

Simo Kärkinen

# Käyttätymislähtöinen ohjelmistokehitys (BDD)

Metropolia Ammattikorkeakoulu  
Insinööri (AMK)  
Tietotekniikan koulutusohjelma  
Insinööri  
29.4.2013

Tekijä(t) Otsikko	Simo Kärkinen Käyttäytymislähtöinen ohjelmistokehitys
Sivumäärä Aika	35 sivua, ei liitteitä 22.4.2013
Tutkinto	Insinööri (AMK)
Koulutusohjelma	Tietotekniikka
Suuntautumisvaihtoehto	Ohjelmistotekniikka
Ohjaaja(t)	Simo Silander, lehtori Markku Luotamo, ICT-arkkitehti
<p>Insinöörityössä oli tavoitteena toteuttaa työnantajalleni uusi web-pohjainen käyttöliittymä-sovellus PKI-varmenteiden ja niiden käytön hallintaan. Sovelluksen salaisuudesta johtuen insinöörityö on kirjoitettu kehitysmenetelmän näkökulmasta ja itse sovelluslogiikkaa ei kuvata juuri ollenkaan.</p> <p>Kehitysmenetelmän tuotteena syntyi kattava joukko automaattisesti suoritettavia testitapauksia, jotka ajetaan automaattisesti kehitysympäristössä pyörivää sovellusta vasten.</p> <p>Käyttöliittymäsovellus toteutettiin Java-ohjelmointikielellä käyttäen Spring-sovelluskehystä ja automaattitestien tekeminen käyttäen JBehave-ohjelmointikehystä.</p> <p>Tuotetta ja kaikkia sen käyttöliittymätestejä ei saatu tämän opinnäytetyön puitteissa valmiiksi vaan niiden kehitys jatkuu edelleen.</p>	
Avainsanat	BDD, käyttäytymislähtöinen kehitys, JBehave

Author(s) Title	Simo Kärkinen Behavior-driven Development
Number of Pages Date	35 pages, no appendices 22 April 2013
Degree	Bachelor of Engineering
Degree Programme	Information technology
Specialisation option	Software Engineering
Instructors	Simo Silander, Senior Lecturer Markku Luotamo, ICT-architect
<p>The purpose of this Bachelor's thesis was to develop a new web-based graphical user interface for managing PKI-certificates and its usage. Because of products confidentiality the study was written from development process perspective, so its application logic is not described.</p> <p>The result of this development process was a comprehensive set of test cases which can be run automatically in the developing environment.</p> <p>The GUI was implemented using the Java programming language and the Spring application development framework and automatic test cases was created using JBehave framework.</p> <p>The product and its test cases are not completed yet but the promotion continues.</p>	
Keywords	BDD, behavior-driven development, JBehave

# Sisälllys

## Lyhenteet

1	Johdanto	1
2	Ohjelmistokehitys ja -menetelmät	2
2.1	Vesiputousmalli	2
2.2	Muut perinteiset kehitysmenetelmät	4
2.3	Ketterät kehitysmenetelmät	4
3	Testauspainotteiset kehitysmenetelmät	6
3.1	TDD	8
3.2	BDD	9
4	BDD käytännössä	10
4.1	JBehave framework	10
4.2	Käyttötapauksesta testitapauksiksi	11
4.3	JBehave Spring-projektissa	11
4.4	Testien tekeminen	14
4.4.1	Tarinan luominen	14
4.4.2	Tarinoiden kytkeminen Javaan	16
4.5	Miten helpottaa testien tekemistä	19
4.5.1	Konfigurointi	20
4.5.2	Testien ajaminen	21
4.5.3	Tuloksen tarkastelu	21
4.6	Ongelmia	22
5	Käyttöliittymätestien automaatio	23
5.1	Jenkins	23
5.2	CI-ympäristön käyttöönotto	24
5.3	Testien suorituksen analysointi	27
5.4	Automatisoinnin haasteita	28
6	Menetelmän analysointi	29
6.1	Työmäärä	29

6.2	Saadaan mitä halutaan	30
6.3	Toimivaa koodia	30
7	Työn arviointia	30
8	Yhteenveto	32
	Lähteet	33

## Lyhenteet

BDD	Behavior-driven development. Käyttäytymislähtöinen ohjelmistokehitysmenetelmä.
TDD	Test-driven development. Testauslähtöinen ohjelmistokehitysmenetelmä.
regressio	Toistokierros. Kun ohjelmaan tehdään muutos, joudutaan se regressiotes- taamaan, jotta voidaan todeta, etteivät tehdyt korjaukset ole rikkoneet muuta ohjelmaa.
WebDriver	Kirjasto, jonka avulla ohjelmakoodista voidaan ohjata selainta.
CI	Continuous Integration, jatkuva integraatio.
PKI	Public Key Infrastructure, julkisen avaimen infrastruktuuri.
GUI	Graphical User Interface, graafinen käyttöliittymä.
Maven	Apache Software Foundationin kehittämä ohjelmien koontityökalu.
UP	Unified Process, yhtenäistetty prosessi. Ohjelmistokehityksen prosessike- hys.
sprint	Ketterän ohjelmistokehityksen kehityssykli.
scrumtiimi	Ketterän kehitysprojektin jäsenet.
scrummaster	Ketterän kehitystiimin vetäjä.
jobi	Jatkuvan integroinnin ympäristössä oleva rutiini, joka voidaan asettaa tekemään halutut asiat halutuin väliajoin.

## 1 Johdanto

Tässä opinnäytetyössä tutustutaan uuteen ketterään ohjelmistokehitysmenetelmään, käyttäytymislähtöiseen kehittämiseen eli BDD:hen. Erilaisia kehitysmenetelmiä ohjelmistotuotannossa on jo useita, mutta edelleen on kysyntää uusille ja tiettyyn tarkoitukseen paremmin soveltuville menetelmille.

Ohjelmistot monimutkaistuvat jatkuvasti johtuen muun muassa ominaisuuksien määrän lisääntymisestä sekä tiedon ja palvelujen jatkuvasta kasvusta, jolloin niiden kehittäminen ja ylläpito vaatii entistä enemmän henkilöresursseja [1]. Lisäksi ympäröivä maailma muuttuu niin nopeasti, että tänään määriteltävälle tuotteelle ei välttämättä ole tuotantoon ehtiessä enää käyttöä. Kehitysvaiheen ongelmana voi olla myös se, että ohjelmiston tilaaja, määrittelijä, toteuttaja ja testaaja eivät puhu samaa kieltä, jolloin lopputulos ei välttämättä ole sille asetetun tarpeen mukainen. Työssä esiteltävä menetelmä tuo helpotuksia edellä kuvattuihin ongelmiin.

Aihetta lähestytään perinteisten ja ketterien kehitysmenetelmien kautta siirtyen testauspainotteisiin kehitysmenetelmiin, jolloin ymmärretään paremmin, miksi uudelle menetelmälle on tarvetta. Työn yhteydessä toteutetaan myös käytännön esimerkkejä automaattisten käyttöliittymätestien tekemisestä, joilla on suuri rooli koko kehitysmenetelmää ajatellen. Sen jälkeen tarkastellaan, miten saadaan nämä automaattitestit osaksi jatkuvaa integraatiota. Lopuksi arvioidaan hieman menetelmää yleensä.

Automaattitesteillä testataan työprojektina tekemääni PKI-varmenteiden hallintaan tarkoitettua käyttöliittymää.

