jamk.fi

Keltainen market -ohjelmiston kehittäminen Java EE:llä

Antti-Pekka Tanskanen

Opinnäytetyö
Toukokuu 2015

Ohjelmistotekniikan koulutusohjelma
Tekniikan ja liikenteen ala

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Kuvailulehti

Tekijä(t)	Julkaisun laji	Päivämäärä
Tanskanen, Antti-Pekka	Opinnäytetyö	20.05.2015
	Sivumäärä 55	Julkaisun kieli Suomi
		Verkkojulkaisulupa myönnetty: x

Työn nimi
Keltainen market -ohjelmiston kehittäminen Java EE:llä

Koulutusohjelma
Ohjelmistotekniikan koulutusohjelma

Työn ohjaaja(t)
Olli Väänänen

Toimeksiantaja(t)
Pertti Laurila

Tiivistelmä

Opinnäytetyössä kuvataan yhden sovelluksen suunnitelmat, ohjelmoinnin tekniikat ja ohjelmointiprosessi. Tutkimuksella pyrittiin sovelluksen kehittämisen ohella selvittämään yksittäisen henkilön mahdollisuuksia kehittää laaja sovellus Java EE:n teknologioilla ja lisätä ohjelmoijan ja/tai organisaation tietämystä siihen liittyvistä teknologioista sekä prosessista.

Tutkimuksen menetelmänä oli tapaustutkimus, jossa tutkijana ja ohjelmoijana toimi sama henkilö. Tutkittavaan organisaatioon kuuluivat toimeksiantaja/keksijä ja ohjelmoija, jotka aikovat perustaa yrityksen sovelluksen ympärille. Sovelluksen ohjelmointi aloitettiin 1.1.2015 ja sitä ohjelmoitiin - neljä päivää viikossa - noin viiden kuukauden ajan.

Tuloksena on saatu toimeksiantajan idean mukainen sovellus ja lisätty ohjelmoijan/organisaation tietämystä Java EE:n teknologioista ja ohjelmointiprosessista. Huomattiin, että vesiputousprosessi toimii hyvin, jos sovelluksen vaatimukset ovat ennalta erittäin tarkasti tiedossa.

Työtä voi käyttää tukimateriaalina Java EE:tä opiskellessa.

Avainsanat (asiasanat)
Java EE, JSF, JPA, EJB

Muut tiedot
Liitteenä oleva vaatimustenmäärittely poistetaan julkaistavasta versiosta, kuten myös sovelluksen toimintaa kuvaavat osuudet.

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

Description

Author(s)	Type of publication	Date
Tanskanen, Antti-Pekka	Bachelor's thesis	20.05.2015
		Language of publication: Finnish
	Number of pages 55	Permission for web publication: x

Title of publication
Developing Keltainen Market software with Java EE

Degree programme
Software Engineering

Tutor(s)
Väänänen, Olli

Assigned by
Laurila, Pertti

Abstract

This bachelor's thesis illustrates the design, chosen technologies, and process of a piece of software. The objective of the thesis was to code a piece of software, study the prospects of one person coding a large-scale software with Java EE – technologies, and to augment the knowledge of the organization and/or the coder of the chosen technologies and process.

The chosen method was action research in which the researcher and coder were the same person. The organisation under the research was the assigner/inventor and the coder, who builds a start up around the piece of software. The coding was started on 1.1.2015, and it was coded – four days in a week – for about five months. As a result, the software based on assigner's idea was coded. The organisation's/coder's knowledge of Java EE technologies and software development process were also augmented. It was noted that waterfall model functions well, should the requirements be exactly known.

The thesis can be used as a support material in studying Java EE -technologies.

Keywords/tags (subjects)
Java EE, JSF, JPA, EJB

Miscellaneous
Attached requirements specification is removed from the published version, such as other parts, which describe the functionality of the program.

SISÄLLYS

TERMIT JA LYHENTEET

CDI	Context & Dependecy Injection
DAO	Data Access Object
EJB	Enterprise Java Bean
GWT	Google Web Toolkit
ID	Identifier
IE-malli	Information Engineering SSADM notaation mukainen malli
Java EE	Java Enterprise Edition
JPA	Java Persistence API
JPQL	Java Persistence Query Language
JSF	Java Server Faces
JVM	Java Virtual Machine
MVC	Model View Controller
ORM	Object Relational Mapping
SSL	Secure Sockets Layer
SQL	Structured Query Language
URL	Uniform Resource Locator
UUID	Universally Unique Identifier
XML	Extensive Markup Language

1 TYÖN LÄHTÖKOHDAT

Osa työhön liittyvästä tiedosta on salaista ja tekstistä puuttuu salaiset kohdat.

1.1 Tehtävä ja tausta

Toimeksiantajalla (Pertti Laurila) oli yritysidea, jota lähdettiin kartoittamaan ja suunnittelemaan. Idea koski digitaalisen ja perinteisen kaupan yhdistämistä tietojärjestelmällä. Idean jalostaminen sovellukseksi sopi hyvin opiskelun tavoitteisiin, ja oli juuri sitä, millaista piti oppia tekemään. Tästä syystä ideaa oli hyvä viedä eteenpäin Jyväskylän ammattikorkeakoulun opintojaksoilla.

Tässä työssä esitellään ohjelmiston suunnitteluputki ideasta ylläpitovaiheeseen: ohjelmiston suunnitelma, sen toteutuminen käytännössä ja teknologioita sekä teknologioiden valintoja. Työssä esitellään kehitetyn ohjelmiston kehittämisprosessi ja arkkitehtuuri eri mallien valossa. Lisäksi käsitellään myös joidenkin keskeisten sovelluksen osien implementaatioita eri teknologioilla – erityisesti on pyritty käsittelemään kohtia, joita ei ollut esitetty kirjallisuudessa, tai niitä oli käsitelty suppeasti.

Tarkoitus on esittää pääkohdiltaan yhden hengen organisaation ohjelmistoprojektin eteneminen, arkkitehtuuri ja teknologiat. Joitakin yksityiskohtia on jätetty pois, mutta niihin löytyy ratkaisu hakukonetta käyttämällä.

Tutkimuksen menetelmänä oli tapaustutkimus, jolla pyritään sovelluksen kehittämisen ohella selvittämään yksittäisen henkilön mahdollisuuksia kehittää laaja sovellus Java EE:n teknologioilla, lisätä tietämystä siihen liittyvien teknologioiden osalta ja kehittyä ohjelmistoprosessien asiantuntijana. Tutkimuksen tekijä toimii sekä tutkijan että kehitettävän organisaation roolissa.

Ensisijainen tavoite oli saada kehitettyä valmis Java EE -sovellus tehtyjen suunnitelmien pohjalta. Tarkoituksena oli siis oppia kehittämään laajoja web-sovelluksia ja luoda työkalupakki, jonka avulla olisi helppo suunnitella ja kehittää sovelluksia omassa yrityksessä. Työkalupakin työkalujen kerääminen alkoi jo ennen Jyväskylän ammattikorkeakouluun hakeutumista Janne Kuhan (2008) kirjaa luettaessa, ja kyseisten teknologioiden osaajana kehittyminen on ollut opiskelun tavoitteena.

1.2 Eteneminen

Sovelluksen kehittäminen alkoi opintojaksolla Database Design, jossa päätettiin aloittaa toimeksiantajan idean työstämien tietokantasuunnitelmaksi. Ideasta oli tehty valmis määrittely, joka tarkentui tietokannan suunnittelun yhteydessä. Tehtyä suunnitelmaa on käytetty pohjana ohjelmistosuunnittelulle, joka toteutettiin ohjelmistosuunnittelun opintojaksolla.

Myöhemmin Java EE -ohjelmoinnin opintojaksolla opeteltiin teknologioita, joilla sovelluksen voisi toteuttaa. Pohjana oli Janne Kuhan (2008) kirjan arkkitehtuuri. Suurin oppi opintojaksolla oli se, miten EJB:t liitetään JSF:ään. Siinä vaiheessa käytettiin @ManagedBean-papuja, mutta tätä työtä tehdessä kyseinen teknologia oli vanhentunut ja käyttöön piti ottaa CDI-pavut. Näiden eroista on käyty paljon keskustelua eri palstoilla ja lopputulos on ollut se, että CDI-papuihin kannattaa siirtyä. Ne eivät ole enää osa JSF:ää, kuten @ManagedBean-pavut olivat. Tämän jälkeen tekeminen oli melko suoraviivaista, tosin validaattorien ja filttereiden kanssa oli ongelmia ja niitä varten jouduttiin ottamaan käyttöön OmniFaces laajennos.

Muilta osin tekeminen oli sitä, että katsottiin suunnitelmista, mitä halutaan ja toteutettiin. Hankalinta oli edellä mainittujen ongelmien lisäksi se, kun ohjelmoija ei tuntenut ennalta kaikkia PrimeFacesin komponentteja. Tästä syystä-

jouduttiin usein palaamaan korjaamaan käyttöliittymää, kun jonkin asian toteuttamiseen löytyi parempi komponentti.

2 OHJELMISTOJEN KEHITYSPROSESSEISTA

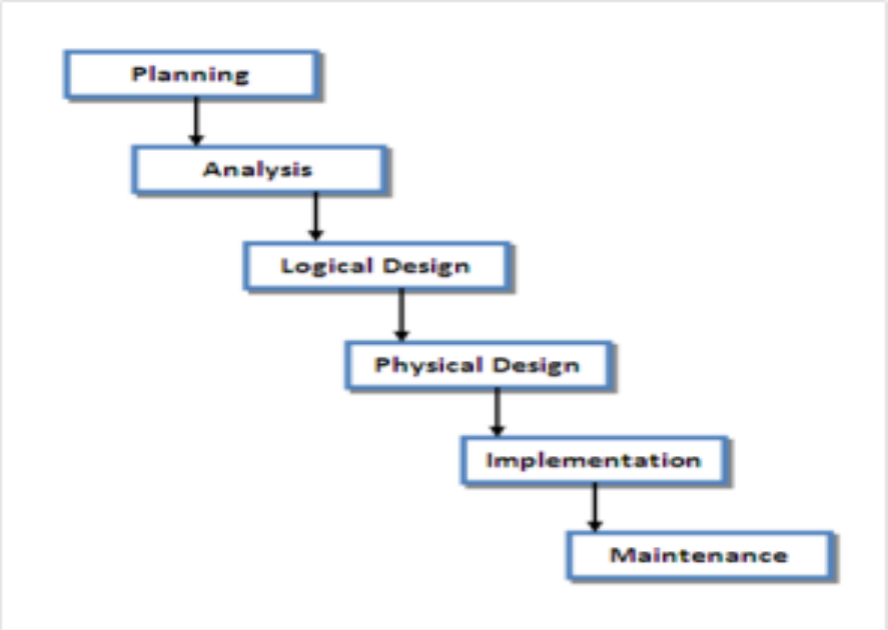
2.1 Yleistä

Ohjelmistojen kehittämiseen on olemassa monenlaisia prosesseja. Tämän opinnäytetyön kaltaiseen yhden ohjelmoijan projektiin voisi ajatella mielestäni vesiputousmallia tai jotain ketterää prosessia, jossa inkerementaalisesti - pyrähdyksittäin - kehitetään prototyyppiä yhteistyössä toimeksiantajan kanssa. Luvuissa 2.2–2.3 on esitelty vesiputousmalli ja ketterä malli. Tässä projektissa päätettiin edetä vesiputousmallin kaltaisesti, koska vaatimukset oli määritelty hyvin toimeksiantajan puolesta. Joitakin kertoja jouduttiin kuitenkin palaamaan vesiputouksessa myös ylöspäin, kun huomattiin vaatimuksia, joita ei ollut otettu huomioon suunnitelmassa.

2.2 Vesiputousmalli

Merkittävin vesiputousmallin heikkous on siinä, jos alussa suunnitellaan huonosti tai valitaan muuten huonosti suunnitteluvaiheessa. Suunnittelussa tehty virhe kallista korjata, koska se löytyy usein mahdollisesti vasta, kun järjestelmää aletaan ottaa käyttöön. Mitä suurempi projekti on, sitä suurempi riski huonolle suunnittelulle on olemassa. (Waterfall Software Development Model 2014.)

Vesiputousmallia (ks. Kuvio 1) tulisikin käyttää vain pienissä tai triviaaleissa projekteissa, joissa kaikki vaatimukset ovat tiedossa, ennen kuin projekti alkaa. Jos on olemassa pieninkin mahdollisuus, että projekti ei ole triviaali tai vaatimukset eivät ole täydelliset, vesiputousmallia ei tulisi käyttää. (Waterfall Software Development Model 2014.)



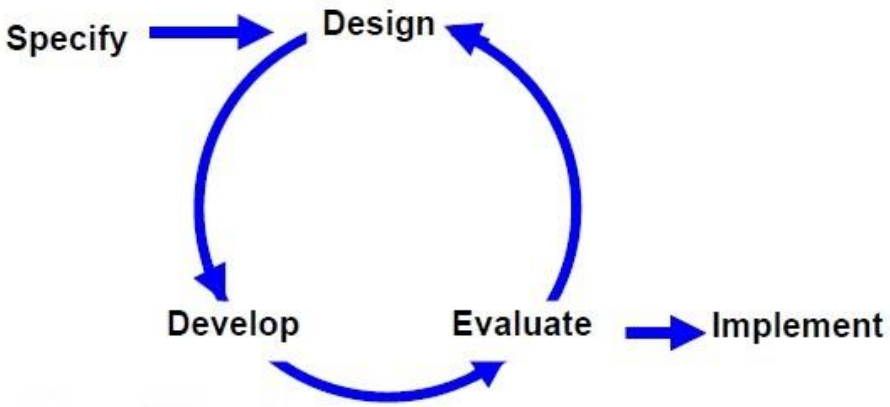
Kuvio 1 – Vesiputousmalli

2.3 Iteratiivinen ja inkrementaalinen malli

Inkrementeittäin kehittäminen (ks. Kuvio 2) toimii hyvin, jos vaatimukset eivät ole täysin selvillä. Vaatimukset voidaan muuntaa muuttuvan maailman mukaisesti. Iteratiivinen osa tarkoittaa sitä, että ohjelman laatua parannetaan asteittain, kun tiedetään, mikä tulee olemaan lopullista. (Cockburn 2008.)

