Haaga-Helia
ammattikorkeakoulu Oy

Testivetoinen Java-kehitys

Inka Haltiapuu

Opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

2016

Haaga-Helia
ammattikorkeakoulu Oy

Tiivistelmä

24.11.2016

Tekijä(t)
Inka Haltiapuu

Koulutusohjelma
Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko	**Sivu- ja liitesivumäärä**
Testivetoinen Java-kehitys	40 + 5

Opinnäytetyön tavoitteena oli tietokantaa käyttävän web-sovelluksen kehitys käyttäen testivetoista kehitysmenetelmää. Työssä selvitettiin, miten testivetoista sovelluskehitystä ja yksikkötestausta tehdään, millaisia etuja ja haittoja testivetoisesta sovelluskehityksestä on ammattikirjallisuudessa ja tutkimuksissa havaittu koituvan, ja mitä etuja ja haittoja opinnäytetön tekijä itse havaitsi testivetoisesta kehitysmallista olevan sovelluskehitystyössä.

Opinnäytetyön tuotoksena syntyi demo web-selaimen kautta käytettävästä varaussovelluksesta, joka soveltuu itsepalvelukirpputorin myyntipaikkavarausten tekoon. Tuotoksen tarkoituksena oli demonstroida pienen web-pohjaisen sovelluksen toteuttamista testivetoisesti Java-teknologioilla ja SQL-tietokannalla.

Työn teoriaosuudessa tarkasteltiin testivetoisesta sovelluskehityksestä kertovaa ammattikirjallisuutta ja aiheesta tehtyjä tutkimuksia. Havaittiin, että vaikka ammattikirjallisuudessa ja kehitysmallin mukaista sovelluskehitystyötä ammatikseen tekevien keskuudessa mallia pidetään kannattavana ja hyödyllisenä, tutkimukset eivät osoittaneet yksiselitteisesti ja ristiriidattomasti, että testivetoinen sovelluskehitys parantaisi ohjelmakoodin laatua, yksikkötestien laatua tai tuottavuutta. Tutkimuksiin tutustuessa kävi selkeästi ilmi, että tutkimuksia oli vaikeaa järjestää siten, että tuloksia olisi voitu vertailla objektiivisesti, ja toisaalta riittävän kattavaa, koko projektin elinkaaren huomioivaa tutkimusta aiheesta ei ole juuri tehty. Selvää oli, että aiheesta tulisi tehdä enemmän tutkimusta, jotta kehitysmallin hyödyistä ja haitoista voitaisiin vetää selkeitä ja ristiriidattomia johtopäätöksiä.

Opinnäytetyön tekijän oma kehitystyö testivetoisella sovelluskehitysmenetelmällä osoitti, että menetelmän noudattamisen avulla päästiin hyviin yksikkötestien kattavuuslukuihin. Testien kattavuus oli yli 90%, ja rivimäärällisesti yksikkötestaukseen liittyviä koodirivejä oli hieman enemmän kuin itse ohjelmakoodia. Kattavat testit helpottivat ohjelmavirheiden paikallistamista ja vähensivät näin virheenkorjaukseen käytettyä aikaa sekä mahdollistivat ohjelmakoodin huolettomamman refaktoroinnin ja laajentamisen, koska testien avulla voitiin nopeasti varmistaa, etteivät muutokset ohjelmakoodiin aiheuttaneet virheitä aiemmin implementoituihin toimintoihin. Aiheesta julkaistujen tutkimusten ja oman sovelluskehitystyön perusteella ei kuitenkaan voitu tulla lopputulokseen, että testivetoinen sovelluskehitys itsessään välttämättä johtaisi laadukkaampaan ohjelmakoodiin ja parempaan tuottavuuteen. Sen sijaan vaikutti siltä, että testivetoinen sovelluskehitysmalli voisi edesauttaa kattavampien yksikkötestien kirjoittamista, mikä puolestaan voi johtaa laadukkaampaan, helpommin laajennettavaan ja ylläpidettävään ohjelmakoodiin sekä korkeampaan tuottavuuteen koko projektin elinkaarta tarkasteltaessa.

Asiasanat
sovelluskehitys, testivetoinen kehitys, yksikkötestaus, Java, Spring, JUnit

Sisällys

1 Johdanto 1
2 Yksikkötestaus 4
2.1 Yksikkötestin rakenne 4
2.1.1 Yksikkötestaus Java-ympäristössä 5
2.2 Testisijaiset 6
2.3 Yksikkötestauksen parhaat käytännöt 6
3 Testivetoinen sovelluskehitys 8
3.1 Punainen-vihreä-refaktorointi 8
3.2 Testivetoisen kehittämisen ammattikirjallisuudessa mainittuja etuja 10
3.3 Tutkimustuloksia testivetoisen kehittämisen vaikutuksista 11
3.3.1 Parantaako testivetoinen kehittäminen ohjelmakoodin rakennetta? 11
3.3.2 Vertaileva tutkimus, Diep, Erdogmus, Layman, Melnik, Shull & Turhan 11
3.3.3 Vertaileva tutkimus, Münch & Mäkinen 13
3.3.4 Mitkä tekijät jarruttavat testivetoisen kehitysmallin käyttöönottoa? 15
4 Kehitystyössä käytetyt välineet 16
5 Kirpputorin varausjärjestelmädemo Paarman kehitystyö 17
5.1 Tietokannan mallinnus ja toteutus 17
5.2 Graafisen käyttöliittymän suunnittelu 17
5.3 Projektin luonti Eclipsessä 18
5.4 Yksikkötestaustyökalujen liittäminen projektiin 18
5.5 Toiminnallisuuden implementointi: Esimerkkinä Luo asiakastili 19
5.5.1 UserController-luokan testaus ja implementointi 19
5.5.2 UserService-luokan testaus ja implementointi 26
5.5.3 UserDAO-luokan testaus ja implementointi 29
5.6 Muun toiminnallisuuden implementointi pähkinänkuoressa 37
6 Pohdinta 39
Lähteet 41
Liitteet 43
Liite 1. Testitapausten pohjana käytetyt skenaariot - esimerkki 43
Liite 2. Tietokannan ER-kaavio 44
Liite 3. Graafisen käyttöliittymän prototyyppi 45

1 Johdanto

Opinnäytetyön aiheena on tietokantaa käyttävän web-sovelluksen kehitys käyttäen testivetoista kehitysmenetelmää. Työn tarkoituksena on selvittää, miten testivetoista sovelluskehitystä tehdään, millaisia etuja ja haittoja siitä on ammattikirjallisuudessa ja tutkimuksissa havaittu koituvan ja mitä hyötyä tämän työn tekijä testivetoisesta kehitysmallista kokee saavansa omassa sovelluskehitystyössään.

Opinnäytetyö on produktityyppinen, ja sen tuotos on demo web-selaimen kautta käytettävästä varaussovelluksesta, joka soveltuu itsepalvelukirpputorin myyntipaikkavarausten tekoon. Tuotoksen tarkoitus on demonstroida pienen web-pohjaisen sovelluksen toteuttamista yksikkötestivetoisesti Java-teknologioilla ja SQL-tietokannalla. Tuotos ei tule suoraan sellaisenaan minkään yrityksen käyttöön.

Kehitystyöllä on valmiin, demotarkoitukseen soveltuvan tuotoksen aikaansaamisen lisäksi opinnäytetyön tekijälle oppimistavoite. Tekijä on kiinnostunut syventämään JavaEE- ja Spring-osaamistaan ja opettelemaan yksikkötestausta.

Sovelluskehityksessä käytetään Java-ohjelmointikieltä. Sovellus ohjelmoidaan Eclipse-ohjelmointiympäristössä käyttäen Javalla tapahtuvaan web-kehitykseen soveltuvia työkaluja kuten Maven, Spring, JSP ja HTML. Testauksessa käytetään JUnitia, Mockitoa, Hamcrestia ja Spring MVC Test Frameworkia.

Tutkimuskysymykset, joihin opinnäytetyössä pyritään löytämään vastaus:

- Mitä tarkoittaa testivetoinen sovelluskehitys?
- Miten tehdään testivetoista sovelluskehitystä Javalla ja sen yksikkötestaustyökaluilla?
- Onko aiheesta julkaistuissa tutkimuksissa testivetoisesta sovelluskehityksestä todettu olevan kiistatonta hyötyä sovelluskehityksessä?
- Kokeeko tämän opinnäytetyön tekijä testivetoisesta sovelluskehityksestä olevan hyötyä sovelluskehityksessä?

Ohjelmistokehitykseen kuuluu monta eri vaihetta – määrittely ja suunnittelu, käyttöliittymäsuunnittelu, ohjelmointi ja testaus. Tässä opinnäytetyössä keskitytään sovelluksen ohjelmointivaiheeseen testivetoisen kehityksen näkökulmasta sekä toteutetaan sovelluksen toimintojen vaatima relaatiotietokanta. Opinnäytetyön tuotoksena syntyvälle varausjärjestelmälle on tehty aiemmin vaatimusmäärittely Haaga-Helia ammattikorkeakoulussa Vaatimusmäärittely-kurssilla, ja sitä käytetään ohjelmoinnin ja tietokantamallinnuksen pohja-

na. Aiheina vaatimusmäärittelyä, käyttöliittymäsuunnittelua ja muita testauksen osa-alueita kuin yksikkötestausta ei käsitellä tässä opinnäytetyössä. Varausjärjestelmän toteutukseen liittyviä aiheita, kuten olio-ohjelmoinnin ja web-palvelinohjelmoinnin periaatteita, Spring-sovelluskehystä tai MVC-mallia ei myöskään käsitellä syvällisesti niiden jäädessä tämän opinnäytetyön näkökulman ulkopuolelle.

On huomattava, että tässä opinnäytetyössä käsiteltävästä testivetoisesta kehitysmallista on olemassa sovelluskehityksen eri vaiheisiin soveltuvia tai muuten hiukan erilaisia variaatioita, kuten ATDD (Acceptance-Test Driven Development), BDD (Behavior-Driven Development) ja STDD (Story-Test-Driven Development). Tässä työssä käsitellään ainoastaan alkuperäistä kehitysmallia, jota kutsutaan testivetoiseksi kehittämiseksi, englanniksi Test-Driven Development, joka tarkoittaa suunnittelumallia, jossa yksikkötesti kirjoitetaan ennen testikohteena olevan sovelluskoodin kirjoittamista. Sekä englannin- että suomenkielisessä kirjallisuudessa testivetoiseen kehitysmalliin viitataan usein akronyymillä TDD, joten tässä opinnäytetyössäkin käytetään kyseistä akronyymiä viitattaessa testivetoiseen kehittämiseen.

Käsiteluettelo:

Java	Sun Microsystemsin kehittämä ja vapaana ohjelmistona ylläpitämä laitteistoriippumaton olio-ohjelmointikieli.
Java EE	Java Enterprise Edition, web- ja palvelinsovellusten kehittämiseen ja ajamiseen soveltuva Java-spesifikaatio, joka sisältää erilaisia ohjelmointirajapinta- ja komponenttimäärityksiä.
JUnit	Kent Beckin ja Erich Gamman kehittämä avoimen lähdekoodin yksikkötestauskirjasto Java-kielellä kirjoitetulle sovelluskoodille.
Koheesio	Olio-ohjelmoinnissa luokalta toivottava ominaisuus, jossa luokka määrittelee yhden selvästi muista erottuvan käsitteen eikä sisällä tähän käsitteeseen liittymättömiä jäseniä.
Refaktorointi	Sovelluskoodin rakenteen muuttaminen, yleensä rakenteen parantamiseksi, muuttamatta sovelluskoodin toiminnallisuutta.
Relaatiotietokanta	Tietokantamalli, jossa data on sarakkeista ja riveistä koostuvissa taulukoissa, joiden välillä on yhteyksiä eli relaatioita.
Sovelluskehys	Kokoelma tiettyyn aiheeseen liittyviä sovelluskoodikirjastoja, joiden tarjoamien valmiiden sovelluskomponenttien avulla voidaan nopeuttaa sovelluskehitystä.
Spring	Pivotal Softwaren vapaana ja avoimen lähdekoodin ohjelmistona kehittämä ja ylläpitämä sovelluskehys Java-kielellä tapahtuvaan ohjelmointiin.

SQL	IBM:n kehittämä standardoitu kyselykieli relaatiotietokantojen käsittelyyn.
TDD	Ks. testivetoinen kehitys.
Testivetoinen kehitys	Sovelluskehitysmenetelmä, jossa kirjoitetaan haluttua toiminnallisuutta varten ensin yksikkötesti ja sen jälkeen sovelluskoodi, joka läpäisee testin.
Vaatimusmäärittely	Dokumentti, joka kuvaa mm. kehitysprojektissa toteutettavalle järjestelmälle asetettuja toiminta- ja laatuvaatimuksia.
Yksikkötestaus	Yksittäisen lähdekoodin osan testaus.

Jotta voidaan ymmärtää testivetoista kehitystä, on ensin ymmärrettävä yksikkötestausta. Luvussa 2 selvitetään, mitä tarkoittaa yksikkötestaus, millainen on yksikkötestin rakenne ja mitkä ovat yksikkötestauksen parhaat käytännöt. Luvussa 3 kuvataan, mitä testivetoinen kehittäminen tarkoittaa ja mitä etuja siitä voi olla kehitystyössä. Lopuksi luodaan katsaus testivetoisen kehityksen vaikutuksista tehtyihin tutkimuksiin. Tietoperustan viimeisessä luvussa 4 käydään läpi tässä opinnäytetyössä toteutettavassa sovelluskehitystyössä käytettävät välineet. Empiirisessä osuudessa luvussa 5 kuvataan varausjärjestelmäsovelluksen kehitystyö. Luvussa 6 pohditaan työn tuloksia ja testivetoisen kehitystyön koettuja hyötyjä ja haittoja.

2 Yksikkötestaus

Sovellustestaus voidaan jakaa neljään tyypilliseen testausvaiheeseen: yksikkötestaus, integraatiotestaus, järjestelmätestaus ja hyväksymistestaus. Yksikkötestauksessa testataan ohjelmakoodin yksittäisiä osia, integraatiotestauksessa osien välistä vuorovaikutusta, järjestelmätestauksessa koko järjestelmää ja hyväksymistestauksella koko järjestelmää asiakkaan toimesta. (Samaroo 2010, 42-47.)

Yksikkötesti on yksinkertainen, nopeasti kirjoitettava ja ajettava testi, jolla testataan pientä, yksittäistä ohjelmakoodin osaa, yleensä metodia, ja testi on itsekin testiluokan metodi. Testikohteena olevan ohjelmakoodin on oltava riippumaton muusta ohjelmakoodista, ulkoisista resursseista ja ajoympäristöstä, jotta olisi selvää, mikä testin kohteena oleva toiminnallisuus on, ja testin epäonnistuessa virheen syy löytyisi mahdollisimman nopeasti. Testikohteen eristämiseksi sen riippuvuuksista käytetään erilaisia riippuvuuksia jäljitteleviä rakenteita eli testisijaisia tai testijäljitelmiä (test doubles), kuten mock, stub ja fake. (Farcic & Garcia 2015, 78-80.) Suomenkielinen termistö ei ole vakiintunutta ja kyseisiä käsitteitä käytetään sellaisenaan englanninkielisinä alan suomenkielisessä kirjallisuudessa, joten tässä opinnäytetyössä käytetään myös englanninkielisiä käsitteitä puhuttaessa erityyppisistä testisijaisista.

2.1 Yksikkötestin rakenne

Yksikkötesti koostuu neljästä osasta. Vaiheiden englanninkieliset nimet vaihtelevat; Appel (2015, 22-23) käyttää Meszarosin luokittelua, jonka mukaan vaiheet ovat nimeltään setup, exercise, verify ja teardown.

