService"
         class="fi.joensuu.cs.th.service.impl.StudentServiceImpl">
      <property name="dao" ref="studentDAO"/>
   </bean>
   <bean id="teacherService"
         class="fi.joensuu.cs.th.service.impl.TeacherServiceImpl">
      <property name="dao" ref="teacherDAO"/>
   </bean>

</beans>
```

Kuva 34: Esimerkkisovelluksen kokoonpanon hallintaa Spring-sovelluskehyksellä.

5 Yhteenveto

Tutkielmassa perehdyttiin ohjelmistoarkkitehtuurin käsitteeseen ja tarkasteltiin sen kehitystä historiallisesta näkökulmasta sekä merkitystä ohjelmistojen suunnittelua tukevana käsitteenä. Ohjelmistoarkkitehtuuri auttaa hahmottamaan ohjelmistojen rakennetta ja käyttäytymistä korkean abstraktiotason tason näkökulmasta sekä tarjoaa lähestymistavan järjestelmän, sen osien ja osien välisten suhteiden kuvaamiseen. Ohjelmistoarkkitehtuuri mahdollistaa myös sidosryhmien kommunikoinnin järjestelmästä yhteisellä käsitteellisellä tasolla tarjoamalla vaaditun abstraktiotason sanaston ja käsitteistön.

Ohjelmiston ja sen arkkitehtuurin suunnittelua koskeviin ongelmiin on suunnittelumallien muodossa hyödynnettävissä uudelleenkäytettäviä ja teknologiariippumattomia ratkaisumalleja. Suunnittelumallin sisältämä informaatio tarjoaa ratkaisumallin lisäksi kontekstin varsinaiselle ongelmalle sekä ratkaisun soveltamisesta johtuville seurauksille. Arkkitehtuurityylit ovat suunnittelumallien idean yleistys koko järjestelmän arkkitehtuuria kantavaksi periaatteeksi, jotka auttavat hahmottamaan järjestelmää sekä selittämään sen toteutuksen rakenteen. Kerrosarkkitehtuuri on ositukseen perustuva arkkitehtuurityyli, joka kuvaa ohjelmiston rakennetta kerroksina, jotka on järjestelty jonkin abstrahointiperiaatteen mukaan nousevaan järjestykseen. Eri kerrosten abstraktiotasojen tunnistaminen ei kuitenkaan aina ole triviaalia, jolloin voidaan soveltaa kerrostuksen perusideaa, jonka mukaan abstraktiotasoltaan korkeammat palvelut toteutetaan abstraktiotasoltaan matalampien palvelujen avulla. Kerrosarkkitehtuuria voidaan soveltaa yhdessä muiden arkkitehtuurityylien kanssa, jolloin se tarjoaa vaihtoehtoisen näkökulman järjestelmän arkkitehtuurille. Esimerkiksi asiakas-palvelin-arkkitehtuurin perustuvissa järjestelmissä kerrosarkkitehtuuria voidaan hyödyntää ryhmittelemään palveluja tiettyihin arkkitehtuurikerroksiin, sekä loogisella että fyysisellä tasolla.

