LAPPEENRANNAN TEKNILLINEN YLIOPISTO
Teknistaloudellinen tiedekunta
Tietotekniikan koulutusohjelma

Diplomityö

Jussi Huotari

YRITYKSEN TUOTERAKENTEEN MÄÄRITTÄMINEN OLIOPARADIGMALLA

Työn tarkastaja: Professori Jari Porras

Työn ohjaaja: Professori Jari Porras

TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Teknistaloudellinen tiedekunta
Tietotekniikan koulutusohjelma

Jussi Huotari

Yrityksen tuoterakenteen määrittäminen olioparadigmalla
Diplomityö

2015

122 sivua, 19 kuvaa, 19 liitettä

Työn tarkastaja: Professori Jari Porras

Hakusanat: olioparadigma, ERP, suunnittelumalli, tuoterakenne.

Tässä työssä tutkitaan yrityksen tuoterakenteen toteuttamista olioparadigmalla osana vanhenevan tuotannonohjausjärjestelmän uudistamista. Työssä on tunnistettu olioparadigman keskeiset rakenteet sekä tarkasteltu suunnittelumalleja, joita hyödyntämällä voidaan määrittää ja toteuttaa monimutkaisia sekä laajennettavia tuoterakenteita. Työn tavoitteena oli muodostaa tuoterakenteen runko, jota muokkaamalla on mahdollista toteuttaa erityyppisiä tuoterakenteita. Työ on rajattu siten, että todellista tuoterakennetta ei toteuteta. Tutkimuksen tuloksena havaittiin, että oliokeskeisyydellä voidaan toteuttaa tuoterakenteita, joita on mahdollista tulevaisuudessa muokata yrityksen tarpeisiin mukauttaen siten, että tuoterakenteen runkoon ei tarvitse tehdä suuria muutoksia. Olioperusteisella lähestymisellä voidaan mallintaa hyvin erityyppisiä ja ymmärrettäviä käsitteitä. Oliokeskeisellä lähestymistavalla voidaan toteuttaa myös muut vanhenevan tuotannonohjausjärjestelmän osa-alueet.

ABSTRACT

Lappeenranta University of Technology
Degree Program in Computer Science

Jussi Huotari

Determinating enterprice production structure by using object paradigm

Master's Thesis

2015

122 pages, 19 figures, 19 appendices

Examiner: Professor Jari Porras

Keywords: object-oriented paradigm, ERP, design patterns, product structure

The objective of this study is to research an implementation of company's product structure by a using object methodology as a part of the aging production control system renewal. The study finds out that by exploiting the main structures of object methodology and explored design templates, a complex and scalable product structures can be defined and implemented. The aim of study was to create a base of product structure which can be modified to implement different types of product structures. The study was restricted that the real product structure was not implemented. The results of the study present that by using the object oriented methodology, the complex product structures can be implemented and in the future these product structures can be modified in accordance with company needs without base product structure massive changes. By using the object oriented approach, not only very different kind of intelligible concepts can be modeled, but also the object-oriented approach can be implemented in other aging of the production control areas.

ALKUSANAT

Työ on toteutettu erään vanhentuvan ERP-järjestelmän uudistamisen suunnittelun yhteydessä, jossa on tarkasteltu tuoterakenteen määrittelyä ja toteutusmahdollisuuksia. Kiitän Eeraa ja Miroa kannustuksesta sekä erityisesti Miroa tarkentavista kysymyksistä.

SISÄLLYS

LIITTEET

SYMBOLI- JA LYHENNELUETTELO

ACID	Atomicity, Consistency preservation, Isolation, Durability
ERP	Enterprise Resource Planning
J2EE	Java 2 Platform Enterprise Edition
ORM	Object Relational Mapping
SOA	Service Oriented Architechture
SQL	Structured Query Language
UOW	Unit of Work
XML	Extensible Markup Language
WWW	World Wide Web

1 JOHDANTO

Olioperusteinen ohjelmistokehitys on keskeisessä asemassa suurien tietojärjestelmien toteutuksissa. Olioperustainen ohjelmistokehitys on systemaattista, ohjelmiston koko elinkaaren käsittävää prosessia, jonka perusteena on kuvata ohjelmisto joukkona vuorovaikutuksessa olevia olioita (Koskimies, 1997). Olioperustaisuus on eräs ohjelmointimalli eli paradigma, joka vastaa ihmisen tapaa hahmottaa ympäröivää maailmaa ja kykenee käsitetasolla hyvin yleiskäyttöiseen ohjelmistojen kuvaamiseen, mallintamiseen ja toteuttamiseen.

Tietojärjestelmien uudelleen käytettävyys on oliokeskeisyydessä tärkeässä osassa. Uudelleen käytettävyydellä tarkoitetaan, että toteutettava järjestelmä hyödyntää jo olemassa olevaa toteutusta jossakin muodossa, joka voi olla yleisiä luokkakirjastoja, luokkien periytymiseen liittyviä tekijöitä tai suunnittelumallien hyödyntämistä. Suunnittelumallilla tarkoitetaan tapaa suunnitella ohjelmiston osia siten, että se ratkaisee jonkin usein esiintyvän ongelman (Fowler, 2002). Useita eri suunnittelumalleja voidaan hyödyntää myös yhdessä, jolloin ne voivat kattaa alkuperäisestä ongelmasta suuren osan. Uudelleen käytettävyys mahdollistaa tietojärjestelmien kustannusten pienentämisen. Tämä koskee sekä järjestelmän toteutusta että sovelluksen koko elinkaarta, sillä jatkokehitys ja ylläpito helpottuvat.

Tietojärjestelmissä tiedon pysyvyydellä on merkittävä rooli (Agrawal & Gehani, 1990). Pysyvyydellä tarkoitetaan tietojärjestelmässä käytettävän tiedon tallennusta fyysiseen muistiin. Pysyvästi tallennettua tietoa on pystyttävä hakemaan käyttöön sekä tarvittaessa haettu tieto on pystyttävä muokkauksen jälkeen tallentamaan muokattuna takaisin fyysiseen muistiin. Alkuperäisen tiedon luonut sovellus on oman suorituksen aikana luonut tiedon, ja tieto on olemassa alkuperäisen sovelluksen suorittamisen jälkeen.

Tietoa on olemassa valtavat määrät, ja tyypillisesti se on käyttäjille arvokasta. Tiedon täytyy olla saatavissa eri sovellusten suoritusten yhteydessä, ja tiedon on säilyttävä useita vuosia. Tietoa käsitteleviin järjestelmiin ja sovelluksiin voidaan tehdä useita eri muutoksia, ja tiedon rakenteeseen voi tulla tarkennuksia sekä lisävaatimuksia. Yhtäaikaisia saman tiedon käyttäjiä voi olla useita. Tiedon oikeellisuudesta muutostilanteissa on pystyttävä

varmistumaan, että useampi käyttäjä ei pysty käsittelemään samaa tietoa niin, että käsittely johtaa virheelliseen lopputulokseen.

1.1 Tausta

Tämän työn taustalla oli tarve selvittää vanhojen toiminnanohjus- eli ERP-järjestelmien (Enterprise Resource Planning) nykyaikaistamista. Yrityksissä on vieläkin laajasti käytössä vanhoja niin kutsuttuja legacy-järjestelmiä, joiden toteutukset ja rakenteet ovat suhteellisen jäykkiä. Kyseisten järjestelmien nykyisen ylläpidon tarve sekä muutosten toteuttaminen ovat työläitä ja vaativat paljon taloudellisia resursseja.

Olemassa olevien legacy-järjestelmien toiminta on mahdollista kuvata ja mallintaa olioparadigman avulla. Olioiden taustalla olevien luokkien sekä niiden suhteiden ja vuorovaikutusten määrittäminen on keskeisessä roolissa oliomallinnuksessa. Lisäksi olioparadigma ei määritä, miten tai millä toteutusmenetelmillä malli toteutetaan, eli toteutuksessa voidaan käyttää valmiita sovelluspalvelimia tai hyödyntää valmiita suunnittelumalleja.

Olioiden käyttäytymiseen ja niiden hallintaan on kiinnitettävä huomiota. Valmiit sovelluspalvelimet huolehtivat olioiden elinkaaresta ja niiden tarvitsemista palveluista hyvin pitkälle ilman, ettei niiden sisäisestä toteutuksesta sovelluskehittäjän tarvitse syvällisesti tietää. Tilanteissa, joissa olioiden hallinta on toteutettava ilman sovelluspalvelimia, on hyödynnettävä suunnittelumalleja, joiden avulla olioiden hallinta on mahdollista toteuttaa tehokkaasti. Lisäksi sovelluspalvelimen toiminta pohjautuu suunnittelumalleihin, joten niiden tuntemuksesta on hyötyä sovelluspalvelimen hyödyntämisessä.

Legacy-järjestelmien nykyaikaistamisessa voidaan edetä vaiheittain. Olemassa olevan järjestelmän rinnalle voidaan kehittää olioperusteisia ratkaisuja, jolloin voidaan muodostaa olemassa olevista legacy-järjestelmän osista palveluita. Tämä mahdollistaa sen, että legacy-järjestelmä voi toimia samaan aikaan, kun uutta järjestelmää kehitetään. Tavoitteena on legacy-järjestelmän osien asteittainen siirto palveluina uuteen ympäristöön.

1.2 Tavoitteet ja rajaukset

Työn tarkoitus on, että työssä muodostetaan olioparadigmalla tuoterakenteen runko, jonka ydin voi toimia tuoterakenteen perustana. Työssä hyödynnetään olemassa olevia suunnittelumalleja, ja niiden avulla pyritään muodostamaan Domain-malli, jota tapauskohtaisesti täydentämällä tuoterakenne voidaan toteuttaa.

Tuoterakenne kuvaa olemassa olevan tuotteen rakenteellisella tasolla ja esittää tuotteen luonteenomaiset piirteet. Tuoterakenteen ominaisuudet voivat vaihdella paljon toimialakohtaisesti sekä tuoterakenteen käyttötarkoituksen mukaan (Pérez-Castillo et al., 2011). Tuoterakenteen käyttötarkoitus määrittelee ominaisuudet, jotka ovat hyvin eriävät esimerkiksi tuotannonohjauksen tai tuotteiden ostosovellusten tuotekatalogien näkökulmasta.

Tuoterakenne on yksinkertaisimmillaan pelkkä taulukko, jonka yksittäiset sarakkeet kuvaavat tuotteen ominaisuuksia. Tuoterakenteen monimutkaisuus kasvaa vaatimusten lisääntymisten seurauksena. Tuoterakenteen tietomallin eri käsitteiden väliset suhteet täytyy pystyä mallintamaan tarkasti, jotta tarkoitukseen sopiva tuoterakenne pystytään toteuttamaan.

Tuoterakenteen käyttökohteena voi olla teollisuusyrityksen valmistamien tuotteiden perustietojen hallinnan osana tai kaupanalan tuotteiden hallinnassa, jossa erityyppisten tuotteiden laajaa ja moninaista hallintaa, luokittelua sekä jäsentelyä joudutaan hyödyntämään. Tuoterakenteen vaatimukset täytyy analysoida tarkasti ennen tuoterakenteen määrittelyä.

Määritettävän tuoterakenteen perusominaisuudet:

- Tuoterakenteen on tuettava erityyppisiä tuotteita, joiden ominaisuudet ovat täysin toisistaan poikkeavat. Lisäksi uudentyyppisiä tuotteita pystyttävä tulevaisuudessa muodostamaan tuoterakenteeseen.

- Tuotteilla on oltava luokittelumenetelmät. Koska erityyppisiä tuotteita on mahdollista muodostaa tulevaisuudessa, luokitteluun on kiinnitettävä erityistä huomiota.
- Tuote voi muodostua useista eri komponenteista. Eri tyypin tuotteilla voi olla sisäisiä rakenteita.

Tutkimuskysymys:
Miten tuotannonohjausjärjestelmän tuoterakenteen runko voidaan määrittää olioparadigman avulla?

Alakysymys:
Miten määritetty tuoterakenne voidaan ottaa käyttöön yrityksen nykyiseen tuotannonohjausjärjestelmään?

Työssä ei voida yleisesti ottaa kantaa yksittäiseen legacy-järjestelmään, eikä työn tarkoitus ole yksityiskohtaisesti kuvata tuoterakennetta eikä tuotteiden tietosisältöä. Legacy-järjestelmät ja niiden vaatimat toiminnallisuudet ovat tapauskohtaisia, joten yleistä ratkaisua ei työssä voida rakentaa.

1.3 Työn rakenne

Tämä työ koostuu 10 luvusta, joista ensimmäisessä esitellään työn taustaa, tavoitteita ja rajauksia.

Luvussa 2 tarkastellaan olion käsitettä tietojärjestelmien kannalta. Luvussa perehdytään olion ominaisuuksiin sekä esitellään olion pysyvyyden kannalta tarvittavia menetelmiä, kuten tietokantoja sekä tietokannassa olevien olioiden käsittelyyn käytettäviä transaktioita.

Luvussa 3 perehdytään pysyvyyden käsitteeseen, pysyvyyden viitekehykseen, olioiden välisiin suhteisiin sekä olioiden tallentamiseen pysyvästi relaatiotietokantaan. Luvussa tarkastellaan myös ristiriitoja, joita esiintyy olioiden ja relaatiotietokantojen tietojen esitystapojen välillä.

Luvussa 4 käydään läpi kerrosarkkitehtuuria, jota yleisesti käytetään menetelmänä pilkkoa sekä jäsennellä suuria kokonaisuuksia loogisiin osiin, mikä helpottaa järjestelmän ylläpitoa sekä kehitystä.

Luvussa 5 käsitellään liiketoimintalogiikan toteuttamiseen liittyviä ongelmia ja niiden ratkaisumalleja. Luku käsittelee liiketoimintalogiikan organisointia eri menetelmillä sekä vertailee niiden eroja sekä hyötyjä.

Luvussa 6 tarkastellaan eri suunnittelumalleja sekä niiden hyötyjä ja haittoja. Suunnittelumallit ovat nykyaikaisten järjestelmien toteutusten osina, ja kappaleessa pohditaan niiden hyödyntämistä. Tavallisesti suunnittelumallit on toteutettu olemassa olevaan ympäristöön, jolloin niiden toteutukset ovat käyttäjältä piilotettu.

Luvussa 7 esitellään olioparadigmalla toteutettu tuoterakenteen malli sekä esitellään tuoterakenteen toteutuksen periaatteet ja rakenteen toteutus.

Luvussa 8 tarkastellaan tuoterakenteen hyödyntämistä käytännössä sekä tarkastellaan, miten tuoterakenne on osa ERP-järjestelmää laajemmin.

Luvussa 9 esitetään työn tulokset sekä vastataan luvussa 1 esitettyihin tutkimuskysymyksiin.

Luvussa 10 kootaan työn keskeiset tulokset, arvioidaan niiden merkitystä sekä tehdään tuloksista johtopäätökset.

2 OLIOPARADIGMA JA PERUSKÄSITTEET

Tuoterakenteen muodostaminen hyödyntämällä olioparadigmaa perustuu olioperusteisuuden pääkäsitteisiin. Tässä luvussa tarkastellaan olioiden perusominaisuuksia, joiden hyödyntäminen on keskeisessä roolissa työn toteutuksessa. Lisäksi luvussa käsitellään olioparadigmaa, olion määritelmää, esitellään relaatiotietokanta sekä olioiden tallennuksessa tarvittavien transaktioiden ominaisuudet.

2.1 Olio

Tuoterakenne koostuu joukosta erityyppisiä olioita, joilla on tietyt ominaisuudet ja toiminnallisuudet. Toiminnallisuudet voidaan toteuttaa olioiden avulla siten, että tuotteiden erityispiirteet pystytään muodostamaan.

Abstraktissa mielessä olio voidaan määritellä ympäristöstään erottuvana kokonaisuutena. Oliolla on oma identiteetti ja sisäinen rakenne sekä suhteet tiettyyn ympäristöön. Teknisessä mielessä olio on ohjelman strukturoinnin perusyksikkö, joka voi olla puhtaasti tietoa sisältävä (esim. tietue), ja se kykenee sisältämään toiminnallisuutta (esim. aliohjelmia). Olion tiedon ja toiminnallisuuden yhdistämistä yhdeksi kokonaisuudeksi kutsutaan kapseloinniksi (encapsulation). (Koskimies, 1997.)

Olioilla on seuraavat keskeiset ominaisuudet: operaatiot, attribuutit, viite sekä suojaus. Oliolle on määritetty tietyt operaatiot eli toiminnallisuudet, jotka se pystyy suorittamaan. Olion operaatioilla on nimi, operaatiot voivat sisältää parametreja ja operaatiot voivat tarvittaessa palauttaa arvon. Olio tallentaa tietoa attribuutteihin eli nimettyihin kenttiin, jotka voivat olla joko tietotyyppejä tai olioita. Attribuutit sisältävät operaation tilan, ja operaation suorituksella voidaan muuta olion tilaa. Jokainen olio tunnistetaan olion identifioivalla tunnisteella eli viitteellä. Kahdella eri oliolla on eri viite, vaikka niiden attribuutit eli tilat olisivat samoja. Olio on myös suojattu, eli olion käyttö on rajattua. Olion attribuuteilla sekä operaatioilla on suojaustasot, joiden avulla olion käyttöä voidaan rajata. Olio pystyy suojauksen ansiosta esittämään toiminnot, jotka se kykenee suorittamaan,

mutta samalla toteutus voidaan suojauksen avulla piilottaa siten, että ulkopuoliset eivät näe varsinaista toteutusta. (Koskimies, 1997.)

Olioiden attribuuttina voi olla viite toiseen olioon, mikä merkitsee sitä, että olioiden välillä on jokin suhde. Olioiden väliset suhteet voidaan määrittää kuten tietokantamallissa (database schema). Olioiden väliset suhteet voivat olla esimerkiksi yhden suhde yhteen (1-1), yhden suhde moneen (1-N) ja monen suhde moneen (M-N) (O'Neil, 2008).

2.2 Luokka

Luokkien määrittelyt sekä niiden välisten keskinäisten suhteiden mallintaminen on keskeisessä asemassa tuoterakenteen toteutuksessa. Luokkien määrittelyt täytyy toteuttaa siten, että ne vastaavat mallinnettavaa ympäristöä eli toteutettavaa tuoterakennetta.

Olioiden piirteet määritetään luokassa (class), jokaisella oliolla on yksikäsitteinen luokka, jonka perusteella olio voidaan luoda. Oliota, joka on luotu luokasta, kutsutaan luokan ilmentymäksi (instance). Luokka on siis olion malli (template). Luokka määrittelee ilmentymän attribuutit sekä operaatiot. Luokka siis määrittelee tieto- ja toiminto-osista muodostuvan kokonaisuuden, jolla on oma rajapinta. Luokka on staattinen käännöksen aikainen käsite, ja olio on dynaaminen ajonaikana luotava rakenne. (Koskimies, 1997.)

Jokaisella luokasta muodostetulla oliolla on oma tila, joka noudattaa luokan määrittelyä (schema), sekä tilan siirtymät, jotka toteuttavat luokan operaatioita. Luokilla on myös tyyppi. Samasta luokasta muodostetuilla olioilla on sama tyyppi, lisäksi tyypitys mahdollistaa eri oliotyyppien välisten viittausten muodostamisen. (Duke et al., 1995.)

2.3 Periytyminen

Periytymisellä (inheritance) on kaksi tavoitetta. Ensimmäisenä periytyminen parantaa koodin uudelleenkäytettävyyttä, jolloin aliluokkien ei tarvitse toistaa perittyjä piirteitä, vaan niitä voidaan suoraan hyödyntää perimisen seurauksena. Erityisen hyödyllistä

uudelleenkäyttö on silloin, kun useat eri sovellukset voivat hyödyntää ja periä yleiskäyttöisiä kirjastoluokkia. Toiseksi periytyminen tukee käsitteelliseen mallintamiseen pohjautuvaa ohjelmistokehitystä, jolloin periytyminen vastaa läheisesti erikoistamisen ja yleistyksen käsitteitä. Eli aliluokka voidaan tulkita erikoistamisena ja yläluokka yleistyksenä, esimerkiksi luokan Henkilö on luokan Mies yleistys ja luokka Mies on luokan Henkilö erikoisuus. (Koskimies, 1997.)

Periytymisessä on kaksi pääasiallista menetelmää: yksittäisperiytyminen (single inheritance) sekä moniperiytyminen (multiple inheritance). Jos luokalla on ainoastaan yksi yliluokka, on kyseessä yksittäisperiytyminen. Jos luovutaan rajoituksesta, että luokalla on ainoastaan yksi yliluokka ja perivällä luokalla on useita yliluokkia, kyseessä on tällöin moniperiytyminen.

Periytymisellä tarkoitetaan luokan (A) piirteiden siirtämistä toiselle luokalle (B). Tällöin luokka (A) on luokan (B) yliluokka ja luokka (B) on luokan (A) aliluokka. Perintä mahdollistaa uusien luokkien määrittelyn jo olemassa olevien määritysten pohjalta. (Duke et al., 1995.)

Periytyminen mahdollistaa abstraktien luokkien käyttämisen. Abstraktilla luokalla tarkoitetaan luokkaa, josta ei suoraan voida muodostaa oliota, vaan abstrakti luokka on perittävä ja aliluokissa on toteutettava abstraktit toiminnot. Periyttäminen mahdollistaa abstraktit rajapinnat, joiden kautta voidaan käsitellä tuntemattomien tahojen rakentamia olioita ennalta määritetyllä tavalla. (Duke et al., 1995.)

Yksittäisperiytymisessä yliluokkasuhteet muodostavat puumaisen rakenteen, jossa luokat ovat solmuja siten, että solmun vanhempi on luokan yliluokka. Perinnässä vanhempi-luokka periyttää piirteensä lapsiluokille, joilla on omia piirteitä, ja tällöin piirteiden määrä lisääntyy kuljettaessa puussa alaspäin.

Moniperiytymisessä luokalla voi olla useita eri yliluokkia. Yliluokkahierarkia ei enää ole puu vaan syklitön verkko. Syklittömyys on seuraus siitä, että luokka ei voi periä itseään suoraan tai epäsuorasti. Moniperintää on järkevää hyödyntää tilanteissa, jossa tarvitaan

useiden eri yliluokkien ominaisuuksia. Moniperintä ei ole mahdollista kuitenkaan jokaisessa oliokielessä. (Koskimies, 1997.)

Moniperiytyminen on ongelmallista useista eri syistä: periytymissuhteiden monimutkaisuus lisääntyy, eri nimikonflikteja voi ilmaantua ja sama luokka voi periytyä eri reittejä useampaan kertaan. Tätä kutsutaan toistuvaksi periytymiseksi. Moniperiytyminen voidaan korvata usein yksittäisellä periytymisellä, ja se on helpommin toteutettavissa silloin, kun kaikki luokat ovat yhtä aikaa suunniteltavina. Olemassa olevien luokkien kohdalla korvaaminen yksittäisperimisellä on vaikeampaa.

Tuoterakenteen määrittelyssä on hyödynnetty yksittäisperiytymistä. Abstraktien luokkien avulla on pystytty toteuttamaan eri luokkien yhteiset ominaisuudet yläluokkiin, jotka perimällä aliluokka pystyy toteuttamaan yläluokan ominaisuudet.

2.4 Aikainen ja myöhäinen sitominen

Sidonta on tärkeässä roolissa olio-ohjelmoinnissa. Sidonta tarkoittaa prosessia, jossa operaation tai funktionkutsut yhdistetään todelliseen koodiin. Sidonta tehdään, kun sovellus käännetään ja kaikki koodissa kutsut funktiot on sidottava ennen kuin koodia voidaan suorittaa.

Operaatioiden tai funktioiden kutsut voidaan jakaa aikaiseen sidontaan (staattinen sidonta, early binding) ja myöhäiseen sidontaan (dynaaminen sidonta, late binding). Jos operaatiot sidotaan käännöksen aikana, kyseessä on aikainen sidota. Aikainen sidota helpottaa toteutusta, ja tällöin on tiedossa kutsuttava koodi jo ajettavaa koodia tuotettaessa. Olio-ohjelmoinnissa tämä mahdollisuus ei ole välttämättä aina voimassa sen vuoksi, että tiettyjen operaatioiden toteutustapa tiedetään vasta operaation sisältävän luokan jälkeläisluokissa. On myös mahdollista, että oliota voidaan käsitellä yliluokan edustajana, eli välttämättä ei tiedetä luokan perusluokkaa, jolloin ei myöskään tiedetä tämänkaltaisten operaatioiden toteutusta. Edellä kuvatussa tilanteessa ei voida käyttää aikaista sidontaa, joten on käytettävä myöhäistä sidontaa. Myöhäisessä sidonnassa ajoympäristö päättää vasta ohjelman suorituksen aikana, mitä operaatiota tai funktiota on kutsuttava.

Operaatioita, joihin myöhäistä sidontaa voidaan soveltaa, kutsutaan virtuaalisiksi. (Koskimies, 1997.)

Myöhäistä sidontaa hyödynnetään tuoterakenteen toteutuksessa laajalti. Yleisesti tuotteista palautetaan viittaus abstraktiin kantaluokkaan, jolloin myöhäisen sitomisen avulla pystytään kutsumaan oikean tuotteen operaatioita. Tästä on paljon hyötyä, koska työssä hakutoiminnot palauttavat tavallisesti abstraktin luokan viittauksen.

2.5 Abstrakti luokka

Luokka, jolla on vähintään yksi virtuaalioperaatio, kutsutaan abstraktiksi luokaksi (Smed et. al., 2007). Virtuaalioperaatio voi olla joko toteutettu eli rungollinen tai ilman runkoa. Virtuaalioperaatio, jolla on runko, toimii oletusrunkona, jos aliluokissa ei runkoa toteuteta. Kyseisiä virtuaalioperaatioita kutsutaan kiinnitetyiksi. Ilman runkoa toteutetuissa tilanteissa virtuaalioperaatio pakottaa jonkin aliluokan toteuttamaan operaation rungon. Jokaisella toimivalla sovelluksella on oltava määritettyjen operaatioiden toteutus, ja tällöin virtuaalioperaatioita kutsutaan avoimiksi virtuaalioperaatioiksi. Luokasta, jolla on avoimia virtuaalioperaatioita, ei voida luoda ilmentymiä eli olioita. (Koskimies, 1997.) Luokkaa, jonka kaikki operaatiot on toteutettu, kutsutaan konkreettiseksi (Smed et. al., 2007).

Tuoterakenteen määrittelyssä on toteutettu kolme abstraktia luokkaa: AbstractProduct, AbstractComponent ja AbstractClassification. Kyseisillä luokilla on avoimia virtuaalioperaatioita, joiden toteutukset löytyvät alaluokista, jotka perivät abstraktit luokat. Tulevaisuudessa abstraktien luokkien avulla voidaan tuoterakenteeseen lisätä uusia erityyppisiä tuotteita.

2.6 Olioparadigma

Olioperusteisen ohjelmistokehityksen ajatus on kiteytettynä, että toteutettava sovellusohjelmisto mallintaa kyseiseen sovellukseen liittyvän ympäristön konkreettisia tai abstrakteja tapahtumia. Rakenne edustaa ja kuvaa siten ajattelumallia, että sovellusohjelmisto jaetaan komponentteihin, jotka edustavat sovelluksen maailman olioita. Nämä oliot ovat yksilöitä, joilla on tiettyjä ominaisuuksia ja tietty käyttäytyminen. Oliot ovat vuorovaikutuksessa keskenään ja hyödyntävät toistensa tarjoamia palveluita.

Olioperustaisuudesta (object-oriented) on muodostunut yksi useasti toistettu avainsana niin teollisessa ohjelmistotuotannossa kuin ohjelmistotekniikan tutkimuksessa. Olioperusteisella ohjelmistokehityksellä tarkoitetaan systemaattista, koko ohjelmiston elinkaaren käsittävää prosessia, joka perustuu kaikissa ohjelmiston eri vaiheissa ohjelmiston kuvaamiseen joukkona keskeisessä vuorovaikutuksessa olevia olioita. (Koskimies, 1997.)

Olioperustaisuus on eräs ohjelmointiparadigma eli malli kuvata erilaisten järjestelmien toimintaa ja rakennetta. Muita yleisiä ohjelmointiparadigmoja ovat proseduraalinen ohjelmointi, funktionaalinen ohjelmointi, logiikkaohjelmointi sekä rajoiteohjelmointi. Olioperusteisuus sopii käytännöllisesti kaikkiin sovelluksiin, eikä sen käytöstä seuraa oleellisia tehokkuushäviöitä. Lisäksi olioperustaisuus vastaa hyvin ihmisen tapaa hahmottaa maailmaa. Näiden ominaisuuksien vuoksi tuoterakenteen toteuttaminen olioperusteisesti on järkevää.

Perinteinen ylhäältä alas (top-down) suuntautuva ohjelmistosuunnittelu on toimintasuuntautunutta, joka lähtee liikkeelle korkeimman tason toiminnoista. Niitä tarkennetaan asteittain, kunnes lopulta päästään ohjelmointikielen tasolle. Toiminnallisesti suuntautunutta ohjelmistokehitystä kritisoidaan useista syistä, koska top-down-menetelmällä toteutetut ohjelmistokomponentit ovat riippuvaisia ylempien tasojen ratkaisuista. Pienikin muutos ylemmissä tasoissa voi aiheuttaa suuria muutoksia koko toteutettuun ohjelmistoon. Tämän tyyppiset muutokset ovat varsin todennäköisiä ja vaikuttavat ohjelmiston ylläpitoon sekä toteutettujen ohjelmistokomponenttien huonoon

uudelleen käytettävyyteen. Uudelleenkäytettävyydellä tarkoitetaan, että ohjelmistokomponentit ovat yleiskäyttöisiä eivätkä ole riippuvaisia yksittäisen ohjelmiston vaatimuksista. Toimintasuuntautunut ohjelmointi on myös usein sopimatonta siitä syystä, että toteutetulla sovelluksella ei ole ylintä tasoa, vaan se on luonnehdittavissa kokoelmaksi erilaisia palveluita. (Koskimies, 1997.)

Tarve olioperustaiselle ohjelmistokehitykselle liittyy ohjelmiston kustannusten kasvuun, ylläpitoon sekä käyttäjien uusiin vaatimuksiin, jotka edellyttävät palaamista sovelluksen alkuvaiheeseen. Uusien vaatimusten toteuttaminen voi johtaa koko ohjelmiston rakenteen muutoksiin. Näiden minimoimiseen tarvitaan tekniikoita, joilla ohjelmiston yleisrakenne saadaan toteutettua mahdollisimman riippumattomaksi järjestelmän toimintaan kohdistuvista muutosvaatimuksista. Tällöin mahdolliset muutokset ja korjaukset olisivat rajattuja ja helpommin toteutettavia. (Koskimies, 1997.)

Olio-orientoitunut menetelmä määrittelee järjestelmän joukoksi olioita, jotka ovat keskinäisessä vuorovaikutuksessa toistensa kanssa ja olioilla on määritetyt rakenteet ja toiminnallisuudet (Duke et al., 1995).

2.7 Tietokanta

Tietokantaa hyödynnetään tietovarastona tuoterakenteen pysyvään tallentamiseen. Tietokanta tarkoittaa kokoelmaa yhteenkuuluvaa ja johonkin tiettyyn käyttötarkoitukseen liittyvää oleellista tietoa (Luoma. 2007). Digitaalisilla tietokannoilla eli tietokokoelmilla tarkoitetaan tietokoneiden massamuistiin tallennettavia tietokokonaisuuksia. Looginen tietokanta (logical database) on looginen tietokantamalli, joka on usein relaatiomallin tai oliomallin mukainen kokoelma tietoalkioita. Useat loogiset tietokannat ovat relaatiotietokantoja, jotka ovat kokoelmia relaatioista (relation) eli tauluja (table), jotka ovat monikoiden (tuble) eli rivien (row) monijoukkoja. (Silberschatz et al., 2010.)

Tietokannan toiminnalle on asetettu seuraavat vaatimukset: itsekuvaileva, useiden yhtäaikaisten käyttäjien tukeminen, näkymien luontikyky sekä yksityisenä sovelluksena toimiminen. Tietokannan on oltava itse kuvaileva, eli tietokantaan tallennetun tiedon on

sisällettävä myös tieto siitä, mikä kunkin tietoalkion merkitys on. Tietokannan tietoa kutsutaan dataksi ja tiedon rakennetta kuvaavaa tietoa metadataksi. Tietokannan on myös tuettava useita yhtäaikaisia käyttäjiä sekä tarjottava useita erilaisia näkymiä tietoon. Useiden yhtäaikaisten käyttäjien palveleminen on erityisen tärkeää ja aiheuttaa haasteita yhtäaikaisten operaatioiden suorittamisessa. Näkymien tarjoaminen on tärkeää, koska kaikille tietokannan käyttäjille ei ole sallittua esittää kaikkia tietokannan tietoja. Näkymillä voidaan rajoittaa eri käyttäjille esitettäviä tietoja. Lisäksi tietokannan ja sitä käyttävien ohjelmien on oltava erillisiä. Käytännössä tämä tarkoittaa, että tietokantaa on käytettävä tietyn rajapinnan kautta siten, että käyttäjiltä ja tietokantaa hyödyntäviltä sovelluksilta piilotetaan esimerkiksi tietojen tallennustapa. Tietokanta sisältää ainoastaan tietoa eli dataa sekä tietoa kuvailevaa metadataa eikä ota kantaa siihen, miten ja millaisissa ohjelmissa tietoa käytetään. (Luoma, 2007.)

