Opinnäytetyö

Tietotekniikan koulutusohjelma

Ohjelmistotekniikka

2009

Tommi Hännikkälä

ITMill Vaadin – Spring Framework -integraatio

TURUN AMMATTIKORKEAKOULU
TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tietotekniikan koulutusohjelma | Ohjelmistotekniikka

Joulukuu 2009 | 65 sivua

Ohjaaja: TkL Jari-Pekka Paalassalo

Tommi Hännikkälä

ITMill Vaadin – Spring Framework -integraatio

Tämän opinnäytetyön tarkoituksena oli luoda Affecto Finland Oy:lle integraatiokirjasto ITMill Oy:n Vaadin -näyttökomponenttikirjaston ja Spring Framework -ohjelmistokehyksen välille. Näin pyrittiin tekemään Vaadin-ohjelmistot valmiimmaksi Enterprise-ympäristöihin.

Kirjasto luotiin asiakasprojektin oheistuotteena ja sen tekeminen kesti noin vuoden. Sinä aikana integrointitarpeen ratkaisuksi pohdittiin useita erilaisia vaihtoehtoja, kuten Vaadimen oma ohje Spring Frameworkin integrointiin. Erillinen kirjasto valittiin kuitenkin toteutettavaksi ratkaisuksi. Työ sisälsi suunnittelun, mallinnuksen, toteutuksen, dokumentoinnin ja parhaiden tapojen esittävän esimerkkiohjelmiston.

Ratkaisun tärkeimpiä ominaisuuksia olivat yleiskäyttöisyys, kehittämistyön nopeutuminen ja riippumattomuus käytettävästä järjestelmästä. Toteutustavoista tärkeimpiä olivat hyvien ohjelmointitapojen, arkkitehtuurien ja hyväksi todettujen ohjelmistotuotannon työtapojen käyttö.

Tuloksena syntyi vuoden kehitystyön jälkeen kirjasto, joka käyttää Javan uusimpia tekniikoita hyväkseen. Se tuo Vaadin-sovellukseen Spring Frameworkin kautta palveluorientoituneen kolmikerrosarkkitehtuurin, uudelleenkäytettävän liiketoimintalogiikan ja mahdollistaa integraation muihin palveluihin.

ASIASANAT:

Spring, framework, Vaadin, integraatio

BACHELOR THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree Programme in Information Technology | Software Engineering

December 2009 | 65 pages

Instructor: Jari-Pekka Paalassalo, Lic.Tech.

Tommi Hännikkälä

ITMill Vaadin – Spring Framework -integration

Purpose of this thesis was to create an integration library between Vaadin user interface library and Spring Framework -library. Work was done for Affecto Finland Ltd. Library was supposed to make Vaadin applications more Enterprise-ready.

Library was created as by-product of the customer project and it took a year to get done. During that time there were lots of choices were considered to be a solution of a need of integration. One of the choices was official instructions to integrate Vaadin. However, external library was selected to be implemented. Work included designing, modeling, implementation, documentation and an application demonstrating best practices.

Main features of the solution were general usability, decreasing the time used for development and independence of used methods of used environment. Most important methods of implementation were use of best programming, architectural methods and programming practices.

After a year of work the result was a library which uses latest technologies of Java language. It brings service oriented architecture (SOA), reusable business logic layer and enables remote integrations to Vaadin applications by Spring Framework.

KEYWORDS:

Spring, framework, Vaadin, integraatio

SISÄLTÖ

KUVAT

ESIMERKIT

KÄSITE- JA LYHENNELUETTELO

Apache Maven	Maven on työkalu Java EE -projekteille, tehty tekemään paketinrakentamisprosessi helpommaksi. Ohjelma pakottaa projektin tiettyyn kehitysstandardiin ja tätä kautta lyhentää kehittämiseen kuluvaa aikaa.
API	Application Programming Interface esittelee toiminnot, joita kutsuva ohjelma voi kirjastosta käyttää.
Java EE	Lyhenne sanoista Java 2 Enterprise Edition. Se on Sun Microsystemsin määrittelemä ohjelmistokehitysalusta monitasoisien Java-sovellusten kehittämiseen.
Jatkuva integraatio	Ryhmä ohjelmiston kehitystapoja, jotka pyrkivät tehostamaan kehitystä ja parantamaan tuotteen laatua. Käytännössä kehittäjät vievät tekemänsä muutokset versiohallintaohjelmistoon, josta jatkuvan integraation ohjelmisto käy hakemassa uusimman version ja ajaa aiemmin määritetyt testit. Kun testit eivät mene läpi, ohjelmisto ilmoittaa viimeisen muutoksen tehneelle kehittäjälle virheestä.
JDBC	Java-rajapinta, joka määrittelee standardin tavan käyttää relaatiotietokantaa.
JNI	Lyhenne sanoista Java Native Interface. Rajapinta mahdollistaa natiivin eli konekielisen ohjelmistokoodin käyttämisen Java-ohjelmissa. Koodi voi olla alun perin kirjoitettu vaikkapa C-, C++- tai Assembly-kielellä.
ORM	Object-relational Mapper on tekniikka muuntaa tyyppiyhteensopimatonta tietoa relaatiotietokannan ja olio-orientoituneen ohjelmointikielen välillä.
Portaali	Portaali on www-pohjainen ohjelmisto, joka yleisesti tarjoaa personalisoinnin, autentikaation, sisällön tuonnin muista lähteistä

	ja tarjoaa esityskerroksen informaatiojärjestelmille.
Portlet	Portletit ovat Java-portaalissa ajettavia ohjelmia, jotka tuovat tietyn osan sisällöstä portaalin luomaan sisältöön. Portletit ovat portaalien hallinnoimia, portaali käsittelee pyynnön ja tuottaa dynaamisen sisällön.
Transaktio	Tietokantatoimintojen suojausmekanismi, jolla on kaksi tehtävää: luoda suojattu tila, jossa voidaan suorittaa useita kyselyjä ja virheen tapahtuessa palataan transaktion alkua olleeseen tilaan; suojata tietokantaa yhtäaikaiselta muuttamiselta, joka ilman eristämistä aiheuttaisi virheen.
Unicode	Merkistöstandardi, joka sisältää suuren osan maailman kirjoitettujen kielten merkeistä.
Web Service	W3C-konsortiumin määrittelemä rajapinta, joka on suunniteltu verkon yli tapahtuvaan kommunikointiin koneiden välillä. Käytännön sovelluksissa tieto lähetetään XML-muodossa.
XML	eXtensible Markup Language on merkintäkieli, jossa tieto ryhmitellään tagien sisään. XML:llä voidaan määritellä tietueita muotomäärittelyn (schema) avulla.
Yksikkötestaus	Yksikkötestaus on tapa todentaa moduulikohtaisesti ohjelman toimivuus automaattitestien avulla. Testeihin on ohjelmoitu määriteltyjä käyttötapauksia, joita vastaan moduulit testataan.

1 Johdanto

Affecto on Pohjoismaiden suurin Business Intelligence -toimittaja. BI-ratkaisujen lisäksi yhtiö tarjoaa paikkatietoratkaisuja, ohjelmistopalveluita sekä sisällön- ja dokumenttienhallinnan ratkaisuja. Toimipisteitä yhtiöllä on kahdeksassa maassa: Suomessa, Virossa, Liettuassa, Latviassa, Ruotsissa, Norjassa, Tanskassa ja Puolassa. [1]

Työn tarkoituksena on tehdä integraatio ITMill Oy:n Vaadin -työkalun ja Spring Framework -ohjelmistokehyksen välille. Idea integraatioon tuli selvitystyöstä asiakasprojektiin. Työstä oli aiemmin tehty ensimmäinen versio, josta piti saada uudelleenkäytettyä mahdollisimman paljon uuteen järjestelmän. Tiedettiin myös, että projektiin tullaan tekemään useita integraatioita muihin järjestelmiin. Tällöin todettiin, ettei Vaadin, joka kesällä 2008 oli vielä ITMill Toolkit -nimellä, sopinut täysin aiottuihin toteutusmalleihin. Haluttiin luoda kirjasto, joka parantaa yhteensopivuutta toteutustapojen välillä ja toisi Vaadimen osaksi yleisesti projekteissa käytettyjä malleja.

Tämän opinnäytetyön tuloksena on tarkoitus luoda integraatiokirjasto, joka tuo mahdollisuuden laajempaan arkkitehtuuriin ja näyttökomponenteista irrotettuun liiketoimintalogiikkaan. Tällä tavalla on tarkoitus saada kehitettäviin projekteihin Vaadinmaista suoraviivaisuutta ja sitä kautta lyhentää toteutusaikaa. Monitasoisempi arkkitehtuuri mahdollistaa liiketoimintalogiikan yksikkötestauksen, joka lisää toimintavarmuutta sovelluksissa. Lopullisen version tulee olla mahdollisimman pitkälle olio-ohjelmoinnin sääntöjen mukainen ja käyttää yleisesti hyväksi havaittuja suunnittelumalleja.

2 Arkkitehtuuri

Arkkitehtuuri määrittää ohjelmiston rakenteen ja sen osien välisen kommunikaation korkealla tasolla. Tällä tavalla pyritään hallitsemaan ohjelman ymmärrettävyyttä ja parantamaan ylläpidettävyyttä. [2] Ymmärrettävyyttä lisää Javan nimeämiskäytäntöjen noudattaminen, kuten myöhemmin tarkemmin esiteltävä JavaBeans API. Kun kehittäjillä on yhteiset säännöt, muiden kehittäjien luomat kirjastot voidaan tuoda helpommin ohjelman käyttöön. Tällöin myös kehitystyökalut mahdollistavat parempaa käytettävyyttä. [3] Ylläpidettävyys saadaan ohjelmistoon noudattamalla määriteltyä

arkkitehtuurimallia ja yleisiä sääntöjä. Tällaisia ovat esimerkiksi suunnittelumallit, Javan nimeämiskäytännöt ja olio-ohjelmoinnin tuomat ominaisuudet. [4] Tämä opinnäytetyö esittelee MVC- ja kolmikerrosarkkitehtuurin. Väärät valinnat arkkitehtuurissa voivat aiheuttaa hankalasti ymmärrettävän ja ylläpidettävän ohjelmiston, jossa muutokset maksavat paljon. Ohjelmiston kehittäjät pystyvät yleisarkkitehtuurin avulla valitsemaan mahdollisista toimintatavoista parhaan mahdollisen. [2] Tämä luku esittelee työssä käytetyt ajatusmallit.

2.1 Olio-ohjelmointi

Olio-ohjelmointi on ohjelmistokehitystapa, jossa ongelma voidaan jakaa pienempiin osiin. Ratkaisu perustuu itsenäisiin olioihin ja niiden välisiin suhteisiin. Kun yhtä ongelmaa ratkaisemaan tehdyt toiminnallisuudet on ryhmitelty yhdeksi kokonaisuudeksi, luodaan mahdollisuus käyttää ratkaisua uudelleen samalla toteutuksella. Tämä vähentää uuden ominaisuuden toteutukseen käytettyä aikaa. [4] Olio-ohjelmoinnin peruskäsitteisiin kuuluvat käsitteet esitellään seuraavaksi.

Luokka

Luokka on pohja oliolle. Se kuvaa olion tilaa ja toiminnallisuutta. Tila määritellään luokkamuuttujien avulla ja toiminnallisuuden metodien avulla. Luokka sisältää toteutuksen metodeille. Poikkeuksena ovat abstraktit metodit, jotka joudutaan toteuttamaan nykyisen luokan perivässä, ei-abstraktissa luokassa. [5]

Olio

Olio on yksi ilmentymä jostain tietystä luokasta. Ilmentymää voidaan kutsua annetulla nimellä, sen tilaa voidaan muuttaa ja toimintoja suorittaa. Kaikki toiminta kohdistuu kutsuttuun olioon, eivätkä muut saman tyypin oliot itsestään jaa tilaa. Abstrakteista luokista ei voi luoda ilmentymiä. [5]

Luokkamuuttuja

Luokkamuuttuja pitää sisällään jonkin tyyppisen arvon, joka luokkaan on määritetty. Jokainen ilmentymä omistaa omat muuttujansa, jolloin myös tila on yksilöllinen tietylle oliolle. Kaikki ilmentymän luokkamuuttujat yhdessä kuvaavat ilmentymän nykyistä tilaa. [5]

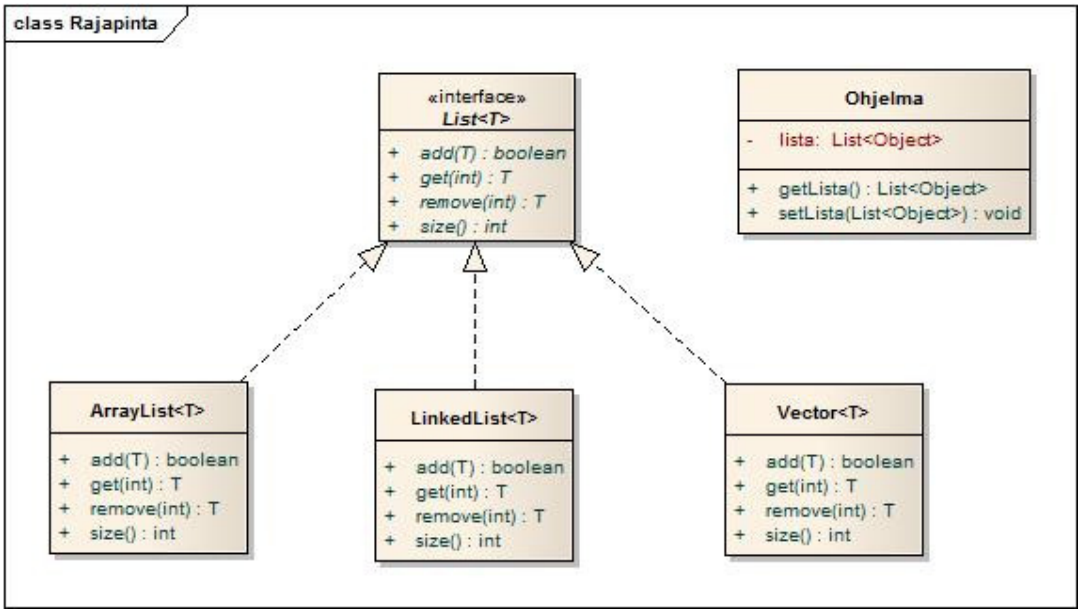
Metodi

Metodi antaa oliolle logiikan. Kaikki olio-ohjelmoinnin toiminnallisuudet on kirjoitettu luokkien metodeihin. Metodit voivat ottaa useita parametrejä sisään ja palauttaa yhden arvon. Perivät luokat voivat ylikirjoittaa metodeja ja tällä tavoin luoda tietylle toteutukselle ominaisen luonteen. Kehittäjät voivat käyttää tätä ominaisuutta uudelleenkäyttääkseen jo luotua ohjelmistokoodia. [5]

Rajapinta

Rajapinta esittelee tyypin toiminnallisuudet ja kuvailee, mitä tietty toiminnallisuus tekee, mutta ei kuitenkaan ota kantaa, miten toiminnallisuus toteutetaan [5]. Tällä tavoin eri toteutus voidaan valita tarpeen mukaan ja kahden saman rajapinnan toteuttavan luokan toteutus on vaihdettavissa.

Kuvassa 1 oleva kaavio esittää Javan mukana tulevia *List*-rajapintaa ja *ArrayList*-, *LinkedList*- ja *Vector*-luokkia. *List*-rajapinnan tarkoituksena on säilyttää kokoelmaa olioita ja mahdollistaa kokoelman käsittely standardilla tavalla. Rajapinnan eri toteutuksilla on erilainen luonne: *ArrayList* on Javan taulukkoon perustuva toteutus, joka ei ole säieturvallinen, yleensä perusnopea. *LinkedList*-toteutuksen ajatus on mahdollistaa pino ja jono, sekään ei ole säieturvallinen. *Vector* on säieturvallinen ja tästä syystä merkittävästi hitaampi kuin *ArrayList* ja *LinkedList*. Kulmasulkeissa oleva *T* kuvastaa tyyppiä, jonka ilmentymiä kokoelma ottaa vastaan. Pääluokassa *Ohjelma* on luokkamuuttuja *lista*. Muuttuja ottaa vastaan minkä tahansa *List*-tyyppisen kokoelman, joka sisältää *Object*-tyyppisiä olioita, esimerkkitapauksessa minkä tahansa sen toteuttavista luokista *ArrayList*, *LinkedList* ja *Vector*. Näin ollen Ohjelma-luokkan sisältävä Java-ohjelma ja sen ajoympäristö määrittelevät, mikä toteutus kehittäjän kannattaa valita. Jos tilanne muuttuu, toteuttavaa luokkaa voidaan vaihtaa, eikä muuhun ohjelmaan tule virheitä.



Kuva 1 Rajapintaesimerkki

Olio-ohjelmoinnissa jokaisella oliolla on tietty tyyppi, joko luokan tai rajapinnan määrittelemä. Kun tiedetään olion tyyppi, tiedetään olion tarjoamat toiminnallisuudet. Jokaisella oliolla on itsenäinen, toisista riippumaton tila. Se määrätään luokkamuuttujien avulla. Toisista riippumattomuuden saa ohitettua määrittämällä muuttuja staattiseksi, jolloin kaikki tyypin objektit saavat fyysisesti saman muuttujan. [3]

Esimerkkinä luokkien itsenäisestä tilasta esimerkkinä luokka Matka [Esimerkki 1], joka sisältää muuttujan *kohde*.