Tutkielmassa esiteltiin esimerkkien kautta kerrosarkkitehtuuriin perustuvan Java-sovelluksen suunnittelua ja toteutusta, jonka kautta pyrittiin ilmentämään eri arkkitehtuurityylien, suunnittelumallien ja teknisten ratkaisujen hyödyllisyys ja yhteensovittaminen käytännönläheisestä näkökulmasta. Esimerkkisovelluksen kerrosarkkitehtuurin rakenteen suunnittelussa keskeinen ongelma oli pysyvyyteen ja tiedonhakuun liittyvien käsitteiden ja logiikan sijoitus. On kyseenalaista kuuluuko DAO-suunnittelumallin toteutus ja sen sisältämä

pysyvyyteen liittyvä logiikka ensinkään sovellusaluelogiikan kerrokseen vai kenties täysin omaksi kerroksekseen. Lähtökohdiltaan esimerkkisovellus suunniteltiin käyttämään omaa kerrosta DAO-suunnittelumallin toteutukselle, joka olisi ollut sovellusaluelogiikan kerroksen alapuolella lähimpänä tietolähdekerrosta. Tämä kerrosjako johti kuitenkin hankaliin riippuvuussuhteisiin alempien kerrosten välillä, jonka seurauksena pysyvyyteen liittyvä logiikka yhdistettiin sovellusalueen kerrokseen. Esimerkkisovelluksen toteutusvaiheen edetessä kävi myös ilmeiseksi, ettei toteutettu DAO-suunnittelumalli ole joustava tilanteissa, jossa sovellusalueen mallin luokalle on tarvetta suorittaa vain perus CRUD-toiminnallisuuden käsittäviä operaatioita. Toteutettu DAO-suunnittelumalli vaatii jokaiselle sovellusalusalueen mallin luokalle pysyvyyden operointia varten erikseen rajapinnan ja sen toteutuksen. Parempi ratkaisumalli olisi ollut käyttää tässä tilanteessa hyvin geneerisesti määriteltyä rajapintaa, jonka toteutus voisi operoida minkä tahansa luokan pysyvyyttä. Tilanteesta muodostuisi merkittävämpi ongelma, jos sovellusalueen malliin kuuluisi useampia luokkia, jolloin tuloksena olisi ylimääräinen ohjelmakoodin