Relaatiotietokannan hallintajärjestelmät ovat transaktiopalvelimia (transaction server). Transaktiopalvelin tarjoaa liittymän, jonka avulla tietokantaa hyödyntävät asiakassovellukset voivat lähettää palvelimelle tietokantaoperaatioiden suorituspyyntöjä. Suorituspyynnöt ovat tavallisesti tietoalkioiden lisäyksiä, muokkauksia, poistoja sekä kyselyitä. Palvelin vastaa asiakkaan pyyntöön palauttamalla operaation tulokset. Pyynnöt määritetään tavallisesti SQL-kyselykielen (Structured Query Language) avulla. (Silberschatz et al., 2010.)

2.8 Transaktiot

Transaktio (transaction) eli tietokantatapahtuma on tietokantaan kohdistuva loogisten tietokantaoperaatioiden sarja, jonka vaikutusten halutaan muodostavan yhden atomisen eli jakamattoman kokonaisuuden. Transaktioiden käytössä asetetaan transaktioiden rajat, eli transaktio alkaa aloituskirjauksella (begin) ja päättyy sitoutumispyyntöön (commit) tai keskeytys- ja peruutuspyyntöön (rollback). (Silberschatz et al., 2010.)

Tietokannat pyrkivät toteuttamaan transaktiomallin, joka kykenee toteuttamaan tiedon oikeellisuuden vaatimat säilytysominaisuudet. Tämän takaamiseen ja samanaikaisuuden ja

toipumisen edellytyksenä tietokannalta vaaditaan seuraavia ACID-ominaisuuksia (Atomicity, Consistency preservation, Isolation, Durability). (Luoma, 2008.)

Transaktioilla vaadittava jakamattomuuden (atomicity) ominaisuus tarkoittaa, että kaikki transaktion aikaansaamat muutokset tietokannan tilaan toteutuvat tai yksikään muutos ei jää voimaan loogiseen tietokantaan. Eli transaktio, joka on suorittanut loppuun commit-operaation, on sitoutunut (committed). Sitoutuneelta transaktiolta vaaditaan jakamattomuuden lisäksi pysyvyyttä (durability), mikä merkitsee, että sitoutuneen transaktion aikaansaamien muutosten on toteuduttava ja jäätävä pysyvästi voimaan tietokantaan. Sitoutuneen transaktion tekemät päivitykset voidaan kumota suorittamalla sitä varten toteutetulla uudella transaktiolla. Ristiriidattomuus (consistency preservation) tarkoittaa, että tapahtuma säilyttää ristiriidattomuutensa, jos se muuttaa tietokannan ristiriidattomasta tilasta toiseen ristiriidattomaan tilaan. Erillisyydellä (isolation) tarkoitetaan, että tietokantatapahtuman tulee edetä ikään kuin sitä suoritettaisiin täysin eristyksistä toisista tietokantatapahtumista eli transaktiot eivät saa vaikuttaa toisiinsa ja keskeneräinen transaktion suoritus ei saa näkyä toisille transaktioille. (Luoma, 2008.)

Tuoterakenteen transaktioiden käsittely on hoidettu sovelluspalvelimen ominaisuuksia hyödyntäen. Sovelluspalvelin huolehtii ja vastaa kaikista tuoterakenteen transaktioista. Transaktioiden hallintaa käytetään laajasti kaikissa tuoterakenteen käyttämissä operaatioissa. Transaktio voi sisältää ensin esimerkiksi tuoteolion luomisen ja seuraavaksi raaka-aineolion luomisen, joka on tarkoitus liittää tuoteoliolle. Jos raaka-aineolion luominen ei onnistu, tapahtuu rollback eikä ensimmäisenä luotu tuote-olio tallennu relaatiotietokantaan. Tämä tarkoittaa, että transaktio onnistuu kokonaisuudessaan tai epäonnistuu ja tietokanta palaa samaan tilanteeseen kuin ennen transaktion aloitusta.

3 PYSYVYYS JA OLIOIDEN KUVAUS RELAATIOTIETOKANNASSA

Olioiden sisältämän tiedon pysyvyys (persistence) on yksi tärkeimmistä järjestelmien toimintaominaisuuksista. Pysyvyydellä tarkoitetaan, että tieto säilyy järjestelmässä sen jälkeen, kun tiedon luonut taho ei ole enää olemassa, eli tiedon olemassaolo ei ole riippuvainen tiedon alkuperäisestä tekijästä. Tuoterakenne muodostetaan olioista, jotka tallennetaan pysyvästi relaatiotietokantaan. Tämän seurauksena pysyvyyden toteuttaminen ja hallinta ovat merkittäviä tekijöitä toteutuksen kannalta.

3.1 Pysyvyys

Olioiden pysyvyys voidaan jakaa kahteen eri pääryhmään: lyhytkestoiseen (volatile) tai pitkäkestoiseen eli pysyviin (persistence). Lyhytkestoiset oliot ovat olemassa ohjelman suorittamisen ajan järjestelmän fyysisessä muistissa, ja niiden olemassaolo päättyy tavallisesti ohjelman suorituksen loppumiseen. Pysyvät oliot täytyy säilyttää pysyvässä muistivarastossa, ja niiden olemassaolo jatkuu sovelluksen suorituksen päättymisen jälkeen. Pysyvänä muistivarastona käytetään tietokantoja, joka kykenevät tallentamaan olion tietosisällön sekä olion yksiköllisen tunnisteen (unique identifier). (Agrawal & Gehani, 1990.)

Olio syntyy ja on olemassa olion luonnin jälkeen, ja sen voi hävittää joko järjestelmä automaattisesti tai ohjelmoija eksplisiittisesti. Olio voidaan hävittää heti, kun oliota ei enää tarvita: esimerkiksi jokin operaation suorituksessa (väliaikainen olio), toisen olion hävittämisen yhteydessä (alistettu olio), ohjelmalohkosta poistumisen yhteydessä (paikallinen olio), roskankerääjän toimesta (viittaamattomat oliot) tai ohjelman suorituksen päättymisen yhteydessä (staattiset oliot). Jos olio jää olemaan, ohjelman suorituksen jälkeen oliota kutsutaan pysyväksi (persistent object). (Koskimies, 1997.)

Pysyvät oliot (persistence objects) eli olion attribuutit tallennetaan tietovarastoon, joka tavallisesti on relaatiotietokanta. Pysyviä olioita hyödyntävä sovellus tarvitsee mekanismit,

joilla voidaan hakea olion tiedot tietovarastosta sovelluksen varaamaan paikalliseen muistiin, jotta sovellus voi hyödyntää pysyvää tietoa. Tämän lisäksi sovellus tarvitsee menetelmät muutetun tiedon tallentamiseen, tiedon luomiseen ja poistamiseen. Menetelmää, joka kykenee toteuttamaan tarvittavan toiminnallisuuden, kutsutaan pysyvyyden viitekehykseksi (persitence framework), joka toteuttaa tarvittavan toiminnallisuuden pysyvien olioiden käsittelyyn. (Larman, 2004.)

3.2 Pysyvyyden viitekehys (persistence framework)

Pysyvyyden viitekehys eli persitence framework toimii abstraktiotasona sovellusten ja tietokannan välillä. Toteutuksessa on huomioitu uudelleenkäytettävyys, laajennettavuus, transaktioiden käsittely ja sovelluskehityksen yksinkertaistaminen. Persitence framework siirtää sekä hallinnoi sovelluksen lokaalissa muistissa olevien olioiden attribuuteissa olevan tiedon siirrosta tietokantaan. Olioiden attribuuttien tiedonsiirto on kaksisuuntainen eli olioista tietokantaan ja tietokannasta olioihin. (Larman, 2004.)

Persitence framework toteuttaa olioiden ja tietokannan välisen kuvauksen (O-R mapping), joka siis määrittelee sen, miten olioiden käsittely tietokannassa toteutetaan. Kuvaus sisältää tarvittavat SQL lausekkeet, jotka kykenevät muodostamaan tarvittavat haku-, lisäys-, muokkaus- ja poistotoiminnot. Tärkeä vaatimus on, että tarvittavat määritykset voidaan tehdä mahdollisimman helposti ilman suurta ohjelmointitarvetta. Kuvaus voidaan toteuttaa hyödyntämällä XML-kieltä (Extensible Markup Language) tai toteuttamalla persitence frameworkin vaatimat toiminnallisuudet. (Larman, 2004.)

Uudelleenkäytettävyyden vaatimuksena on, että useat eri sovellukset voivat hyödyntää samaa tarvittavaa toiminnallisuutta ilman, että samaa toiminallisuutta tarvitsee toteuttaa useaan kertaan. Tämä vähentää ylläpidon tarvetta sekä pienentää sovelluksen kustannuksia.

Transaktioiden hallinta on tärkeässä roolissa, sillä persistence framework pystyy huolehtimaan useiden eri käyttäjien toiminnoista siten, että käyttäjien tekemät toiminnot pysyvät eristettyinä toisistaan. Lisäksi transaktion suorituksen seurauksena tapahtuneet

olioiden luonnit, muokkaukset sekä poistamiset on pystyttävä hallitsemaan. (Larman, 2004.)

Työssä hyödynnettiin JBoss Application Server 7.1:tä, joka huolehtii pysyvyyden viitekehyksen toteutuksesta.

3.3 Olioiden kuvaus relaatiotietokannassa

Pysyvien olioiden attribuutit eli niiden sisältämä tieto voidaan käsitellä siten, että niiden tietosisältö ja suhteet saadaan kuvattua relaatiotietokannan tauluihin. Toisin sanoen olioiden tilat ja niiden väliset suhteet on tallennettu pysyvästi tietokantaan, jolloin niiden tilat ovat pysyviä eikä niiden pysyvyys riipu tietoa käyttävistä sovelluksista. Relaatiotietokannat eivät tunne olioiden välisiä perintäsuhteita eikä myöskään muita olioiden välisiä suhteita, joten tämä tarvittava riippuvuus täytyy toteuttaa ohjelmallisesti. (Ambler, 2000.)

3.3.1 Olion attribuuttien kuvaus tietokannan sarakkeisiin

Yksinkertaisin tapaus on, että oliolla ei ole viittauksia toisiin olioihin. Olion attribuutit, jotka edustavat perustietotyyppejä (integer, char, string, double, float, ym.), voidaan kuvata olion luokkaa vastaavaksi tietokantataulun sarakkeeksi. Luokan perustietotyypeille löytyy vastineet relaatiotietokannoista, joten kuvaus on mahdollista toteuttaa.

Core-luokka on määrittely sovelluksessa esimerkiksi seuraavasti:

```
public class Core
{
	private int		id;
	private String		name;
	private int		value;
}
```

Core-luokkaa edustaa relaatiotietokantaan luotu taulu SQL-lausekkeella:

```
create table Core(id int, name varchar[50], value int);
```

Nämä määrittelyt mahdollistavat Core-luokasta muodostettaville olioille pysyvyyden, jossa Core-luokasta muodostettavilla olioilla on vastaavat attribuutit määritetty relaatiotietokannassa olevassa Core-taulussa.

Core-luokan id-attribuutti on erityisessä asemassa, ja se muodostaa olioiden identiteetin eli jokaisella oliolla on oma yksilöivä tunnus.

3.3.2 Olioiden välisten assosiaatioiden kuvaus

Olioiden väliset keskinäiset suhteet ovat olennaisessa roolissa oliomallissa. Perintäsuhteiden lisäksi olioiden välillä on muitakin viittauksia, kuten yhdestä yhteen (1:1, one-to-one), yhdestä moneen (1:N, one-to-many) ja monesta moneen (N:M, many-to-many).

Suhteita on olemassa kahta eri tyyppiä: yksisuuntaisia (unidirectional) ja kaksisuuntaisia (bidirectional). Yksisuuntaisessa suhteessa vain toisella kuvaussuhteen oliolla on viittaus toiseen olioon. Kaksisuuntaisessa molemmilla olioilla on viittaukset toisiinsa. (Jendrock et al., 2007.)

Olioiden välisistä suhteista voi muodostua runsaslukuinen joukko, joka sisältää suuren määrän pieniä luokkia. Kuvatun kaltainen rakenne muodostaisi suuren joukon tietokantatauluja. Yleisenä sääntönä on, että jokaista luokkaa vastaa tietokannassa oleva taulu. On olemassa tilanteita, jolloin on järkevää upottaa (embedded value) useamman luokan attribuutit yhteen tietokantatauluun kuin luoda jokaisesta pienestä oliosta oma tietokantataulu. (Fowler, 2002.)

Olioiden välillä olevien monesta moneen eli N:M-suhteiden kuvaamisessa voidaan käyttää assosiaatiotaulua (association table), jossa assosiaatiotaulun jokainen rivi edustaa yhtä olioiden välistä suhdetta. Assosiaatiotaulu lisätään kahden liitettävän taulun väliin.

Olioilla on usein viittauksia toisiin olioihin. Olioiden välisten viittauksien toteutus on toteutettu tavallisesti siten, että viittaus on tallennettu olion attribuuttiin. Attribuutin arvona

on olion yksilöivä tunnus (Identity Field). Menetelmää kutsutaan myös Foreign Key Mapping -menetelmäksi. Jokaisella oliolla on attribuuttinaan oma yksilöivä tunniste, joka toimii pääavaimena (primary key) ja olion yksilöivänä tunnisteena. Tilanteissa, joissa olio viittaa toiseen olioon, viittauksessa käytetään viitattavan olion yksilöivää tunnistetta, eli olio, joka viittaa toiseen olioon, tallentaa omaan attribuuttinsa viitattavan olion pääavaimen. (Fowler, 2002.)

Tuoterakenteen eri olioiden välisten suhteiden määrittelyyn on kiinnitetty paljon huomiota, ja olioiden välisten suhteiden toteutuksella on suuri vaikutus uudelleen käytettävyyteen sekä tuoterakenteen laajennettavuuteen. Esimerkkinä tuoterakenteen eri osioiden välisistä viittauksista voidaan mainita yhden suhde yhteen -viittaus tuotteen ja raaka-aineen välillä. Yhden suhde moneen -viittaus esiintyy tuotteella, joka voi koostua useista eri komponenteista, sekä monen suhde moneen esiintyy komponenteilla, jotka voivat kuulua useisiin eri tuotteisiin.

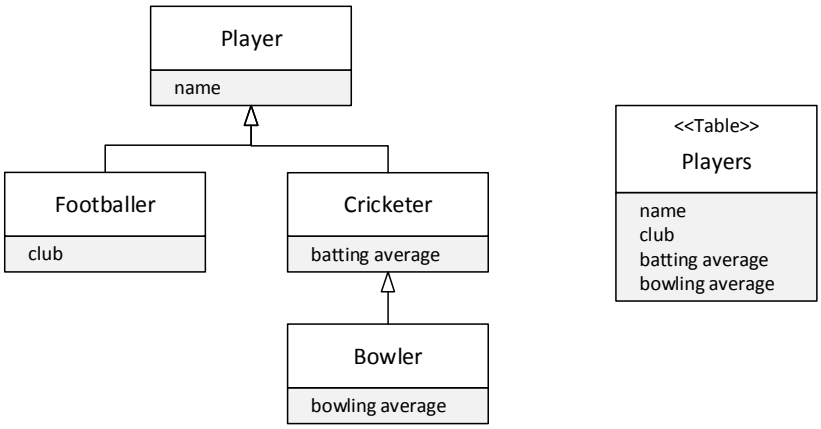
3.4 Olioiden perintäsuhteet

Liiketoimintakerrosten toteuttaville olioille on tavallista, että ne muodostavat perintäsuhteita. Pysyvien olioiden tapauksessa relaatiotietokannat eivät tue perintää, joten periytyminen täytyy toteuttaa relaatiotietokannassa toisella tavalla. Fowler esittää kolme mahdollista toteutusvaihtoehtoa: Toteutetaan yksi taulu kokonaista luokkahierarkiaa varten, muodostetaan yksi taulu jokaista luokkahierarkian luokkaa kohden tai yksi taulu jokaista mahdollista luokkahierarkiassa esiintyvää konkreettista luokkaa kohden. Fowler nimittää näitä kolmea kuvaustapaa nimillä Single Table Inheritance, Class Table Inheritance ja Concrete Table Inheritance. (Fowler, 2002.)

Tuoterakenne on määritetty siten, että siinä hyödynnetään perintää laajasti. Tuotteiden ja komponenttien toteutuksessa on eroteltu yhteiset osat abstraktiin luokkaan, jonka konkreettinen tuotetyyppi perii ja toteuttaa. Tällöin on tarkasteltava olioiden periytymisen toteuttamista relaatiotietokantaan. Työssä on hyödynnetty Class Table Inheritance -perintää laajasti, koska sen ominaisuudet ovat hyödynnettävissä tuoterakenteen toteutuksessa.

3.4.1 Yksi taulu koko luokkahierarkialle (Single Table Inherintace)

Tässä menetelmässä perintä on toteutettu siten, että kaikki luokkahierarkian oliot tallennetaan yhteen tietokantatauluun. Tämä kyseinen taulu sisältää siis kaikkien olioiden tiedot.



Kuva 1. Single Table Inheritance (Fowler, 2002).

Menetelmän hyvinä puolina voidaan mainita, että luokkahierarkian kuvaaminen yhden ainoan taulun avulla yksinkertaistaa tietokantarakennetta ja olioiden haut tietokannasta nopeutuvat. Näin luokkahierarkian hakemisessa ei tarvita liitoslauseita (join), joten tarvittavat SQL-lauseet ovat yksinkertaisia. Lisäksi olioiden välisten suhteiden muutokset eli attribuuttien siirto eri hierarkiatasolle on helppoa, koska tietokantamuutoksia ei tarvitse toteuttaa. (Mak & Guruzu, 2010.)

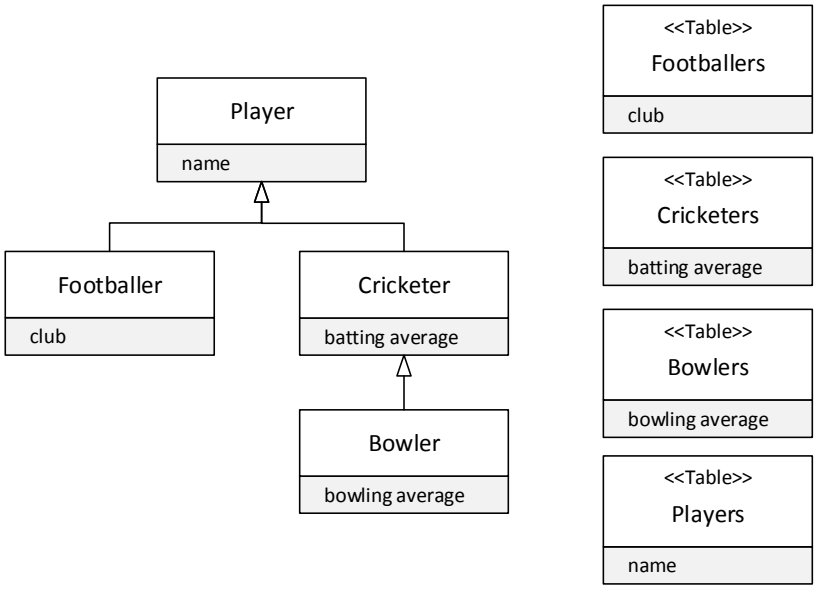
Menetelmällä on myös heikkous. Jos luokkahierarkia on suuri ja monimutkainen, voi tauluun tallennettavissa sarakkeissa olla paljon tyhjiä sarakkeita, mikä voi aiheuttaa tietokannan suorituskyvyn heikkenemistä. Tosin nykyiset tietokannat, kuten Oracle, pystyvät pakkaamaan tällaiset taulut. Jos tarvittava tietokantataulu kasvaa sarakkeiden lukumäärällisesti suureksi, voivat indeksointi ja tietokantalukitukset aiheuttaa suorituskyvyn heikkenemistä. Lisäksi osaa taulun sarakkeista saatetaan tarvita tietyssä tilanteessa mutta toisessa tilanteessa taas ei. Tämä aiheuttaa hämmennystä, jos tilannetta katsotaan pelkästään tietokannan kohdalta. (Fowler, 2002.)

Yksi taulu koko luokkahierarkialle -menetelmässä sovelluksen tietomalli ja tietokantaan tallennettavat suhteiden kuvaukset ovat epämääräiset. Tämän seurauksena mallia ei ole hyödynnetty tuoterakenteessa.

3.4.2 Yksi taulu luokkaa kohti (Class Table Inheritance)

Class Table Inheritance -menetelmässä periytymisen kuvaamisessa käytetään yhtä taulua kaikkia oliomallin luokkia kohden. Luokat voivat olla abstrakteja tai konkreettisia. Perintäsuhteet kuvataan tietokannassa vierasavainviittauksilla siten, että aliluokan taulussa on vierasavain, joka viittaa yliluokan tauluun.

Menetelmän vahvuutena on, että jokainen taulu on helposti ymmärrettävissä eikä se hukkaa tietokannan resursseja turhille sarakkeille. Lisäksi sovelluksen tietomalli ja tietokantaan tallennettavat suhteiden kuvaukset ovat hyvin suoraviivaiset. Tämän vuoksi perintämenetelmää voidaan käyttää laajennettavassa ja uudelleenkäytettävässä tuoterakenteessa.



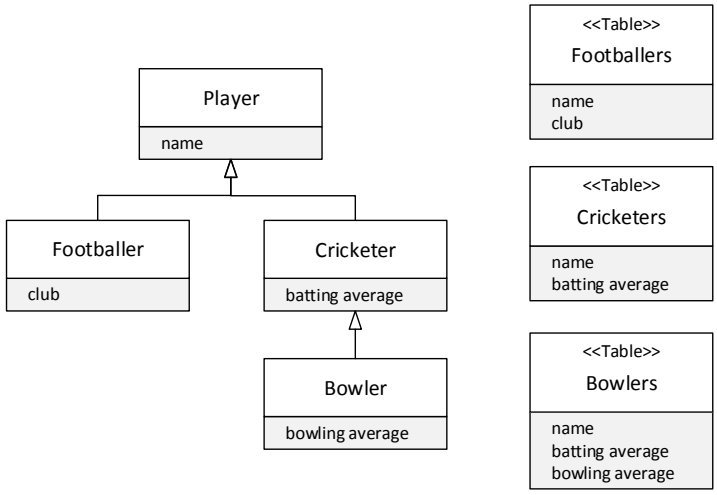
Kuva 2. Class Table Inheritance (Fowler, 2002).

Menetelmän ongelmat liittyvät tehokkuuteen. Jos luokkahierarkiassa on paljon luokkia ja ongelmana on saada haettua tiedot useista tauluista tehokkaasti, joudutaan tekemään liitoslauseita (join) ja mahdollisesti useita eri tietokantahakuja. Näitä ongelmia voidaan

kuitenkin ratkoa tietokantahakuja optimoimalla ja hyödyntämällä indeksointia. Toinen ongelma liittyy siihen, että tehtäessä luokkien rakenteisiin muutoksia täytyy myös tietokantataulut päivittää vastaamaan uutta rakennetta. Lisäksi ongelmana voivat olla yläluokkataulut, joiden käyttäminen voi muodostaa pullonkaulan sovellukseen. Jos niihin täytyy jatkuvasti tehdä muutoksia, tietokantalukitukset voivat haitata sovelluksen toimintaa. (Fowler, 2002.)

3.4.3 Yksi taulu konkreettista luokkaa kohti (Concrete Table Inheritance)

Tässä menetelmässä käytettään yhtä tietokantataulua jokaista konkreettista luokkaa kohti, eli abstrakteille luokkahierarkian luokille ei muodosteta omia tauluja kuten Class Table Inheritance -menetelmässä. Jos kaksi tai useampi aliluokka perii saman yläluokan, perinnän toteutuksessa jokainen aliluokka toteuttaa abstraktin yläluokan attribuutit omaan tietokantatauluunsa. Abstraktin yläluokan kentät ovat siis useaan kertaa aliluokissa. (Mak & Guruzu 2010.)



Kuva 3. Concrete Table Inheritance (Fowler, 2002).

Menetelmän vahvuuksina voidaan pitää sitä, että jokainen taulu on itsenäinen eikä tauluilla ole merkityksettömiä sarakkeita. Koska ylä- ja aliluokan attribuutit ovat samassa taulussa, tietokannan käsittely helpottuu ja tehostuu. Menetelmän heikkouksina on pääavainten (primary key) vaikea hallinta, koska on vaikeaa valvoa tietokannan suhteita abstrakteihin luokkiin. Konkreettinen luokka voi sovelluksen suorituksen aikana olla assosiaatiossa minkä tahansa abstraktin yläluokan aliluokan kanssa. Lisäksi jos abstraktiin luokkaan

tehdään muutoksia, muutokset täytyy myös toteuttaa kaikkiin aliluokkiin, mikä taas heikentää luokkahierarkian ylläpidettävyyttä. (Fowler, 2002.)

Abstraktiin luokkaan tehtävien muutosten vaikutukset kaikkiin aliluokkiin sekä sovelluksen suorituksen aikana tietokannan suhteiden valvonta abstrakteihin luokkiin aiheuttavat sen, että menetelmää ei hyödynnetä tuoterakenteen perinnän toteutuksessa.

4 KERROSARKKITEHTUURI

Tässä kappaleessa tarkastellaan kerrosarkkitehtuuria, joka on yksi keskeisimmistä sovellusarkkitehtuurityypeistä. Tuoterakenne on tavallisesti osana suurempaa ERP-järjestelmää, joka koostuu useista muista kokonaisuuksista, kuten laskutuksesta ja varastoinnista. Eri osien väliset toiminnallisuudet on järkevää toteuttaa eri kerroksiin ja kerroksien välisiin suhteisiin on kiinnitettävä huomiota. Tuoterakenne kannattaa toteuttaa omaan kerrokseen siten, että tuoterakenteella ei ole riippuvuuksia muihin kerroksiin.

4.1 Kerrosarkkitehtuurin ominaisuudet

Kerrostus (layering) on arkkitehtuuri, jonka avulla sovelluksia voidaan jäsentää jakamalla niitä toiminnallisuuksien mukaan eri ryhmiin eli kerroksiin, joista jokainen muodostettu kerros kuvaa tietyn tason abstraktiota. Kerrostus on yksi yleisimmistä tekniikoista, joita ohjelmistosuunnistelussa hyödynnetään jakamalla monimutkaisia järjestelmiä sopiviin osiin. (Fowler, 2002.) Tuoterakenteen toteutusta on tarkasteltava siten, että sen kerrostaminen järjestelmässä on järkevää ja tarkoituksen mukaista.

Järjestelmäkerrokset muodostetaan teknisten ominaisuuksien perusteella, ja kerrosten järjestys on usein nouseva siten, että matalimmalla tasolla on laitteet ja korkeimmalla tasolla ihmiset (Buschmann et al., 1996). Kerrosten muodostamisen perusteina voi olla tehtävä, toiminto, toiminnan kohde tai laitteisto. Kerrosten eri ominaisuuksien vuoksi eri kerrosten tunnistaminen voi osoittautua vaikeaksi. Kerrostuksen periaate on, että ylemmän kerroksen palvelut käyttävät hyväkseen alemman kerroksen palveluita ja alempi kerros on täysin tietämätön ylemmän kerroksen olemassaolosta. (Fowler, 2002.) Kun ylempi kerros käyttää pelkästään alemman kerroksen palveluita, on tällöin kyse suljetusta tai puhtaasta kerrosarkkitehtuurista. Suljetusta kerrosarkkitehtuurista on tietyissä tilanteissa tarvetta poiketa. Poikkeamia on kahdentyyppisiä: palvelukutsu siirtyy alemmasta kerroksesta ylempään kerrokseen (hierarchy breach) tai palvelukutsu ohittaa kerroksen kulkiessa ylimmistä kerroksista alempiin kerroksiin (bridging). Tällöin on kyseessä avoin kerrosarkkitehtuuri. (Laine & Paakki 2001.)

Avoimen kerrosarkkitehtuurin hyödyntämisellä pyritään lisäämään joustavuutta ja suorituskykyä. Tämä johtaa huonompaan ylläpidettävyyteen, koska kerrosten väliset riippuvuudet lisääntyvät. Suljettu kerrosarkkitehtuuri takaa paremman ylläpidettävyyden ja selkeyden, mutta voi johtaa huonompaan suorituskykyyn. (Laine & Paakki 2001.)

Fowlerin mukaan kerroksiin jakamisesta on seuraavia hyötyjä:

- Yksittäisen kerroksen toiminnallisuus voidaan ymmärtää johdonmukaisena kokonaisuutena, jolloin muiden sovelluksen kerrosten toiminnallisuuksia ei tarvitse tuntea.
- Kerrosten väliset riippuvuudet voidaan minimoida.
- Kerroksen toiminnallisuus voidaan korvata vaihtoehtoisella toteutuksella.
- Kerroksen toteuttamisen jälkeen toteutetun kerroksen palveluita voidaan käyttää ylemmän tason kerrosten toteutuksissa.
- Eri standardien toteuttaminen mahdollistuu kerrosten hyödyntämisellä.

Kerrostamisen mukana Fowlerin kuvaa seuraavat ongelmat:

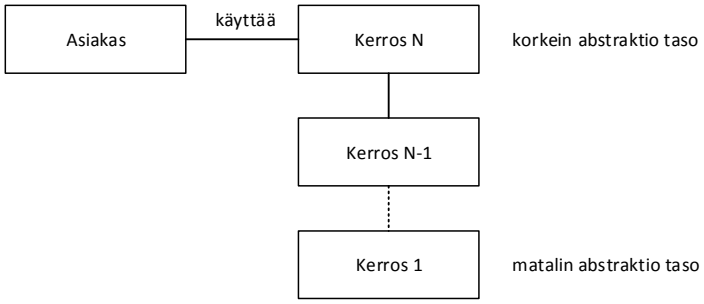
- Kerrokset kykenevät kapseloimaan tiettyjä kokonaisuuksia hyvin, mutta tiettyjä ongelmia esiintyy. Korkealla tasolla tehty muutos aiheuttaa kaskadi-muutoksen, eli korkean ylätason muutos voi johtaa useaan alakerrokseen muutokseen. Esimerkiksi käyttöliittymään tehdään lisäys, joka vaatii uuden tiedon tallentamisen tietokantaan. Tällöin muutokset on tehtävä mahdollisesti kaikkiin kerroksiin käyttöliittymän ja tietokannan välillä.
- Ylimääräiset kerrokset voivat heikentää sovelluksen suorituskykyä. Kerrosten välisissä siirtymisissä tarvitsee tavallisesti tehdä tiedon muutoksia esitysmuodosta toiseen. Kerrosten järkevällä suunnittelulla ja optimoinnilla voidaan kuitenkin ongelmaa pienentää.

4.2 Kerrostus

Kerrostuksen toteuttamisessa Buschmannin et. al. mukaan sovelluksen rakenteen jakaminen oikeaan kerrosten lukumäärään sekä kerrosten asettaminen toiminnallisuuden kannalta sopivaan järjestykseen on tärkeää. Kerrosten muodostaminen aloitetaan

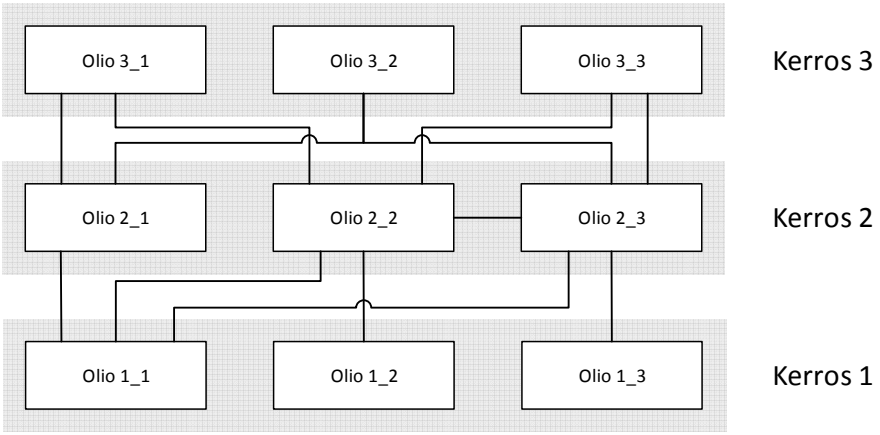
matalimman abstraktion tasolta ja kerrosta kutsutaan ensimmäiseksi kerrokseksi (1. kerros). Ensimmäisen kerroksen päälle lisätään kerroksia aina, kun abstraktiotaso vaihtuu, siten, että kerros J lisätään kerroksen J-1 päälle, kunnes saavutetaan ylimmän tason abstraktio eli kerros N.