Oman näkemykseni mukaan tietokantojen inkrementaalinen tapa on varmempi ja tässäkin projektissa on ollut ongelmia esimerkiksi sen kanssa, kun vaatimuksissa lukee jotain, mutta myöhemmin keskusteluissa kuitenkin halutaan hieman erilaista.

Voi kuitenkin viedä paljon aikaa, jos täytyy muokata esimerkiksi tietokannan rakennetta. Tämä ei ole ongelma silloin, jos työtä tehdään esimerkiksi tuntiveloituksella, mutta jokin vastuu pitäisi yrityksen/kehittäjän kuitenkin kantaa, ettei järjestelmän kehittämisen kustannuksia nosteta turhan takia.



Kuvio 2 – Inkrementaalinen malli

2.4 Muut mallit

Myös muita kehitysprosesseja on olemassa, kuten RUP, Six sigma, RAD, Lean ja spiraalimalli. Nämä kuitenkin vaikuttivat melko suurten organisaatioiden käyttöön suunnitelluilta, joten niitä ei tässä tarkemmin käsitellä.

3 SUUNNITTELU

3.1 Tietokanta

3.1.1 Suunnittelu

Tietokannan suunnittelun yhteydessä määräytyi myös koko ohjelman toiminta. Aluksi asiakkaan vaatimusten (toimeksiantajan suunnitelma**Virhe. Viitteen lähdettä ei löytynyt.** ja sähköpostikeskustelut) perusteella luotiin tietokannan vaatimusten määrittely. Tämän perusteella luotiin käsitemalli, josta transformoitiin suunnitteluputken mukaisesti normalisoitu IE–malli (Hovi, Huotari & Lahdenmäki 2005).

Suunnitteluputki koostuu viidestä vaiheesta: käsiteanalyysi, tarveanalyysi, normalisointi, tietokannan toteutus ja suorituskyvyn viritys. Näistä käsitemalli aloitetaan karkealta tasolta, jota pikkuhiljaa tarkennetaan. Sitä voi oliopohjaisissa järjestelmissä verrata ohjelman luokkakaavioon. Tarveanalyysissä käsitemallia tarkennetaan sovelluksen vaatimusten perusteella lisäten kaikki puuttuvat tiedot. Tämän jälkeen tarkennettu malli normalisoidaan. Yleisin käytettävä ratkaisu on kolmas normaalimuoto. Normalisoinnin tarkoituksena on poistaa turha toisteisuus, mutta tietokanta voidaan normalisoinnin avulla optimoida myös hakuja tai kirjoituksia varten. (Hovi ym. 2005.) Neljäs vaihe toteutuu ORM:n avulla, joka luo valitun tietokanta-ajurin avulla valittuun tietokantaan oliomallin mukaiset taulut.

Suunniteltavasta ohjelmasta oli olemassa oleva liiketoimintaominaisuuksia määrittelevä dokumentti, jonka lisäksi vaatimuksia tarkennettiin sähköpostitse toimeksiantajalta.

Aluksi tietokanta suunniteltiin käytettäväksi ilman ORM:a, mutta onneksi se toimii myös sen kanssa. Nähtävästi sama suunnittelumenetelmä soveltuu perinteisiin ja ORM–tietokantoihin.

3.1.2 Muutokset ohjelmoinnin edetessä

Ohjelmistoa kehittäessä huomattiin, että suunniteltu tietokanta ei mahdollista kaikkia toimintoja, kuten oli suunniteltu, ja siihen jouduttiin tekemään muutoksia.

Edelleen ohjelmaa kehittäessä mieleen tuli, että voisi olla kaiken varalta parempi käyttää tuotetaulussa yksilöidyn tietokanta-ID:n sijasta UUID:tä, joka saadaan generoitua Javan kirjastojen avulla. [tästä on poistettu tekstiä]

Myös person-taululle jouduttiin tekemään muutoksia, kun ohjelmaan haluttiin sähköpostiosoitteen vahvistusviesti. Tauluun lisättiin UUID-kenttä ja confirmed-kenttä. UUID:tä käytetään parametrina sähköpostilla lähetettävässä URL-osoitteessa, jota klikkaamalla käyttäjä aktivoituu. Confirmed-kenttä on boolean-arvo, joka on false, jos käyttäjä ei ole aktivoinut tunnustaan ja muuttuu true-arvoon, kun käyttäjä aktivoituu.

Suurin muutostyö tuli Category-taulun osalta, kun alkuperäisessä suunnitelmassa ei voinut olla samannimisiä alikategorioita esimerkiksi seuraavasti DVD->Musikki ja Blue-Ray->Musiikki, koska kategorian nimi oli taulun pääavain (primary key). Tämä ongelma ratkaistiin lisäämällä tauluun ID-kenttä, johon generoituu uniikki avain, johon sitten alikategoriat viittaavat. Se hankaloittaa hieman kategorioiden lisäämistä, mutta on välttämätön. [poistettu viittaus poistettuun kuvioon IE-mallista, johon on tehty tarvitut muutokset]

Tietokannan suunnitelman saaminen yhdellä kertaa täydelliseksi vaikuttaisi olevan melko hankalaa. Tämän osalta kehitysprosessi alkoi muistuttaa hieman ketteriä menetelmiä, sillä usein paljastui uusia vaatimuksia, joita ei ollut otettu

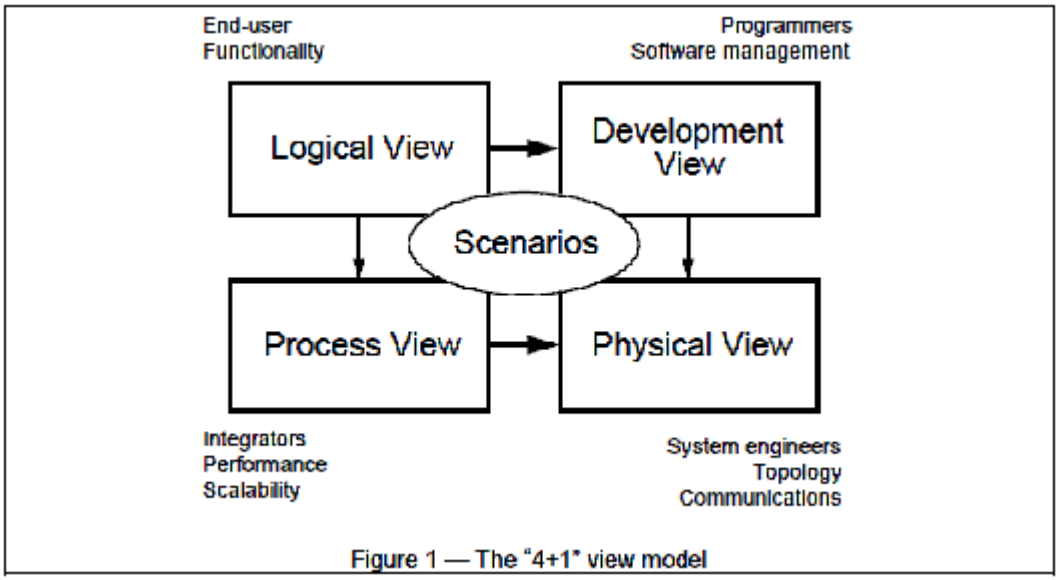
suunnitellessa huomioon. Toisaalta, jos ajatellaan vesiputousmallin mukaisesti, usein jouduttiin palaamaan edellisiin vaiheisiin. Olisikin mielenkiintoista tietää, kuinka valmiiksi pitkään tietokantoja suunnitelleet ammattilaiset saavat suunnitelman yhdellä kertaa, ja miten tietokannan suunnittelu kannattaa tehdä ketteriä menetelmiä käytettäessä.

Myös käyttöliittymän suunnitelmaa muutettiin, kun toimeksiantaja halusi oikeaan yläkulmaan valikon, jossa lukee käyttäjätunnus ja jota kautta voi tehdä toimintoja. Se oli muutenkin parempi vaihtoehto, sillä sovelluksen yläpalkin tila on rajallinen, ja esimerkiksi pääkäyttäjän palkki olisi tullut muuten täyteen.

Viimeisenä uutena vaatimuksena haluttiin vielä hakutoiminto tuotteille. Tämä toteutettiin osittain tietokantamoottorin avulla, mutta yleispätevää tapaa poistaa html-tagit sql-kentästä ei löydetty, joten se osa haluttiin tehdä Javalla Jsoup-kirjaston avulla, jotta tietokantatuotteen vaihtaminen onnistuu yhä pelkkää persistence.xml:ää muuttamalla. Myöhemmin, jos sovellusta tarvitsee optimoida, kirjoittamalla tietokantamoottorispesifinen funktio/stored procedure saattaa haku nopeutua.

3.2 Ohjelmiston suunnittelu

Ohjelmiston suunnitelma tehtiin 4+1 arkkitehtuurimallin mukaiseksi. Siitä monet eri projektiin osallistuvat henkilöt voivat selvittää itselleen sen, mitä haluavat ohjelmistosta tietää. Järjestelmän suunnittelijat (system engineer) käyttävät fyysistä näkymää (physical view). Loppukäyttäjät, asiakkaat ja tietokantaspesialistit käyttävät loogista näkymää (logical view). Projektipäälliköt käyttävät kehitysnäkymää (development view). (Kruchten 1995.) Kuvio 3 kuvaa 4+1 arkkitehtuurin.



Kuvio 3 – 4+1 arkkitehtuuri (Kruchten 1995)

Esimerkkinä loogisen näkymän kaavioista esitetään luokkanäkymä ja skenaarioista käyttötapaus kaavio. Luokkanäkymä on hyvin samankaltainen tietokantasuunnittelussa käytettyyn käsitemalliin verrattuna. Siihen on vain lisätty luokkien attribuutit ja metodit. Käyttötapauskaaviosta selviää, miten ohjelmaa on tarkoitus käyttää, ja millaisia käyttäjiä sillä on tarkoitus olla.

3.3 Ulkoasun suunnittelu

Sovelluksen ulkoasua suunniteltiin hieman jo ohjelmistosuunnittelun opintojaksolla. Siinä ei kuitenkaan ollut huomioitu läheskään kaikkia näkymiä, joita sovelluksessa tulisi olemaan. Ulkoasun pääkohdat suunniteltiin luonnostelemalla kynällä ja paperilla. Osittain edettiin luonnostelematta. Sovellusta pyrittiin rakentamaan mahdollisimman helppokäyttöiseksi, mutta todennäköisesti jotkin käyttöohjeet siihen täytyy kuitenkin vielä kirjoittaa.

Suunnittelupäätöksiä pyrittiin tekemään Steve Krugin (2006) ohjeiden mukaisesti noudattaen erityisesti hänen ensimmäistä neuvoaan siitä, että käyttäjän tarvitsisi ajatella mahdollisimman vähän sovellusta käyttäessään. [poistettu viittaus liitteeseen, joka on poistettu]

4 ARKKITEHTUURI

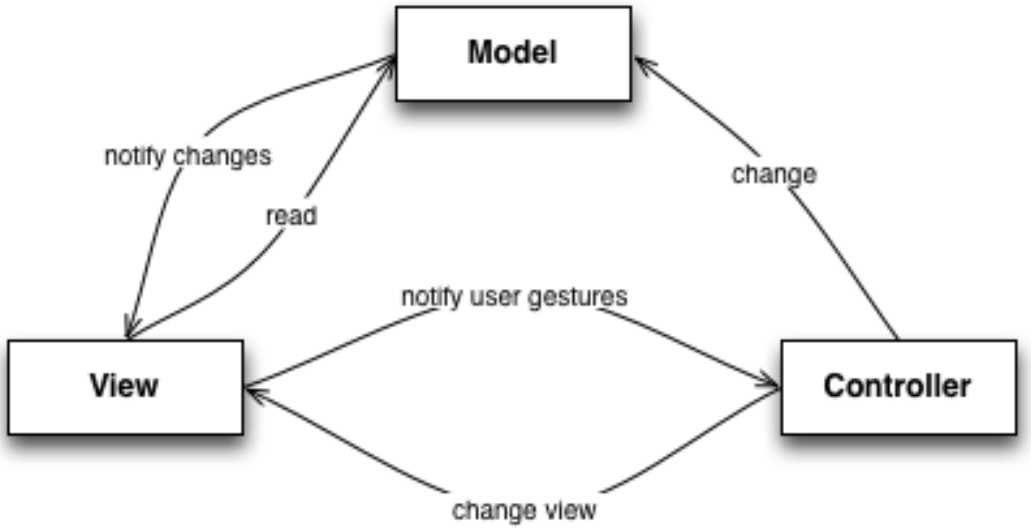
4.1 Yleistä

Ohjelmistoarkkitehtuurin hyödyt tulevat esille erityisesti silloin, jos ohjelmoitava tuote on pitkäikäinen ja monimutkainen. Tällöin arkkitehtuurille voidaan laskea muitten hyötyjen lisäksi rahallinen arvo, joka tulevaisuudessa säästetään ylläpidon kustannuksissa. Ohjelmiston arkkitehtuuri on määrittely- ja sopimusjoukko, joka on toteutettu vaatimusten pohjalta. (Vogel 2011.) Lähestymistapana ei kuitenkaan ollut ohjelmistoarkkitehtuurin kehittäminen, vaan sovelluksen arkkitehtuuri kopioitiin suoraan Kuhan (2008) kirjasta lisäten siihen facade–suunnittelumallin mukainen (ks. luku 4.5) palvelutaso, jonka käyttö opittiin ohjelmistoprojektissa Code Centerin kanssa. Koska käyttöliittymäkerrokseksi valittiin JSF, ohjelma noudattelee myös MVC-arkkitehtuuria.