toisto. Esimerkkisovelluksen kerrosarkkitehtuuri suunniteltiin alunperin noudattamaan suljettua kerrosarkkitehtuuria, mutta muutettiin myöhemmin avoimeksi kerrosarkkitehtuuriksi. Tämä oli seurausta siitä, että sovellusalueen mallin luokkia oli mahdollista hyödyntää tiedon mallintamisessa myös ylemmillä kerroksilla ja suljetun kerrosarkkitehtuurin mukaisesti ylempi kerros saa käyttää vain lähimmän alemman kerroksen komponentteja ja palveluja. Tällöin kerrosten välille olisi jouduttu tekemään delegaatteja sovellusalueen mallin luokkia varten ilman sen suurempaa hyötyä. Palvelukerroksen osalta ajatuksia herätti kerroksen tarpeellisuus, sillä liiketoimintalogiikan toteutus muodostui paljolti delegoinnista sovellusalueen kerrokselle pelkästään pysyvyyttä varten. Olisiko parempi lähestymistapa ollut toteuttaa kaikki liiketoimintalogiikka suoraan sovellusalueen kerroksessa, mahdollisesti liiketoimintaan liittyvässä sovellusalueen mallin oliossa ja jättää palvelukerroksen vastuulle vain tietokantatransaktioiden hallinta, jää jatkotutkimusten varaan.

Esimerkkisovelluksen toteutuksesta heränneiden ajatusten perusteella kerrosarkkitehtuurin soveltaminen käytännössä on toimiva lähestymistapa hahmottaa monimutkaiselta vaikuttavaa järjestelmää ja ryhmitellä sen palveluja. Jokainen kerros oli mahdollista hahmottaa omana osakokonaisuutena sovelluksen toiminnassa, jolloin yksittäisen kerroksen vastuut ja riippuvuudet olivat paremmin tunnistettavissa. Kerrosarkkitehtuurin hyödyntäminen lisää myös uudelleenkäytettävyyttä, sillä jos samalle sovellusalueelle toteutettaisiin toinen järjestelmä olisivat sovellusalueen kerros ja palvelukerros uudelleenkäytettävissä. Vastaavasti

jos sovelluksen käyttöliittymää olisi tarvetta esittää www-selaimen ohella myös toisenlaisessa ympäristössä, tulisi vain sovelluslogiikan kerros ja esityskerros korvata. Kerrosarkkitehtuurin soveltaminen sen periaatteiden mukaisesti sisältää kuitenkin oman haasteellisuutensa. Jos kyseessä on yksinkertainen järjestelmä tai jos uudelleenkäytettävyys ei ole laatutavoitteena, on kerrosarkkitehtuurin soveltamisen järkevyyttä syytä harkita.