Edellisessä kerrosten määrittelemisessä on huomioitava, että siinä ei oteta kantaa, missä järjestyksessä kerrokset suunnitellaan, vaan siinä tarjotaan vain käsitteellinen näkymä. Lisäksi ei oteta kantaa, täytyykö yksittäisen kerroksen J olla monimutkainen osajärjestelmä, joka täytyy edelleen pilkkoa pienempiin tai joka välittää palvelupyyntöjä kerrokselta J+1 kerrokselle J-1. Tärkeää on, että samassa kerroksessa sijaitsevat osat toimivat samalla abstraktiotasolla.



Kuva 4. Kerrostus (Buschmann et al., 1996).

Kerroksen J palvelut on toteutettu kerroksen J-1 tarjoamista palveluista, eli ylimmän kerroksen palvelut toteutetaan alemman kerroksen palveluiden avulla. Ongelmana on päättää, mitä kerroksia täytyy muodostaa ja mitkä ovat kerrosten väliset vastuut (Fowler, 2002).



Kuva 5. Kerrosten oliot (Buschmann et al., 1996).

Yksittäiset kerrokset koostuvat useista monimutkaisista komponenteista eli olioista, joilla on erityyppisiä toimintoja ja ominaisuuksia. Eri kerrosten oliot kutsuvat toistensa operaatioita kerrosten rajapintojen määritysten mukaan. (Buschmann et al., 1996.)

4.3 Kerrosten kehitys sovelluskehityksessä

Eräajojärjestelmien (batch systems) aikakaudella ei kerrostukselle nähty erityistä tarvetta, jolloin toteutetut sovellukset suoritettiin eräajoina ja sovellukset toteuttivat niille asetetut tehtävät. Kerrosten ilmeinen tarve alkoi kasvaa 1990-luvulla asiakas-palvelin-arkkitehtuurin (client/server) yleistyessä. Kyseinen tunnetaan myös kaksikerrosarkkitehtuurina, jossa asiakassovellukset sisältävät käyttöliittymän ja liiketoimintalogiikan toteutuksen. Palvelinsovellus oli tavallisesti relaatiotietokanta, jota asiakassovellus hyödynsi toiminnoissaan. Kaksikerrosarkkitehtuuri suoriutuu tehtävästään, jos sovelluksen pääasiallinen tehtävä on esittää tietoa sekä tehdä päivityksiä relaatiotietokantaan. Ongelmia aiheuttaa käyttöliittymään upotettu liiketoimintalogiikka sekä liiketoimintalogiikan ylläpito. Lisäksi sovelluksen osien uudelleen käytettävyys on heikkoa. Jos liiketoimintalogiikka monimutkaistuu, ongelmana on myös se, että sovelluksen hallinta vaikeutuu ja duplikaattikoodin määrä kasvaa. Tämä johtaa siihen, että yksinkertainen muutos voi aiheuttaa muutoksia useisiin kohtiin sovelluksessa, jossa duplikaattikoodia on käytössä. (Fowler, 2002.)

Näiden ongelmien ratkaisuun on vaihtoehtona hyödyntää palvelinsovelluksen tietokantaa ja pyrkiä toteuttamaan liiketoimintalogiikkaa tallennetuilla proseduureilla (stored procedure). Tallennettujen proseduurien käyttö perustuu siihen, että liiketoimintalogiikkaa voidaan tallentaa näihin tietokantaproseduureihin ja asiakassovellus eli käyttöliittymä kutsuu tallennettuja proseduureja sopivilla parametreilla. Menetelmän ongelmana on, että tietokantaproseduureilla on rajoittuneet mahdollisuudet datan strukturointiin, mikä johtaa vaikeasti ylläpidettävään toteutuskoodiin. Menetelmän etuina voidaan pitää SQL-standardia, joka mahdollisti eri tietokantatoimittajan. Todellisuudessa tietokannan vaihto ei ollut ongelmatonta, koska tietokantaproseduurit eivät kuuluneet SQL-standardiin, joka käytännössä rajoitti tietokannan mahdollisuuden vaihtoa järkevillä kustannuksilla. (Fowler, 2002.)

Kaksikerrosarkkitehtuurin yleistyessä myös oliopohjainen suunnittelu ja olio-ohjelmointi alkoivat kasvattaa suosiotaan. Olio-ohjelmointiyhteisöllä oli ratkaisu edellä kuvattuun ongelmaan: lisätään yksi kerros eli siirrytään kolmikerrosarkkitehtuuriin. Tässä ratkaisussa on esityskerros (presentation) käyttöliittymälle, liiketoimintakerros (domain) liiketoimintalogiikkaa varten sekä tietovarastolle eli tietokannalle (data source). Eli pyritään ratkaisemaan ongelmat erottamalla liiketoimintalogiikka täysin omaksi kerrokseksi. Kolmikerrosarkkitehtuuri alkoi voimakkaasti yleistyä WWW (world wide web) yleistymisen seurauksena, kun kaksikerrosarkkitehtuurisovelluksia oli pystyttävä käyttämään selainkäyttöliittymillä. Tämän seurauksena asiakassovellusten liiketoimintalogiikka täytyi pystyä erottamaan omaan kerrokseen. Näin sovelluksia voitiin käyttää sekä perinteisellä käyttöliittymällä että selainpohjaisilla versioilla. (Fowler, 2002.)

4.4 Kolmikerrosarkkitehtuuri

Kerrosarkkitehtuuria hyödyntävissä sovelluksissa voidaan erottaa kolme keskeistä kerrosta. Lisäksi kerrosarkkitehtuuria hyödynnettäessä näiden kolmen kerroksen lisäksi sovellukset voivat sisältää muitakin kerroksia, jotka tukevat eri toimintoja. Keskeiselle kolmelle kerrokselle on lähteistä riippuen eri nimiä. Brown et al. käyttää kerroksista seuraavia nimityksiä: käyttöliittymäkerros/esityskerros (presentation), liiketoimintakerros (domain

logic) ja tietokantakerros (data source). Myös Fowler käyttää kerroksista samoja nimityksiä.

Presentation eli esityskerros koostuu olioista, jotka vastaavat käyttäjän syötteistä ja sovelluksen tulosten esittämisestä (Brown, 2003). Esityskerros huolehtii käyttäjän ja järjestelmän välisestä vuorovaikutuksesta sekä kerroksen vastuulla on tiedon esittäminen käyttäjille ja käyttäjän syöttämien komentojen tulkitseminen liiketoimintakerroksen ja tietokantakerroksen suorittamista varten (Fowler, 2002).

Domain Logic eli liiketoimintakerros on järjestelmän monimutkaisin kerros niin kerroksen toiminnan ymmärtämisen kuin toteutuksen kannalta. Liiketoiminnan mallintaminen hyödyntämällä oliokeskeisyyttä mahdollistaa tehokkaan järjestelmän toteutuksen, jolloin oliopohjaisuutta voidaan hyödyntää ongelmien ratkaisussa sekä lisätä toteutuksen joustavuutta. (Brown, 2003.) Liiketoimintakerroksen vastuulla on liiketoimintalogiikan vaatimusten toteuttaminen. Kerros sisältää varastoituun tietoon kohdistuvia toimintoja, esityskerrokselta tulevia tiedon oikeellisuuksien tarkistuksia sekä kommunikointia tietokantakerroksen kanssa. (Fowler, 2002.)

Data Source eli tietokantakerroksen tehtävänä on vastata järjestelmän tiedon tallennuksesta, tiedon hakemisesta tai kommunikoinnista muiden ulkoisten järjestelmien kanssa (Brown, 2003). Tietokantakerroksen vastuulla on kommunikointi järjestelmien kanssa jotka suorittavat tarvittavia toimintoja järjestelmän puolesta. Näitä ovat esimerkiksi transaktiomonitorit, toiset sovellukset ja viestinvälitysjärjestelmät. Kerroksen tärkeimpänä tehtävänä on kuitenkin huolehtia yhteyksistä relaatiotietokantaan. (Fowler, 2002.)

Työssä tarkasteltavan tuoterakenteen toteutuksessa on hyödynnetty kolmikerrosarkkitehtuuria ja tuoterakenteen toiminnallisuus on toteutettu liiketoimintakerrokseen. Tuoterakenteen pysyvä tieto tallennetaan tietokantakerrokseen, jota liiketoimintakerros käyttää ja hyödyntää. Työssä ei ole toteutettu esityskerrosta, mutta esityskerroksen toteutuksessa voidaan hyödyntää tuoterakenteen julkisivua eli facadea, joka tarjoaa esityskerrokselle tarvittavat operaatiot.

5 SOVELLUSLOGIIKKAMALLIT

Tuoterakenteen toteutuksessa hyödynnettiin Domain-mallia, joka mahdollistaa monimutkaisten liiketoimintasovellusten toteuttamisen. Tässä kappaleessa kuvataan transaktiosarja-, Table Module -malli, Domain-malli sekä Service Layer -malli, joiden ominaisuuksia vertaamalla voidaan päätellä tuoterakenteen vaatimuksille parhaiten soveltuva sovelluslogiikkamalli.

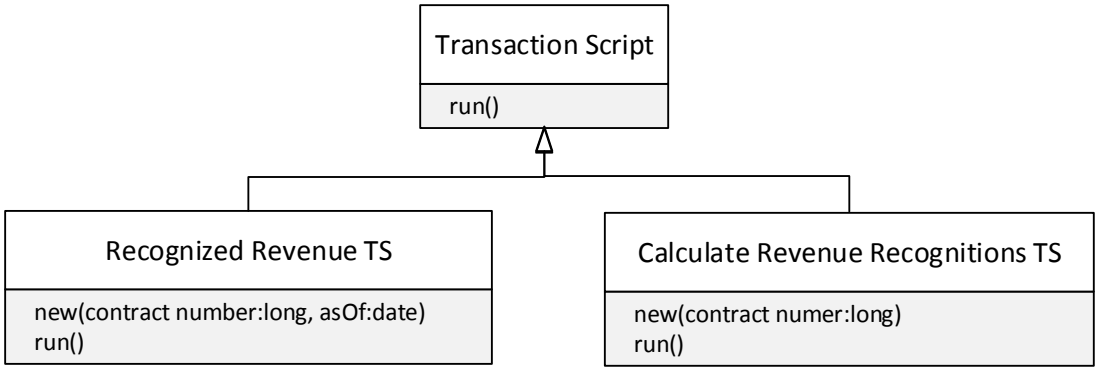
5.1 Transaktiosarja (Transaction Script)

Monista liiketoimintasovelluksista (business application) voidaan ajatella, että ne koostuvat transaktiosarjoista. Jokainen asiakkaan ja palvelimen välillä määritetty transaktio sisältää eri määrän liiketoimintalogiikkaa. Yksinkertaisimmillaan liiketoimintalogiikkatapahtuma voi olla tiedon esitystä, joka on haettu tietokannasta, tai koostua erittäin monimutkaisista vaiheista, tarkistuksista ja laskennoista. Transaktiosarjamallissa (Transaction Script) liiketoimintalogiikka jaetaan yksittäisiin proseduureihin siten, että jokainen yksittäinen proseduuri käsittelee yhden toiminnallisuuden. Jokaisella yksittäisellä liiketoimintatapahtumalla on oma transaktionsarja, jonka toiminnallisuus voidaan jakaa pienempiin aliproseduureihin. (Fowler, 2002.)

Transaktiosarjamallin sarjat sisältävät sovelluksen logiikan, ja niiden käyttö riippuu laajasti sovelluksen kerrosten toteutuksesta. Transaktiosarjat on pyrittävä eriyttämään ja kapseloimaan mahdollisimman paljon omiin kokonaisuuksiin ja mahdollisesti toteuttamaan ne omina erillisinä luokkina. Niillä ei saa olla riippuvuuksia esimerkiksi käyttöliittymän toteutukseen.

Transaktionsarjan rajapintakuvaus:

```
public interface TransactionScript {
	abstract public void run();
}
```



Kuva 6. Transaction Script (Fowler, 2002).

Transaktiosarjamallin etu on sen yksinkertaisuus, ja sen toiminta soveltuu parhaiten tilanteisiin, jossa sovelluslogiikka on yksinkertaista ja logiikan määrä vähäistä. Tällöin mallin yksinkertaisuus ja nopea kehittäminen voivat olla merkittäviä tekijöitä, ja ne voivat edesauttaa yksinkertaisen ratkaisun syntymistä. (Fowler, 2002.)

Logiikan kompleksisuuden kasvaessa transaktiosarjojen ylläpidolle ja kehitykselle tulee paljon haasteita, esimerkiksi toteutetun sovelluskoodin luettavuus heikkenee, eri transaktiokoodien kahdentumista voi esiintyä ja suunnittelun selkeys alkaa heikentyä (Gamma et al., 1994). Tämän seurauksena transaktiosarjaa ei ole järkevää hyödyntää toteutettavassa tuoterakenteessa.

5.2 Table Module -malli

Kompleksisuuden kasvaessa transaktiosarjan heikkoudet tulevat esille, ja liiketoimintalogiikan ja tiedon hallintaan sekä kapselointiin on kiinnitettävä enemmän huomiota. Table Module -mallin tarkoitus on, että liiketoimintalogiikka asetetaan luokkaan ja yhtä tietokantataulua vastaa vain yksi luokka. Muodostettu luokka vastaa taulun rivien lisäyksestä, poistoista, muokkauksista, latauksista ja eri hakutoiminnoista. Lisäksi luokka sisältää liiketoimintalogiikan vaatimat toteutukset. Lisäksi on huomioitava, että Table Module -malli ei tue kunnolla perintää, minkä vuoksi Table Module -mallia ei ole järkevää hyödyntää tuoterakenteen toteutuksessa.

Kun viitataan tietokannassa olevaan olioon, mallin käytössä on tavallisesti tiedettävä olion yksilöivä tunniste, joka tavallisesti on tietokannan taulun pääavain. Mallin käytössä on tavallista, että käsiteltävä tieto on taulu muotoista. Lisäksi malliin käyttämä tieto on hyvin keskeisessä asemassa, koska pääsy mallin tietoon on suoraviivaista. Mallin etuna voidaan mainita, että se sallii paketoida datan ja datan käsittelyn yhteen ja lisäksi sen mallintaminen relaatiotietokantaan on selkeää. Mallin heikkoutena on puolestaan se, että malli ei pysty hallinnoimaan monimutkaista sovelluslogiikkaa. Suorat olioiden väliset viittaukset eivät ole mahdollisia, eikä polymorphismi toimi kunnolla. (Fowler, 2002.)

5.3 Domain Model -malli

Liiketoimintalogiikka voi olla hyvin monimutkaista. Liiketoiminnan säännöt ja menetelmät voivat olla eri tapauksissa hyvin vaikeasti kuvattavia, ja monimutkaisuus voi johtua erikoistapauksista ja mallinnuksen kompleksisuudesta. Domain-malli pystyy luomaan olioiden välille vuorovaikutteisen verkon, jossa jokainen olio edustaa mallin kannalta merkittävää tekijää sovellusalueessa.

Sovelluksessa Domain-mallin oliot muodostavat kerroksen, joka mallintaa liiketoimintalogiikkaa, jota varten malli on toteutettu. Mallin oliot siis mallintavat liiketoimintaa, ja olioihin on toteutettu liiketoimintasäännöt.

Domain-malli voidaan jakaa kahteen eri osa-alueeseen: yksinkertaiseen (Simple Domain Model) ja monimuotoiseen (Rich Domain Model). Molemmat tapaukset sisältävät myös toiminnallisuuden, ei pelkästään tietokantaan tallennettavia tietoja. Yksinkertaisessa mallissa Domain-mallin oliot muistuttavat suuresti tietokanta tauluja. Monimuotoisessa Domain-mallissa oliot ja tietokanta näyttävät erilaisilta olioiden välisten monimutkaisten verkostojen, perintähierarkian ja mahdollisten optimointien vuoksi. (Larman, 2004.)

Koska liiketoimintakerroksen logiikassa voi tapahtua paljon muutoksia, on tärkeää, että pystytään toteuttamaan muutokset sekä testaamaan kerroksen muutosten toiminnallisuutta. Tämän vuoksi on järkevää pyrkiä pitämään Domain-malli mahdollisimman eriytettynä

muista sovelluksen kerroksista, eli mallin riippuvuudet kannattaa pitää mahdollisimman pieninä muiden sovelluskerrosten välillä.

Domain-mallin ongelmana voi olla se, että mallin olioiden lukumäärä kasvaa suureksi. Tämän vuoksi on tärkeää pohtia mallin käytöstä aiheutuvaa monimutkaisuutta ja verrata mallista saatavia hyötyjä. Jos mallinnettava kohde on muutosherkkä, sisältää paljon liiketoimintasääntöjä tai laskentaa, on mallin käyttö järkevää. Jos mallinnuksen kohde on yksinkertainen, sisältää yksinkertaisia tarkastuksia ja laskentaa, voi transaktiosarja olla parempi vaihtoehto. (Fowler, 2002.)

Työn toteutuksessa on hyödynnetty monimuotoista Domain-mallia, joka soveltuu tuoterakenteen sovelluslogiikkamalliksi ominaisuuksiensa puolesta. Domain-mallin toteutus voidaan toteuttaa hyödyntämällä suunnittelumalleja tai hyödyntää olemassa olevia sovelluskehitysympäristöjä. Työssä on hyödynnetty J2EE:tä, jotka kykenevät pysyvien olioiden toteutukseen. Jos malli toteutetaan suunnittelumallien avulla, on kiinnitettävä huomiota olioiden ja tietokannan väliseen vuorovaikutukseen. Tällöin joudutaan hyödyntämään Data Mapper -suunnittelumallia, joka mahdollistaa tietokannan ja liiketoiminnan erottamisen. Lisäksi mallin olioiden toiminnoista joudutaan pitämään kirjaa, mihin hyödynnetään Unit of Work -suunnittelumallia. Koska malli on erotettu omaksi kerrokseksi, mallia käytetään Service Layerin välityksellä.

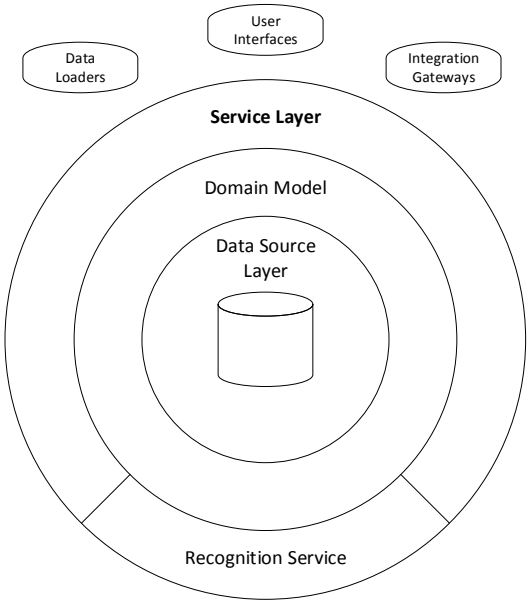
5.4 Service Layer -malli

Palvelukerros eli Service Layer kapseloi sovelluksen liiketoimintalogiikan ja määrittelee rajat ja operaatiot, jotka toimivat ylempien kerrosten käyttämien operaatioiden rajapintoina. Tämän lisäksi palvelukerros koordinoi sovelluksen operaatioiden hallintaa.

Sovellukset tarvitsevat erityyppisiä rajapintoja ulkoiseen ympäristöön, joiden välityksellä sovellus vastaanottaa dataa, jota sovellus tarvittaessa tallentaa tai käyttää liiketoimintaoperaatioiden syötteinä. Rajapintoina ovat tyypillisesti tiedon lataajat (data loader), käyttöliittymät ja integraatiokanavat (integration gateways). Rajapintojen eri toimintojen vuoksi ne tarvitsevat yhtenäisiä yleisiä toimintoja, joilla ne voivat olla

vuorovaikutuksessa sekä muokata sovelluksen dataa ja kutsua liiketoimintalogiikkaoperaatioita. (Fowler, 2002.)

Palvelukerroksen määrittelemät rajapinnat määrittelevät siis joukon operaatioita, joita sovelluksen asiakkaat voivat käyttää. Lisäksi kerros kapseloi sovelluksen liiketoimintalogiikan, kontrolloi transaktioita ja koordinoi operaatioiden palauttamia tuloksia. (Broemmer, 2003.)



Kuva 7. Service Layer (Fowler, 2002).

Palvelukerroksen toteuttamisessa on kaksi perustoteutusvaihtoehtoa: julkisivu eli facade-lähestyminen sekä operaatiopohjainen (operation script) lähestyminen.

Facade-lähestymisessä palvelukerros toteutetaan joukkona kevyitä facadeja, jotka hyödyntävät Domain-mallia. Facaden toteuttavat luokat eivät sisällä liiketoimintalogiikkaa, vaan kaikki toiminnallisuus on sijoitettu Domain-malliin. Tästä seuraa, että palvelukerros on muodostettu joukosta facadeja, jotka toimivat rajana ja sisältävät operaatiot, joita sovelluksen asiakkaat voivat käyttää. Tuoterakenteen toteutuksessa on toteutettu facade, joka sisältää tuoterakenteen toiminnallisuuden operaatiot.

Operaatiopohjaisessa toteutuksessa palvelukerros toteutetaan joukkona luokkia, jotka toteuttavat sovelluksenlogiikan suoraan. Palvelukerros on siis toteutettu transaktiosarjoina. Ne on lajiteltu sopiviin luokkiin, jotka toteuttavat tietyt tehtävät. Jokaista edellä kuvattua luokkaa voidaan pitää palveluna (service). Nämä palvelut siis näkyvät asiakkaille rajapinnassa, joita asiakkaat voivat käyttää. (Fowler, 2002.)

5.5 Sovelluslogiikkamallien vertailu

Tuoterakenteen toteutuksessa on tarkoitus mallintaa yrityksen liiketoimintalogiikan sääntöjä. Yrityksen toiminnan seurauksena liiketoimintalogiikan sääntöihin tulee lukuisia muutoksia ja poikkeuksia tuoterakenteen elinkaaren aikana. Tämän vuoksi sovelluslogiikan on oltava modulaarista, jotta muutosten toteuttaminen olisi mahdollisimman yksinkertaista ja tehokasta.

Transaktiosarjan tarkoitus on jakaa sovelluslogiikka yksittäisiin toimintaohjeisiin eli operaatioihin. Jokaista operaatiota kohti muodostetaan oma transaktiosarja, esimerkkinä yhdestä transaktiosarjasta voisi olla uuden tuotteen muodostaminen. Transaktiosarjan hyvinä puolina on niiden yksinkertaisuus, jos sovelluslogiikka on yksinkertaista ja liiketoimintalogiikan määrä on vähäistä. Mallin heikkoutena on puolestaan se, että liiketoimintalogiikan kompleksisuuden kasvaessa saattaa esiintyä transaktioiden kahdentumista, suunnittelun selkeys alkaa heikentyä ja toteutuksessa saattaa esiintyä kahdennettua koodia. Transaktiosarja on parhaimmillaan yksinkertaisen ja pysyvän sovelluslogiikan toteutuksessa.

Sovelluslogiikan kompleksisuuden kasvaessa transaktiosarjan heikkouksien kumoamiseen voidaan hyödyntää Table Module -mallia. Mallin tarkoituksena on järjestää sovelluslogiikka luokkiin siten, että karkeasti yhtä luokkaa vastaa yksi tietokantataulu ja muodostetun luokan operaatiot vastaavat kaikista tietokantatauluun kohdistuvista operaatioista. Mallin heikkoutena on, että luokkien perinnän toteuttaminen on työlästä. Tämä voi johtaa tilanteeseen, että yhtä tietokantataulua käytetään jokaista konkreettista luokkaa kohti (Concrete Table Inheritance). Tällöin abstraktiin luokkaan tehdyt muutokset jouduttaisiin toteuttamaan jokaiseen perittyyn konkreettiseen luokkaan.

Kompleksisen sovelluslogiikan sekä siihen liittyvien muutosten käsittelyyn ja mallintamiseen hyödynnetään Domain Model -mallia. Domain Model -malli muodostaa toisiinsa liittyvien olioiden välisen verkoston. Siinä oliot edustavat liiketoimintaa, johon malli on rakennettu. Domain Model -malli muistuttaa tietokantamallia, mutta tietokantaan tallennetaan ainoastaan tietoa, ja Domain Model -mallin pitää käsittää sekä toiminnot ja tiedon. Oliot voivat muodostaa monimutkaisia verkostoja sekä perintähierarkioita. Domain Model -malli kannattaa toteuttaa omaan kerrokseen, jolla ei ole riippuvuuksia sovelluksen muihin kerroksiin. Sovelluslogiikan muuttuessa Domain Model -malli mahdollistaa muutosten toteutuksen siten, että muutosten vaikutukset ovat vähäisiä sovelluksen muiden kerrosten toimintoihin verrattuna.

Service Layer -malli eli palvelukerros kapseloi sovelluksen sovelluslogiikan ja määrittelee operaatiot, jotka toimivat ylempien kerrosten rajapintoina sekä koordinoi sovelluksen operaatioita. Palvelukerros voidaan toteuttaa julkisivuna eli facadena, jolloin joukko facadeja hyödyntävät Domain Model -mallia. Facade ei sisällä liiketoimintalogiikkaa vaan kaikki toiminnallisuus on Domain Model -mallissa. Palvelukerroksen käyttäminen mahdollistaa tehdä muutoksia Domain Model -malliin ilman, että muutokset vaikuttavat palvelukerroksen käyttäjiin, jos facaden rajapintoihin ei tule muutoksia.

6 SUUNNITTELUMALLIT (DESIGN PATTERNS)

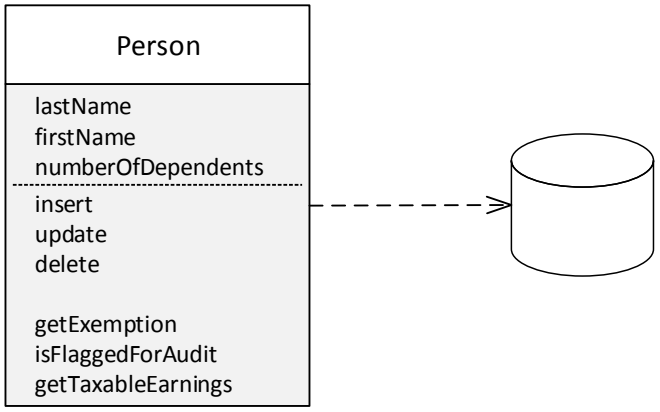
Suunnittelumalli on yleinen ratkaisu usein esiintyvään ohjelmistojen arkkitehtuuri- tai suunnitteluongelmaan tietyssä kontekstissa eli yhteydessä.

Pysyvyyden viitekehys (persistence framework) toimii abstraktiotasona sovellusten ja tietokannan välillä sekä toimii sovelluskehyksenä. Fayadin et al. mukaan sovelluskehys on uudelleenkäytettävä, puoliksi valmis sovellus, josta voidaan toteuttaa useita eri tarkoituksiin sopivia ratkaisuita. Sovelluskehys toimii runkona, jonka ympärille toteutetaan tarvittavat toiminnallisuudet.

Kappaleen tarkoitus on esitellä pysyvyyden viitekehyksen toiminnassaan käyttämiä tärkeimpiä suunnittelumalleja (design patterns) ja pyrkiä yleisellä tasolla kuvaamaan, miten olioiden ja relaatioiden välinen yhteistoiminta automatisoidaan ja mitä niiden käytännön toteuttaminen valmiiden suunnittelumallien avulla tarkoittaa.

6.1 Active Record

Aktiivinen tietue (Active Record) on suunnittelumalli, jonka keskeinen ajatus on kapseloida oliomallin olioiden sovelluslogiikan lisäksi myös tietokannan käsittelyyn liittyvä logiikka ja toiminnallisuus. Aktiivinen tietueen ydin on, että jokainen toteutettu kohdealueen oliomalli vastaa tarkasti tietokannassa olevaa tietomallia. Voidaan ajatella, että tietokannan taulut ovat luokkia, taulun rivit ovat olioita ja taulun kentät ovat olioiden attribuutteja. Jokainen oliomallin luokka on vastuussa omien tietojensa lataamisesta, tallentamisesta sekä tietoon kohdistuvasta sovelluslogiikasta. Sovelluslogiikan toimintoja voi olla myös aktiivisten tietueiden ulkopuolella, kuten transaktiosarjoissa. (Fowler, 2002.)



Kuva 8. Aktiivisen tietueen luokka, attribuutit ja operaatiot (Fowler, 2002).

Luokkan määrittely:

```
class Person...

private String lastName;
private String firstName;
private int numberOfDependents;
```

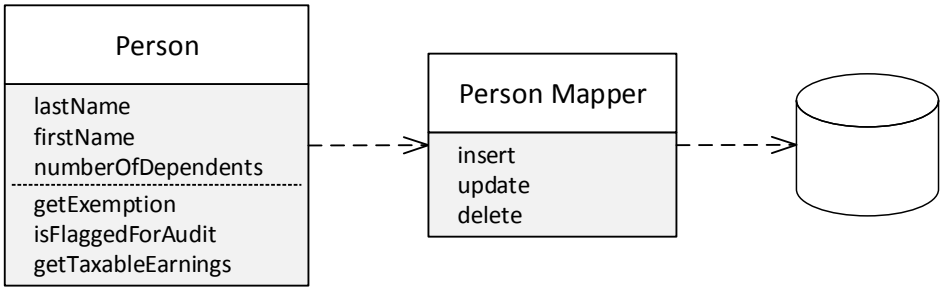
Tietokanta taulu:

```
create table people (ID int primary key, lastname varchar, firstname varchar,
number_of_dependents int);
```

Aktiivisen tietueen luokan tietosisällön täytyy vastata vastaavan tietokantataulun sisältöä siten, että jokaiselle tietokantaulun sarakkeelle löytyy vastaava luokan attribuutti. Jos luokalla on viittaus toiseen luokkaan, voidaan viittaus toteuttaa käyttämällä vierasavainmenetelmää (Foreign Key Mapping) eli kuvataan olioviittaus tietokantataulussa olevaksi vierasavaimeksi.

6.2 Data Mapper

Aktiivisten tietueen kohdalla sovelluslogiikka ja tietokantaoperaatiot ovat tiukasti sidottuina toisiinsa, ja näiden kahden ominaisuuden erottaminen johtaa Data Mapper -mallin hyödyntämiseen.



Kuva 9. Person-luokka ja sitä vastaava PersonMapper-luokka (Fowler, 2002).

Data Mapper -suunnittelumalli on kerros, joka erottaa sovelluksen muistissa olevat oliot tietokannasta. Mallin tärkein tehtävä on olioiden tiedon siirto sovellusohjelman ja tietokannan välillä. Tarkoitus on pitää mallin komponentit mahdollisimman riippumattomina toisistaan ja kommunikaatiossa käytettävät osat pyritään piilottamaan. Data Mapperia hyödynnettäessä muistissa olevien olioiden ei siis tarvitse tietää, missä olioiden tiedot sijaitsevat tietokannassa. Liiketoimintakerroksen olioiden ei tarvitse tietää, että käytössä on tietokanta, eikä niiden tarvitse tietää tietokannan SQL-lauseita eikä tietokannan rakennetta eli tietokantaskeemaa. (Fowler, 2002.)

Data Mapper muodostetaan tavallisesti jokaiselle pysyvälle luokalle. Tällöin Data Mapper pystyy toteuttamaan olioiden lataamisen, luomiseen, päivittämiseen sekä poistamiseen. On mahdollista, että siihen on toteutettu myös erityyppisiä hakutoiminnallisuuksia, ja Data Mapper on saatettu yhdistää Unit of Work -suunnittelumalliin.

Fowler esittelee abstraktin mapperin, joka toimii kaikkien Data Mapper -luokkien yläluokkana ja sisältää kaikkien mapperien yhteiset ominaisuudet, kuten olion lataamisen, luonnin, muokkaamisen ja poiston tietokannasta. AbstractMapper sisältää abstraktit operaatiot eli findStatement, insertStatement, updateStatement, removeStatement, doLoad, doInsert, doUpdate ja doRemove, jotka perivän luokan on toteutettava. Operaatiot findStatement, insertStatement, updateStatement ja removeStatement pitävät sisällään SQL-lauseet, joilla olion haku, luonti, muokkaus ja poisto toteutetaan.

doLoad-operaatio toteuttaa tietokannasta haetun tiedon asettamisen olion atribuutteihin. doInsert-, doUpdate- ja doRemove-operaatiot toteuttavat uuden olion luonnin, olion

muokkaamisen ja olion poistamisen tietokannasta, joita kutsutaan abstraktin luokan operaatioiden avulla. (Fowler, 2002.)

Seuraavassa esimerkissä on kuvattu luokka Person ja sen Data Mapper -luokka PersonMapper. Esimerkistä voidaan huomata, miten tietokanta ja olion rakenne erotetaan toisistaan. PersonMapper perii AbstractMapperin. Tässä esitellään ainoastaan lataus ja uuden olion tallennus. Olion päivitys ja poisto voidaan toteuttaa vastaavasti.