- Setup-vaiheessa alustetaan testikohteen tarvitsema kiinteä tietorakenne, kuten testissä käytettävän luokan olio tai testijäljitelmä-olio, siinä tilassa missä sen halutaan olevan ennen itse testejä. Tässä vaiheessa luodaan siis testitapauksen esiehdot, joita testattavan toiminnallisuuden käyttäminen vaatii.
- Exercise-vaiheessa käynnistetään se toiminnallisuus, jonka toimintaa testin on tarkoitus testata, esimerkiksi käytetään luokan metodia ja asetetaan metodin antama tulos muuttujaan.
- Verify-vaiheessa tarkastetaan, vastaako toiminnallisuuden aikaansaama lopputulos odotettua arvoa, esimerkiksi verrataan suoritetun metodin tuloksena muuttujaan asetettua arvoa odotettuun arvoon.
- Teardown-vaiheessa testiympäristö saatetaan takaisin siihen tilaan, jossa se oli ennen testiä. Alustukset tai itse testausaktiviteetin aikana luodut pysyvät tilat, jotka ovat edelleen olemassa, on poistettava, jotta ne eivät vaikuta seuraaviin testeihin.

2.1.1 Yksikkötestaus Java-ympäristössä

Java-kehityksessä suosituin yksikkötestauksen sovelluskehys on helppokäyttöinen JUnit. JUnit on ohjelmakoodikirjasto, jonka avulla voidaan kirjoittaa ja ajaa yksikkötestejä sekä raportoida niiden tulokset. (Koskela 2008, 36.) JUnitin logiikka seuraa edellämainittuja yksikkötestin rakenteen vaiheita. Before-metodit suorittavat alustustoimenpiteet, jotka ajetaan joko kerran ennen kaikkien luokan testien ajamista (BeforeClass-annotoitu metodi) tai kerran ennen kunkin luokan testin ajamista (Before-annotoitu metodi). Test-metodit suorittavat toiminnallisuuden sekä sen antaman arvon vertailun odotettuun arvoon ja after-metodit suorittavat siivoustoimenpiteet, joko kerran kaikkien luokan testien ajon jälkeen (AfterClass-annotoitu metodi tai kerran kunkin luokan testin ajon jälkeen (After-annotoitu metodi). (Farcic & Garcia 2015, 22-24).

```java

import org.junit.After;
import org.junit.AfterClass;
import org.junit.Before;
import org.junit.BeforeClass;
import org.junit.Test;

public class AppTest {
    @BeforeClass
    public static void beforeClass() {
        //Alusta tietorakenteet, jotka alustetaan vain kerran koko testiluokan ajon aikana.
    }
    @Before
    public void before() {
        //Alusta tietorakenteet, jotka alustetaan ennen jokaisen luokan testin ajoa.
    }
    @After
    public void after() {
        //Tee siivoustoiminnot, jotka täytyy suorittaa jokaisen luokan testin ajon jälkeen.
    }
    @Test
    public void testi1() {
        //1. Käynnistä toiminnallisuus.
        //2. Tutki, antaako toiminnallisuuden lopputulos halutun arvon.
    }
    @Test
    public void testi2() {
        //1. Käynnistä toiminnallisuus.
        //2. Tutki, antaako toiminnallisuuden lopputulos halutun arvon.
    }
    @AfterClass
    public static void afterClass() {
        //Tee siivoustoiminnot, jotka täytyy suorittaa kaikkien luokan testien ajon jälkeen.
    }
}
```

Kuva 1. JUnit-yksikkötestaustyökalulla toteutettavan yksikkötestin perusrakenne

Yksikkötestin rakenne on aina samanlainen riippumatta ohjelmointikielestä ja yksikkötestaukseen käytetystä työkalusta. Kaikille yhteinen rakenne ja selkeä vaiheiden erottelu parantavat testin luettavuutta huomattavasti. Testin strukturointi oikein on tärkeä osa testin suunnittelua, ja sitä pitäisi noudattaa, jotta testi pysyy ymmärrettävänä. (Appel 2015, 24.)

2.2 Testisijaiset

Yksittäisten sovelluskoodin osien täytyy yleensä toimia vuorovaikutuksessa muiden koodin osien, kolmansien osapuolten koodikirjastojen ja toisten järjestelmien kanssa, jotta sovellus toimisi halutulla tavalla. Yksikkötestauksessa halutaan kuitenkin eristää testikohde muista komponenteista, joista se on riippuvainen. Jos testikohde on riippuvainen komponenteista, joita ei voida kontrolloida, kuten vaikkapa verkkopalvelusta, tietokantayhteydestä, kolmannen osapuolen kirjastosta tai muista ohjelmakoodin osista, joita ei välttämättä ole vielä edes toteutettu, niiden toteutus on kesken tai niitä ei ole testattu, testin lopputuloksen oikeellisuudesta ei voida olla varmoja. Ulkoinen komponentti voi itse olla viallinen tai sen käyttäytyminen voi muuttua esimerkiksi uudemman version käyttöönoton myötä. Lisäksi komponenttien alustus ja käyttö voivat hidastaa yksikkötestien ajoa, mikä taas tekee yksikkötestaukselle olennaisesta testien säännöllisestä, tiheällä syklillä suorittamisesta epäkäytännöllistä. (Appel 2015, 45.)

Riippuvuuksien poistamiseksi käytetään testisijaisia, jotka voidaan jakaa viiteen alatyyppiin:

- **Dummy** on olio, joka ei toteuta mitään toiminnallisuutta eikä vaikuta testikohteen toimintaan mitenkään muuten kuin toimimalla riippuvuuden passiivisena sijaisena, esimerkiksi metodin parametrina, jotta ohjelmakoodin voi suorittaa.
- **Fake** on olio, jolla on jokin toimintalogiikka, minkä tarkoitus on korvata testikohteen ulkoinen riippuvuus, esimerkiksi ulkoinen tietokanta järjestelmän sisällä toimivalla tietokantaa jäljittelevällä rakenteella.
- **Stub** on olio, joka tuottaa ennaltamäärättyjä, kovakoodattuja arvoja testikohteeseen.
- **Mock** on olio, joka todentaa, että testikohde käyttää riippuvuutta halutulla tavalla.
- **Spy** on olio, joka tallentaa tietoja testikohteen käymästä vuorovaikutuksesta riippuvuuden kanssa.

Testisijaisten käyttöön on kehitetty erilaisia sovelluskehyksiä, kuten JMock, EasyMock ja Mockito, joista Mockito lienee suosituimpia. (Appel 2015, 49-58.)

2.3 Yksikkötestauksen parhaat käytännöt

Tim Ottinger ja Brett Schuchert ovat kehittäneet viisi kriteeriä, jotka hyvän yksikkötestin tulisi täyttää. Tätä periaatetta kutsutaan nimellä FIRST. Nimi on akronyymi, joka muodostuu näiden viiden kriteerin ensimmäisistä kirjaimista.

Fast: testin pitää olla nopea. Jos testit ovat hitaita, kehittäjät alkavat väistämättä ajamaan niitä harvemmin ajan säästämiseksi.

Isolated: testin pitää käsitellä selkeästi tiettyä yksittäistä toimintoa, jotta se kykenee osoittamaan virheen yksiselitteisesti. Testit pitää voida ajaa missä tahansa järjestyksessä ja testin tulos ei saa riippua muista testeistä. Testin on alustettava itse tarvitsemansa tietorakenteet ja siivottava jälkensä suorituksen päätteeksi.

Repeatable: testit pitää voida ajaa toistuvasti vaivatta. Ne eivät saa olla riippuvaisia ulkoisista resursseista, kuten järjestelmästä, tietoverkosta, tiedostoista tai tietokannasta. Testin tuloksen on oltava sama jokaisella ajokerralla kaikissa ympäristöissä, jos sen kohteena oleva ohjelmakoodi ei ole muuttunut.

Self-validating: testin täytyy yksinkertaisesti joko mennä läpi tai ei, eikä vaatia syvempää testitulosten analysointia.

Timely: testi pitää kirjoittaa ennen sen testaaman toiminnallisuuden toteuttavan sovelluskoodin kirjoittamista. Kun testit kirjoitetaan jälkikäteen, neljää edellistä periaatetta on vaikeampi seurata ja testien laatu kärsii. (Martin 2009, 132-133.)

Kehittäjä voi testata sovelluskoodia ilman yksikkötestejä tulostamalla koodin suorittamien operaatioiden tuloksia lokiin tai konsolille, mutta testauslogiikan sijoittaminen sovelluskoodin lomaan ei ole hyvien ohjelmointikäytäntöjen mukaista. Se tekee sovelluskoodista monimukaisempaa, vaikeuttaa ylläpitämistä ja voi aiheuttaa järjestelmävirheen. Tulostus- tai lokiinkirjoituslauseet ajetaan aina ohjelman ajon yhteydessä, jolloin ne tuottavat tarpeetonta tietoa, kasvattavat ohjelman suoritusaikaa ja huonontavat sovelluskoodin luettavuutta. (Acharya 2014, 9.) Hyvin toteutettu yksikkötestaus vähentää ohjelmavirheitä, parantaa ohjelmakoodin laatua, nopeuttaa kehitystyötä, toimii matalan tason dokumentaationa ja lisää kehittäjän luottamusta siihen, että ohjelmakoodia voi refaktoroida ja muokata aiheuttamatta vaikeasti jäljitettäviä ohjelmavirheitä (Appel 2015, 2-6).

3 Testivetoinen sovelluskehitys

Testivetoinen sovelluskehitys liittyy ohjelmistokehitysprojektin ohjelmointivaiheeseen, jossa tuotetaan sovellukselta vaaditut toiminnallisuudet toteuttava sovelluskoodi. Ohjelmoija testaa tuottamansa ohjelmakoodin toimivuutta kirjoittamalla automatisoituja yksikkötestejä, joiden tarkoituksena on vahvistaa, että ohjelmakoodin osat toteuttavat ohjelmalta vaaditut toiminnot ohjelman spesifikaatioihin dokumentoidulla tavalla, ennen kuin osat yhdistetään. (Samaroo 2010, 42-43.)

Perinteisessä sovelluskehitysmallissa sovelluskoodi tai sen osa kirjoitetaan ensin, ja sen jälkeen kirjoitetaan yksikkötestit. Testivetoinen sovelluskehitys tarkoittaa sitä, että toimitaan päinvastaisessa järjestyksessä perinteiseen malliin nähden. Ensin kirjoitetaan sovellukselta haluttua toiminnallisuutta testaava yksikkötesti ja vasta sen jälkeen toiminnallisuuden toteuttava sovelluskoodi. (Farcic & Garcia 2015, 3.)

3.1 Punainen-vihreä-refaktorointi

TDD:ssä toistetaan hyvin lyhyttä ja yksinkertaista iteratiivista kehityssykliä, jota kutsutaan nimellä punainen-vihreä-refaktorointi. Ensin kirjoitetaan yksinkertaisin mahdollinen yksikkötesti halutulle toiminnallisuudelle ja ajetaan kaikki testit. Uusin testi ei saa mennä läpi, koska sen kohteena olevaa sovelluskoodia ei ole vielä olemassa. Seuraavaksi kirjoitetaan mahdollisimman nopeasti ja yksinkertaisesti sovelluskoodi, jonka on läpäistävä testi. Sovelluskoodia saa kirjoittaa vain sen verran, että se läpäisee testin. Ajetaan taas kaikki testit ja todennetaan, että kaikki testit, mukaan lukien uusin testi, menevät läpi. Lopulta sovelluskoodi refaktoroidaan ja ajetaan taas kaikki testit todentaen, että ne menevät läpi. Sitten sykli aloitetaan taas alusta, lisäten vuorotellen testikoodia ja sovelluskoodia, kunnes haluttu toiminnallisuus on toteutettu. (Farcic & Garcia 2015, 3.)

Refaktorointi on tärkeä osa TDD-sykliä. Vihreässä vaiheessa ohjelmakoodi kirjoitetaan mahdollisimman nopeasti, tavoitteena vain saada testi menemään läpi, ja refaktorointivaiheessa poistetaan ohjelmakoodista mahdollinen toisto ja tarkistetaan, että ohjelmalogiikka ei ole tarpeettoman monimutkaista ja että ohjelmakoodi on selkeää. Vihreässä vaiheessa ohjelmakoodiin siis lisätään uutta toiminnallisuutta, ja refaktorointivaiheessa ohjelmakoodia optimoidaan muuttamatta sen toiminnallisuuksia. Esimerkiksi turhan toiston poistaminen ohjelmakoodista on tärkeää, koska saman asian ilmaiseminen useassa eri paikassa ohjelmakoodia johtaa siihen, että asian muuttuessa on muistettava tehdä muutokset jokaiseen paikkaan erikseen (Aejmelaeus 2009, 17).

Kuva 2. TDD:n kehityssykli

TDD:ssä siis yksikkötestit kirjoittaa kehittäjä, joka kirjoittaa myös vastaavan sovelluskoodin. Testaaminen on näin ollen ns. lasilaatikkotestausta, mikä tarkoittaa sitä, että testattavan komponentin sisäinen logiikka on testaajan tiedossa. (Aejmelaeus 2009, 11.) Yksinkertainen konkreettinen esimerkki TDD:n vaiheista voitaisiin kuvailla seuraavanlaisella laskin-esimerkillä, jossa vaadittu toiminnallisuus on, että laskin suorittaa kertolaskun:

- Luodaan testi, joka alustaa Laskin-luokan ja käyttää sen metodia kerro(3, 5) odottaen, että metodi palauttaa tuloksen 15.
- Ajetaan testi. Testi ei käänny, sillä kyseistä luokkaa ja metodia ei ole vielä olemassa. Tästä tiedetään, että testi toimii, kuten odotetaan, ja luokkaa ei jo ole olemassa.
- Seuraavaksi luodaan itse sovelluskoodiin luokka Laskin ja sille metodi kerro(luku1, luku2), joka ohjelmoidaan palauttamaan kovakoodattu arvo 0.
- Ajetaan testi, joka kääntyy mutta ei mene läpi, koska odotettu tulos oli 15 mutta saatu tulos oli 0. Näin saadaan validoitua, että kun testi kääntyy, se toimii, kuten odotetaan.
- Seuraavaksi halutaan päästä vihreään vaiheeseen. Kovakoodataan kerro-metodille palautusarvoksi 15.
- Ajetaan testi, joka menee läpi. Näin validoidaan, että testi toimii.
- Seuraavaksi koodi refaktoroidaan. Kerro-metodi ohjelmoidaan toimimaan siten, että palautettava arvo ei ole kovakoodattu vaan se saadaan kertomalla metodille annetut parametrit keskenään.
- Ajetaan testi, joka menee läpi. Toiminnallisuus on nyt valmis.

Esimerkki oli hyvin yksinkertainen, eikä todellisessa tilanteessa noin yksinkertaista toiminnallisuutta tarvitsisi välttämättä ohjelmoida noin pienin askelin. Esimerkki havainnollistaa kuitenkin hyvin TDD-kehityssyklin ja testi edellä -periaatteen. (Aejmelaeus 2009, 11-15.)

3.2 Testivetoisen kehittämisen ammattikirjallisuudessa mainittuja etuja

Testivetoinen kehittäminen on inkrementaalinen sykli, jossa sovellukseen lisätään toiminnallisuuksia yksi kerrallaan iteratiivisesti. Koska yksittäiset syklit ovat hyvin nopeita ja niiden kesto mitataan ennemmin minuuteissa kuin tunneissa tai päivissä, sovellus on jatkuvasti hyvin lähellä tilaa, jossa se toimii, sen toiminnot on testattu ja siinä ei ole juuri virheitä, jolloin se on nopeasti julkaisukelpoinen, vaikka siitä puuttuisikin jotain vaadituista toiminnoista. (Koskela 2008, 19.)