Esimerkki 1 Matka-luokan lähdekoodi

```
/**
 * Matka-demoluokka.
 *
 * @author Tommi Hännikkälä <tommi@hannikkala.com>
 */
public class Matka {

    /**
     * Matkakohde.
     */
    private String kohde;

    public String getKohde() {
        return kohde;
    }

    public void setKohde(String kohde) {
        this.kohde = kohde;
    }
}
```

Kun luodaan kaksi eri *Matka*-tyyppistä oliota, niillä voi olla, kuten reaalimaailmassakin eri kohde. Asetetaan ensimmäiselle *Matka*-oliolle kohteeksi Intia ja toiselle Kanada [Esimerkki 2].

Esimerkki 2 Käynnistävän luokan lähdekoodi

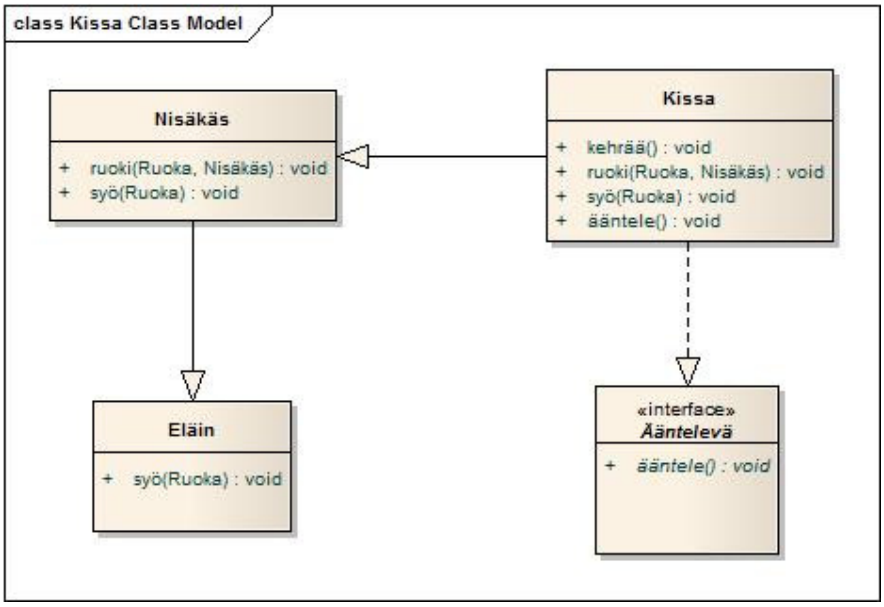
```
/**
 * Matkademo - pääluokka.
 *
 * @author Tommi Hännikkälä <tommi@hannikkala.com>
 */
public class Main {

    public static void main(String[] args) {
        Matka matka1 = new Matka();
        Matka matka2 = new Matka();
        matka1.setKohde("Intia");
        matka2.setKohde("Kanada");
        System.out.println("Ensimmäisen matka kohde on " + matka1.getKohde()
                + " ja toisen matkan kohde on " + matka2.getKohde() + ".");
    }

}
```

Ajettaessa saadaan tuloksena matkojen eri kohteet [Esimerkki 3].

Esimerkki 3 Ajetun ohjelman tulostus

```
Ensimmäisen matka kohde on Intia ja toisen matkan kohde on Kanada.
```

Olio-ohjelmoinnin ominaisuuksiin kuuluu perintä, ylikirjoittaminen ja monimuotoisuus Perinnällä voi luoda yleisluontoisista toimintaluokista uusia, erikoistapauksiin sopivia luokkia ilman, että koko toteutusta pitää kirjoittaa uudelleen. Ylläpito helpottuu, kun muutokset tehdään keskitetysti. Yleisluokat voidaan määrittää abstrakteiksi, jolloin niistä ei itsestään voi tehdä ilmentymiä. Myös metodeja voi merkitä abstrakteiksi, tällöin perivän luokan tulee toteuttaa metodi. Monimuotoisuudella puolestaan tarkoitetaan sitä, että kun luokat, joista oliot luodaan, voivat periä useita alaluokkia ja toteuttaa monia rajapintoja. [4]

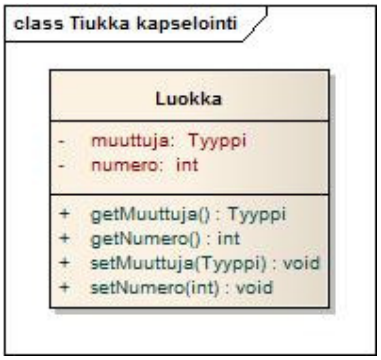


Kuva 2 Kissaesimerkin luokkamalli

Esimerkiksi *Kissa* voi olla luokan *Nisäkäs*-yläluokka, joka puolestaan on luokan *Eläin*-yläluokka. Tällöin *Kissa*-tyyppinen olio on *Eläin*, *Nisäkäs* sekä *Kissa*. *Kissa* voi myös toteuttaa rajapinnan *Äänteleväl*, tällöin olio on edellä mainittujen lisäksi myös *Äänteleväl* ja sen tulee toteuttaa rajapinnan metodit. Yllä olevassa tapauksessa *Kissa*-luokan ei tarvitse toteuttaa ruoki- ja syö-metodeja ollenkaan, koska *Nisäkäs* toteuttaa ne jo [Kuva 2].

Oliopohjaisten kielten arkkitehtuurit luotiin mahdollistamaan kehitettävän ohjelmiston joustavuus ja ylläpidettävyys. Olio-ohjelmointi tarjoaa omat ratkaisutavat ongelmiin. Ne ovat nimeltään tiukka kapselointi, heikko kytkentä ja korkea yhteenkuuluvuus. [5]

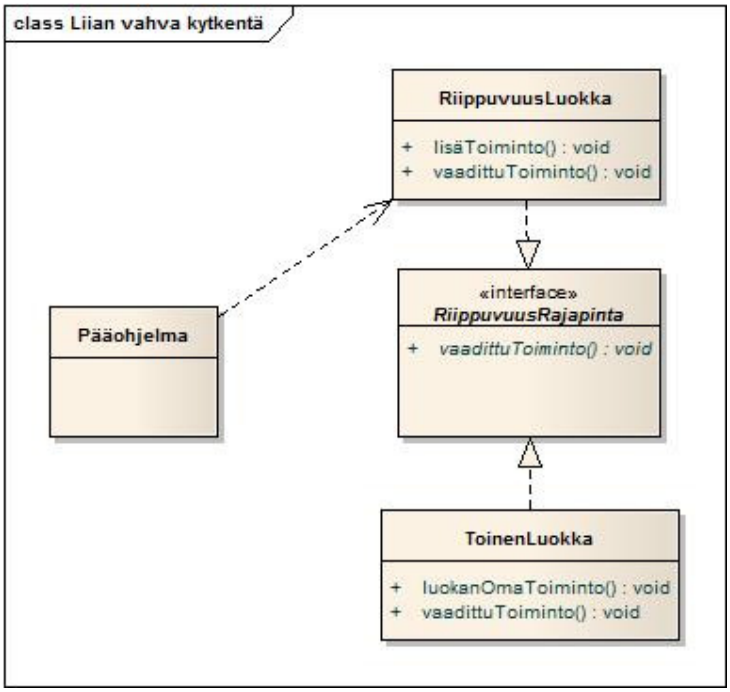
Tiukka kapselointi ohjeistaa piilottamaan näkyvyysmääreillä luokkamuuttujat ja tarjoamaan niiden muuttamiseen metodit. Ajatuksena on mahdollistaa toteutuksen muuttaminen. Javan syntaksissa ei ole mahdollista määritellä muuttujia rajapintaan, ainoastaan metodeja. Riippuvuus luokkamuuttujasta aiheuttaa aina riippuvuuden luokasta. Olio-ohjelmoinnissa pyritään tarjoamaan riippuvuudet rajapintojen kautta. Tästä johtuen luokkamuuttujat tulisi pitää mahdollisimman näkymättömissä ja tarjota luku- ja asetusmetodit. [5] Esimerkkinä *Luokka*-niminen luokka, jossa kaksi yksityistä muuttujaa *muuttuja* ja *numero* [Kuva 3]. Koska molempien muuttujien näkyvyysmääre on asetettu yksityiseksi, ulkopuoliset luokat eivät voi lukea eivätkä kirjoittaa siihen. Tästä johtuen myöskään riippuvuutta luokkamuuttujaan ei pääse syntymään.



Kuva 3 Esimerkkiluokka tiukasta kapselonnista

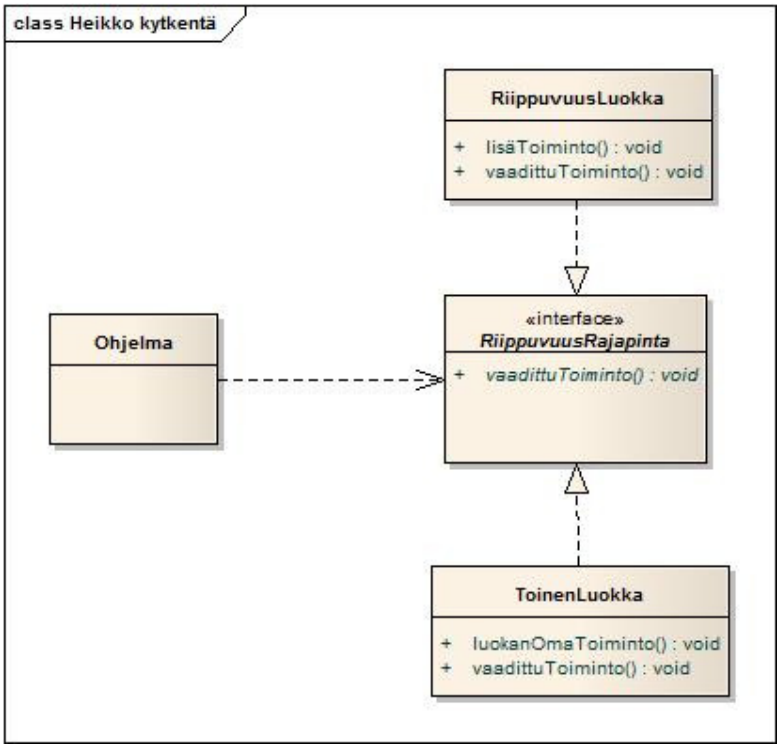
Heikko kytkentä tarkoittaa tapaa, jossa riippuvuuksista tiedetään mahdollisimman vähän. Tämä tuo toteutukseen joustavuutta ja lisää ylläpidettävyyttä, kun riippuvuus tarjoaa ainoastaan tarpeelliset toiminnallisuudet. [5]

Esimerkkinä on kaavio, jossa *Pääohjelma*-nimisessä luokassa on riippuvuus *RiippuvuusLuokka*-tyyppiseen olioon [Kuva 4]. Liiketoimintalogiikka muuttuu siten, että *RiippuvuusLuokan* sijaan halutaan *ToinenLuokka*-tyyppinen olio, jossa on myös *Pääohjelman* tarvitsema metodi *vaadittuToiminto*. Kun riippuvuus vaihdetaan, on mahdollista, että ohjelman jokin muu osa vaatii metodin, joka on ainoastaan *RiippuvuusLuokan* olioissa. Kun näin tapahtuu, ei riitä tyypin korjaaminen vain yhteen paikkaan vaan ohjelma tulee tarkastaa viittauksien varalta.



Kuva 4 Riippuvuus luokan kautta

Olio-ohjelmoinnin perussääntöjen mukaisesti riippuvuus pitäisi olla rajapinnassa [Kuva 5]. Tässä tilanteessa *Ohjelma*-luokka saa suoritettua vaaditun toiminnon, eikä sen näkökulmasta ole merkitystä, mikä toteutus toiminnot suorittaa. Näin on saatu tuotua ohjelmaan riippuvuus heikolla kytkennällä.



Kuva 5 Riippuvuus rajapinnan kautta

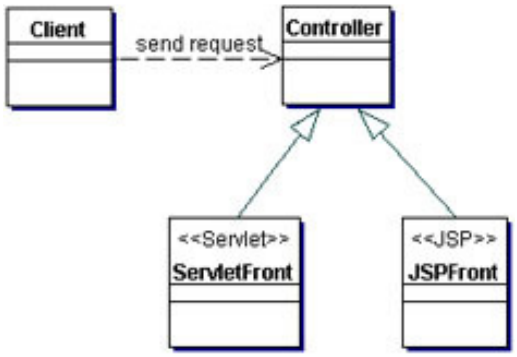
Korkea yhteenkuuluvuus tarkoittaa ajatusta, että yksi luokka pitää sisällään yhteenkuuluvat toiminnallisuudet. Toiminnallisuuden hajoaminen ympäri ohjelmistoa voi hankaloittaa toiminnallisuuksien uudelleenkäyttämistä sekä hankaloittaa ylläpitoa. [5]

Käytännössä tavat johtavat siihen, että riippuvuudet pitäisivät olla rajapintojen kautta. [5]

2.2 Suunnittelumallit

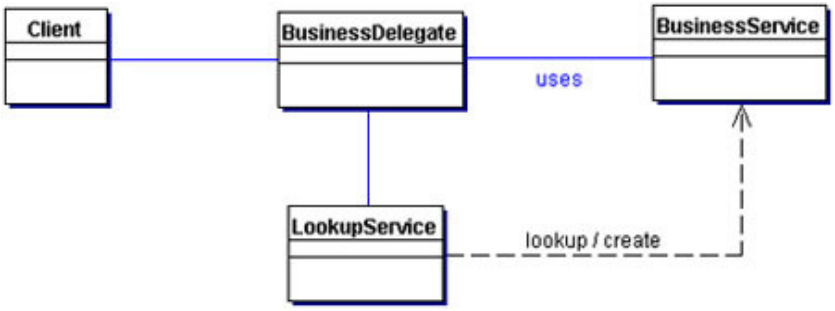
Suunnittelumallit ovat yleisiä, uudelleenkäytettäviä ratkaisuja yleisiin suunnitteluongelmiin. [6] Tässä luvussa esitellään ainoastaan ne Java EE -suunnittelumallit, joita työssä esiintyy.

Front Controller -malli toimii keskitettynä käsittelijänä kaikille pyynnöille. Käytetään, kun tarvitaan yleistä käsittelyä kaikille pyynnöille. Tämä sopii esimerkiksi käyttäjän autentikointiin tai auktorisointiin. Front Controllerilla on merkittävä rooli näytönhallinnassa koko sovelluksen yli. [6] Kuvassa 6 on esitettynä Front Controller -mallin luokkakaavio sekä osien yhteydet toisiinsa.



Kuva 6 Front Controller -suunnittelumallin luokkakaavio [7]

Business Delegate -malli helpottaa liiketoimintalogiikkaa keskittämällä sitä rajapintaan, joka jakaa toiminnallisuuden taustajärjestelmiin. Voidaan käyttää, kun sovelluksesta on useampia yhteyksiä eri järjestelmiin ja tästä syystä näyttöä ohjaava kerros monimutkaistuu merkittävästi. Mallin muihin sovelluksiin kuuluu välimuistimekanismit, joilla vältetään samojen kyselyiden suorittamista etäjärjestelmissä. [6] Kuvassa 7 luokkakaaviona Business Delegate -mallin elementit ja niiden suhteet toisiinsa.

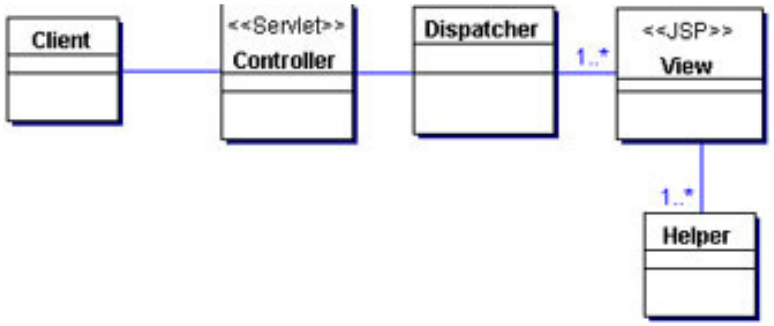


Kuva 7 Business Delegate -suunnittelumallin luokkakaavio [8]

View Helper -mallia käytetään silloin, kun näyttölogiikka alkaa sekoittua liiketoimintalogiikan kanssa tai näyttölogiikka on muuten monimutkaista. Tällöin ylläpito monimutkaistuu merkittävästi ja muutokset maksavat enemmän. View Helperillä saa vietyä liiketoimintalogiikan omaan apuluokkaan, jolloin toiminnoista tulee uudelleenkäytettävää ja modulaarisempaa. [6] Kuvassa 8 on esitettynä View Helper -mallin elementit ja niiden suhteet toisiinsa.

Kuva 8 View Helper -suunnittelumallin luokkakaavio [9]

Dispatcher View tarjoaa ratkaisun ongelmaan, jonka Front Controllerin ja View Helperin ratkaisut aiheuttavat; liiketoimintalogiikka jakautuu osiin ja keskitetty hallinta puuttuu. Tämä johtaa siihen, että toteutus voi löytyä useammasta paikasta ja ylläpito hankaloituu merkittävästi. Ratkaisuna ongelmaan Dispatcher View tarjoaa Front Controllerin ja View Helpereiden yhdistämistä, jolloin Dispatcher View käsittelee asiakkaan pyynnön ja valmistelee dynaamisen näytön vastaukseksi. [6] Kuva 9 esittää Dispatcher View -mallin elementit ja niiden suhteet toisiinsa.