AbstractMapper-luokan karkea esittely on alla. Lisäksi esimerkki sisältää luokan DomainObject, jonka tarkoitus on kerätä yhteen kaikkien pysyvien olioiden yhteiset ominaisuudet, kuten yksilöivä tunnus eli ID.

```
class DomainObject {
       private Long ID;
       public Long getID() { return ID; }
       public void setID(Long ID) {
              Assert.notNull("Cannot set a null ID", ID);
              this.ID = ID;
       }
       public DomainObject(Long ID) {this.ID = ID;}
       public DomainObject() {}
}

class AbstractMapper...
      // Object loading SQL
      abstract protected String findStatement();

      //Object insert SQL
      abstract protected String insertStatement();

      //Object update SQL
      abstract protected String updateStatement();

      //Object remove SQL
      abstract protected String removeStatement();

      //
      abstract protected void doInsert(DomainObject subject, PreparedStatement
      insertStatement) throws SQLException;


      abstract protected void doUpdate(DomainObject subject, PreparedStatement
      updateStatement) throws SQLException;


      abstract protected void doRemove(DomainObject subject, PreparedStatement
      removeStatement) throws SQLException;


      abstract protected DomainObject doLoad(Long id, ResultSet rs) throws
      SQLException;


      protected DomainObject abstractFind(Long id) {
              PreparedStatement findStatement = null;
              try {
```

```
                findStatement = DB.prepare(findStatement());
                findStatement.setLong(1, id.longValue());
                ResultSet rs = findStatement.executeQuery();
        rs.next();
        result = load(rs);
        return result
}
catch (SQLException e) {
        throw new ApplicationException(e);
        }
finally {
        DB.cleanUp(findStatement);
}


protected DomainObject load(ResultSet rs) throws SQLException {
        Long id = new Long(rs.getLong(1));
        DomainObject result = doLoad(id, rs);
        return result;
}

public Long insert(DomainObject subject) {
        PreparedStatement insertStatement = null;

        try {
                insertStatement = DB.prepare(insertStatement());
                subject.setID(findNextDatabaseId());
                insertStatement.setInt(1, subject.getID().intValue());
                doInsert(subject, insertStatement);
                insertStatement.execute();
                return subject.getID();
        } catch (SQLException e) {
        throw new ApplicationException(e);
        } finally {
                DB.cleanUp(insertStatement);
        }
}

// Need to be implemented
// public void update(DomainObject subject)...
// public void remove(DomainObject subject)...
```

Toteutetaan Person-luokka, jonka pysyvät olioiden attribuutit halutaan tallentaa tietokantaan. Lisäksi luokalle voidaan määrittää operaatiot, johon luokan liiketoiminatalogiikat toteutetaan.

```
class Person extends DomainObject

private String lastName;
private String firstName;
private int numberOfDependents;
```

Vastaava tietokantataulu määritetään:

```
create table Person (ID int primary key, lastname varchar,
firstname varchar, number_of_dependents int)
```

Luokan Person Data Mapperin toteutus:

```
class PersonMapper extends AbstractMapper...

        protected String findStatement() {
```

```
        return "SELECT " + COLUMNS + " FROM person" + " WHERE id = ?";
}

protected String insertStatement() {
        return "INSERT INTO person VALUES (?, ?, ?, ?)";
}

public static final String COLUMNS = " id, lastname, firstname,
number_of_dependents ";

public Person find(Long id) { return (Person) abstractFind(id); }


protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
        String lastNameArg = rs.getString(2);
        String firstNameArg = rs.getString(3);
        int numDependentsArg = rs.getInt(4);
        return new Person(id, lastNameArg, firstNameArg, numDependentsArg);
}

protected void doInsert(
        DomainObject abstractSubject, PreparedStatement stmt)
        throws SQLException
{
        Person subject = (Person) abstractSubject;
        stmt.setString(2, subject.getLastName());
        stmt.setString(3, subject.getFirstName());
        stmt.setInt(4, subject.getNumberOfDependents());
}
```

AbstractMapper mahdollistaa helposti uusien luokkien määrittelyt ja niiden liittämisen tietokantaan. Olioiden väliset suhteet voidaan määrittää vierasavaimella ja tarvittaessa hoitaa myös niiden lataukset ja päivitykset doInsert-, doUpdate- ja doRemove-operaatioiden avulla.

6.3 Identity Map

Ladattaessa tietokannasta olioita sovelluksen muistiin täytyy varmistaa, että jokainen ladattu olio on ladattu ainoastaan yhteen kertaan (yhden transaktion aikana). Identity Map -menetelmä toteuttaa tämän varmistuksen. Identity Map on ensimmäinen sovelluskehyksen osa, jossa tarkastellaan olion lataamista tietokannasta, kun olioon viitataan liiketoimintakerroksessa. Identity Map -toiminnan edellytys on, että tietokannasta ladattavat oliot säilötään sopivaan Map-rakenteeseen. Aina kun uusi olio on ladattava tietokannasta, tarkistetaan ensin Identity Mapistä, onko kyseistä oliota ladattu aikaisemmin muistiin. Jos oliota ei ole ladattu, se ladataan tietokannasta ja lisätään sen yksilöivä viittaus Identity Mapiin. (Fowler, 2002.)

Usein on perusteltua toteuttaa Identity Map osaksi Unit of Work -suunnittelumalliin. Tällöin olioiden yksilöivän tunnisteen täytyy olla uniikkeja sekä samaa tyyppiä. Lisäksi Identity Mapin sisällön on oltava tyhjä transaktion alussa, ja se täytyy myös tyhjentää transaktion päättyessä.

6.4 Unit of Work (UoW)

Liiketoimintakerroksen suorittamien toimintojen vuoksi tehdyistä muutoksista on pidettävä listaa: mitä olioita tietokannasta on haettu, mitkä oliot ovat muuttuneet, mitä uusia olioita on luotu ja mitkä oliot on mahdollisesti poistettu tietokannasta.

Suunnittelumalli, joka kykenee toteuttamaan edellä mainitut toiminnat, on Unit of Work (UoW). Unit of Workin tehtävänä on ylläpitää listaa kaikista muutoksista, joita käsitellään liiketoimintaloogisten transaktioiden (business transaction) aikana. Liiketoimintalooginen transaktio tarkoittaa liiketoimintakerroksen toiminnallista kokonaisuutta, ei siis tietokantatransaktiota. (Broemmer, 2003.)

Liiketoimintaloogisen transaktion alussa luodaan UoW-olio. Liiketoimintakerroksessa suoritetaan eri toimintoja, ja erityyppisiä liiketoimintaoliota ladataan, muokataan, luodaan ja poistetaan. Tehdyt muutokset kirjataan UoW-olioon. Kun liiketoimintakerroksen suorittaman operaation vaatimat oliomuutokset on tehty, UoW toteuttaa vaadittujen muutosten päivittämisen tietokantaan. UoW aloittaa tietokantatransaktion, joka tekee tarvittavat samanaikaisuuden hallintaa liittyvät tarkastukset, päivittää muutokset tietokantaan ja lopettaa tietokantatransaktion. Tämä tarkoittaa, että UoW kutsuu olioiden Data Mappereita, jotka pystyvät suorittamaan tietokantakomennot. (Fowler, 2002.)

Tämän seurauksena sovelluskehittäjän ei tarvitse liiketoimintalogiikan toteutuksen yhteydessä tehdä suoria kutsuja tietokantaan eikä myöskään ylläpitää muutoslistaa. Sovelluskehittäjän tarvitsee ainoastaan rekisteröidä tarvittavat oliot UoW:lle, jotta suunnittelumalli pystyy valvomaan olioiden muutokset. Rekisteröinnissä on kaksi tapaa: Liiketoimintakerroksen toteuttava koodi toteuttaa rekisteröinnin (sovelluskehittäjän vastuulla) UoW:n, tai olio itse rekisteröityy UoW:n muutosten tai latauksen seurauksena. Jos rekisteröinti jätetään sovelluskehittäjän vastuulle, tällöin on mahdollista, että

rekisteröinti unohdetaan, joten on järkevämpää, että olio itse suorittaa rekisteröinnin. (Broemmer, 2003.)

Suunnittelumalli Identity Map voidaan yhdistää UoW:hen suhteellisen helposti, koska kaikki Identity Mapin vaatimat tiedot löytyvät UoW:stä, eli tieto siitä, mitkä oliot on ladattu fyysiseen muistiin liiketoimintatransaktion alusta.

Unit of Workia käytetään tavallisesti jonkin tapahtuman (event) seurauksena. Tällöin tapahtuma vastaanotetaan ja UoW luodaan ja tarvittava liiketoimintatransaktio käynnistetään. Sanoman vastaanottavan suunnittelumalli on facade. Facaden tarkoitus on etsiä sanomaa vastaava operaatio ja suorittaa sen vaatimat toiminnot. Tällöin UoW huolehtii olioiden muutoksista ja facaden suorituksen päättyessä hoitaa muutosten tallennuksen tietokantaan.

Flowerin UoW:n määritys on toteutettu kolmen listarakenteen mukaan, jotka listaavat liiketoimintatransaktion tapahtumat. Transaktion alussa kaikki listat ovat tyhjiä. Transaktion edetessä olioiden muutokset tallennetaan eri listoihin (uusi, muokattu, poistettava) ja lopuksi commitin yhteydessä kutsutaan luokkien Data Mappereita, jotka toteuttavat tietokantaoperaatiot. (Fowler, 2002.)

```
class UnitOfWork...
       private List newObjects = new ArrayList();
       private List dirtyObjects = new ArrayList();
       private List removedObjects = new ArrayList();
       public static abstract UnitOfWork instance();
```

Unit of Workin olion rekisteröinti voidaan toteuttaa seuraavasti:

```
class UnitOfWork...

public void registerNew(DomainObject obj) {
       Assert.notNull("id not null", obj.getId());
       Assert.isTrue("object not dirty", !dirtyObjects.contains(obj));
       Assert.isTrue("object not removed", !removedObjects.contains(obj));
       Assert.isTrue("object       not       already       registered       new",
       !newObjects.contains(obj));
       newObjects.add(obj);
}

public void registerDirty(DomainObject obj) {
       Assert.notNull("id not null", obj.getId());
       Assert.isTrue("object not removed", !removedObjects.contains(obj));
       if (!dirtyObjects.contains(obj) && !newObjects.contains(obj)) {
```

```
                dirtyObjects.add(obj);
        }
}

public void registerRemoved(DomainObject obj) {
        Assert.notNull("id not null", obj.getId());
        if (newObjects.remove(obj)) return;
        dirtyObjects.remove(obj);
        if (!removedObjects.contains(obj)) {
                removedObjects.add(obj);
        }
}

public void registerClean(DomainObject obj) {
        Assert.notNull("id not null", obj.getId());
}
```

Listojen käsittelyyn on toteutettu tarkistuksia. Esimerkiksi muokatuksi merkitty olio ei voi olla uusien olioiden listalla. Tässä tapauksessa registerCleanin toteutus on jätetty toteuttamatta, mutta sen paikalle on mahdollista toteuttaa Identity Map, joka pitäisi listaa tietokannasta ladatuista olioista.

Mikäli olio hoitaa rekisteröinnin itsenäisesti, jokaisen olion tilaa muuttavan operaation täytyy lisätä rekisteröinti:

```
class Person extends DomainObject {
        ...
        void setLastName(String alastName) {
                lastName = a.lastName;
                UnitOfWork.instance().registerDirty(this);
        }
        ...
```

Tämän seurauksena muokattu olio siirtyy muokattujen olioiden listalle.
Commit-operaatio hakee olioiden Data Maperit ja kutsuu jokaiselle oliolle listan mukaisen (insert, update, delete) tietokantaoperaation.

```
class UnitOfWork...
        public void commit() {
                insertNew();
                updateDirty();
                deleteRemoved();
        }
```

Seuraavassa on esitelty ainoastaan uusien olioiden lisäys:

```
private void insertNew() {
        for (Iterator objects = newObjects.iterator(); objects.hasNext();) {
                DomainObject obj = (DomainObject) objects.next();
                MapperRegistry.getMapper(obj.getClass()).insert(obj);
        }
}
```

Kaikkien UoW:tä käyttävien olioiden on rekisteröidyttävä MapperRegistryyn, ennen kuin UoW voi käyttää olioiden Data Mappereita.

6.5 Lazy Load

Olioiden lataamisessa tietokannasta on huomioitava olioiden väliset riippuvuussuhteet. Oliot voivat sisältää hyvin paljon keskinäisiä riippuvuuksia, jolloin voi syntyä olioiden välisten viitteiden muodostama laaja verkko. Ladatun olion käytössä joudutaan usein viittaamaan olioon viittaaviin toisiin olioihin, ja ongelmaksi tulee tietojen lataamisen laajuus.

Jos yhden objektin lataaminen aiheuttaa myös kaikkien olioon viitattujen olioiden luomisen ja lataamisen, tämä johtaa siihen, että viitteiden alaviitteet ym. on myös luotava. Tästä seuraa, että yhden olion luomisesta aiheutuu hallitsematon määrä tietokantakyselyitä, mikä lopulta johtaa siihen, että koko tietokanta on ladattu fyysiseen muistiin. Edellä mainittu toiminta ei ole järkevää, eikä tavallisesti lataamisen yhteydessä tarvitse ladata kaikkea olion viitteiden tietoa, etenkin jos tarvitaan vain murto-osaa ladatusta tiedosta. Kyseistä olioiden lataamismenetelmää kutsutaan innokkaaksi lataamiseksi (eager loading). (Corsaro et al. 2002.)

Latausten määrällä on suuri vaikutus järjestelmän suorituskykyyn, ja ongelmia syntyy yhtäaikaisten olioiden käsittelyn yhteydessä.

Ongelmaan on olemassa vaihtoehto lazy loading, joka hallitsee tietokannasta ladattavien olioiden suhteiden välisiä syvyyksiä. Lazy loading -menetelmässä keskeytetään olioiden lataus tiettyyn kohtaan olioiden välisessä hierarkiassa siten, että hierarkian ladattavat osat haetaan tietokannasta vasta tarvittaessa. Lazy load -mallin toteutukseen on olemassa neljä eri toteutustapaa: Lazy Initialization, Proxy, Value Holder ja Ghost. (Fowler, 2002.)

Lazy Initialization -toteutuksessa jokaisen olion attribuutin käsittelyn yhteydessä tehdään tarkistus, jossa tarkistetaan, onko olio alustettu. Jos oliota ei ole alustettu, olio alustetaan ennen kuin olion attribuuttia käsitellään. Menetelmässä on huomioitava, että olion

attribuuttien käsittely toteutetaan aina operaatioiden kautta. Lazy Initialization -toteutus soveltuu hyvin Active Record -mallin kanssa käytettäväksi.

Proxy on välittäjä, joka kontrolloi oliolle saapuvia pyyntöjä. Proxy näyttää ulospäin oliolta ja alustaa varsinaisen olion vasta silloin, kun olion palveluita käytetään ensimmäisen kerran. Alustuksen jälkeen proxy välittää saapuvan pyynnön alustetulle oliolle. Proxy-toteutus soveltuu hyvin Data Mapper -mallin kanssa käytettäväksi.

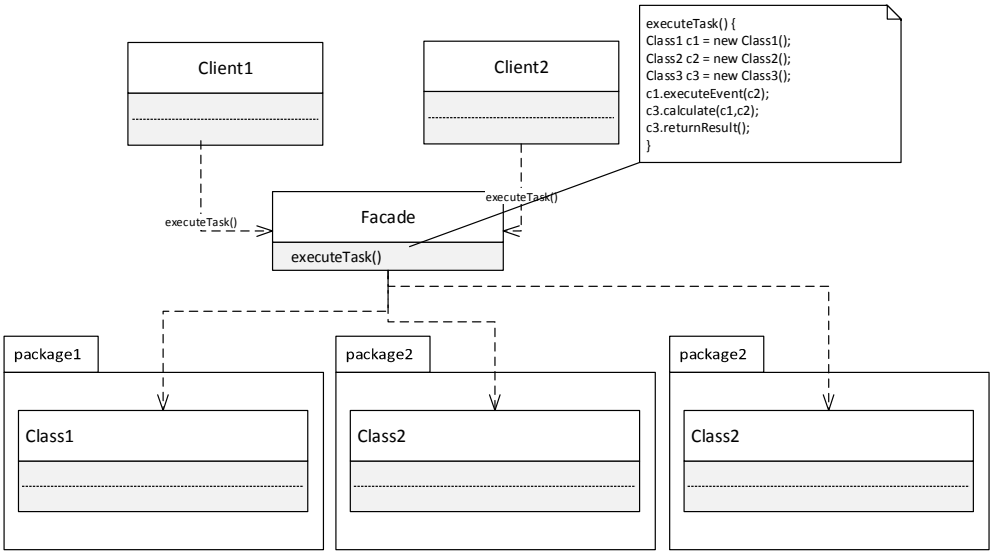
Ghost-toteutuksessa olio on vain osittain alustettu. Kun olio ladataan tietokannasta, sillä on vain yksilöivä tunniste eli ID, ja kun sen palveluita käytetään, olio lataa itsensä. Ghost-oliota voidaan pitää proxynä, joka toimii omana välittäjänään. (Fowler, 2002.)

6.6 Facade

Facade eli julkisivu on suunnittelumalli, jolla piilotetaan toteutuksen monimutkaisuus ja tarjotaan palveluiden käyttäjille näkymät (rajapinnat), joiden avulla käyttäjä pystyy hyödyntämään palveluita.

Oliosuuntautuneessa mallissa on suuret määrät pieniä luokkia, joilla on paljon pieniä operaatioita. Tämä johtaa hyvään kontrolloitavuuteen ja sovelluksen rakenteen ymmärtämiseen. Etäkutsut (remote call) ovat huomattavasti raskaampia kuin paikalliset kutsut, koska etäkutsujen tapauksissa joudutaan tekemään enemmän tarkastuksia ja kutsujen välittäminen on raskasta. Tämän vuoksi etäkutsujen määrää pyritään minimoimaan, ja käytettävien etäolioiden tapauksessa tarvitaan menetelmä, jolla vaatimukset toteutetaan. Facade-suunnittelumalli kykenee edellä mainittuun tehtävään. Facaden tehtävänä on toimia rajapintana, jonka asiakas näkee, sekä piilottaa monimutkainen toteutus. Facade itse ei sisällä toimintalogiikkaa. (Deepak et al., 2001.)

Facadea hyödynnetään tavallisesti silloin, kun tarvitaan yksikertainen rajapinta, joka mahdollista monimutkaisen järjestelmän käyttämisen. Käyttäjän ei siis tarvitse ymmärtää monimutkaisen järjestelmän toimintaa hyödyntäessään sen palveluita.



Kuva 10. Facade suorittaa asiakkaiden pyytämän toiminnon hyödyntämällä eri palveluita.

7 TUOTERAKENTEEN MÄÄRITYS

Teknologian nopea kehitys sekä kasvava kilpailu asettavat yritysten toiminnalle jatkuvasti uusia vaatimuksia. Nopea kehitys asettaa myös tuotteiden hallintaan vaatimuksia, sillä modernit tuotteet ovat monimutkaisia ja ne voivat koostua useista eri osista. Tietojärjestelmien suunnittelussa voidaan ottaa huomioon tuotteiden mallintamiseen tarkoitetut dynaamiset ja joustavat rakenteet. (Wu et. al 2012).

Tuotteen ominaisuudet voivat olla hyvin vaihtelevia, ja niitä ovat esimerkiksi tuotteen muoto, käytettävät raaka-aineet, valmistusprosessiin liittyvät tekijät, tuotteen kokoamiseen liittyvä tieto, tuotteen eri osien väliset suhteet, tuotteen ylläpitoon liittyvä tieto, tuotteen luotettavuuteen liittyvät tekijät sekä tuotteen turvallisuuteen liittyvät informaatio (Khan & Hussain, 2013).

Tässä luvussa määritetään tuoterakenteen runko hyödyntäen olioparadigmaa sekä suunnittelumalleja siten, että työlle asetetut tuoterakenteen perusominaisuudet on esitetty. Tuoterakenne muodostetaan olioparadigman avulla ja siinä hyödynnetään luokkien ja olioiden ominaisuuksia. Olioiden piirteet määritetään luokissa, ja jokaisella oliolla on yksikäsitteinen luokka (ks. kappale 2.2). Aluksi on muodostettava määrittelyn mukainen luokkamalli, jossa luokkien väliset suhteet voidaan muodostaa (ks. kappale 3.2).

Työssä määritettävän tuoterakenteen perusominaisuudet:

- Tuoterakenteen on tuettava erityyppisiä tuotteita, joiden ominaisuudet ovat täysin toisistaan poikkeavat. Lisäksi uudentyyppisiä tuotteita on pystyttävä tulevaisuudessa muodostamaan tuoterakenteeseen.
- Tuotteilla on oltava luokittelumenetelmät. Koska erityyppisiä tuotteita on mahdollista muodostaa tulevaisuudessa, luokitteluun on kiinnitettävä erityistä huomiota.
- Tuote voi muodostua useista eri komponenteista. Eri tyypin tuotteilla voi olla sisäisiä rakenteita.

Tässä työssä toteutettava tuoterakenne on yleisellä tasolla, eikä se toteuta suoraan reaalimaailmasta olevaa tuoterakennetta. Toteutettu tuoterakenne mahdollistaa erityyppisten tuotteiden muodostamisen olioparadigmalla, ja työn toteutuksesta voidaan johtaa reaalimaaliman tuotteita. Työn hyödyntämisen kannalta tuotteet on tärkeää määrittää mahdollisimman tarkasti ja tehdä niiden perusteella domain-malli.

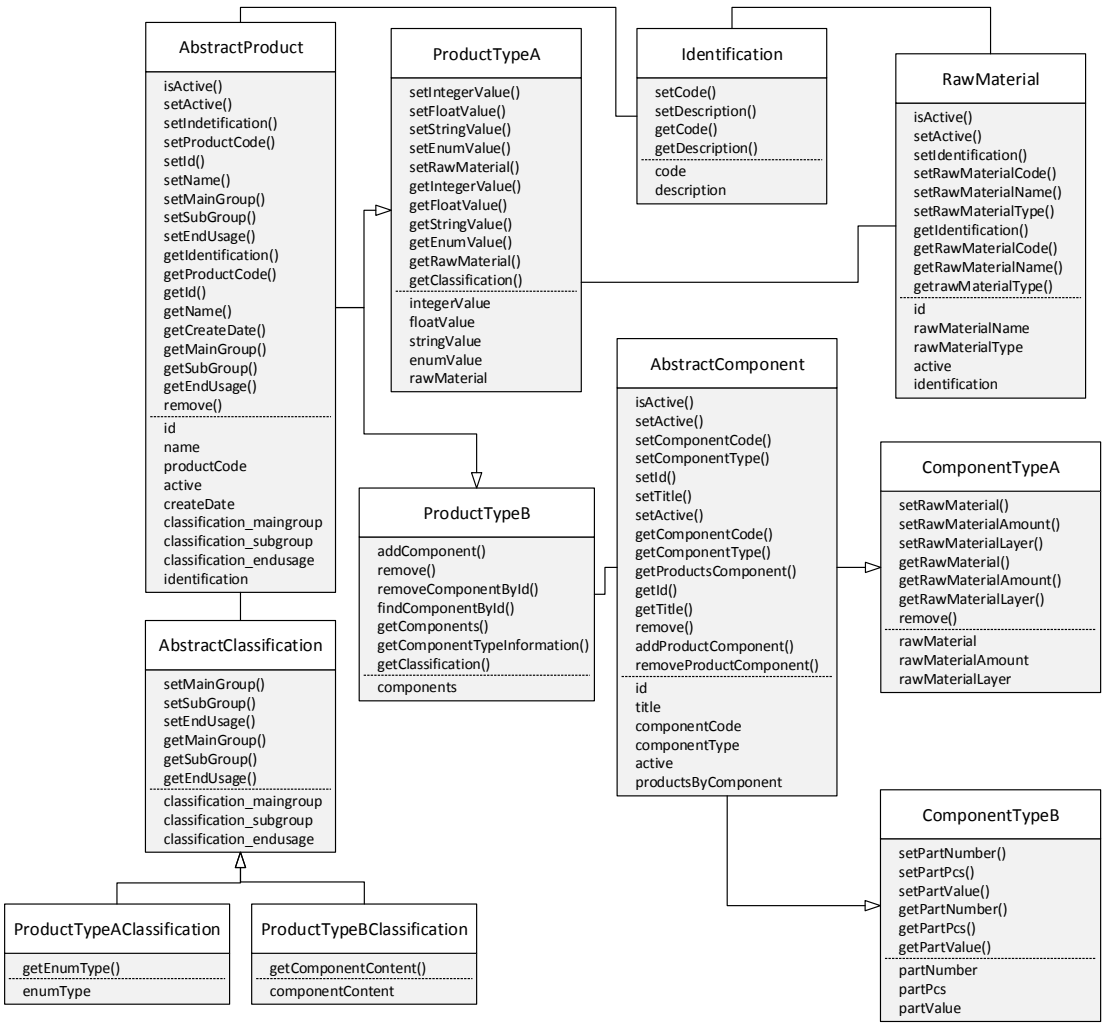
Työssä toteuteltiin J2EE:llä tuoterakenne, jonka lähdekoodi on kuvattu liitteissä. J2EE mahdollistaa pysyvien olioiden hallinnan ilman, että ohjelmoijan tarvitsee niitä toteuttaa.

7.1 Domain-malli

Domain-malli voidaan jakaa kahteen eri osa-alueeseen: yksinkertaiseen ja monimuotoiseen (Fowler, 2002). Työssä hyödynnetään monimuotoista mallia johtuen olioiden välisistä verkostoista ja niiden välisistä perintähierarkioista. Domain-mallissa on huomioitava myös luokkien lukumäärän kasvaminen, mikä lisää kompleksisuutta. Koska mallin vaatimuksena on uusien tuotetyyppien ja tuotekomponenttien lisääminen tulevaisuudessa, mikä vaatii uusia liiketoimintasääntöjä, voidaan monimutkaisemman mallin käyttöä pitää perusteltuna.

Tuoterakenne on tavallisesti yhtenä osakokonaisuutena toteuttamassa osaa yritysten ERP-järjestelmässä, joka muodostuu joukosta eri toiminnallisuuksia. Eri toiminnallisuudet voidaan jakaa eri kerroksiin. Toteutetussa Domain-mallissa tuoterakenteen liiketoimintalogiikka kapseloidaan yhteen kerrokseen. Tämän avulla Domain-malli pyritään pitämään mahdollisimman eriytettynä muista mahdollisista kerroksista. Näin mallin riippuvuudet pystytään pitämään mahdollisimman pieninä muiden sovelluskerrosten välillä.

Työssä määritetty tuoterakenne koostuu erityyppisistä tuotteista, joilla on erityyppiset rakenteet ja ominaisuudet. Tuoterakenteen malli on määritetty siten, että tuotteilla on kaikille yhteiset toteutukset sekä tuotetyypistä riippuvat omat toteutukset. Näin tuoterakennemalliin voidaan tulevaisuudessa lisätä uusia erityyppisiä tuoterakenteita niin, että olemassa oleviin tuotetyyppeihin ei tarvitse tehdä suuria muutoksia.



Kuva 11. Tuoterakenteen Domain-malli.

Työssä toteutettiin kaksi erityyppistä tuotetta: ProductTypeA ja ProductTypeB. ProductTypeA on yksinkertaisempi, ja tuoterakenne sisältää raaka-aineen ja joukon attribuutteja, jotka kuvaavat tuotteen ominaisuuksia. ProductTypeB on monimutkaisempi ja muodostuu joukosta erityyppisiä komponentteja. Komponentteja on olemassa erityyppisiä, ja niillä on myös yhteiset osat sekä komponentista riippuvat komponenttikohtaiset osat. Työssä toteutettiin kaksi erityyppistä komponenttia: ComponentTypeA ja ComponentTypeB.

ComponentTypeA sisältää raaka-aineen, raaka-aineen määrän sekä raaka-aineen kerroksen, joka kuvaa komponentin asettelua tuotteen sisällä. ComponentTypeB sisältää tiedon osanumerosta, osien lukumäärästä sekä osan mahdollisesta arvosta. Erityyppisiä komponentteja kuten myös tuotteita voidaan muodostaa tuoterakennemalliin tulevaisuudessa lisää.

Mallissa on otettu huomioon luokkien uudelleen käytettävyys ja raaka-aine on toteutettu siten, että raaka-aine voi olla suoraan joko tuotteen osana ja myös osana komponenttia. Lisäksi raaka-aineella ja tuotteen yläluokalla (AbstractProduct) on upotettuna Identification-luokka, joka sisältää valmistajaviitteet.

Tuotteiden luokittelu on mukana Domain-mallissa. Luokittelu jaetaan tuotteiden yhteisiin luokittelukriteereihin ja tuotteiden omiin tuotetyyppikohtaisiin kriteereihin. Tämä mahdollistaa uusien tuotetyyppikohtaisten luokittelujen lisäämisen uusien tuotetyyppien muodostamisen yhteydessä.

Domain-mallin toteuttama tuoterakenteen hallinta eli määritettyjen tuotetyyppien ja komponenttien käsittely on piilotettu facaden eli julkisivun suunnittelumallin sisälle. Facaden tarkoitus on piilottaa toteutuksen monimutkaisuus ja tarjota tuoterakenteen käyttöön rajapinta, jonka avulla tuoterakenteen käyttäjät pystyvät hyödyntämään tuoterakenteen palveluita.

7.2 Tuotteen rakenne

Erityyppisillä tuotteilla tarkoitetaan sitä, että mallinnettavien tuotteiden rakenne eroaa toisistaan merkittävästi. Tämän vuoksi on järkevää pyrkiä toteuttamaan useita erityyppisiä tuotteita ja toteuttamaan ne eri luokissa. Toinen vaihtoehto on, että useiden erityyppisten tuotteiden rakenne yritetään upottaa jonkin tuoteluokan sisään. Tämä vaihtoehto on huono, koska uuden tuotetyypin lisääminen merkitsee olemassa olevien tuotetyyppien mahdollista muokkaamista. Lisäksi sovelluksen ylläpito monimutkaistuu. Tämä vaihtoehto myös vaikeuttaa tuoterakenteeseen lisättävien uusien tuotetyyppien toteuttamista ja voi myös

aiheuttaa jo olemassa olevien tuotetyyppien rakenteisiin muutoksia ja toiminnallisuuden muuttumista.

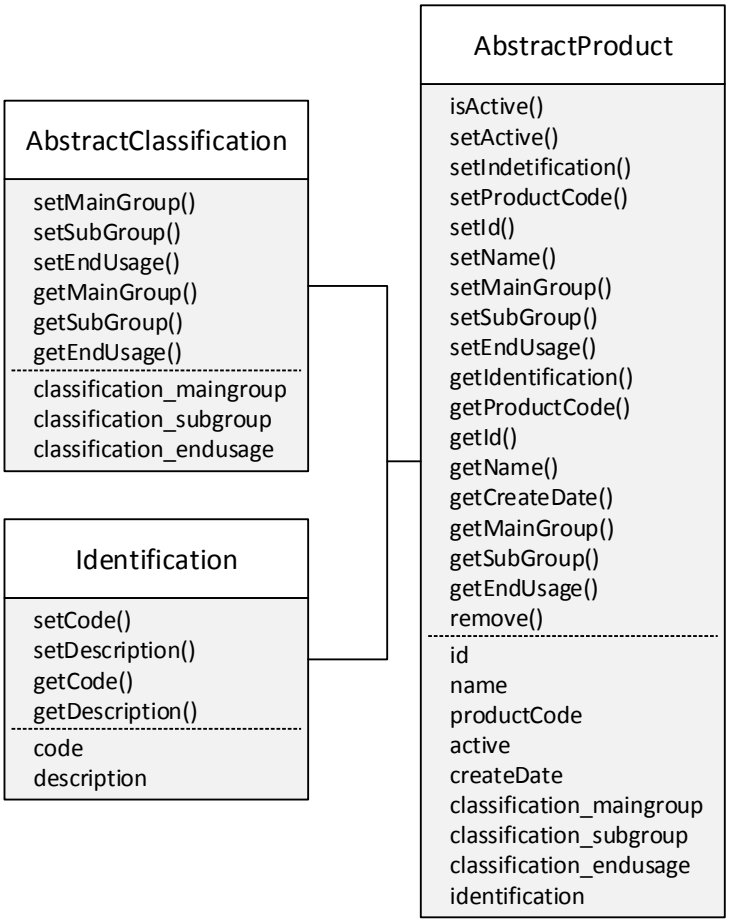
Tuotteen hallintaan liittyvät toteutukset pystytään kapseloimaan erityyppisten tuotetyyppiluokkien sisään. Toteutuksessa on huomioitava, että tulevaisuudessa joudutaan määrittelemään uudentyyppisiä tuotteita, joten tuoterakenteeseen on pystyttävä lisäämään uusia tuotetyyppejä niin, että tuotteiden rakenne tai toiminta ei muutu. Koska yhden operaation takana on useita eri toteutusvaihtoehtoja, on pystyttävä hyödyntämään myöhäistä sidontaa sekä olioiden periytymistä (ks. kappale 2.4).