TDD tekee sovelluskoodista helpommin ymmärrettävän, ylläpidettävän, muutettavan, jatkokehitettävän tai refaktoroitavan, koska kehitysmalli pakottaa kehittäjän kirjoittamaan kattavat testit ja suunnittelemaan sovelluskoodin rakenteen mahdollisimman modulaariseksi siten, että eri osat ovat mahdollisimman vähän riippuvaisia toisistaan. Koska TDD:n yksi teesi on, että ohjelmakoodia saa kirjoittaa vain sen verran, että yksikkötesti läpäistään, testien kattavuus TDD:n mukaan kehitetyssä sovelluksessa on yleensä noin 90%. Esimerkiksi Martinin kehittämässä FitNesse-sovelluksessa 45000 koodirivistä lähes 20000 riviä oli yksikkötestejä. Kun testien kattavuuteen voidaan luottaa ja koodin rakenne on modulaarinen, kehittäjällä on matalampi kynnys refaktoroida koodia, mikä parantaa sovelluskoodin laatua, ja refaktoroidessa tai lisättäessä tai muutettaessa toiminnallisuuksia sovellukseen on pienempi riski, että muutokset aiheuttavat vaikeasti jäljitettäviä, aikaavieviä virheitä. Kattavat ja ajantasaiset yksikkötestit toimivat myös luotettavasti sovelluksen matalan tason dokumentaationa. (Martin 2007, 33-35.)

Osa kehitystyöstä on debuggausta eli virheiden etsintää ohjelmakoodista. Virheet tulevat kalliiksi. Debuggaukseen voi kulua aikaa jopa 100-200% suhteessa ohjelmointiin käytettyyn aikaan. Debuggaukseen käytettävän ajan arviointi projektin aikataulua suunnitellessa on vaikeaa. Testivetoisesti suunnitellessa virheet löytyvät nopeasti, koska testit ovat kattavia ja niitä ajetaan jatkuvasti. Virhe on helposti jäljitettävissä ja siten nopeammin korjattavissa. Sabre ja Workshare ovat raportoineet ohjelmavirheiden vähentyneen kymmenkertaisesti testivetoisen kehityksen seurauksena, mikä on kohottanut tuottavuutta. TDD helpottaa projektin aikataulun arviointia, kun vaikeasti arvioitavissa olevan osan suhteellinen osuus projektiin käytettävästä ajasta pienenee merkittävästi. (Martin 2007, 34-35.)

TDD:n eduiksi lasketaan myös lopputuotteen parempi vastaavuus sille määriteltyihin vaatimuksiin (Koskela 2008, 8-10). Kun yksikkötestit kirjoitetaan sovelluskoodin kirjoituksen jälkeen, saattaa käydä niin, että testit kirjoitetaan vahvistamaan, mitä sovelluskoodi tekee sen sijaan että ne vahvistaisivat sen, että sovellus tekee sitä, mitä asiakas siltä vaatii.

Testit, kuten vaatimusmäärittely, pitäisi täten kirjoittaa ennen sovelluskoodia, jotta ne määrittelisivät sovelluskoodin eikä toisinpäin. (Farcic & Garcia 2015, 8-9.)

3.3 Tutkimustuloksia testivetoisen kehittämisen vaikutuksista

Alan ammattikirjallisuuden kirjoittajien kertomien näkemysten lisäksi on hyödyllistä tarkastella myös alan tutkijoiden näkemyksiä. TDD:n vaikutuksista on tehty tieteellistä tutkimusta, joista tässä tarkastellaan neljää.

3.3.1 Parantaako testivetoinen kehittäminen ohjelmakoodin rakennetta?

Aniche & Gerosan (2015, 4-9) kvalitatiivisessa tutkimuksessa tutkittiin 14 brasilialaista kehittäjää kuudesta eri yrityksestä. Tutkimuksessa kehittäjille annettiin ohjelmointitehtäviä ratkaistavaksi sekä testivetoisen kehitysmallin mukaan että ilman sitä, ja sen tavoitteena oli selvittää, parantaako TDD kehittäjien mielestä sovelluskoodin rakennetta.

Kehittäjien haastattelu osoitti, että 13 kehittäjää 14:stä koki, ettei testivetoinen kehittäminen vaikuttanut siihen, miten he suunnittelivat sovelluksen luokkarakenteen, vaan ammatillinen kokemus ja ymmärrys olio-ohjelmoinnin hyvistä käytännöistä vaikuttivat suunnitteluun eniten. Moni kehittäjistä oli kuitenkin sitä mieltä, että testivetoinen kehittäminen vaikutti positiivisesti sovelluksen ulkoiseen laatuun. 11 kehittäjää sanoi, että koodin refaktorointi ja toiminnallisuuksien muuttaminen tuntui turvallisemmalta, kun tehtyjä muutoksia voi jatkuvasti validoida luodulla testijoukolla. Nopea palaute ohjelmakoodin toimivuudesta koettiin tässä suhteessa myös arvokkaaksi - kehittäjät kokivat uskaltavansa tehdä rohkeammin muutoksia koodiin tietäessään, että sen aiheuttaessa virheen, syy olisi helposti identifioitavissa. He mainitsivat myös, että testien kirjoittaminen ennen sovelluskoodin kirjoittamista auttoi hahmottamaan, miten luokat ovat keskenään vuorovaikutuksessa, ja noudattamaan siten paremmin hyviä olio-ohjelmointikäytäntöjä.

3.3.2 Vertaileva tutkimus, Diep, Erdogmus, Layman, Melnik, Shull & Turhan

Diep, Erdogmus, Layman, Melnik, Shull & Turhan (2010b, 210) kävivät läpi satoja tutkimuksia liittyen TDD:hen, valiten lopulta 32 eri tutkimusta tarkempaan analyysiin selvittääkseen, onko TDD:llä vaikutusta sovelluksen laatuun. Melnik, joka on alan arvostettu tutkija ja noudattaa työssään Microsoftilla testivetoista kehitysmallia, kommentoi tutkijoiden tutkimuksesta kirjoittamassa lehtiartikkelissa (2010a, 17) analyysin tuloksia oman kokemuksensa valossa.

Ensimmäiseksi tarkasteltiin sovelluksen ulkoista laatua. Tutkimukset tarjosivat kohtalaisia todisteita siitä, että TDD parantaa sovelluksen ulkoista laatua siten, että tehtävät on hyvin määritelty, regressiotestausta on tehty säännöllisesti, ohjelmavirheitä on vähemmän, virheet on löydetty aiemmassa vaiheessa ja muutostiheys on pienempi. Vaikutus näytti selkeältä vähemmän kurinalaisesti tieteellistä metodia noudattaneissa tutkimuksissa, kun taas kontrolloiduissa tutkimuksissa tulokset olivat epäselviä. Vaaka kääntyi positiiviseksi, kun kaikkien tutkimusten tuloksille annettiin yhtäläinen arvo. (Diep ym. 2010b, 210-212.) Melnik oli tuloksista samaa mieltä; hänen mukaansa TDD vähentää virheitä, motivoi kurinalaisempaan ohjelmointiin ja lyhentää keskimääräistä virheenkorjausaikaa. (Diep ym. 2010a, 17.)

Toisena kohteena oli ohjelmakoodin sisäinen laatu, kuten olio-ohjelmoinnin hyvien käytäntöjen mukainen rakenne ja ohjelmakoodin kompleksisuus. Tutkijoiden mukaan tutkimusten tuloksiset olivat ristiriitaisia eikä niistä voitu vetää johtopäätöstä, että TDD parantaisi tai huonontaisi ohjelmakoodin sisäistä laatua. Se vaikutti vähentävän joitain koodin ei-toivottuja sisäisiä ominaisuuksia, kuten kompleksisuutta ja turhaa toistoa, mutta huonontavan toisia, kuten lisäävän komponenttien välisiä kytkentöjä ja madaltavan koheesiota. TDD vaikutti tuottavan sovelluskoodia, joka oli vähemmän kompleksista luokka- ja metoditasolla, mutta enemmän kompleksista pakkaus- ja projektitasolla. Tutkijat huomauttivat kuitenkin, että tulokset saattoivat riippua myös TDD:n ulkopuolisista seikoista, kuten kehittäjien vaihtelevista ohjelmointitaidoista. (Diep ym. 2010b, 212.) Melnik oli ristiriitaisista tuloksista painokkaasti eri mieltä. Hänen mukaansa kehitystiimit, jotka ovat tottuneita käyttämään TDD:tä, tuottavat puhtaampaa, ymmärrettävämpää, ylläpidettävämpää ja helpommin laajennettavaa ohjelmakoodia. Hän mainitsi, että heidän tiiminsä siirryttyä TDD-malliin ohjelmavirheiden määrä laski. Hän huomautti kuitenkin myös, että kehitystiimin kokemus ja ammattitaito voi vaikuttaa siihen, kuinka paljon vaikutusta laatuun TDD:hen siirtymisellä on. (Diep ym. 2010a, 17-18.)

Kolmas tutkimuskohde oli TDD:n vaikutus tuottavuuteen. Tutkijat totesivat, että tämä on TDD:n kontroversiaalisin ulottuvuus, koska pitkäaikaisvaikutuksista ei ole päästy yksimielisyyteen. Puolustajat argumentoivat, että tuottavuus kohoaa, koska ohjelmavirheitä on vähemmän, ne huomataan nopeasti, niiden korjaaminen käy nopeammin ja automatisoitujen testien jatkuvan ajamisen myötä uusia ohjelmavirheitä tehdään harvemmin, kun taas vastustajat huomauttavat, että TDD:ssä testien kirjoittamiseen kuluva aika laskee tuottavuutta. Eri tutkimukset antoivat niin erilaisia tuloksia, että artikkelin kirjoittajat eivät voineet päätellä muuta kuin että suurten testijoukkojen hallinta ei osoita johdonmukaista tuottavuuden laskua. (Diep ym. 2010b, 213.) Melnikin mukaan alkuvaiheessa, kehittäjien vasta opetellessa TDD:tä, tuottavuus voi laskea hetkeksi, mutta pitkällä tähtäimellä vaikutus

tuottavuuteen riippuu siitä, miten tuottavuutta lasketaan. Erilaiset laskutavat vaikuttavat siihen, millaisia tuloksia asian tutkimisesta saadaan. Melnik raportoi, että hänen TDD-mallia noudattava tiiminsä kirjoitti vähemmän ohjelmakoodia testikoodi mukaan lukien, koodin uudelleentyöstämisen tarve oli pienempi ja koodin ylläpito oli helpompaa virheiden korjaamiseen käytetyn ajan laskiessa, mikä vaikuttaa sovelluksen koko elinkaaren kustannuksiin. Lopuksi Melnik vielä huomautti, että TDD tulisi ymmärtää ennemmin suunnittelu- kuin kehitystekniikkana, koska sen päätavoite on saada kehittäjä miettimään järjestelmän organisointia, paremman testauksen ollessa oikeastaan vain sivuvaikutus. (Diep ym. 2010a, 18.)

Neljänneksi tarkasteltiin testien laatua eli niiden määrää, kattavuutta, tuottavuutta ja niihin panostamista. Jälleen tutkimukset antoivat ristiriitaisia tuloksia, mutta kaikki tulokset yhteenlaskien tutkijat olivat sitä mieltä, että on hiukan viitteitä siitä, että TDD ei ainakaan huononna testien laatua ja johtaa usein parempaan laatuun kuin vaihtoehtoiset menetelmät. (Diep ym. 2010b, 214.)

3.3.3 Vertaileva tutkimus, Münch & Mäkinen

Münch & Mäkinen (2014, 4-12) analysoivat 19 tieteellistä julkaisua, jotka käsittelivät TDD:tä. He identifioivat 10 sisäistä ja ulkoista laatuominaisuutta, joihin TDD vaikutti: ohjelmavirheiden määrä ja tiheys, testien kattavuus, ohjelmakoodin kompleksisuus, luokkien väliset kytkennät, testien tai ohjelmakoodin koko, työhön käytetty aika, ulkoinen laatu, tuottavuus ja ylläpidettävyys.

Lukuisat TDD:n positiivisista vaikutuksista liittyivät ohjelmavirheisiin tai vikoihin, ja myös ulkoiseen laatuun, kompleksisuuteen, kokoon ja ylläpidettävyyteen liittyi positiivisia vaikutuksia. Negatiivisesti TDD oli useimmiten vaikuttanut työmäärään sekä tuottavuuteen, ja jossain määrin myös kytkentöihin. Vaikutukset ohjelmakoodin kokoon ja testien kattavuuteen osoittautuivat epämääräisiksi.

Ohjelmavirheiden määrää oli TDD:n käyttöönoton jälkeen saatu vähennettyä mm. IBM:llä ja Microsoftilla, joista ensimmäisellä ohjelmavirheiden määrä oli puolittunut, vaikkakin vakavien ohjelmavirheiden suhteellinen määrä oli pysynyt samana. Jälkimmäisellä virheiden määrä oli vähentynyt 60-90%. Kaikissa tutkimuksissa, esimerkiksi sellaisissa joiden osanottajia olivat opiskelijat eivätkä ammattilaiset, eroja virhetiheydessä ei oltu todettu.

Testien kattavuudessa oli ammattilaisten keskuudessa päästy keskimäärin 80-90 %:iin TDD:n käyttöönoton seurauksena. Tutkijat huomauttivat, että opiskelijat, joiden ohjelmoin-

tikokemus oli ylipäätään vähäisempi kuin ammattilaisten, eivät kyenneet yltämään samoihin kattavuusprosentteihin TDD:n käyttöönotosta huolimatta kokemuksen vähyyden vuoksi.

Koodin kompleksisuuden, eli luokkien ja metodien rakenteen monimutkaisuuden, havaittiin vähenevän TDD:n seurauksena etenkin ammattilaisilla. Opiskelijoita käsittelevät tutkimukset antoivat ristiriitaisia tuloksia.

Luokkien väliset kytkennät ja luokkien koheesio ovat hiukan vaikeasti mitattavia ominaisuuksia, joita ei useissa tutkimuksissa mitata. Tutkijat löysivät vain kolme tutkimusta, jossa nämä oli otettu huomioon, ja kaikissa näissä tutkimuksissa TDD:n vaikutus oli ollut negatiivinen.

Testikoodin kokoa tutkittaessa havaittiin, että esimerkiksi Microsoftilla testirivien määrän suhde sovelluskoodin rivien määrään TDD-projekteissa oli joissain tapauksissa jopa 0,89 ja vähimmillään 0,39 riippuen projektin koosta. IBM:llä luku oli keskimäärin 0,48. Tutkijat huomauttivat, että myös ilman TDD:tä voidaan päästä kohtuullisiin suhdelukuihin, jos noudatetaan hyviä testauskäytäntöjä, mutta luvut jäävät useimmiten silti TDD-projektien lukujen alle. Tässä tapauksessa myös opiskelijoilla TDD:n vaikutus suhdelukuun oli sama tai vain hiukan alempi. Mitä taas tuli sovelluskoodin luokkien ja metodien sisältämään rivimäärään, joissain tapauksissa sen oli havaittu laskeneen, toisissa ei. Eräässä tutkimuksessa havaittiin, että sovelluskoodia oli vähemmän kutakin käyttäjätarinaa kohti TDD:tä käyttävässä projektissa.

Työhön käytetty aika kasvoi joissain tapauksissa. Esimerkiksi IBM ja Microsoft arvioivat TDD:n pidentävän kehitystyöhön käytettyä aikaa 10-30%. Muissakin tapauksissa kehittäjät olivat kokeneet, että kehitystyöhön käytetty aika oli kasvanut TDD:n seurauksena. Muutama tutkimus ei osoittanut TDD:llä olevan vaikutusta työhön käytettyyn aikaan.