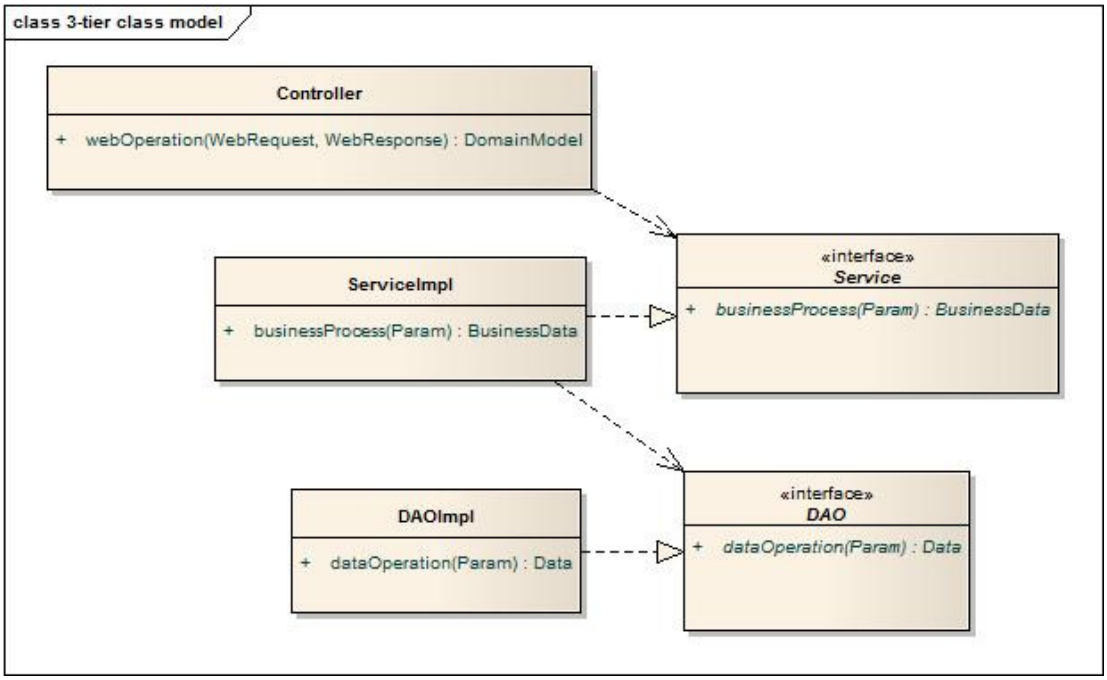


Kuva 9 Dispatcher View -suunnittelumallin luokkakaavio [10]

2.3 Palveluorientoitunut kolmikerrosarkkitehtuuri

Palveluorientoitunut kolmikerrosarkkitehtuuri (service oriented architecture, SOA) on tapa organisoida ja käyttää hajautettuja toimintoja, jotka voivat olla toteutettu eri tekniikoilla ja olla muiden tahojen omistuksessa. Se käyttää hyväkseen olio-ohjelmoinnista tuttua käsitettä heikko kytkentä pyrkien vähentämään epäsuoria riippuvuuksia kerroksien välillä. Palveluorientoituneisuuden arkkitehtuurimalliin tuo ulkoisten järjestelmien integrointien mahdollistaminen; toiminnallisuudet tarjotaan asiakasjärjestelmille yhteen toimivina palveluina. Jokainen palvelu voi olla integroitu useisiin muihin järjestelmiin, josta seuraa paitsi hajautettu infrastruktuuri, myös yhteenkuuluvien toiminnallisuuksien keskittäminen. [11]

Pohjatyönä käytetään Java EE -alustan ajatusta monitasoisesta palvelusta. Ajatuksena on jakaa toiminnallisuus näyttökerrokseen, palvelukerrokseen ja tietovarastokerrokseen. Kuvassa 10 Controller-luokka vastaa näyttökerrosta, Service-rajapinta sekä ServiceImpl-luokka palvelukerrosta ja DAO-rajapinta sekä DAOImpl-luokka tietovarastokerrosta. Liiketoimintalogiikan sijoittamisesta palvelukerrokseen saavutettava etu on modulaarinen arkkitehtuurikerros, jota voidaan käyttää uudelleen muualta sovelluksesta. [12]



Kuva 10 Palveluorientoituneen kolmikerrosarkkitehtuurin toteutus rajapintojen avulla

Näyttö- ja integraatiokerros

Näyttökerroksessa, jota Java EE -suunnittelussa kutsutaan web-kerrokseksi, hoidetaan käyttöliittymään liittyvät asiat. Kerroksen tehtäviin kuuluu myös ottaa asiakassovellukselta tuleva pyyntö vastaan, tarkistaa pyynnön oikeellisuus ja käsitellä se palvelukerrosta apuna käyttäen. Pyynnön tulos, myös mahdollinen virheilmoitus, viedään näyttökerroksen toimesta asiakasjärjestelmälle. Sovelluksen luonteesta riippuen, näyttökerroksen kanssa voi toimia myös integraatiokerros, joka julkaisee palveluja muihin ulkoisiin järjestelmiin. Integraatiokerroksen erona näyttökerrokseen verrattuna on muiden järjestelmien palveleminen käyttäjien sijasta. [12]

Palvelukerros

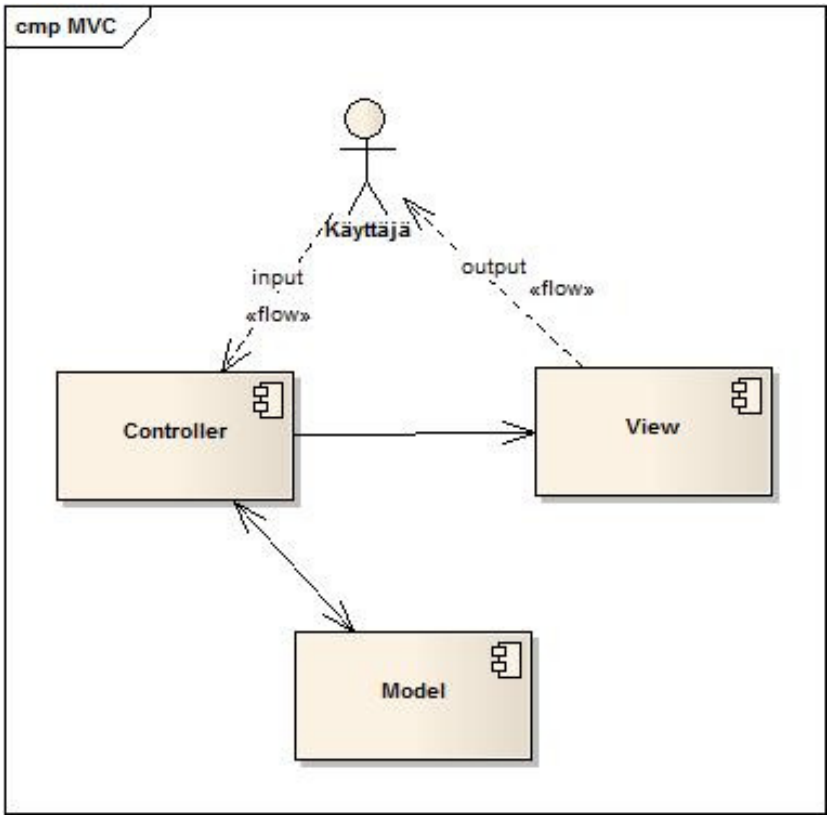
Palvelukerroksessa tehdään kaikki sovelluksen tai palvelun luonteeseen liittyvä toiminnallisuus. Tällöin palvelukerroksen rajapinta esittelee kaikki sovelluksen ominaisuudet. Toiminnallisuus tuodaan palvelukerrokseen, jotta se olisi uudelleenkäytettävissä näyttökerroksesta ja myös mahdollisesta integraatiokerroksesta. [12]

Tietovarastokerros

Palvelukerros käyttää tiedonkäsittelyyn tietovarastokerrosta. Kerroksen tehtäviä on tarjota rajapinta järjestelmän tietovarastoon ja varmistaa tiedon eheys. Kerros voi olla toteutettu tietokantayhteydellä, Web Servicellä integraationa toiseen palveluun tai viestijonolla. [12]

2.4 Model-View-Controller

Model-View-Controller on rakenteellinen olio-ohjelmoinnin arkkitehtuuri, jossa pyritään erottamaan liiketoimintalogiikka näyttötoiminnallisuuksista [Kuva 11]. Etuna erottelussa on liiketoimintalogiikan ja näyttölogiikan yhtäaikainen kehittäminen sekä mahdollisuus erilaisiin tapoihin näyttää tieto. Huonona puolena MVC kasvattaa kehitettävän sovelluksen monimutkaisuutta. Arkkitehtuuri sisältää monia ohjelmoinnin suunnittelumalleja: Front Controller, View Helper, Dispatcher View, Business Delegate ja Observer. [6]



Kuva 11 Model-View-Controller -relaatiomalli

Kontrolleri

Kontrolleri toimii keskitettynä käyttäjän pyyntöjen käsittelijänä Front Controller -suunnittelumallin mukaisesti ja samalla jakaa Business Delegate -mallin mukaisesti varsinaisen liiketoimintalogiikan käsittelyn Model-kerrokselle. Model-kerrokselta vastauksen saatuaan kontrolleri delegoi kerätyt tiedot View-kerrokselle, joka hoitaa tiedon näyttämisen. [6]

Model

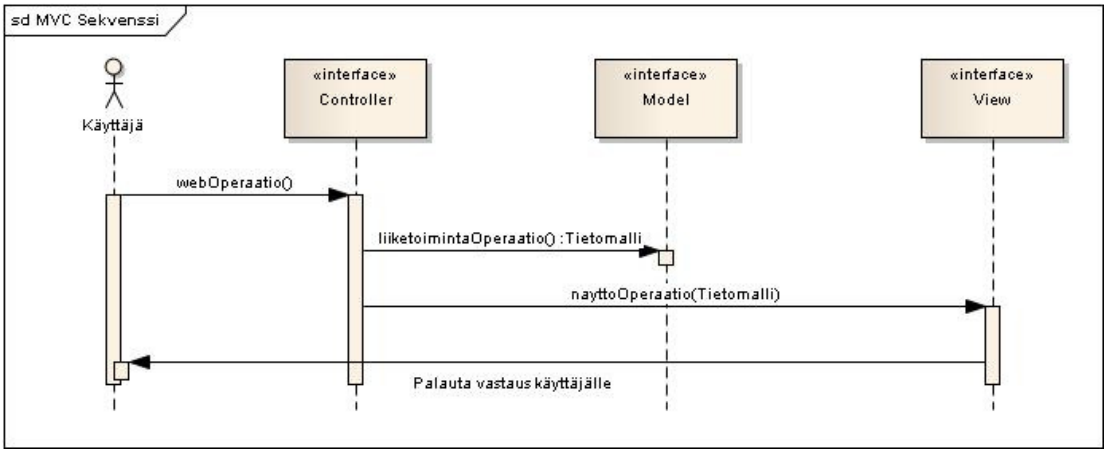
Model-kerros kuvastaa liiketoimintalogiikkaa ja siihen liitettyä tietovaraston käsittelylogiikkaa. Se vastaa tiedon käsittelystä, ylläpidosta ja muutosten tallentamisesta eli käytännössä Model-kerros antaa sovellukselle luonteen. [6] Kolmikerrosarkkitehtuurissa Model vastaa kolmikerrosarkkitehtuurin palvelukerrosta ja tietovarastokerrosta.

View

View ottaa kontrollerilta vastaan tietomallin. Mallin mukaisesti kerros luo näytön ja lähettää sen käyttäjälle.

MVC toimii seuraavien askelien perusteella [Kuva 12] [6]:

1. Käyttäjä aiheuttaa toiminnon sovellukseen. Tämä voi tapahtua vaikka napin painamisella.
2. Kontrolleri ottaa pyynnön vastaan.
3. Kontrolleri herättää käyttäjän toimintaan liittyvän käsittelijän, joka tekee muutokset tietomalliin.
4. Näyttö saa oman tietomallinsa ja luo näytön sen perusteella.
5. Rajapinta jää odottamaan käyttäjän toimia, jolloin ketju alkaa alusta.



Kuva 12 MVC-sekvenssikaavio

2.5 Aspektiohjelmointi

Olio-ohjelmoinnin tarjoama ajatusmalli pyrkii luomaan uudelleenkäytettävää ohjelmistokoodia [5]. Sen tarjoamat suunnittelumallit kuten modulaarisuus, periytyvyys ja monimuotoisuus tukevat periaatetta ja olio-ohjelmointi on saavuttanut suosiota mallintamisessa sekä monimutkaisten järjestelmien toteuttamisessa. Suurissa järjestelmissä käytäntö on osoittanut, että muutosten hallinta on yhä vaikeampaa modularisoida. Pienen muutoksen tekeminen voi aiheuttaa muutoksen useisiin eri moduuleihin. [13] Aspektiohjelmoinnin termein koodi pirstaloituu ja sekoittuu. Nämä asiat hankaloittavat ylläpitoa ja kehittämistä. Sekoittumisella tarkoitetaan tilannetta, jossa yhdellä ohjelman moduulilla on useita riippuvuuksia. Tästä johtuu myös koodin pirstaloituminen. Kun liiketoimintalogiikka on jakautunut useampiin riippuvuuksiin, olio-ohjelmoinnin korkean yhteenkuuluvuuden periaate ei toteudu ja ylläpito hankaloituu merkittävästi. Sekoittuneita ja pirstaloituneita ominaisuuksia kutsutaan poikkileikkaaviksi ominaisuuksiksi. Aspektiohjelmoinnin tavoitteena on parantaa poikkileikkaavien ominaisuuksien modularisointia [14], kuten olio-ohjelmointi modularisoi yleisiä ominaisuuksia.

Aspekteina toteutetaan yleisimmin toissijaisia ohjelman ominaisuuksia. Tällaisia ovat esimerkiksi virheiden kirjoittaminen lokitiedostoihin, tietokantatransaktion aloittaminen tai virheenetsintä. Kaikki toiminnallisuudet, jotka eivät ole välttämättömiä ohjelman luonteen kannalta, ovat tässä kontekstissa toissijaisia. Aspekteilla voi modularisoida nämä kehittäjälle tärkeät ominaisuudet erilleen liiketoimintalogiikasta.

3 Tekniikka

3.1 Java

Java-kieli on Sun Microsystemsin kehittämä ohjelmointikieli ja se julkaistiin vuonna 1995. Java-kielen pääperiaatteisiin kuuluu: ”Kirjoita kerran, aja missä tahansa”. Tämä tarkoittaa, että kerran käännetty ohjelma on mahdollista ajaa missä tahansa käyttöjärjestelmässä ja konearkkitehtuurissa, joka on Java Virtual Machinen (JVM) tukema. Ohjelma käännetään tavukoodiksi, jota ajetaan Javan virtuaalikoneessa. [15]

Java on vahvasti ja staattisesti tyypitetty kieli. Tämä tarkoittaa sitä, että jokaisella oliolla on tyyppi, jota ei voi muuttaa. Java on melkein kokonaisuudessaan oliopohjainen kieli, poikkeuksena tehokkuussyistä ainoastaan sisäiset primitiiviset tyypit: int, long, boolean, byte, float, double, char ja short. [3]

Javalla luodut luokat voidaan sijoittaa pakettirakenteeseen. Jokainen luokka on osa pakettirakennetta, joten paketit voidaan ajetella olevan Java-kielen moduuleja. Paketit auttavat kehittäjää organisoimaan ja jakamaan ohjelmaa luokkien tehtävien mukaan. [16] Erilaisia luokkien tehtäviä ovat esimerkiksi liiketoimintalogiikka-, tietovarasto- ja tiedonsiirtotehtävät. Tämä helpottaa organisoimaan suurikokoista projektia [16].