## 2 Ohjelmistokehitys ja -menetelmät

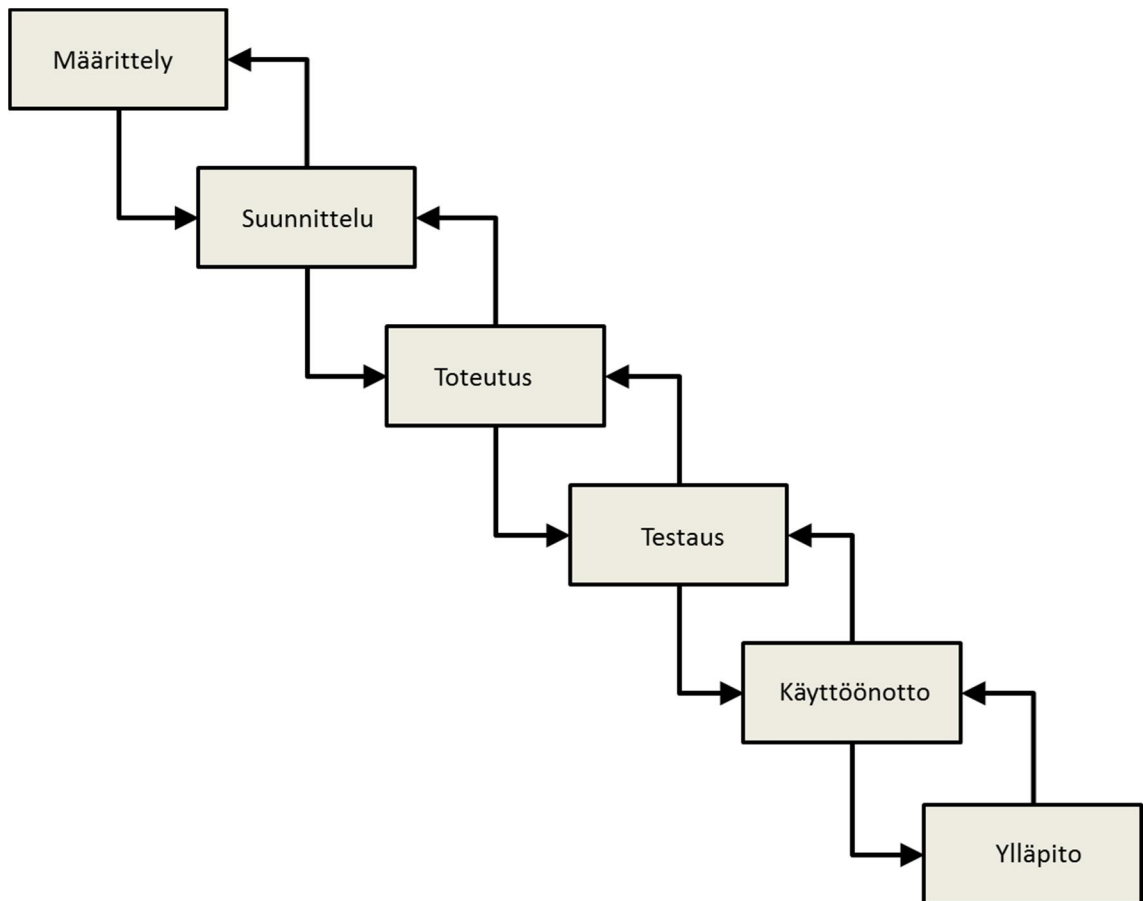
Ohjelmistokehityksessä tai -tuotannossa on kyse tietokoneohjelmien tuottamisesta. Koska toistettavia asioita voidaan suorittaa ohjelmallisesti huomattavasti nopeammin kuin ihmisen tekemänä, usein ohjelman tilaajana on asiakas, joka haluaa sen tehostamaan jotakin toimintaansa. [2.]

Uuden ohjelman tuottaminen on yleensä haastava prosessi. Niinpä ilman suunniteltuja toimintatapoja onkin miltei mahdoton tuottaa tehokkaasti hyvää lopputuotetta. Kehitysmenetelmien puute 1960-luvulla johti kriisiin, koska ohjelmien tekeminen oli tehotonta, järjestelmät sisälsivät paljon ohjelmointivirheitä ja kustannukset karkasivat käsistä. Tämä johti siihen, että ohjelmistoissa alkoi ilmetä paljon virheitä, eivätkä ne enää auttaneet siihen tarpeeseen, mihin ne oli suunniteltu. Niinpä nähtiin tarpeelliseksi alkaa tutkia, miten tuottaa tehokkaasti ohjelmistoja, joissa laatu olisi mahdollisimman korkea.

Alettiin kehittää erilaisia kehitysmenetelmiä. Monisäikeisyytensä vuoksi näytti siltä, että kehitysprosessi kannattaisi jakaa osiin niin, että ensin tehdään yksi osa valmiiksi asti ja vasta sen jälkeen edetään seuraavaan vaiheeseen. Tätä ositusta kutsutaan vaihejakomalliksi. Sen perusajatusta voinee pitää onnistuneena, koska vielä nykyäänkin eri kehitysmenetelmät ovat vahvasti osiin jaettuja. [2.]

### 2.1 Vesiputousmalli

Perinteisessä vesiputousmallissa kehitysprosessi etenee vesiputouksen tapaan vaiheesta toiseen, tasolta seuraavalle (kuvio 1). Kussakin vaiheessa ollaan niin kauan, kunnes todetaan, että se on valmis.



Kuvio 1. Vesiputousmallin etenemisperiaate

Alussa asiakkaalla on vaatimukset, tai ne tehdään yhdessä asiakkaan ja kehitystä tekevän tahon kanssa. Niiden pohjalta osataan määritellä lopputuote halutunlaiseksi. Sen jälkeen suunnitellaan, kuinka tuote voidaan toteuttaa. Kun kaikki on lyöty lukkoon, voidaan aloittaa itse toteutusvaihe. Toteutusvaiheen ollessa valmis, voidaan siirtyä testausvaiheeseen, sitten käyttöönottoon ja ylläpitoon. Kunkin valmiin vaiheen tuotoksena on tietyt dokumentit, joiden perusteella seuraava vaihe voidaan suorittaa.

Laajoissa kokonaisuuksissa ja yksittäisissä isoissa tuotteissa vesiputousmalli voi olla edelleenkin varsin käyttökelpoinen tapa toteuttaa ohjelmistoja. Sen suurin heikkous kuitenkin on se, että alussa pitäisi heti tietää tarkasti, minkälainen lopputuotteen pitää olla. Käytännössä kuitenkin asiakkaiden tuotteelle asettamat vaatimukset muuttuvat, ja menetelmä on hyvin kankea tällaisessa tilanteessa. Kuten johdannossa jo mainittiin, pitkän kehitysprosessin vuoksi voi käydä niin, että asiakkaan ohjelmistolle asettamiin vaatimuksiin on tullut jo muutoksia tai kenties tarvetta koko tuotteelle ei enää ole.

## 2.2 Muut perinteiset kehitysmenetelmät

Vesiputousmallin lisäksi on joukko muita perinteisiä kehitysmalleja kuten prototyyppimalli, yhtenäistetty prosessi ja suihkulähdemalli. Prototyyppimallissa tuotetaan ensin lopputuotteen ulkoasu ja asiakkaan hyväksyttyä aletaan spiraalimaisesti laajentaa toiminnallisuutta niin, että toteutetaan liiketoiminnalliset ominaisuudet ja muu syvemmän tason logiikka. Tietokantakerros tehdään viimeisenä. Menetelmän hyvänä puolena on se, että hyvin varhaisessa vaiheessa asiakas näkee, mitä on tulossa, ja samalla paljastuvat mahdolliset väärinkäsitykset.