4.2 MVC-arkkitehtuuri

Kehitettävän sovelluksen arkkitehtuuri noudattaa MVC-arkkitehtuuria, joka tuli otetuksi käyttöön, kun käyttöliittymä päätettiin toteuttaa JSF:llä. MVC on lyhenne sanoista Model, View ja Controller.

View, eli näkymä näyttää sen hetkisen mallin (model) tilan. Näkymiä samaan malliin voi olla useita. Malli on ohjelman tila ja näkymä nimensä mukaisesti se, mitä siitä tilasta halutaan näyttää käyttäjälle. Gamman, Helmin, Johnsonin ja Vlissidesin (2000) kirjasta tehdyn tulkinnan mukaan ennen MVC:tä nämä kaksi olivat niin sanottua spagettikoodia, jonka ylläpito ja uudelleenkäyttö olivat hankalaa. Käsittelijän (controller) avulla nämä kaksi voitiin erottaa toisistaan (Gamma ym. 2000, 14).



Kuvio 4 – MVC-arkkitehtuuri

Käsittelijä toimii protokollana näkymän ja mallin välillä. Näkymän on oltava ajan tasalla mallin kanssa, ja jos malli muuttuu, täytyy näkymän päivittyä. MVC-arkkitehtuuri pohjautuu useaan eri suunnittelumalliin. (Gamma ym. 2000, 16)

Kehitetty sovellus ei ole täysin linjassa Gamman ja muiden (2000) esittämän MVC-arkkitehtuurin kanssa, siltä osin, että MVC:hen kuuluu heidän mukaan käsittelijöiden luokkahierarkia, joka helpottaa uusien käsittelijöiden luomista edellisten perusteella. Sovelluksessa on yhteinen DAO-rajapinta ja abstrakti DAO-luokka, mutta se käyttää tämän lisäksi fasadia, josta kerrotaan enemmän luvussa 4.5.

4.3 Tietotaso

4.3.1 Tietotason koodi

Tietotaso sisältää kaikki ohjelmisto- ja tietokantasuunnitelman oliot. Niistä on tehty JPA:n annotaatioilla entiteettejä, jotka voidaan tallentaa tietokantaan. Alla on esimerkki tietokannan City-taulun pohjalta tehdystä entiteetistä, johon tallennetaan kaupungin nimi ja postinumero. Tässä ei generoida automaattis-

ta juoksevaa ID:tä, koska postinumero toimii pääavaimena (primary key). Postinuneron pituus on rajattu viiteen merkkiin, kenttää ei voi päivittää ja generoitavan tietokantataulun nimeksi tulee post_code.

```
@Entity
@Table(name = "City")
public class City {
   private static final long serialVersionUID = 1L;

   @Id()
   @Column(name = "post_code", updatable=false, length = 5)
   private String postCode = “”;

   @NotNull
   @Column(name = "city", length = 100, nullable = false)
   private String city = "";

  …getteri ja setterit…
}
```

4.3.2 Tietotason testit

Tietotason testit kirjoitettiin jUnit 4:llä. Tietokantaan tallentamista varten täytyy luoda persistence.xml-tiedosto. Siinä kerrotaan, mitkä luokat ovat tallennettavia entiteettejä ja mihin tietokantaan ne on tarkoitus tallentaa. Tässä työssä käytetään kehitysvaiheessa NetBeanssin mukana tulevaa Derby-tietokantaa. Testauksessa käytettiin HSQLDB:tä. Kun ohjelma lopulta otetaan käyttöön, voidaan tietokanta helposti vaihtaa toiseen persistence.xml:n tietoja muuttamalla. Alla esimerkki persistence.xml–tiedostosta:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
  </persistence-unit>
  <persistence-unit name="TestPU" transaction-type="RESOURCE_LOCAL">
    <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
    <class>Objects.City</class>
    [muut entiteettiluokat]

    <properties>
          <property name="javax.persistence.jdbc.driver" val-
ue="org.hsqldb.jdbcDriver"/>
          <property name="javax.persistence.jdbc.url" value="jdbc:hsqldb:mem:KM"/>
          <property name="javax.persistence.jdbc.user" value="km"/>
          <property name="javax.persistence.jdbc.password" value="km"/>
          <property name="eclipselink.ddl-generation" value="drop-and-create-
          tables"/>
          <property name="eclipselink.target-database" val-
          ue="org.eclipse.persistence.platform.database.HSQLPlatform" />
```

```
            <property name="eclipselink.weaving" value="static"/>
            <property name="eclipselink.debug" value="ALL"/>
            <property name="eclipselink.logging.level.sql" value="FINEST"/>
            <property name="eclipselink.logging.level" value="FINEST"/>
            <property name="eclipselink.logging.level.cache" value="FINEST"/>
            <property name="eclipselink.canonicalmodel.subpackage" value="test"/>
            </properties>
  </persistence-unit>
</persistence>
```

Testausta varten on hyvä olla päällä valinta *<property name="eclipselink.ddl-generation" value="drop-and-create-tables"/>*, jotta ei jokaisen testiajon välissä tarvitse tyhjentää tietokantaa käsin. Myös arvo *<property name="eclipselink.canonicalmodel.subpackage" value="test"/>* on hyödyllistä tietää, jos haluaa samaan persistence.xml–tiedostoon usean peristence–unit:n, niin tällöin jokaisella persistence–unitl:lla täytyy olla uniikki arvo. Päällä on myös debuggausviestit, niin nähdään, jos jotain menee pieleen. Ne on tuotantoversiossa muistettava ottaa pois päältä.

Kun persistence.xml on luotu ja asetettu oikeaan hakemistoon, voidaan alkaa kirjoittamaan testejä, jotka käyttävät luodun persistence.xml:n jotain persistence–unit:a. jUnit4:ssä on hyvät @Before ja @After annotaatioit, joilla voidaan tehdä operaatiot, jotka halutaan tehdyksi ennen testien ajoa ja testien ajon jälkeen. Tässä tapauksessa luodaan entityManageri, se täytyy muistaa lopuksi myös sulkea. entityManagerin avulla voidaan entiteetti-luokille tehdä tietokantaoperaatioita, kuten tallennus, poisto ja haku. Myöhemmin esitellään myös oma JPQL–kieli (ks. luku 4.4.2), jolla voidaan SQL:n tapaan tehdä hakuja. Alla on esimerkki city–entiteetin jUnit 4 testistä, joka testaa sitä, ettei kahta samaa pääavainta voi tallentaa tietokantaan:

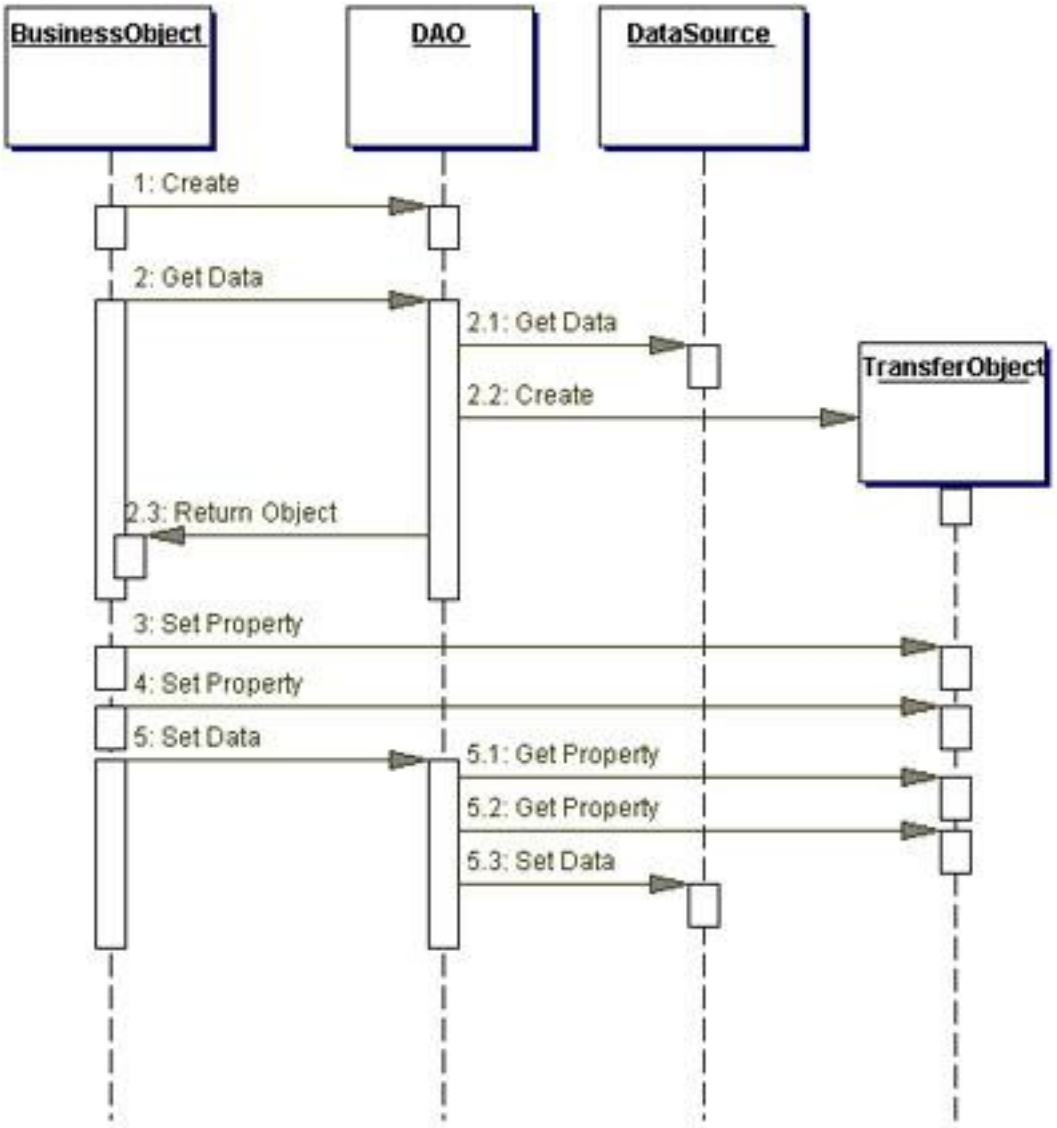
```
public class cityTest {

    private EntityManagerFactory entityManagerFactory;
    private EntityManager entityManager;

    @Before
    public void luoEntityManager() {

 this.entityManagerFactory=Persistence.createEntityManagerFactory("TestPU");
      this.entityManager = entityManagerFactory.createEntityManager();
    }

    @After
```

```
    public void suljeEntityManager() {
        if (entityManager != null){
            this.entityManager.close();
        }
    }

    @Test(expected = javax.persistence.PersistenceException.class)
    public void integrityTest() throws javax.persistence.PersistenceException{
        City city1 = new City();
        City city2 = new City();

        city1.setCity("Jyväskylä");
        city1.setPostCode("40100");

        city2.setCity("Tampere");
        city2.setPostCode("40100");

        EntityTransaction tx = this.entityManager.getTransaction();

        tx.begin();
        this.entityManager.persist(city1);
        this.entityManager.persist(city2);
        this.entityManager.flush();

        tx.rollback();
    }
}
```

4.4 DAO–rakenne

4.4.1 Yleistä

DAO on lyhenne englannin kielisistä sanoista: Data Access Object. Yksittäisen DAO:n tarkoitus on toimia ikään kuin kerroksena ohjelman ja tietokannan välillä. DAO hoitaa yhteyden tietolähteen kanssa hankkien ja tallentaen sitä (Core J2EE Patterns 2002). Siihen on helppo jälkeenpäinkin lisätä operaatioita, joita tarvitsee oliolle tehdä tietokannan avulla. Se voi sisältää esimerkiksi tiettyjen olioiden hakemisen, määrän laskemisen, ja tietyn kategorian alakategorioiden etsinnän. Jos operaatiossa tarvitse yhdistellä useata DAO:a, tehdään se myöhemmin esiteltävässä palvelu–kerroksessa (ks. luku 4.5). Alla sekvenssi diagrammi DAO:n toiminnasta.



Kuvio 5 – DAO sekvenssi diagrammi (Core J2EE Patterns, 2002)

Liiketoiminta olio (BusinessObject) esittää bisneslogiikkaa, joka toimii DAO:n asiakkaana. Tässä opinnäytetyössä tuohon geneeriseen malliin on lisätty bisneslogiikan ja DAO:n väliin vielä palvelukerros (ks. luku 4.5), joka yhdistää DAO:t. (Core J2EE Patterns 2002)

DAO on yllä olevan kuvan keskeisin olio. Se abstrahoi alla olevan tietokannan. Tietolähde (DataSource) on ajuri, joka on yhteydessä tietokantaan. Tarjonta eri tietolähteille on hyvin laaja ja se voi olla mitä tahansa suorasta tiedostosta XML:n kautta oliotietokantoihin. (Core J2EE Patterns 2002)

Kun DAO hakee tietokannasta tiedon, generoi se sen pohjalta siirto objektin (TransferObject), joka tämän opinnäytetyön tapauksessa on JPA:n entiteetti. Jos käytössä ei olisi JPA, täytyisi olio generoida käsin. (Core J2EE Patterns 2002)

Tässä sovelluksessa toteutettiin ensin yleinen DAO–rajapinta, joka toteutettiin yleisessä abstraktissa DAO:ssa mukaillen Janne Kuhan (2008) kirjan esitystä, mukaan ottamatta geneerisen rajapinnan rajaamista vain entiteetti tyyppisille olioille, koska tässä projektissa mukana on entiteettejä, joille ei tule automaattisesti kasvavaa pääavainta (primary key).

4.4.2 JPQL

JPQL (JPA Query Language) on SQL:n kaltainen kyselykieli, jolla ei haeta tietokannan tauluista, vaan sovelluksen oliomallista. Syntaksi on kuitenkin hyvin SQL:n kaltainen, ja jos sitä osaa, oppii myös JPQL:n nopeasti. (Kuha 2008)

Sovelluksessa JPQL:ää on käytetty DAO–luokissa, joissa entiteettejä manipuloidaan. Sen avulla voi tehdä samoja operaatioita, kuin SQL:llä, kuten UPDATE, DELETE ja SELECT tyyppisiä lauseita. Esimerkiksi kysely, joka hakee kaikki City–tyyppiset oliot, olisi seuraavan kaltainen:

```
SELECT c FROM City AS c
```

Kyselyä varten tarvitaan entiteetin hallinnoija (entity manager), jonka createQuery–metodilla luodaan TypedQuery–tyyppinen olio, josta ajetaan itse kysely. Alla on koodiesimerkkii:

```
TypedQuery<City> q = entityManager.createQuery(“SELECT c FROM City AS c”,
City.class);
q.setFlushMode(FlushModeType.AUTO);
List<City> result = q.getResultList();
```

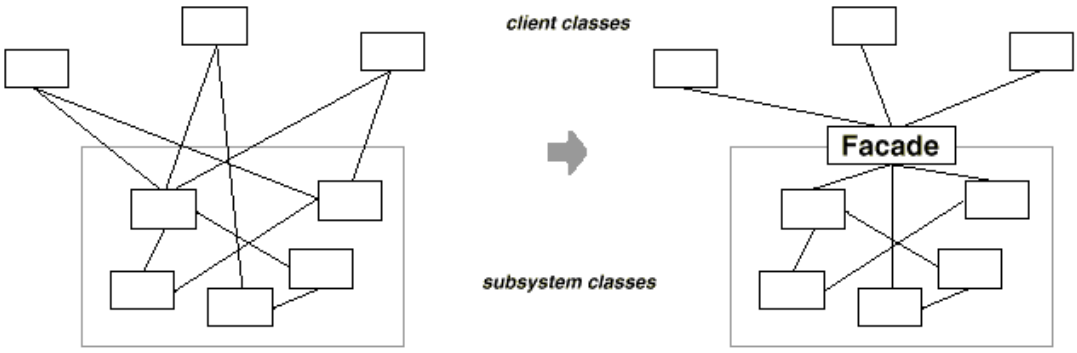
Aivan kaikenlaisia kyselyitä ei kuitenkaan ole mahdollista tehdä JPQL:llä. Esimerkiksi JPQL:ssä ei voi funktioiden sisällä tehdä aritmeettisia laskutoimituksia, kuten *SELECT SUM(price*quantity).* Jos JPQL:n ominaisuudet eivät riitä, on otettava käyttöön natiivit SQL-kyselyt. Se onnistuu entitymanagerin

createNativeQuery-funktion avulla, jota voi käyttää tarvittaessa createQuery-funktion sijasta. (Vasilev. 2008) Sovellusta kehittäessä oli ongelmia JPQL:n kanssa ainakin hakiessa tietoa monimutkaisilla EXIST-alikyselyillä useasta taulusta – SQL-lause ei generoitunut oikein ja käyttöön oli otettava natiivit SQL-kyselyt.

4.5 Fasadi suunnittelumalli (Service-luokka)

Fasadin tarkoitus on määritellä korkeamman tason rajapinta, joka tekee alijärjestelmästä helpomman käyttää. Se auttaa monimutkaisuuden hallinnassa. (Gamma 2000)

Alla olevasta kuvasta voidaan nähdä, miten fasadi yhdistää alemman tason luokat yhdeksi rajapinnaksi, jota ylempi taso käyttää. Tässä opinnäytteessä ohjelmoitavassa ohjelmassa sen on tarkoitus vähentää JSF:n papujen monimutkaisuutta siten, että niihin ei tarvitse injektoida, kuin yksi olio. Myös DAO–luokkien vuorovaikutus voidaan toteuttaa fasadissa. Fasadi toimii rajapintana MVC -arkkitehtuurin mallin (model) ja käsittelijän (controller) välillä. Fasadin avulla voitaisiin tehdä myös erillisiä alijärjestelmiä (Gamma 2000).



Kuvio 6 – Fasadi-suunnittelumalli (Gamma 2000)

Käytännössä fasaadiin, eli sovelluksen Service-luokkaan lisättiin kaikki DAO:t koosteolioiksi ja niitä käytettiin sen kautta.

5 ESITYSKERROS

5.1 Sovelluskehyksen valinta

Janne Kuha (2008) jakaa websovelluskehykset tapahtumapohjaisiin - ja komponenttipohjaisiin sovelluskehyksiin. Tässä opinnäytetyössä komponenttipohjaista sovelluskehystä edustaa JSF–pohjainen Prime Faces ja tapahtumapohjaista GWT–pohjainen Vaadin. Tapahtumapohjaisen – ja komponenttipohjaisen sovelluskehyksen erot ovat kärjistetysti siinä, että komponenttipohjaista kehystä ohjelmoidaan, kuin oltaisiin kehittämässä web-sivua – komponentteja sivulle lisäämällä – kun tapahtumapohjaista kehystä ohjelmoidaan, kuten ohjelmoitaisiin Javalla – lisäämällä aliolioita johonkin pääolioon, jotka sitten näkyvät, kun sivu ladataan selaimeen.

Jos on käyttänyt Javan työpöytäsovelluksille tarkoitettuja käyttöliittymän rakennukseen tarkoitettuja työkalua, tuntee olonsa varmasti kotoisemmaksi tapahtumapohjaisten websovelluskehysten kanssa. Jos taas on ohjelmoinut enemmän HTML:llä, niin komponenttipohjainen websovelluskehys on parempi ratkaisu, tai ainakin nopeampi oppia.

5.2 Prime Faces

Primefaces on JSF:n komponenttikirjasto, johon on olemassa edelleen komponentteja lisäävä laajennos - Primefaces Extension -, jota ei kuitenkaan tässä opinnäytetyössä ohjelmoidussa sovelluksessa tarvittu. Primefaces on avointa lähdekoodia ja lisensoitu Apache License 2.0 lisenssillä. Apache License 2.0 takaa sen, että Primefacesia käyttävän ohjelman voi lisensoida edelleen haluamallaan tavalla. (Apache license 2004) Prime Faces:lla on toteutettu mm.

eBay, joten se vaikuttaisi soveltuvan hyvin verkkokaupan kaltaisiin järjestelmiin.

5.3 Vaadin

Vaadin on tapahtumapohjainen sovelluskehys, joka on lisensoitu Apache 2.0 lisenssillä. Sen orientaatio on erityisesti web-sovellusten kehittäminen, pelkkien web-sivujen sijasta. (Vaadin FAQ) Vaadin soveltuu kuulemma hyvin järjestelmiin, joissa pitää esittää paljon dataa.

5.4 Vertailu

Projektin suunnittelussa käytettiin apuna Janne Kuhan (2008) kirjaa, mutta ei kuitenkaan Springiä. Tarkoitus oli käyttää uutta EJB 3.0 -teknologiaa. Kyseinen teknologia ei suoraan toimi Vaadin–kehyksen kanssa, vaan pitäisi käyttää Vaadin–kehyksen CDI-lisäosaa, jonka avulla @Inject–annotaatiota käyttäen olisi mahdollista injektoida olioita käytettäviksi. JSF:ään perustuvassa teknologiassa toimii suoraan EJB–teknologia, joka oli suuri etu päätöstä tehdessä.

Vaadinta varten NetBeanssiin sai ladattua komponentin, jonka avulla voitiin generoida valmiita Vaadinprojekteja. JSF–pohjainen teknologia NetBeansissa oli kuitenkin helpompi aloittaa, kun sen voi projektia luodessa valita klikkaamalla. NetBeanssissa on perus JSF:n lisäksi laajennokset: Prime Faces, ICE Faces ja Rich Faces. Näistä valittiin Prime Faces, koska siinä on viivakoodikomponentti (versiosta 5.1 alkaen), joka helpottaa huomattavasti kehitettävän sovelluksen ohjelmoimista.

Vaadin projektissa oli myös pahoja ongelmia projektin asentamisessa palvelimelle. Jos teki jonkin (väärän) muutoksen koodiin, niin saattoi käydä niin, ettei tämän jälkeen projekti enää asentunut (deploy) palvelimelle. Tähän ongelmaan ei auttanut, vaikka muutoksen poisti. Joku korruptoi projektin, ja se

oli sitten aina luotava uudestaan. Tämä oli testausvaiheessa hyvin ärsyttävää ja suuri miinus.

Molempia kehyksiä olin kuitenkin käyttänyt aikaisemmissa projekteissa: JSF:ää opintojaksolla Java EE–ohjelmointi ja Vaadinta opintojaksolla Ohjelmistoprojekti, jossa toteutimme Vaadinsovellusta valmiiksi määriteltyyn tyhjään Spring–pohjaiseen projektiin. JSF:ää osasin ehkä kuitenkin vähän paremmin. Vaadin–kehyksen olemassa oleva kehittäjäyhteisö ei ole myöskään vakuuttanut minua siinä mielessä, että aina löytyisi helposti ongelmaan vastaus. Näistä syistä projekti päätettiin toteuttaa käyttäen Prime Faces–kehystä. Taulukko 1 pisteyttää sovelluskehykset tätä projektia/tutkimusta ajatellen asteikolla 1–5.

Taulukko 1 – Pisteytystaulukko

	PrimeFaces	Vaadin
EJB 3.0 -tuki	4	1
Luotettavuus	5	2
Aikaisempi kokemus	3	2
Projektin aloittaminen	5	4
Käyttäjätuki	4	2
Viivakoodin tulostus	4	1
Yhteensä	25	12

Kokeillessani EJB 3.0 – komponentteja Vaadin-kehyksessä, syntyi ongelmia ja niitä oli hankala saada toimimaan. Erikseen ladattavalla lisäosalla myös EJB:t olisi saanut toimimaan. PrimeFacesiinn EJB:n lisääminen @Named-annotaatiolla varustettuun CDI-papuun onnistuu helposti @EJB-annotaatiota

käyttäen. Ongelmia ilmaantui kuitenkin myöhemmin, kun myös filttereissä haluttiin käyttää EJB:itä. Ongelma oli onneksi kuitenkin kierrettävissä.

Sovellusta ajettaessa syntyi luotettavuusongelmia Vaadin-kehyksen kanssa. Projekti täytyi luoda aina ajoittain luoda uudelleen, koska muuten virheilmoitus ei hävinnyt. Se ei hävinnyt, vaikka peruutti kaikki tiedostoihin tehdyt muutokset ja käänsi sovelluksen puhdistaen uudelleen. Primefacesin kohdalla ei ollut tätä vertailua tehdessä koettu minkäänlaisia luotettavuusongelmia. Myöhemmin tosin JSF-sivun käyttämien papujen kohdalla ilmaantui ongelmia – pavut ikään kuin katosivat, eikä niistä saanut tulostettua mitään -, mutta onneksi ongelma poistui aina käynnistämällä Netbeans uudelleen. Myöhemmin ilmeni ajoittain myös: ”java.lang.OutOfMemoryError: PermGen space” –virhe, joka johtuu stackoverflow.com:n vastausten mukaan luokkalataajan muistivuodosta. Joka kerta, kun sovellus sijoitetaan (deploy) palvelimelle, luodaan uusi luokkalataaja ja sovelluksen luokat ladataan uudelleen. Se kuluttaa muistia perm gen tilasta. Tilasta pääsi pois nopeimmin tappamalla Task Managerilla java.exe:n. (PermGen space error – Glassfih Server. 2011) Näiden ongelmien takia, PrimeFacesin luotettavuus numeroa voisi laskea viidestä neljään, mutta annetut numerot jätetään kuitenkin muutaman päivän testauksen jälkeisiksi.

Aikaisemmin olin käyttänyt molempia kehyksiä. JSF:n, johon PrimeFaces perustuu, tunsin paremmin, ja sain sen avulla helposti luotua uuden sitä käyttävän projektin. Vaadinta olin käyttänyt vain valmiiksi luodussa projektissa, enkä osannut kunnolla luoda uutta projektia, varsinkaan sellaista, joka käyttäisi EJB 3.0 – teknologiaa. Itse ohjelmointi kyseisillä sovelluskehyksillä on molemmilla melko suoraviivaista, mutta itseäni miellyttää enemmän JSF:n tapa, joka muistuttaa perus HTML-sivun ohjelmointia.

Primefaces projektin luominen Netbeanssin avulla on erittäin yksinkertaista. Muutamalla klikkauksella valmis Primefaces projekti on valmiina aloitetta-

vaksi. Netbeans tosin loi Primefaces 5.0 – projektin, vaikka 5.1 oli uusin versio ja tarvitsin viivakoodin tulostusta, joka on mukana vasta versiosta 5.1 alkaen. Version sai päivitettyä vaihtamalla projektista tiedoston PrimeFacesin.jar uudempaan. Vaadin-projektin luomiseen täytyi asentaa lisäosa ja IDE:nä käyttöön tulisi ottaa Eclipse, johon se täytyi asentaa. Tämän jälkeen projektin luominen oli hyvin suoraviivaista.

Käyttäjätukena toimi lähinnä Stackoverflow.com:n kysymykset ja vastaukset. Sain sellaisen vaikutelman, että PrimeFaces:n ja yleensäkin JSF:n liittyviä kysymyksiä ja vastauksia on huomattavasti enemmän. Tämä on tärkeää, sillä projekteissa tulee usein ongelmatilanteita, varsinkin jos käyttää teknologioita, joita ei tunne kovin syvällisesti. Vaadin-kehykseen liittyen oli myös joitakin vastauksia ja heillä on myös hyvä foorumi vaadin.com:ssa. Kuitenkin itse ymmärrän usein paremmin toisen ohjelmoijan ohjeita, kuin ohjekirjaa. Siksi PrimeFaces saa käyttäjätukeen paremmat pisteet.

Viimeinen osio koski viivakoodien tulostusta. PrimeFaces 5.1 tukee sitä suoraan. Tosin selvisi myöhemmin, että kaksi jar-tiedostoa piti vielä erikseen lisätä projektiin. Vaadin-kehykseen en suoraan löytänyt viivakoodin kirjoittajaa, mutta joitakin erillisiä kirjastoja on olemassa. Niidenkin käyttö olisi ollut varmasti aivan mahdollista, mutta PrimeFaces 5.1 tekee saman lähes ilman mitään erillistä työtä. Siksi pisteet tästä PrimeFacesille.

Alun perin tarkoitus oli toteuttaa sovellus Vaadin-kehystä käyttäen, mutta ilmaantui muiden muassa luotettavuusongelmia, niin oli pakko etsiä muitakin vaihtoehtoja.

6 SOVELLUKSEN TEKNOLOGIAT MVC-ARKKITEHTUURIN MUKAISESTI

6.1 Yleistä

Luvuissa 6.2–6.4 teknologiat on pyritty jakamaan MVC-mallin (Model view Controller/Malli näkymä käsittelijä) mukaisesti eri lukuihin, sen mukaisesti, mihin ne kuuluvat. Ne kaikki kuuluvat OmniFacesia lukuun ottamatta Java EE – spesifikaatioon, joka koostuu monista teknologioista, joiden on katsottu olevan hyödyllisiä enterprise–tason sovelluksia kehittäessä (Java EE 7 Technologies). Java EE -spesifikaation kaikkia teknologioita ei tarvitse tuntea, jotta voi kehittää Java EE–sovelluksen.

Esitetyt teknologiat käyttävät yleisesti annotaatioita. Ne eivät suoraa vaikuta ohjelman semantiikkaan, mutta ne voivat vaikuttaa siihen, miten ohjelman ulkopuoliset kirjastot ja työkalut muodostavat ajettavan ohjelman semantiikan. (Annotations. 2010) Esimerkiksi *@Transient* on annotaatio, jonka vaikutuksesta luokan attribuuttia, johon annotaatio kohdistuu, ei tule lisätä tietokannan kentäksi (Java Persistence 2.1 Expert Group. 2013, 493).

6.2 Malli

6.2.1 JPA - Java Persistence API

JPA, eli Java Persistence API on spesifikaatio, joka määrittää rajapinnat olioiden hakemiselle, tallentamiselle ja hallinnoimiselle relaatiotietokannoissa. Rajapinnalle on niin avoimen lähdekoodin -, kuin kaupallisiakin toteutuksia. Myös palvelimen tulee tukea sitä, jos sitä halutaan käyttää. (Java Persisten-

ce/What is JPA?) JPA on tapa, jolla olioiden tallentamista tietokantaan voidaan hallinnoida.

JPA toimii annotaatioiden avulla, joista tärkeimpiä ovat: @Id, @Entity ja @Column. Näiden kolmen avulla voidaan jo luoda yksikertainen JPA:ta käyttävä luokka, joka tallentuu tietokantaan. Esimerkkinä seuraava City-luokka/taulu, joka yhdistää postinumeron ja kaupungin:

```
@javax.persistence.Entity
@Table(name = "City")
public class City implements Serializable {
   private static final long serialVersionUID = 1L;

   @Id()
   @NotNull
   @Column(name = "post_code", updatable=false, length = 5, nullable = false)
   private String postCode = null;

   @NotNull
   @Column(name = "city", length = 100, nullable = false)
   private String city = null;

//Setterit ja getterit
}
```

Yllä olevassa lähdekoodiesimerkissä @javax.persistence.Entity määrittää, että City-luokka on entiteetti, jota hallinnoidaan JPA:n avulla ja @Table-annotaation avulla määrätään relaatiotietokantaan luotavan taulun nimi. @Id()-tagi määrää sen luokan attribuutin, jota käytetään relaatiotaulun perusavaimena.

Jokaisella entiteetillä pitää olla perusavain. Se voi olla myös yhdistelmä useita kenttiä, mutta tällöin on käytettävä eri annotaatiota. (Java Persistence 2.1 Expert Group. 2013, 29) City-luokan tapauksessa se ei ole automaattisesti generoituva, vaan se on asetettava ennen olion tallentamista tietokantaan (muuten JPA heittää poikkeuksen). @Column määrittää tietokannan taulun kentän ja sen name-attribuutilla kentälle voi eksplisiittisesti antaa nimen. Esimerkki koodissa on määritelty myös, ettei perusavaimena olevaa kenttää voi päivittää, se ei voi olla null ja sen pituus on oltava viiden merkin mittainen.

JPA tarvitsee myös persistence.xml -tiedoston, jonka oikean tallennuspaikan löytäminen oli sovellusta ohjelmoidessa hankalaa. Lopulta selvisi, että se kuuluu projektinNimi/src/META-INF -kansioon. Kyseisessä tiedostossa määritetään se, mihin tietokantaan tallennetaan, millä ehdoilla ja mitä (debug-) viestejä kaiutetaan konsoliin (ks. myös luku 4.3.2). Projektissa persistence.xml:n tehtiin erilliset asetukset testaamista ja itse ohjelman käyttöä varten, sillä ne käyttävät eri tietokantoja. Alla on esimerkki sovellusta testattaessa käytetystä Derby-tietokannan persistense-unit -määrityksistä:

```
<persistence-unit name="KMPU" transaction-type="JTA">
        <provider>org.eclipse.persistence.jpa.PersistenceProvider</provider>
        <jta-data-source>jdbc/__default</jta-data-source>
        <class>Objects.Category</class>
        [muut entiteetit]
        <properties>

                <property name="javax.persistence.jdbc.driver" val-
ue="org.apache.derby.jdbc.ClientDriver"/>
                <property name="javax.persistence.jdbc.url" val-
ue="jdbc:derby://localhost:1527/sample;create=true"/>
                <property name="javax.persistence.jdbc.user" value="APP"/>
                <property name="javax.persistence.jdbc.password" value="APP"/>

                <!--   <property name="eclipselink.ddl-generation" value="drop-and-
create-tables"/>  -->
                <property name="eclipselink.ddl-generation" value="create-tables"/>
                <property name="eclipselink.weaving" value="static"/>
                <property name="eclipselink.debug" value="ALL"/>
                <property name="eclipselink.logging.level.sql" value="FINEST"/>
                <property name="eclipselink.logging.level" value="FINEST"/>
                <property name="eclipselink.logging.level.cache" value="FINEST"/>

                <property name="eclipselink.canonicalmodel.subpackage" val-
ue="web"/>
                <property name="javax.persistence.schema-
generation.database.action" value="create"/>
        </properties>
</persistence-unit>
```

6.2.2 EJB - Entreprise JavaBeans

Tyypillisesti EJB sisältää bisneslogiikkaa, joka liittyy entrprise–tason sovelluksiin. Niitä voidaan käyttää monissa eri ajoympäristöissä ilman uudelleen kääntämistä. (EJB 3.2 Expert Group. 2013)

EJB:itä on kahdenlaisia: sessio-papuja ja entity-papuja. Suurin ero sessio- ja entity-papujen välillä on se, että entity-pavut eivät ole riippuvaisia palveli-

mesta, vaan ne selviytyvät palvelimen kaatumisesta tai systeemin virheistä. Myös usea asiakas voi käyttää entity-papua samanaikaisesti. Papusäiliö (container) huolehtii entity-pavun synkronoinnista ja transaktioista. (Sengul, Gish & Tremlett 2000) Ajoympäristöä sanotaan säiliöksi (container) (JSR-299 Expert Group 2009)

Sessio-papu ei ole jaettu usean asiakkaan kesken. Ne ovat tyypillisesti lyhytikäisiä, eivätkä ne selviä palvelimen kaatumisesta tai papusäiliön (ccontainer) uudelleen käynnistämisestä. Se tuhotaan, sitä mukaan, kun asiakas ei enää sitä tarvitse. Sessio-papu voi huolehtia myös transaktioista. (Sengul, Gish & Tremlett, 2000) Kun tässä opinnäytteessä puhutaan EJB:stä, tarkoitetaan sessio-papua.

Kehitetyssä sovelluksessa käytettiin vain lokaaleja EJB:itä, jotka sijaitsevat samassa JVM:ssä (Java Virtual Machine), kuin muukin sovellus (pavun asiakas). Remote ja local-pavuilla on myös se ero, että local-papu toimii pass-by-referense -tekniikalla ja remote-papu toimii pass-by-value -tekniikalla. Eli lokaalin pavun pitäisi olla myös nopeampi, jos sitä vain voidaan tilanteessa käyttää. Jos sovellusta ei tarvitse pilkkoa pienempiin osiin eri JVM:iin, on local-bean parempi vaihtoehto. (EJB 3.2 Expert Group 2013)

EJB (tässä sovelluksessa/opinnäytteessä sessio EJB) voi olla tilallinen, tilaton tai singleton. Tilallinen EJB tarkoittaa sitä, että EJB:stä voi olla samaan aikaan käytössä monta instanssia, jotka ovat eri tilassa. Jokaiselle käyttäjälle luodaan oma papu ja se poistetaan, kun käyttäjä katkaisee yhteyden. Tilaton EJB tarkoittaa, että EJB:stä voi olla käytössä monta instanssia samanaikaisesti, mutta ne eivät ylläpidä tilaa. Tällöin on mahdollista luoda vähemmän papuja, mitä on käyttäjiä. Tilattoman pavun tilan tulee sopia kaikille ohjelman käyttäjille, jos se otetaan käyttöön. Singleton-papu tarkoittaa sitä, että pavusta luodaan ohjelmaan vain yksi instanssi, jota kaikki käyttäjät käyttävät. Sitä käytetään, jos pavun tila täytyy voida jakaa koko ohjelmalle. Sen täytyy olla saavutetta-

vissa monista säikeistä samanaikaisesti. (Jendrock, Cervera-Navarro, Evans, Gollapudi, Haase, Markito & Srivathsa 2013)

6.3 Näkymä

Näkymiä muodostettaessa käytettiin seuraavia teknologioita:

- JSF
- PrimeFaces
- OmniFaces

JSF on käyttöliittymäkehys (user interface framework), jonka avulla voidaan kytkeä käyttäjän tapahtumat palvelimen koodiin. Sen avulla saadaan web-sivulle näkymiä sovelluksen datasta ja dataa käyttäjältä sovellukseen (JavaServer Faces Specification 2013, 47). PrimeFaces on komponenttikirjastolaajennos JSF:ään ja OmniFaces on kirjasto, joka korjaa JSF:n puutteita.

6.3.1 Syötteen validointi

Alla oleva koodi toteuttaa AJAX:lla toimivan validoinnin, joka testaa syötteen oikeellisuuden, kun kentästä siirrytään pois. Koodirivissa: *<p:ajax event="blur" update="firstNameErrors, labelForFirstname" />* event-kenttä määrittää sen, minkälainen tapahtuman on oltava, jotta AJAX-toiminto laukeaa.

```
<p:outputLabel id="labelForFirstname" for="firstName" value="Etunimi: " />
<p:inputText id="firstName" value="#{reqisterUserWizzard.firstname}" re-
quired="true" validatorMessage="Pituus pitää olla 2-30 merkkiä" label="Firstname"
        requiredMessage="Etunimi on pakollinen tieto">
        <f:validateLength minimum="2" maximum="30" />
        <p:ajax event="blur" update="firstNameErrors, labelForFirstname" />
</p:inputText>
<h:message id="firstNameErrors" for="firstName" style="color:red" />
```

Yllä esitetyn tavan pitäisi toimia kaikissa JSF:n komponenttikirjastoissa.

Myöhemmin huomattiin, että PrimeFacesissa on erikseen *<p:clientValidator />*-tagi, joka toteuttaa täsmälleen saman toiminnon. Tagi lisätään validoitavan

kentän lapsielementiksi. Tällöin on myös käytettävä <h:message />:n sijasta <p:message>:a ja lisättävä web.xml:n seuraava tagi (Cagatay Civici.):