Java tarjoaa luokan muuttujille ja metodeille määreet, joilla näkyvyyttä voidaan rajoittaa. Määreitä on neljä: *public*, *protected*, oletus ja *private*. *Public*-määre tarkoittaa, että muuttuja tai metodi näkyy kaikkialle. *Protected*-määreellä merkitty luokan osa näkyy samaan luokkaan sekä luokan aliluokille. Oletusmääre tarkoittaa osan näkymistä samaan luokkaan ja samaan pakettiin, mutta ei aliluokille. *Private*-määre rajoittaa näkyvyyden ainoastaan samaan luokkaan. [3] Olio-ohjelmoinnin kapselointisäännön perusteella luokkamuuttujat saavat aina *private*-määreen. Tällöin voidaan olla varmoja, että muuttujalla ei ole ulkoisia riippuvuuksia. [5]

Kun Java-sovelluksessa tapahtuu poikkeava tilanne, jota toiminto ei pysty käsittelemään, sen tulisi heittää poikkeus, joka on java.lang.Exception-luokan olio. Poikkeava tilanne voi johtua esimerkiksi virheestä tietokantakyselyssä tai ohjelmointivirheestä. Poikkeus tulisi käsitellä sovelluksen sisällä, esimerkiksi web-sovelluksessa käyttäjälle tulisi näyttää virheilmoitus ja kirjoittaa tapahtunut virhe lokitiedostoon virheen selvittämistä varten. Käsittelemätön poikkeus kaataa ohjelman. Java sisältää myös virheluokan java.lang.Error. Se kuvastaa virhettä, joka on

järjestelmälle kriittinen, eikä virheenkäsittelijän tulisi ottaa sitä kiinni. Tällaisia ovat esimerkiksi muistin loppuminen tai laiterikko. [17]

3.1.1 Servlet

Java Servlet Technology on Sun Microsystemin kehittämä rajapinta, jolla luodaan web-ohjelmia Java-kielellä. Servletit ovat protokolla- ja alustariippumattomia palvelinohjelmia, jotka käsittelevät web-pyyntöjä ja tuottavat niiden mukaisesti dynaamisesti vastauksia. Servlet-tekniikkaa käytetään usein tarjoamaan Java EE -palveluja monikäyttäjäympäristössä. Sovellukset voivat käyttää muita järjestelmiä, kuten tietokantaa JDBC-rajapinnan kautta tai millä tahansa ohjelmointikielellä toteutettua Web Serviceä. [18]

Servletien elinkaari jakautuu viiteen osaan: lataaminen, alustus, palvelu, tuhoaminen ja roskien keruu. Yksi web-applikaatio voi sisältää yhden tai useamman servletin. Pyynnön ohjaaminen tietylle servletille tapahtuu Servlet API:n määrittelemissä asetuksissa. Servlet pystyy käsittelemään kaikki seitsemän HTTP-pyyntötyyppiä: GET, POST, HEAD, TRACE, DELETE, OPTIONS ja PUT. Kukin näistä ohjautuu omana metodiinsa servlet-luokassa, jolloin kehittäjä pystyy käyttämään Servlet API:n rajapintoja pyynnön ja servletin parametrien hakemiseen. [18]

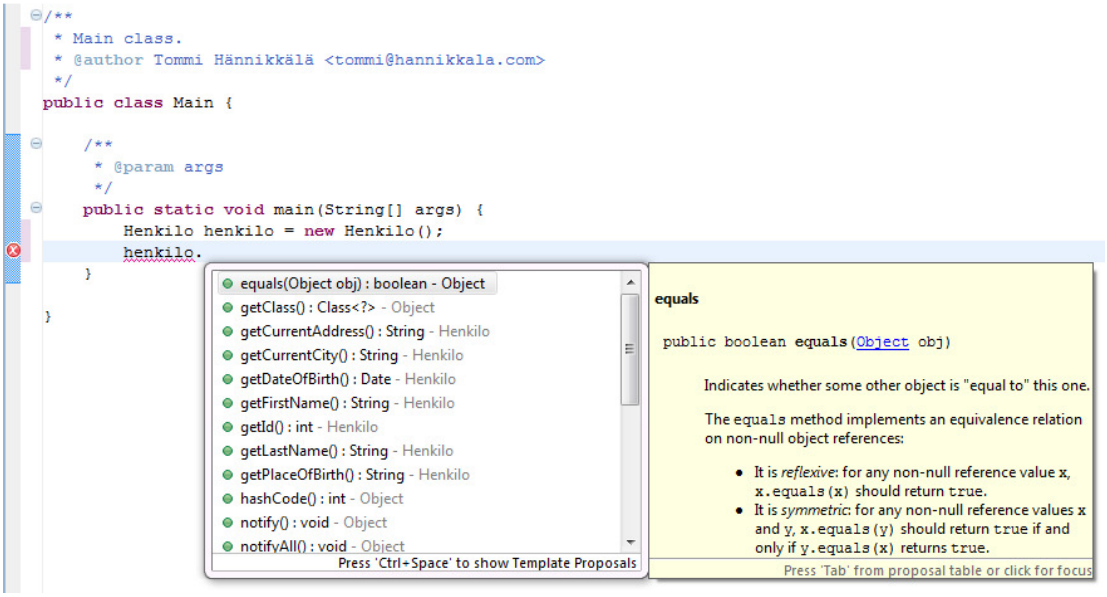
Tässä opinnäytetyössä luotavaa integraatiokirjastoa käytetään, kun luodaan www-sovelluksia Vaadin-servletiä pohjana käyttäen. Servlet on johtava tekniikka www-sovelluksien luontiin Java-kielellä, joten valinta on ilmeinen.

3.1.2 JavaBeans

JavaBeans on Sun Microsystemsin määrittelemä rajapinta ja nimeämiskäytäntökokoelma. Tästä lähtien JavaBean-oliota kutsutaan nimellä Java-papu. Pavut ovat uudelleenkäytettäviä ohjelmistokomponentteja, joka eivät sisällä toiminnallista logiikkaa ollenkaan, ainoastaan muuttujia ja niitä vastaavat muuttujan asetus- ja lukumetodit. Nimeämiskäytäntö määrittelee seuraavia asioita [19]:

- Luokkien nimet kirjoitetaan isolla alkukirjaimella ja sanan vaihtuessa tulee iso kirjain. Esimerkiksi *HyvinLaajaLuokka*.
- Metodien nimet kirjoitetaan pienellä alkukirjaimella ja sanan vaihtuessa tulee iso kirjain. Esimerkiksi *taallaTehdaanAsioita(Parametri p)*.
- Muuttujien, niin luokka- kuin paikallistenkin muuttujien nimet, kirjoitetaan samoin kuin metodien nimet. Esimerkiksi *Tyyppi muuttuja = new Tyyppi();*.

Pavulla voi kuvata yhtä tietuetta, esimerkiksi tietokannan riviä. Muitakin tapoja tietueen varastointiin on, mutta ohjelmointiympäristöt osaavat lukea luokan metodit, joten yhtä muuttujaa vastaavat asetus- ja lukumetodit tulevat esiin paremmin kuin Javan kokoelmaluokkien vastaavat metodit. Ainoastaan pavun metodit luettaessa tietueen sisältö ja yksittäinen tieto tulee helpommin käyttöön. Kuvassa 13 on luokka *Henkilo*, jossa ominaisuudet *currentAddress*, *dateOfBirth*, *firstName* ja *id*.



Kuva 13 Java-pavun käsittely kehitysympäristössä

3.1.3 Annotaatiot

Annotaatiot esiteltiin Javan versiossa 1.5. Ne ovat tapa lisätä metatietoa luokille, metodeille, muuttujille, parametreille ja paketeille. Annotaatiot eivät tee itsessään mitään, mutta metatiedot voidaan lukea luokista ja taustalogiikka voi käyttää niihin asetettuja tietoja ajon aikaisesti. Annotaatioihin voi luoda metodeja ja niissä on mahdollista olla oletusarvo. Metodit voivat palauttaa ainoastaan jonkin primitiivityypin, String-tyyppisen merkkijonon, luokan tai enum-listan. Metodien arvot voidaan asettaa käytettäessä annotaatiota, eikä niitä voi ajonaikaisesti muuttaa. [20]

Esimerkkinä on luokka, jossa annotaatio *Resource*. Sen sisällä *Generated*-annotaatiolla merkitty muuttuja ja *Depredicated*-annotaatiolla merkitty metodi. *Generated*-annotaatiossa on myös esimerkkiarvo, joka voidaan lukea ajonaikaisesti. Kehitysympäristö osaa tulkita *Depredicated*-annotaation vanhentuneen toiminnon merkiksi, tästä syystä metodin nimi on yliviivattu [Kuva 14].

```java
/**
 * Annotaatiodemon pääluokka.
 *
 * @author Tommi Hännikkälä
 */
@Resource
public class AnnotaatioDemo {

    @Generated("arvo")
    protected int muuttuja;

    @Deprecated
    public void metodi() {
        // Liiketoimintalogiikkaa
    }
}
```

Kuva 14 Esimerkki annotaatioilla merkitystä luokasta

3.1.4 AspectJ

AspectJ on aspektiohjelmoinnin toteutus Java-kielelle. Sillä on mahdollista modularisoida Se tuo kieleen yhden uuden käsitteen, liittymäkohdan sekä uusia rakenteita, liitoskohtamäärittelyn (pointcut), kehotteen (advice), tyyppien väliset deklaraatiot (inter-type declarations) ja aspektin (aspect). [21]

Aspekti on AspectJ:ssa modulaarisuuden yksikkö. Se käyttäytyy luokan tavoin, mutta voi sisältää myös liitoskohtamäärittelyjä, kehotteita sekä tyyppien välisiä deklaraatioita.

AspectJ:n dynaamiset osat ovat liittymäkohta, liitoskohtamäärittely sekä kehote. Liittymäkohta on tarkasti määritelty kohta ohjelmassa. Liitoskohtamäärittely valitsee tietyt liittymäkohdat ja arvot niissä kohdissa. Kehote on koodimoduuli, joka ajetaan kun ohjelman ajossa saavutaan liittymäkohtaan. [21]

Tyyppien väliset deklaraatiot antavat kehittäjälle mahdollisuuden luoda tiloja, jotka leikkaavat usean luokan yli tai muuttaa periytymissuhdetta luokkien välillä. Luokkien toteutusta ei tarvitse muuttaa ollenkaan, kaikki muutokset toteutetaan aspektiin.

3.2 Vaadin

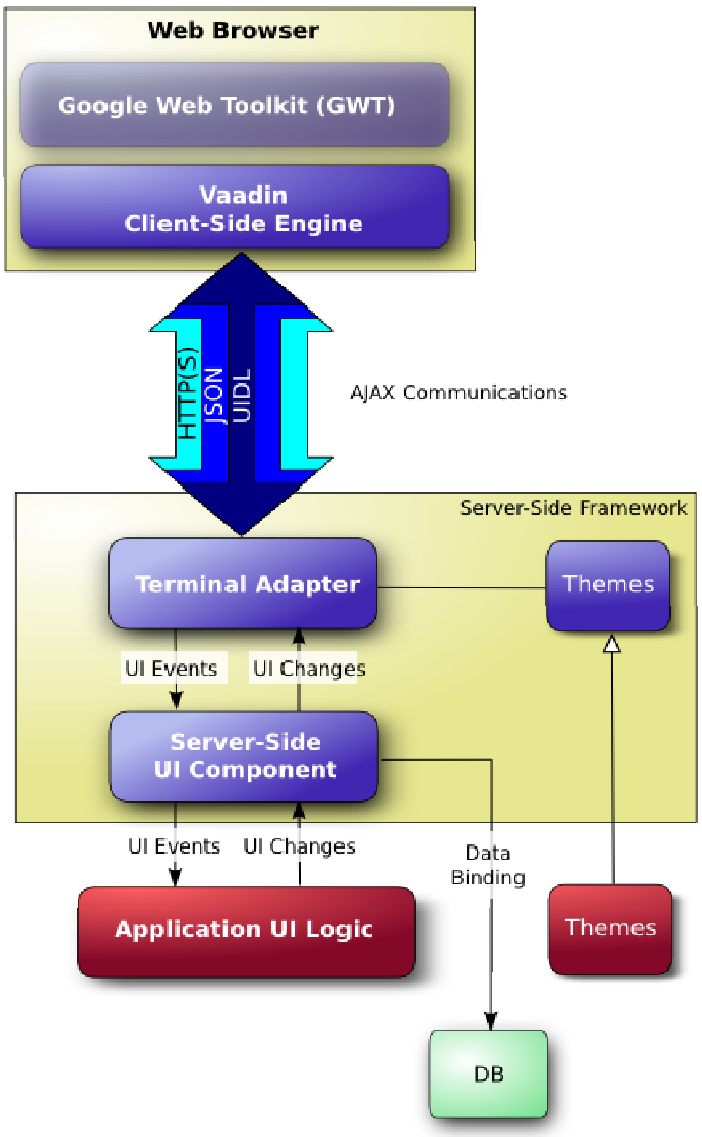
Vaadin on Javalla toteutettu kehysohjelmisto, jolla voi luoda www-ohjelmistoja ja dynaamisia käyttöliittymiä ainoastaan Java-kielellä. Sen on luonut turkulainen ITMill, joka kehittää Vaadinta edelleen. Kirjasto on entisiltä nimiltään Millstone ja myöhemmin ITMill Toolkit. Vuonna 2007 ITMill Toolkit julkaistiin avoimen lähdekoodin Apache 2.0 -

lisenssillä, joka mahdollistaa myös suljettujen sovellusten kehittämisen. Vaadin pohjautuu Google Web Toolkitiin ja on käytettävissä yleisimmillä selaimilla. Vaadin-ohjelmistoja voi luoda niin servlet-tekniikalla kuin portlet-tekniikalla; Vaadimen sisäinen arkkitehtuuri tukee tätä mahdollisuutta. [22]

Vaadimen hyviin puoliin kuuluu nopea käyttöliittymien luonti ja asiakaspuolen toteutus, joka on muilla yleisillä tekniikoilla hankala toteuttaa. Ohjelmisto sisältää myös hyvin suoraviivaisen ohjelmistoarkkitehtuurin, jota integraatiossa Spring Frameworkin kanssa haluttiin tuoda esiin mahdollisuuksien mukaan. Huonoja puolia Vaadimessa on kieliversioinnin puuttuminen eikä se myöskään itsessään mahdollista irrotettavaa, monikäyttöistä liiketoimintalogiikkaa, antaa kuitenkin kehittäjälle mahdollisuuden jonkin kolmannen osapuolen kirjaston käyttöön. Yleisesti Javalla toteutettaviin enterprise-web-sovelluksiin halutaan mahdollistaa integrointien mahdollisuus, uudelleenkäytettävä liiketoimintalogiikka on välttämätön, jotta esityskerros voidaan integraation osalta toteuttaa vaikkapa Web Servicenä. Tästä johtuen Vaadin on pelkästään soveltumaton yksinään suurempien yritysten ohjelmiston pohjaksi.

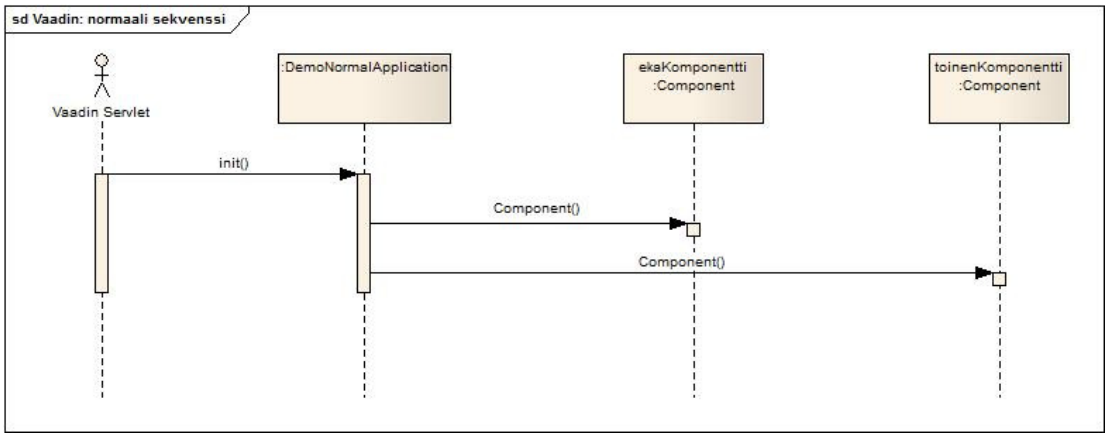
3.2.1 Arkkitehtuuri

Vaadimella luotua ohjelmaa ajetaan Java www -palvelimella servletinä, joka käsittelee HTTP-pyyntöjä. Ohjelman alustuksessa asetetaan käyttöliittymään komponentteja, joille asetetaan kuuntelijoita. Kuuntelijoita voi asettaa erilaisiin tapahtumiin, kuten nappulan painamiseen tai hiiren liikkeeseen tietyn komponentin yli. Kun tietty tapahtuma toteutuu, Vaadimeen liittyvä Terminal Adapter ottaa vastaan asiakaspuolelta tulevat pyynnöt, jakaa ne eteenpäin palvelinpään tapahtumakäsittelijöille, jotka ovat kehittäjän luomia. Käsittelijät käyttävät apuna liiketoimintalogiikkaa ja lähettävät muutoksia Terminal Adapterin kautta. Asiakaspään Vaadin-komponentit tekevät halutut muutokset käyttöliittymään. Kuvassa 15 esimerkkikaavio Vaadimen toiminnasta. [23]



Kuva 15 Kaavio Vaadimen arkkitehtuurista [23]

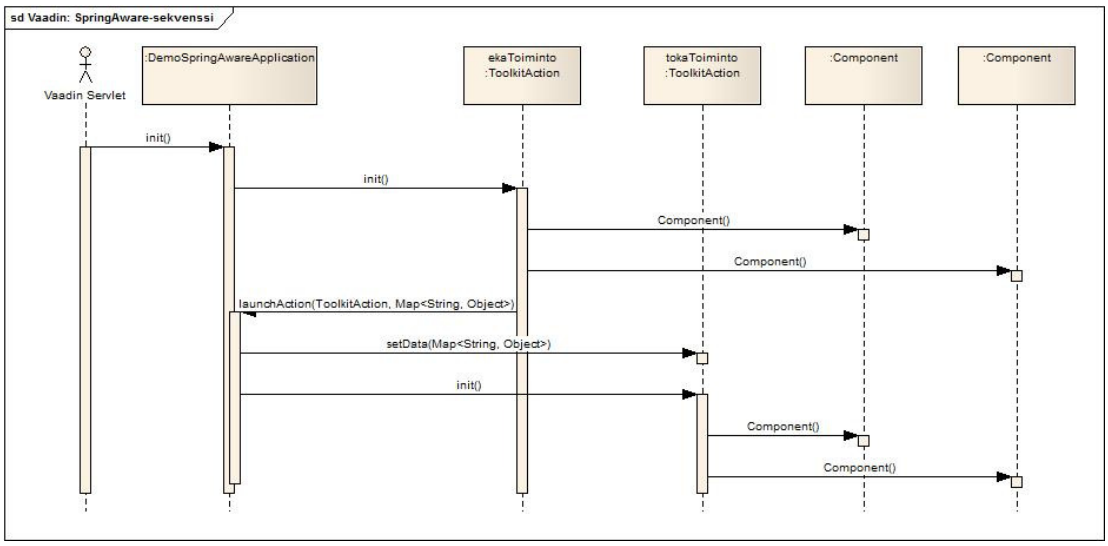
Luodun ohjelman sekvenssi kulkee Vaadimen kaksitasoisessa, tavallisessa arkkitehtuurimallissa siten, että Vaadin-ohjelmaan lisätään näyttökomponentteja [Kuva 16]. Kuvaa on yksinkertaistettu, sillä oikeassa tilanteessa komponentit voivat olla ryhmittelyelementtejä, jolloin ne voivat olla sisäkkäin. Jonkin toiminnon kuuntelijassa voidaan käyttöliittymää päivittää.