Viitteet

Alexander, C., Ishikawa, S., Silverstein, M. A. (1977) *Pattern Language*. Oxford University Press, New York, USA.

Bass, L., Clements, P., Kazman, R. (1998) *Software architecture in practice*. Addison-Wesley Inc., Boston, USA.

Bauer, C., King, G. (2007) *Java Persistence With Hibernate*. Manning Publications, Greenwich, USA.

Bennett, S., McRobb, S., Farmer, R. (2006) *Object-Oriented Systems Analysis and Design Using UML (3rd ed.)*. McGraw Hill Higher Education, UK.

Booch, G. (1991) *Object-Oriented Analysis And Design With Applications*. Benjamin-Cummings Publishing Co., Redwood City, USA.

Brown, K., Niswonger, J., Hester, G. (2001) *Enterprise Java Programming with IBM WebSphere*. Addison-Wesley Inc., USA.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. (1996) *Pattern-Oriented Software Architecture: A System Of Patterns*. Wiley, New York, USA.

Ellis, J., Ho, L. (2001) *JDBC 3.0 Specification, Final Release*. WWW-sivusto, `http://java.sun.com/products/jdbc/download.html` (1.4.2008).

Evans, E. (2003) *Domain-Driven Design: Tacking Complexity in the Heart of Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA.

Fowler, M. (2000) *POJO - An acronym for: Plain Old Java Object*. WWW-sivusto, `http://www.martinfowler.com/bliki/POJO.html` (1.4.2008).

Fowler, M. (2002) *Patterns of Enterprise Application Architecture.* Addison-Wesley, Boston, USA.

Fowler, M. (2003) *AnemicDomainModel*. WWW-sivusto, http://martinfowler.com/bliki/AnemicDomainModel.html (1.4.2008).

Fowler, M. (2004) *Inversion of Control Containers and the Dependency Injection pattern*. WWW-sivusto, http://martinfowler.com/articles/injection.html (1.4.2008).

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts, USA.

Garlan, D., Perry, D. (1995) Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering* **21**(4), 269-274 (Saatavana myös: http://users.ece.utexas.edu/~perry/work/papers/tse21-4.pdf, 20.5.2008).