Unified Process, UP eli yhtenäistetty prosessi on iteratiivinen ja inkrementaalinen ohjelmistokehityksen prosessikehys. Se ei itsessään ole valmis prosessi vaan kehys, jota tulee muokata tarpeiden ja organisaation mukaan. UP:n tunnetuin, dokumentoitu versio on Rational Software Corporationin kehittämä RUP, Rational Unified Process. [3.]

Suihkulähdemalli on samankaltainen kuin vesiputousmalli, mutta siinä asiat tehdään päinvastoin, eli ensin ohjelmoidaan tuote, jonka jälkeen tuotetaan dokumentit [4].

## 2.3 Ketterät kehitysmenetelmät

Siinä missä vesiputousmallissa projektin alussa on määritelty koko lopputuote, ja jokainen vaihe tehdään alusta loppuun, pyritään ketterässä kehittämisessä (agile software development) paloittelemaan työ pieniin osiin, jonka jälkeen näissä pienissä osissa sovelletaan vesiputousmallin vaiheita. Nämä osat eli "sprintit" kestävät tyypillisesti yhdestä neljään viikkoa. Kullekin sprintille suunnitellaan yhdessä tuotteen omistajan kanssa, mitä uusia ominaisuuksia otetaan työn alle ko. sprintissä. Tämän jälkeen ne määritellään, toteutetaan ja testataan. Jokaisen sprintin lopuksi pyritään siihen, että työn alle otetut ominaisuudet ovat siinä kunnossa, että ne voidaan julkaista. Yleensä ensimmäisten sprintien jälkeen ei ole järkevää julkaista siihen mennessä tehtyjä ominaisuuksia, mutta niiden tulisi kuitenkin olla julkaisukelpoisia. Omistajan tulee valita työn alle otettavat ominaisuudet sen mukaan, että kriittisimmät ominaisuudet toteutetaan ensin ja lopussa on siten pienemmän prioriteetin ominaisuudet. Tämä siksi, että omistaja voi tarvittaessa päättää kehityksen lopettamisesta esimerkiksi aikataulun venyessä tai budjetin ylittyessä, ja silti tuote toimii kriittisimmiltä osin. [5; 6.]

Ketterässä kehityksessä on tyypillistä, että dokumentteja ei tuoteta, ellei niitä tarvita, vaan projektin jäsenien kesken tieto liikkuu pääasiassa kasvotusten. Onkin tavallista, että ketterä tiimi istuu yhdessä tilassa, jolloin keskusteleminen on helppoa ja kaikki projektin jäsenet ovat koko ajan tilanteen tasalla. Ketterissä menetelmissä korostetaan toimivan ohjelmiston merkitystä, eikä niinkään tuotettujen dokumenttien määrää. Tästä ja yleisestä harhaluulosta huolimatta ketterä ohjelmistokehitys ei ole, tai ei tule olla kuritonta, suunnittelematonta "koodailua".[5.]

Ketteriä kehitysmenetelmiä on useita. Tutuin ja yleisimmin käytetty niistä lienee Scrum [7]. Muita ketteriä menetelmiä on mm. Extreme programming (XP), DSDM eli Dynamic Systems Development Method, Crystal Methods, Feature Driven Development, Lean Software Development ja Kanban Software Development [8].

Scrumista puhuttaessa tarkoitetaan yleensä ohjelmistokehitysprosessia, mutta se soveltuu myös yleisemmin projektinhallintaan. Scrumin idea onkin alunperin vuonna 1986 kuvattu lähestymistapana tuotekehitykseen yleensä. Menetelmässä on ajatus, että yksi monitaitoinen ryhmä, scrum-tiimi, suorittaa koko kehitysprojektin alusta loppuun. Tarkoituksena on olla tilanteisiin sopeutuva, itsenäinen ja nopeasti muutoksiin reagoiva projektiryhmä. [7; 9.]

Scrum-tiimi on siis itsenäinen ryhmä, joka päättää itse omista toimistaan. Tiimissä on kolmenlaisia rooleja: tuoteomistaja, scrummaster ja kehitystiimi. Tuoteomistajan tehtävänä on maksimoida kehitystyön arvo, joten hän varmistaa, että scrum-tiimi toteuttaa kussakin sprintissä sellaisia vaatimuksia, jotka ovat tuotteen valmistumisen kannalta kriittisimpiä. Kehitystiimi tekee työmääräarviot, joiden mukaan sprintin tehtävälistalle otetaan toteuttavia ominaisuuksia. Tuoteomistajan tehtävänä on liiketoiminta-alueesta ymmärtävänä varmistaa, että scrum-tiimi ymmärtää, mistä tuotteesta on kyse. Kehitystiimin tehtävänä on hoitaa sprintiin valitut ominaisuudet toimivaksi ohjelmaksi. Vaikka kehitystiimin jäsenillä voi olla jotain erityisiä osaamisia tai joku on erikoistunut tiettyyn alueeseen, on koko kehitystiimi vastuussa ohjelman tilasta ja kehityksestä. Scrummasterin tehtävä on huolehtia, että tiimi pystyy tekemään töitään optimaalisella tavalla. Mahdollisista esteistä tai hidasteista tiimiläiset raportoivat scrummasterille. Scrummasterilla ei ole määräysvaltaa muuhun tiimiin nähden, mutta hän vastaa siitä, että työtä tehdään scrumin mukaisesti. [9.]

Scrum-projekti alkaa siten, että visioidaan yhdessä projektin tavoitteita ja miksi se tehdään. Visioinnin jälkeen muodostetaan työlista (Product Backlog) tuotteen tarvitsemista ominaisuuksista, jonka tuotteen omistaja priorisoi. Tämä priorisoitu lista toimii kehitysjonona, josta valitaan järjestyksessä kuhunkin sprinttiin ominaisuuksia prioriteettinsa mukaan ja niitä pyritään ottamaan vain sen verran, että ne ehditään tekemään sprintin aikana. Sprintin aikana vaatimuksia ei saa muuttaa, vaan tiimillä tulee olla työrauha ja sillä on oikeus työskennellä parhaaksi katsomallaan tavalla tavoitteiden saavuttamiseksi. Sprintin aikana työn edistyminen näkyy edistymiskäyrässä (Burndown Chart), joka kuvaa sprintissä jäljellä olevaa työmäärää ajan funktiona. Lisäksi voi olla myös käyrä, joka kuvaa samaa asiaa koko projektin tasolla.

Sprintin aikana tiimi kokoontuu joka päivä lyhyeen, alle viidentoista minuutin päiväpalaveriin, joka pidetään seisten ajan venähtämisen ehkäisemiseksi. Tähän päiväpalaveriin osallistuvat kaikki tiimin jäsenet. Jos tiimin jäsenillä olisi keskusteltavaa enemmän, kuin päiväpalaverin puitteissa ehtii, tulee asianosaisten sopia erillinen palaveri, joka voi jatkua vaikka välittömästi päiväpalaverin jälkeen neuvotteluhuoneessa tai muussa siihen soveltuvassa paikassa.

Kunkin sprintin lopussa järjestetään sprinttikatselmus, jossa scrum-tiimi yhdessä sidosryhmien kanssa selvittää, mitä sprintissä on saatu aikaan ja tarvittaessa sopeuttaa kehitysjonoa. Katselmuksessa kehitystiimi esittää tuotedemon, jonka pohjalta keskustellaan seuraavaksi työnalle otettavista ominaisuuksista. Tämä keskustelu voi toimia pohjustuksena seuraavalle sprintin suunnittelupalaverille, mutta tässä tilaisuudessa ei ole tarkoitus vielä lyödä lukkoon seuraavan sprintin tehtäviä.