Kuva 16 Vaadimen käyttöliittymän kaksitasoinen alustussekvenssi

Kolmitasoiseen arkkitehtuuriin luotiin toimintotaso ohjelman ja näyttökomponenttien väliin [Kuva 17]. Tarkoituksena on saada erotettua eri näytöt keskenään ja toimintotasolla tuodaan myös Spring Frameworkistä tuotavat toiminnallisuudet näyttökomponettien käyttöön. Toiminnosta toiseen siirryttäessä on mahdollista lähettää alustusparametrejä, joilla lähettävä toiminto voi mahdollistaa seuraavan näytön erikoispiirteitä.

Asiakasprojektissa oli tarve tarjota ohjelmalle pohjaelementti, jolla saadaan päivitettyä koko sivua. Integraatioon luotiin mahdollisuus piirtää yhden toiminnon luoma käyttöliittymä haluttuun ryhmittelykomponenttiin.



Kuva 17 Vaadimen käyttöliittymän kolmitasoinen alustussekvenssi

3.2.2 AJAX ja JSON

AJAX on akronyymi sanoista Asynchronous JavaScript and XML. Kyseessä on usea vanha tekniikka yhdessä luomassa uusia tapoja toimia. AJAXilla voi luoda web-käyttöliittymään dynaamisia toiminnallisuuksia, joissa ei tarvitse päivittää koko sivua kerralla, vaan ainoastaan halutut osat. AJAX-kutsut ovat asynkronisia eli ne suoritetaan taustalla siten, ettei muun käyttöliittymän toiminnot hidastu kutsuista. Tekniikan huonoihin puoliin kuuluu JavaScript-kielen heikko standardointi, jonka vuoksi lähes jokaisen valmistajan selaimessa on erilainen tulkki. Kehittäjälle tämä aiheuttaa usein haasteita. AJAXin käyttö aiheuttaa heikkoa käytettävyyttä tilanteissa, jossa käyttäjä on poistanut JavaScript-tuen pois päältä. Myöskään monet hakurobotit eivät osaa käsitellä AJAXia oikein. [24] Vaadin käyttää AJAX-tekniikkaa sivujen osittaiseen päivittämiseen.

AJAXin toimintaa voi tehostaa JSON-notaatiolla. JSON on kevyt tiedonsiirtokieli, joka on helposti ihmisen ja koneellisesti luettavissa. Notaatio on standardoitu joulukuussa 1999. Syntaksi määrittelee kaksi erilaista syntaksia, kokoelman avain/arvo-pareja sekä järjestetyn listan arvoja. Käytännössä kaikki nykyaikaiset kielet tukevat näitä tietorakenteita, siispä notaatiosta tulee hyvin ohjemistokieliriippumaton. JSON on alun perin luotu JavaScriptiin, mutta notaatiota on mahdollista käyttää muihinkin käyttötarkoituksiin. Myös XML-kieltä voidaan käyttää tiedonsiirrossa, mutta XML-tunnisteet lisäävät siirrettävän datan määrää huomattavasti. [25]

3.2.3 Google Web Toolkit

Google Web Toolkit on avoimen lähdekoodin ohjelmisto, joka on lisensoitu Apache 2.0 -lisenssillä. Sillä voi luoda AJAX-käyttöliittymän pelkällä Java-kielellä, jonka GWT kääntää sen optimoiduksi JavaScriptiksi. Kehittäjän näkökulmasta käyttöliittymää voi ajonaikaisesti paikantaa virheitä suoraan Java-koodista. Tällöin kehittäjän ei tarvitse opetella JavaScriptiä ollenkaan. [26]

Vaadin käyttää GWT:a näyttökomponenttien ja JavaScriptin luomiseen. Kaikki Vaadin-komponentit ovat mukautettuja GWT-komponetteja.

3.2.4 Tietorakennemalli

Vaadimen tietorakennemalli mahdollistaa informaation suoraa liittämistä näyttökomponentteihin ja tietojen muokkaamista näyttökomponentista käsin. Malli rakentuu seuraavien termien ympärille:

- ominaisuus (property) sisältää avaimen, jolla tieto on indeksoitu sekä ominaisuuden arvon

- tietue (item) sisältää useita eri avaimen omaavia ominaisuuksia, jotka yhdessä muodostavat tietueen, esimerkiksi tietokannan rivi
- säiliö (container) sisältää useita tietueita, muodostaen listauksen kaikesta käytössä olevasta informaatiosta, esimerkiksi yhden tietokantahaun tuloksista [27]

Versiosta 5.4 lähtien Vaadimessa on ollut mukana BeanItemContainer-niminen tietosäiliö. Tällaiseen säiliöön voi asettaa yhden tai kokoelman Java-papuja. Tällöin JavaBeans-nimeämiskäytännön mukaisesti nimetyt muuttujan nimet tulevat tietueen avaimeksi ja muuttujan arvot muuttuvat avainta vastaavaksi arvoksi. [27]

Vaikka Vaadimessa on kattava tuki JavaBeaneille, silti perus web-lomakkeen näyttökomponentille on rajoituksia ulkoasun suhteen ja Vaadimen oma ratkaisu ongelmaan on luoda HTML-kielellä sivu, johon elementit asetetaan. Asiakasprojektiin jouduttiin tekemään useita, hyvin monimutkaisia lomakkeita ja ongelma tuli ratkaista tehokkaammalla tavalla, joka mukailee säiliöitä käyttävää tapaa. Kun asiakasprojekti aloitettiin kesällä 2008, BeanItemContainer oli hyvin alkuvaiheessa eikä sisältynyt viralliseen julkaisuun. Näin ollen se ei soveltunut tuotantokäyttöön ja ongelma ratkaistiin integraatiossa.

3.3 Spring Framework

Spring Framework on tämän hetken johtavia avoimen lähdekoodin Java-ohjelmistokehyksiä. Pyrkimys on saada Java EE -ohjelmointia helpommaksi ja tehokkaammaksi. Kehys tarjoaa kehittäjäystävällisen, joustavan webohjelmistoalustan, joka on rakennettu Springin perustan päälle. Springillä on mittava käyttäjäyhteisö, jotka tarjoavat käyttäjätukea keskustelupalstoilla. Kaupallista tukea tarjoamaan perustettiin SpringSource-niminen yritys. [28]

Kehys on luotu rajapintojen ympärille, jolloin vaihdettavuus ja sidonta riippuvuuksissa vähenevät merkittävästi verrattuna luokkien käyttöön. Tällä tavoin saadaan kehyksen osat modulaariseksi. Toteutuksen perusstrategiana on ollut, ettei Spring pyri korvaamaan jo luotuja, hyvin toimivia sovelluksia, vaan pikemminkin integroimaan tehokkaasti ja helposti ratkaisut ohjelmistoon. [28]

3.3.1 IoC

IoC on lyhenne sanoista Inversion of Control. Kyseessä on suunnittelumalli sekä kokoelma tekniikoita, joilla pystyy yksinkertaistamaan liiketoimintalogiikkaa. IoC:in

asetuksissa määritellään toiminnalliset moduulit ja niille asetetaan riippuvuudet. Käytännössä Spring Frameworkin asetukset ovat XML-tiedostoissa, joissa esitellään toiminnallisuudet Spring-papuina ja liitetään niihin riippuvuudet. Springin versiossa 2.5 lähtien riippuvuudet on voinut määritellä myös annotaatioiden avulla. IoC on Spring Frameworkin keskeinen osa. [29] Spring-pavut sisältävät myös toiminnallisia luokkia, eivätkä näin ollen ole sama asia kuin Java-pavut.

3.3.2 Data binding

Data binding tarkoittaa tapaa tuoda tietoa lähteestä ja sijoittaa se tiettyyn rakenteeseen, tässä tapauksessa JavaBeans-rajapinnan mukaiseen Java-papuun. Springin omaa tiedonkytkijää käytetään lähinnä webrajapinnasta tulevaan tietoon. [30] Spring Frameworkissa tietokannasta tuleva tieto muunnetaan pavuksi erityisessä ORM-ohjelmistossa. Tällaisia ovat Apachen iBATIS, JBossin Hibernate, Sun Microsystemin JDO ja JPA sekä Oraclen TopLink. [31]

Spring Frameworkin liiketoimintalogiikka on suunniteltu toimimaan Java-papujen kanssa. Esimerkiksi www-lomake tuodaan tiedon kytkennässä Java-papuun, jonka jälkeen sitä voidaan käsitellä Springin validaattorissa. Validoinnin jälkeen papu on liiketoimintalogiikan käytössä. [30]

Springin data bindingia käytetään tässä opinnäytetyössä laajentamaan Vaadimen toimintaa www-lomakkeiden käsittelyä ja mahdollistamaan JavaBeanien käyttö Vaadimen tietorakennemallin mukaisesti.

3.3.3 Monikielisyyden tuki

Monikielistäminen tarkoittaa informaatioteknologiassa ohjelmiston suunnittelemista siten, että ohjelman käyttöliittymä pystytään näyttämään useilla kielillä. Tällöin taustatoteutus ei muutu vaan käyttöliittymässä olevat tekstit vaihtuvat. Tämä pitää sisällään myös Unicode-merkistön. [32]

Spring Frameworkissä on sisäänrakennettu tuki monikielisyydelle. Kehys osaa hakea käännetyt tekstit eri lähteistä MessageSources-rajapinnan kautta ohjelmassa käytettävän lokalisoinnin mukaisesti. [33]

Aiemmin mainittiin, että Vaadin ei tue monikielisyyttä. Nämä toiminnot tuodaan Spring Frameworkista ja integraatio hoitaa kielen muutokset mahdollisuuksien mukaan.

3.3.4 Annotaatiot

Spring Framework sisältää annotaatioita helpottamaan ja nopeuttamaan työskentelyä. Tässä luvussa esitellään ainostaan ne, joita todennäköisimmin tullaan käyttämään integraatiolla luotavissa ohjelmissa.

Autowired-annotaatiota käytetään tuomaan Spring-papuun riippuvuuksia. Tällä tavalla voidaan korvata suuri osa IoC:n XML-asetuksista. Yhden luokan asetukset voidaan keskittää itseensä, jolloin kehittäminen tulee suoraviivaisemmaksi ja ylläpito helpottuu. Autowiredia voi määrittää Qualifier-annotaatiolla, jotta useista samantyyppisistä Spring-pavuista saadaan valittua haluttu [Kuva 18]. [33]

```
/**
 * Demoluokka @Autowired - annotaation käyttöön.
 *
 * @author Tommi Hännikkälä
 */
public class AutowiredDemo {

    @Autowired
    @Qualifier("demoService")
    protected DemoService demoService;
}
```

Kuva 18 Autowired-annotaation luokan esimerkkikoodi

Spring IoC sisältää toiminnon, jolla voi hakea luokkia valitusta paketista ja luo niistä Spring-papuja. Luokat, joista halutaan luoda papuja, tulee merkitä jollain stereotyyppiannotaatiolla. Niitä Spring Frameworkin versio 2.5 esittelee neljä: Component, Controller, Service ja Repository. Controller-annotaatio kuuluu Spring MVC -pakettiin ja sillä annetaan MVC Controller -rooli luokalle. [33] Tässä opinnäytetyössä näyttökerros luodaan Vaadimella, joten Controller-annotaatiota ei tarvita.

Component on perusannotaatio, joka määrittelee, että luokka on Spring-papu. Annotaatiosta voi tehdä myös omia, mukautettuja annotaatioita. Service ja Repository ovat Componentin yläannotaatioita. [33]

Service-annotaatio määrittelee, että luokka kuuluu kolmikerrosarkkitehtuurin palvelukerrokseen. Kehyksen versiossa 2.5 ei Service-annotaatiolle ole määritetty mitään muuta erikoistoiminnallisuutta. [33]

Repository määrittelee kolmikerrosarkkitehtuurin tietovarastokerrokseen kuuluvan pavun. Annotaatiolla on erikoistoiminnallisuus tulkata tietovarastosta tulevat poikkeukset. [33]

Transactional-annotaatio liittyy Spring Frameworkin tietokannan transaktiohallintaan. Annotaatio voidaan määrittää luokka- ja metoditasolle ja sitä käytetään normaalisti palvelukerroksen luokissa. Annotaatio tuo hallinnan lähemmäs oikeata koodia ja nopeuttaa ohjelmiston kehittämistä XML-muotoisiin asetuksiin verrattuna. Transactional-annotaatio on toteutettu Spring Frameworkin aspektina. [34]

Esimerkissä 4 on yksinkertainen esimerkki palvelukerroksen luokasta, jonka metodeja kutsuttaessa aloitetaan tietokantatransaktio.

Esimerkki 4 Service- ja Transactional-annotaatioiden esimerkkiluokka

```
/**
 * Esimerkkiluokka @Service - annotaatiosta.
 *
 * @author Tommi Hännikkälä
 */
@Service("demoService")
@Transactional
public class DemoServiceImpl implements DemoService {

}
```

4 Integraatio

Integraation tarkoitus on tuoda useita alijärjestelmiä yhteen toimivaksi järjestelmäksi ja varmistaa, että alijärjestelmät toimivat yhdessä [35]. Vaadin – Spring Framework -integraation tarkoituksena on tuoda molempien ohjelmistopakettien vahvuudet yhteen. Vaadimen näyttötoiminnallisuuskomponentteihin upotetaan Springin toiminnot, jolla liiketoimintalogiikka siirretään uudelleenkäytettävyyden vuoksi palvelukerrokseen. Ainoastaan tarpeelliset palvelurajapinnat tuodaan näyttötoiminnallisuuskomponenttiin. Tällä keinolla saadaan uudelleenkäytettävää liiketoimintalogiikkaa esimerkiksi integraatiotarkoituksiin.

4.1 Arkkitehtuuri

Integraatiolle asetettiin seuraavat vaatimukset:

- Integraation tulee olla täysin riippumaton Springin ja Vaadimen sisäisistä ratkaisuista.
- Integraation tulee toimia alustariippumattomasti kaikissa alustoissa, joissa Vaadin ja Spring toimivat.
- Integraation tulee helpottaa kehittäjän työtä.
- Integraation tulee tarjota mahdollisuus kolmikerrosarkkitehtuuriin ja tätä kautta uudelleenkäytettävään liiketoimintalogiikkaan.
- Tuotettua ohjelmistoa tulee voida ajaa servletinä ja portletina.
- Tuotetun ohjelmiston tulee olla testattavissa. Vaadimelle löytyy oma testikehys, liiketoimintalogiikka voidaan testata yksikkötestauksella.

Integraatio on riippumaton niin Spring Frameworkin kuin Vaadimenkin sisäisistä toteutuksista silloin, kun ohjelmistojen väliset riippuvuudet eivät osoita toistensa tarjoamiin luokkiin, ainoastaan rajapintoihin. Jos tätä periaatetta noudatetaan, voidaan olla hyvin varmoja, etteivät pienemmät päivitykset kumpaankaan ohjelmistoon riko yhtenäisyyttä. Tähän tulokseen päästään, jos yleisenä periaatteena oleva käsitys, että alempien versionumeroiden muuttuminen eivät aiheuta ongelmia yhteensopivuudessa [36].

Alustariippumattomuudesta voidaan olla varmoja, kun käytetään ainoastaan Javaa ja kun ei käytetä käyttöjärjestelmän omia kirjastoja. Käytännössä tämä tarkoittaa, että projektissa näkyy dll-päätteisiä tiedostoja. Yleensä käytettävien kirjastojen dokumentaatio varoittaa ja jopa tarjoaa versiot eri käyttöympäristöille. Java tarjoaa käyttöjärjestelmän kirjastojen lataamiseen JNI-rajapintaa.

Kehittäjän työtä pyritään helpottamaan jakamalla ohjelmisto useampiin kerroksiin ja mahdollistamalla annotaatiopohjaiset asetukset aliluokille. Arkkitehtuurikerrokset voidaan testata automaattitestauksilla. Tämä helpottaa kehittäjää, kun vastuun testauksesta voi siirtää jatkuvaan integraatioon tarkoitetulle palvelimelle.