```
<context-param>
    <param-name>primefaces.CLIENT_SIDE_VALIDATION</param-name>
    <param-value>true</param-value>
</context-param>
```

Tällöin välttyy erikseen kirjoittamasta ID-kenttiä label- ja message-elementeille, joka on yllä esitetyn tekniikan työläin osa. Aikaisempi koodi refaktoroitiin käyttämään jälkimmäistä tapaa, paitsi kustomoitujen validaattoreiden osalta, jotka eivät toimineet suoraan kyseisen tekniikan kanssa, vaan sivulle olisi pitänyt kirjoittaa myös JavaScriptiä. Alla on kuva <h:message />-tagin mukaisesta syötteen validointivirheestä.

Kuvio 7 – Syötteen validointi

Ilman yllä esitettyjä tapoja kentät validoituvat lomakkeen lähetyksen yhteydessä. Tehdyssä ohjelmassa validointiin käytettiin <f:validateDoubleRange />– ja <f: validateLength />-tageja. Edeltävällä validoitiin liukuluvut ja jälkimmäisellä tekstin pituus. Lisäksi määrittelemällä tekstikentälle attribuutti: *required="true"*, muuttui kenttä pakolliseksi, jota ilman lomaketta ei voi lähettää.

Kentille on myös hyvä asettaa seuraavan kaltaiset attribuutit: *validatorMessage="Pituus pitää olla 2-30 merkkiä"*, *requiredMessage="Etunimi on pakollinen tieto"* ja *label="Firstname"*. Ensimmäinen viesti näytetään, jos edellisessä kappaleessa esitetyn tagin validointi laukeaa. Toinen viesti näytetään, jos pakollinen tieto puuttuu ja kolmas attribuutti määrää sen nimen, jota virheilmoituksissa kysei-

sestä kentästä käytetään. Käytettäessä kustomoitua validaattoria on siinä validoitava kaikki muu paitsi pakollinen tieto, koska muuten validointiviestit eivät näy oikein.

6.3.2 Kustomoidut validaattorit

Kustomoitujen validaattorien avulla voidaan tehdä syötteen validointeja, jotka eivät valmiiksi sisään rakennettujen validointien avulla onnistu. Sovelluksessa haluttiin erikseen validoida mm. käyttäjätunnus siten, että ohjelma ilmoittaa, jos käyttäjätunnus on varattu. Tietokanta pitää tästä erikseen huolen, mutta käyttäjää on hyvä ohjeistaa.

Validaattoreissa yritettiin käyttää samaa tekniikkaa, kuin convertereissa, eli luoda luokasta requestScoped-papu ja käyttää sitä *<f:validator binding="#{papu}" />* -tagin avulla. Kyseinen tagi kuitenkin vaatii attribuutin *validatorId*, ja sille jonkin arvon, joten kyseinen lähestymistapa ei onnistunut. Käyttöön oli otettava OmniFaces kirjasto, joka mahdollistaa @EJB- ja @Inject-tagien käytön validaattoreissa (ja convertereissa) (OmniFaces – What is OmniFaces?).

OmniFaces on työkalukirjasto JSF:n. Se on kehitetty ammattilaisten usein kohtaamien ongelmien ratkaisemiseen. Se ei sisällä juuri lainkaan käyttöliittymäkomponentteja, vaan keskittyy paikkaamaan ongelmia, joita JSF API:ssa on. Sitä käytetään yleensä jonkin komponenttikirjaston ohella. (OmniFaces – What is OmniFaces?)

OmniFacesia käyttäen validaattorin koodi näyttää seuraavalta:

```
@FacesValidator("usernameExistsValidator")
public class UsernameExistsValidator implements Validator {

    @EJB
    Service service;

    @Override
    public void validate(FacesContext context, UIComponent component, Object
value) throws ValidatorException {
```

```
                if(value==null)
                        return;

                if(((String)value).length() < 2 || ((String)value).length() > 30) {
                FacesMessage msg = new FacesMessage("Käyttäjänimen validointi
                epäonnistui.",

                "Pituus pitää olla väliltä 1-30 merkkiä.");
                        msg.setSeverity(FacesMessage.SEVERITY_ERROR);
                        throw new ValidatorException(msg);
                }

                Person testUser;
                try {
                        testUser = service.getPersonByUserName(value.toString());
                }
                catch(Exception ex){
                        return;
                }

                if(testUser != null) {
                        FacesMessage msg = new FacesMessage("Käyttäjänimen
validointi epäonnistui.",

                "Käyttäjänimi on jo olemassa.");
                        msg.setSeverity(FacesMessage.SEVERITY_ERROR);
                        throw new ValidatorException(msg);
                }
        }

}
```

Validoitavassa input-kentässä se otetaan käyttöön seuraavasti:

```
<p:inputText value="#{reqisterUserWizzard.username}" required="true" validator="usernameExistsValidator" />
```

6.3.3 Captcha

Primefacesin komponentteihin kuuluu myös captcha, jonka avulla voidaan varmistaa, että rekisteröityvä henkilö ei ole botti. Kyseinen komponentti vaati rekisteröitymisen Googlen palveluun, josta saatiin julkinen - ja salainen avain. Ne lisättiin web.xml-tiedostoon seuraavasti:

```
<context-param>
    <param-name>primefaces.PUBLIC_CAPTCHA_KEY</param-name>
    <param-value>[public key]</param-value>
</context-param>

<context-param>
    <param-name>primefaces.PRIVATE_CAPTCHA_KEY</param-name>
    <param-value>[private key]</param-value>
</context-param>
```

Primefacesin komponentti ei toistaiseksi käytä Googlen reCaptchan ajax API:a, joten sen ei tue ajax toimintoja, joita kehitettävä sovellus kuitenkin käyttää. Tämän ongelman pystyi kiertämään asettamalla captchan erilliseen dialogiin ja avaamalla se rekisteröitymisen lopuksi seuraavasti:

```
<p:dialog widgetVar="captchaDlgWar" modal="false" closable="false" resizable="false"
        header="Todista olevasi ihminen..." width="350" height="200">
<h:form id="capt">
  <h:panelGrid columns="1">
          <p:captcha label="Captcha"
                                    id="captchaId"
                                    language="fi"
                                    theme="white"
                                    required="true"
                                    requiredMessage="Syötä captchan teksti"
                                    validatorMessage="Tulkintasi meni pieleen. Yritä uudelleen klikkaamalla jatka rekisteröitymistä"
                                    secure="true"/>
          <p:commandButton value="Rekisteröidy" ajax="false"
                                                icon="ui-icon-check" action="#{reqisterUserWizzard.save}" />
  </h:panelGrid>
</h:form>
</p:dialog>
```

Dialogin saa aukeamaan nappulasta seuraavalla tavalla:

```
<p:commandButton value="Jatka rekisteröitymistä..." type="button" onclick="PF('captchaDlgWar').show();" />
```

Kuvio 8 esittää captchan, joka on auennut rekisteröitymisen yhteydessä.

Kuvio 8 – Captcha

6.3.4 Sapluunat (template)

Sapluunat ovat ikään kuin pohjia, joita käytetään uudestaan. Esimerkiksi monta kertaa toistuva navigointipalkki on hyvä tehdä sapluunojen avulla, koska tällöin yhtä sivua muokkaamalla muuttuu jokainen siitä sapluunasta

luotu sivu. Sapluunojen käyttö JSF:ssä (ja Prime Facesissa) on melko yksinkertaista.

Sapluunaa varten täytyy luoda itse sapluuna ja oletusarvot jokaiselle sapluunassa määritetylle muutettavalle sivun osiolle. Sapluunaan luodaan osiot käyttämällä *ui:insert* tagia. Kyseiseen tagiin määritellään attribuutti *name,* joka määrää sen, miten kyseiseen osioon myöhemmin viitataan. Tagin *ui:include* avulla määrätään insert-tagin sisään sen oletusarvo. Sen voi ylikirjoittaa sapluunaa käyttävässä sivussa tagilla *ui:define* ja määrittämällä sen name-attribuutiksi sama, joka on määriteltynä sapluunassa. Alla esimerkkikoodia sapluunasta ja sitä käyttävästä sivusta:

```
<div id="content">
        <ui:insert name="content" >
          <ui:include src="/template/checker/checkerContent.xhtml" />
        </ui:insert>
</div>
```

Yllä määritellään sapluunaan osio, johon laitetaan oletusarvona tiedostossa checkerContent.xhtml määriteltävä sisältö.

```
<title><ui:insert name="title">Default Title</ui:insert></title>
```

Otsikko määritellään sapluunaan yllä olevan kaltaisesti.

```
<ui:composition template="../template/checker/checkerLayout.xhtml">
        <ui:define name="title">Kirjaa sisään</ui:define>
        <ui:define name="content">
                <h2>This is checkIn content</h2>
        </ui:define>
</ui:composition>
```

Yllä on koodi, joka ottaa käyttöön sapluunan ja lisää siihen oman content nimisen osion ja sivun otsikon.

6.4 Käsittelijä

6.4.1 Taustapavut (Backing Beans)

Taustapavun getterit ja setterit näkyvät JSF:n sivulle ja papujen tilaa voi muuttaa niiden avulla. Tätä tarkoitusta varten pavulle täytyy antaa EL-nimi. Se on-

nistuu @Named-annotaation avulla. Alla on esimerkki ostoskoritaustapavun julistuksesta:

```
@Named("shoppingChart")
@SessionScoped
public class ShoppingChart implements Serializable {
        …koodi…
}
```

Annotaatio @SessionScoped määrää sen, että taustapapu on olemassa yhtä kauan, kuin sessio on olemassa. Seuraavat laajuudet (scope) ovat olleet käytössä Java EE 6:sta alkaen (Jendrock. ym. 2010):

- Request
- Session
- Application
- Dependent
- Conversation

Myöhemmin mukaan on tullut myös JSF:stä tutu näkymälaajuus (ViewScope), joka tarkoittaa sitä, että taustapapu tuhoutuu vasta, kun JSF - tai JSP-sivu vaihtuu toiseen. Ennen tätä, kyseinen annotaatio oli mahdollista saada käyttöön OmniFaces-kirjaston avulla.

EL-nimen omaavat pavut ovat käytettävissä JSF - ja JSP-sivuilla. (JSR-299 Expert Group 2009) Taustapavut toimivat linkkinä Service-luokan ja JSF-sivujen välillä. Service-luokka liitettiin taustapapuun @EJB-annotaatiolla. Service-luokan liittäminen taustapapuun tapahtui seuraavalla tavalla:

```
@EJB
 Service service;
```

Taustapapujen kehittämiseen ei tässä sovelluksessa ollut suunnitelmaa, vaan niitä luotin sitä mukaan, kun JSF-sivuja luotiin. Usein yksi taustapapu toimi usealle sivulle, eli ne olivat melko hyvin uudelleen käytettäviä.

Kun taustapapu tarvitsi tietoa toiselta pavulta, esimerkiksi siitä, että onko käyttäjällä riittävät käyttöoikeudet, niin pavun tarvitsi saada tietoa toiselta

pavulta. Toisen pavun lisääminen taustapapuun onnistui @Inject-annotaatiolla. Seuraavalla tavalla:

```
@Inject
 Login login;
```

Yllä olevalla koodilla Login-taustapavun metodit saatiin käyttöön sitä tarvitsevissa taustapavuissa.

6.4.2 Virheiden käsittely

JSF ei sellaisenaan tue AJAX-teknologian virheiden käsittelyä. Kehitettävässä sovelluksessa niitä kuitenkin oli ja käytettävä ohjelmistokehys (PrimeFaces) toimii vakioasetuksilla AJAX:lla. AJAX:n virheidenkäsittely saatiin toimimaan OmniFacesin avulla. (OmniFaces – Full Ajax ExceptionHandler)

Tässä vaiheessa virheistä käsitellään vain sessioiden vanhentuminen, koska beta-testauksessa on hyvä, että virheilmoitukset erottuvat selkeästi.

AJAX-virheiden käsittelyyn tarvitaan kahden Omnifaces-poikkeuskäsittelijän käyttöön ottamista. Toinen käsittelee AJAX-kutsut ja toinen tavalliset. Omnifacesin jar-tiedoston lisäämisen jälkeen käyttöönotto vaati seuraavat vaiheet:

faces-config.xml:n piti lisätä rivit:

```
<factory>
        <exception-handler-factory>
                org.omnifaces.exceptionhandler.FullAjaxExceptionHandlerFactory
        </exception-handler-factory>
</factory>
```

web.xml:n piti lisätä:

```
<error-page>
        <exception-type>
                javax.faces.application.ViewExpiredException
        </exception-type>
        <location>
                /errorPages/viewExpired.xhtml
          </location>
</error-page>
```

javax.faces.application.ViewExpiredException laukeaa, kun session vanhenee ja <location>-tagissa oleva polku on se sivu, joka näytetään, kun sovellus seu-

raavan kerran päivitetään. Käyttämällä <exception-type>-tagin sijasta esimerkiksi seuraavaa koodia: *<error-code>404</error-code>,* voidaan virheen 404 sattuessa ohjata käyttäjä erilliselle virhesivulle. (OmniFaces – Full Ajax ExceptionHandler)

Tavallisten poikkeusten käsittelyyn tarvitsi lisätä web.xml-tiedostoon seuraavat rivit (Class FacesExceptionFilter. 2014):

```
<filter>
    <filter-name>facesExceptionFilter</filter-name>
    <filter-class>org.omnifaces.filter.FacesExceptionFilter</filter-class>
 </filter>
 <filter-mapping>
    <filter-name>facesExceptionFilter</filter-name>
    <servlet-name>facesServlet</servlet-name>
 </filter-mapping>
```

6.4.3 Filtterit

Login-filtteri

Login-filtterin avulla rajataan eri käyttöoikeuksisien käyttäjien pääsyä sovelluksen eri sivuille. Kehitettävän ohjelman tapauksessa siinä käytettiin seuraava annotaatiota, jolla määriteltiin, mitkä kansiot filtteröidään:

```
@WebFilter(filterName = "LoginFilter", urlPatterns = {"/faces/admin/*"})
```

Kyseinen annotaatio kirjoitetaan filtteriluokan julistuksen ylle. Filtteriluokka toteuttaa Filter rajapinnan.

Filtteriin oli injektoitava olio, josta voitiin tarkistaa, kuka käyttäjä on:

```
@Inject
 private Login login;
```

Itse filtteriointi tehtiin seuraavasti:

```
@Override
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse resp = (HttpServletResponse) response;
        String pageRequested = req.getRequestURL().toString();

        if (login == null || !login.getLoggedIn()) {
                resp.sendRedirect("../mainCategories.xhtml");
                //resp.sendError(HttpServletResponse.SC_UNAUTHORIZED);
        }
```

```
        else if (pageRequested.contains("/admin/") && !login.hasRole(Role.ADMIN)) {
                resp.sendRedirect("../mainCategories.xhtml");
                //resp.sendError(HttpServletResponse.SC_UNAUTHORIZED);
        }
        else {
                chain.doFilter(request, response);
        }

}
```

Login-luokkassa on roolit seuraavan enum:n kaltaisesti:

```
public enum Role {
   ADMIN, SELLER, BUYER;
}
```

Olisi ollut mahdollista, oikeudettoman käyttäjän tapauksessa, ohjata suoraan käyttäjäoikeuksien puuttumisesta ilmoittavalle virhesivulle seuraavalla tavalla: *resp.sendError(HttpServletResponse.SC_UNAUTHORIZED),* mutta itsestäni vaikutti kuitenkin tietoturvallisemmalta ohjata käyttäjä ohjelman pääsivulle. Näin käyttäjä ei tiedä, että on olemassa jokin sivu, jolle hän ei pääse, ja jolle hän voisi mahdollisesti yrittää päästä.

NoCache-filtteri

Kyseisen filtterin avulla määritellään, ettei selain laita sivuja välimuistiin, vaan jos esimerkiksi selaimen back-nappulaa painetaan, ladataan sivu palvelimelta asti uudestaan. Tämän tekemiselle on kaksi syystä: ensimmäinen on se, ettei back-nappulalla päästä takaisin sivuille, jonka käyttö on rajoitettu vain tietylle ryhmälle. Toinen syy on se, jos sivu ladataan välimuistista, tällöin sivulla ei ajeta kyseisen sivun init-metodeja, ja tällöin esimerkiksi tuotesivun breadCrumb (paneeli, joka näyttää missä kategoriassa ollaan), ei päivity oikein.

Runko NoCahce-filtterille on samanlainen, kuin Login-filtterillä, mutta filtteröitäviksi valitaan kaikki sivut. Alla on koodi, joka poistaa selaimen välimuistin käytöstä:

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain
chain) throws IOException, ServletException {
      HttpServletRequest httpReq = (HttpServletRequest) request;
      HttpServletResponse httpRes = (HttpServletResponse) response;
```

```
        if (!httpReq.getRequestURI().startsWith(
                        httpReq.getContextPath() +
          ResourceHandler.RESOURCE_IDENTIFIER)) {

            httpRes.setHeader("Cache-Control",
                    "no-cache, no-store, must-revalidate"); // HTTP 1.1.
            httpRes.setHeader("Pragma", "no-cache"); // HTTP 1.0.
            httpRes.setDateHeader("Expires", 0); // Proxies.
        }

        chain.doFilter(request, response);
    }
```

6.4.4 Converterit

Converterin avulla JSF muuntaa olion merkkijonoksi tai merkkijonon olioksi. Niitä tarvitaan siksi, koska HTML on pelkkää tekstiä.