Ulkoinen laatu, joka tässä ymmärretään käyttäjien tai asiakkaiden kokemuksena tuotteen laadusta, ei näyttänyt muuttuvan suuntaan tai toiseen TDD:tä käytettäessä. Kahdessa tutkimuksessa ammattilaisilta ja opiskelijoilta itseltään kysyttäessä he olivat sitä mieltä, että laatu olisi jossain määrin korkeampi.

Tuottavuudesta oltiin useassa tapauksessa sitä mieltä, että se oli laskenut TDD:n käyttöönoton seurauksena, koska kehittäjät joutuivat käyttämään enemmän aikaa testikoodin kirjoittamiseen. Pienessä määrässä tutkimuksia vaikutus oli ollut päinvastainen.

Ylläpidettävyydestä ei ollut tehty havaintoja kuin yhdessä tutkimuksessa. Sen mukaan muutosten tekeminen myöhemmin oli onnistunut TDD-mallin mukaan tehdylle sovellukselle nopeammin, ja kehittäjät olivat itse sitä mieltä, että sovelluskoodi oli ylläpidettävämpää.

3.3.4 Mitkä tekijät jarruttavat testivetoisen kehitysmallin käyttöönottoa?

TDD:n käyttöönoton esteitä ovat tutkineet Causevic, Punnekkat & Sundmark (2011, 337-344) analysoimalla 48 TDD:stä tehtyä empiiristä tutkimusta. Heidän aiempi tutkimuksensa oli osoittanut, että TDD:tä käytetään vähemmän kuin sitä todellisuudessa haluttaisin käyttää.

Tutkijat identifioivat analyysissaan 7 tekijää, jotka jarruttivat TDD:n käyttöönottoa sovelluskehitysalalla.

- Kehitystyöhön käytetyn ajan kasvaminen koettiin ongelmallisena, vaikka pitkällä tähtäimellä koko projektiin käytetyn ajan mahdollisen lyhenemisen ja tuotteen laadun paranemisen myötä tämä ei välttämättä vaikuttaisi tuottavuuteen.
- TDD:n osaamisen, tiedon ja taidon puute rajoitti mallin käyttöönottoa useissa tapauksissa.
- TDD-mallin protokollaan olennaisesti kuuluva etukäteen tehdyn suunnittelutyön vähyys aiheutti ongelmia etenkin suurien ja monimutkaisten sovellusten kohdalla.
- Kehittäjien testaustaitojen puute nähtiin ongelmana joissain tapauksissa, sillä TDD perustuu kehittäjän kykyyn suunnitella hyviä testejä.
- TDD-protokollan seuraaminen muodostui hankalaksi joissain tapauksissa. Kehittäjät eivät seuranneet sitä riittävän tarkasti, jolloin TDD:n hyötyjä ei ehkä saavutettu. Tiukka aikataulu saattoi olla tässä kohdassa merkittävä osatekijä.
- Ongelmat työkalujen käyttöönotossa rajoittivat useissa tapauksissa TDD:n käyttöönottoa. Etenkin graafisen käyttöliittymän ja verkkopohjaisten sovellusten testaustyökalujen suhteen oli ollut ongelmia.
- TDD:n käyttö työstäessä jo olemassaolevaa, vanhaa koodia, oli koettu vaikeaksi. Kehitysmalli perustuikin alun perin nimenomaan uuden ohjelmakoodin tuottamiseen.

Tutkijat huomauttivat oman ja myös muiden tekemän aiemman tutkimuksen osoittaneen, että TDD:n koettiin parantavan ohjelmakoodin laatua, ja tutkimukseen osallistuvien asenne testivetoista kehitystä kohtaan oli yleisesti ottaen positiivinen.

Tutkijoiden mielestä jarruttavia tekijöitä olisi syytä tutkia lisää. Lisätutkimusten kohteeksi he mainitsivat TDD:n oppimiskäyrän vaikutuksen kehitystyöhön käytettyyn aikaan ottaen huomioon koko projektin elinkaaren, ei vain testien ja sovelluskoodin kirjoittamiseen käytettyä aikaa. Muita hyödyllisiä tutkimuskohteita tutkijoiden mukaan olisivat etukäteen tehdyn suunnittelutyön puutteen aiheuttamat haasteet monimutkaisissa sovelluksissa, yksikkötestien huonon laadun vaikutukset TDD:n onnistumiseen sekä tarvittava yksikkötestauksen osaamisen taso, jotta TDD voi onnistua.

4 Kehitystyössä käytetyt välineet

Java-ohjelmointi tehtiin avoimen lähdekoodin ohjelmointiympäristössä, Eclipse Neon.1:ssä. Ohjelmistoprojektin hallinnan aputyökaluna käytettiin Mavenia. Sovelluskehyksenä käytettiin Springiä, avoimen lähdekoodin Java-ohjelmointiin tarkoitettua sovelluskehystä. Spring tarjoaa apuvälineitä käyttöliittymä-, liiketoiminta- ja tietovarastokerroksen toteuttamiseen. Web-sivujen käyttöliittymäkerroksen dynaamisen sisällön käsittelyyn käytettiin JSP-teknologiaa. Tässä työssä käytettiin Javan versiota 8, Mavenin versiota 4, Springin versiota 4.0, Servletin versiota 3.1 ja JSP:n versiota 2.3.

Web-sovelluksen ajamiseen käytettiin Tomcatia. Tomcat on sovelluspalvelin, jolla voidaan suorittaa Java-servletejä ja kuvantaa web-sivuja, jotka sisältävät JSP-ohjelmakoodia. Tässä työssä käytettiin versiota 8.

Tietokannan mallintamiseen käytettiin UMLet-nimistä työkalua, joka on itsenäisenä sovelluksena tai Eclipseen lisäosana asennettava ilmainen, avoimen lähdekoodin UML-mallinnustyökalu. Tässä työssä käytettiin versiota 14.2.

Tietokantana käytettiin sovelluksen sisällä ajettavaa HSQLDB-tietokantaa.

Graafisen käyttöliittymän prototyypin suunnitteluun käytettiin NinjaMock-nimistä sovellusta. Se on selainpohjainen graafisten käyttöliittymien rakenteen suunnittelusovellus eli mockup-työkalu.

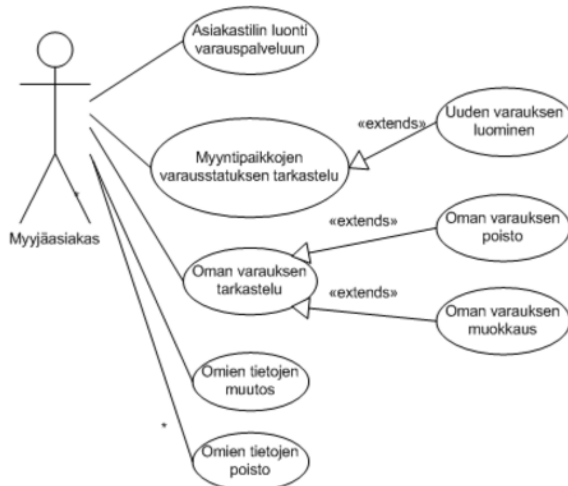
Yksikkötestaustyökaluina toimivat JUnit, Spring Test Framework, Hamcrest ja Mockito. Spring Test Framework on testikirjasto, joka on nimenomaan optimoitu Spring-komponenteilla tehdyn ohjelmakoodin yksikkö- ja integraatiotestaamiseen ja sitä voi käyttää JUnitin tai TestNG:n kanssa. Hamcrest puolestaan on testikirjasto, jonka avulla voidaan kirjoittaa monipuolisempia saadun ja halutun tuloksen vertailijoita, *matchereita*. Mockiton avulla voidaan luoda mock-olioita.

Testien kattavuuden tarkasteluun käytettiin Eclipseen lisäosana saatavaa EclEmma-työkalua. EclEmman avulla voidaan tarkastella yksityiskohtaisesti, mitä koodirivejä Junit-testit kattavat.

Versionhallintaan käytettiin Gitiä, Eclipsen Git-työkalua EGitiä sekä GitHub Desktopia paikallisen koodivaraston ja etävaraston synkronointiin. Ohjelmakoodi on nähtävillä julkisessa etävarastossa GitHubissa osoitteessa https://github.com/InkaH/paarma.

5 Kirpputorin varausjärjestelmädemo Paarman kehitystyö

Sovelluksen toteutuksen pohjana käytettiin Paarmalle tehtyä vaatimusmäärittelyä, jonka toteuttivat Vaatimusmäärittely-kurssilla tämän opinnäytetyön tekijä yhdessä Lasse Leipolan kanssa. Paarma-demo toteuttaa vaatimusmäärityksessä mainituista käyttötapauksista myyjäasiakkaan vaatimat toiminnallisuudet. Käyttötapauksia tarkentavia skenaarioita käytettiin yksikkötestien suunnittelun pohjana (Liite 1).



Kuva 3. Paarma-demon toteuttamat käyttötapaukset

5.1 Tietokannan mallinnus ja toteutus

Mallinnettiin SQL-tietokanta ER-mallilla UML-luokkakaavioita käyttävällä kuvaustekniikalla. ER-mallinnus on laajasti käytetty oliopohjainen menetelmä käsitteiden ja niiden suhteiden kuvaamiseen. Mallinnuksen työvälineenä käytettiin UMLetia, jonka saa lisättyä Eclipse-ympäristöön lisäosana, ja mallinnuksen pohjana toimi Paarman vaatimusmäärittelyssä määritelty säilytettävien tietojen luokkakaavio. ER-mallin mukaisesti mallinnuksessa kuvattiin kohdetyypit, kohdetyyppien attribuuttien nimet, tietotyypit ja arvojoukot sekä kohdetyyppien väliset yhteydet ja osallistumisrajoitteet. (Liite 3.)

5.2 Graafisen käyttöliittymän suunnittelu

Sovelluksen graafisen käyttöliittymän rakenne suunniteltiin niin ikään vaatimusmäärittelyn skenaarioiden pohjalta. Työkaluna prototyypin suunnittelussa käytettiin NinjaMockia. (Liite 4.) Vaikka demon graafisen ilmeen suunnittelu jäikin tämän työn ulkopuolelle, auttaa käyttöliittymän rakenteen suunnittelu myös itse sovelluskehitystyötä.

5.3 Projektin luonti Eclipsessä

Luotiin uusi Spring MVC-projekti Eclipse-kehitysympäristössä. Näin Eclipse loi projektille jo valmiiksi tyypillisen tiedostorakenteen ja Spring MVC-projektin perusrungon tarvitsemat moduulit Mavenin Project Object Modeliin eli POMiin. POM on xml-tiedosto, joka sisältää informaatiota projektista sekä Mavenin sovelluksen kääntämiseen tarvitsemat konfiguraatiotiedot. Maven-projektin pom.xml –tiedosto laajentaa oletuksena Super POM:in, joka sisältää tyypillisiä projektin konfiguraatioita, jotka ovat kaikille projekteille usein yhteisiä. Sovellukseen tarvittavia moduuleita voi hakea Mavenin ylläpitämästä moduulikirjastosta osoitteesta mvnrepository.com.

Vietiin projekti Git-versionhallintaan ja GitHubiin. Määriteltiin Tomcat 8 -palvelin ajoympäristöksi.

5.4 Yksikkötestaustyökalujen liittäminen projektiin

Yksikkötestaukseen tarvittiin 4 moduulia, jotka löytyivät Mavenin moduulikirjastosta – JUnit-moduuli Javan yksikkötestaukseen yleisesti sekä Spring-sovellusten testaukseen erityisesti optimoitu moduuli spring-test, joka sisältää esim. WebApplicationContextin ja Servlet API:n mockaukseen soveltuvia luokkia, jotka ovat hyödyllisiä testattaessa Spring MVC -sovelluksia. Hamcrest puolestaan tarjoaa metodeita, ”matchereita”, jotka helpottavat odotetun ja saadun tuloksen erityyppisiä vertailuja toisiinsa. Mock-olioita on kätevää luoda Mockitolla. Liitettiin moduulit pom.xml-tiedostoon.

```
<!-- Test -->
<dependency>
    <groupId>org.hamcrest</groupId>
    <artifactId>hamcrest-all</artifactId>
    <version>1.3</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
    <exclusions>
        <exclusion>
            <artifactId>hamcrest-core</artifactId>
            <groupId>org.hamcrest</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>1.9.5</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-test</artifactId>
    <version>${spring.version}</version>
    <scope>test</scope>
</dependency>
```

Kuva 4. Yksikkötestaukseen tarvittavat moduulit Mavenin Project Object Modelissa

5.5 Toiminnallisuuden implementointi: Esimerkkinä Luo asiakastili

Ensimmäisenä toiminnallisuutena implementoitiin asiakastilin luominen, joka toteutti käyttötapauksen Luo asiakastili MA01. Jotta toiminnallisuuden implementoinnin demonstroiminen tässä työssä sujuisi yksinkertaisemmin ja kuvista ei tulisi niin suuria, päätettiin rajoittaa käyttäjän, eli User-luokan, käyttämiä attribuutteja kolmeen: id, firstName ja lastName. Lopullisessa toteutuksessa lisättäisiin kaikki loput vaatimusmäärittelyn ja tietokantamallin mukaiset attribuutit.

Sovellus kehitettiin MVC-mallin mukaisesti siten, että käyttöliittymä, liiketoimintalogiikka ja tietovarastoyhteydet olivat toisistaan erillisissä kerroksissa, ja datan välittäjänä käyttöliittymän ja muiden palveluiden välillä toimi controller. Mallin mukaisesti controller välittää datan service-luokille datan käsittelemiseksi sovelluksen liiketoimintalogiikan vaatimalla tavalla, ja service-luokat käyttävät DAO-luokkia datan välittämiseksi tietokannalle ja takaisin. Service-luokat tarjoavat palveluita domain-luokille, jotka vastaavat usein esim. yksittäistä tietokantataulua. MVC-arkkitehtuurin toteuttamiseen ei ole yhtä ainoaa tapaa, vaan kehittäjien käyttämät ohjelmistorakenteet voivat erota hieman toisistaan.

5.5.1 UserController-luokan testaus ja implementointi

Kehitystyö aloitettiin controllerin testauksesta ja toteutuksesta. Controllerin testauksessa haluttiin todentaa, että http-pyynnöt ja vastaukset sisältävät haluttua dataa ja ohjautuvat haluttuihin paikkoihin, mikä on web-lomakkeiden käsittelyssä olennainen seikka.