Garlan, D., Shaw, M. (1994) *An Introduction to Software Architecture.* School of Computer Science, Carnegie Mellon University, Pittsburgh, USA (Saatavana myös: http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf, 1.4.2008).

HSQLDB (2008) *HSQLDB - Lightweight 100% Java SQL Database Engine*. WWW-sivusto: http://hsqldb.org/ (1.6.2008)

IEEE (2000) *Recommended Practice for Architectural Description of Software-Intensive Systems.* IEEE Standard 1471-2000.

Java Community Process (2001) *JSR 14: Add Generic Types To The Java Programming Language*. WWW-sivusto, http://jcp.org/en/jsr/detail?id=14 (1.4.2008).

Java Community Process (2003a) *JSR 152: JavaServer Pages 2.0 Specification*. WWW-sivusto, http://jcp.org/en/jsr/detail?id=152 (1.4.2008).

Java Community Process (2003b) *JSR 153: Enterprise JavaBeansTM 2.1*. WWW-sivusto, http://www.jcp.org/en/jsr/detail?id=153 (1.4.2008).

Java Community Process (2003c) *JSR 154: Java Servlet 2.4 Specification*. WWW-sivusto, http://www.jcp.org/en/jsr/detail?id=154 (1.4.2008).

Java Community Process (2004) *JSR 127: JavaServer Faces*. WWW-sivusto, http://www.jcp.org/en/jsr/detail?id=127 (1.4.2008).

Java Community Process (2005) *JSR 220: Enterprise JavaBeans 3.0*. WWW-sivusto, http://www.jcp.org/en/jsr/detail?id=220 (1.4.2008).

Java Community Process (2006) *JSR 244: Java Platform, Enterprise Edition 5 (Java EE 5) Specification*. WWW-sivusto, http://jcp.org/en/jsr/detail?id=244 (1.4.2008).

Johnson, R., Hoeller, J., Arendsen, A., Sampaleanu, C., Harrop, R., Risberg, T., Davison, D., Kopylenko, D., Pollack, M., Templier, T., Vervaet, E., Tung, P., Hale, B., Colyer, A., Lewis, J., Leau, C., Fisher, M., Brannen, S., Laddad, R. (2008) *Spring Java/J2EE Application Framework Reference Documentation*. Spring, http://static.springframework.org/spring/docs/2.5.x/spring-reference.pdf (1.4.2008).

Koskimies, K., Mikkonen, T. (2005) *Ohjelmistoarkkitehtuurit*. Talentum Media Oy.

Kruchten, P. (1995) The 4+1 View Model of Architecture. *IEEE Software* **12**(6), 42-50 (Saatavana myös: http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf, 1.4.2008).