### 3 Testauspainotteiset kehitysmenetelmät

Ohjelmistojen monimutkaistuessa joudutaan etsimään jatkuvasti erilaisia ja erityyppisiin projekteihin sopivia kehitysmenetelmiä. Toiset ohjelmat ovat kriittisempiä kuin toiset, jolloin niiden toimintavarmuuteen ja virheettömyyteen täytyy panostaa suuremmat resurssit ja käyttää menetelmiä, jotka pienentävät riskiä huolimattomuusvirheille ja muutenkin pyrkivät lähestymään tuotettavaa ohjelmaa eri näkökulmista.

Testausta painottavia kehitysmenetelmiä käytetään usein jonkun ketterän kehitysmenetelmän rinnalla. Tällaisia tunnettuja menetelmiä ovat testauslähtöinen kehitys TDD ja käyttäytymislähtöinen kehitys BDD. Nimestään - ja siitä, että testauslähtöisen ohjelmistokehityksen sivutuotteena syntyy joukko käyttökelpoisia testitapauksia - huolimatta ne eivät ole testausmenetelmiä vaan suunnittelumenetelmiä, jotka kuuluvat ohjelmiston toteuttajalle, vaikkakin BDD:ssä on kehittäjän syytä tehdä yhteistyötä myös testaa- jien kanssa. [10.]

Perinteisemmällä tavalla ohjelmoitaessa kehityssykli menee tyypillisesti niin, että ensin suunnitellaan, sitten toteutetaan ja tämän jälkeen testataan. Testauspainotteisissa kehitysmenetelmissä toimitaankin päinvastoin. Ensin tehdään testit, olivat ne sitten yksikötestejä tai vaikka automatisoituja hyväksymistestejä, joihin perehdytään tässä työssä myöhemmin.



Kuvio 2. Perinteinen ohjelmistokehityssykli

Kuvion 2 mukaisesti perinteisellä tavalla tehden ensin suunnitellaan, miten haluttu toiminto, luokka, metodi tai muu sellainen kannattaa toteuttaa. Sitten toteutetaan se ja lopuksi testataan, että tuotettu koodi tekee sitä, mitä sen halutaan tekevän.

Testiohjatuissa menetelmissä suunnitellaan ensin testit (kuvio 3), joilla voidaan todentaa, että ohjelma tai ohjelman osa toimii halutulla tavalla.



Kuvio 3. Testausohjattu kehityssykli

Vasta testien luomisen jälkeen itse ohjelmakoodi tuotetaan ja kehitystä tehdään, kunnes kaikki testit menevät läpi. Kun kaikki testit on läpäisty, voidaan ikään kuin suunnitella koodi uudelleen eli refaktoroida koodia muokkaamalla koodin sisäistä rakennetta

muokkaamatta ulospäin näkyvää toiminnallisuutta, jotta koodista saadaan mahdollisimman selkeää ja tehokasta. [11.]

### 3.1 TDD

Nykyisen TDD:n (Test-driven Development) juuret löytyvät Extreme Programming (XP) -metodologian määritelmistä. Erillään XP:stä se esitettiin ensimmäisen kerran Kent Beckin kirjassa *Test-Driven Development By Example* (Addison Wesley, 2003). TDD:n mukaista kehitystä on kuitenkin tehty jo paljon kauemmin, ja nykyään TDD on saanut paljon huomiota myös muiden kuin XP:n harrastajien parissa. [11; 12.]

TDD:stä on olemassa erilaisia variaatioita ja näin siitä puhuessa tulee asianosaisille selvittää, mitä muotoa TDD:stä tarkoitetaan. Roy Osherov sanookin TDD:tä esiintyvän neljässä muodossa [11].

- Testiohjattu kehitys: Testit kirjoitetaan ennen itse sovelluskoodia. Noudatetaan voimassaolevaa arkkitehtuuria.
- Testisuuntautunut kehitys: Testausta painotetaan ja testejä kirjoitetaan paljon, mutta ei välttämättä ennen sovelluskoodia.
- Testiohjattu suunnittelu: TDD toimii kokonaisvaltaisena suunnittelutyökaluna. Testit ohjaavat suunnitteluprosessia ja sovellusarkkitehtuuri tulee lennossa testien ohjaamana.
- Testiohjattu kehitys ja suunnittelu: Suurten linjojen arkkitehtuuri on valmiiksi suunniteltu, mutta TDD ohjaa pienemmän tason arkkitehtuuria.

TDD:ssä ideana on tehdä ensin yksikkötestit, jotka testaavat mahdollisimman kattavasti toiminnot, mitä tuotettavan koodin täytyy tehdä. Yksikkötesteillä voidaan testata mitä tahansa koodia. Tavallisesti testauksen kohteena ovat luokat ja niiden metodit. Yksittäisen metodin testauksessa testataan erilaiset syötteet ja tarkistetaan, että metodi toimii oikein kussakin tapauksessa, eikä esimerkiksi kaadu käyttäjän syöttäessä virheellisiä syötteitä. Erilaisilla työkaluilla voi analysoida tuotettua koodia ja sen testikattavuutta, jos näin syvälle haluaa mennä. Kun on tehty testit, joiden perusteella koodin voi-

daan ajatella toimivan halutulla tavalla, tuotetaan itse ohjelmakoodi. Tämän jälkeen koodataan, kunnes kaikki testit ovat menneet läpi. Viimeisessä vaiheessa, kun testit on läpäisty, niin sen jälkeen on syytä vielä katsoa, että koodi on selkeää ja että siinä ei tehdä turhia asioita. Testien avulla varmistetaan se, että koodi ei siistittäessä rikkoudu. Perinteisemmässä kehitysmenetelmässä on vaarana se, että kun koodi on saatu kerran toimivaksi, ei sitä enää uskalleta muuttaa, koska pelätään sen rikkovan jotain muuta.

### 3.2 BDD

BDD (Behavior-driven Development) on jatkumoa aiemmin kehitetylle TDD-kehitysmenetelmälle. Kummatkin edellä mainitut kehitysmenetelmät sopivat hyvin ketteriin ohjelmistokehitysprojekteihin. Kehitysmenetelmä on vasta vuonna 2006 saanut ensiaskeleensa [13]. Sen luojana pidetään Dan Northia ja sitä alettiin kehittää TDD:n yhteydessä ilmenneiden kysymysten vuoksi. Kysymyksiä aiheutui muun muassa siitä, että mistä aloittaa, mitä testata ja mitä jättää testaamatta, kuinka paljon testata ja niin edelleen. [14.]

BDD:n ero TDD:hen nähden ei ole kovin suuri, mutta siinä, kun TDD:ssä suunnitellaan testit, jotka testaavat koodin toimintaa, BDD:ssä testataan ohjelman toimintaa käyttäjälle tarkoitetun käyttöliittymän kautta.

BDD:ssä voidaan siis toimia niin, että kun kaikki suunnitellut testit menevät hyväksytysti läpi, voidaan ajatella ohjelman toimivan kuten pitääkin. Menetelmän pääajatus on, että määrittelijä, toteuttaja ja testaaja yhdessä suunnittelevat nämä automatisoitavat testit selkokieelisesti kunkin käyttötapauksen kohdalta. Myöhemmin toteuttaja rakentaa sopivat välineet, jotta nämä selkokieiset testitapaukset osaisivat ohjata ohjelman suoritusta halutulla tavalla.