Toteutuksessa määritetään abstrakteja yliluokkia (ks. kappale 2.5), jotka kapseloivat yhteiset toiminnallisuudet. Konkreettiset aliluokat perivät abstraktin yliluokan ja toteuttavat tarvittavat toiminnallisuudet. Aliluokissa voidaan perittyjen ominaisuuksien lisäksi toteuttaa uusia tietorakenteita ja operaatioita. Lisäksi on mahdollista myös korvata eli ylikirjoittaa jo perittyjä operaatioita. Eräs tärkeä ominaisuus on, että aliluokan oliot kuuluvat myös yliluokkaan, joten aliluokka on tyyppiyhteensopiva yliluokan kanssa.

Tuoterakenteen mukaiset tuotteet tallennetaan pysyvästi tietokantaan, josta niitä voidaan käsitellä. Tuoterakenteen tuotteet ovat siis pysyviä (persistence) olioita (ks. kappale 3.1), ja olioiden periytymisessä on käytetty Class Table Inheritance -menetelmää (ks. kappale 3.4.2), jossa käytetään yhtä taulua jokaista oliomallin luokkaa kohden. Nämä luokat voivat olla abstrakteja tai konkreettisia. Perintäsuhteet kuvataan tietokannassa vierasavainviittauksilla siten, että aliluokan taulussa on yliluokan taulun vierasavain.

7.2.1 Tuoterakenteen yliluokka AbstractProduct

Määritetään tuoterakenteen ylitaso, abstraktiyliluokka AbstractProduct, joka sisältää kaikkien tuoterakenteiden yhteiset osat eli yhteiset attribuutit sekä operaatiot. Yhteisiä ominaisuuksia tuotteille on tieto tuotteen aktiivisuudesta, tuotekoodi, tuotteen nimi, viittaukset tuotteen valmistajaan sekä yleiset tuotteen luokitteluun kuuluvat attribuutit, kuten tuotteen pääryhmä, tuotteen aliryhmä ja tuotteen loppukäyttökohde.



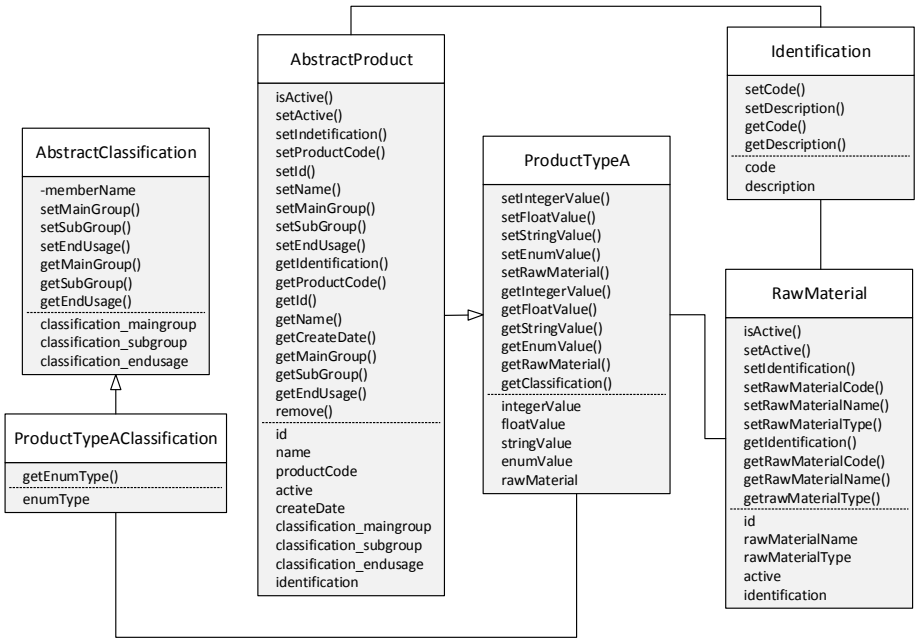
Kuva 12. AbstractProduct-, AbstractClassification- ja Identification-luokat.

AbstractProduct-luokka on abstrakti, joten siitä ei voi suoraan muodostaa oliota. Konkreettisen luokan on perittävä abstraktiluokka ja toteutettava abstraktin luokan ominaisuudet sekä konkreettisen luokan omat tarvittavat toiminnallisuudet. AbstractProduct-luokka sisältää viittauksen AbstractClassification-luokaan, joka toimii luokittelun ylätasona, ja sisältää luokittelun perusjaottelun (pääluokka, aliluokka ja loppukäyttökohde). AbstractProduct-luokan perivät konkreettiset luokat toteuttavat myös AbstractClassification-luokan, joka mahdollistaa sen, että erityyppiset tuoterakenteet voivat muodostaa omia luokittelumalleja. AbstractProduct-luokka sisältää viittauksen tuotteen valmistajaan, ja toteutuksessa on hyödynnetty Single Table Inheritance:a (ks. kappale 3.4.1), jossa luokka on upotettu AbstracrProduct-luokkaan. Samaa menetelmää on hyödynnetty myös raaka-aineiden käsittelyssä RawMaterial-luokan kanssa. Tuotteen tai raaka-aineen valmistajasta tiedetään ainoastaan valmistajan identifioiva tunnus.

Olioihin, jotka on muodostettu konkreettisesta luokasta ja perivät abstraktin yläluokan (AbstractProduct), voidaan kohdistaa erityyppisiä viittauksia. Viittaus voidaan muodostaa konkreettiseen luokkaan tai abstraktiin luokkaan. Huomioitavaa on, että hakutoiminnoilla voidaan palauttaa erityyppisiä tuotteita ja hakutoiminto palauttaa listan AbstractProduct-tyyppisiä viittauksia. Tällöin voidaan suorittaa tyypinmuunnokset hakutuloksen yksittäisille olioille ja voidaan käsitellä erityyppisiä tuoteolioita.

7.2.2 Tuote ProductTypeA

ProductTypeA on konkreettinen luokka, jonka tarkoitus on mallintaa olemassa olevaa tuotetta, jolla on omat luonteenomaiset piirteet. Luokka perii sekä toteuttaa AbstractProduct-luokan sekä ProductTypeA:n tuotteen ominaiset piirteet.



Kuva 13. ProductTypeA-, RawMaterial- ja ProductTypeClassification-luokat.

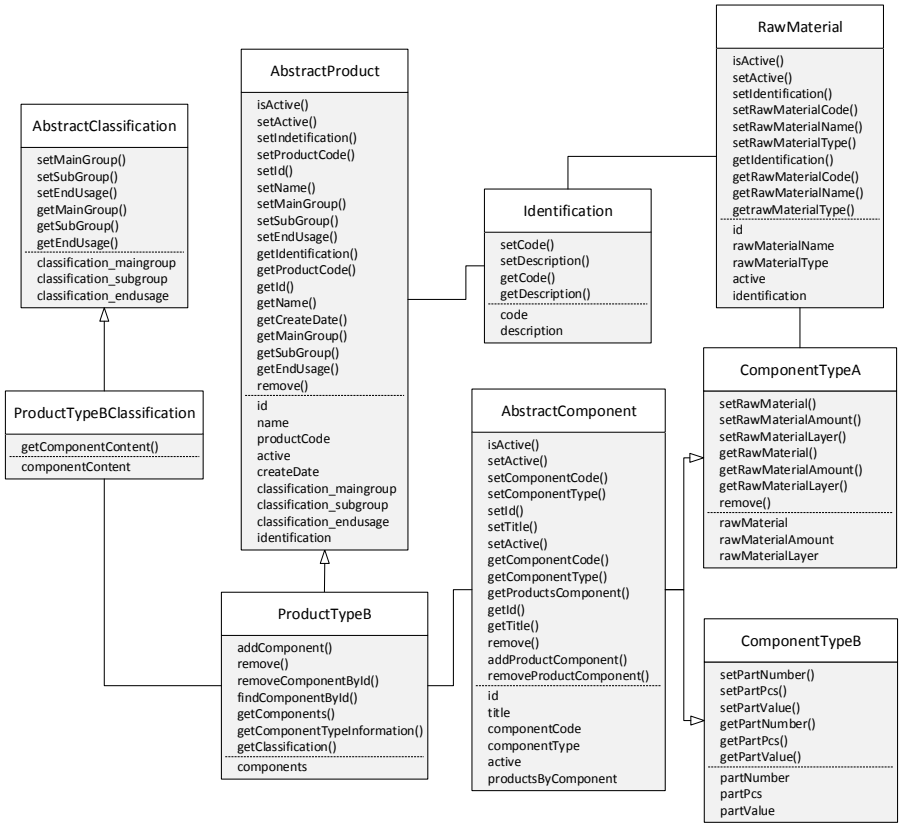
ProductTypeA koostuu raaka-aineesta sekä joukosta erityyppisiä attribuutteja. Attribuutit kuvaavat tuotteessa olevan raaka-aineen määrää, tuotteen tyyppiä sekä muita tuotteen luonteenomaisia piirteitä. ProductTypeA kuvaa tuotetta, jolla on riippuvuus raaka-

aineeseen siten, että tuotteella voi olla vain yksi raaka-aine (eli 1-n-suhde). Raaka-aineet ovat tietokannan omia pysyviä olioita, joita ProductTypeA:ta edustavat oliot sisältävät.

Luokittelun toteuttamiseksi on määritetty ProductTypeAClassification-luokka, joka perii AbstractClassification-luokan ja toteuttaa ProductTypeA:n luokittelun erityispiirteet. ProductTypeA:n luokittelun erikoispiirteenä on, että tuotteen luokittelu riippuu käytetystä raaka-aineesta. Luokittelun toteutus on toteutettu siten, että raaka-aineen lisäämisen tai poiston yhteydessä tarkastetaan asetettavan raaka-aineen tyyppi, jonka perusteella ProductTypeA:n luokittelu asetetaan automaattisesti. Asetettavan raaka-aineen tyyppi vaikuttaa ProductTypeA:n luokitteluun, eikä luokittelua tarvitse asettaa kutsumalla manuaalisesti luokitteluoperaatioita.

7.2.3 Tuote ProductTypeB

Toisen tuotetyypin muodostaa ProductTypeB-luokka. ProductTypeB-luokka perii myös luokan AbstractProduct sekä toteuttaa listarakenteen, johon voidaan säilöä tuoteolion sisältämät komponentit. Komponenteille on yhteinen yläluokka AbstractComponent, joka sisältää komponenttien yhteiset osat. ProductTypeB:llä on monen suhde moneen (n-m) AbstractComponent-luokan kanssa, joten komponentti voi kuulua useaan eri tuotteeseen ja myös komponentti itse tietää, mihin tuotteeseen se kuuluu.



Kuva 14. ProductTypeB-luokka sekä AbstractComponent-, ComponentTypeA- ja ComponentTypeB-luokat.

ProductTypeB:n tyypin olioiden lataamiseen tietokannasta on kiinnitettävä huomiota. Koska luokan rakenne on muodostettu Class Table Inheritance -menetelmää hyödyntämällä, luokan tiedot on tallennettu useisiin eri tietokantatauluihin (AbstractProduct, ProductTypeB, AbstractComponent, ComponentTypeA, ComponentTypeB ja RawMaterial). Tämän vuoksi on järkevää hyödyntää Lazy-load-suunnittelumallia.

Jos tietokannasta joudutaan hakemaan suuri joukko ProductTypeB:n edustamia olioita, on tärkeää tarkastella sitä, kannattaako ladata jokaisen olion kaikki komponentit ja komponenttien mahdollisesti sisältämät raaka-aineet. Tämä tarkoittaa sitä, ladataanko kaikki oliot ja niiden mahdollisesti sisältämät oliot käyttämällä innokasta lataamista (eager loading) vai voidaanko hyödyntää lazy-load-menetelmiä, esimerkiksi proxyä, jolloin ladataan ja alustetaan alioliot vasta sitten, kun aliolioita tarvitaan. Tällöin voidaan välttää mahdolliset ylimääräiset olioiden lataamiset tietokannasta.

ProductTypeB-luokka sisältää operaatiot, joilla voi muokata olion komponenttilistaa, eli luokka sisältää komponentin lisäämisen, poisto sekä erityyppiset hakutoiminnot. Lisäksi on huomioitava, että komponentin poisto ei poista tietokannassa pysyvästi säilöttyä komponenttia ja sen sisältämiä tietoja, vaan ainoastaan viittauksen, että komponentti kuuluu tietylle tuotteelle.

ProductTypeB:n luokittelumenetelmät eroavat ProductTypeA:n luokittelusta, ja tämän vuoksi on määritetty ProductTypeBClassification-luokka. ProductionTypeBClassification-luokka perii AbstractClassification-luokan ja toteuttaa ProductTypeB:n luokittelun erityispiirteet. ProductTypeB:n luokittelun erikoispiirteenä on, että luokittelu tapahtuu tuotteella olevien komponenttityyppien perusteella. Lisättäessä tai poistettaessa tuotteelta komponentteja luokka suorittaa automaattisesti luokittelun, joka perustuu tuotteella olevien komponenttien tyyppiin. Jos luokalla on pelkästään ComponentTypeA:ta sisältäviä komponentteja, luokitellaan tuote sisältämään ainoastaan ComponenTypeA:n komponentteja. Vastaavasti toimitaan myös ComponentTypeB-komponenttien kanssa. Jos tuote sisältää molempia komponentteja, luokitellaan tuote siten, että se sisältää molempia komponentteja.

7.3 Komponenttien rakenne

Tuoterakenteen tuotteet voivat muodostua useista erityyppisistä komponenteista. Työn komponentit muodostuvat abstraktista luokasta AbstractComponent, joka sisältää komponenttien yhteiset osat. Työssä toteutettiin komponentit ComponentTypeA ja ComponentTypeB.

7.3.1 Komponenttien yliluokka AbstractComponent

Tuotetyypin mukaan tuotteilla voi olla sisäisiä rakenteita. Sisäiset rakenteet on mallinnettu komponenteilla. Komponentteja voi olla useita erityyppisiä, ja ne pystytään toteuttamaan erityyppisillä rakenteilla. Toteutuksessa on määritetty, että kaikilla komponenteilla on

yhteiset rakenteet sekä komponenttityyppikohtaiset yksilölliset rakenteet. Komponenttien yhteiset rakenteet on kapseloitu AbstractComponent-luokan sisään, jolloin konkreettiset komponentit perivät AbstractComponent-luokan.

Kuva 15. AbstractComponent-luokka sekä luokan operaatiot ja attribuutit.

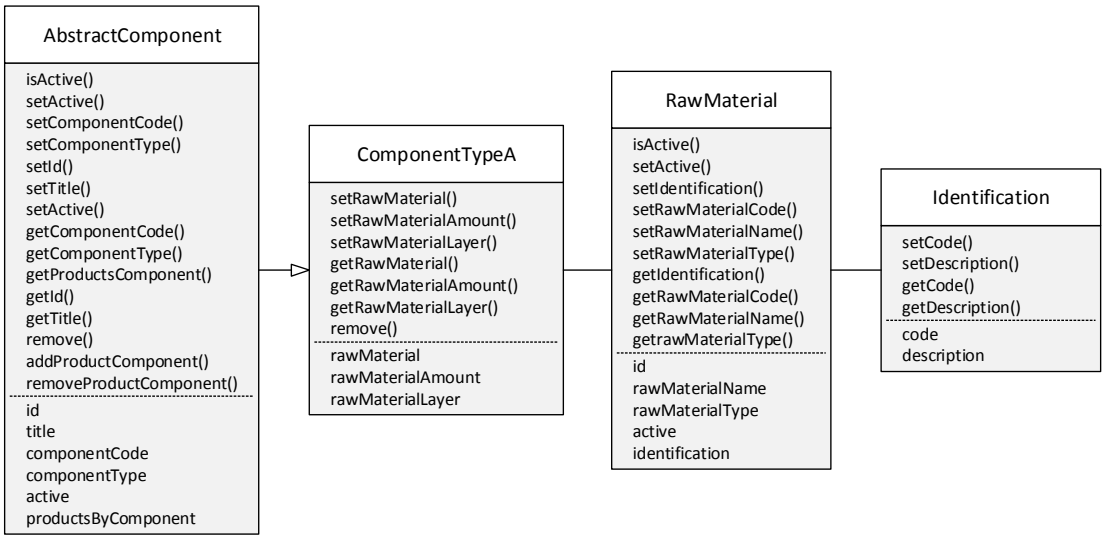
Eri komponenttityypeillä on yhteiset osat, jotka AbstractComponent-luokka sisältää: yksilöivä tunnus, otsikko, tyyppi, komponenttikoodi ja aktiivisuus. Lisäksi luokka sisältää hakutoiminnot, joilla komponentteja voidaan hakea. AbstractComponent-luokasta ei voida suoraan muodostaa oliota.

AbstractComponent-luokalla ja AbstractProduct-luokalla on monen suhde moneen (n-m). Tämä mahdollistaa sen, että konkreettinen komponentti tietää, mihin kaikkiin tuotteisiin se kuuluu. Koska tämä toiminnallisuus on tehty AbstractComponent-yliluokkaan, kaikki luokan perivät konkreettiset luokat saavat tämän toiminnallisuuden. Myös tulevaisuudessa toteutettavat uudet komponentit sisältävät tämän toiminnallisuuden.

AbstractComponent-luokalle on määritetty komponenttityyppi. Komponenttityypin avulla on mahdollista luokitella komponentit, joita voidaan hyödyntää komponenttien hakukriteereinä tai lopputuotteiden luokittelussa.

7.3.2 Komponentti ComponentTypeA

ComponentTypeA edustaa komponenttia, joka sisältää yhden raaka-aineen ja raaka-aineen käyttöön liittyviä attribuutteja, kuten raaka-aineen määrä ja raaka-aineen sijaintitiedon. Raaka-aineoliot muodostetaan RawMaterial-luokasta, ja raaka-aineet ovat pysyviä olioita tietokannassa. Komponentin ja raaka-aineen välinen suhde on yhden suhde moneen (1-n), eli komponentilla voi olla vain yksi raaka-aine, joka voidaan asettaa raaka-ainejoukosta.



Kuva 16. ComponentTypeA-luokka ja toteutukseen liittyvät luokat AbstractComponent, RawMaterial ja Identification.

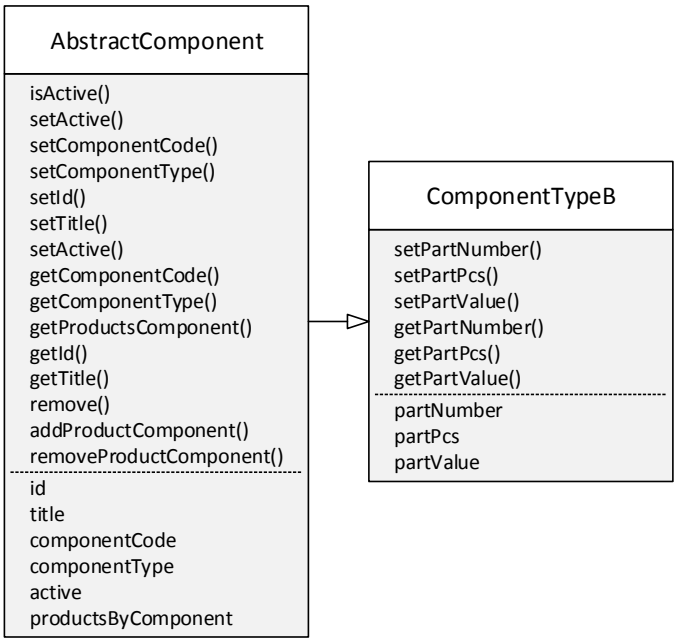
ComponentTypeA:n avulla voidaan muodostaa tuotteita, joissa on useita eri raaka-aineita useissa kerroksissa. Tällöin lopullinen tuote pystyy tarvittaessa laskemaan tuotteelle kuuluvat eri raaka-aineet ja niiden määrät. ComponentTypeA:ta voidaan muokata siten, että se toimii esimerkiksi tuotteen reseptinä.

ComponentTypeA sisältää vain yhden viittauksen raaka-aineolioon. Komponentin lataamisessa voidaan ottaa kantaa raaka-aineeseen eli siihen, täytyykö ComponentTypeA:n

mukana ladata muistiin myös komponentin sisältämä raaka-aine. Lazy loadin hyödyntäminen voi olla hyödyllistä silloin, kun haetaan suuri joukko tuotteita, joilla on paljon ComponentTypeA:n sisältämiä komponentteja, joita kaikkia ei välttämättä tarvitse käsitellä transaktion aikana.

7.3.3 Komponentti ComponentTypeB

Komponenttityyppi ComponentTypeB on yksinkertaisempi kuin ComponentTypeA. ComponentTypeB ei sisällä viittauksia toisiin pysyviin olioihin vain on enemmän tietue, joka sisältää joukon eri attribuutteja. ComponentTypeB:n tarkoitus on pystyä luetteloimaan se, mistä valmiista osista lopullinen tuote muodostuu. Komponenttityypin attribuutit ovat osanumero, osanumeroa vastaavan osien lukumäärä sekä osaan liittyvän arvo. ComponentTypeB:llä voidaan mallintaa esimerkiksi elektroniikkakomponentteja, joista lopullinen tuote muodostuu, tai pakkausmateriaaleja, johon lopullinen tuote pakataan.

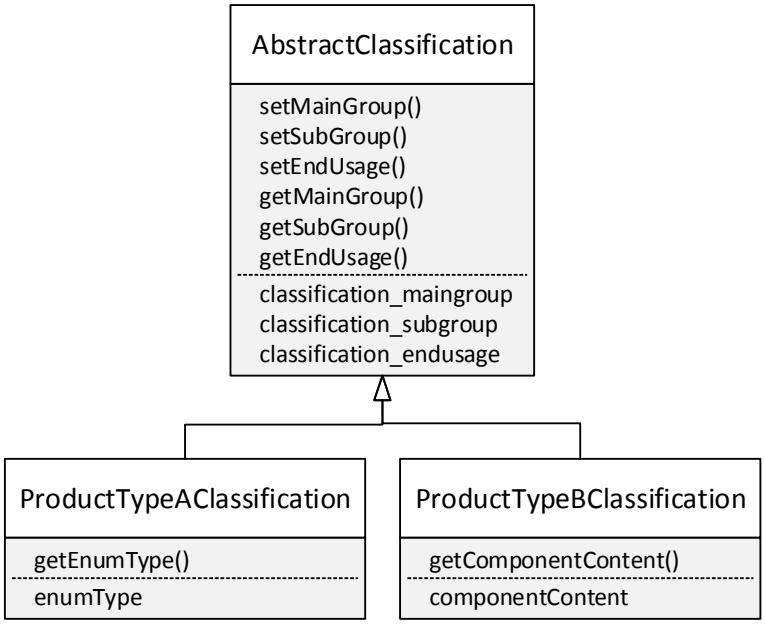


Kuva 17. ComponentTypeB-luokka ja abstraktiluokka AbstractComponent.

AbstractComponent-luokassa on määritetty komponenttityyppi, jota voidaan hyödyntää ComponentTypeB-komponentin tapauksessa. Komponentit voidaan luokitella edellisen esimerkin mukaan elektroniikkakomponentteihin tai pakkausmateriaaleihin.

7.4 Tuotteiden luokittelu

Suuren tuotejoukon käsittelyä selkeyttää ja nopeuttaa tuotteiden luokittelu. Tuotteiden luokittelun haasteena on uusien luokittelumenetelmien tarve, sillä uusia tuotetyyppejä tulee lisää tulevaisuudessa. Yksi mahdollisuus on, että kaikille tuotteille on yhteiset luokittelusäännöt ja jokainen tuotetyyppi toteuttaa tarvittaessa omat erikoisluokittelupiirteensä.



Kuva 18. AbstracClassification-abstraktiluokka sekä toteuttavat luokat ProductionTypeAClassification ja ProductionTypeBClassification.

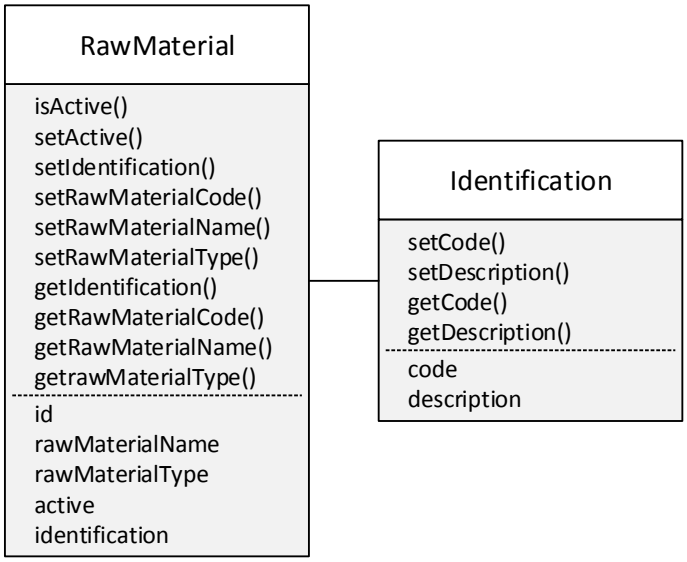
Työssä on muodostettu AbstractClassification-luokka, joka sisältää tuotepääryhmän, tuotealiryhmän sekä tuotteen loppukäyttökohteen. ProductTypeA:n erikoisluokittelupiirteet on määritetty luokassa ProductTypeAClassification ja ProductTypeB:n erikoispiirteet luokassa ProductTypeBClassification.

ProductTypeAClassification määrittelee luokitteluperusteen ProductTypeA:lle lisätyn raaka-ainetyypin perusteella. Kun ProductTypeA:lle asetetaan raaka-aine, asetetaan sille samalla automaattisesti myös luokittelu. Lisäksi jos raaka-aine poistetaan tuotteelta, vaihdetaan tuotteen luokittelu.

ProductTypeBClassification määrittelee luokitteluperusteen ProductTypeB:n sisältämien komponenttien perusteella. Luokitus muuttuu lisättävien ja poistettavien komponenttien perusteella. Luokittelu on toteutettu seuraavasti: Luokalla ei ole komponentteja, tuote sisältää ainoastaan ComponentTypeA:ta, tuote sisältää ainoastaan ComponentTypeB:tä tai tuote sisältää sekä ComponentTypeA:ta että ComponentTypeB:tä.

7.5 Raaka-aine RawMaterial

Raaka-aineet ovat pysyviä olioita, ja raaka-aineolioiden luominen tapahtuu tuoterakenteesta riippumatta. Raaka-aineet ovat yleisiä, ja niitä voidaan liittää tuote- ja komponenttityypeille. Raaka-aineluokalle on määritetty seuraavat ominaisuudet: raaka-aine tunnus, raaka-aineen nimi, raaka-aineen tyyppi sekä raaka-aineen valmistajaan liittyvät tunnistetiedot. Raaka-aineen tunnistetieto Identification on luokka, joka on upotettu RawMaterial-luokkaan. Identification-luokkaa käytetään myös AbstractProduct-luokan toteutuksessa. Raaka-aineelle on määritetty tyyppi, joka kuvaa raaka-ainetta, ja sen avulla raaka-aineita voidaan hakea tehokkaammin. Raaka-ainetyyppi voi kuvata raaka-aineesta esimerkiksi sen, onko raaka-aine kaasu, metalli tms. Raaka-aineelle ei ole työssä muodostettu yläluokkaa, mutta jos raaka-aineilla on hyvin erilaiset ominaisuudet, raaka-aineille kannattaa määrittää yläluokka. Yläluokkaan kapseloidaan raaka-aineiden yhteiset osat, kuten tunnus ja raaka-aineiden valmistajan tunnistetiedot. Raaka-aineen tyyppikohtaiseen aliluokkaan määritetään raaka-ainetyypin omat raaka-ainetta kuvaavat ominaisuudet, kuten yksikkö, raaka-aineen laatu ym.



Kuva 19. RawMaterial- ja Identification-luokka.

7.6 Facade

Tuoterakenteen käytön helpottamiseksi on hyödynnetty Facade-suunnittelumallia, jonka tarkoitus on piilottaa toteutuksen monimutkaisuus ja tarjota käyttäjälle rajapinta, jonka avulla käyttäjä pystyy hyödyntämään tuoterakenteen palveluita.

Toteutettu Facade-rajapinta toteuttaa peruspalvelut toteutetun tuoterakenteen käyttämiseen. Palveluihin kuuluu raaka-aineiden muodostaminen, raaka-aineiden hakeminen, ComponentTypeA:n lisääminen, ComponenetTypeB:n lisääminen, komponenttien hakutoiminnot, ProductTypeA:n lisääminen, ProductTypeA:n raaka-aineen hallinta, ProductTypeB:n lisääminen, komponenttien lisääminen sekä poistaminen ProductTypeB:ltä, komponenttien hakeminen ProductTypeB:ltä sekä suuri joukko eri pysyvien komponenttien hakuun liittyviä palveluita.

```
public interface Facade {

// Get DAO-objects
public abstract AbstractComponentDao getAbstractComponentDao();
public abstract AbstractProductDao getAbstractProductDao();
public abstract ProductTypeBDao getProductTypeBDao();
public abstract RawMaterialDao getRawMaterialDao();

//Set DAO-objects
```

```
public abstract void setAbstractComponentDao(final AbstractComponentDao
      abstractComponentDao);
public abstract void setAbstractProductDao(final AbstractProductDao
      abstractProductDao);
public abstract void setProductTypeBDao(final ProductTypeBDao productTypeBDao);
public abstract void setRawMaterialDao(final RawMaterialDao rawMaterialDao);

// ProductTypeA handling
public abstract ProductTypeA createProductTypeA(final String name, final Integer
      productCode, final Identification id, final Boolean active);
public abstract ProductTypeA updateProductTypeARawMaterial(final ProductTypeA
      productTypeA, final RawMaterial rawMaterial);

// ProductTypeB handling
public abstract ProductTypeB createProductTypeB(final String name, final Integer
      productCode, final Identification id, final Boolean active);
public abstract ProductTypeB addAbstractComponent(final ProductTypeB product,
      final AbstractComponent component);
public abstract ProductTypeB removeComponentById(final ProductTypeB prod, final
      AbstractComponent comp);

//ComponentTypeA handling
public abstract ComponentTypeA createComponentTypeA(final String title, final
      Integer componentCode, final Boolean active, final RawMaterial
      rawMaterial, final Float amount, final Integer layer);

//ComponentTypeB handling
public abstract ComponentTypeB createComponentTypeB(final String title, final
      Integer componentCode, final Boolean active, final Integer partNumber,
      final Float partValue, final Integer partPcs);

//AbstractProduct finders
public abstract AbstractProduct findAbstractProductByProductCode(Integer
      productCode);
public abstract AbstractProduct findAbstractPoductById(final long id);
public abstract AbstractProduct updateProduct(final AbstractProduct
      abstractProduct);
public abstract Set<AbstractProduct> findProductByComponent(final Long id);
public abstract List<AbstractProduct> findByActiveProduct(final Boolean active);
public abstract List<AbstractProduct> findByProductName(final String name);
public abstract AbstractComponent findAbstractComponentByComponentCode(Integer
      componentCode);

//AbstractComponent finder
public abstract List<AbstractComponent> findByComponentTitle(final String title);
//RawMaterial handling and finders
public abstract RawMaterial createRawMaterial(final String name, final Integer
      code, final RawMaterialType type,final Identification id);
public abstract RawMaterial findByRawMaterialCode(final Integer code);
public abstract List<RawMaterial> findRawMaterialByActiveStatus(final Boolean
      active);
public abstract List<RawMaterial> findRawMaterialByName(final String name);

}
```

Tuoterakenteen Facaden toteutuksessa rajapinta on pyritty toteuttamaan siten, että käyttäjän ei tarvitse tuntea pysyvien olioiden yksilöiviä tunnuksia, vaan Facaden käytössä riittää se, että tuntee pysyvien olioiden ominaisuuksia, kuten tuotekoodi, tuotteen aktiivisuus, komponenttikoodi tai tuotteen nimi.

8 MALLIN HYÖDYNTÄMINEN

Tässä kappaleessa käsitellään määritetyn tuoterakenteen hyödyntämistä reaalimaailmassa esiintyvien tuotteiden hallintaan.

Tuoterakenne on tavallisesti yhtenä osana suurempaa ERP-järjestelmää, ja se on kapseloitu ERP-järjestelmästä niin, että tuoterakenteella ei ole riippuvuuksia toisiin ERP-järjestelmän osiin. Vanhojen legacy-järjestelmien toteutuksissa voi tuoterakenteessa esiintyä riippuvuussuhteita järjestelmän osien suhteen, mikä voi hankaloittaa tuoterakenteen uudistamista. Lisäksi vanhat legacy-järjestelmät voivat olla monimutkaisia niiden puutteellisen määrittelyn ja kontrolloimattoman ylläpidon vuoksi (Pérez-Castillo et al., 2011).