Laine, H. (2000) *Ohjelmistoarkkitehtuurit*. Helsingin yliopisto, Tietojenkäsittelytieteen laitos, http://www.cs.helsinki.fi/u/laine/arkki/k00/arkki1.pdf (1.4.2008).

Laine, H., Paakki, J. (2001) *Ohjelmistotuotanto, Ohjelmistosuunnittelu 1*. Helsingin yliopisto, Tietojenkäsittelytieteen laitos, http://www.cs.helsinki.fi/u/paakki/ohtuk03-luento9.pdf (1.4.2008).

Mann, K. D. (2004) *JavaServer Faces in Action*. Manning Publications, Greenwich, USA.

Marinescu, F. (2002) *EJB Design Patterns: Advanced Patterns, Processes, and Idioms*. John Wiley & Sons, Inc., USA.

OMG (2005) *Unified Modeling Language Specification,* Version 1.4.2, formal/05-04-01, ISO/IEC 19501:2005(E). Object Management Group, `http://www.omg.org/docs/formal/05-04-01.pdf` (11.6.2008).

Perry, D. E., Wolf, A. L. (1992) Foundations for the Study of Software Architecture. Software Engineering Notes. *ACM Sigsoft Software Engineering Notes* **17**(4), 40-52.

Schussel, G. (1995) *Client/Server Past, Present, and Future*. WWW-sivusto, `http://www.dciexpo.com/geos/dbsejava.htm` (1.4.2008).

Shaw, M. (1989) *Larger Scale Systems Require Higher-Level Abstractions*. Computer Science Department and Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, USA.

Shaw, M., Garlan, D. (1996) *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, USA.

SpringSource (2008) *Spring Framework*. WWW-sivusto, `http://www.springframework.org/` (1.4.2008).

Sun Microsystems (1994) *Java Reference BluePrints*. WWW-sivusto, `http://java.sun.com/reference/blueprints/` (1.4.2008).

Sun Microsystems (2002a) *Core J2EE Patterns - Data Access Object*. WWW-sivusto, `http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html` (1.4.2008).

Sun Microsystems (2002b) *Java BluePrints - Model-View-Controller.* WWW-sivusto, `http://java.sun.com/blueprints/patterns/MVC-detailed.html` (1.4.2008).

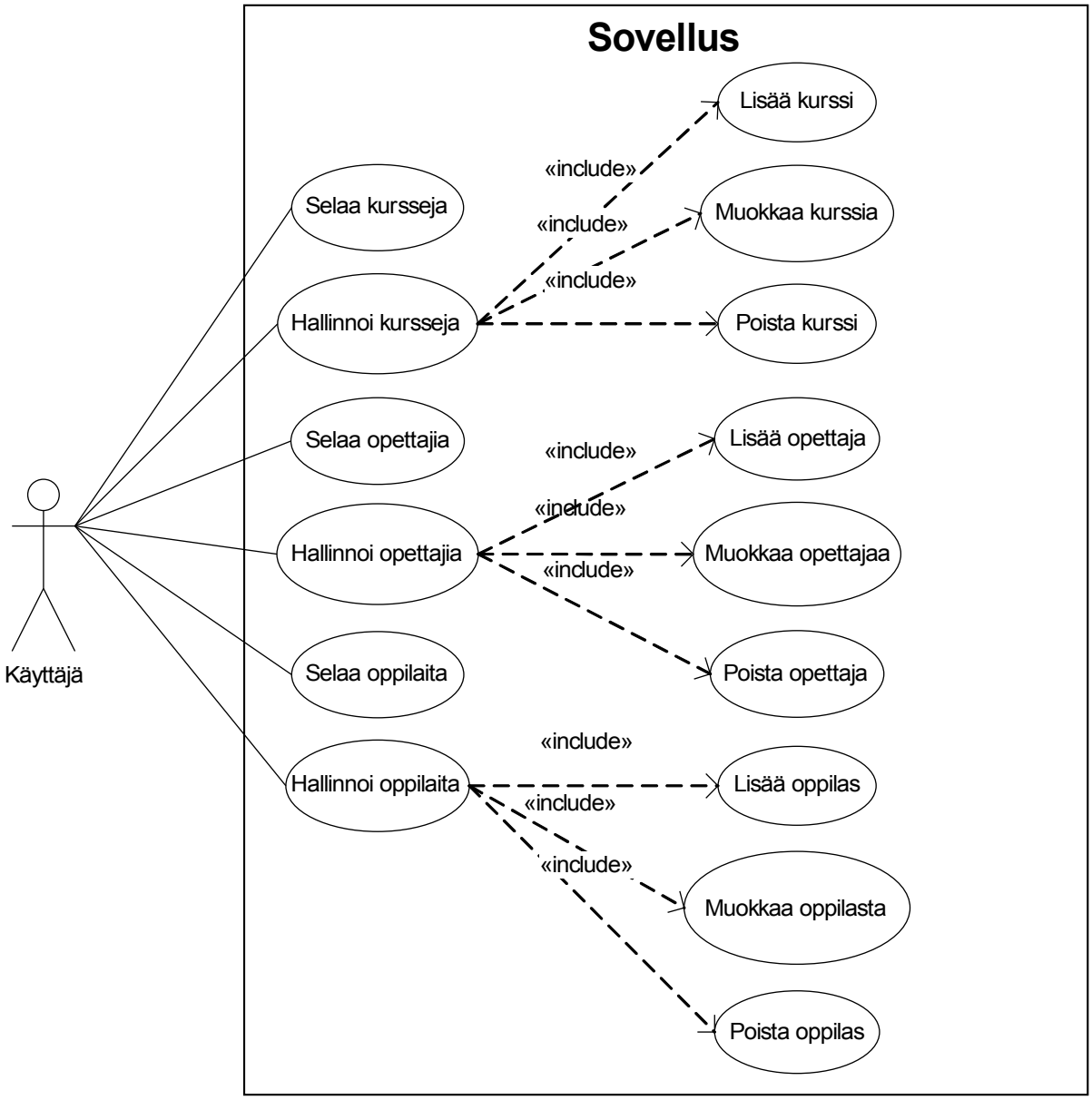
The Apache Software Foundation (2008a) *Apache Tomcat*. WWW-sivusto, `http://tomcat.apache.org/` (1.4.2008).