Näillä käyttöliittymän kautta ohjelmaa testaavilla testeillä on kaksi suurta hyötyä. Nykyaian ohjelmistojen monimutkaisuus ja nopeat muutostarpeet aiheuttavat paljon testaustyötä, mikäli jokaisen muutoksen jälkeen koko ohjelma joudutaan testaamaan uudelleen. Ei ole järkeä, että monimutkaisen ohjelmiston testaamiseen osoitettu testausammattilainen testaa aina uudelleen ja uudelleen samoja perustoiminnallisuuksia, vaan testaajan resurssit on kyettävä osoittamaan todellisten ongelmakohtien etsimiseen. Lisäksi kone tekee tämän uudelleentestauksen eli regressiotestauksen paljon

varmemmin kuin ihminen. Kone tekee testauksen aina juuri sillä tavalla, kuin se on ohjelmoitu tekemään, mutta ihminen ei välttämättä aina muista tehdä kaikkea tai kaikkia asioita samalla tavalla. Regressiotestauksen työkuorman keventämiseen on ollut jo pidemmän aikaa olemassa työkaluja, joilla voidaan nauhoittaa testejä käyttöliittymään ja testit ajettua uudelleen, kun ohjemaan tulee muutoksia. Ongelma on kuitenkin se, että testiautomaatio edellyttää testaajalta kohtuullista ohjelmointiosaamista ja muutenkin siinä aiheutuu viivettä, jos odotellaan, että testaajat ajavat testit esimerkiksi järjestelmätestiympäristössä. Tyypillisesti kehitystyötä tehdään kuitenkin eri ympäristössä, kuin järjestelmätestaus, jota on tarkoituksenmukaista tehdä hieman stabiilimmassa ympäristössä. Jos sen sijaan ohjelmoijalla olisi tehtynä kattava testipatteri, joka ajetaan aina, kun ohjelmasta julkaistaan uusi versio joko paikallisesti kehittäjän työasemalle tai kehitysympäristöön, saadaan heti tieto siitä, jos tehty muutos aiheutti käyttöliittymän toimintaan virheitä.

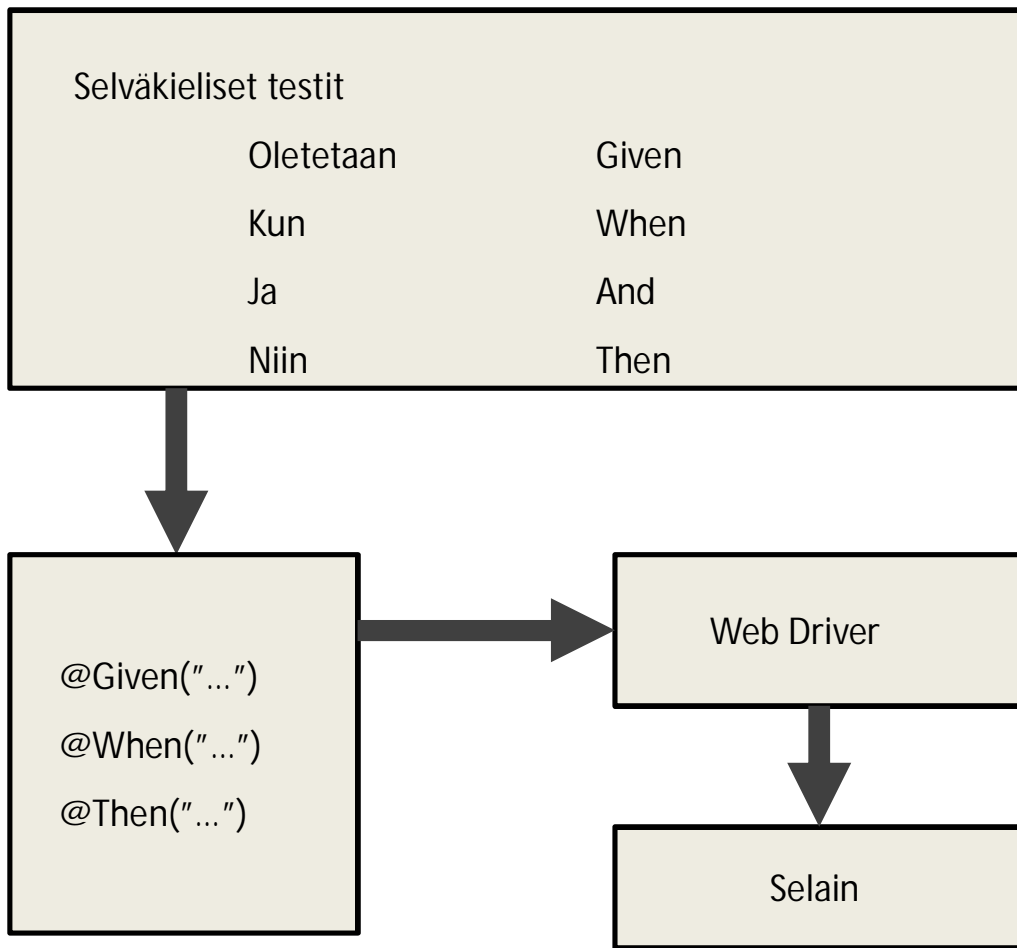
Suurin hyöty BDD-testeistä on kuitenkin ohjelman ylläpitovaiheessa, jolloin ylläpitäjänä ei välttämättä ole sama henkilö kuin sen toteuttanut henkilö. Jos sovelluksesta ei ole olemassa minkäänlaista automaattitestipatteria, on muutosten tekeminen erittäin vaivalloista, kun ei voi tietää, miten moneen paikkaan yksi asia voi vaikuttaa. Jos kehittäjällä on käytössään kattavat testit, hänen ei tarvitse pelätä niin paljon muutoksien tekemistä ja täten aikaa säästyy, kun ei tarvitse välttämättä ensin perehtyä jokaiseen ohjelman osaan.

## 4 BDD käytännössä

Käyttäytymislähtöinen ohjelmistokehitysmenetelmä on menetelmä, jossa pääpaino on ohjelmiston käyttäytymisellä. Eli ohjelmiston kehittäminen nojaa testeihin, joita voidaan ajaa käsin tai automaattisesti esimerkiksi jatkuvan integroinnin ympäristössä.

### 4.1 JBehave framework

JBehave on avoimen lähdekoodin Java-pohjainen ohjelmistokehys BDD-testien tekemiseen. Sen avulla voidaan selkokielliset, myös ei-teknisten ihmisten ymmärtämät testit kytkeä Java-koodiin, jolla voidaan ohjata käyttöliittymää web driverin kautta. Seuraava kuvio selventää kytköksiä.