Liiketoiminnan uudet vaatimukset sekä lisääntyvä informaation määrä aiheuttavat jo ennestään kompleksisille legacy-järjestelmille lisää kompleksisuutta. Toisaalta järjestelmien eroosioilmiö (software erosion phenomenon) vaikuttaa myös legacy-järjestelmiin, koska niiden toteutus on tehty kauan sitten ja niiden käyttämä teknologia on vanhentunutta. Vanhojen järjestelmien uusiminen tulee järjestelmän elinkaaren lopussa vastaan. (Pérez-Castillo et al., 2011.)

Olemassa oleva legacy-järjestelmä voidaan korvata alusta asti siten, että uusi järjestelmä suunnitellaan ja toteutetaan tyhjältä pöydältä, mutta tämä on erittäin kallista. Lisäksi legacy-järjestelmiin on aikojen saatossa upotettu suuret määrät liiketoimintatietoa, ja järjestelmän korvaaminen kokonaan uudella voi hävittää kertyneen liiketoimintatiedon. (Pérez-Castillo et al., 2011.)

Pérez-Castillon et. al. mukaan eroosioilmiön torjumisen ratkaisuna on kehittää järjestelmiä hylkäämättä nykyisiä. Tämä mahdollistaa legacy-järjestelmissä olleen liiketoimintalogiikan säilyttämisen ja kustannusten pienentämisen.

Uudelleensuunnittelu (reengineering) on pääasiallinen menetelmä, jolla hallitaan legacy-järjestelmien evolutiivista ylläpitoa. Uudelleensuunnitteluprosessi koostuu legacy-järjestelmän tutkimisesta sekä muuttamisesta siten, että järjestelmä toteutetaan uusilla menetelmillä ja uudessa muodossa. (Pérez-Castillo et al., 2011.)

Legacy-järjestelmien uudelleen suunnittelu on kuitenkin haastavaa, ja useat uudelleensuunnitteluprojekti ovat epäonnistuneet. Yhtenä syynä ovat legacy-järjestelmien monimutkaisuus sekä järjestelmien komponenttien ja niiden välisten riippuvuuksien tunnistaminen. (Pérez-Castillo et al., 2011.)

Työssä toteutetun tuoterakenteen hyödyntäminen voidaan aloittaa olemassa olevan legacy-järjestelmän tuoterakenteen nykyisen toteutuksen tarkastelulla. Legacy-järjestelmän ominaisuudet ja toiminnot kartoitetaan ja mallinnetaan. Lisäksi legacy-järjestelmän tuoterakenteen eri osien väliset suhteet ja riippuvuudet määritetään ja mallinnetaan. Myös legacy-järjestelmän puutteet ja rajoitteet tunnistetaan. Näiden tietojen avulla voidaan uudelleensuunnittelun avulla aloittaa uuden Domain-mallin määrittely uudelle tuoterakenteelle. Domain-mallin suunnittelussa on myös otettava huomioon tuoterakenteen uudet vaatimukset sekä pyrittävä tekemään mallista mahdollisimman joustava. Näin mallia laajentamalla voidaan toteuttaa tulevaisuuden vaatimuksia niin, että uuden tuoterakenteen perusrakenteeseen ei tarvitse tehdä suuria muutoksia. Työssä määritetyn tuoterakenteen toteutusta voidaan peilata legacy-järjestelmän tuoterakenteeseen ja tarvittaessa hyödyntää soveltuvin osin rakenteen ominaisuuksia ja määrittää tarvittavat tuotteet sekä niille tuotteita kuvaavat sisäiset rakenteet, attribuutit ja operaatiot. Työn tuoterakenteen tuotetyyppien yhteisten osien (AbstractProduct) sekä eri tuotetyyppien omia toteutuksia on mahdollista hyödyntää. Sama koskee myös tuotetyyppien mahdollisesti sisältämiä komponentteja eli niiden yhteisiä osia (AbstractComponent) sekä komponenttikohtaisia toteutustapoja.

Toteutuksessa on järkevää käyttää olemassa olevaa teknologiaa, esimerkiksi J2EE:tä, joka toteuttaa pysyvät oliot, sisältää olioiden käsittelymenetelmät ja huolehtii transaktioista sekä olioiden siirrosta fyysisen muistin ja tietokannan välillä. Näin pystytään keskittymään tuoterakenteen muodostamiseen niin, että olioiden hallinnan vaatimia toiminnallisuuksia ei tarvitse itse toteuttaa.

Vanhan ERP-järjestelmän muidenkin osien, ei pelkästään tuoterakenteen, uudistamisessa voidaan hyödyntää oliokeskeistä lähestymistä. Vastaavalla tavalla voidaan mallintaa ERP-järjestelmän osien, kuten, laskutuksen, varaston ym., vaatimukset ja tehdä niiden avulla

omat Domain-mallit. Olioperusteisuudella voidaan toteuttaa vastaavasti muiden osien määritykset. ERP-järjestelmän eri toiminnallisuudet (tuoterakenne, myynti, varasto ym.) kannattaa rajata siten, että niiden väliset sisäiset riippuvuudet voidaan minimoida. Tällöin eri toiminnallisuuksista voidaan muodostaa palveluita.

Yksi vaihtoehto on hyödyntää palvelukeskeistä arkkitehtuuria (Service-oriented architechture, SOA). Legacy-järjestelmien uudelleensuunnittelun avulla voidaan olemassa olevan ERP-sovelluksen eri osa-alueet toteuttaa omiksi kokonaisuuksiksi (komponentit, palvelut), jotka vastaavat tietyistä liiketoimintakokonaisuuksista. SOA mahdollistaa tiedonvaihdon eri ohjelmistopalveluiden välillä ja tukee uudelleenkäytettävyyttä erottamalla palvelun rajapinnan komponentin toteuttamisesta. SOA:a hyödynnetään laajasti legacy-järjestelmien uudistamisessa. SOA:n ominaisuuksia ovat löyhä sitominen (loose coupling), ketteryys (agility), joustavuus (exibility), uudelleenkäytettävyys (reusability), autonomia (reusability), tilattomuus (statelessness), löydettävyys (discoverability) ja kustannusten pienentäminen. (Almonaies et al., 2010.)

Tuoterakenteesta, kuten muistakin ERP-järjestelmän osista, voidaan muodostaa palveluita, joita SOA:a hyödyntämällä voidaan toteutettua legacy-järjestelmän uudistus. Tuoterakenteen facade pystyisi toimimaan tuoterakennepalvelun rajapintana, jonka toteutus perustuu tuoterakennepalveluun.

9 TULOKSET

Työn tarkoituksena oli selvittää vanhan toiminnanohjausjärjestelmän tuoterakenteen uudistamista. Uudistamisen taustalla oli tarve pystyä mallintamaan tarvittaessa monimuotoisia tuotteita, joilla voi olla sisäisiä rakenteita. Uudistamisen lähtökohdaksi otettiin olioperustainen lähestymistapa ja työn toteutuksessa pyrittiin hyödyntämään oliokeskeisyyden tärkeimpiä ominaisuuksia.

Työn tutkimuskysymys oli: ”Miten tuotannonohjausjärjestelmän tuoterakenteen runko voidaan määrittää olioparadigman avulla?”

Olioperusteisuus vastaa ihmisen tapaa hahmottaa ympärillä olevaa maailmaa. Olioparadigma mahdollistaa systemaattisen koko ohjelmiston elinkaaren käsittävän menetelmän, joka perustuu ohjelmiston kuvaamiseen joukkona keskeisessä vuorovaikutuksessa oleviin olioihin.

Olemassa olevan legacy-tuotannonohjausjärjestelmän tuoterakenteen määrittelyyn on kiinnitettävä huomiota ja ympäristö on pystyttävä mallintamaan ja kuvaamaan. Legacy-järjestelmän keskeiset osat, niiden väliset riippuvuudet, suhteet tuotannonohjausjärjestelmän muihin osa-alueisiin täytyy selvittää ja mallintaa. Selvityksen perusteella voidaan laatia luokkamalli, joka kuvaa tuoterakennetta. Uusia tuoterakenteen ominaisuuksia on verrattava legacy-tuoterakenteen ominaisuuksiin ja muodostettava Domain-malli, joka määrittää tuoterakenteen sekä tuoterakenteen osien välisen suhteen. Määrittämällä tuoterakenne luokkamallina ja muodostamalla luokkia hyödyntäen oliokeskeisyyden menetelmiä, kuten luokkien perintää ja myöhäistä sitomista, voidaan rakentaa tuoterakennemalli, jota voidaan laajentaa tulevaisuudessa. Lisäksi tuoterakenteen luokille voidaan lisätä reaalimaalimaa mallintavia attribuutteja, jotka voivat olla luokkia, joilla on omat rakenteet ja riippuvuudet. Tuoterakenne voidaan toteuttaa myös hyödyntämällä suunnittelumalleja, jotka tarjoavat ratkaisun tai toteutuksen tiettyyn ongelmaan. Tuoterakenteen käsittely ja tallentaminen pysyvästi relaatiotietokantaan voidaan toteuttaa suunnittelumalleilla. Olioperusteisia malleja on käytetty myös

sovelluspalvelimissa, jotka toteuttavat pysyvyyden ja pysyvyyden hallintaan tarvittavat menetelmät.

Työn alakysymys oli: ”Miten määritetty tuoterakenne voidaan ottaa käyttöön yrityksen nykyiseen tuotannonohjausjärjestelmään?”

Tuoterakenne on ERP-järjestelmän yksi osa, ja tuoterakennetta käyttää ja hyödyntää ERP-järjestelmän eri osat, kuten varasto ja myynti. Tuoterakenne on toteutettava siten, että sillä ei ole sisäisiä riippuvuuksia muihin ERP-järjestelmän osiin. Työssä muodostettiin julkisivu eli facade, joka toimii tuoterakenteen käytön rajapintana. Julkisivu sisältää operaatiot, joilla eri tuotetyyppejä voidaan esimerkiksi luoda, hakea ja muokata. Muodostettavasta tuoterakenteesta on järkevää muodostaa palvelu, joka on kaikkien tuoterakennetta tarvitsevien muiden ohjelmisto-osien käytettävissä. Vanhan legacy-järjestelmän osista kannattaa rakentaa palveluita, jotka toteuttavat ERP-järjestelmän toiminnallisuuden, jossa tuoterakenne on yhtenä palveluna. Yrityksen ERP:n eri osia voidaan suunnitella, määrittää ja toteuttaa yhtäaikaisesti, jolloin sovelluskehitys nopeutuu ja järjestelmän eri osien väliset riippuvuudet voidaan havaita helpommin. Lisäksi palvelukeskeisyys helpottaa järjestelmän eri osien testaamista.

10 JOHTOPÄÄTÖKSET

Tietojärjestelmien toteutuksessa järjestelmän tuottavuuteen, siirrettävyyteen, yhteensopivuuteen sekä elinkaareen on otettava kantaa, jolloin järjestelmälle asetetut vaatimukset voidaan toteuttaa kustannustehokkaasti sekä järkevillä resursseilla. Lisäksi olemassa oleviin järjestelmiin kohdistuu yritysten liiketoiminnasta johtuvia muutosvaatimuksia, joiden toteuttamiseen järjestelmän koko elinkaaren aikana on pystyttävä vastaamaan. Yhtenäinen tietomalli sekä järkevästi toteutettu järjestelmäarkkitehtuuri pystyy vastamaan näihin tavoitteisiin. Edellä mainittuihin vaatimuksiin pystytään olioperusteisella järjestelmäkehityksellä tuomaan ratkaisuja.

Työn taustalla oli tarkoitus uudistaa vanhan legacy-tuotannonohjausjärjestelmän tuoterakenne. Tuoterakenteen määrityksessä oli huomioitavaa, että tulevaisuudessa tuoterakenteeseen voidaan joutua lisäämään uusia tuotetyyppejä, jolloin olemassa oleviin tuotetyyppeihin ei jouduta tekemään suuria muutoksia. Olioperusteisuuden ja suunnittelumallien hyödyntäminen tuoterakenteen uudistamisessa on perusteltua. Tuoterakenteen uudet vaatimukset on pystyttävä huomioimaan tulevaisuudessa, koska uudet vaatimukset voivat edellyttää muutoksia koko tuoterakenteeseen. Muutoksiin tarvitaan tekniikoita, joilla tuoterakenteen yleiset osat saadaan mahdollisimman riippumattomiksi uusista vaatimuksista. Näin yleisrakenne pystytään saamaan mahdollisimman riippumattomaksi uusista muutosvaatimuksista. Oliopohjainen lähestymistapa mahdollistaa määritettävän sovelluksen käsitemaailman mallintamisen, eivätkä toiminnalliset muutokset yleensä aiheuta suuria muutoksia koko ohjelmiston rakenteeseen. Oletuksena on, että muutokset eivät vaikuta sovelluksen käsitemaailmaan.

Legacy-järjestelmän osista kannattaa muodostaa kokonaisia palveluita, joilla ei ole suoria riippuvuuksia toisiinsa, ja palveluita käytetään rajapintojen kautta. Palveluiden avulla pystytään eristämään eri ERP:t omiksi kokonaisuuksiksi, joiden hallinta on selkeämpää kuin yhden suuren järjestelmän hallinta.

Tuoterakenteen erityyppisten tuotteiden tallentamiseen pysyvästi tietokantaan on myös kiinnitettävä huomiota. Olemassa olevia sovelluspalvelimia kannattaa hyödyntää, sillä ne

toteuttavat olioiden pysyvyyden niin, että niiden toteuttamiseen ei tarvitse käyttää aikaa eikä resursseja. Sovelluspalvelintoteutukset ovat tehokkaita, ja ne helpottavat sovelluksen toteutusta merkittävästi.

LÄHTEET

Agrawal, R., & Gehani, N. H. (1990). Rationale for the design of persistence and query processing facilities in the database programming language O++. *Hull et al.[104]*, 25-40.

Ambler, S. W. (2000). Mapping objects to relational databases: What you need to know and why [verkkodokumentti]. [viitattu 25.9.2014]. Saatavilla http://fi.ort.edu.uy/innovaportal/file/2032/1/mappingobjectstorelationaldatabases.pdf

Bauer, C., & King, G. (2005). Hibernate in action.

Brown, K. (2003). Enterprise java programming with IBM webSphere. Addison-Wesley Professional.

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

Broemmer, D. (2003). J2EE best practices: Java design patterns, automation, and performance (Vol. 8). John Wiley & Sons.

Corsaro, A., Schmidt, D. C., Klefstad, R., & O'Ryan, C. (2002, September). Virtual component: a design pattern for memory-constrained embedded applications. In Proceedings of the 9th Annual Conference on the Pattern Languages of Programs.

Deepak, A., Crupi, J., & Malks, D. (2001). Core J2EE Patterns: Best Practices and Design Strategies. Sun Microsystems. Palo Alto.

Duke, R., Rose, G., & Smith, G. (1995). Object-Z: A specification language advocated for the description of standards. Computer Standards & Interfaces, *17*(5), 511-533.

Fowler, M. (2002). Patterns of enterprise application architecture. Addison-Wesley Longman Publishing Co., Inc.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). Design patterns: elements of reusable object-oriented software. Pearson Education.

Jendrock, E., Ball J., Carson D., Evans I., Fordin S. & Haase K. (2007). Java EE 5 Tutorial, Sun Microsystems [verkkodokumentti]. [viitattu 03.10.2014]. Saatavilla http://java.sun.com/javaee/5/docs/tutorial/doc/

Koskimies, K. (1997). *Pieni oliokirja*. Suomen atk-kustannus.

Khan, W. & Hussain, A. (2013). Automatic object-oriented coding facility for product life cycle management of discrete products. Internal Journal of Computer Integrated Manufacturing, 2014. Vol. 27, No 1, 60-84.

Larman, C. (2004). Applying UML and Design Patterns. An introduction to object-oriented analysis.

Leppänen, M. (2008). Tietokannat ja tiedonhallinnan perusteet Osa 2 [verkkodokumentti]. [viitattu 29.9.2014]. Saatavilla http://users.jyu.fi/~harjma/ITKA/Luentomoniste/Osa_5.pdf

Luoma, O. (2007). Tietokannat I [verkkodokumentti]. [viitattu 25.9.2014]. Saatavilla http://staff.cs.utu.fi/~olallu/TKAN1/Osa1.pdf

Nekrestyanov, I., Novikov, B., & Pavlova, E. (1998, January). Designing persistence for real-time distributed object systems. In Advances in Databases and Information Systems (pp. 248-259). Springer Berlin Heidelberg.

Pérez-Castillo, R., De Guzman, I. G. R., & Piattini, M. (2011). Knowledge Discovery Metamodel-ISO/IEC 19506: A standard to modernize legacy systems. Computer Standards & Interfaces, 33(6), 519-532.

Laine, H. & Paakki, J. (2001). Ohjelmistotuotanto, Ohjelmistosuunnittelu 1. Helsingin yliopisto, Tietojenkäsittelytieteen laitos [verkkodokumentti]. [viitattu 28.10.2014]. Saatavilla http://www.cs.helsinki.fi/u/paakki/ohtuk03-luento9.pdf

Luoma, O. (2007). Tietokannat I [verkkodokumentti]. [viitattu 25.9.2014]. Saatavilla http://staff.cs.utu.fi/~olallu/TKAN1/Osa1.pdf

Mak, G., & Guruzu, S. (2010). Hibernate Recipes: A Problem-Solution Approach. Apress.

O'Neil, E. J. (2008, June). Object/relational mapping 2008: hibernate and the entity data model (edm). In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (pp. 1351-1356). ACM.

Silberschatz A., Korth H. F. & Sudarshan S. (2010) Database System Concepts.
Sixth Edition. McGraw-Hill

Smed, J., Hakonen, H. & Raita, T. (2007). Sopimuspohjainen olio-ohjelmointi Java-kielellä.

Wu, J., Poppa, M., Leu X & Liu, F. (2012) Integrated function structure and object-oriented design framework Missouri University of Science and Technology, Rolla, MO 65409, USA

LIITE 1. AbstractProduct-luokka

```
package entity;

import java.io.Serializable;
import java.util.Date;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Embedded;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.OneToOne;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;
import javax.persistence.TableGenerator;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;

import classification.AbstractClassification;
import classification.AbstractClassification.Classification_EndUsage;
import classification.AbstractClassification.Classification_MainGroup;
import classification.AbstractClassification.Classification_SubGroup;
import entity.RawMaterial.RawMaterialType;

/**
 * Base for all products.
 */

@Entity
@Table(name="abstractproduct")
@SequenceGenerator(name       =       "abstractproduct_sequence",       sequenceName       =
"abstractproduct_id_seq")
@TableGenerator(     name="abstractproduct_id",   table="primary_keys",   pkColumnName="key",
pkColumnValue="abstractproduct",
               valueColumnName="value")

@NamedQueries( {
    @NamedQuery(name = "AbstractProduct.findByActive",
                query = "SELECT a FROM AbstractProduct a where a.active = :active"),
    @NamedQuery(name = "AbstractProduct.findByName",
                query = "SELECT a FROM AbstractProduct a WHERE a.name = :name"),
    @NamedQuery(name = "AbstractProduct.findByProductCode",
                query   =   "SELECT   a   FROM   AbstractProduct   a   WHERE   a.productCode   =
:productCode") })


@Inheritance(strategy = InheritanceType.JOINED)
public abstract class AbstractProduct implements Serializable{
    /**
     * AbstractProduct class need to be inherit and add needed functionality to subclasses.
     */
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO, generator = "abstractproduct_sequence")
    private Long id;

    @Column(length = 125, nullable = false)
    private String name;

    @Column(length = 1, nullable = false)
    private Boolean active;

    @Column
    private Integer productCode;

    @Column(updatable = false)
    private Date createDate;
```

(jatkuu)

LIITE 1. (jatkoa)

```
@Column(length = 1, nullable = false)
        private Classification_MainGroup     mainGroup;

    @Column(length = 1, nullable = false)
        private Classification_SubGroup      subGroup;
```

```
    @Column(length = 1, nullable = false)
        private Classification_EndUsage      endUsage;

    @Embedded
    private Identification identification;


    public AbstractProduct() {
        createDate = new Date(System.currentTimeMillis());
    }

    public AbstractProduct(final String name, final Identification identification) {
        this.name = name;
        this.identification = identification;
        createDate = new Date(System.currentTimeMillis());
    }

    public Boolean isActive() { return active; }

    public void setActive(final Boolean active){ this.active = active; }
    public void setIdentification(final Identification identification) {
        this.identification = identification;
    }
    public void setProductCode(Integer productCode) { this.productCode = productCode; }
    public void setId(Long id) { this.id = id; }
    public void setName(final String name) { this.name = name; }

    public Long getId() { return id; }
    public String getName() { return name; }
    public Identification getIdentification() { return identification; }
    public Integer getProductCode() { return productCode; }
    public Date getCreatedDate() {return createDate;}

    @OneToOne(mappedBy = "abstractproduct",
              cascade = CascadeType.PERSIST, optional = true)

    public abstract void remove();

    public abstract AbstractClassification getClassification();

    public void setMainGroup(Classification_MainGroup mainGroup)
        {this.mainGroup = mainGroup;}
    public void setSubGroup(Classification_SubGroup subGroup) {this.subGroup = subGroup;}
    public void setEndUsage(Classification_EndUsage endUsage) {this.endUsage = endUsage;}

    protected Classification_MainGroup getMainGroup() {return mainGroup;}
    protected Classification_SubGroup getSubGroup() {return subGroup;}
    protected Classification_EndUsage getEndUsage() {return endUsage;}

}
```

LIITE 2. ProductTypeA-luokka

```
package entity;

import java.io.Serializable;
import java.util.Calendar;
import java.util.Date;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.ManyToOne;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;
import javax.persistence.TableGenerator;
import classification.AbstractClassification;
```

```
import classification.ProductTypeAClassification;
import classification.ProductTypeAClassification.EnumType;
import entity.AbstractComponent.ComponentType;

/**
 *  ProductTypeA inherit AbstractProduct class and implements ProductTypeA functionality.
 */
@Entity
@Table(name="producttypea")
@SequenceGenerator(name = "productypea_sequence", sequenceName = "producttypea_id_seq")
@TableGenerator(      name="producttypea_id",   table="primary_keys",   pkColumnName="key",
pkColumnValue="producttypea",
                valueColumnName="value")
public class ProductTypeA extends AbstractProduct implements Serializable{

       @ManyToOne(cascade = { CascadeType.MERGE, CascadeType.PERSIST })
    private RawMaterial rawMaterial;

    @Column(nullable = true)
    private Integer integerValue;

    @Column(nullable = true)
    private Float floatValue;

    @Column(length = 125,nullable = true)
    private String stringValue;

    @Column(length = 1, nullable = true)
       private EnumType      enumValue;

    public ProductTypeA() {
       this.enumValue = classification.ProductTypeAClassification.EnumType.NO_TYPE;
    }

    public void setIntegerValue(final Integer value) { integerValue = value;}
    public void setFloatValue(final Float value) { floatValue = value;}
    public void setStringValue(final String value) { stringValue = value;}
    private void setEnumValue(final EnumType value) { enumValue = value;}

    public Integer getIntegerValue() { return integerValue;}
    public Float getFloatValue() { return floatValue;}
    public String getStringValue() { return stringValue;}
    public EnumType getEnumValue() { return enumValue;}

    public void setRawMaterial(final RawMaterial rawMaterial) {
       this.rawMaterial = rawMaterial;

       switch (rawMaterial.getRawmaterialType())
       {
              case NOT_SET:
                     setEnumValue(ProductTypeAClassification.EnumType.NO_TYPE);
                            break;
              case RAWMATERIAL_TYPE_A:
                     setEnumValue(ProductTypeAClassification.EnumType.TYPE_A);
                     break;
```

(jatkuu)

LIITE 2. (jatkoa)

```
              case RAWMATERIAL_TYPE_B:
                     setEnumValue(ProductTypeAClassification.EnumType.TYPE_B);
                     break;

              case RAWMATERIAL_TYPE_C:
                     setEnumValue(ProductTypeAClassification.EnumType.TYPE_C);
                     break;
              case RAWMATERIAL_TYPE_D:
                     setEnumValue(ProductTypeAClassification.EnumType.TYPE_D);
                     break;
              default: setEnumValue(ProductTypeAClassification.EnumType.NO_TYPE);
                     break;
       }
    }
    public RawMaterial getRawMaterial(){return rawMaterial;}


    @Override
    public void remove() { }

    @Override
    public AbstractClassification getClassification()
```

```
    {
        return new ProductTypeAClassification(
                                getMainGroup(),
                                getSubGroup(),
                                getEndUsage(),
                                enumValue);


    }
}
```

LIITE 3. ProductTypeB-luokka

```
package entity;

import java.io.Serializable;
import java.util.Date;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.ManyToMany;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;
import javax.persistence.TableGenerator;
import classification.AbstractClassification;
import classification.ProductTypeAClassification;
import classification.ProductTypeBClassification;
import classification.ProductTypeBClassification.ComponentContent;
import exception.AbstractComponentNotFound;

/**
 *  ProductTypeB inherit AbstractProduct class. ProductTypeB have many to many
 *  relationship whit AbstractComponent. ProductionTypeB define mappedBy which
 *  means that class is the master of the relationship and class also take care
 *  of cascading changes to the database.
 */

@Entity
@Table(name="producttypeb")
@SequenceGenerator(name = "producttypeb_sequence", sequenceName = "producttypeb_id_seq")
@TableGenerator(      name="producttypeb_id",   table="primary_keys",   pkColumnName="key",
pkColumnValue="producttypeb",
              valueColumnName="value")

public class ProductTypeB extends AbstractProduct implements Serializable{
    //@Column(length = 20, nullable = false)
       @Column(length = 1, nullable = true)
    ComponentContent componentContent;

    /**
     * AbstractComponent may belong several different ConcreteComponent. The
     * mappedBy connects the relationship to the AbstractComponent. When class
     * contains merge or persist operation, changes to this collection
     * and the contents of the collection will be updated.
     *
     * As well as if update the properties of the AbstracComponent in the set,
     * when we persist the ProductTypeB, the AbstractComponent will also get updated.
     */
    @ManyToMany(mappedBy = "productsByComponent", cascade = { CascadeType.PERSIST,
            CascadeType.MERGE }, fetch = FetchType.EAGER)
    private Set<AbstractComponent> components;

    public ProductTypeB() { }

    public ProductTypeB(final String name, final Identification identification,
        final Boolean active) {
        super(name,identification);
        setActive(active);
    }

    public ProductTypeB(final String name, final Identification identification,
        final Boolean active, final AbstractComponent... components) {
        super(name,identification);
        for (AbstractComponent c : components) {
            addComponents(c);
        }
    }
```

(jatkuu)

LIITE 3. (jatkoa)

```
    @Override
    public void remove() {
        for (AbstractComponent c : getComponents()) {
            c.removeComponent(this);
        }
        getComponents().clear();
```

```
    }


    public void removeComponentById(long id) {
        for (AbstractComponent c : getComponents()) {

               if(c.getId() == id ) c.removeComponent(this);
        }

        setComponentContent();
        //throw new AbstractComponentNotFound(id);
    }

    public AbstractComponent findComponentById(long id) {

       for (AbstractComponent c : getComponents()) {
               if(c.getId() == id ) return c;
       }

       throw new AbstractComponentNotFound(id);
    }

    public Set<AbstractComponent> getComponents() {
        if (components == null) {
               components = new HashSet<AbstractComponent>();
        }
        return components;
    }

    public void addComponents(final AbstractComponent component) {
        getComponents().add(component);
        component.addComponent(this);
    }

    public void setComponentContent(){
       this.componentContent = getComponentTypeInformation();
    }

    private ComponentContent getComponentTypeInformation()
    {
       Boolean ATypeFound = false;
       Boolean BTypeFound = false;

       final Set<AbstractComponent> list = getComponents();

           for (AbstractComponent r : list) {
               if (r instanceof ComponentTypeA) { ATypeFound = true;}
               if (r instanceof ComponentTypeB) { BTypeFound = true;}
           }
           if (ATypeFound && BTypeFound)
               return
               classification.ProductTypeBClassification.ComponentContent.INCLUDE_BOTH;
           if (ATypeFound)
               return classification.ProductTypeBClassification.ComponentContent.INCLUDE_A;
           if (BTypeFound)
               return classification.ProductTypeBClassification.ComponentContent.INCLUDE_B;

           return classification.ProductTypeBClassification.ComponentContent.EMPTY;
    }
```

(jatkuu)

LIITE 3. (jatkoa)

```
@Override
    public AbstractClassification getClassification()
    {
       return new ProductTypeBClassification(
                               getMainGroup(),
                               getSubGroup(),
                               getEndUsage(),
                               componentContent);


    }
```

LIITE 4. AbstractComponent-luokka

```
package entity;

import java.io.Serializable;
import java.util.HashSet;
import java.util.Set;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.ManyToMany;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.OneToOne;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;
import javax.persistence.TableGenerator;
import javax.persistence.Inheritance;
import javax.persistence.InheritanceType;


/**
 * Base for all products components.
 */
@Entity
@Table(name="abstractcomponent")
@SequenceGenerator(name       =       "abstractcomponent_sequence",       sequenceName       =
"abstractcomponent_id_seq")
@TableGenerator(    name="abstractcomponent_id",  table="primary_keys",  pkColumnName="key",
pkColumnValue="abstractcomponent",
               valueColumnName="value")

@NamedQueries( {
    @NamedQuery(name = "AbstractComponent.findByActive",
                query = "SELECT a FROM AbstractComponent a where a.active = :active"),
    @NamedQuery(name = "AbstractComponent.findByTitle",
                query = "SELECT a FROM AbstractComponent a WHERE a.title = :title"),
    @NamedQuery(name = "AbstractComponent.findByCode",
                query  =  "SELECT  a  FROM  AbstractComponent  a  WHERE  a.componentCode  =
:componentCode") })


@Inheritance(strategy = InheritanceType.JOINED)
public abstract class AbstractComponent implements Serializable{
    /**
     *  AbstractComponent  class  need  to  be  inherit  and  add  needed  functionality  to
subclasses.
     */

        public enum ComponentType {COMPONENT_TYPE_A,COMPONENT_TYPE_B}

        @Id
    @GeneratedValue(strategy        =        GenerationType.AUTO,        generator        =
"abstractcomponent_sequence")
    private Long id;

    @Column(length = 125, nullable = false)
    private String title;

    @Column
    private Integer componentCode;

    @Column(length = 1, nullable = true)
        private ComponentType  componentType;

    @Column(length = 1, nullable = true)
    private Boolean active;
```

(jatkuu)

LIITE 4. (jatkoa)

```
    @OneToOne(mappedBy = "abstractcomponent",
              cascade = CascadeType.PERSIST, optional = true)

    public abstract void remove();

    public AbstractComponent() { }
    public AbstractComponent(final String title) { this.title = title; }
```

```
    public Long getId() { return id;}
    public Boolean isActive() { return active; }
    public Integer getComponentCode() { return componentCode; }
    public ComponentType getComponentType() {return componentType; }
    public String getTitle() { return title; }

    public void setId(Long id) { this.id = id; }
    public void setActive(final Boolean active){ this.active = active;}
    public void setComponentCode(Integer componentCode)
       { this.componentCode = componentCode; }
    public void setComponentType(final ComponentType componentType)
       { this.componentType = componentType; }
    public void setTitle(String title) { this.title = title; }

    /**
     * Concrete product might be several AbstractComponents and AbstractComponent
     * can belong several Concrete Product. That caused a many-to-many relationship
     * relationship between Concrete Products and AbstractComponent. A relation is
     * bidirectional.
     */
    @ManyToMany
    private Set<AbstractProduct> productsByComponent;

    public void addComponent(final AbstractProduct c) {
        getComponentProducts().add(c);
    }

    public Set<AbstractProduct> getComponentProducts() {
        if(productsByComponent == null) {
              productsByComponent = new HashSet<AbstractProduct>();
        }
         return productsByComponent;
    }

    public void removeComponent(final AbstractProduct c) {
       getComponentProducts().remove(c);
    }
}
```

LIITE 5. ComponentTypeA-luokka

```
package entity;

import java.io.Serializable;
import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.ManyToOne;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;
import javax.persistence.TableGenerator;

/**
 * AbstractComponent class need to be inherit and define ComponenTypeA functionality.
 */
@Entity
@Table(name="componenttypea")
@SequenceGenerator(name       =       "componenttypea_sequence",       sequenceName       =
"componenttypea_id_seq")
@TableGenerator(     name="componenttypea_id",   table="primary_keys",   pkColumnName="key",
pkColumnValue="componenttypea",
                valueColumnName="value")
public class ComponentTypeA extends AbstractComponent implements Serializable{
        @ManyToOne(cascade = { CascadeType.MERGE, CascadeType.PERSIST })
    private RawMaterial rawMaterial;

    @Column(nullable = true)
    private Float rawmaterialAmount;

    @Column(nullable = true)
    private Integer rawmaterialLayer;

    public ComponentTypeA() { }

    public void setRawMaterial(final RawMaterial rawMaterial) {
        this.rawMaterial = rawMaterial;
    }
    public void setRawMaterialAmount(final Float amount) {rawmaterialAmount = amount;}
    public void setRawMaterialLayer(final Integer layer) {rawmaterialLayer = layer;}

    public RawMaterial getRawMaterial() { return rawMaterial; }
    public Float getRawMaterialAmount() {return rawmaterialAmount;}
    public Integer getRawMaterialLayer() {return rawmaterialLayer;}

    @Override
    public void remove() { }


}
```