Kolmikerrosarkkitehtuuri on tuotu ohjelmistoon Spring Frameworkin ja sen tarjoamien Autowired-annotaatioiden avulla. Myös Spring Frameworkin vanhempi tapa, asettamismetodit ovat käytössä, mutta vanhentunut. Näillä kahdella tavalla tuodaan palvelukerroksen rajapinnat Vaadimen komponenttien käyttöön. Taustalla Spring

tarjoaa mahdollisuuden liittää IoC-toiminnallisuudellaan tietovarastoiden käsittelyluokat palvelukerrokseen. Lisäksi Spring tarjoaa JDBC-rajapinnan ja transaktioiden käyttöön tarkoitetut toiminnallisuudet.

Vaadin mahdollistaa ohjelmistojen julkaisun servlet- ja portaalitekniikalla samanaikaisesti. Integraatio on liian kaukana arkkitehtuurimallissa servlet- tai portaalitekniikan mahdollistavia toimintoja, jotta sillä olisi mahdollisuutta vaikuttaa toiminnallisuuteen. Spring Framework tarjoaa mahdollisuuden portaalin käyttöön, joten sekään ei aiheuta ongelmia. Tämän työn puitteissa Spring Frameworkin osat ovat arkkitehtuurimallissa niin kaukana servlet- ja portaalitoimintoja, etteivät ne pääse vaikuttamaan.

4.2 Toiminnallisuudet

Suurin kehittäjälle näkyvä muutos on uuden arkkitehtuuritason tuominen Vaadimeen. Tasoa kutsutaan toimintokerrokseksi (Action) ja kyseessä on Spring Frameworkissä määritelty papu, jonka Vaadimen ohjelmakerros (Application) hakee Spring IoC:sta. Tällöin IoC on vienyt toimintoon määritetyt riippuvuudet valmiiksi. Uusi arkkitehtuuritaso auttaa myös hajauttamaan näyttötoimintoja laajemmalle ja pyrkii tällä tavoin selkeyttämään näyttökomponenttien ohjauslogiikkaa. Ohjeena kehittäjälle on sijoittaa yksi näyttö yhden toiminnon sisään. Tällä tavalla eri kokonaisuudet saadaan pidettyä erillään ja ylläpito helpottuu.

Vaadimen mukautettu näyttökomponentti on yhden näytön osan, välilehden tai vaikka monikäyttöisen komponenttiryhmän, yhteen kokoava komponentti. Integraatio sisältää luokat, jotka mahdollistavat mukautetuille komponenteille Spring-toiminnallisuudet. Näin on mahdollista tuoda Spring-papuja suoraan mukautettuihin näytön osiin. Toimintokerros ja laajennetut mukautetut komponentit mahdollistavat lomakkeen käsittelyn delegoinnin keskitettyyn kohteeseen. Esimerkiksi lomake voi olla jaettu usealle eri välilehdelle, tällöin kaikkien välilehtien tiedot pitäisi lukea yhteen ja käsitellä. Integraation kanssa on mahdollista delegoida vaikkapa toimintokerros käsittelemään kaikkien lomakekentät.

Integraatio sisältää lomakeolion, joka lukee lähetettäessä määritetyt lomakkeen kentät Java-papuun. Lomakeolioon voidaan määrittää Spring Frameworkin Validator-rajapinnan toteuttavia luokkia, joka tarkastaa kentät esimerkiksi vääränlaisen syötteen varalta. Jos kentissä on vikaa, lomakeolio tuo virheet näytönhallintakomponenttiin, joka lomakkeen käsittelyä kutsui. Näyttökomponentissa voidaan tällöin käsitellä virheet

halutulla tavalla. Lomakeolio osaa myös lukea annetusta Java-pavusta arvot näyttökomponentteihin.

Lomakekentät määritetään *FormAttribute*-annotaatioilla. Tällöin annotaatiot luetaan toiminnon käynnistämisen yhteydessä, kun näyttökomponentit on alustettu. Komponentit kerätään *ToolkitFormObject*-luokkaan, joka sisältää kaikki lomakkeisiin liittyvät toiminnallisuudet, kuten papujen ja näyttökomponenttien välillä muuntaminen, tiedon oikeellisuuden tarkastus ja virheilmoitukset.

Vaadimen esimerkkiohjelmassa ehdotettiin tapaa, jossa näytön vaihtaminen tapahtuu nappulan painamisen jälkeen. Ratkaisu toimii siten, että ohjelmaolio pitää sisällään lukumetodin komponentille, johon näyttö halutaan piirtää. Komponentti toimii siis pääelementtinä kaikille näytöille ohjelmassa. Integraatiossa tämä toiminnallisuus on tuotu valmiina ohjelmaluokkaan.

Integraatio sisältää myös toimintohistorian. Kehittävässä sovelluksessa on mahdollisuus siirtyä yksi toiminto taaksepäin säilyttäen myös näyttökomponenttien tilan. Käytännössä esimerkiksi lomakekenttien arvot, pudotusvalikoiden sisältö ja muu tila sivulla pysyvät muuttumattomina.

4.3 Vertailu Vaadimen ehdotettuun integraatioon

Vaadimen ohjesivulta löytyy ohje integraatioon Spring Frameworkin kanssa. Ohje sisältää kaksi tapaa Spring-papujen tuomiseen Vaadin-ohjelmaan. Ensimmäinen on tapa, joka sopii vanhemmille Spring Frameworkin versioille. Kyseessä on erillinen avustajaluokka, joka luodaan jokaiseen näyttökomponenttiin erikseen. Jälkimmäinen tapa on Spring Frameworkin 2.5 -versiolle luotu tapa, jossa käytetään annotaatioita ja AspectJ-käsittelyä. [37]

Ensimmäisen tavan hyvä puoli on ratkaisun laaja toimivuus. Kokonaan Javalla luotu ratkaisu toimii jokaisessa standardin mukaisessa applikaatiopalvelimessa ilman lisätyötä. Huonot puolet ovat kertautuva lisätyö avustajaluokan tuomiseksi jokaiseen komponenttiin sekä Spring Frameworkin uusien, työtä nopeuttavien ominaisuuksien käyttämättä jättäminen.

Toisen tavan hyvät puolet ovat suoraviivaisuus kehittäjälle ja tätä myötä työn nopeutuminen. Huono puoli ratkaisussa on AspectJ-käsittelijän riippuvuus applikaatiopalvelimesta [38]. Jokaiselle eri applikaatiopalvelimelle täytyy asentaa oma kirjasto ja tämä aiheuttaa lisävaivaa kehittäjälle sekä ylläpitäjälle [38].

Molemmat tavat ovat ristiriidassa integraation vaatimusten kanssa ja näin ollen kumpaakaan ei otettu sellaisenaan käyttöön. Molemmissa integraatiomalleissa on pyritty mahdollisimman pienellä työllä saamaan Spring-pavut käyttöön Vaadin-ohjelmaan. Ajatusmaailma on lähtökohtaisesti erilainen, koska Vaadimen sivujen ehdottamien integraatioiden näkökulma on ainoastaan Vaadimen suunnasta, Spring Frameworkin vahvuudet ja mahdollisesti tehokkaammat ominaisuudet jätetään kokonaan huomioimatta.

Tämän opinnäytetyön puitteissa toteutetussa integraatiossa haluttiin nimenomaan ratkaisu, jolla nämä kaksi ratkaisua saadaan toimimaan yhdessä parhaalla mahdollisella tavalla; jatkettiin Vaadimen omia luokkia täyttämään vaatimukset ja luotiin välitaso, jotta Spring Framework pystyy käsittelemään yhden näytön komponentit kerralla. Tämä mahdollistaa kehittäjän näkökulmasta suoraviivaisempaa tapaa toimia. Annotaatiot käsitellään integraation sisällä, Spring Frameworkin avulla, eikä kehittäjän tarvitse huolehtia siitä.

4.4 Esimerkkiohjelma

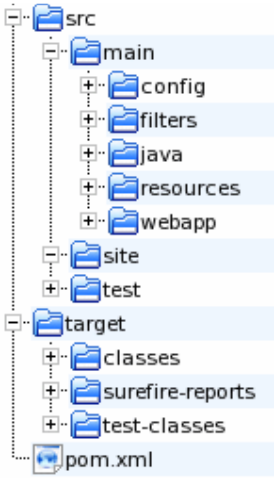
Tämän luvun on tarkoitus esitellä integraatiota apuna käyttäen luotu sovellus, joka sisältää kaikki integraation ominaisuudet ja yleiset ohjeet parhaiksi todetuista käyttötavoista.

4.4.1 Hakemistorakenne

Projektin pakettirakenne on valittu Apache Maven -määrittelyn mukaiseksi. Maven määrittelee hakemistorakenteen esimerkin 5 mukaiseksi. Hakemiston *src* alla ovat kaikki valmiiseen tuotteeseen tarvittavat lähdetiedostot ja *target*-hakemiston alle tulevat automaattisesti luodut tiedostot; käännetyt tiedostot, julkaisuvalmiit paketit sekä yksikkötestien raportit. Kehitettävän sovelluksen lähdetiedostot ovat hakemistossa *src/main*. Java-lähdekoodien juurihakemiston polku on *src/main/java*. Tähän hakemistoon luodaan tuotteen pakettirakenne ja logiikka. Toinen sovellussuunnittelun kannalta merkittävä hakemisto ovat *src/main/resources*, johon sijoitetaan sovelluksen tarvitsemat resurssit, jotka eivät ole Java-lähdekoodia. [39] Tällaisia voivat olla Spring Frameworkin XML-asetustiedostot ja ORM-kirjaston asetukset. Servlet-sovellus vaatii tämän lisäksi hakemiston *src/main/webapp* alle Servlet-määrittelyiden mukaiset asetustiedostot [39]. Hakemiston *src/test* alle tulevat yksikkötesteihin tarvittava lähdemateriaali; src/test/java sisältää Java-lähdekoodin ja src/test/resources muun testien tarvitseman materiaalin. Hakemistorakenteen juuressa oleva pom.xml-tiedosto

on Mavenin asetustiedosto ja sisältää mm. sovelluksen tarvitsemat kolmannen osapuolen kirjastot, tiedot versiohallinnasta sekä mahdolliset erikoistiedot projektin käsittelyyn. [39]

Esimerkki 5 Maven-työkalun hakemistorakenne



Standardoidulla hakemistorakenteella saadaan projektin rakenne yleiskäyttöiseksi ja pakettien rakentaminen helpottuu ja standardinmukaistuu. [39] Integraation vaatimuksiin ei kuulu Mavenin käyttö, mutta standardoitu rakenne auttaa uusia kehittäjiä ymmärtämään projektia helpommin.

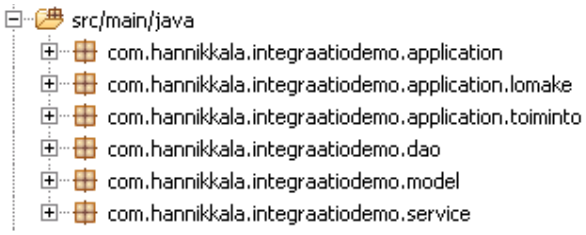
4.4.2 Pakettirakenne

Pakettirakenne on suunniteltu erottamaan näyttökomponentit ja liiketoimintalogiikka toisistaan. Tämän lisäksi myös luonteeltaan erilaiset luokat, esimerkiksi eri arkkitehtuurikerroksen luokat sijoitetaan eri paketteihin paremman löydettävyyden vuoksi.

Kuvassa 19 näkyy Maven-tyyppisen hakemistorakenteen Java-lähdekoodihakemisto *src/main/java*. Sen alla puumuodossa on paketti *com.hannikkala.integraatiodemo.application*. Tähän pakettiin tulee projektin ohjelmatiedosto sekä kaikki toimintoluokat. Jos ohjelma on todella suurikokoinen tai sisältää useita Vaadin-ohjelmaluokkia, on mahdollista myös luoda *application*-paketin alle *action*-paketti toimintokerroksen luokkia varten. Pienikokoisessa, yhden ohjelmaluokan sovelluksessa asia ei ole ongelma. Kun kaikki toiminnot ovat samassa paketissa, voi Spring Frameworkin pavuetsijälle antaa komennon etsiä papuja ainoastaan tuosta paketista. *Application*-paketin alla on *lomake*- sekä *toiminto*-nimiset paketit. Ne kuvastavat kahta eri toimintoluokkaa, jotka löytyvät *application*-paketin alta.

Hakemistoihin on tarkoitus luoda kaikki toimintokohtaiset näytöt. Tässä esimerkissä ei ole yleiskäyttöisiä näyttökomponentteja, mutta sellaiset voisi tarvittaessa sijoittaa esimerkkitapauksessa pakettiin *com.hannikkala.integraatiodemo.application.global*. Paketti *com.hannikkala.integraatiodemo.dao* sisältää kolmikerrosarkkitehtuurin tietovarastokerroksen luokat, *com.hannikkala.integraatiodemo.model* sisältää JavaBean-luokat sekä *com.hannikkala.integraatiodemo.service* palvelukerroksen luokat.

Projekti tarvitsee myös Spring Frameworkin asetustiedostot, mutta niitä ei sijoiteta kyseisen päähakemiston alle. Myös ohjelmistokohtaiset asetukset tulevat eri päähakemiston alle.



Kuva 19 Esimerkkiprojektin lähdekoodin pakettirakenne

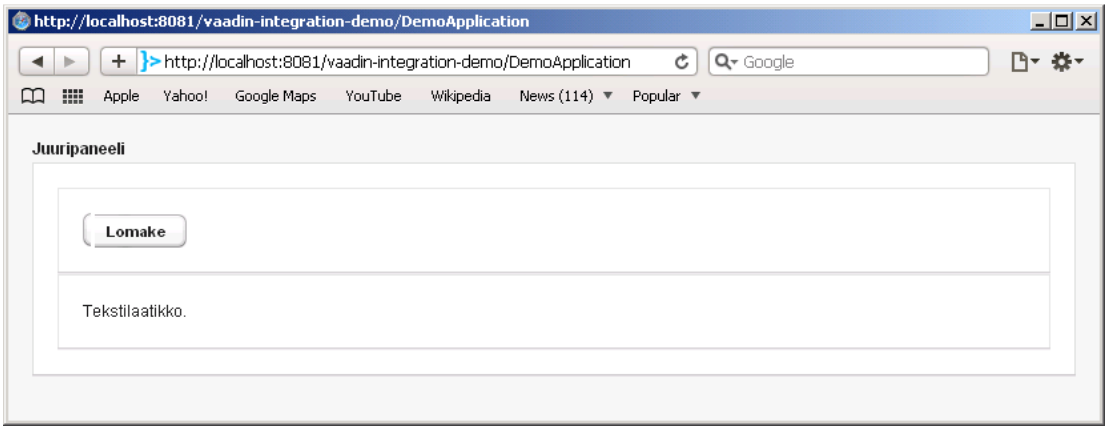
4.4.3 Pääohjelma

Pääohjelman lähdekoodi löytyy liitteestä 1. Sen toiminnallisuus ei vaikuta mitenkään näyttötoimintoihin. Se alustaa ohjelman ja delegoi toiminnon luomana ensimmäisen näytön.

4.4.4 Normaali näyttö

Tämän luvun tarkoituksena on näyttää kuvin esimerkki kehitettävän sovelluksen käyttäjälle näkyvästä näytöstä. Tässä näyttötyypissä ei ole ollenkaan lomakkeen käsittelyominaisuuksia. Tämä ei tarkoita kuitenkaan, ettei se voisi sisältää dynaamisia toiminnallisuuksia. Luonteeltaan sivu voisi olla sisältö-, navigointi- tai vaikka tulossivu.

Esimerkkinäytössä [Kuva 20] on ainoastaan staattista tekstikenttää esittelevä elementti sekä linkkinä lomakenäytölle toimiva painike, jonka painamiseen on asetettu kuuntelija sivun vaihto -toiminnon suorittamiseksi. Toiminnon lähdekoodi löytyy liitteestä 2, ylemmän paneelin koodi liitteestä 3 ja alemman paneelin liitteestä 4.

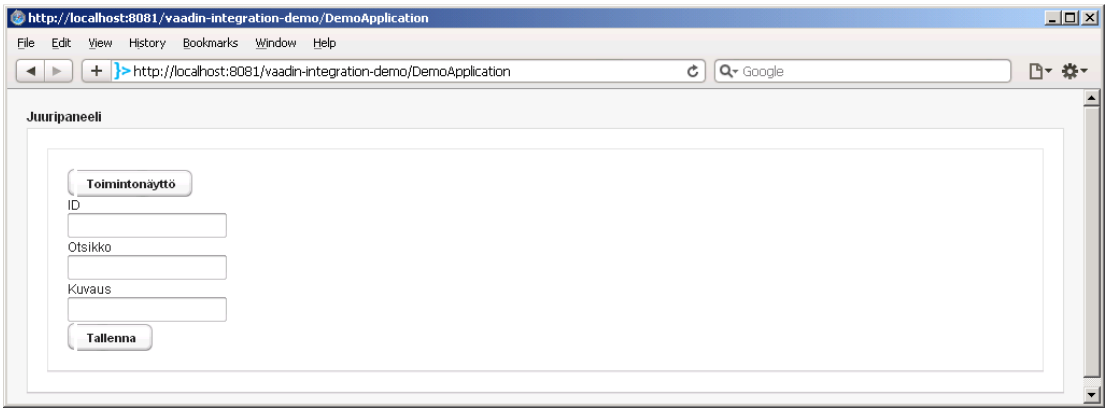


Kuva 20 Ruutukaappaus perusnäytöstä

4.4.5 Lomakenäyttö

Tässä luvussa näytetään esimerkki sovelluksen näytöstä, joka sisältää lomakkeen. Lomakekentät voidaan määritellä integraation sisältämillä annotaatioilla. Tällöin taustalogiikka käsittelee annotaatiot ja luo lomakekäsittelijän asetukset niiden perusteella.