Yleensä converter julistetaan @FacesConverter-annotaation avulla. JSF:ssä on kuitenkin ollut ongelma, jos converteriin injektoidaan EJB (ks. luku 6.2.2) tai taustapapu (ks. luku 6.4.1). Kun halutaan välttää nullpointer exception (jos ei haluta käyttää omniFaces-kirjastoa), täytyy converteri annotoida @Named-annotaatiolla. Alla on esimerkki eräästä kehitettävässä ohjelmassa käytetystä converterista:

```
@Named("categoryConverter")
@RequestScoped
public class CategoryConverter implements Converter{

    @EJB
    Service service;

    @Override
     public Object getAsObject(FacesContext context, UIComponent component,
      String value) {
        if (value == null) {
            return null;
        }

        if(value.trim().equals(""))
            return null;

        Category category = service.getCategoryByPK(value);

        return category;
    }

    @Override
    public String getAsString(FacesContext context, UIComponent component, Ob-
ject value) {
        if (value == null) return null;
        if (value instanceof String) return (String)value;
```

```
        Category category;

        if(value instanceof Category) {
            category = (Category)value;
            String shopName = category.getName();
            return shopName;
        } else throw new ConverterException("Something wrong!" + value.hashCode()
+ value.toString());
    }

}
```

Esitetty converteri muuntaa tekstimuotoisen kategorian oliomuotoiseksi ja toisin päin. Esimerkin converteri liitetään käytettävään komponenttiin tagilla: *<f:converter binding="#{payment_methodConverter}" />*.

Converterin olisi voinut tehdä myös käyttämällä Omnifacesia, @FacesConverter-tagia käyttäen, mutta päädyttiin tähän vaihtoehtoon, koska se lienee nopeampi, sillä se ei käytä ylimääräistä kirjastoa.

6.5 jUnit 4

jUnit on Javan yksikkötestaus työkalu, jonka avulla tässä projektissa testattiin entiteetit ja DAO:t. jUnit 4 on hyvin integroitu NetBeans kehitysympäristöön. Uuden testijoukon voi luoda kätevästi muutamalla hiiren klikkauksella. Ne ilmestyvät projektiin omaan kansioon, johon niille kannattaa luoda erikseen nimetty paketti.

jUnit 4 tarjoaa annotaatioita, joista tärkeimmät ovat @Before, @After ja @Test. @Before:n avulla merkitään metodi, joka halutaan ajettavaksi ennen, kuin testijoukon testejä aletaan ajaa seuraavan kaltaisesti:

```
@Before
public void luoEntityManager() {
  this.entityManagerFactory=Persistence.createEntityManagerFactory("KMTestPU");
  this.entityManager = entityManagerFactory.createEntityManager();
}
```

@After-tagin avulla merkitään testijoukon jälkeen ajettava metodi seuraavan kaltaisesti:

```
@After
public void suljeEntityManager() {
```

```
    if (entityManager != null){
        this.entityManager.close();
}
  }
```

Myöhemmin @After-metodiin lisättiin myös seuraavat rivit:

```
entityManager.getTransaction().begin();
entityManager.createNativeQuery("TRUNCATE SCHEMA PUBLIC AND COMMIT
NO CHECK").executeUpdate();
entityManager.getTransaction().commit();
```

Yllä olevien rivien avulla tietokanta tyhjennetään jokaisen testitapauksen jälkeen, jotta sinne ei jää mahdollisesti seuraavia testejä sotkevaa tietoa. Komento: ”TRUNCATE SCHEMA PUBLIC AND COMMIT NO CHECK” on HSQLDB:n, eikä välttämättä toimi kaikissa tietokannoissa. HSQLDB on keskusmuistissa toimiva nopea tietokanta, joka on hyvä testausvaiheessa (Kuha, 2008).

@Test-annotaatiolla merkitään itse ajettava testitapaus.

Itse testaaminen tapahtuu assert-metodien avulla. Yleisin testi on sen kaltainen, että jotain arvoa verrataan toiseen. Tällöin käytetään Assert.assertEquals-metodia, jonka ensimmäisenä attribuuttina on virheilmoitus, joka näytetään, jos verratavat arvot eivät ole samat. Toisena attribuuttina annetaan se, mikä arvon pitäisi olla ja kolmantena annetaan arvo, jonka pitäisi olla sama, kuin toisena attribuuttina annettu arvo. Alla esimerkki yhdestä Assert.assertEquals -metodin käytöstä:

```
Assert.assertEquals("Väärä nimi", "Ö-kauppa", shop.getName());
```

7 TOTEUTUS

7.1 Netbeans

Käytetään, koska todettiin Java EE–ohjelmointi opintojaksolla, että soveltuu hyvin JSF-ohjelmointiin ja voi generoida automaattisesti PrimeFaces-projektin. Aluksi oli vaikeuksia luoda aliprojekti rakenne, mutta tämä onnistui, kun ymmärrettiin luoda aliprojektista tavallinen Javaprojekti, eikä web-projekti, joka ei käännä JAR-tiedostoa, jonka pääprojekti tarvitsee aliprojektilta. Tähän oli hyvät virheilmoitukset Netbeanssissa, joiden avulla asia saatiin ratkaistua.

7.2 Entiteetit

Ohjelmointi aloitettiin ohjelmoimalla jokainen olio ja tekemällä siitä JPA-entiteetti. Entiteeteille kirjoitettiin jUnit 4-testit. Erikoisimpia entiteettejä olivat ne, jotka eivät perineet Entity-luokkaa, vaan niille määriteltiin perusavaimeksi String-tyyppinen muuttuja. Ne testattiin erityisen tarkasti.

Automaattinen perusavaimen luonti saatiin aikaiseksi annotaatiolla: *@GeneratedValue(strategy = GenerationType.AUTO).* Tämä asetettiin entiteettien luokkahierarkian ylimpään entiteettiin. Entity-luokasta periytyville entiteeteille, luodaan automaattisesti kasvava numero perusavaimeksi. Näin id voidaan helposti muuttaa kaikkiin sen kaltaisiin entiteetteihin.

Perusavaimiin liittyvien ongelmien ratkaisemisen jälkeen alettiin kirjoittaa entiteeteille DAO-luokkia ja niille testejä.

7.3 DAO:t

Jokaiselle entiteetille kirjoitettiin oma DAO (Data Access Object) Janne Kuhan (2008) ohjeiden mukaisesti. DAO:t testattiin jUnit 4 kehystä käyttäen. DAO:hin kirjoitettiin sitä mukaa uusia metodeja, kun niitä tarvittiin taustapavuissa (ks. luku 6.4.1). Suurin osa metodeista käytti JPQL:ää käsitellessään entiteettejä. DAO:ja käytetään service-luokan kautta.

7.4 Pavut (JavaBeans)

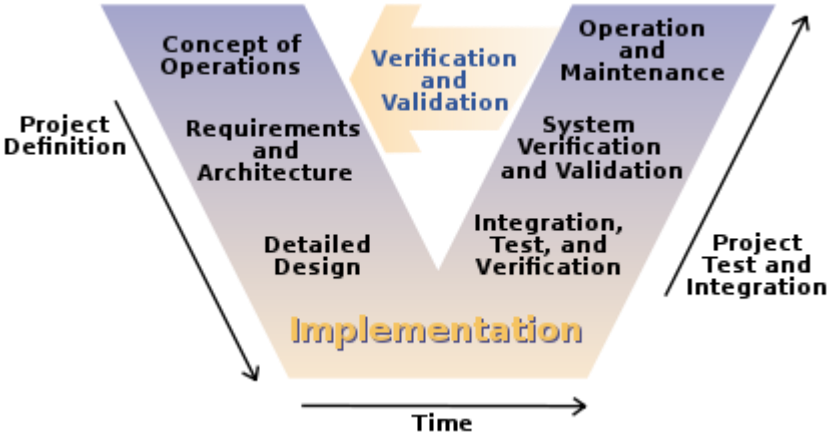
Taustapapujen ohjelmoimista ei suunniteltu mitenkään. Ne kirjoitettiin ad hoc, sitä mukaan, kun sovelluksen käyttöliittymä tarvitsi toimintoja. En tiedä, miten pavut kannattaisi ennalta suunnitella, saati testata. Ja se voisi olla hyvä jatkotutkimusaihe. Voisi tutkia myös sitä, millaiseen arkkitehtuuriin tilansa pitävät (statefull) EJB:t soveltuvat, kun papujen kanssa kyseisestä ominaisuudesta ei ole hyötyä, silla taustapapu voi pitää tilansa sessiossa. EJB:t olivat tässä sovelluksessa DAO-luokkia, joiden ei tarvinnut pitää tilaa.

7.5 Testauksen V-malli

Kuten edellä mainittu, sovellukseen oli selkeä spesifikaatio, jonka perustella sitä alettiin toteuttaa. Tämän voi ajatella olevan V-mallin (ks. Kuvio 9) kohta *Concept of Operations*. Opintojaksoilla Tietokantojen suunnittelu ja Ohjelmisto suunnittelu, toteutettiin vaihe *Requirements and Architecture*. Ohjelmistosuunnittelu-opintojaksolla sivuttiin hieman myös vaihetta *Detailed Design*. Vaihetta *Detailed Design* ei periaatteessa viety aivan loppuun saakka, koska JSF–sivujen vaatimat pavut toteutettiin ad hoc implementaatio-vaiheessa. Suunnitelman implementoinnin yhteydessä tehtiin myös yksikkötestit.

Toimeksiantaja ei ole löytänyt sovelluksesta suuria puutteita, joten kohdan *System Verification* voi myös katsoa läpiviedyksi. *System Validation* vaihe on

hankala todentaa, koska muita ohjelmoijia ei ole, eikä koodin laadusta voi keskustella kenenkään kanssa. Olen kuitenkin pyrkinyt ohjelmoimaan sovelluksen parhaita käytäntöjä käyttämällä - ilman purkkaviritelmiä. Ainoastaan JSF:n converterien kanssa joutui tekemään parhaista käytänteistä poikkeavan ratkaisun (ks. luku 6.4.4) ja captchan kanssa oli ongelmia, koska se ei tällä hetkellä tue ajaxia (ks. luku 6.3.3). *Operation and Maintenance* -kohtaan mennään vasta beta-testauksen jälkeen. Eikä ohjelman käyttöä varten ole vielä ostettu viivakoodinlukijaa ja kokeiltu sitä, jotta sovellus olisi täysin valmis.



Kuvio 9 – Testauksen V-malli

Hieman toimeksiantajan kanssa oli kuitenkin keskustelua esimerkiksi siitä, olisiko parempi, että käyttäjätunnus olisi sähköpostiosoite. Tämän kaltaista käytäntöä ei ole ainakaan Huuto.netissä, josta osittain otettiin mallia ja asia jätettiin hautumaan. Keskustelua syntyi myös uloskirjautumisnappulasta, joka toimeksiantajan mielestä olisi parempi, jos siinä lukisi käyttäjän käyttäjätunnus. Se muutettiin sellaiseksi, ja sitä klikkaamalla aukeaa nyt alavalikko, josta voi valita omien tietojen muuttamisen ja ulos kirjautumisen. Siihen saattaa mahdollisesti myöhemmin tulla myös lisää toimintoja. Pääadminin kohdalla valintoja on useampi.

8 POHDINTA

Tämän opinnäytetyön ensisijainen tavoite oli saada kehitettyä valmis Java EE -sovellus tehtyjen suunnitelmien pohjalta, toimeksiantajan yritysidean toteuttamiseksi. Sovellus saatiin kehitettyä. Toisena tavoitteena oli luoda työkalupakki, jonka avulla voisi kehittää sovelluksia omassa yrityksessä. Tämä onnistui ainakin siinä mielessä, että tämä opinnäytetyö sisältää kuvauksia siitä, miten eri teknologioita käytetään. Tässä opinnäytetyössä esitetyt teknologiat soveltuvat laajojen web-sovellusten kehittämiseen. Toimeksiantajakin vaikuttaa tyytyväiseltä, vaikka ei ole kaikkea tämän kirjoitushetkellä vielä testannut.