LIITE 6. ComponentTypeB-luokka

```
package entity;

import java.io.Serializable;
import java.util.Calendar;
import java.util.Date;

import javax.persistence.CascadeType;
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.ManyToOne;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;
import javax.persistence.TableGenerator;

/**
 * AbstractComponent class need to be inherit and define ComponenTypeB functionality.
 */
@Entity
@Table(name="componenttypeb")
@SequenceGenerator(name       =       "componenttypeb_sequence",       sequenceName       =
"componenttypeb_id_seq")
@TableGenerator(     name="componenttypeb_id",   table="primary_keys",   pkColumnName="key",
pkColumnValue="componenttypeb",
                valueColumnName="value")
public class ComponentTypeB extends AbstractComponent implements Serializable{

    @Column(nullable = true)
    private Float partValue;

    @Column(nullable = true)
    private Integer partNumber;

    @Column(nullable = true)
    private Integer partPcs;

    public ComponentTypeB() { }

    public void setPartNumber(final Integer partNumber) {this.partNumber = partNumber;}
    public void setPartValue(final Float partValue) {this.partValue = partValue;}
    public void setPartPCS(final Integer partPcs) {this.partPcs = partPcs;}

    public Integer getPartNumber() {return partNumber;}
    public Integer getPartPCS() {return partPcs;}
    public Float getPartValue() {return partValue;}

    @Override
    public void remove() { }

}
```

LIITE 7. RawMaterial-luokka

```
package entity;
import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Embedded;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.NamedQueries;
import javax.persistence.NamedQuery;
import javax.persistence.SequenceGenerator;
import javax.persistence.Table;
import javax.persistence.TableGenerator;

@Entity
@Table(name="rawmaterial")
@SequenceGenerator(name = "rawmaterial_sequence", sequenceName = "rawmaterial_id_seq")
@TableGenerator(          name="book_id",     table="primary_keys",     pkColumnName="key",
pkColumnValue="rawmaterial",
              valueColumnName="value")

@NamedQueries( {
    @NamedQuery(name = "RawMaterial.findByRawMaterialName",
                query  =  "SELECT  r  FROM  RawMaterial  r  where  r.rawmaterialName  =
:rawmaterialname"),
    @NamedQuery(name = "RawMaterial.findByRawMaterialCode",
                query  =  "SELECT  r  FROM  RawMaterial  r  WHERE  r.rawmaterialCode  =
:rawmaterialcode"),
    @NamedQuery(name = "RawMaterial.RawMaterialByActiveStatus",
                query = "SELECT r FROM RawMaterial r WHERE r.active = :active")})

public class RawMaterial implements Serializable {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO, generator = "rawmaterial_sequence")
    private Long id;

    @Column(length = 125, nullable = false)
    private String rawmaterialName;

    @Column(nullable = false,unique = true)
    private Integer rawmaterialCode;

    @Column(length = 1, nullable = false)
       private RawMaterialType       rawmaterialType;

    @Column(length = 1, nullable = true)
    private Boolean active;

    @Embedded
    private Identification identification;

    public enum RawMaterialType {NOT_SET, RAWMATERIAL_TYPE_A, RAWMATERIAL_TYPE_B,
       RAWMATERIAL_TYPE_C, RAWMATERIAL_TYPE_D}

    public RawMaterial() { }

    public RawMaterial(String name, Integer code, RawMaterialType type) {
       rawmaterialName = name;
       rawmaterialCode = code;
       rawmaterialType = type;
    }

    public Boolean isActive() { return active; }
    public void setActive(final Boolean active){ this.active = active; }
    public Identification getIdentification() { return identification; }
    public void setId(Long id) { this.id = id; }
    public void setRawMaterialName(final String rawmaterialName) {
        this.rawmaterialName = rawmaterialName;
    }
```

(jatkuu)

LIITE 7. (jatkoa)

```
    public void setRawMaterialCode(final Integer code) { rawmaterialCode = code;}
    public void setRawMaterialType(final RawMaterialType type) { rawmaterialType = type;}

    public void setIdentification(final Identification identification) {
        this.identification = identification;
    }
```

```
    public Long getId() { return id; }
    public String getRawMaterialName() {return rawmaterialName; }
    public RawMaterialType getRawmaterialType(){return rawmaterialType;}
    public Integer getRawMaterialCode(){return rawmaterialCode;}

    @Override
    public boolean equals(final Object object) {
        if (object instanceof RawMaterial) {
            final RawMaterial rhs = (RawMaterial) object;
            return getRawMaterialCode().equals(rhs.getRawMaterialCode());
        }
        return false;
    }
}
```

LIITE 8. Identification-luokka

```
package entity;

import java.io.Serializable;
import javax.persistence.Column;
import javax.persistence.Embeddable;

/**
 * Avoid to repeat code and description in both AbstractProduct and RawMaterial can
 * be created embedded class. The class includes all vendors related information.
 * The fields of embedded class end up as columns in the table that contains the
 * class that embeds this entity. If vendor related columns need to be added to
 * this class is a correct place.
 */

@Embeddable
public class Identification implements Serializable{
        private static final long serialVersionUID = -598968925590693310L;

        @Column(length = 20, nullable = false)
    private String code;
    @Column(length = 30, nullable = false)
    private String description;

    public Identification() {

    }

    public Identification(final String code, final String description) {
        setCode(code);
        setDescription(description);
    }

    public String getCode() {
        return code;
    }

    public void setCode(final String code) {
        if (code != null) {
            this.code = code;
        } else {
            this.code = "";
        }
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(final String description) {
        if (description != null) {
            this.description = description;
        } else {
            this.description = "";
        }
    }

    public boolean equals(final Object object) {
        if (object instanceof Identification) {
            final Identification rhs = (Identification) object;
            return rhs.getCode().equals(getCode())
                    && rhs.getDescription().equals(getDescription());
        }
        return false;
    }

    public int hashCode() {
        return getCode().hashCode() * getDescription().hashCode();
    }
}
```

LIITE 9. AbstractClassification-, ProductTypeAClassification- ja ProductTypeBClassification-luokat

```
package classification;

public abstract class AbstractClassification {

        AbstractClassification(){}

    public static enum Classification_MainGroup {MAIN_GROUP_A, MAIN_GROUP_B,
         MAIN_GROUP_C, MAIN_GROUP_D}
    public static enum Classification_SubGroup {SUB_GROUP_A, SUB_GROUP_B, SUB_GROUP_C,
        SUB_GROUP_D}
    public static enum Classification_EndUsage {END_USAGE_A, END_USAGE_B, END_USAGE_C,
        END_USAGE_D}

    private Classification_MainGroup mainGroup;
    private Classification_SubGroup subGroup;
    private Classification_EndUsage endUsage;

    public Classification_MainGroup getMainGroup() {return mainGroup;}
    public Classification_SubGroup getSubGroup() {return subGroup;}
    public Classification_EndUsage getEndUsage() {return endUsage;}

    protected void setMainGroup(Classification_MainGroup mainGroup)
        {this.mainGroup = mainGroup;}
    protected void setSubGroup(Classification_SubGroup subGroup)
        {this.subGroup = subGroup;}
    protected void setEndUsage(Classification_EndUsage endUsage)
        {this.endUsage = endUsage;}
}
////////////////////////////////////////////////////////////////////////////////////
package classification;

public class ProductTypeAClassification extends AbstractClassification{

        public static enum EnumType {NO_TYPE, TYPE_A, TYPE_B, TYPE_C, TYPE_D,TYPE_E}

        private EnumType enumType;

        public ProductTypeAClassification(Classification_MainGroup mainGroup,
        Classification_SubGroup   subGroup,   Classification_EndUsage   endUsage,   EnumType
        enumType){
                this.setMainGroup(mainGroup);
                this.setSubGroup(subGroup);
                this.setEndUsage(endUsage);
                this.enumType = enumType;
        }
        public EnumType getEnumType(){return enumType;}
}

////////////////////////////////////////////////////////////////////////////////////
package classification;

public class ProductTypeBClassification extends AbstractClassification{
        public static enum ComponentContent {EMPTY, INCLUDE_A, INCLUDE_B, INCLUDE_BOTH}

        private ComponentContent componentContent;

        public ProductTypeBClassification(Classification_MainGroup mainGroup,
        Classification_SubGroup subGroup, Classification_EndUsage endUsage,ComponentContent
        componentContent){
                this.setMainGroup(mainGroup);
                this.setSubGroup(subGroup);
                this.setEndUsage(endUsage);
                this.componentContent = componentContent;
        }

        public ComponentContent getComponentContent(){return componentContent;}
}
```

LIITE 10. Poikkeusten käsittely (Exception-luokat)

```
package exception;

import javax.ejb.ApplicationException;

/**
 * A simple unchecked exception reflecting a particular business rule violation.
 * AbstractComponent can not be created if a same Component code already exists.
 */
@ApplicationException
public class AbstractComponentAlreadyExists extends RuntimeException {

        private static final long serialVersionUID = -5979482587038881488L;

        final Integer abstractComponentCode ;

    public AbstractComponentAlreadyExists(final Integer abstractComponentCode) {
        this.abstractComponentCode = abstractComponentCode;
    }

    public Integer getAbstractComponentCode() {
        return abstractComponentCode;
    }
}
////////////////////////////////////////////////////////////////////////////////////////
package exception;

import javax.ejb.ApplicationException;

/**
 * A simple unchecked exception reflecting a particular business rule violation.
 * AbstractComponent not found by using component code.
 */
@ApplicationException
public class AbstractComponentCodeNotFound extends RuntimeException {
        private static final long serialVersionUID = 6673029848258503496L;
        final Integer abstractComponentCode ;

    public AbstractComponentCodeNotFound(final Integer abstractComponentCode) {
        this.abstractComponentCode = abstractComponentCode;
    }

    public Integer getAbstractComponentId() {
        return abstractComponentCode;
    }
}
////////////////////////////////////////////////////////////////////////////////////////
package exception;

import javax.ejb.ApplicationException;

/**
 * A simple unchecked exception reflecting a particular business rule violation.
 * AbstractComponent not found by component id.
 */
@ApplicationException
public class AbstractComponentNotFound extends RuntimeException {
        private static final long serialVersionUID = 6673029848258503496L;
        final Long abstractComponentId ;

    public AbstractComponentNotFound(final Long abstractComponentId) {
        this.abstractComponentId = abstractComponentId;
    }

    public Long getAbstractComponentId() {
        return abstractComponentId;
    }
}
```

(jatkuu)

LIITE 10. (jatkoa)

```
package exception;

import javax.ejb.ApplicationException;

/**
 * A simple unchecked exception reflecting a particular business rule violation.
```

```
 * AbstractProduct can not be created if a same Product code already exists.
 */
@ApplicationException
public class AbstractProductAlreadyExists extends RuntimeException {
        private static final long serialVersionUID = 1634460351825750998L;

        final Integer abstractProductCode ;

    public AbstractProductAlreadyExists(final Integer abstractProductCode) {
        this.abstractProductCode = abstractProductCode;
    }

    public Integer getAbstractProductCode() {
        return abstractProductCode;
    }
}
////////////////////////////////////////////////////////////////////////////////
package exception;

import javax.ejb.ApplicationException;

/**
 * A simple unchecked exception reflecting a particular business rule violation.
 * AbstractProduct not found by using product id.
 */
@ApplicationException
public class AbstractProductNotFound extends RuntimeException {

        private static final long serialVersionUID = -8000728864644694659L;
        final Integer rawMaterialtId ;

    public AbstractProductNotFound(final Integer rawMaterialtId) {
        this.rawMaterialtId = rawMaterialtId;
    }

    public Integer getAbstractProductId() {
        return rawMaterialtId;
    }
}
////////////////////////////////////////////////////////////////////////////////
package exception;

import javax.ejb.ApplicationException;

/**
 * A simple unchecked exception reflecting a particular business rule violation.
 * RawMaterial can not be created if a same RawMaterial code already exists.
 */
@ApplicationException
public class RawMaterialAlreadyExists extends RuntimeException {

        private static final long serialVersionUID = 2206503743287020088L;
        final Integer rawMaterialCode ;

    public RawMaterialAlreadyExists(final Integer rawMaterialCode) {
        this.rawMaterialCode = rawMaterialCode;
    }

    public Integer getRawMaterialCode() {
        return rawMaterialCode;
    }
}
```

(jatkuu)

LIITE 10. (jatkoa)

```
package exception;

import javax.ejb.ApplicationException;

/**
 * A simple unchecked exception reflecting a particular business rule violation.
 * RawMaterial can not be find by abstractProductId.
 */
@ApplicationException
public class RawMaterialNotFound extends RuntimeException {

        private static final long serialVersionUID = -2274089977168237723L;
        final Integer abstractProductId ;
```

```
    public RawMaterialNotFound(final Integer abstractProductId) {
        this.abstractProductId = abstractProductId;
    }

    public Integer getAbstractProductId() {
        return abstractProductId;
    }
}
////////////////////////////////////////////////////////////////////////////////////////////

package exception;

import javax.ejb.ApplicationException;

@ApplicationException
public class TooManyEnityObjectExists extends RuntimeException {

        private static final long serialVersionUID = 6080002280196353882L;
        private final Class clazz;
    private final Object key;

    public TooManyEnityObjectExists(final Class clazz, final Object key) {
        this.clazz = clazz;
        this.key = key;
    }

    public Class getClazz() {
        return clazz;
    }

    public Object getKey() {
        return key;
    }
}
```

LIITE 11. BaseDao-luokka

```
package service;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;

/**
 * A simple base class for all dao's. It offers 2 features. First, it has the
 * entity manager attribute. Second, it makes it possible to have a common test
 * base class with the getDao() method to allow for automatic initialization.
 */
public abstract class BaseDao {
       @PersistenceContext(unitName = "miniappEJB")

    private EntityManager em;

    public void setEm(final EntityManager em) {
        this.em = em;
    }

    public EntityManager getEm() {
        return em;
    }
}
```

LIITE 12. AbstractProductDao- ja AbstractProductDaoBean-luokat

```
package service;
import java.util.List;
import entity.AbstractComponent;
public interface AbstractComponentDao {
       public abstract void create(final AbstractComponent r);
       public abstract AbstractComponent retrieve(final Long id);
       public abstract void remove(Long id);
       public abstract AbstractComponent update(AbstractComponent r);
       public abstract AbstractComponent findById(Long id);
       public abstract List<AbstractComponent> findByTitle(final String title);
       public abstract List<AbstractComponent> findByComponentCode(final Integer
              componentCode);
}
////////////////////////////////////////////////////////////////////////////////////
package service;
import java.util.List;
import javax.ejb.Stateless;
import entity.AbstractProduct;
/**
 * This class offers the basic create, read, update, delete functions required
 * for a AbstractProduct.
 */
@Stateless
public class AbstractProductDaoBean extends BaseDao implements AbstractProductDao {
       @Override
       public void create(final AbstractProduct r) {
        getEm().persist(r);
    }
    @Override
        public AbstractProduct retrieve(final Long id) {
         return getEm().find(AbstractProduct.class, id);
    }
    @Override
    public void remove(Long id) {
         final AbstractProduct r = retrieve(id);
         if (r != null) {
             r.remove();
             getEm().remove(r);
         }
         getEm().flush();
    }
    @Override
        public AbstractProduct update(AbstractProduct r) {
         return getEm().merge(r);
    }
    @Override
        public AbstractProduct findById(Long id) {
         return getEm().find(AbstractProduct.class, id);
    }
    @Override
        @SuppressWarnings("unchecked")
    public List<AbstractProduct> findByActive(final Boolean active) {
        return
        getEm().createNamedQuery("AbstractProduct.findByActive").setParameter("active",
                active).getResultList();
    }
    @Override
        @SuppressWarnings("unchecked")
    public  List<AbstractProduct> findByName(final String name){
        return getEm().createNamedQuery("AbstractProduct.findByName").setParameter("name",
                name).getResultList();
    }
    @Override
        @SuppressWarnings("unchecked")
    public List<AbstractProduct> findByProductCode(final Integer productCode){
        return
        getEm().createNamedQuery("AbstractProduct.findByProductCode").setParameter(
        "productCode", productCode).getResultList();
    }
}
```

LIITE 13. AbstractComponentDao- ja AbstractComponentDaoBean-luokat

```
package service;
import java.util.List;
import entity.AbstractComponent;
```

```
public interface AbstractComponentDao {
        public abstract void create(final AbstractComponent r);
        public abstract AbstractComponent retrieve(final Long id);
        public abstract void remove(Long id);
        public abstract AbstractComponent update(AbstractComponent r);
        public abstract AbstractComponent findById(Long id);
        public abstract List<AbstractComponent> findByTitle(final String title);
        public abstract List<AbstractComponent> findByComponentCode(final Integer
                componentCode);
}
////////////////////////////////////////////////////////////////////////////////////
package service;
import java.util.List;
import javax.ejb.Stateless;
import entity.AbstractComponent;
/**
 * This class offers the basic create, read, update, delete functions required
 * for a AbstractComponent.
 */
@Stateless
public class AbstractComponentDaoBean extends BaseDao implements AbstractComponentDao {
        @Override
        public void create(final AbstractComponent r) {
         getEm().persist(r);
    }
    @Override
        public AbstractComponent retrieve(final Long id) {
         return getEm().find(AbstractComponent.class, id);
    }
    @Override
    public void remove(Long id) {
         final AbstractComponent r = retrieve(id);
         if (r != null) {
             r.remove();
             getEm().remove(r);
         }
         getEm().flush();
    }
    @Override
        public AbstractComponent update(AbstractComponent r) {
         return getEm().merge(r);
    }
    @Override
        public AbstractComponent findById(Long id) {
         return getEm().find(AbstractComponent.class, id);
    }
    @Override
        @SuppressWarnings("unchecked")
    public  List<AbstractComponent> findByTitle(final String findByTitle){
        return
        getEm().createNamedQuery("AbstractComponent.findByTitle").setParameter("title",
                        findByTitle).getResultList();

    }
    @Override
        @SuppressWarnings("unchecked")
    public List<AbstractComponent> findByComponentCode(final Integer componentCode){
        return
        getEm().createNamedQuery("AbstractComponent.findByCode").setParameter(
        "componentCode", componentCode).getResultList();
    }

}
```

LIITE 14. RawMaterialDao- ja RawMaterialDaoBean-luokat

```
package service;
import java.util.List;
import entity.RawMaterial;
public interface RawMaterialDao {
        public abstract void remove(final Long id);
        public abstract RawMaterial update(final RawMaterial d);
        public abstract RawMaterial createRawMaterial(final RawMaterial rm);
        public abstract RawMaterial retrieve(final Long id);
        public abstract List<RawMaterial> findByRawMaterialName(final String name);
        public abstract List<RawMaterial> findByRawMaterialCode(final Integer code);
        public abstract List<RawMaterial> findRawMaterialByActiveStatus(final Boolean
                active);
}
////////////////////////////////////////////////////////////////////////////////////
package service;
import java.util.List;
```

```
import javax.ejb.Stateless;
import entity.RawMaterial;
@Stateless
public class RawMaterialDaoBean extends BaseDao implements RawMaterialDao {

    @Override
    public  void remove(final Long id)
    {
        final RawMaterial m = retrieve(id);
        if (m != null) {
            getEm().remove(m);
        }
    }
    @Override
        public RawMaterial update(final RawMaterial m) {
        return getEm().merge(m);
    }
    @Override
        public RawMaterial createRawMaterial(final RawMaterial rm){
        //final RawMaterial r =
                new RawMaterial(rm.getRawMaterialName(),rm.getRawMaterialCode(),
                rm.getRawmaterialType());
        getEm().persist(rm);
        return rm;
    }
    @Override
        public RawMaterial retrieve(final Long id) {
        return getEm().find(RawMaterial.class, id);
    }
    @Override
    @SuppressWarnings("unchecked")
    public List<RawMaterial> findByRawMaterialName(final String name){
        return
        getEm().createNamedQuery("RawMaterial.findByRawMaterialName").setParameter(
                "rawmaterialname",name).getResultList();
    }
    @Override
    @SuppressWarnings("unchecked")
    public List<RawMaterial> findByRawMaterialCode(final Integer code){
        return
        getEm().createNamedQuery("RawMaterial.findByRawMaterialCode").setParameter(
        "rawmaterialcode",code).getResultList();
    }
    @Override
    @SuppressWarnings("unchecked")
    public List<RawMaterial> findRawMaterialByActiveStatus(final Boolean active){
        return
        getEm().createNamedQuery("RawMaterial.RawMaterialByActiveStatus").setParameter(
        "active",active).getResultList();
    }

}
```

LIITE 15. FacadeBeanRemote-luokka

```
package service;

import java.util.List;
import java.util.Set;
import javax.ejb.Remote;
import entity.AbstractComponent;
import entity.AbstractProduct;
import entity.ComponentTypeA;
import entity.ComponentTypeB;
import entity.ProductTypeA;
import entity.ProductTypeB;
import entity.RawMaterial.RawMaterialType;
import entity.Identification;
import entity.RawMaterial;

@Remote
public interface FacadeBeanRemote {

	public ProductTypeA createProductTypeA(final String name,final Integer productCode,
		final Identification id, final Boolean active);

	public ProductTypeA updateProductTypeARawMaterial(final ProductTypeA productTypeA,
		final RawMaterial rawMaterial);

	public ProductTypeB createProductTypeB(final String name,final Integer productCode,
		final Identification id, final Boolean active);

	public ProductTypeB addAbstractComponent(final ProductTypeB product,
		final AbstractComponent component);

	public ProductTypeB removeComponentById(final ProductTypeB prod,
		final AbstractComponent comp);

	public ComponentTypeA createComponentTypeA(final String title,
		final Integer componentCode, final Boolean active,
		final RawMaterial rawMaterial, final Float amount, final Integer layer);

	public ComponentTypeB createComponentTypeB(final String title,
		final Integer componentCode, final Boolean active,
		final Integer partNumber, final Float partValue, final Integer partPcs);

	public AbstractProduct findAbstractProductByProductCode(Integer productCode);

	public AbstractProduct findAbstractPoductById(final long id);

	public AbstractProduct updateProduct(final AbstractProduct abstractProduct);

	public Set<AbstractProduct> findProductByComponent(final Long id);

	public List<AbstractProduct> findByActiveProduct(final Boolean active);

	public List<AbstractProduct> findByProductName(final String name);

	public AbstractComponent findAbstractComponentByComponentCode(
		Integer componentCode);
	public List<AbstractComponent> findByComponentTitle(final String title);

	public RawMaterial createRawMaterial(final String name, final Integer code,
		final RawMaterialType type,final Identification id);
	public RawMaterial findByRawMaterialCode(final Integer code);

	public List<RawMaterial> findRawMaterialByActiveStatus(final Boolean active);

	public List<RawMaterial> findRawMaterialByName(final String name);

	public ComponentTypeA createComponentTypeA(final String name);

	public ProductTypeB addComponent(final ProductTypeB prod,
		final ComponentTypeA type);

}
```

LIITE 16. FacadeBeanLocal-luokka

```
package service;
import java.util.Date;
import java.util.List;
import java.util.Set;
import javax.ejb.Local;
import entity.AbstractComponent;
import entity.AbstractProduct;
import entity.Address;
import entity.Author;
import entity.Book;
import entity.ComponentTypeA;
import entity.ComponentTypeB;
import entity.Dvd;
import entity.Fine;
import entity.Identification;
import entity.Name;
import entity.Patron;
import entity.ProductTypeA;
import entity.ProductTypeB;
import entity.RawMaterial;
import entity.Resource;
import entity.RawMaterial.RawMaterialType;

@Local
public interface FacadeBeanLocal {
	public ProductTypeA createProductTypeA(final String name,final Integer productCode,
		  final Identification id, final Boolean active);
	public ProductTypeA updateProductTypeARawMaterial(final ProductTypeA productTypeA,
		  final RawMaterial rawMaterial);

	public ProductTypeB createProductTypeB(final String name,final Integer productCode,
		  final Identification id, final Boolean active);
	public ProductTypeB addAbstractComponent(final ProductTypeB product,
		 final AbstractComponent component);
	public ProductTypeB removeComponentById(final ProductTypeB prod,
		 final AbstractComponent comp);

	public ComponentTypeA createComponentTypeA(final String title,
		 final Integer componentCode, final Boolean active, final RawMaterial
		 rawMaterial, final Float amount, final Integer layer);

	public ComponentTypeB createComponentTypeB(final String title,
		 final Integer componentCode, final Boolean active, final Integer partNumber,
		 final Float partValue, final Integer partPcs);

	public AbstractProduct findAbstractProductByProductCode(Integer productCode);

	public AbstractProduct findAbstractPoductById(final long id);

	public AbstractProduct updateProduct(final AbstractProduct abstractProduct);

	public Set<AbstractProduct> findProductByComponent(final Long id);

	public List<AbstractProduct> findByActiveProduct(final Boolean active);
	public List<AbstractProduct> findByProductName(final String name);

	public AbstractComponent findAbstractComponentByComponentCode(
		 Integer componentCode);
	public List<AbstractComponent> findByComponentTitle(final String title);

	public RawMaterial createRawMaterial(final String name, final Integer code,
		 final RawMaterialType type,final Identification id);
	public RawMaterial findByRawMaterialCode(final Integer code);
	public List<RawMaterial> findRawMaterialByActiveStatus(final Boolean active);

	public List<RawMaterial> findRawMaterialByName(final String name);
	public ComponentTypeA createComponentTypeA(final String name);

	public ProductTypeB addComponent(final ProductTypeB prod,
		 final ComponentTypeA type);
}
```

LIITE 17. Facade-luokka

```
package service;
import java.util.List;
import java.util.Set;
import entity.AbstractComponent;
import entity.AbstractProduct;
```

```
import entity.ComponentTypeB;
import entity.Identification;
import entity.ProductTypeB;
import entity.RawMaterial;
import entity.ProductTypeA;
import entity.ComponentTypeA;
import entity.RawMaterial.RawMaterialType;
public interface Facade {

	public abstract AbstractComponentDao getAbstractComponentDao();
	public abstract AbstractProductDao getAbstractProductDao();
	public abstract ProductTypeBDao getProductTypeBDao();
	public abstract RawMaterialDao getRawMaterialDao();

	public abstract void setAbstractComponentDao(
		final AbstractComponentDao abstractComponentDao);
	public abstract void setAbstractProductDao(
		final AbstractProductDao abstractProductDao);
	public abstract void setProductTypeBDao(final ProductTypeBDao productTypeBDao);
	public abstract void setRawMaterialDao(final RawMaterialDao rawMaterialDao);

	public abstract ProductTypeA createProductTypeA(final String name,
		final Integer productCode, final Identification id, final Boolean active);
	public abstract ProductTypeA updateProductTypeARawMaterial(
		final ProductTypeA productTypeA, final RawMaterial rawMaterial);

	public abstract ProductTypeB createProductTypeB(final String name,
		final Integer productCode, final Identification id, final Boolean active);
	public abstract ProductTypeB addAbstractComponent(final ProductTypeB product,
		final AbstractComponent component);
	public abstract ProductTypeB removeComponentById(final ProductTypeB prod,
		final AbstractComponent comp);

	public abstract ComponentTypeA createComponentTypeA(final String title,
		final Integer componentCode, final Boolean active,
		final RawMaterial rawMaterial, final Float amount, final Integer layer);

	public abstract ComponentTypeB createComponentTypeB(final String title,
		final Integer componentCode, final Boolean active, final Integer partNumber,
		final Float partValue, final Integer partPcs);
	public abstract AbstractProduct findAbstractProductByProductCode(
		Integer productCode);
	public abstract AbstractProduct findAbstractPoductById(final long id);
	public abstract AbstractProduct updateProduct(
		final AbstractProduct abstractProduct);
	public abstract Set<AbstractProduct> findProductByComponent(final Long id);
	public abstract List<AbstractProduct> findByActiveProduct(final Boolean active);
	public abstract List<AbstractProduct> findByProductName(final String name);

	public abstract AbstractComponent findAbstractComponentByComponentCode(
		Integer componentCode);
	public abstract List<AbstractComponent> findByComponentTitle(final String title);

	public abstract RawMaterial createRawMaterial(final String name, final Integer code,
		final RawMaterialType type,final Identification id);
	public abstract RawMaterial findByRawMaterialCode(final Integer code);
	public abstract List<RawMaterial> findRawMaterialByActiveStatus(
		final Boolean active);
	public abstract List<RawMaterial> findRawMaterialByName(final String name);

	public abstract ComponentTypeA createComponentTypeA(final String name);
	public abstract ProductTypeB addComponent(final ProductTypeB prod,
		final ComponentTypeA type);
	}
```

LIITE 18. FacadeBean-luokka

```
package service;
import java.util.List;
import java.util.Set;
import javax.ejb.EJB;
import javax.ejb.Stateless;
import entity.AbstractComponent;
import entity.ComponentTypeA;
import entity.ComponentTypeB;
import entity.Identification;
import entity.ProductTypeA;
import entity.ProductTypeB;
import entity.RawMaterial;
import entity.AbstractProduct;
import entity.RawMaterial.RawMaterialType;
import exception.AbstractComponentCodeNotFound;
import exception.AbstractProductNotFound;
```

```
import exception.EntityDoesNotExist;
import exception.RawMaterialAlreadyExists;
import exception.SeveralRawMaterialExistSameCode;
import exception.SeveralAbstractProductExistSameCode;
import classification.ProductTypeAClassification;
import classification.ProductTypeBClassification;

/**
 * This class provides a basic facade to the Product Handling System. If we had a user
 * interface, it would interact with this object rather than dealing with all of
 * the underlying DAOs.
 */
@Stateless
public class FacadeBean implements Facade,FacadeBeanRemote,FacadeBeanLocal {

    @EJB
    private AbstractProductDao abstractProductDao;
    @EJB
    private AbstractComponentDao abstractComponentDao;
    @EJB
    public ProductTypeBDao productTypeBDao;
    @EJB
    public RawMaterialDao rawMaterialDao;

        public static final String RemoteJNDIName =
                FacadeBean.class.getSimpleName() + "/remote";

        public static final String LocalJNDIName =
                FacadeBean.class.getSimpleName() + "/local";

    @Override
        public void setProductTypeBDao(final ProductTypeBDao productTypeBDao){
         this.productTypeBDao = productTypeBDao;
    }
    @Override

        public ProductTypeBDao getProductTypeBDao() {
         return productTypeBDao;
    }
    @Override
        public AbstractProductDao getAbstractProductDao() {
         return abstractProductDao;
    }
    @Override
        public void setAbstractProductDao(final AbstractProductDao abstractProductDao) {
         this.abstractProductDao = abstractProductDao;
    }
    @Override
        public AbstractComponentDao getAbstractComponentDao() {
         return abstractComponentDao;
    }
```

(jatkuu)

LIITE 18. (jatkoa)

```
    @Override
        public void setAbstractComponentDao(final AbstractComponentDao abstractComponentDao)
{
         this.abstractComponentDao = abstractComponentDao;
    }
    @Override
        public RawMaterialDao getRawMaterialDao() {
         return rawMaterialDao;
    }
    @Override
        public void setRawMaterialDao(final RawMaterialDao rawMaterialDao) {
         this.rawMaterialDao = rawMaterialDao;
    }
    @Override
        public ProductTypeA createProductTypeA(final String name,final Integer productCode,
                final Identification id, final Boolean active){
         final ProductTypeA a = new ProductTypeA();
         a.setName(name);
         a.setActive(active);
         a.setIdentification(id);
         a.setProductCode(productCode);

        a.setMainGroup(
        classification.AbstractClassification.Classification_MainGroup.MAIN_GROUP_A);
        a.setSubGroup(
```

```
    classification.AbstractClassification.Classification_SubGroup.SUB_GROUP_A);
    a.setEndUsage(
    classification.AbstractClassification.Classification_EndUsage.END_USAGE_A);
   // a.setEnumValue(classification.ProductTypeAClassification.EnumType.NO_TYPE);
    getAbstractProductDao().create(a);
    return a;
}
@Override
    public ProductTypeB createProductTypeB(final String name,final Integer productCode,
           final Identification id, final Boolean active){
    final ProductTypeB b = new ProductTypeB(name,id,active);
    b.setProductCode(productCode);
    b.setMainGroup(
    classification.AbstractClassification.Classification_MainGroup.MAIN_GROUP_B);
    b.setSubGroup(
    classification.AbstractClassification.Classification_SubGroup.SUB_GROUP_B);
    b.setEndUsage(
    classification.AbstractClassification.Classification_EndUsage.END_USAGE_B);
    b.setComponentContent();
    getAbstractProductDao().create(b);
    return b;
}

@Override
    public ComponentTypeA createComponentTypeA(final String name){
    final ComponentTypeA a = new ComponentTypeA();
    a.setTitle(name);
    a.setComponentType(AbstractComponent.ComponentType.COMPONENT_TYPE_A);
    getAbstractComponentDao().create(a);
    return a;
}
@Override
public ComponentTypeA createComponentTypeA(final String title,
    final Integer componentCode, final Boolean active, final RawMaterial rawMaterial,
    final Float amount, final Integer layer){

    final RawMaterial m = getRawMaterialDao().retrieve(rawMaterial.getId());
    final ComponentTypeA a = new ComponentTypeA();
    a.setTitle(title);
    a.setComponentType(AbstractComponent.ComponentType.COMPONENT_TYPE_A);
    a.setActive(active);
    a.setComponentCode(componentCode);
    a.setRawMaterial(m);
```