Esimerkkinä [Kuva 21] on luotu Java-papu nimeltään Papu. Se sisältää ominaisuudet *id*, *title* ja *description*. Ne ovat yhdistetty järjestyksessä kenttiin ID, Otsikko ja Kuvaus, jolloin tekstikenttiin syötetyt arvot kirjoitetaan papuun. Jos arvo on väärän muotoinen esimerkiksi ID-kenttään, joka on kokonaislukutyyppinen, on syötetty merkkijono, syntyy virhetilanne, jolloin luodaan virheilmoituksen sisältävä elementti ja sijoitetaan se käyttöliittymään. Tallennus kutsuu palvelukerrosta lähetetyllä pavulla ja näyttää syötetyt tiedot virheilmoituselementissä. Lomakkeen toimintokerroksen koodi löytyy liitteestä 5, näyttöpaneelin liitteestä 6. Näyttöpaneeliin on liitetty esimerkkiohjelman ainoa palvelurajapinta, DemoService. Sen lähdekoodi löytyy liitteestä 7 ja rajapinnan toteuttava luokka, DemoServiceImpl, liitteestä 8.



Kuva 21 Ruutukaappaus lomakenäytöstä

4.4.6 Lomakkeen käsittely

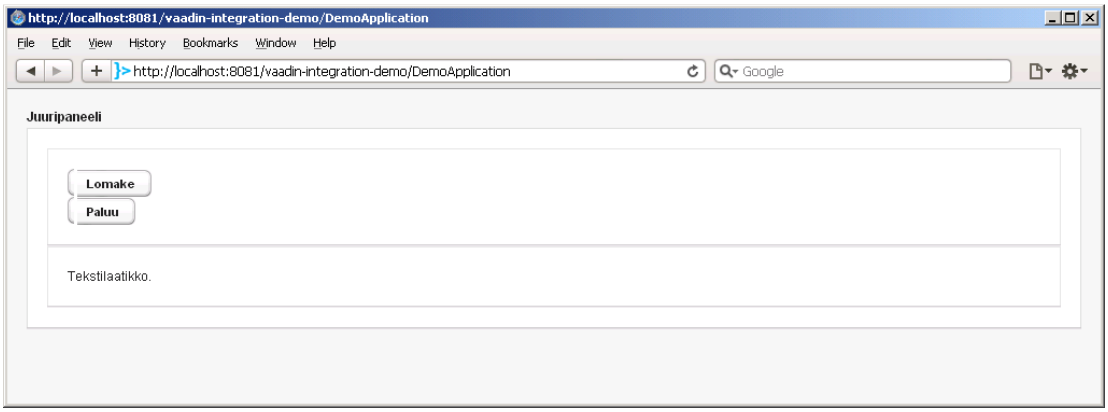
Tämä luku näyttää, miten integraatiokirjasto käsittelee lähetetyn lomakkeen tiedot.

Integraation virheenkäsittely suunniteltiin ensin tarjoamaan toiminnallisuudet ainoastaan yhdessä mukautetussa komponentissa tai toiminnossa sijaitsevat lomakekentät. Tällöin virheenkäsittely toteutettiin komponentin luokassa olevalla onError-metodilla, johon tullaan jos Spring Frameworkin validaattoreista yksi tai useampi ei mennyt läpi. Mukautettu virheenkäsittely luotiin ylikirjoittamalla metodi. Logiikaltaan tapa oli hyvin suoraviivainen ja käytti olio-ohjelmoinnin perusmekanismia. Useampaan komponenttiin hajautetut lomakkeet eivät kuuluneet vaatimuksiin tällöin. Lomakkeen hajauttamisen myötä entinen tapa ei enää toiminut ja integraatioon lisättiin virhekuuntelija. Tällä tavalla saatiin mukautetumpi tapa käsitellä virheilmoitukset. Lisäominaisuuden myötä myös kieliversioidut virheilmoitukset saatiin toteutettua ja tiettyä kenttää koskevat virheet saadaan käsiteltyä juuri sovelluksessa halutulla tavalla.

Aluksi Vaadin servlet alustaa kaikki näyttökomponentit, integraation mukana myös lomakkeen kentät ja vie viittaukset talteen lomakeolioon, jonka jälkeen näyttökomponentissa luodaan nimetön instanssi virhekuuntelijasta. Koska instanssi luodaan komponentin metodin sisälle, se pystyy käsittelemään komponentissa sekä metodissa näkyviä paikallisia ja luokkamuuttujia. Virhekäsittelijärajapinta sisältää kaksi metodia, toinen virheiden lisäämiseen käyttöliittymään ja toinen kaikkien poistamiseen. Kun validaattori löytää virheen, se kutsuu virhekuuntelijan lisäysmetodia. Kehittäjä saa tällöin luoda mukautetun tavan näyttää virheet käyttöliittymässä.

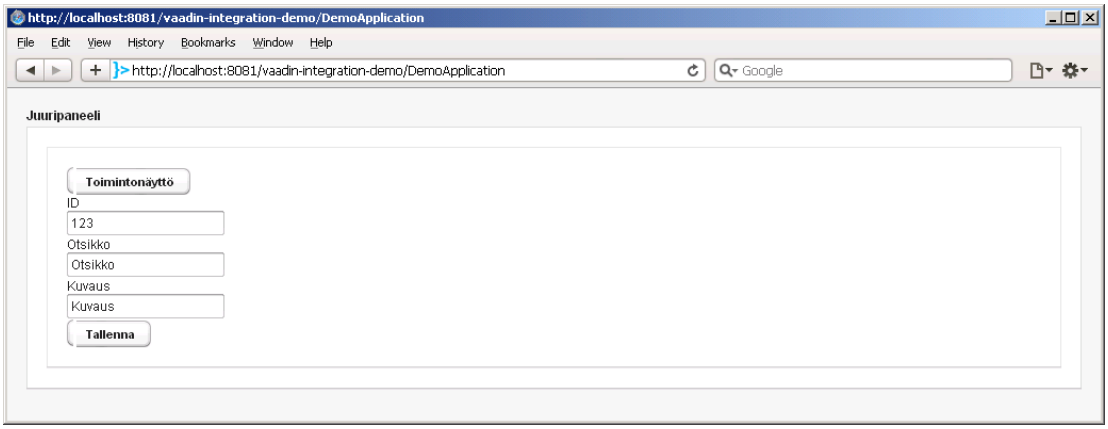
4.4.7 Toimintohistoria

Lomakenäytössä [Kuva 21] näkyy painike Toimintonäyttö. Sitä painamalla pääsee takaisin ensimmäiselle näytölle. Koska toiminnosta on luotu uusi instanssi, logiikka huomaa, että sovellukseen on kertynyt historiatietoa ja Paluu-painike tulee näkyviin [Kuva 22]. Tässä vaiheessa historiassa on kaksi Toiminto-oliota ja yksi Lomake-olio. Painettaessa paluupainiketta, sovellus palaa takaisin lomakenäytölle.



Kuva 22 Perusnäyttö paluu-painikkeella

Toimintohistoria sisältää näyttökomponenttien tilan, joten esimerkiksi pudotusvalikoiden sisällön voi säilyttää. Arvojen säilyminen riippuu siitä, onko toiminto määritetty käyttäjäsessioon tallennettavaksi vai halutaanko toimintoa kutsuttaessa luoda uusi ilmentymä. Esimerkkitapauksessa syötetyt arvot halutaan säilyttää ja paluupainikkeesta painamisen jälkeen esiin tuleva näyttö sisältää edelliset arvot tekstikentissä [Kuva 23].



Kuva 23 Lomakenäyttö, jossa edelliset syötetyt tiedot

4.4.8 Spring Frameworkin asetukset

Liitteessä 9 on kuvattuna Spring Frameworkin asetustiedoston sisältö. Asetukset käytännössä määrittävät Springin etsimään komponentteja paketeista *com.hannikkala.integraatiodemo.application* sekä *com.hannikkala.integraatiodemo.service*. Ensiksi mainittuun luotiin edellä ohjelman ja kaikkien toimintoluokkien toteutukset. Service-paketti sisältää palvelukerroksen toteutuksen.

4.5 Tulevaisuus

Integraation kehitys eteni tämän opinnäytetyön aikana paljon, mutta molemmat kolmannen osapuolen kirjastot kehittyvät nopeasti, joten kehitystyötä riittää jatkossakin. Jo tällä hetkellä integraatio on saanut jo uusia toimintoja ja uutta suuntaa ensimmäisistä versioista. Sen jälkeen on tullut Spring Frameworkin versio 2.5, joka toi kehykseen pavun luontiin tarkoitetut annotaatiot. Ne nopeuttivat ja helpottivat kehitystyötä, koska enää ei tarvinnut kirjoittaa XML-asetuksia jokaisesta pavusta erikseen vaan annotaatiot hoitivat suuren osan työstä. Myös ohjelmistokoodin keskittäytyminen toi nopeuttavia piirteitä kehittäjän työhön; annotaatio kirjoitetaan yleisesti joko luokan tai annotaation omaavan luokan metodin yläpuolelle. Kun ohjelmiston osalle erityismerkityksen tuova osa on sijoitettu lähelle kehittäjän kiintopistettä, vähentyy samalla kehitystyötä hidastava ylimääräinen etsiminen vaikkapa XML-asetuksista. Vaadin oli integraation alkuvaiheessa versiossa 5.2 ja vielä nimellä ITMill Toolkit. Muutoksessa tuli uusi pakettirakenne alusta lähtien ja käyttäjäkohtaisessa sessiossa olevat käyttöliittymäkomponentit. Tämä muutos pakotti luomaan integraatiosta uuden haaran versionhallintaan. Nyt ainoastaan Vaadimelle eli versiolle 6.0 tarkoitettu haara on aktiivisessa kehityksessä.

Nyt huomattuja, enemmänkin kosmeettisia vikoja ovat esimerkiksi toimintokerroksen luokkiin pakollisesti sijoitettava pavun Scope-annotaatio, joka tulee asettaa jokaiseen erikseen. Scope kertoo Spring IoC:lle, miten pavusta tulee tehdä uusia instansseja. Vaihtoehtoina ovat kaikille käyttäjille sama instanssi, mikä on myös vakiona, käyttäjäkohtainen sessiopapu tai joka pavun haulla uusi instanssi. Integraation perusajatuksena on, että Scope on vaihtoehtoisesti joka pyynnölle uusi instanssi tai erikoistapauksissa käyttäjän sessiokohtainen instanssi. Vika ei ole käyttöä estävä, muttei myöskään optimaalinen tilanne. Tulevaisuuden versioissa asia tulisi korjata jollakin johdonmukaisella tavalla.

Toinen havaittu ongelma tuli Vaadimen käyttäjäsessiossa olevien mukana. Kun sovellukseen tekee muutoksen, sovelluspalvelin käynnistää itsensä ja sessiossa olevat oliot serialisoidaan. Tämä tarkoittaa käytännössä olion muuntamista merkkijonoksi. Sovelluspalvelinta käynnistettäessä merkkijonot muunnetaan takaisin oliomuotoon. Spring Frameworkin luokat ja lokikirjasto eivät ole serialisoituvia, tämän vuoksi kirjasto heittää poikkeuksen aina palvelinta käynnistettäessä. Vika on esteettinen, eikä näin ollen ole kirjaston käyttöä estävä. Ratkaisuna voisi olla joko olioksi muuntamisen yhteydessä luoda uudet ilmentymät tai vaihtoehtoisesti luoda serialisoituvat luokat integraatiokirjastoon.

5 Yhteenveto

Tuloksena luotu kirjasto on huomattavasti laajempi kuin alun perin suunniteltiin. Samalla kirjasto sisältää yleiskäyttöisempiä piirteitä, jotka helpottavat uusien ominaisuuksien luontia ja ohjelmistokoodin ylläpitoa. Kyseessä on kuitenkin sivutuotteena luotava kirjasto, jonka on tarkoitus vastata asiakasprojektien tarpeisiin. Tästä huolimatta kirjasto sisältää useita helpottavia ominaisuuksia eikä aseta ylimääräisiä rajoituksia kehittäjälle.

Integraation kehitystyö on kestänyt yli vuoden ja on kokenut suuria uudistuksia sinä aikana, mm. nimen muuttuminen ITMill Toolkitista Vaadimeksi ja annotaatioiden mukaanotto. Kehitys jatkuu, kun lisää ajatuksia kirjaston parantamiseksi syntyy. Myös molempien integraation kohteiden kehittyminen aiheuttaa muutoksia ja molemmat ovat aktiivisessa kehityksessä. Lisäksi Javan uudet ominaisuudet tuovat uusia ajatusmalleja ja ratkaisutapoja.

Integraatio on ollut haastavin projekti, johon olen tähän mennessä ottanut osaa. Tämän vuoksi se on ollut myös mielenkiintoisin. Luovuutta ja opittuja taitoja on saanut käyttää ja uutta on täytynyt omaksua nopeasti. Projekti on antanut mahdollisuuden tutustua normaalissa työssä käytettävien ominaisuuksien sisäiseen toteutukseen ja sitä kautta kehittänyt uusia ajatuksia sekä ammatillisia taitoja.

LÄHTEET

1. Affecto: *Yhtiö*, *Affecto.fi*. Saatavilla: http://affecto.fi/page.php?page_id=46. Viitattu 31.8.2009.

2. David G, Shaw M: *An Introducion to Software Architecture*. 1.1994. http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf. Viitattu 12.8.2009.

3. Raposa RF: *Declarations and access control*. *SCJP: Sun Certified Programmer for Java Platform, Standard Edition 6 Study*; The McGraw-Hill Companies; 2008.

4. Raposa RF: *Clarity and Maintainability*. *SCJP: Sun Certified Programmer for Java Platform, Standard Edition 6 Study*; The McGraw-Hill Companies; 2008.

5. Raposa RF: *Object orientation*. *SCJP: Sun Certified Programmer for Java Platform, Standard Edition 6 Study*; The McGraw-Hill Companies; 2008.

6. Bridgewater D: *J2EE Patterns*. *Sun Certified Web Component Developer Stude Guide*; McGraw-Hill/Osborne; 2005.

7. Sun Microsystems: *Front Controller*, *Core J2EE Patterns*. 2002. Saatavilla: http://java.sun.com/blueprints/corej2eepatterns/Patterns/FrontController.html. Viitattu 2.9.2009.

8. Sun Microsystems: *Business Delegate*, *Core J2EE Patterns*. 2002. Saatavilla: http://java.sun.com/blueprints/corej2eepatterns/Patterns/BusinessDelegate.html. Viitattu 2.9.2009.

9. Sun Microsystems: *View Helper*, *Core J2EE*. 2002. Saatavilla: http://java.sun.com/blueprints/corej2eepatterns/Patterns/ViewHelper.html. Viitattu 2.9.2009.

10. Sun Microsystems: *Dispatcher View*, *Core J2EE Patterns*. 2002. Saatavilla: http://java.sun.com/blueprints/corej2eepatterns/Patterns/DispatcherView.html. Viitattu 2.9.2009.

11 Nickul D, Reitman L, Ward J, Wilber J: *Service Oriented Architecture (SOA) and*

. *Specialized Messaging Patterns*. 12.2007. http://www.adobe.com/enterprise/pdfs/Services_Oriented_Architecture_from_Adobe.pdf. Viitattu 24.9.2009.

12 Ruggiero R, Brooks R: *The Java 2 Platform Enterprise Edition (J2EE) Three-Tier*
. *Model*, *Information Management*. 2.2005. Saatavilla: http://www.information-management.com/infodirect/20050225/1020766-1.html. Viitattu 21.6.2009.

13 EL-Manzalawy Y: *Aspect Oriented Programming*, *Developer.com*. 5.2.2004.
. Saatavilla: http://www.developer.com/design/article.php/3308941/Aspect-Oriented-Programming.htm. Viitattu 16.9.2009.

14 Sorsa M. *Aspektiympäristöt AspectJ ja Spring AOP*. Joensuu 2008.
.

15 Sun Microsystems: *What is the Java Platform?*. 30.4.1996. Saatavilla:
. http://java.sun.com/docs/white/platform/javaplatform.doc1.html. Viitattu 24.7.2009.

16 Hautus E: *Improving Java software through package structure analysis*. 2002.
. http://www.xs4all.nl/~ehautus/papers/PASTA.pdf. Viitattu 15.9.2009.

17 Raposa RF: *Flow Control, Exceptions, and Assertions*. *SCJP: Sun Certified*
. *Programmer for Java Platform, Standard Edition 6 Study*; The McGraw-Hill Companies; 2008.

18 Sun Microsystems: *Java Servlet API White Paper*, *Sun Developer Network*.
. Saatavilla: http://java.sun.com/products/servlet/whitepaper.html. Viitattu 31.8.2009.

19 Hamilton G: *JavaBeans Spec*. 8.8.1997. Saatavilla:
. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html. Viitattu 15.8.2009.

20 Sun Microsystems: *Annotations*, *JDK 5.0 Developer's Guide*. 18.12.2007.
. Saatavilla: http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html. Viitattu 15.8.2009.