8.1 Menetelmän näkökulma

Tapaustutkimuksen näkökulmasta tarkasteltuna tämä tutkimus toi lisätietoa käytettävien ohjelmistoprosessien valinnasta siten, että vesiputousmalli vaikuttaisi hyvältä vaihtoehdolta, jos sovelluksen vaatimukset on ennalta hyvin selvitetyt. Muutamia kertoja jouduttiin kuitenkin peruuttamaan vesiputousta takaisinpäin, kun huomattiin, että jonkin vaatimuksen tarpeellisuutta ei ollut huomattu tarpeeksi ajoissa. Sovelluksen hyvin suunniteltu arkkitehtuuri myötäili muutoksia ja se vaikuttaisi soveltuvan myös ketterämpään kehitykseen. Vaikutelmaa tukee se, että Janne Kuhan (2008) kirjassa lähes samaa arkkitehtuuria käytettiin esimerkkiprojektissa, joka toteutettiin ketterillä menetelmillä. Tätä voi pitää myös eräänä tuloksena, koska organisaatio (ohjelmoija) oppi uudesta prosessista (vesiputousmalli) ja uskaltaa tarvittaessa tulevaisuudessa käyttää samoja teknologioita ja arkkitehtuuria myös ketterässä prosessissa, koska alun perin lähes samankaltaista arkkitehtuuria sen kaltaiseen prosessiin suositellaan (Kuha 2008). Aikaisempaa kokemusta ketteristä menetelmistä oh-

jelmoijalla on Jyväskylän ammattikorkeakoulun opintojaksolta Ohjelmisto-prosessi.

Opinnäytetyö tekeminen ja erityisesti toimeksiantajan kanssa työskentely oli hyvää oppia tulevaisuutta varten. Täten organisaation osaaminen ja tietämys lisääntyi ja sen käyttämä (ohjelmisto)prosessi kehittyi. Näin myös koko organisaatio kehittyi, koska sen toimintatapoihin vaikutettiin ja luotiin uutta tietoa, joka on toimintatutkimuksen tavoite. (Toimintatutkimus 2015; Dick & Swepson 2013) Prosessin voi nyt aloittaa kysymyksellä: ovatko sovelluksen speksit hyvin tiedossa? Jos ei ole, niin käytetään ketteriä menetelmiä, ja jos on, niin voidaan käyttää vesiputousmallia.

Uusi haaste olisi kehittää sovellus ketterästi, mutta siihen sisältyy se ongelma, että jos kärjistetään, niin miten voi kehittää sovellusta, jos toimeksiantajakaan ei täysin tiedä, mitä haluaa? Ja tietysti aloittelevan yrittäjän näkökulmasta keskeistä on se, mistä asiakkaan/asiakkaita voi ensinnäkään löytää? Tätä voisi mahdollisesti joku markkinointia opiskeleva tutkia.

8.2 Sovelluksen kritiikki

Sovelluksen ulkoasusta vaikuttaisi puuttuvan jotain. Se voisi näyttää vielä paremmalta, mutta siihen ei ollut käytettyjen teknologioiden puitteissa kunnollista mahdollisuutta. PrimeFacesiin on helposti vaihdettavia teemoja, joita voi myös ostaa lisää. Niiden avulla sovelluksen ilmettä voi helposti muuttaa, mutta joku tekijä siitä mielestäni silti puuttuu, jotta se näyttäisi suuremman/kehittyneemmän organisaation tekemältä. Osasyy voi olla logon ja mainosten puuttuminen. Jatkossa voisi siis pyrkiä miettimään, mikä tekee sovelluksesta ammattimaisen näköisen, ja joku voisi sitä myös tutkia ja myös sitä, kuinka merkityksellistä se on käyttäjälle.

Myös virhesivuille voisi tehdä erilliset ohjelman teemaa tukevat sivut, mutta tällä hetkellä, kun sovellus on menossa beta-testaukseen, vaikuttaisi olevan parempi, että virheilmoituksen erottuvat selvästi, jotta ne osataan huomioida helpommin ja niiden aiheuttaja korjata lopulliseen versioon.

Sovellusta kehitettäessä huomattiin myös, että taustapapujen (ks. luku 6.4.1) suunnittelu ennalta on hankalaa. Joku voisi kehittää siihen jonkun menetelmän. Niitä voisi ehkä myös testata - ainakin Netbeans ehdotti niille automaattista testien luomista. Kehitetty sovellus ei ole muutenkaan täysin testattu ja esimerkiksi Janne Kuhan (2008) kirjassaan esittämä sijaisolio-testaus (mock object) easymockin avulla jätettiin kokonaan tekemättä. Yksikkötestit pyrittiin aina tekemään, mutta testikattavuutta ei selvitetty. Täten sovelluksen laadun voi kyseenalaistaa, mutta eihän tässä mitään pankkijärjestelmää olla tekemässä, vaikka rahan arvoista dataa sen välityksellä liikkuukin. Sellaista dataa sisältävien sivujen liikenne on tarkoitus salata SSL:llä (https-protokolla).

Tuotteiden selailusivun keskimmäinen palkki ei ole myöskään samassa linjassa sivupalkkien kanssa. Keskipalkin koko vaihtelee sen mukaan, montako tuotetta siinä näkyy. Yhdellä kertaa näkyvien tuotteiden määrä on rajattu viiteen, mutta viiden tuotteen näkyessäkin elementin korkeus saattaa vaihdella usean kuvapisteen (pixel) verran.

8.3 Prosessin kritiikki

Ohjelmistoa kehittäessä havaittiin, että sovelluksen tietokantasuunnitelmaa oli tarkennettava ajoittain, kun ohjelmaan oli saatava ominaisuuksia, joihin ei ollut osattu ennalta varautua. Huomattiin myös, että ulkoasun suunnittelu oli melko hankalaa, sillä ohjelmoija ei tuntenut kaikkia PrimeFacesin komponentteja ennalta, eikä täten siis kyennyt suoraan sanomaan, millaisen ulkoasun voi luoda. Tämä olisi ollut mahdollisesti vältettävissä lukemalla jokin PrimeFacesilla kehittämisestä kertova kirja. Sovellusta pyrittiin osittain sen takia vain

pääpiirteittäin luonnostelemaan paperille. Toinen osatekijä paperin käyttämiseen oli se, että missään valmiissa työkalussa ei ole PrimeFacesin kaltaisia komponentteja, joilla ulkoasua voisi suunnitella. Ulkoasu myös muuttui muutaman kerran kehittämisen aikana, niin toimeksiantajan pyynnöstä, kuin uuden paremman komponentin löytymisestä (ohjelmoijan oppimisesta) johtuen. PrimeFacesissa on yli 100 komponenttia ja extension-kirjaston mukana niitä saa vielä paljon lisää.

Työn valmistuttua ohjelmoija tunsi lähes kaikki PrimeFacesin peruskomponentit (ei extension-kirjaston) ja osaisi työn päätyttyä sanoa, että mitä kyseisellä komponenttikirjastolla voi tehdä. Nyt myös toimeksiantajan kanssa voisi sovelluksen ulkoasua suunnitella paremmin.

Ongelmallista työn suunnittelussa oli myös se, että kommunikointi toimeksiantajan kanssa käytiin vain sähköpostitse, eikä ohjelmoija/tutkija ole työn valmistumiseen mennessä koskaan nähnyt häntä. Mitä suuremmat rahat ovat kyseessä, sitä suositellumpaa käyttöliittymän suunnitteleminen toimeksiantajan kanssa on, jotta saavutetaan suurempi yhteisymmärrys siitä, millainen sovellus tulee olemaan. Niin vähennetään riskiä siitä, että sovelluksen verifiointi menee toimitusvaiheessa pieleen.

8.4 Tulokset

Tuloksena saatiin prosessi ja työkalut, joita käyttämällä voidaan kehittää Java EE -sovellus yhden ohjelmoijan organisaatiossa.

Koska työssä pyrittiin esittämään erityisesti niitä kohtia, joita ei ollut, tai ei ollut riittävän selkeästi, tai tarpeeksi syvällisesti esitetty kirjallisuudessa, voi tätä opinnäytetyötä käyttää oppikirjojen lisänä, jos on kiinnostunut Java EE -teknologioista. Jos samankaltaisia tuloksia haluaa, suosittelen aloittamaan Janne Kuhan (2008) kirjasta ja tämän jälkeen käyttöliittymäpuolen kehittämi-

sessä jatkavan tämän opinnäytetyön ohjeilla, jos JSF soveltuu ohjelmistonkehittäjälle/-jille. Käyttöliittymän ulkoasun suunnittelun voisi tehdä tarkemmin. Myös filttereiden, convertereiden ja validaattorien luomisessa ja ymmärtämisessä tästä työstä voi olla apua.

Tutkimuksella saatiin tietämystä eri ohjelmistoprojektin roolien toiminnasta ja huomattiin, että yhden ohjelmoijan organisaation on mahdollista kehittää laaja Java EE -sovellus. Opittiin myös PrimeFacesin komponenteista ja osataan varautua nyt myös siihen, että kaikkia vaatimuksia ei saada vaatimusten määrittelyssä kerättyä. Tällöin esimerkiksi, jos projektin aikataulu on kriittinen, vaikka laskutuksen takia, osataan varata aikaa sille, kun joku vaatimus on jäänyt löytymättä. Eniten uutta teknologioiden osalta opittiin JSF:stä.

LÄHTEET

Annotations. 2010. Viitattu 27.4.2014.
http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html

Apache License. 2004. Version 2.0. Viitattu 24.4.2015.
http://www.primefaces.org/license

Cagatay Civici. N.D. PrimeFaces User Guide 5.2. First Edition. Viitattu 27.4.2015.
http://www.primefaces.org/docs/guide/primefaces_user_guide_5_2.pdf

Class FacesExceptionFilter. 2014. Viitattu 27.4.2015.
http://omnifaces.org/docs/javadoc/2.0/org/omnifaces/filter/FacesExceptionFilter.html

Cockburn, A. 2008. Using Both Incremental and Iterative Development. STSC CrossTalk. USAF Software Technology Support Center 21 (5): 27–30.

Core J2EE Patterns - Data Access Object. 2002. Sun Microsystems, Inc. Viitattu 15.12.2014. http://www.oracle.com/technetwork/java/dataaccessobject-138824.html

Dick, B. & Swepson, P. 2013 Action research FAQ: "frequently asked questions". Viitattu 27.4.2015. http://www.aral.com.au/resources/arfaq.html

EJB 3.2 Expert Group. 2013. JSR-000345 Enterprise JavaBeans Specification. Version 3.2 Final Relase. Viitattu 27.4.2015.
https://jcp.org/aboutJava/communityprocess/final/jsr345/index.html

Gamma, E., Helm, R., Johnson R. & Vlissides J. 2000. Design patterns: Elements of Reusable Object–Oriented Software. Boston: Addison Wesley.

Hovi, A., Huotari, J. & Lahdemäki, T. 2005. Tietokantojen suunnittelu & indeksointi.

Java EE 7 Technologies. N.D. Viitattu 24.4.2015.
http://www.oracle.com/technetwork/java/javaee/tech/index.html

Java Persistence/What is JPA? N.D. Viitattu 24.4.2015.
http://en.wikibooks.org/wiki/Java_Persistence/What_is_JPA%3F

Java Persistence 2.1 Expert Group. 2013. JSR 338: Java(tm) Persistence API. Version 2.1, Final Relase. Viitattu 24.4.2015.
https://jcp.org/aboutJava/communityprocess/final/jsr338/index.html

Java Server Faces Specification. 2013. Oracle America, Inc. Version 2.2.

Jendrock, E., Cervera-Navarro, R., Evans, I., Gollapudi, D., Haase, K., Markito, W., Srivathsa, C. 2013. The Java EE 6 Tutorial. Viitattu 27.4.2014. http://docs.oracle.com/javaee/6/tutorial/doc/gipjg.html

JSR-299 Expert Group. 2009. JSR-299: Context and Dependency Injection for the Java EE Platform. Final Relase. Viitattu 27.4.2015. https://jcp.org/aboutJava/communityprocess/final/jsr346/index.html

Kruchten, P. 1995. Architectural Blueprints — The "4+1" View Model of Software Architecture. IEEE Software 12 (6), pp. 42-50.

Krug, S. 2006. Don't Make Me Think–A Common Sense Approach to Web Usability. Second Edition.

Kuha, J. 2008. Tehokas Java EE – sovellustuotanto. Porvoo: Docendo.

OmniFaces – What is OmniFaces? N.D. Viitattu 27.4.2015. http://omnifaces.org/

OmniFaces – Full Ajax ExceptionHandler. N.D. Viitattu 27.4.2015. http://showcase.omnifaces.org/exceptionhandlers/FullAjaxExceptionHandler

PermGen space error – Glassfih Server. 2011. Viitattu 24.4.2015. http://stackoverflow.com/questions/7683434/permgen-space-error-glassfish-server

Sengul, S., Gish, J.W., Tremlett, J.F. 2000. Building a Service Provisioning System Using The Enterprise JavaBean Framework.

Toimintatutkimus. 2015. Wikipedia, Vapaa tietosanakirja. Päivitetty 10.2.2015. Viitattu 27.4.2015. http://fi.wikipedia.org/wiki/Toimintatutkimus

Vaadin FAQ. N.D. Viitattu 24.4.2015. https://vaadin.com/faq

Vasilev, Y. 2008. Querying JPA Entities with JPQL and Native SQL. Viitattu 24.4.2015. http://www.oracle.com/technetwork/articles/vasiliev-jpql-087123.html

Vogel, O. 2011. Software Architecture. 1st Edition. Germany: Springer Verlag.

Waterfall Software Development Model. 2014. Viitattu 17.12.2014. http://www.oxagile.com/company/blog/the-waterfall-model/

KUVIOT