Toteutuksen ensimmäisellä TDD-kierroksella haluttiin todentaa, että

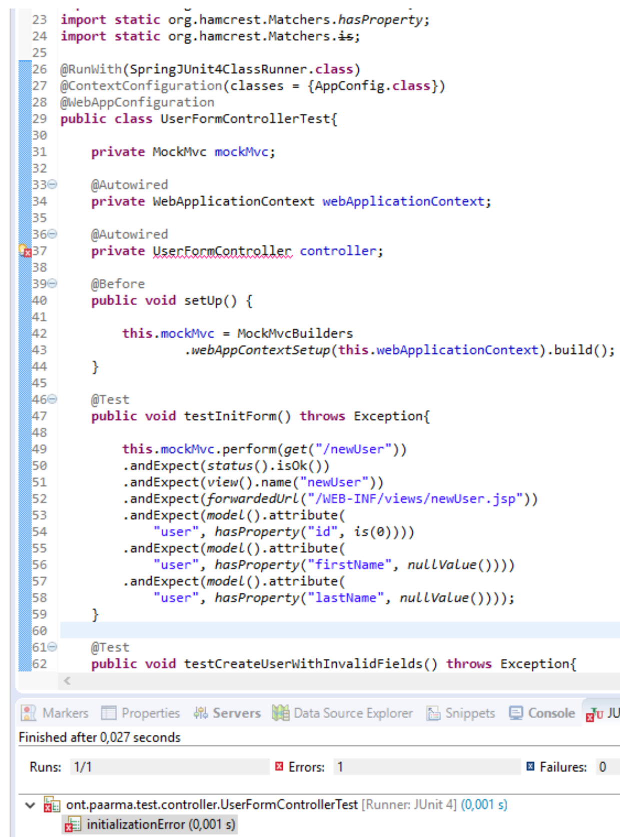
- GET-pyyntö käsitellään osoitteeseen /newUser saavuttaessa,
- pyynnön palauttama http-statuskoodi on 200 (Ok),
- näkymän nimi on newUser,
- edelleenohjaus tapahtuu haluttuun osoitteeseen ja
- osoitteeseen saavuttaessa controller asettaa modeliin tyhjän User-olion, jonka attribuutteihin on asetettu kyseisen attribuutin tietotyypin oletusarvo (esim. int-tyypille 0 ja String-tyypille null), jotta käyttäjälle voidaan näyttää näkymässä tyhjä lomake ja Spring tietää, mitkä käyttäjän lomakekenttiin syöttämät tiedot kuuluvat mihinkin User-olion attribuutteihin.

@RunWith(SpringJUnit4ClassRunner.class) –annotaatiolla käytettiin Spring JUnit –runneria, jonka avulla on helppo asettaa Spring ApplicationContexteja testeille, kuten seu-

raavalla rivillä sitten tehtiinkin annotaatiolla @ContextConfiguration, asettaen kontekstiksi AppConfig-luokan, joka sisältää sovelluksen tarvitsemia asetuksia. @WebAppConfiguration-annotaatiota tarvittiin vielä servletin kontekstin asettamiseksi. UserFormControllerTest-luokalle haettiin konstruktori-injektiolla pari beania, UserForm-Controller ja WebApplicationContext, Before-metodissa luotiin uusi MockMvc-olio testeissä käytettäväksi, ja itse testissä sitten testattiin yllämainittuja toiminnallisuuksia.

Spring MVC Test –sovelluskehyksen olennaiset luokat testattaessa http-pyyntöjen käsittelyä ovat MockMvc, MockMvcRequestBuilders ja MockMvcResultMatchers. MockMvc:n tarjoamalla perform -metodilla MockMvcRequestBuilder mockaa halutun HTTP-pyynnön ja MockMvcResultMatcher sekä Hamcrest-matcherit tutkivat saadut vastaukset verraten niitä haluttuihin vastauksiin.

TDD-mallin mukaisesti testi tehtiin ennen toiminnallisuuden implementointia, joten koska controlleria ei oltu vielä toteutettu, testi ei tietystikään mennyt läpi.



Kuva 5. UserFormControllerin 1. testin punainen vaihe ja JUnitin tuottama tulos

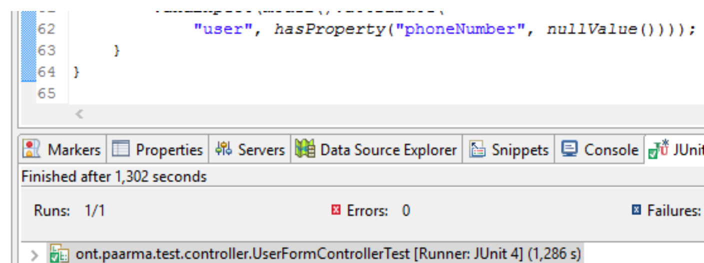
Seuraavaksi luotiin UserFormController toteuttamaan halutut toiminnot, jotta testi saatiin menemään läpi. Controlleriin konfiguroitiin haluttu url, http-pyynnön tyyppi ja paluuarvona näkymän nimi. Luotiin myös User-luokka käyttäjän säilytettäviä tietoja varten, jolla oli tie-

tokantamallin Kayttaja-taulun mukaisesti attribuutit id, firstName ja lastName gettereineen ja settereineen. Kuten edellä mainittiin, implementaation demonstrointiin tässä työssä ei otettu mukaan kaikkia attribuutteja sujuvamman esityksen vuoksi. Luokan attribuuteille annettiin myös validaatiosääntöjä, kuten @NotEmpty eli attribuutilla on oltava jokin arvo, ja @Length määrittelemään attribuutin minimi- ja/tai maksimipituus. Luokasta parametrittomalla oletuskonstruktorilla luotu olio asetettiin palautettavaksi http-pyynnön mukana nimellä ”user”, jotta Spring voi luoda olion attribuutteihin bindatun tyhjän lomakkeen näkymään.

```
1  package ont.paarma.controller;
2
3  import ont.paarma.model.User;
4
5  import org.springframework.stereotype.Controller;
6  import org.springframework.ui.Model;
7  import org.springframework.web.bind.annotation.RequestMapping;
8  import org.springframework.web.bind.annotation.RequestMethod;
9
10 @Controller
11 public class UserFormController{
12     @RequestMapping(value = "/newUser", method = RequestMethod.GET)
13     public String setUpForm(Model model) {
14         User user = new User();
15         model.addAttribute("user", user);
16         return "newUser";
17     }
18 }
19
```

Kuva 6. UserFormControllerin toteutus

Controllerin ja User-luokan toteutuksen jälkeen testit ajettiin taas läpi, ja JUnit näytti vihreää tällä kertaa.



Kuva 7. UserFormControllerin 1. testi, vihreä vaihe

Seuraava vaihe oli refaktorointi eli ohjelmakoodin siistiminen ja optimointi toiminnallisuutta muuttamatta. Tällä kierroksella ohjelmakoodi kattoi niin pienen osan toiminnallisuutta, että refaktoroitavaa ei controllerissa ollut. Siirryttiin seuraavalle kierrokselle, eli luomaan testiä, joka todentaa, että uuden käyttäjän antamat tiedot käsitellään oikein.

Toteutuksen toisella TDD-kierroksella haluttiin todentaa, että controller osaa käsitellä käyttäjän lomakkeella lähettämän epävalidin datan halutulla tavalla:

- osoitteesta /newUser tuleva POST-pyyntö käsitellään metodissa,

- http-pyynnön contentType on application/x-www-form-urlencoded,
- http-pyynnön parametreina lähetetään lomakkeen kenttien tiedot,
- istuntoon asetetaan uusi User-olio,
- pyynnön palauttama http-statuskoodi on 200 (Ok),
- palautetun näkymän nimi on newUser, eli palataan takaisin lomakkeeseen koska data oli epävalidia,
- edelleenohjaus tapahtuu haluttuun osoitteeseen,
- http-pyynnön parametreinä saapuvat firstName- ja lastName -attribuutit eivät ole validointisääntöjen mukaisia,
- parametreinä saadut attribuutit ovat samat, mitä testin alussa sille annettiin ja
- mitään mock-olion metodeita ei kutsuttu testin aikana, koska annettu data oli epävalidia jolloin kyseisen mock-olion palveluita ei tule vielä käyttää.

Lisättiin testiluokkan käyttöön käyttäjätietojen käsittelylle palveluja tarjoava UserService-luokka annotaatiolla @Autowired, jota käytettäisiin mock-oliona. Lisättiin testiluokan setUp()-metodiin UserService-luokasta luodun userServiceMock-olion arvojen nollaus Mockiton avulla testien välissä metodilla Mockito.reset(), jotta eri testeissä asetetut arvot eivät vuotaisi muihin testeihin.

Testin apuluokaksi tehtiin luokka TestUtil, jolla voitiin luoda testidataa halutuilla parametreilla. Tässä testissä haluttiin luoda String-muuttujia, joista toinen oli User-luokassa asetettua 50 merkin maksimipituutta pitempi ja toinen 2 merkin minimipituutta lyhyempi. Mock-olion vuoksi tarvittiin vielä toinenkin testiapuluokka, jossa mock-olioita voitaisiin luoda. Luotiin luokka TestContext, johon määriteltiin UserService-luokasta luotavaksi mock-olio Mockiton mock()-metodilla.

Sen jälkeen toteutettiin yllämainittujen toiminnallisuuksien testaus asettamalla mockMvc-oliolle POST-pyyntö, sille haluttu osoite, contentType, parametrit eli ne lomakkeen kentät, jotka käyttäjä tulisi täyttämään, sekä sessioattribuutiksi User-olio. Sen jälkeen MockMvc-luokan .andExpect-metodeilla tarkistettiin, että saadut tulokset vastasivat odotettuja tuloksia. Tulosten tarkastelun sujuvoittamiseksi käytettiin Hamcrestin matchereita hasProperty() ja is(). Lopuksi Mockiton verifyZeroInteractions()-metodin avulla tarkastettiin, että userServiceMock-olion kanssa ei tapahtunut kommunikointia, koska data oli epävalidia, joten käyttäjää ei saanut lisätä eli UserService-luokan add()-metodia ei saanut kutsua.

```
39  @Autowired
40  private UserService userServiceMock;
41
42  @Before
43  public void setUp() {
44
45      //reset mock between tests
46      Mockito.reset(userServiceMock);
47
48      this.mockMvc = MockMvcBuilders
49              .webAppContextSetup(this.webApplicationContext).build();
50  }
51
53  public void testInitForm() throws Exception{
66
67  @Test
68  public void testCreateUserWithInvalidFields() throws Exception{
69      String firstName = TestUtil.createString(51);
70      String lastName = TestUtil.createString(1);
71
72      this.mockMvc.perform(post("/newUser")
73              .contentType(MediaType.APPLICATION_FORM_URLENCODED)
74              .param("firstName", firstName)
75              .param("lastName", lastName)
76              .sessionAttr("user", new User())
77      )
78              .andExpect(status().isOk())
79              .andExpect(view().name("newUser"))
80              .andExpect(forwardedUrl("/WEB-INF/views/newUser.jsp"))
81              .andExpect(model().attributeHasFieldErrors("user", "firstName"))
82              .andExpect(model().attributeHasFieldErrors("user", "lastName"))
83              .andExpect(model().attribute("user", hasProperty("id", is(0))))
84              .andExpect(model().attribute("user", hasProperty("firstName", is(firstName))))
85              .andExpect(model().attribute("user", hasProperty("lastName", is(lastName))));
86       verifyZeroInteractions(userServiceMock);
87  }
88 }
```

Markers | Properties | Servers | Data Source Explorer | Snippets | Console | JUnit | Task List | Out

Finished after 0,021 seconds

Runs: 1/1 Errors: 1 Failures: 0

ont.paarma.test.controller.UserFormControllerTest [Runner: JUnit 4] (0,004 s)
initializationError (0,004 s)

Kuva 7. Käyttäjätietojen luonti invalidilla datalla, punainen vaihe

TDD-mallin mukaisesti testi luotiin ennen toiminnallisuuksien ja mahdollisesti tarvittavien uusien luokkien implementointia, joten testi ei mennyt läpi. Vihreään vaiheeseen pääsemiseksi luotiin tarvittava UserService-luokka, jolla oli add()-metodi käyttäjän tietojen lisäämiseksi. ja lisättiin UserControlleriin addUser()-metodi, joka vastaanottaisi /newUser-osoitteeseen tulevia http-POST-pyyntöjä ja käsittelisi niiden välittämiä parametreja halutulla tavalla. Tässä vaiheessa haluttiin implementoida vain testin läpiviemiseksi tarvittava koodi, eli käsittely invalidille datalle ja siitä seuraavan näkymän palautus sekä myös validin datan lähettämisestä seuraava UserServicen add()-metodin kutsu, jotta voitaisiin todentaa että sitä ei suoritettu invalidia dataa vastaanotettaessa.

```
18  @Controller
19  @SessionAttributes("user")
20  public class UserFormController{
21
22      private UserService service;
23
24      @Autowired
25      public UserFormController(UserService service){
26          this.service = service;
27      }
28
30      public String setUpForm(Model model) { 
35
36      @RequestMapping(value = "/newUser", method = RequestMethod.POST)
37      public String add(@Valid @ModelAttribute("user") User user,
38              BindingResult result, RedirectAttributes attributes){
39              if(result.hasErrors()) {
40                  return "newUser";
41              }
42
43              User addedUser = service.add(user);
44              return null;
45      }
46  }
47
```

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console
nished after 1,576 seconds
Runs: 2/2    Errors: 0    Failures:
ont.paarma.test.controller.UserFormControllerTest [Runner: JUnit 4] (1,557 s)
```

Kuva 8. Controllerin toteutus invalidin datan käsittelyä varten. Alhaalla näkyvissä JUnitin ilmoitus että testit ovat menneet läpi controllerin uuden metodin sekä UserService-rajapinnan implementoinnin jälkeen.

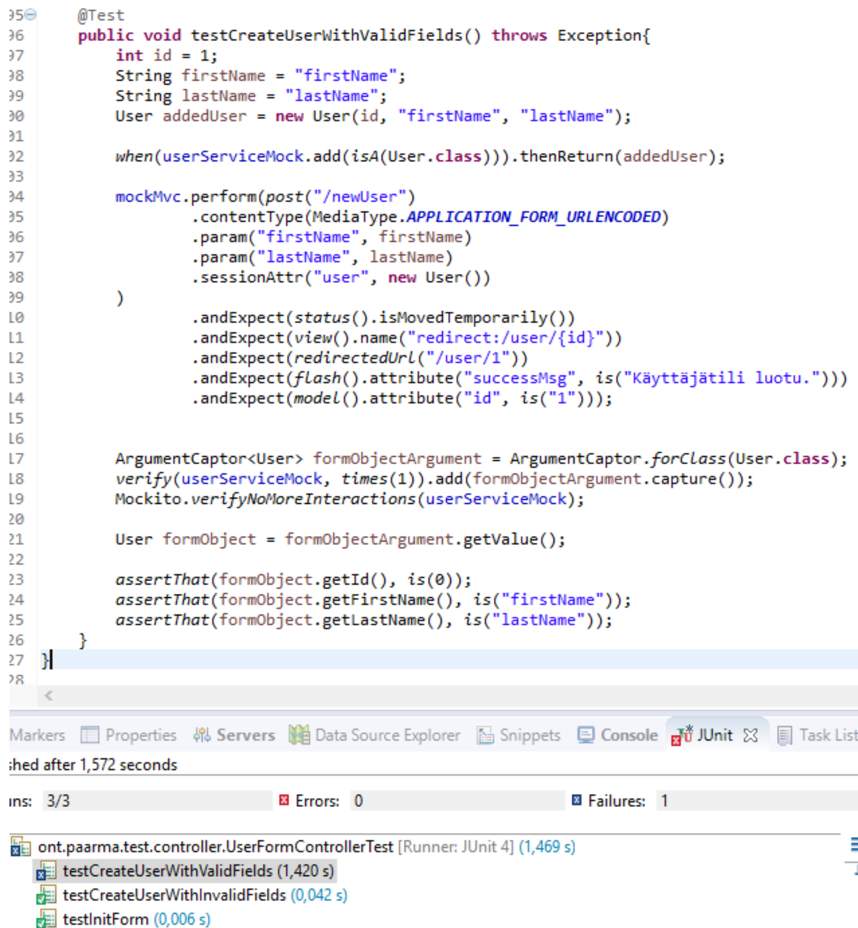
Refaktoroitavaa ei taaskaan ollut, joten edettiin seuraavaalle kierrokselle – testaamaan validin datan lähetystä ja käsittelyä.

Kolmannella TDD-kierroksella haluttiin todentaa, että

- valideilla attribuuteilla vastaanotettu olio välitetään edelleen UserServicen add()-metodin käsiteltäväksi ja se palauttaa controllerille olion takaisin,
- pyynnön palauttama http-statuskoodi on 302, jota käytetään uudelleenohjaukseen,
- palautetun näkymän nimi on ”redirect:user/{id},
- uudelleenohjaus tapahtuu haluttuun osoitteeseen ”/user/1”,
- id-attribuutin arvo on 1,
- mock-olion add()-metodia kutsutaan vain kerran ja sen parametrina oli User-luokan olio,
- mock-oliota ei kutsuta enää tämän jälkeen ja
- User-luokan attribuuttien arvot ovat oikeat.

Luotiin User-luokasta olio testiarvoilla 1, ”firstName” ja ”lastName”. Mockiton .when- ja .thenReturn -metodeilla asetettiin userService-mock-olio palauttamaan add-metodia kutsuttaessa addedUser-olion.

Tällä kierroksella uutena testimetodina käytettiin Mockiton ArgumentCaptoria. Sen avulla voidaan testata argumentin arvoja, kun sille suoritetaan toimintoja. Tässä sillä testattiin, että User-oliolla, joka http-pyynnössä välitetään userServicen add()-metodin parametrina, on halutut arvot eli id:nä 0 koska id-arvoa ei todellisuudessa anneta koskaan käyttäjän toimesta, ja etu- ja sukunimiarvoina ne arvot, jotka http-pyynnön parametreina annettiin. Mockiton VerifyNoMoreInteractions-metodilla tarkastettiin, että userServicea ei käytetty enää add()-operaation jälkeen.



Kuva 9. Punainen vaihe. Testi ei mene läpi, koska validilla datalla käyttäjän lisäystä ei ole vielä implementoitu.

Odotetusti testi ei mennyt läpi. Seuraavaksi implementoitiin haluttu toiminnallisuus. Controlleriin lisättiin validin datan käsittelylogiikka uudelleenohjauksineen ja luotiin UserService-rajapinnan implementoiva luokka ja siihen add()-metodin implementointi. Tässä vaiheessa metodi ei tehnyt muuta kuin asetti olion id-attribuuttiin kovakoodatun arvon, joka myöhemmin luodaan tietokannan avulla, ja palautti olion, koska käyttäjän tiedot tulee näyttää onnistuneen lisäyksen jälkeen uudella sivulla.