Kuvio 4. Periaatekuva selväkielisten testiaskeleiden päätyemisestä käyttöliittymälle

#### 4.2 Käyttötapauksesta testitapauksiksi

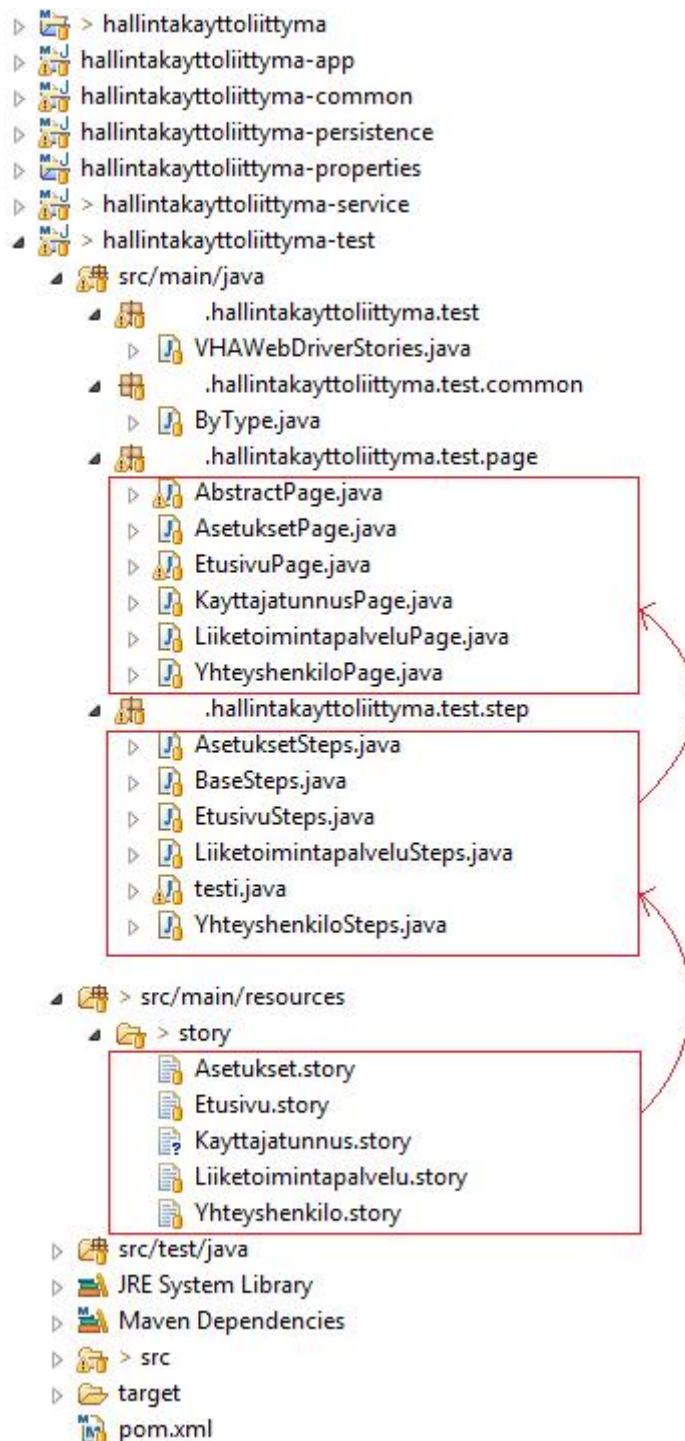
Jos tuotettavasta sovelluksesta on olemassa käyttötapaukset, testitapausten eli testiskenaarioiden luominen on suhteellisen suoraviivaista. Karkeasti ottaen esiehdosta muovaillaan "Oletetaan"-rivi, joka alustaa ohjelman haluttuun tilaan. Käyttäjän tekemiset siirretään "Kun"- ja "Ja"-riveille ja sitten lopputulos siirretään "Niin"-riville. Jos käyttötapauksissa on poikkeuksia tai käsittelysääntöjä, niiden kanssa joutuu soveltamaan. Niitä voi joko parametrisoida tai sitten tehdä kokonaan uusia testiskenaarioita.

#### 4.3 JBehave Spring-projektissa

JBehave-projektin kotisivulta [15] löytyy runsaasti teknistä tietoa aiheesta ja mm. valmiita arkkityyppejä, joiden avulla pystyy helposti luomaan uuden Maven-projektin, jos-

sa JBehave-moduuli on valmiina. Arkkityyppi on ikään kuin muotti, jonka avulla voidaan generoida halutunkaltaisia projektipohjia.

Minulla oli käytettävissä projektin luomiseen valmis arkkityyppi, jossa oli myös JBehave-moduuli olemassa ja riippuvuudet kunnossa. Omassa projektissani JBehave-testit tulivat omaan testimoduuliin.



Kuvio 5. Testimoduuli Eclipsen Package Explorer –näkyssä.

Kuviossa 5 näkyy testimoduulin rakenne. Siinä näkyy kohtalaisen havainnollisesti JBehave-testien eri osat. Src/main/resources-kansiossa on selväkieliset testitapaukset. Ne kytkeytyvät sen yläpuolella näkyviin \*Steps.java-luokkiin, joista kutsutaan tarpeen mu-

kaan \*Page.java-luokan metodeita. Page-luokan metodeista pystytään web driverin välityksellä ohjaamaan käyttöliittymää.

#### 4.4 Testien tekeminen

JBehaven avulla testejä tehdessä työ voidaan jakaa viiteen osaan:

1. tarinan kirjoitus, selkokielineen testi
2. tarinoiden testiaskeleiden kytkeminen Javaan
3. tarinoiden konfigurointi
4. tarinoiden "ajo"
5. tulosten tarkastelu.

Nämä vaiheet toteuttamalla voidaan luoda testipatteri, joka voidaan ajaa aina muutoksien yhteydessä ja myös ajastettuna esim. jatkuvan integroinnin (CI) ympäristössä [15]. Jatkuvan integroinnin ympäristön asioita tarkastellaan myöhemmin.

##### 4.4.1 Tarinan luominen

Ensiksi kirjoitetaan tarina, jolla tullaan testaamaan tuotettavassa sovelluksessa jokin asia tai kokonaisuus. Yhdessä tarinassa voi olla monta testiä. Toteutetussa projektissa avainsanat on kirjoitettu suomeksi. Tässä esittelyssä avainsanojen perään on kirjoitettu vastaava yleinen avainsana. Avainsanoja voidaan lokalisoida tai muuttaa paremmin omiin tarkoituksiin sopiviksi [16].

JBehavella testejä tehdessä avainsanat ovat seuraavia:

- Tarina (Narrative) – Testisetti, joka on tyypillisesti yhdessä tiedostossa.
- Skenaario (Scenario) – Testisetin yksi testitapaus.

- Oletetaan (Given) – Tilanne, johon sovellus alustetaan, jotta testi voidaan tehdä.
- Kun (When) – Käyttäjän tekemä toiminto.
- Ja (And) – Jos käyttäjän toimintoja on useita, merkataan myöhemmät toiminnot näin.
- Niin (Then) – Lopputilanne, jossa sovelluksen pitää olla, jotta testi hyväksytään. Tämän jälkeen voi tulla myös uusia "Kun"-rivejä, jolloin testiä vain jatketaan.

Tarkastellaan seuraavaksi esimerkkitapausta, josta ilmenee, kuinka tarinat kytkeytyvät suoritettavaan ohjelmakoodiin.

```

1 Tarina:
2 Yhteyshenkilöihin liittyvät
3
4 Meta:
5 @vha-test
6 @yhteyshenkilo
7
8 Skenaario: Etusivun avaaminen
9 Oletetaan käyttäjä antaa etusivun osoitteen selaimeen
10 Niin sovelluksen etusivu aukeaa
11
12 Skenaario: Yhteyshenkilön lisääminen
13 Oletetaan käyttäjä on etusivulla
14 Kun käyttäjä syöttää <asiakastunnuksen> ja hakee yhteyshenkilöitä
15 Ja valitsee Uusi yhteyshenkilö
16 Ja syöttää uuden yhteyshenkilön <etunimen>, <sukunimen>, <katuosoitteen>, <postinumeron>
17 Ja tallentaa yhteyshenkilön
18 Niin näytetään Yhteyshenkilön luonti onnistui
19
20 Esimerkit:
21 |asiakastunnuksen|etunimen|sukunimen|katuosoitteen|postinumeron|postitoimipaikan|
22 |6000059-8|Teppo|Testaaja|Testikatu 1 A 3|00510|Helsinki|0441441456|teppo.testaaja@vha.fi|
23

```

Koodiesimerkki 1. Esimerkki tarinasta

Koodiesimerkin 1 tarinaesimerkissä testataan sovelluksen yhteyshenkilöihin liittyviä toimintoja. Tarinoihin voi lisätä omia meta tageja, joiden avulla voi esimerkiksi testien tekovaiheessa ajaa vain tietyllä tagilla olevia testejä, jotta ei tarvitse aina ajaa koko testipatteria. Tämä on varsin hyödyllinen ominaisuus, sillä testien ajaminen voi kestää pitkäänkin, koska ne tehdään käyttöliittymällä.