The Apache Software Foundation (2008b) *Apache Struts Framework*. WWW-sivusto, `http://struts.apache.org/` (1.4.2008).

Walls, C., Breidenbach, R. (2005) *Spring in Action*. Manning Publications, Greenwich, USA.

LIITE 1 : Esimerkkisovelluksen käyttötapauskaavio

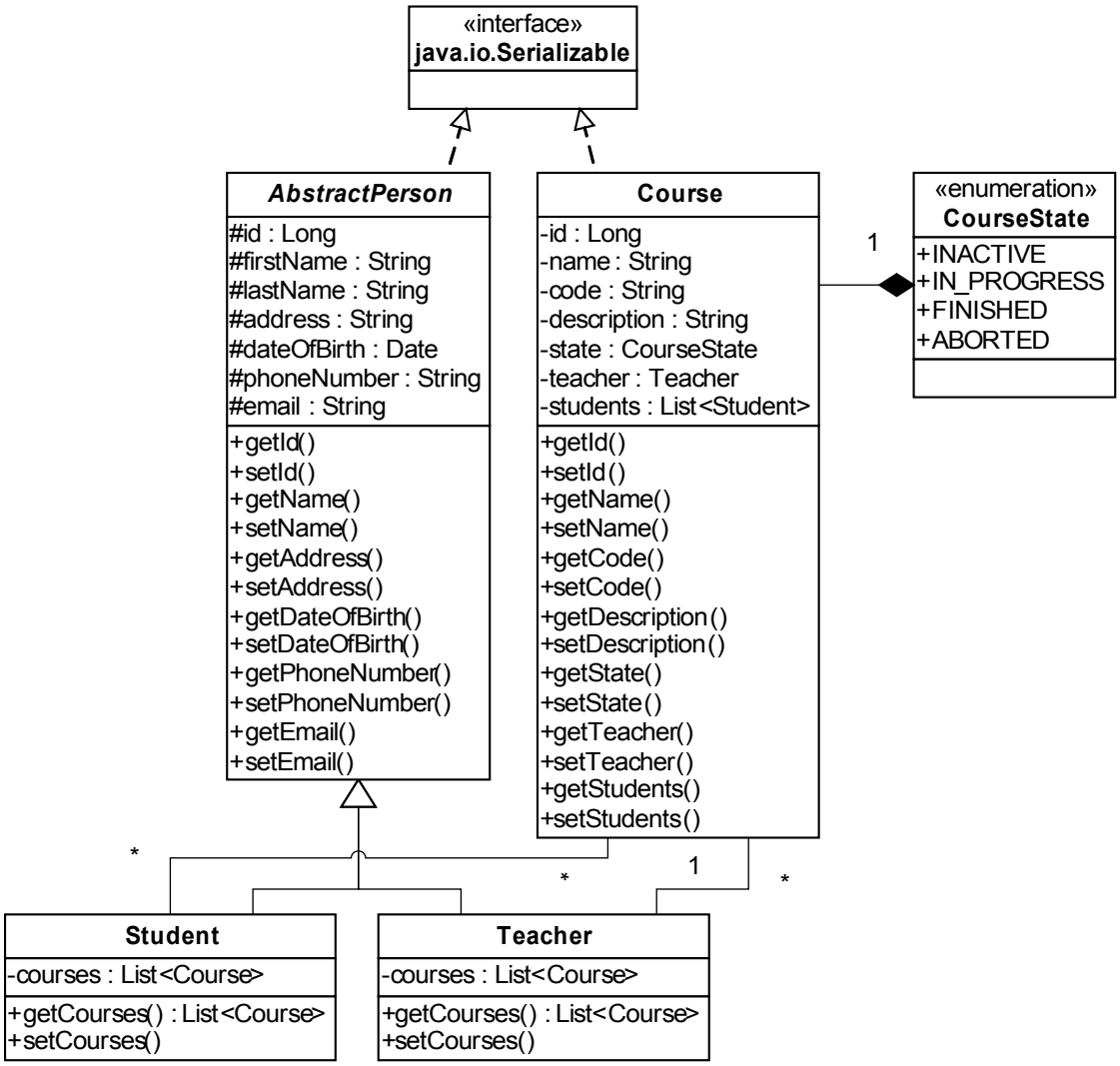
Tässä liitteessä on kuvattu esimerkkisovelluksen toiminnallisuus käyttötapauskaaviona (kuva 35). Jokainen hallinnoinnin käyttötapaus sisältää lisäyksen, muokkauksen ja poiston. Selaus-käyttötapaukset tarkoittavat olemassa olevan tiedon selausta ja suodatusta (haku).



Kuva 35: Käyttötapauskaavio esimerkkisovelluksen toiminnallisuudesta.

LIITE 2 : Esimerkkisovelluksen sovellusalueen malli

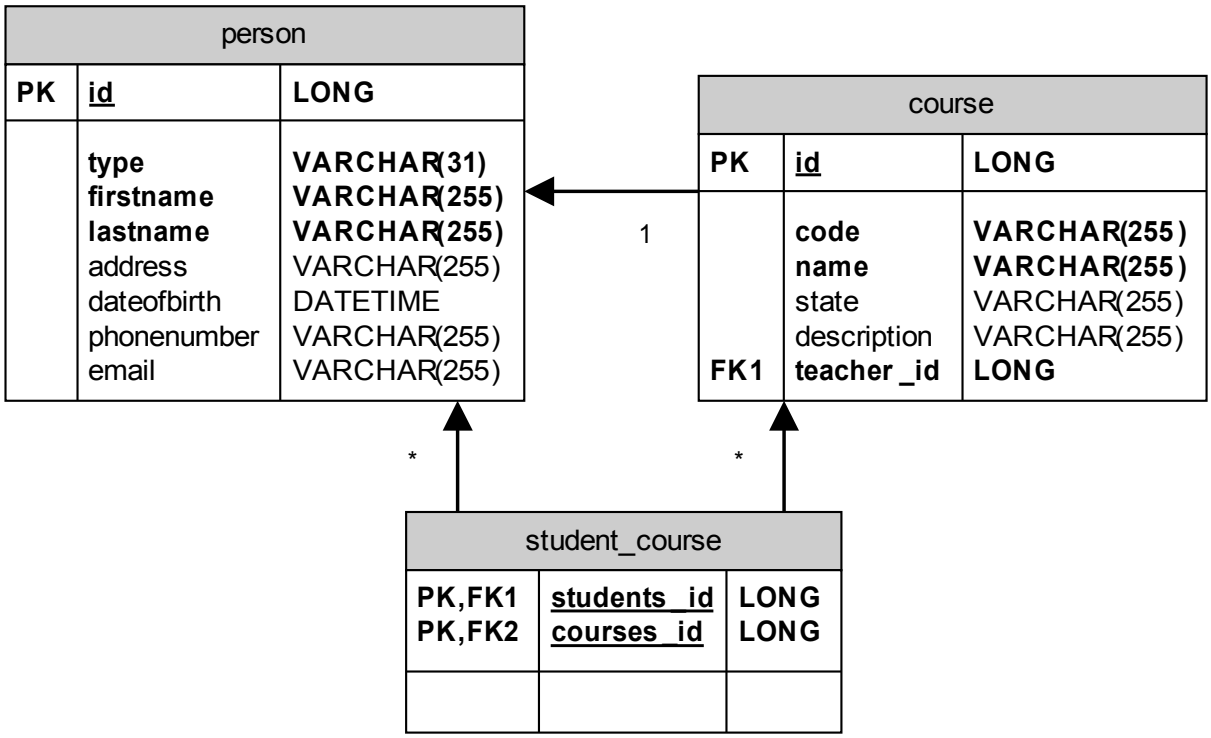
Tässä liitteessä on kuvattu esimerkkisovelluksen käyttämän sovellusalueen mallin sisältämät luokat sekä niiden väliset suhteet (kuva 36). Luokat koostuvat kurssista (`Course`), kurssin tilasta (`CourseState`) sekä abstraktista kantaluokasta (`AbstractPerson`), jonka toteutuksen perivät oppilas (`Student`) ja opettaja (`Teacher`) luokat. Lisäksi luokat kurssi, opettaja ja oppilas toteuttavat `java.io.Serializable` -rajapinnan.



Kuva 36: Esimerkkisovelluksen sovellusalueen malli.

LIITE 3 : Esimerkkisovelluksen tietokantakuvaus

Tämä liite sisältää esimerkkisovelluksen käyttämän relaatiotietietokannan kuvauksen (kuva 37). Relaatiot koostuvat person- ja course -tauluista, sekä student_course -välitaulusta. PK tarkoittaa primääriavainta, FK vierasavainta, lihavoitu teksti kuvaa kentän pakollisuutta. Myös kardinaliteetit on ilmennetty. Person taulussa type -sarake toimii diskriminaattorina, jota käytetään tallennetun rivin tunnistuksessa.



Kuva 37: Esimerkkisovelluksen käyttämä tietokantakuvaus.

LIITE 4 : Sovellusalueen mallin Java-ohjelmakoodi

Tämä liite sisältää esimerkkisovelluksen käyttämän sovellusalueen mallin Java-ohjelmakoodi, joka koostuu liitteessä 2 kuvatuista luokista kurssi (Course), kurssin tila (CourseState), oppilas (Student) ja opettaja (Teacher).