21 Xerox Corporation, Palo Alto Research Center Inc.: *Introduction to AspectJ*, *Getting started with AspectJ*. Saatavilla:

. http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html. Viitattu 16.9.2009.

22 ITMill Oy: *FAQ*, *vaadin.com*. Saatavilla: http://vaadin.com/faq. Viitattu 17.9.2009.
.

23 ITMill Oy: *Book - Architecture*, *vaadin.com*. Saatavilla: http://vaadin.com/book/-
. /page/architecture.html. Viitattu 20.6.2009.

24 Ekonoja A, Lahtonen T, Mäntylä J: *Ajax (Asynchronous JavaScript and XML) -*
. *Luento 15*, *Informaatioteknologia - Jyväskylän yliopiston IT-tiedekunta ja avoin yliopisto*. 19.4.2008. Saatavilla: http://appro.mit.jyu.fi/sovellukset/luennot/luento15/. Viitattu 14.6.2009.

25 *Introducing JSON*. Saatavilla: http://json.org/. Viitattu 20.6.2009.
.

26 Google: *Product Overview*, *Google Web Toolkit*. Saatavilla:
. http://code.google.com/webtoolkit/overview.html. Viitattu 16.8.2009.

27 ITMill Oy: *Binding components to data*, *Book - vaadin.com*. Saatavilla:
. http://vaadin.com/book/-/page/datamodel.html. Viitattu 16.8.2009.

28 SpringSource: *About Spring*, *SpringSource.org*. Saatavilla:
. http://www.springsource.org/about. Viitattu 20.6.2009.

29 A.P.Rajshekhar: *The Spring Framework: Understanding IoC*. Saatavilla:
. http://www.devshed.com/c/a/Java/The-Spring-Framework-Understanding-IoC/.
Viitattu 20.6.2009.

30 SpringSource: *Validation, Data-binding, the BeanWrapper, and PropertyEditors*,
. *The Spring Framework - Reference Documentation*. Saatavilla:
http://static.springsource.org/spring/docs/2.5.x/reference/validation.html. Viitattu 15.8.2009.

31 SpringSource: *Object Relational Mapping (ORM) data access*, *The Spring*
. *Framework Reference*. Saatavilla:
http://static.springsource.org/spring/docs/2.5.x/reference/orm.html. Viitattu

16.8.2009.

32. Ishida R, Miller SK: *Localization vs. Internationalization*, *W3C I18n FAQ*. 19.10.2006. Saatavilla: http://www.w3.org/International/questions/qa-i18n. Viitattu 16.6.2009.

33. SpringSource: *The IoC container*, *The Spring Framework - Reference Manual*. Saatavilla: http://static.springsource.org/spring/docs/2.5.x/reference/beans.html. Viitattu 16.8.2009.

34. SpringSource: *Transaction management*, *The Spring Framework Reference Manual*. Saatavilla: http://static.springsource.org/spring/docs/2.5.x/reference/transaction.html. Viitattu 1.9.2009.

35. Computer Information Systems Department: *CIS8020 Syllabus*. Saatavilla: http://www2.cis.gsu.edu/cis/program/syllabus/graduate/CIS8020.asp. Viitattu 16.8.2009.

36. Atwood J: *What's In a Version Number, Anyway?*, *Coding Horror*. 15.2.2007. Saatavilla: http://www.codinghorror.com/blog/archives/000793.html. Viitattu 2.9.2009.

37. ITMill Oy: *Spring Integration*, *Wiki - vaadin.com*. Saatavilla: http://vaadin.com/wiki/-/wiki/Main/Spring%20Integration?p_r_p_185834411_title=Spring%20Integration. Viitattu 8.9.2009.

38. SpringSource: *Aspect Oriented Programming with Spring*, *The Spring Framework Reference*. Saatavilla: http://static.springsource.org/spring/docs/2.5.x/reference/aop.html. Viitattu 15.9.2009.

39. Smart JF: *An introduction to Maven 2*, *JavaWorld*. 5.12.2005. Saatavilla: http://www.javaworld.com/javaworld/jw-12-2005/jw-1205-maven.html. Viitattu 11.9.2009.

LIITTEET

LIITE 1 Vaadin-ohjelman lähdekoodi – DemoApplication.java

LIITE 2 Staattisen näytön toimintokerroksen lähdekoodi – Toiminto.java

LIITE 3 Staattisen näytön ylemmän näyttöpaneelin lähdekoodi – YlaPaneeli.java

LIITE 4 Staattisen näytön alemman näyttöpaneelin lähdekoodi – AlaPaneeli.java

LIITE 5 Lomakenäytön toimintokerroksen lähdekoodi – Lomake.java

LIITE 6 Lomakenäytön näyttöpaneelin lähdekoodi – LomakePaneeli.java

LIITE 7 Lomakenäyttöön liitetyn palvelurajapinnan lähdekoodi – DemoService.java

LIITE 8 Lomakenäyttöön liitetyn palvelurajapinnan toteuttavan luokan lähdekoodi – DemoServiceImpl.java

LIITE 9 Spring Frameworkin asetustiedosto – applicationContext.xml

LIITE 1 Vaadin-ohjelman lähdekoodi

```
package com.hannikkala.integraatiodemo.application;


import com.affecto.integration.vaadin.SpringAwareApplication;
import com.vaadin.ui.Panel;
import com.vaadin.ui.Window;
import com.vaadin.ui.Window.Notification;


/**
 * Spring Framework - Vaadin - integraation esimerkkiohjelman ohjelmaluokka.
 *
 * @author Tommi Hännikkälä <tommi@hannikkala.com>
 */
public class DemoApplication extends SpringAwareApplication {

	private static final long serialVersionUID = -181662916602217853L;


	/* (non-Javadoc)
	 * @see com.affecto.integration.vaadin.SpringAwareApplication#handleException(java.lang.Throwable)
	 */
```

```
        @Override
        protected void handleException(Throwable e) {
                getMainWindow().showNotification("Virhe: " +
e.getMessage(), Notification.TYPE_ERROR_MESSAGE);
        }

        /* (non-Javadoc)
         * @see com.vaadin.Application#init()
         */
        @Override
        public void init() {
                Window window = new Window();
                setMainWindow(window);
                Panel main = new Panel("Juuripaneeli");
                setBodyContainer(main);
                w.addComponent(main);
                launchAction("toiminto");
        }

}
```

LIITE 2 Staattisen näytön toimintokerroksen lähdekoodi

```
package com.hannikkala.integraatiodemo.application;


import org.springframework.beans.factory.config.BeanDefinition;

import org.springframework.context.annotation.Scope;

import com.affecto.integration.vaadin.ToolkitAction;

import com.affecto.integration.vaadin.annotation.Action;

import com.hannikkala.integraatiodemo.application.toiminto.AlaPaneli;

import com.hannikkala.integraatiodemo.application.toiminto.YlaPaneli;

import com.vaadin.ui.VerticalLayout;


/**

 * Spring Framework - Vaadin - integraation esimerkkiohjelman staattisen näytön toimintoluokka.

 *

 * @author Tommi Hännikkälä <tommi@hannikkala.com>

 */

@Action

@Scope(BeanDefinition.SCOPE_PROTOTYPE)

public class Toiminto extends ToolkitAction {


		private VerticalLayout pohja;
```

```
	/* (non-Javadoc)
	 * @see com.affecto.integration.vaadin.ToolkitAction#init()
	 */
	@Override
	public void init() throws Exception {
			pohja = new VerticalLayout();
			setCompositionRoot(pohja);

			pohja.addComponent(new YlaPaneli(getApplication()));
			pohja.addComponent(new AlaPaneli(getApplication()));
	}

}
```

LIITE 3 Staattisen näytön ylemmän näyttöpaneelin lähdekoodi

```
package com.hannikkala.integraatiodemo.application.toiminto;


import com.affecto.integration.vaadin.SpringAwareApplication;

import com.affecto.integration.vaadin.SpringAwareCustomComponent;

import com.vaadin.ui.Button;

import com.vaadin.ui.Panel;

import com.vaadin.ui.Button.ClickEvent;

import com.vaadin.ui.Button.ClickListener;


/**

 * Spring Framework - Vaadin - integraation esimerkkiohjelman staattisen näytön ylempi paneli.

 *

 * @author Tommi Hännikkälä <tommi@hannikkala.com>

 */

public class YlaPaneli extends SpringAwareCustomComponent {


		private static final long serialVersionUID = -1803898850628583211L;


		public YlaPaneli(SpringAwareApplication application) {

			super(application);
```

```
                initComponents();
        }

        /* (non-Javadoc)
         *                                                                    @see
com.affecto.integration.vaadin.SpringAwareCustomComponent#initComponents()
         */
        @Override
        public void initComponents() {
                Button b = new Button("Lomake");

                b.addListener(new ClickListener() {

                        private static final long serialVersionUID = -
5735385097466002626L;

                        @Override
                        public void buttonClick(ClickEvent event) {

getSpringAwareApplication().launchAction("lomake");
                        }
                });

                Panel p = new Panel();
```

```
                p.addComponent(b);

                if(getSpringAwareApplication().isHistoryAvailable()) {
                        Button backButton = new Button("Paluu");

                        backButton.addListener(new ClickListener() {

                                private static final long
serialVersionUID = 43725794178879993 46L;

                                @Override
                                public void
buttonClick(ClickEvent event) {
        getSpringAwareApplication().goBackHistory();
                                }
                        });

                        p.addComponent(backButton);
                }

                setCompositionRoot(p);
        }
```

```
}
```

LIITE 4 Staattisen näytön alemman näyttöpaneelin lähdekoodi

```
package com.hannikkala.integraatiodemo.application.toiminto;


import com.affecto.integration.vaadin.SpringAwareApplication;

import com.affecto.integration.vaadin.SpringAwareCustomComponent;

import com.vaadin.ui.Label;

import com.vaadin.ui.Panel;


/**

 * Spring Framework - Vaadin - integraation esimerkkiohjelman staattisen näytön alempi paneli.

 *

 * @author Tommi Hännikkälä <tommi@hannikkala.com>

 */

public class AlaPaneli extends SpringAwareCustomComponent {


	private static final long serialVersionUID = -2833767422712076056L;


	public AlaPaneli(SpringAwareApplication application) {

		super(application);

		initComponents();

	}
```

```
		/* (non-Javadoc)
		 *                                                                    @see
com.affecto.integration.vaadin.SpringAwareCustomComponent#initComponents()
		 */
		@Override
		public void initComponents() {
			Label label = new Label("Tekstilaatikko.");
			Panel p = new Panel();
			p.addComponent(label);
			setCompositionRoot(p);
		}

}
```

LIITE 5 Lomakenäytön toimintokerroksen lähdekoodi

```
package com.hannikkala.integraatiodemo.application;


import org.springframework.context.annotation.Scope;

import com.affecto.integration.vaadin.ToolkitFormAction;

import com.affecto.integration.vaadin.annotation.Action;

import com.hannikkala.integraatiodemo.application.lomake.LomakePaneeli;

import com.hannikkala.integraatiodemo.model.Papu;


/**

 * Spring Framework - Vaadin - integraation esimerkkiohjelman lomakenäytön toimintoluokka.

 *

 * @author Tommi Hännikkälä <tommi@hannikkala.com>

 */

@Action

@Scope("session")

public class Lomake extends ToolkitFormAction<Papu> {


		private static final long serialVersionUID = 7512978876724952740L;


		private LomakePaneeli lomakeP;
```

```
		/* (non-Javadoc)
		 * @see com.affecto.integration.vaadin.ToolkitFormAction#init()
		 */
		@Override
		public void init() throws Exception {
			lomakeP = new LomakePaneeli(getApplication());
			setCompositionRoot(lomakeP);
		}
}
```

LIITE 6 Lomakenäytön näyttöpaneelin lähdekoodi

```
package com.hannikkala.integraatiodemo.application.lomake;



import org.springframework.beans.factory.annotation.Autowired;

import com.affecto.integration.vaadin.SpringAwareApplication;

import com.affecto.integration.vaadin.SpringAwareCustomFormComponent;

import com.affecto.integration.vaadin.annotation.FormAttribute;

import com.affecto.integration.vaadin.form.FormObject;

import com.affecto.integration.vaadin.form.ToolkitFormObject.FormErrorEvent;

import com.affecto.integration.vaadin.form.ToolkitFormObject.FormErrorListener;

import com.hannikkala.integraatiodemo.model.Papu;

import com.hannikkala.integraatiodemo.service.DemoService;

import com.vaadin.terminal.UserError;

import com.vaadin.ui.Button;

import com.vaadin.ui.Label;

import com.vaadin.ui.Panel;

import com.vaadin.ui.TextField;

import com.vaadin.ui.Button.ClickEvent;

import com.vaadin.ui.Button.ClickListener;


/**

 * Spring Framework - Vaadin - integraation esimerkkiohjelman lomakenäytön paneli.
```

```
 *
 * @author Tommi Hännikkälä <tommi@hannikkala.com>
 */
public class LomakePaneeli extends SpringAwareCustomFormComponent<Papu> {

        private static final long serialVersionUID = -9041750269994280136L;

        @Autowired
        private DemoService demoService;

        @FormAttribute(path="id")
        private TextField idText = new TextField("ID");
        @FormAttribute(path="title")
        private TextField titleText = new TextField("Otsikko");
        @FormAttribute(path="description")
        private TextField descriptionText = new TextField("Kuvaus");

        private Panel virhePanel = new Panel("Virheet");

        public LomakePaneeli(SpringAwareApplication application) {
                super(application);
                initComponents();
        }
```

```
	/* (non-Javadoc)
	 *                                                                    @see
com.affecto.integration.vaadin.SpringAwareCustomComponent#initComponents()
	 */
	@Override
	public void initComponents() {
			virhePanel.setVisible(false);
			Button toimintoButton = new Button("Toimintonäyttö");
			toimintoButton.addListener(new ClickListener() {

				private static final long serialVersionUID = -
2171080320743061373L;

				@Override
				public void buttonClick(ClickEvent event) {

	getSpringAwareApplication().launchAction("toiminto");
				}
			});

			Panel p = new Panel();
			p.addComponent(toimintoButton);
			p.addComponent(virhePanel);
```

```
                p.addComponent(idText);
                p.addComponent(titleText);
                p.addComponent(descriptionText);
                getSubmit().setCaption("Tallenna");
                getSubmit().addListener(this);
                p.addComponent(getSubmit());
                setCompositionRoot(p);

                getFormObject().addErrorListener(new
FormErrorListener<Papu>() {

                        private static final long serialVersionUID =
8122939403056888009L;

                        @Override
                        public void clearErrors(FormObject<Papu>
formObject) {

            virhePanel.removeAllComponents();
                                virhePanel.setVisible(false);
                        }

                        @Override
                        public void errorOccurred(FormErrorEvent
event) {
```

```
                                                    if(event.getSourceComponent()
!= null) {

                event.getSourceComponent().setComponentError(new
UserError(event.getResolvedMessage()));
                                                    }

                                                    virhePanel.setVisible(true);
                                                    virhePanel.addComponent(new
Label(event.getResolvedMessage()));
                                            }
                        });
        }

        /* (non-Javadoc)
         *                                                                  @see
com.affecto.integration.vaadin.SpringAwareCustomFormComponent#onSubmit(java.la
ng.Object)
         */
        @Override
        protected void onSubmit(Papu model) {
                virhePanel.removeAllComponents();
                virhePanel.setVisible(true);
                demoService.toiminto(model);
                getFormObject().readModel(new Papu());
```

```
                    virhePanel.addComponent(new Label("Toiminto onnistui.
Bean: " + model.getId() + ", " + model.getTitle() + ", " + model.getDescription()));

                }



}
```

LIITE 7 Lomakenäyttöön liitetyn palvelurajapinnan lähdekoodi

```
package com.hannikkala.integraatiodemo.service;


import com.hannikkala.integraatiodemo.model.Papu;


/**
 * Spring Framework - Vaadin - integraation esimerkkiohjelman palvelurajapinta.
 *
 * @author Tommi Hännikkälä <tommi@hannikkala.com>
 */
public interface DemoService {

                /**
                 * Jokin toiminto.
                 * @param papu Parametrina menevä papu.
                 */
                void toiminto(Papu papu);
}
```

LIITE 8 Lomakenäyttöön liitetyn palvelurajapinnan toteuttavan luokan lähdekoodi

```
package com.hannikkala.integraatiodemo.service;


import org.springframework.stereotype.Service;

import com.hannikkala.integraatiodemo.model.Papu;


/**

 * Spring Framework - Vaadin - integraation esimerkkiohjelman palvelurajapinnan toteutus.

 *

 * @author Tommi Hännikkälä <tommi@hannikkala.com>

 */

@Service("demoService")

public class DemoServiceImpl implements DemoService {


		/* (non-Javadoc)

		 * @see com.hannikkala.integraatiodemo.service.DemoService#toiminto(com.hannikkala.integraatiodemo.model.Papu)

		 */

		@Override

		public void toiminto(Papu papu) {

			// TODO: Luo toteutus
```

```
                }


}
```

LIITE 9 Spring Frameworkin asetustiedosto

```
<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springframework.org/schema/beans"

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

        xmlns:context="http://www.springframework.org/schema/context"

        xsi:schemaLocation="

                http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd

                http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">


        <context:component-scan                                base-
package="com.hannikkala.integraatiodemo.application" />

        <context:component-scan                                base-
package="com.hannikkala.integraatiodemo.service" />

</beans>
```