Tarinassa on kaksi eri skenaariota. Ensimmäisessä skenaariossa testataan vain, että sovelluksen etusivu aukeaa. Toisessa skenaariossa testataan, että uuden yhteyshenkilön luominen onnistuu. Skenaarioita voi myös parametrizoida esimerkin mukaan, sillä ei ole järkevää kirjoittaa omaa skenaariota eri parametreille. Jos edellä olevassa esimerkissä haluttaisiin testata kahden eri yhteyshenkilön luominen, niin Esimerkit-osioon (Examples) voitaisiin kirjoittaa vain uusi rivi toisilla parametreilla. Näin ollen skenaario suoritettaisiin niin monta kertaa kuin sitä seuraavassa Esimerkit-osiossa on rivejä.

#### 4.4.2 Tarinoiden kytkeminen Javaan

Tarkastellaan edellä olevassa tarinassa ollutta skenaariota, jossa luotiin uusi yhteyshenkilö. Tarinassa oli rivi:

```
16 Ja syöttää uuden yhteyshenkilön <etunimen>, <sukunimen>, <katuosoitteen>, <postin
```

Koodiesimerkki 2. Tarinan parametrisoitu rivi

Java-ohjelmassa pitää nyt olla määriteltynä sellainen koodi, joka osaa tehdä halutun asian. "Ja" on avainsana, joka viittaa tässä tapauksessa @When annotaatioon, koska sitä edeltävällä rivillä oli "Kun"-avainsana. Tarkasteltava rivi kytkeytyy alla näkyvään metodiin.

```

14 @When("syöttää uuden yhteyshenkilön <etunimen>, <sukunimen>, <katuosoite>")
15 public void insertUudenYhteyshenkilönTiedot(
16     @Named("etunimen") String etunimi,
17     @Named("sukunimen") String sukunimi,
18     @Named("katuosoitteen") String katuosoite,
19     @Named("postinumeron") String postinumbero,
20     @Named("postitoimipaikan") String postitoimipaikka,
21     @Named("puhelinnumeron") String puhelinnumero,
22     @Named("sähköpostin") String sähköposti
23 ) {
24     abstractPage.insertText("/*[@id='etunimi']", etunimi);
25     abstractPage.insertText("/*[@id='sukunimi']", sukunimi);
26     abstractPage.insertText("/*[@id='osoite.katuosoite']", katuosoite);
27     abstractPage.insertText("/*[@id='osoite.postinumbero']", postinumbero);
28     abstractPage.insertText("/*[@id='osoite.postitoimipaikka']", postitoimipaikka);
29     abstractPage.insertText("/*[@id='puhelinnumero']", puhelinnumero);
30     abstractPage.insertText("/*[@id='sähköposti']", sähköposti);
31
32 }
33

```

Koodiesimerkki 3. Esimerkki selkokielisten tarinoiden kytkemisestä Java-koodiin.

Parametrisoidut muuttujat voidaan ottaa @Named-annotaation avulla Java-muuttujiin, jolloin niiden käsittely on helpompaa. Sitten koodista voidaan kutsua erilaisia toimintoja, joita on ohjelmoitu taustalle. Koodiesimerkissä 3 rivillä 24 kutsutaan abstractPage-luokan metodia insertText, joka osaa laittaa halutun arvon parametrina annetun XPath-hin osoittamaan kenttään. Haluttu muuttuja laitetaan siis web driverin avulla selaimessa näkyvään HTML-elementtiin, jonka id on etunimi (ks. kuvio 6). Muut muuttujat laitetaan vastaavasti omille paikoilleen.

Seuraavaksi halutaan tallentaa täytetyt tiedot:

```

17 Ja painaa Tallenna

```

Koodiesimerkki 4. Tarinan skenaariossa oleva tallennusrivi.

Koodiesimerkin 4 rivi suorittaa seuraavan koodin. Jälleen avainsana "Ja" viittaa @When-annotaation, koska edellinen rivi oli "Kun"-avainsanalla varustettu (ks. koodiesimerkki 1).

```

@When("tallentaa yhteyshenkilön")
public void saveYhteyshenkilo(){
    abstractPage.clickByCssSelector("div.painikerivi input.oikea");
}

```

Koodiesimerkki 5. Esimerkki Javakoodiin kytketystä tarinan rivistä. HTML-elementin etsintä CSS-selektorilla.

Käydään siis painamassa sellaista elementtiä, jonka CSS-tyylitys vastaa määrittelyä "div.painikerivi input.oikea", joka tarkoittaa ensimmäistä elementtiä sellaisen HTML-divin sisällä, jonka CSS-luokka on "painikerivi" ja jossa on itsessään CSS-luokka "oikea" (ks. kuvio 6). Saman asian voisi hoitaa käyttämällä kuvassa näkyvää id:tä, mutta ensimmäisellä tavalla saavutetaan se hyöty, että samaa koodia voidaan tarvittaessa käyttää useimmissa lomakkeen lähetytapauksissa, koska käytännössä lomakkeen "submit" on käytettävyyden vuoksi lähes aina painikerivillä oikeassa reunassa.

The screenshot shows a web form titled "Yhteyshenkilö" (Contact Person). It contains several input fields for personal and company information. At the bottom, there are two buttons: "Edellinen" (Previous) and "Tallenna" (Save). A red box highlights the "Tallenna" button, and a red text label next to it shows the CSS selector used to find it: "div.painikerivi input.oikea".

Kuvio 6. Esimerkkejä sivulla olevista HTML-elementtien attribuuteista.

Tekemässäni projektissa olen kasannut AbstractPage.java-nimiseen luokkaan yleisiä metodeja, kuten että voidaan kysyä tietoja HTML-elementeistä XPathin, CSS:n, elementtien nimen ja id:n avulla. Vastaavalla tavalla on toteutettu myös elementtien painamiset ja niissä olevan tiedon tarkistukset. Seuraavassa kuviossa näkyy esimerkkejä näistä yleiskäyttöisistä AbstractPage-luokan metodeista.

```

154 public void clickLinkByPartialText(String text) {
155     clickLinkByPartialText(text);
156     sleep();
157 }
158
159 public void clickElementById(String string) {
160     findElement(By.id(string)).click();
161     sleep();
162 }
163
164 public void clickByCssSelector(String selector){
165     findElement(By.cssSelector(selector)).click();
166     sleep();
167 }
168
169 public void openLinkByText(String linkText, int timeout) {
170     findElement(By.linkText(linkText)).click();
171     sleep();
172 }
173
174 public void selectComboValue(String value, String id){
175     sleep();
176     Select select = new Select(findElement(By.id(id)));
177     select.selectByValue(value);
178 }
179

```

Koodiesimerkki 6. Esimerkkejä AbstractPage-luokan metodeista

Kun esimerkissä halusin tallentaa yhteyshenkilön, kutsuin metodia `clickByCssSelector()` ja annoin parametriksi halutun CSS-määrittelyn. Koodiesimerkissä 12 näkyvän `AbstractPage.java`-luokan koodi komentaa web driverin painamaan CSS-määrittelyn mukaista elementtiä.