```
package fi.joensuu.cs.th.model;

import java.io.Serializable;
import java.util.Date;
import javax.persistence.*;

@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(
    name="type",
    discriminatorType=DiscriminatorType.STRING
)
@Table(name="person")
public abstract class AbstractPerson implements Serializable {
      private static final long serialVersionUID = 1L;

      protected Long id;
      protected String firstName;
      protected String lastName;
      protected String address;
      protected Date dateOfBirth;
      protected String phoneNumber;
      protected String email;

      protected AbstractPerson() {
      }

      @Id @GeneratedValue(strategy=GenerationType.AUTO)
      public Long getId() {
            return id;
      }

      public void setId(Long id) {
            this.id = id;
      }

      @Column(nullable=false)
      public String getFirstName() {
            return firstName;
      }

      public void setFirstName(String firstName) {
            this.firstName = firstName;
      }

      @Column(nullable=false)
      public String getLastName() {
            return lastName;
      }
```

```
	public void setLastName(String lastName) {
		this.lastName = lastName;
	}

	@Basic
	public String getAddress() {
		return address;
	}

	public void setAddress(String address) {
		this.address = address;
	}

	@Temporal(TemporalType.TIMESTAMP)
	public Date getDateOfBirth() {
		return dateOfBirth;
	}

	public void setDateOfBirth(Date dateOfBirth) {
		this.dateOfBirth = dateOfBirth;
	}

	@Basic
	public String getPhoneNumber() {
		return phoneNumber;
	}

	public void setPhoneNumber(String phoneNumber) {
		this.phoneNumber = phoneNumber;
	}

	@Basic
	public String getEmail() {
		return email;
	}

	public void setEmail(String email) {
		this.email = email;
	}

	/**
	 * @return <p>firstName lastName</p>
	 */
	@Transient
	public String getFullName() {
		return firstName+" "+lastName;
	}

	/**
	 * @return <p>lastName, firstName</p>
	 */
	@Transient
	public String getFullNameForList() {
		return lastName+", "+firstName;
	}
}
```

```
package fi.joensuu.cs.th.model;

import java.io.Serializable;
import java.util.List;
import javax.persistence.*;

@Entity
@Table(name="course")
public class Course implements Serializable {
	private static final long serialVersionUID = 1L;

	private Long id;
	private String name;
	private String code;
	private String description;
	private CourseState state = CourseState.INACTIVE;
	private Teacher teacher;
	private List<Student> students;

	public Course() {
	}

	@Id @GeneratedValue(strategy=GenerationType.AUTO)
	public Long getId() {
		return id;
	}

	public void setId(Long id) {
		this.id = id;
	}

	@Column(nullable=false)
	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	@Column(nullable=false, unique=true)
	public String getCode() {
		return code;
	}

	public void setCode(String code) {
		this.code = code;
	}

	@Basic
	public String getDescription() {
		return description;
	}

	public void setDescription(String description) {
		this.description = description;
	}

	@ManyToOne(cascade={CascadeType.PERSIST, CascadeType.MERGE})
	public Teacher getTeacher() {
		return teacher;
```

```
	}

	public void setTeacher(Teacher teacher) {
		this.teacher = teacher;
	}

	@Enumerated(EnumType.STRING)
	public CourseState getState() {
		return state;
	}

	public void setState(CourseState state) {
		this.state = state;
	}

	@ManyToMany(mappedBy="courses", cascade={CascadeType.PERSIST,
			CascadeType.MERGE})
	public List<Student> getStudents() {
		return students;
	}

	public void setStudents(List<Student> students) {
		this.students = students;
	}
}
```

```
package fi.joensuu.cs.th.model;


public enum CourseState {
	INACTIVE,
	IN_PROGRESS,
	FINISHED,
	ABORTED
	;
}
```

```
package fi.joensuu.cs.th.model;

import java.io.Serializable;
import java.util.List;
import javax.persistence.*;

@Entity
@DiscriminatorValue("S")
public class Student extends AbstractPerson implements Serializable {
	private static final long serialVersionUID = 1L;

	private List<Course> courses;

	public Student() {
	}

	@ManyToMany(cascade={CascadeType.PERSIST, CascadeType.MERGE})
	@JoinTable(name="student_course")
	@OrderBy("code")
	public List<Course> getCourses() {
		return courses;
	}
```

```
	public void setCourses(List<Course> courses) {
		this.courses = courses;
	}
}


package fi.joensuu.cs.th.model;

import java.io.Serializable;
import java.util.List;
import javax.persistence.*;

@Entity
@DiscriminatorValue("T")
public class Teacher extends AbstractPerson implements Serializable {
	private static final long serialVersionUID = 1L;

	private List<Course> courses;

	public Teacher() {
	}

	@OneToMany(mappedBy="teacher", cascade={CascadeType.PERSIST,
			CascadeType.MERGE})
	@OrderBy("code")
	public List<Course> getCourses() {
		return courses;
	}

	public void setCourses(List<Course> courses) {
		this.courses = courses;
	}
}
```