```
39     @RequestMapping(value = "/newUser", method = RequestMethod.POST)
40     public String add(@Valid @ModelAttribute("user") User addedUser,
41             BindingResult result, RedirectAttributes attributes){
42             if(result.hasErrors()) {
43                 return "newUser";
44             }
45 
46             User addedToDbUser = service.add(addedUser);
47             attributes.addAttribute("id", addedToDbUser.getId())
48                           .addFlashAttribute("successMsg", "Käyttäjätili luotu.");
49             return createRedirectViewPath("/user/{id}");
50     }
51 
52     private String createRedirectViewPath(String requestMapping) {
53         StringBuilder redirectViewPath = new StringBuilder();
54         redirectViewPath.append("redirect:");
55         redirectViewPath.append(requestMapping);
56         return redirectViewPath.toString();
57     }
58 }
```

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console  JUnit
inished after 1,587 seconds
Runs: 3/3        Errors: 0        Failures: 0
> ont.paarma.test.controller.UserFormControllerTest [Runner: JUnit 4] (1,472 s)
```

Kuva 10. Controllerin add-metodin valmis implementointi. Alla Junitin kaikki testit näyttävät vihreää.

Näiden toiminnallisuuksien toteuttamisen jälkeen testit menivät läpi. Controllerin oikea toiminta uuden käyttäjän lisäyksessä oli nyt todennettu, ja seuraavaksi siirryttiin toteuttamaan uuden käyttäjän lisäys Service-kerroksessa.

5.5.2 UserService-luokan testaus ja implementointi

UserService-luokan testauksessa haluttiin todentaa, että luokka käyttää UserDAO-kerrosta tarkoituksenmukaisesti välittäen datan controllerista DAO-luokalle ja tehden datalle tarvittaessa liiketoimintalogiikan vaatimia toimenpiteitä. Testeillä testattiin, että

- UserServicen add()-metodia kutsuttaessa se kutsuu UserDAO:n addUser()-metodia kerran ja
- UserServicen add()-metodi palauttaa User-olion arvoilla id=1, firstName=”etunimi” ja lastName=”sukunimi”, kun metodi saa parametrikseen User-olion arvoilla firstName=”etunimi” ja lastName=”sukunimi”. Koska User-oliolle luodaan id lopulta vasta tietokannan toimesta, id-tietoa ei ole vielä käyttäjäsyötteenä eikä täten UserService-luokan add()-metodiin tulevassa User-oliossa, mutta DAO-luokan käsittelystä palatessaan User-oliolla on oltava id-arvo.

Luotiin testiluokka UserServiceTest, jolle luotiin kaksi testiä testaamaan yllämainittuja toimintoja.

```java
13 public class UserServiceTest {
14
15     @Autowired
16     private UserDAO userDAOMock;
17     @Autowired
18     private UserService userService;
19
20     @Before
21     public void setUp() {
22         userDAOMock = Mockito.mock(UserDAO.class);
23         userService = new UserService(userDAOMock);
24         //reset mock between tests
25     }
26
27     @After
28     public void tearDown(){
29         Mockito.reset(userDAOMock);
30     }
31
32     @Test
33     public void testUserServiceAddMethod_callsUserDaoAddUserOnce() {
34         User user = TestUtil.createTestUserNoId();
35         userService.add(user);
36         Mockito.verify(userDAOMock).addUser(user);
37     }
38
39     @Test
40     public void testUserServiceAddMethod_userDaoAddUserReturnsUserObject() {
41         User userNoId = TestUtil.createTestUserNoId();
42         User userWithId = TestUtil.createTestUserWithId();
43         Mockito.when(userDAOMock.addUser(userNoId)).thenReturn(userWithId);
44         User actual = userService.add(userNoId);
45         Assert.assertEquals(userWithId, actual);
46     }
```

Markers | Properties | Servers | Data Source Explorer | Snippets | Console | JUnit | Git S

Finished after 0,014 seconds

Runs: 1/1 | Errors: 1 | Failures: 0

ont.paarma.test.UserServiceTest [Runner: JUnit 4] (0,001 s)
initializationError (0,001 s)

Kuva 11. UserService-luokan testaus, punainen vaihe

Testin aluksi luotaisiin testattavasta luokasta olio, ja sen tarvitsemista riippuvuuksista mock-oliot. SetUp-vaiheessa siis luotiin UserService-luokan olio sekä UserService-luokan käyttämästä UserDAO-luokasta mock-olio, joka palautettaisiin myös alkutilaan jokaisen testin jälkeen. TestUtil-luokkaan luotiin kaksi metodia User-olioiden luomiseen. Ensimmäisessä testissä käytettiin UserService-luokan metodia add() ja Mockitolla verifioitiin, että metodin käytön tuloksena UserDAO-luokan mock-olion addUser-metodia kutsuttaisiin kerran.

```
 4
 5 public class TestUtil {
 6
 7     public static String createString(int length) {
 8         StringBuilder builder = new StringBuilder();
 9
10         for (int index = 0; index < length; index++) {
11             builder.append("x");
12         }
13
14         return builder.toString();
15     }
16
17     public static User createTestUserNoId(){
18         return new User("firstName", "lastName");
19     }
20
21     public static User createTestUserWithId(){
22         return new User(1, "firstName", "lastName");
23     }
24 }
25
```

Kuva 12. TestUtil-luokka

Toisessa testissä luotiin User-luokan oliot TestUtil-luokan avulla ja asetettiin Mockitolla UserDAO-luokan mock-olio palauttamaan User-luokan olio etu- ja sukunimellä sekä id:llä, kun sille annettaisiin parametrinä User-luokan olio etu- ja sukunimellä. Käytettiin UserService-luokan metodia add() antaen sille parametriksi User-luokan olio etu- ja sukunimellä ja Assert-metodilla verrattiin, tuottiko metodi tuloksena DAO-luokan palauttaman olion id:llä, etunimellä ja sukunimellä.

Ensimmäisessä vaiheessa testit eivät menneet läpi, koska UserService-luokkaa ei oltu vielä implementoitu halutulla tavalla ja UserDAO-luokkaa ei vielä ollut olemassa. Seuraavaksi implementoitiin luokat.

```
 7
 8 @Service
 9 public class UserService{
10     private UserDAO userDAO;
11
12     public UserService(UserDAO userDAO) {
13         this.userDAO = userDAO;
14     }
15
16     public User add(User user) {
17         return userDAO.addUser(user);
18     }
19 }
```

Kuva 13. UserService-luokan implementointi

UserService-luokka toteutettiin siten, että luokka käyttäisi UserDAO-luokkaa ja add-metodissa sen addUser-metodia palauttaen DAO-luokan palauttaman olion. UserDAO-luokkaan ei vielä toteutettu toiminnallisuuksia, vaan luotiin vain kyseinen luokka jotta UserService-luokan testaus olisi mahdollista.

```java
import ont.paarma.model.User;

public class UserDAO {

public User addUser(User user) {

        throw new UnsupportedOperationException();
    }
}

```

Kuva 14. UserDAO-luokan luonti ilman toiminnallisuuksia

Kun luokat oli implementoitu, ajettiin testit uudelleen todentaen, että kaikki testit menivät läpi. Lopuksi refaktoroitiin ohjelmakoodi ja ajettiin taas kaikki testit todentaen, että ne näyttivät edelleen vihreää.

5.5.3 UserDAO-luokan testaus ja implementointi

UserDAO-luokan testauksessa haluttiin todentaa, että luokka käyttää tietokantaa tarkoituksenmukaisesti. Tietokantana käytettiin HSQLDB-tietokantaa sovelluksensisäisesti, mikä soveltui hyvin demo-tyyppisen pienen sovelluksen tietokannan toteuttamiseen.

Jotta tietokanta saatiin sovelluksen käyttöön, lisättiin tarvittavat konfiguraatiot. Ensin Mavenin POM:iin lisättiin HSQLDB- ja Spring JDBC-kirjastot.

```xml
    <!-- DB -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>${spring.version}</version>
    </dependency>
    <!-- HyperSQL DB -->
    <dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>2.3.1</version>
    </dependency>
```

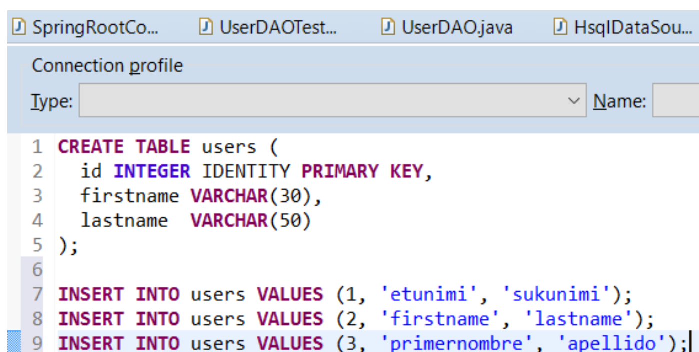
Kuva 14. Tietokannan käyttöön tarvittavat kirjastot POM:ssa

Sitten määriteltiin Springin konfiguraatioihin DataSource-bean, jossa kerrottiin kyseessä olevan EmbeddedDatabase tyyppiä HSQL ja annettiin tiedostopolut tietokannan create- ja insert-tiedostoihin.

```java
package ont.paarma.config;

import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabase;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseBuilder;
import org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType;

@Profile("hsql")
@Configuration
public class HsqlDataSource {

    @Bean
    public DataSource dataSource() {

        //shutdown unnecessary, EmbeddedDatabaseFactoryBean does it
        EmbeddedDatabaseBuilder builder = new EmbeddedDatabaseBuilder();
        EmbeddedDatabase db = builder.setType(EmbeddedDatabaseType.HSQL)
                .addScript("db/sql/create-db.sql")
                .addScript("db/sql/insert-data.sql").build();
        return db;
    }
}
```

Kuva 15. HSQL-tietokannan alustus Springissä

Seuraavaksi luotiin tietokannan luonti- ja lisäyslauseet testauskäyttöä varten. Tässä vaiheessa luotiin ainoastaan yksi taulu käyttäjän id-, etunimi- ja sukunimitietoja varten ja lisättiin tauluun pari riviä testidataa.




```sql
CREATE TABLE users (
  id INTEGER IDENTITY PRIMARY KEY,
  firstname VARCHAR(30),
  lastname  VARCHAR(50)
);

INSERT INTO users VALUES (1, 'etunimi', 'sukunimi');
INSERT INTO users VALUES (2, 'firstname', 'lastname');
INSERT INTO users VALUES (3, 'primernombre', 'apellido');
```

Kuva 16. Tietokannan luonti- ja lisäyslauseet. Toteutuksessa nämä on eritelty omiin tiedostoihinsa.

Sen jälkeen tarvitsi vain lisätä Springin RootConfig-tiedostoon tietokanta-beanin injektointi sekä JdbcTemplate-luokka tietokantayhteyden ja kyselyiden suorittamiseen.

```
 9
10 @Configuration
11 @ComponentScan({"ont.paarma"})
12 public class SpringRootConfig {
13
14     @Autowired
15     DataSource dataSource;
16
17     @Bean
18     public NamedParameterJdbcTemplate getNamedParameterJdbcTemplate()
19         return new NamedParameterJdbcTemplate(dataSource);
20     }
21
22     //db manager for hsqldb
```

Kuva 17. JdbcTemplaten lisäys ja DataSource-beanin injektointi

Seuraavaksi tehtiin UserDAO-luokan testit. Testeissä haluttiin tällä kierroksella todentaa, että UserDAO-luokan addUser()-metodia kutsuttaessa parametrinä User-olio, jolle on annettu tietyt arvot kenttiin firstName ja lastName, metodi palauttaa

- User-olion, joka ei ole null,
- jolla on id-arvo, joka on 4, sillä id:n luonti on asetettu auto incrementiksi ja testikannasta tiedetään, että edellisen tietueen id:n arvo on 3 ja
- jolla on samat arvot kentissä firstName ja lastName kuin User-oliolla, joka metodille välitettiin parametrina.

```
12 import ont.paarma.dao.UserDAO;
13 import ont.paarma.model.User;
14
15 public class UserDAOTest {
16
17     private EmbeddedDatabase db;
18     UserDAO userDao;
19
20     @Before
21     public void setUp() {
22         db = new EmbeddedDatabaseBuilder()
23             .setType(EmbeddedDatabaseType.HSQL).addScript("db/sql/create-db.sql")
24             .addScript("db/sql/insert-data.sql").build();
25     }
26
27     @After
28     public void tearDown() {
29         db.shutdown();
30     }
31
32     @Test
33     public void testAddUser(){
34         NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(db);
35         UserDAO userDao = new UserDAO();
36         userDao.setNamedParameterJdbcTemplate(template);
37         User user = TestUtil.createDBUser();
38         User returned = userDao.addUser(user);
39         Assert.assertNotNull(returned);
40         Assert.assertEquals(4, returned.getId());
41         Assert.assertEquals(user.getFirstName(), returned.getFirstName());
42         Assert.assertEquals(user.getLastName(), returned.getLastName());
```

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console  JUnit  Git Staging
Finished after 0,426 seconds
Runs: 1/1        Errors: 1        Failures: 0

ont.paarma.test.UserDAOTest [Runner: JUnit 4] (0,4(
    testAddUser (0,408 s)

Failure Trace
java.lang.Error: Unresolved compilation problems:
    The method setNamedParameterJdbcTemplate(NamedParame
    The method addUser(User) is undefined for the type UserDAO
```

Kuva 18. UserDAO-luokan testit, punainen vaihe

Testiluokassa ensin setUp()-metodissa luotiin tietokanta ja määriteltiin sen tarvitsemien sql-skriptien sijainti. TearDown-vaiheeseen määriteltiin tietokantayhteyden sulkeminen. Itse testissä alustettiin JdbcTemplate, UserDAO-luokan olio, jolle annettiin JdbcTemplate sql-toimintojen suorittamiseksi sekä luotiin TestUtil-luokan avulla User-luokan olio TesUtil-luokkaan lisätyn uuden luontimetodin avulla. Testattavan metodin paluuarvo asetettiin toiseen muuttujaan, jotta seuraavissa Assert-lauseissa voitiin vertailla, olivatko metodille välitetyn ja sen palauttaman olion arvot toisiaan vastaavia.

Koska UserDAO-luokan addUser-metodia ei vielä oltu implementoitu, testi ei mennyt läpi. Seuraavaksi implementoitiin metodi nopeasti kovakoodaamalla se palauttamaan halutut arvot sekä lisättiin UserDAO-luokkaan JdbcTemplate-riippuvuus. Arvojen kovakoodaus ensimmäisellä implementointikierroksella kuuluu testivetoisessa kehitysmenetelmässä suotavaan toimintapaan. Sen avulla on helppo varmistaa, että itse testi toimii kuten pitää-kin.