(jatkuu)

LIITE 18. (jatkoa)

```
    a.setRawMaterialAmount(amount);
    a.setRawMaterialLayer(layer);
    getAbstractComponentDao().create(a);
    return a;
}
@Override
    public ComponentTypeB createComponentTypeB(final String title,
           final Integer componentCode, final Boolean active, final Integer partNumber,
           final Float partValue, final Integer partPcs){
    final ComponentTypeB b = new ComponentTypeB();
    b.setTitle(title);
    b.setComponentType(AbstractComponent.ComponentType.COMPONENT_TYPE_B);
    b.setActive(active);
    b.setComponentCode(componentCode);
    b.setPartNumber(partNumber);
    b.setPartValue(partValue);
    b.setPartPCS(partPcs);
    getAbstractComponentDao().create(b);
    return b;
    }
@Override
public ProductTypeB addComponent(final ProductTypeB prod, final ComponentTypeA type){
    ProductTypeB p = (ProductTypeB) getAbstractProductDao().findById(prod.getId());
    p.addComponents(type);
    getProductTypeBDao().update(p);
    return p;
}
@Override
public ProductTypeB addAbstractComponent(final ProductTypeB product,
    final AbstractComponent component){
    ProductTypeB p = (ProductTypeB) getAbstractProductDao().findById(product.getId());
    AbstractComponent a = getAbstractComponentDao().findById(component.getId());
    System.out.println(a.getComponentProducts().size());
    System.out.println(p.getComponents().size());
    p.addComponents(a);
```

```
    p.setComponentContent();
    getAbstractProductDao().update(p);
    return p;
}
@Override
public ProductTypeB removeComponentById(final ProductTypeB prod,
    final AbstractComponent comp){
    ProductTypeB p = (ProductTypeB) getAbstractProductDao().findById(prod.getId());
    p.removeComponentById(comp.getId());
    getAbstractProductDao().update(p);
    return p;
}
@Override
    public List<AbstractProduct> findByActiveProduct(final Boolean active) {
     return getAbstractProductDao().findByActive(active);
}
@Override
public List<AbstractProduct> findByProductName(final String name){
    return getAbstractProductDao().findByName(name);
}
@Override
public List<AbstractComponent> findByComponentTitle(final String title){
    return getAbstractComponentDao().findByTitle(title);
}
@Override
public Set<AbstractProduct> findProductByComponent(final Long id){
    final AbstractComponent a = getAbstractComponentDao().findById(id);
    System.out.println(a.getComponentProducts().size());
    return a.getComponentProducts();
}
```

(jatkuu)

LIITE 18. (jatkoa)

```
    @Override
    public RawMaterial createRawMaterial(final String name, final Integer code,
            final RawMaterialType type,final Identification id) {

            final List<RawMaterial> checkExists =
                    getRawMaterialDao().findByRawMaterialCode(code);

    if (checkExists.size() == 0){
            final RawMaterial rm = new RawMaterial(name,code,type);
            rm.setIdentification(id);
            rm.setActive(true);
            return getRawMaterialDao().createRawMaterial(rm);
    }
    else throw new RawMaterialAlreadyExists(code);
}

@Override
public List<RawMaterial> findRawMaterialByName(final String name){
    return getRawMaterialDao().findByRawMaterialName(name);
}
@Override
public List<RawMaterial> findRawMaterialByActiveStatus(final Boolean active){
    return getRawMaterialDao().findRawMaterialByActiveStatus(active);
}
@Override
    public ProductTypeA updateProductTypeARawMaterial(final ProductTypeA productTypeA,
            final RawMaterial rawMaterial){
    ProductTypeA p =
            (ProductTypeA) getAbstractProductDao().findById(productTypeA.getId());
    p.setRawMaterial(rawMaterial);
    getAbstractProductDao().update(p);
    return p;
}
@Override
public AbstractProduct updateProduct(final AbstractProduct abstractProduct){
    return getAbstractProductDao().update(abstractProduct);
}
@Override
public AbstractProduct findAbstractProductByProductCode(Integer productCode){
    List<AbstractProduct> abstractProductList =
            getAbstractProductDao().findByProductCode(productCode);

    if (abstractProductList.size() == 1) return abstractProductList.get(0);
    if (abstractProductList.size() > 1)
            throw new SeveralAbstractProductExistSameCode(productCode);
```

```
        throw new AbstractProductNotFound(productCode);
    }
    @Override
    public  AbstractComponent  findAbstractComponentByComponentCode(Integer  componentCode){

        List<AbstractComponent> abstractComponentList =
                getAbstractComponentDao().findByComponentCode(componentCode);
        System.out.println(abstractComponentList.size());
        if (abstractComponentList.size() == 1) return abstractComponentList.get(0);
        if (abstractComponentList.size() > 1)
                throw new SeveralAbstractProductExistSameCode(componentCode);

        throw new AbstractComponentCodeNotFound(componentCode);
    }
    @Override
    public AbstractProduct findAbstractPoductById(final long id){
        if (getAbstractProductDao().findById(id) == null)
                throw new EntityDoesNotExist(AbstractProduct.class, id);
        return getAbstractProductDao().findById(id);
    }
```

(jatkuu)

LIITE 18. (jatkoa)

```
 @Override
    public RawMaterial findByRawMaterialCode(final Integer code){
           final List<RawMaterial> rawMaterialList  =
                getRawMaterialDao().findByRawMaterialCode(code);

           if (rawMaterialList.size() == 1) return rawMaterialList.get(0);
           if (rawMaterialList.size() > 1) throw new SeveralRawMaterialExistSameCode(code);
           throw new AbstractProductNotFound(code);
   }

}
```

LIITE 19. FacadeBeanTest-luokka

```
package serviceTest;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertFalse;
import static org.junit.Assert.assertTrue;
import java.util.Hashtable;
import java.util.List;
import java.util.Set;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;
import classification.ProductTypeAClassification;
import classification.ProductTypeBClassification;
import service.FacadeBean;
import service.FacadeBeanRemote;
import entity.AbstractProduct;
import entity.ComponentTypeB;
import entity.RawMaterial.RawMaterialType;
import exception.AbstractComponentNotFound;
import exception.EntityDoesNotExist;
import exception.RawMaterialAlreadyExists;
import entity.ProductTypeA;
import entity.ProductTypeB;
import entity.ComponentTypeA;
import entity.AbstractComponent;
import entity.Identification;

public class FacadeBeanTest  {
    private static final long ID_FOOL = -554234333l;

    public static final Identification IDENTIFICATION_0000 =
        new Identification("0000","Own Product");
    public static final Identification IDENTIFICATION_0001 =
        new Identification("0001","Vendor_0001");
    public static final Identification IDENTIFICATION_0002 =
        new Identification("0002","Vendor_0002");
    public static final Identification IDENTIFICATION_0003 =
        new Identification("0003","Vendor_0003");
    public static final Identification IDENTIFICATION_0004 =
        new Identification("0004","Vendor_0004");
    public static final Identification IDENTIFICATION_0005 =
        new Identification("0005","Vendor_0005");
    public static final Identification IDENTIFICATION_0006 =
        new Identification("0006","Vendor_0006");
    public static final Identification IDENTIFICATION_0007 =
        new Identification("0007","Vendor_0007");
    public static final Identification IDENTIFICATION_0008 =
        new Identification("0008","Vendor_0008");
    public static final Identification IDENTIFICATION_0009 =
        new Identification("0009","Vendor_0009");
    public static final Identification IDENTIFICATION_0010 =
        new Identification("0009","Vendor_0010");

    public static final Integer RAW_MATERIAL_CODE_01 = 1;
    public static final Integer RAW_MATERIAL_CODE_02 = 2;
    public static final Integer RAW_MATERIAL_CODE_03 = 3;
    public static final Integer RAW_MATERIAL_CODE_04 = 4;
    public static final Integer RAW_MATERIAL_CODE_05 = 5;
    public static final Integer RAW_MATERIAL_CODE_06 = 6;
    public static final Integer RAW_MATERIAL_CODE_07 = 7;
    public static final Integer RAW_MATERIAL_CODE_08 = 8;
    public static final Integer RAW_MATERIAL_CODE_09 = 9;
    public static final Integer RAW_MATERIAL_CODE_10 = 10;

    public static final Integer PRODUCT_CODE_0001 = 1;
    public static final Integer PRODUCT_CODE_0002 = 2;
    public static final Integer PRODUCT_CODE_0003 = 3;
    public static final Integer PRODUCT_CODE_0004 = 4;
    public static final Integer PRODUCT_CODE_0005 = 5;
    public static final Integer PRODUCT_CODE_0006 = 6;
    public static final Integer PRODUCT_CODE_0007 = 7;
    public static final Integer PRODUCT_CODE_0008 = 8;

    public static final Integer COMPONENT_CODE_0001 = 1;
    public static final Integer COMPONENT_CODE_0002 = 2;
    public static final Integer COMPONENT_CODE_0003 = 3;
    public static final Integer COMPONENT_CODE_0004 = 4;
```

```
    public static final Integer COMPONENT_CODE_0005 = 5;
    public static final Integer COMPONENT_CODE_0006 = 6;
    public static final Integer COMPONENT_CODE_0007 = 7;
    public static final Integer COMPONENT_CODE_0008 = 8;
    public static final Integer COMPONENT_CODE_0009 = 9;
    public static final Integer COMPONENT_CODE_0010 = 10;
    public static final Integer COMPONENT_CODE_0011 = 11;
    public static final Integer COMPONENT_NOT_EXISTS = -999;

    private FacadeBeanRemote facade;

    public void printAbstractProduct(AbstractProduct abstractProduct)
    {
final AbstractProduct prod =
 facade.findAbstractProductByProductCode(abstractProduct.getProductCode());

       if (prod instanceof ProductTypeA) {
              ProductTypeA a = (ProductTypeA) prod;
              System.out.println("PRODUCT_TYPE_A");
              System.out.println("PRODUCT_NAME = " + a.getName());
              System.out.println("PRODUCT_CODE = " + a.getProductCode());
              System.out.println("PRODUCT_ACTIVE = " + a.isActive());
              System.out.println("CREATE_DATE = " + a.getCreatedDate());
              System.out.println("PRODUCT_VENDOR_DESCRIPTION = " +
               a.getIdentification().getDescription());
              System.out.println("PRODUCT_VENDOR_CODE = " +
               a.getIdentification().getCode());
              System.out.println("PRODUCT_RAWMATERIAL_NAME = " +
               a.getRawMaterial().getRawMaterialName());
              System.out.println("PRODUCT_RAWMATERIAL_CODE = " +
               a.getRawMaterial().getIdentification().getCode());
              System.out.println("PRODUCT_RAWMATERIAL_DESCRPTION = " +
               a.getRawMaterial().getIdentification().getDescription());
              System.out.println("PRODUCT_STRING_VALUE = " + a.getStringValue());
              System.out.println("PRODUCT_INTEGER_VALUE = " + a.getIntegerValue());
              System.out.println("PRODUCT_FLOAT_VALUE = " + a.getFloatValue());
              System.out.println("PRODUCT_ENUM_VALUE = " + a.getEnumValue());
              }

              if (prod instanceof ProductTypeB) {
                     ProductTypeB b = (ProductTypeB) prod;
                     System.out.println("PRODUCT_TYPE_B");
                     System.out.println("PRODUCT_NAME = " + b.getName());
                     System.out.println("PRODUCT_CODE = " + b.getProductCode());
                     System.out.println("PRODUCT_ACTIVE = " + b.isActive());
                     System.out.println("CREATE_DATE = " + b.getCreatedDate());
                     System.out.println("PRODUCT_VENDOR_DESCRIPTION = " +
                      b.getIdentification().getDescription());
                     System.out.println("PRODUCT_VENDOR_CODE = " +
                      b.getIdentification().getCode());

                     final Set<AbstractComponent> list = b.getComponents();

                     for (AbstractComponent r : list) {

                            if (r instanceof ComponentTypeA) {

                            ComponentTypeA compA = (ComponentTypeA) r;
                            System.out.println("\t-----------------------------------");
                            System.out.println("\tCOMPONENT_TYPE_A = ");
                            System.out.println("\tCOMPONENT_TITLE = " +
                            compA.getTitle());
```

(jatkuu)

LIITE 19. (jatkoa)

```
                            System.out.println("\tCOMPONENT_TYPE = " +
                             compA.getComponentType());
                            System.out.println("\tCOMPONENT_CODE = " +
                             compA.getComponentCode());
                            System.out.println("\tCOMPONENT_RAWMATERIAL_NAME = " +
                             compA.getRawMaterial().getRawMaterialName());
                            System.out.println("\tCOMPONENT_RAWMATERIAL_CODE = " +
                             compA.getRawMaterial().getIdentification().getCode());
                            System.out.println("\tCOMPONENT_RAWMATERIAL_DESCRPTION = " +
                            compA.getRawMaterial().getIdentification().getDescription());
                            System.out.println("\tCOMPONENT_RAWMATERIAL_AMOUNT = " +
                             compA.getRawMaterialAmount());
                            System.out.println("\tCOMPONENT_RAWMATERIAL_LAYER = " +
                             compA.getRawMaterialLayer());
```

```
                        System.out.println("\t-----------------------------------");
                }
                if (r instanceof ComponentTypeB) {

                        ComponentTypeB compB = (ComponentTypeB) r;
                        System.out.println("\t-----------------------------------");
                        System.out.println("\tCOMPONENT_TYPE_B = ");
                        System.out.println("\tCOMPONENT_TITLE = " +
                         compB.getTitle());
                        System.out.println("\tCOMPONENT_TYPE = " +
                         compB.getComponentType());
                        System.out.println("\tCOMPONENT_CODE = " +
                         compB.getComponentCode());
                        System.out.println("\tCOMPONENT_PART_NUMBER = " +
                         compB.getPartNumber());
                        System.out.println("\tCOMPONENT_PART_VALUE = " +
                         compB.getPartValue());
                        System.out.println("\tCOMPONENT_PART_PCS = " +
                         compB.getPartPCS());
                        System.out.println("\t-----------------------------------");
                }
            }

            }
}

@Before
public void setupLibrary() {

    try
            {
            final Hashtable jndiProperties = new Hashtable();

            jndiProperties.put(Context.URL_PKG_PREFIXES, "org.jboss.ejb.client.naming");
            jndiProperties.put(InitialContext.INITIAL_CONTEXT_FACTORY,
             "org.jboss.naming.remote.client.InitialContextFactory");
            jndiProperties.put(InitialContext.PROVIDER_URL, "remote://localhost:4447");
            jndiProperties.put("jboss.naming.client.ejb.context", true);

            jndiProperties.put(Context.SECURITY_PRINCIPAL, "user");
            jndiProperties.put(Context.SECURITY_CREDENTIALS, "passwd");

            // create a context passing these properties
            //final Context context = new InitialContext(jndiProperties);

           // The app name is the application name of the deployed EJBs. This is
           // typically the ear name without the .ear suffix. However, the
           // application name could be overridden in the application.xml of the
           // EJB deployment on the server.
           // Since we haven't deployed the application as a .ear, the app name for us
           // will be an empty string

            final String appName = "";
```

(jatkuu)

LIITE 19. (jatkoa)

```
           // This is the module name of the deployed EJBs on the server. This is
           //typically the jar name of the
           // EJB deployment, without the .jar suffix, but can be overridden via the
           //ejb-jar.xml. In this example, we have deployed the EJBs in a jboss-as-ejb-
           // remote-app.jar, so the module name is jboss-as-ejb-remote-app

           final String moduleName = "miniappEJB";

           // AS7 allows each deployment to have an (optional) distinct name. We
           // haven't specified a distinct name for our EJB deployment, so this is an
           // empty string

           final String distinctName = "";

           // The EJB name which by default is the simple class name of the bean
           //implementation class

            final String beanName = FacadeBean.class.getSimpleName();
           // the remote view fully qualified class name
           final String viewClassName = FacadeBeanRemote.class.getName();

           // let's do the lookup
           facade = (FacadeBeanRemote) context.lookup("ejb:" + appName + "/" +
                  moduleName + "/" + distinctName + "/" + beanName + "!" +
                  viewClassName);
```

```
                } catch (NamingException e)
                {
                        e.printStackTrace();
                        /*
                         * Runtimeexception is rethrow because there is no need to continue
                           if exception happens.
                         */
                        throw new RuntimeException(e);
                }
    }

    @BeforeClass
    public static void initContainer() {
    }

    // Note, the following method is unchanged
    @BeforeClass
    public static void setupDates() {

    }

 /*
    @After
    public void cleanupDatabase() {
        EntityManagerFactory f = Persistence.createEntityManagerFactory("miniappEJB");
        EntityManager em = f.createEntityManager();
        em.getTransaction().begin();
        em.createNativeQuery("delete                                                                    from
abstractcomponent_abstractproduct").executeUpdate();
        em.createNativeQuery("delete from componenttypeb").executeUpdate();
        em.createNativeQuery("delete from componenttypea").executeUpdate();
        em.createNativeQuery("delete from abstractproduct").executeUpdate();
        em.createNativeQuery("delete from producttypea").executeUpdate();
        em.createNativeQuery("delete from producttypeb").executeUpdate();
        em.createNativeQuery("delete from abstractproduct").executeUpdate();
        em.createNativeQuery("delete from rawmaterial").executeUpdate();
        em.getTransaction().commit();
    }
 */
```

(jatkuu)

LIITE 19. (jatkoa)

```
@Test
    public void _addRawMaterials(){
                facade.createRawMaterial("Stainless Steel", RAW_MATERIAL_CODE_01,
                        RawMaterialType.RAWMATERIAL_TYPE_A,IDENTIFICATION_0001);
                facade.createRawMaterial("Red LED", RAW_MATERIAL_CODE_02,
                        RawMaterialType.RAWMATERIAL_TYPE_A,IDENTIFICATION_0002);
                facade.createRawMaterial("1k resistor 0.25w", RAW_MATERIAL_CODE_03,
                        RawMaterialType.RAWMATERIAL_TYPE_A,IDENTIFICATION_0003);
                facade.createRawMaterial("Plastic enclosure", RAW_MATERIAL_CODE_04,
                        RawMaterialType.RAWMATERIAL_TYPE_A,IDENTIFICATION_0004);
                facade.createRawMaterial("Fiber yarn 1200TEX", RAW_MATERIAL_CODE_05,
                        RawMaterialType.RAWMATERIAL_TYPE_A,IDENTIFICATION_0005);
                facade.createRawMaterial("Fiber yarn 2400TEX", RAW_MATERIAL_CODE_06,
                        RawMaterialType.RAWMATERIAL_TYPE_A,IDENTIFICATION_0006);
                facade.createRawMaterial("Fiber yarn 0600TEX", RAW_MATERIAL_CODE_07,
                        RawMaterialType.RAWMATERIAL_TYPE_A,IDENTIFICATION_0007);
                facade.createRawMaterial("Fiber yarn 2400TEX", RAW_MATERIAL_CODE_08,
                        RawMaterialType.RAWMATERIAL_TYPE_A,IDENTIFICATION_0008);
                facade.createRawMaterial("Stitch yarn", RAW_MATERIAL_CODE_09,
                        RawMaterialType.RAWMATERIAL_TYPE_A,IDENTIFICATION_0009);
                facade.createRawMaterial("Packing material", RAW_MATERIAL_CODE_10,
                        RawMaterialType.RAWMATERIAL_TYPE_A,IDENTIFICATION_0010);

        try{
             assertTrue(facade.findRawMaterialByActiveStatus(true).size() == 10);
             assertTrue(facade.findRawMaterialByActiveStatus(false).size() == 0);
        } finally { }
     }

    @Test(expected = RawMaterialAlreadyExists .class)
    public void addMaterial_0001_again(){
        try{
                facade.createRawMaterial("Stainless Steel", RAW_MATERIAL_CODE_01,
                        RawMaterialType.RAWMATERIAL_TYPE_A,IDENTIFICATION_0001);
        } finally {}
    }

    @Test
    public void _addProductA(){
        final ProductTypeA productA = facade.createProductTypeA("725-1",PRODUCT_CODE_0001,
                IDENTIFICATION_0000, true);

                productA.setRawMaterial(facade.findByRawMaterialCode(RAW_MATERIAL_CODE_01));
                productA.setIntegerValue(1000);
                productA.setStringValue("Final Product");
                facade.updateProduct(productA);

        try{
                final AbstractProduct found =
                        facade.findAbstractPoductById(productA.getId());
             assertTrue(found instanceof ProductTypeA);
             assertTrue(((ProductTypeA) found).getRawMaterial().
                equals(facade.findByRawMaterialCode(RAW_MATERIAL_CODE_01)));
             ProductTypeAClassification classification = (ProductTypeAClassification)
                ((ProductTypeA) found).getClassification();
             assertTrue(classification.getEnumType() ==
                ProductTypeAClassification.EnumType.TYPE_A);
        } finally { }
        printAbstractProduct(productA);
    }

    @Test
    public void _addProductB_ComponentTypeA(){
        final ProductTypeB productB =
                facade.createProductTypeB("Led lamp",PRODUCT_CODE_0002, IDENTIFICATION_0000,
                true);
```

(jatkuu)

LIITE 19. (jatkoa)

```
        final ComponentTypeA c1 =
                facade.createComponentTypeA("Red       LED",COMPONENT_CODE_0001,       true,
                facade.findByRawMaterialCode(RAW_MATERIAL_CODE_02), new Float(1), 1);

        final ComponentTypeA c2 = facade.createComponentTypeA("1k resistor 0.25w",
                COMPONENT_CODE_0002,                                                    true,
                facade.findByRawMaterialCode(RAW_MATERIAL_CODE_03), new Float(1), 1);

        final ComponentTypeA c3 = facade.createComponentTypeA("Plastic enclosure",
                COMPONENT_CODE_0003,                                                    true,
                facade.findByRawMaterialCode(RAW_MATERIAL_CODE_04), new Float(1), 1);

        facade.addAbstractComponent(productB, c1);
        facade.addAbstractComponent(productB, c2);
        facade.addAbstractComponent(productB, c3);

        try{
                final AbstractProduct found =
                 facade.findAbstractPoductById(productB.getId());
            assertTrue(found instanceof ProductTypeB);
            assertEquals(((ProductTypeB) found).getComponents().size(),3);
            ProductTypeBClassification classification =
               (ProductTypeBClassification) ((ProductTypeB) found).getClassification();
            assertTrue(classification.getComponentContent() ==
               ProductTypeBClassification.ComponentContent.INCLUDE_A);
            assertFalse(classification.getComponentContent() ==
               ProductTypeBClassification.ComponentContent.INCLUDE_B);
            assertFalse(classification.getComponentContent()                            ==
            ProductTypeBClassification.ComponentContent.INCLUDE_BOTH);
        } finally { }
        printAbstractProduct(productB);
    }

    @Test
    public void _addProductB_ComponentTypeB(){
        final ProductTypeB productB = facade.createProductTypeB("Xenon lamp",
                PRODUCT_CODE_0007, IDENTIFICATION_0000, true);

        final ComponentTypeB c1 = facade.createComponentTypeB("Xenon 50W",
                COMPONENT_CODE_0011, true, 10101, new Float(50), 1);

        facade.addAbstractComponent(productB, c1);

        try{
                final AbstractProduct found =
                facade.findAbstractPoductById(productB.getId());
            assertTrue(found instanceof ProductTypeB);
            assertEquals(((ProductTypeB) found).getComponents().size(),1);
            ProductTypeBClassification classification =
               (ProductTypeBClassification) ((ProductTypeB) found).getClassification();
            assertTrue(classification.getComponentContent() ==
               ProductTypeBClassification.ComponentContent.INCLUDE_B);
            assertFalse(classification.getComponentContent() ==
               ProductTypeBClassification.ComponentContent.INCLUDE_A);
            assertFalse(classification.getComponentContent() ==
                 ProductTypeBClassification.ComponentContent.INCLUDE_BOTH);
        } finally { }
        printAbstractProduct(productB);
    }

    @Test
    public void _addProductB_ComponentTypeA_ForComponentRemove(){
        final ProductTypeB productB = facade.createProductTypeB("GlassFiber Issue ver 1",
                PRODUCT_CODE_0003, IDENTIFICATION_0000, true);

        final ComponentTypeA c1 = facade.createComponentTypeA("Fiber yarn 1200TEX",
                COMPONENT_CODE_0004,                                                    true,
        facade.findByRawMaterialCode(RAW_MATERIAL_CODE_05), new Float(100), 1);
```

(jatkuu)

LIITE 19. (jatkoa)

```
        final ComponentTypeA c2 = facade.createComponentTypeA("Fiber yarn 2400TEX",
        COMPONENT_CODE_0005,  true,  facade.findByRawMaterialCode(RAW_MATERIAL_CODE_06),  new
        Float(200), 2);

        final ComponentTypeA c3 = facade.createComponentTypeA("Fiber yarn 0600TEX",
        COMPONENT_CODE_0006,  true,  facade.findByRawMaterialCode(RAW_MATERIAL_CODE_07),  new
        Float(200), 3);

        final ComponentTypeA c4 = facade.createComponentTypeA("Fiber yarn 2400TEX",
        COMPONENT_CODE_0007,  true,  facade.findByRawMaterialCode(RAW_MATERIAL_CODE_08),  new
        Float(100), 4);

        final ComponentTypeA c5 = facade.createComponentTypeA("Stitch yarn",
        COMPONENT_CODE_0008,  true,  facade.findByRawMaterialCode(RAW_MATERIAL_CODE_09),  new
        Float(5), 5);

        facade.addAbstractComponent(productB, c1);
        facade.addAbstractComponent(productB, c2);
        facade.addAbstractComponent(productB, c3);
        facade.addAbstractComponent(productB, c4);
        facade.addAbstractComponent(productB, c5);

        try{
                final AbstractProduct found =
                facade.findAbstractPoductById(productB.getId());
            assertTrue(found instanceof ProductTypeB);
            assertEquals(((ProductTypeB) found).getComponents().size(),5);
        } finally { }
        printAbstractProduct(productB);
    }

    @Test
    public void _removeComponentFromProductB_ComponentTypeA(){
        final ProductTypeB productB =
                (ProductTypeB) facade.findAbstractProductByProductCode(PRODUCT_CODE_0003);
        final Integer removedComponent = COMPONENT_CODE_0006;

        Set<AbstractComponent> components = productB.getComponents();

        for (AbstractComponent r : components) {

                if (r.getComponentCode() == removedComponent)
                        {
                        facade.removeComponentById(productB, r);
                        }
        }
        try{
                final AbstractProduct found =
                        facade.findAbstractPoductById(productB.getId());
            assertTrue(found instanceof ProductTypeB);
            assertEquals(((ProductTypeB) found).getComponents().size(),4);
        } finally { }
        printAbstractProduct(productB);
    }

    @Test(expected = AbstractComponentNotFound.class)
    public void _componentNotFoundFromProductB(){
        final ProductTypeB productB =
                (ProductTypeB) facade.findAbstractProductByProductCode(PRODUCT_CODE_0003);
        productB.findComponentById(COMPONENT_NOT_EXISTS);
    }

    @Test
    public void _addSameComponentToSeveralProductB_ComponentTypeA(){
        final ComponentTypeA c =
        facade.createComponentTypeA("COMMON COMPONENT",COMPONENT_CODE_0009, true,
         facade.findByRawMaterialCode(RAW_MATERIAL_CODE_09), new Float(100), 4);
```

(jatkuu)

LIITE 19. (jatkoa)

```
        final ProductTypeB productB4 = facade.createProductTypeB("GlassFiber Issue ver 2",
        PRODUCT_CODE_0004, IDENTIFICATION_0000, true);

        final ProductTypeB productB5 = facade.createProductTypeB("GlassFiber Issue ver 3",
        PRODUCT_CODE_0005, IDENTIFICATION_0000, true);

        final ProductTypeB productB6 = facade.createProductTypeB("GlassFiber Issue ver 4",
        PRODUCT_CODE_0006, IDENTIFICATION_0000, true);

        facade.addAbstractComponent(productB4, c);
        facade.addAbstractComponent(productB5, c);
        facade.addAbstractComponent(productB6, c);
        }

    @Test
    public void _findProductsByComponentTitle() {
        final List<AbstractComponent> list =
                facade.findByComponentTitle("COMMON COMPONENT");
        System.out.println(list.size());

         for (AbstractComponent c : list) {
                System.out.println(c.getTitle());
                Set<AbstractProduct> prod = facade.findProductByComponent(c.getId());
                System.out.println(prod);
                System.out.println(prod.size());
                for (AbstractProduct p : prod) {
                        System.out.println("\t" + p.getName());
                }
       }
         try{
             assertEquals(list.size(),1);
        } finally { }

    }

    @Test
    public void _findProductsByComponentID() {

        final AbstractComponent a =
                facade.findAbstractComponentByComponentCode(COMPONENT_CODE_0009);
        System.out.println(a);
        System.out.println(a.getTitle());
        Set<AbstractProduct> prod = facade.findProductByComponent(a.getId());
        System.out.println(prod);
        System.out.println(prod.size());

        for (AbstractProduct p : prod) {
                System.out.println("\t" + p.getName());
        }
        try{
             assertTrue(a instanceof ComponentTypeA);
             assertEquals(prod.size(),3);
        } finally { }
    }

    @Test(expected = EntityDoesNotExist.class)
    public void _lookupAbstractProductThatDoesNotExist() {
        facade.findAbstractPoductById(ID_FOOL);
    }

    @Test
    public void _addComponentTypeB() {
        facade.createComponentTypeB("Component typeB", COMPONENT_CODE_0010, true, 1,
        new Float(1), 1);
    }
```

(jatkuu)

LIITE 19. (jatkoa)

```
    @Test
    public void _addExistingProductBExistingAbstractComponent() {

        final AbstractComponent a =
f               acade.findAbstractComponentByComponentCode(COMPONENT_CODE_0010);
        List<AbstractProduct> productList = facade.findByProductName("Led lamp");

        try{
             assertEquals(productList.size(),1);
             ProductTypeB product = (ProductTypeB) productList.get(0);
             final int orginalComponentCount = product.getComponents().size();
             facade.addAbstractComponent(product, a);
             final ProductTypeB loaded = (ProductTypeB)
                facade.findAbstractProductByProductCode(product.getProductCode());
             assertTrue(loaded.getComponents().size() == (orginalComponentCount+1));

        } finally { }
    }

    @Test
    public void _createProductBWithComponentAandB()
    {
        final ProductTypeB b = facade.createProductTypeB("Compound Product",
        PRODUCT_CODE_0008, IDENTIFICATION_0000, true);

        final ComponentTypeA c1 =
        (ComponentTypeA) facade.findAbstractComponentByComponentCode(COMPONENT_CODE_0003);

        final ComponentTypeB c2 =
        (ComponentTypeB) facade.findAbstractComponentByComponentCode(COMPONENT_CODE_0010);

        ProductTypeB t = facade.addAbstractComponent(b, c1);

        t = facade.addAbstractComponent(b, c2);

        try{
               final AbstractProduct found = facade.findAbstractPoductById(b.getId());
             assertTrue(found instanceof ProductTypeB);
             assertEquals(((ProductTypeB) found).getComponents().size(),2);
             ProductTypeBClassification classification =
                (ProductTypeBClassification) ((ProductTypeB) found).getClassification();
             assertTrue(classification.getComponentContent() ==
                ProductTypeBClassification.ComponentContent.INCLUDE_BOTH);
             assertFalse(classification.getComponentContent() ==
                ProductTypeBClassification.ComponentContent.INCLUDE_A);
             assertFalse(classification.getComponentContent() ==
                ProductTypeBClassification.ComponentContent.INCLUDE_B);
        } finally { }

        printAbstractProduct(t);
    }

    /*
   private RawMaterial getRawMaterialByCode(Integer code)
   {
        List<RawMaterial> rawMaterialList = library.findByRawMaterialCode(code);

        RawMaterial found = null;

        if (rawMaterialList.size() == 1) found = rawMaterialList.get(0);
        if (rawMaterialList.size() > 1) throw new SeveralRawMaterialExistSameCode(code);

        return found;
   }
*/

}
```