```java
14 import ont.paarma.model.User;
15
16 @Repository
17 public class UserDAO {
18
19     NamedParameterJdbcTemplate namedParameterJdbcTemplate;
20
21     @Autowired
22     public void setNamedParameterJdbcTemplate(NamedParameterJdbcTemplate namedParameterJdbcTemplate) {
23         this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;
24     }
25
26     public User addUser(User user) {
27
28         return new User(4, "testFirst", "testLast");
29     }
30
```

Kuva 19. UserDAO-luokan ja addUser-metodin ensimmäinen implementointi kovakoodatuilla arvoilla

Sitten ajettiin testit todentaen, että ne menevät läpi, kun metodi palauttaa kovakoodattuja arvoja. AddUser()-metodissa luodulla oliolla on luonnollisesti samat etu- ja sukunimitiedot kuin testiluokassa luodulla ja metodille parametrinä lähetetyllä oliolla.

```java
30     }
31
32     @Test
33     public void testAddUser(){
34         NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(db);
35         UserDAO userDao = new UserDAO();
36         userDao.setNamedParameterJdbcTemplate(template);
37         User user = TestUtil.createDBUser();
38         User returned = userDao.addUser(user);
39         Assert.assertNotNull(returned);
40         Assert.assertEquals(4, returned.getId());
41         Assert.assertEquals(user.getFirstName(), returned.getFirstName());
42         Assert.assertEquals(user.getLastName(), returned.getLastName());
43     }
```

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console  JUnit  Git Staging
Finished after 0,448 seconds
Runs: 1/1    Errors: 0    Failures: 0
ont.paarma.test.UserDAOTest [Runner: JUnit 4] (0,40    Failure Trace
    testAddUser (0,400 s)
```

Kuva 20. UserDAO-luokan addUser()-metodin ensimmäisen implementoinnin testaus, vihreä vaihe

Testi meni odotetusti läpi, joten siirryttiin refaktorointivaiheeseen implementoimaan metodi dynaamiseksi. AddUser-metodi haluttiin toteuttaa siten, että kun tietokantaan viedään User-olion firstName- ja lastName-arvot ja tietokanta luo uudelle riville automaattisesti id:n, luotu id-arvo saadaan tietokannasta talteen ja asetetaan oliolle sen setId-metodilla, jotta lopulta saadaan tilin luoneelle käyttäjälle luontitapahtuman jälkeen tulostettua näytölle käyttäjän antamien tietojen lisäksi myös id:n arvo, joka sovelluksessa toimisi asiakasnumerona. Tätä varten tietokantakyselyn suorittavalle NamedParameterJdbcTemplate-oliolle annettiin sql-kyselyn ja siihen User-oliosta saatujen parametrien lisäksi Keyholder, johon tietokannan luoma id-arvo asetettiin JdbcTemplaten update()-metodin suorituksessa.

```java
 8  import org.springframework.beans.factory.annotation.Autowired;
 9  import org.springframework.jdbc.core.RowMapper;
10  import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
11  import org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate;
12  import org.springframework.jdbc.core.namedparam.SqlParameterSource;
13  import org.springframework.jdbc.support.GeneratedKeyHolder;
14  import org.springframework.jdbc.support.KeyHolder;
15  import org.springframework.stereotype.Repository;
16
17  import ont.paarma.model.User;
18
19  @Repository
20  public class UserDAO {
21
22      NamedParameterJdbcTemplate namedParameterJdbcTemplate;
23
24      @Autowired
25      public void setNamedParameterJdbcTemplate(NamedParameterJdbcTemplate namedParameterJdbcTemplate) {
26          this.namedParameterJdbcTemplate = namedParameterJdbcTemplate;
27      }
28
29      public User addUser(User user) {
30
31          SqlParameterSource parameters = new BeanPropertySqlParameterSource(user);
32          KeyHolder idHolder = new GeneratedKeyHolder();
33          String sql = "INSERT INTO users (firstname, lastname) VALUES (:firstName, :lastName)";
34
35          namedParameterJdbcTemplate.update(
36                  sql, parameters, idHolder);
37
38          int generatedId = (idHolder.getKey().intValue());
39          user.setId(generatedId);
40          return user;
41      }
```

Kuva 21. UserDAO-luokan addUser()-metodin dynaaminen implementointi

Implementoinnin jälkeen ajettiin UserDAO-testit uudelleen läpi todentaen, että testitulos oli vihreä. Seuraavaksi päätettiin vielä testata, että luotu käyttäjä löytyisi tietokannasta id-arvon perusteella hakien. Tätä varten tehtiin testimetodi testFindById(), jossa yksinkertaisesti asetettiin userDao-luokan findById()-metodin paluuarvo User-olioon ja tarkistettiin Asserteilla, että olion arvot olivat samat kuin edellisessä testissä saadun olion arvot eli voitaisiin todentaa, että kyseinen rivi löytyy edelleen tietokannasta.

```java
46      @Test
47      public void testFindById() {
48          NamedParameterJdbcTemplate template = new NamedParameterJdbcTemplate(db);
49          UserDAO userDao = new UserDAO();
50          userDao.setNamedParameterJdbcTemplate(template);
51
52          User user = userDao.findById(4);
53          Assert.assertNotNull(user);
54          Assert.assertEquals(4, user.getId());
55          Assert.assertEquals("testFirst", user.getFirstName());
56          Assert.assertEquals("testLast", user.getLastName());
57      }
58  }
```

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console  JUnit  Git Staging
Finished after 0,531 seconds
Runs: 2/2    Errors: 1    Failures: 0

ont.paarma.test.UserDAOTest [Runner: JUnit 4] (0,51      Failure Trace
    testAddUser (0,468 s)                                 java.lang.Error: Unresolved compilation problem:
    testFindById (0,048 s)                                The method findById(int) is undefined for the type UserDAO
```

Kuva 22. UserDAO-luokan findById()-metodin testaus, punainen vaihe

Testin luonnin jälkeen ajettiin UserDAO-testit todentaen, että uusi testi ei mennyt läpi. Sen jälkeen siirryttiin implementoimaan käyttäjän haku id:n perusteella UserDAO-luokkaan.

```
43⊖    public User findById(int id) {
44
45         Map<String, Object> params = new HashMap<String, Object>();
46         params.put("id", id);
47
48         String sql = "SELECT * FROM users WHERE id=:id";
49         User result = namedParameterJdbcTemplate.queryForObject(
50                 sql,params, new UserMapper());
51         System.out.println(result);
52         return result;
53     }
54
55⊖    private static final class UserMapper implements RowMapper<User> {
56
57⊖        public User mapRow(ResultSet rs, int rowNum) throws SQLException {
58             User user = new User();
59             user.setId(rs.getInt("id"));
60             user.setFirstName(rs.getString("firstname"));
61             user.setLastName(rs.getString("lastname"));
62             return user;
63         }
64     }
```

Kuva 23. UserDAO-luokan findById()-metodin implementointi

Metodia varten piti luoda apumetodi UserMapper, joka sijoittaisi tietokannasta saatavan tietueen User-olioon. Tietokantakysely sai parametrikseen metodille parametrina välitetyn id-arvon, jonka perusteella se haki id-arvoa vastaavan tietueen taulusta.

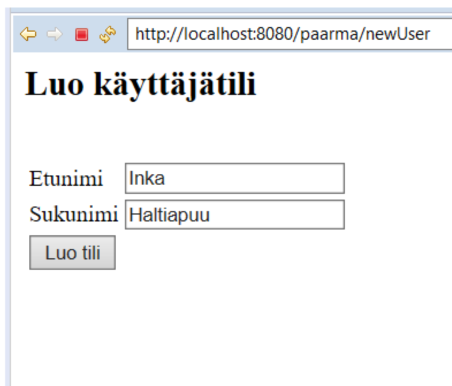
Testiä ajettaessa testi ei kuitenkaan mennyt läpi ja saatiin virheilmoitus, että odotettu result oli 1 mutta saatu oli 0. Riviä ei siis löytynyt taulusta. Tämä johtui siitä, että testiluokassa (ks. Kuva 18) tietokanta luotiin aina uudelleen ja suljettiin ennen jokaista testiä, jolloin testAddUser()-testissä lisätty rivi ei ollutkaan enää tietokannassa testFindById()-testiä suoritettaessa. Tämä ratkaistiin muuttamalla setUp()- ja tearDown()-metodit @BeforeClass- ja @AfterClass -annotoiduiksi metodeiksi, eli ne suorittaisiin vain kerran, @BeforeClass ennen kaikkia luokan testejä ja @AfterClass niiden jälkeen. Näin tietokantaan ensimmäisessä testissä lisätty rivi säilyi tietokannassa toisen testin ajoon saakka.

```
17 public class UserDAOTest {
18
19     private static EmbeddedDatabase db;
20     UserDAO userDao;
21
22     @BeforeClass
23     public static void setUp() {
24         db = new EmbeddedDatabaseBuilder()
25             .setType(EmbeddedDatabaseType.HSQL).addScript("db/sql/create-db.sql")
26             .addScript("db/sql/insert-data.sql").build();
27     }
28
29     @AfterClass
30     public static void tearDown() {
31         db.shutdown();
32     }
33
```

```
Markers  Properties  Servers  Data Source Explorer  Snippets  Console  JUnit  Git Staging
Finished after 1,692 seconds
Runs: 7/7    Errors: 0    Failures: 0
> ont.paarma.test.UserServiceTest [Runner: JUnit 4] (0,629 s)      Failure Trace
> ont.paarma.test.UserControllerTest [Runner: JUnit 4] (0,547 s)
> ont.paarma.test.UserDAOTest [Runner: JUnit 4] (0,043 s)
```

Kuva 24. UserDAOTest-luokan alustustoimenpiteet muutettiin ennen jokaista testiä suoritettavasta vain kerran suoritettavaksi @BeforeClass- ja @AfterClass -annotaatioilla. Todennettiin, että kaikki testit menivät läpi.

Ajettiin kaikki testit läpi todentaen, että ne näyttivät vihreää. Näin oli implementoitu toiminnallisuus asiakastilin luominen, joka toteutti käyttötapauksen Luo asiakastili MA01. Kaikille toiminnallisuuden toteuttavan ohjelmakoodin metodeille luotiin testivetoisen kehitysmällin mukaan ensin testi, sitten toteutus.



Kuva 25. Käyttäjätilin luonti

Käyttäjätilin luonti graafisen käyttöliittymän kautta tapahtuu demossa siten, että käyttäjä syöttää tietonsa lomakkeelle ja painaa nappia.

Kuva 26. Käyttäjätili luotu.

Kun käyttäjätili on luotu onnistuneesti, käyttäjä saa nähtäväkseen tiedot, joilla tili luotiin, ja voi edetä myyntipaikan varaukseen.

5.6 Muun toiminnallisuuden implementointi pähkinänkuoressa

Sovellusdemoon toteutettiin vielä seuraavat toiminnallisuudet testivetoista kehitysmenetelmää käyttäen:

- käyttäjätietojen muokkaus,
- uuden varauksen luonti,
- käyttäjän omien varausten katselu,
- käyttäjän omien varausten muokkaus ja
- käyttäjän omien varausten poisto.

Valmiin sovelluksen testikattavuutta tarkasteltiin Eclipsen lisäosalla EclEmma, jonka avulla voidaan yksityiskohtaisesti tarkastella, kuinka suuren osan ohjelmakoodista Junit-testit kattavat.

Markers | Properties | Servers | Data Source Explorer | Snippets | Console | JUnit | Git Staging | Coverage

ont.paarma.test (27.11.2016 20:57:27)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
paarma	91,9 %	2 461	216	2 677

Kuva 27. EclEmman tuottama testien kattavuusraportti

Lopputulokseksi saatiin 91,9%, joka ei ole hassumpi luku. Aivan kaikkia ohjelmakoodin osia ei saatu testattua. Rivimäärällisesti yksikkötesteihin liittyviä koodirivejä oli 870, kun taas itse ohjelmakoodia oli 810 riviä.

Valmiin sovelluksen ohjelmakoodi on julkisesti nähtävillä osoitteessa https://github.com/InkaH/paarma.

6 Pohdinta

Kiistatonta näyttöä TDD:n hyödyistä ei vaikuta olevan. Ammattikirjallisuudessa testivetoista kehitysmenetelmää pidetään hyödyllisenä ja tehokkaana, mutta tutkimusten perusteella näyttäisi olevan vaikeaa erottaa, mitkä hyödyistä ja haitoista liittyvät nimenomaan testivetoiseen kehitysmalliin eivätkä koituisi jälkikäteen kirjoitetuista yhtä kattavista testeistä aivan samalla.

Tutkimuksiin tutustuessa ilmeni, että aiheesta on hiukan vaikeaa tehdä tutkimusta, koska tutkitaan kertaluontoisia asioita eli ohjelmistoprojekteja. Samaa projektia ei voi tehdä sama henkilö kahdesti, ja kehittäjien ohjelmointi- ja yksikkötestaustaidot vaikuttavat suuresti lopputulokseen, mikä tekee lähes mahdottomaksi tilanteen, jossa voitaisiin verrata objektiivisesti toisiinsa saman projektin suorittamista TDD:n avulla ja ilman sitä.

TDD:n vaikutusta tuottavuuteen, mikä lienee olennaisin ominaisuus ohjelmistoprojektissa, oli vaikea tutkia ja siitä ei oltu saatu ristiriidattomia tuloksia. Tuottavuutta ei voida luotettavasti mitata ainoastaan kehittäjien ajankäytöllä itse kehitystyössä, vaan olisi kyettävä ottamaan huomioon projektin koko elinkaari mukaanlukien sovelluksen ylläpidon ja mahdollisen jatkokehityksen, joihin kattavat yksikkötestit vaikuttavat suuresti. Näin kattavaa tutkimusta on kuitenkin tehty vähän.

Itse kehittäjänä koin, että TDD:n ja ylipäätään yksikkötestauksen opettelu hidastaa kehitytyötä, aluksi paljonkin. Kehitystyöstä noin 3/4 kului testien kirjoittamiseen, kun kokemusta yksikkötestien kirjoittamisesta oli vähäisesti. Toisaalta kehitystyöhön käytettävä aika lyheni mielestäni selvästi, kun virheet on helppo paikallistaa ohjelmakoodin ollessa kattavasti testattu, ja ohjelmakoodin rakenne ja sovelluslogiikka oli jo testejä kirjoittaessa seljennyt, joten implementointi sujui nopeammin. Lisäksi koodin refaktorointi ja uusien toiminnallisuuksien implementointi onnistui huolettomammin, kun voi luottaa, että testit kertovat, jos tuli rikkoneeksi aiempia toiminnallisuuksia uusien myötä.

Työni perusteella olen sitä mieltä, että yksikkötestauksen hyvään hallintaan kannattaisi ehdottomasti panostaa, vaikka kehittäjältä meneekin viikkoja oppia sujuva testien kirjoittaminen, koska ohjelmavirheet ja etenkin niiden vaikea jäljitettävyys hidastavat kehitystyötä huomattavan paljon. Yksikkötestaamisen ymmärtäminen auttoi ainakin itseäni myös kirjoittamaan modulaarisempaa, selkeämpää ja laadukkaampaa koodia, kun ohjelmakoodin kirjoittamista tuli tarkasteltua sujuvan yksikkötestaamisen näkökulmasta. Testivetoisen kehitysmenetelmän käytön seurauksena testien kattavuus kehitetyssä sovelluksessa oli yli 90%, ja testaukseen liittyviä koodiriivejä oli 870, kun taas itse ohjelmakoodia oli 810 riviä.

Tämä tulos oli yhdenmukainen sen ammattikirjallisuudessa ja osassa tutkimuksia huomioidun seikan kanssa, että TDD-menetelmän avulla päästään yleensä korkeisiin testikattavuuslukuihin ja testejä tulee kirjoitettua määrällisesti enemmän. Tutkimuksiin tutustumisen ja oman työni perusteella en ole varauksetta sitä mieltä, että TDD itsessään olisi tae hyvin testatulle tai laadukkaalle ohjelmakoodille. Tärkeintä on hyvä ymmärrys oikeanlaisten yksikkötestien toteuttamisesta riippumatta siitä, kirjoitetaanko testit ennen ohjelmakoodia vai sen jälkeen. Pitäisin kuitenkin mahdollisena, että TDD-malliin sitoutuminen ohjelmistoprojektissa auttaisi ottamaan huomioon yksikkötestien kirjoitukseen kuluvan ajan projektin aikataulussa ja voisi vähentää näin yksikkötestaamisen laiminlyöntiä, kun kehittäjät kokisivat että aikaa testien kirjoittamiseen on riittävästi.

Aiheena TDD oli mielenkiintoinen ja ajankohtainen. Sovelluskehitystyössä yksikkötestaaminen on aina läsnä hieman ongelmallisena osa-alueena, jolle tuntuu olevan vaikeaa osoittaa riittävästi resursseja. TDD-mallin mukaiseen sovelluskehitykseen törmää silloin tällöin ja myös rekrytointi-ilmoituksissa se joskus vilahtaa avainsanana. Koin itselleni olevan suurta hyötyä siitä, että sain työtä tehdessäni tutustua aiheesta tehtyyn tutkimukseen hahmottaakseni, mitkä voisivat olla TDD-mallin vahvuudet tavalliseen, jälkikäteen testaamiseen nähden. Hyödyn varmasti urallani myös siitä, että opin kirjoittamaan yksikkötestejä Java-kielellä yleensä ja TDD-menetelmällä erityisesti. Lisäksi sovelluksen toteuttaminen syvensi Java- ja Spring-osaamistani. Teen mielelläni tulevaisuudessa kehitystyötä TDD-mallia hyödyntäen.

Työn lopputuloksena syntyi valmis demo kirpputorin ajanvarausjärjestelmästä. Demon ohjelmakoodi on julkisesti nähtävillä osoitteessa https://github.com/InkaH/paarma. Demo kattaa kuvassa 3 mainitut käyttötapaukset myyntipaikkojen varausstatuksen tarkastelua lukuunottamatta. Tarvittaessa demon pohjalta voidaan kehittää tuotantoon julkaistava kirpputorin varausjärjestelmä.

Lähteet

Acharya, S. 2014. Mastering Unit Testing Using Mockito and JUnit. Packt Publishing. Birmingham.

Aejmelaeus, W. 2009. Test-Driven Development. Arcada University of Applied Science. Helsinki.

Aniche, M. & Gerosa, M. 2015. Does test-driven development improve class design. A qualitative study on developers perceptions. Journal of the Brazilian Computer Society, 21, 15. URL: http://journal-bcs.springeropen.com/articles/10.1186/s13173-015-0034-z. Luettu: 7.5.2016.

Appel, F. 2015. Testing with JUnit. Master high-quality software development driven by unit tests. Packt Publishing. Birmingham.

Causevic, A., Punnekkat, S. & Sundmark, D. 2011. Factors Limiting Industrial Adoption of Test Driven Development. A Systematic Review. 2011 Fourth IEEE International Conference on Software Testing, Verification and Validation. s. 337-346. URL: http://www.ipr.mdh.se/pdf_publications/1993.pdf. Luettu: 7.5.2016.

Diep, M., Erdogmus, H., Layman, L., Melnik, G., Shull, F. & Turhan, B. 2010a. What Do We Know about Test-Driven Development. IEEE Software, 27, 6, s. 16-19. URL: https://www.academia.edu/20223369/What_Do_We_Know_about_Test-Driven_Development. Luettu: 7.5.2016.

Diep, M., Erdogmus, H., Layman, L., Melnik, G., Shull, F. & Turhan, B. 2010b. How Effective is Test-Driven Development. Teoksessa Oram, A. & Wilson, G. (toim.). Making Software. What Really Works and Why We Believe It, s. 207–219. O'Reilly. Kalifornia. URL: http://hakanerdogmus.net/weblog/wp-content/uploads/tdd-sr-book-chapter.pdf. Luettu: 7.5.2016.

Farcic, V. & Garcia, A. 2015. Test-Driven Java Development. Invoke TDD principles for end-to-end application development with Java. Packt Publishing. Birmingham.

Koskela, L. 2008. Test Driven – Practical TDD and Acceptance TDD for Java Developers. Manning. Greenwich.

Martin, R. 2007. Professionalism and Test-Driven Development. IEEE Software, 24, 3, s. 32-36. URL: http://pdfs.semanticscholar.org/2647/7e21edb19335fbae184092b739d1a6c06957.pdf. Luettu: 5.5.2016.

Martin, R. 2009. Clean Code. A Handbook of Agile Software Craftsmanship. Prentice Hall. New Jersey.

Münch, J. & Mäkinen, S. 2014. Effects of Test-Driven Development. A Comparative Analysis of Empirical Studies. Helsingin yliopisto. Helsinki. URL: http://tuhat.halvi.helsinki.fi/portal/files/29553974/2014_01_swqd_author_version.pdf. Luettu: 7.5.2016.

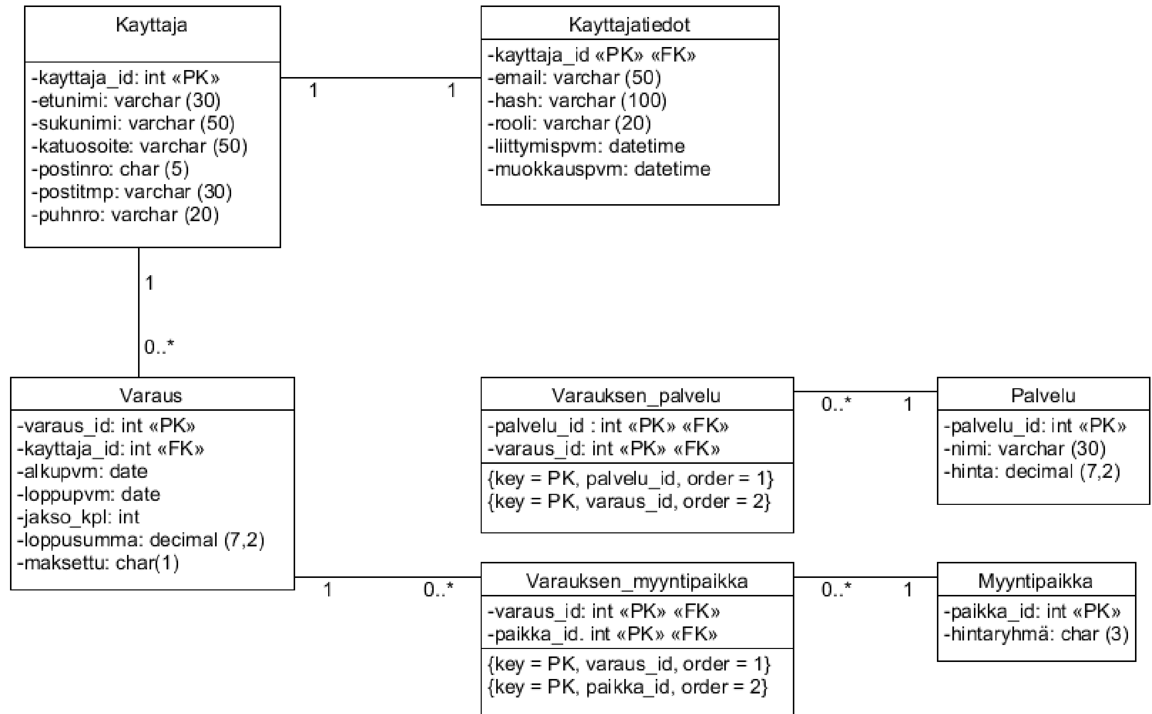
Samaroo, A. 2010. Life Cycles. Teoksessa Hambling, B. (toim.). Software Testing. An ISTQB-ISEB Foundation Guide, s. 34-56. BCS The Chartered Institute for IT. Swindon.

Liitteet

Liite 1. Testitapausten pohjana käytetyt skenaariot - esimerkki

Tapahtuman nimi	Asiakastilin luonti varauspalveluun
Käyttötapauksen nimi ja numero	Luo asiakastili MA01
Triggeri	Asiakas klikkaa websivuilla linkkiä, josta pääsee asiakastilin luontiin.
Ennakkoehdot	Asiakas on yrityksen websivuilla ja sivut toimivat normaalisti.
Interested Stakeholders	omistaja, lainsäädäntö, as. palv. henkilökunta, markkinointi, it-ylläpito, ulkopuoliset konsultit (tietoturva-asiantuntija)
Active Stakeholders	myyjäasiakas
Normaalitilanteen työvaiheet	1. Asiakas täyttää lomakkeeseen vaaditut asiakastiedot : etunimi, sukunimi, katuosoite, postinumero, postitoimipaikka, puhelinnumero ja sähköpostiosoite sekä haluamansa salasanan ja lähettää lomakkeen. 2. Järjestelmä vahvistaa asiakastilin luonnin ja lähettää asiakkaalle vahvistussähköpostin.
Vaihtoehtoiset työvaiheet	-
Poikkeukset	2.1 Lomaketietojen lähetys ei onnistu puutteellisen/väärän tiedon takia tai järjestelmän toimintahäiriön vuoksi. 2.2 Järjestelmä ilmoittaa, mikä tieto on puutteellinen, ja ohjeistaa sen kirjoittamisessa oikein, tai ilmoittaa toimintahäiriöstä ja ohjeistaa miten seuraavaksi toimitaan.
Lopputulos	Asiakas on luonut itselleen asiakastilin yrityksen varausjärjestelmään tarvittavilla tiedoilla, ja voi nyt kirjautua sisään varausjärjestelmään.

Liite 2. Tietokannan ER-kaavio



Liite 3. Graafisen käyttöliittymän prototyyppi

Kirjaudu sisään

Sähköposti: email@mail.com

Salasana: *************

Unohdin salasanani...

Luo uusi käyttäjätili... 1

Peruuta Kirjaudu

1 Luo uusi käyttäjätili... –dialogi

Uuden käyttäjän rekisteröinti

Etunimi | Sukunimi

Katuosoite | Postinumero

Postitoimipaikka | Puhelinnumero

Sähköpostiosoite

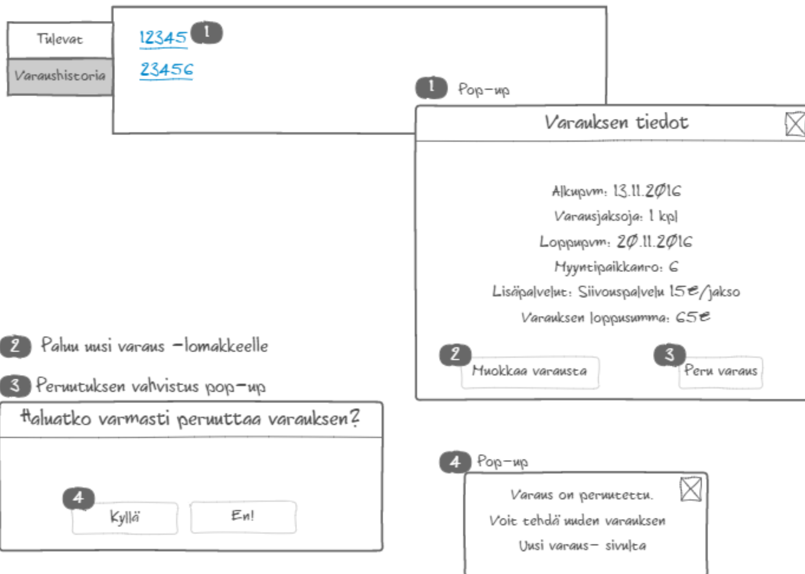
Salasana | Vahvista salasana

☑ Haluan uutisia ja tarjouksia sähköpostiini!

Peruuta Rekisteröidy

Kirva
Uusi varaus | Omat varaukset | Käyttäjätiedot | Ohje
Hei, [käyttäjä]
Uusi varaus
Alkupvm 13.11.2016 1
Varausjaksoja 1 kpl varausjakso on aina 7 päivää
Loppupvm 20.11.2016
Myyntipaikkanro 6 Katso myyntipaikkakartta
Lisää myyntipaikka..
Lisäpalvelut Siivouspalvelu 15€/jakso
Lisää palvelu..
Varauksen loppusumma: 65 €
Kampanjakoodi
Tyhjennä
Tee varaus 2
1 Pvm-valikosta avautuu kalenteri
13.11.2016
<< Marraskuu 2016 >>
Ma Ti Ke To Pe La Su
1 2 3 4 5
6 7 8 9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30
2 Varausvahvistus-ponnahdusikkuna
Vahvista varaus
Alkupvm: 13.11.2016
Varausjaksoja: 1 kpl
Loppupvm: 20.11.2016
Myyntipaikkanro: 6
Lisäpalvelut: Siivouspalvelu 15€/jakso
Varauksen loppusumma: 65€
Vahvistetaanko varaus?
Peruuta
Vahvista

Kirva
Uusi varaus | Omat varaukset | Käyttäjätiedot | Ohje
Hei, [käyttäjä]


Tulevat
Varaushistoria
12345 1
23456
1 Pop-up
Varauksen tiedot
Alkupvm: 13.11.2016
Varausjaksoja: 1 kpl
Loppupvm: 20.11.2016
Myyntipaikkanro: 6
Lisäpalvelut: Siivouspalvelu 15€/jakso
Varauksen loppusumma: 65€
2 Muokkaa varausta
3 Peru varaus
2 Paluu uusi varaus -lomakkeelle
3 Peruutuksen vahvistus pop-up
Haluatko varmasti peruuttaa varauksen?
4 Kyllä
En!
4 Pop-up
Varaus on peruutettu.
Voit tehdä uuden varauksen
Uusi varaus- sivulta

Käyttäjätiedot

Muokkaa käyttäjätietoja

Kalle

Käyttäjä

Katu A 1 C

00330

Kaupunki X

1234567890

kalle@email.com

Vaihda salasanaa:

uusi salasana

uusi salasana uudelleen

Haluan uutisia ja tarjouksia sähköpostiini!

Peruuta

Tallenna