Lisäksi olen määritellyt omat luokat sivukohtaisille toiminnoille. Nämä Page-luokat sitten osaavat käskä itse web driveria suorittamaan halutun toiminnon. Koodiesimerkissä 12 näkyy joitakin esimerkkejä yleisistä toiminnoista, joita voidaan kutsua `*Steps.java`-luokista.

#### 4.5 Miten helpottaa testien tekemistä

Omassa projektissani olen huomannut, että jos käyttöliittymäkehitystä tehdään ajattelematta tuotteen testattavuutta, on JBehave-testien tekeminen hidasta ja hankalaa. Kokemuksen kautta oppii erilaisia keinoja toteuttaa käyttöliittymän sivut niin, että se

sisältää yksilöityjä HTML-elementtejä tai muita järkeviä keinoja haluttujen elementtien löytämiseen. Yksi hyvä keino niiden löytämiseen on käyttää id-attribuuttia aina, kun se on mahdollista. Kuitenkin, kuten aiemmin kuvattiin, voi olla järkevää käyttää CSS-luokkia, jolloin yhdellä määrittelyllä voidaan tarkoittaa vain yleisesti lomakkeen lähetystä ottamatta kantaa siihen, onko ko. napin id "tallenna", "lähetä" jne. Näin voidaan helpottaa sitä, ettei jokaisen sivun "Edellinen", "Seuraava", "Tallenna" yms. napeille tarvitse määritellä omanlaista testiä.

Edellä kuvattuja id:itä ja CSS-luokkia voidaan kuitenkin helposti lisäillä sivuihin myöhemminkin. Haastavampaa on silloin, kun halutaan etsiä käyttöliittymästä joku tietty asia, esimerkiksi yhteyshenkilö ja vaikkapa painaa kyseisellä taulukon rivillä jotain muuta HTML-elementtiä. Tällaisissa tilanteissa aikaa menee usein melko paljon.

#### 4.5.1 Konfigurointi

Konfiguroinnissa on tarkoitus määrittää, kuinka tarinoissa olevat testit löytävät ne koodit, jotka todellisuudessa suorittavat halutut toiminnot. Lisäksi voidaan määrittää, missä testin tuloksia voi tarkastella. [17].

```

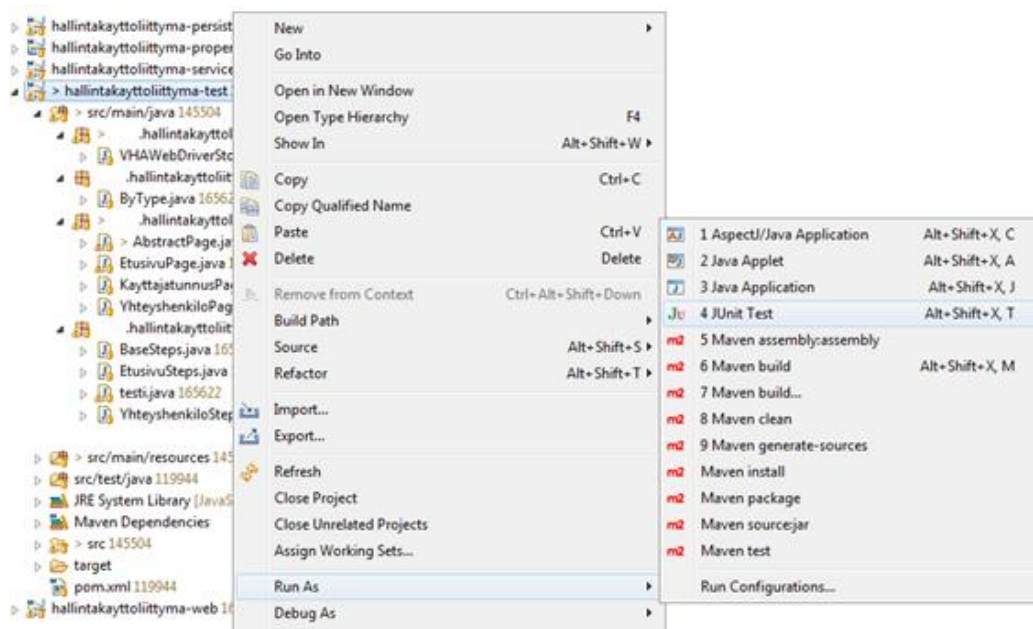
8
9 public class VHAWebDriverStories extends AbstractWebDriverStories {
10     @Override
11     public Object[] useStepInstances() {
12         return new Object[] {
13             new EtusivuSteps(getWebDriverProvider()),
14             new YhteyshenkilöSteps(getWebDriverProvider()),
15             new AsetuksetSteps(getWebDriverProvider()),
16             new LiiketoimintapalveluSteps(getWebDriverProvider())
17         };
18     }
19
20     @Override
21     public String useStoryPattern() {
22         return "**/*.story";
23     }
24
25 }
```

Koodiesimerkki 7. Esimerkki VHAWebDriverStories.java-luokasta.

Edellä olevassa koodiesimerkissä näkyy WHAWebDriverStories.java-luokassa olevat määrittelyt. Metodissa useStoryPattern() kerrotaan, että tarinoita etsitään sellaisista tiedostoista, joiden pääte on ".story". Metodissa useStepInstances() puolestaan määritetään, mistä löytyy @Given, @When ja @Then -annotaatiot, joihin skenaarioissa esiintyvät stepit kytketään.

#### 4.5.2 Testien ajaminen

Testit käynnistetään ajamalla sovelluksen test-moduuli JUnit-testinä seuraavaan tapaan.

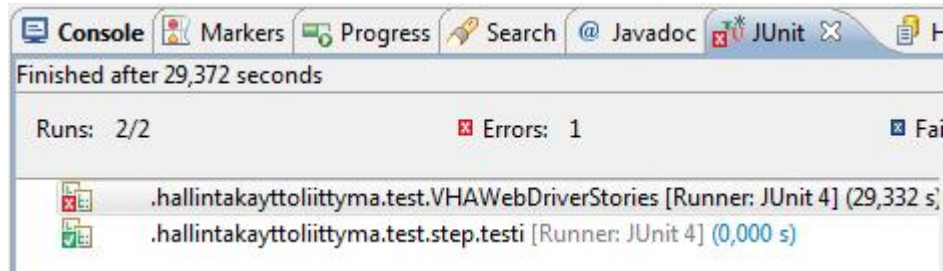


Kuvio 7. JBehave-testien käynnistys

Testit voidaan käynnistää myös komentoriviltä suorittamalla Maven-projektin testit.

#### 4.5.3 Tuloksen tarkastelu

Jos testit ajetaan suoraan Eclipsestä, tieto testien onnistumisesta tai epäonnistumisesta näkyy suoraan JUnit-näkymässä seuraavasti:



Kuvio 8. Tieto JBehave-testien onnistumisesta

Testin koko suoritusketjun voi katsella konsolilta tai monin eri tavoin. Tuettuja tapoja on muun muassa tulostus tiedostoon tekstinä, html:nä ja xml:nä [18].

Myöhemmin, kun testit siirretään jatkuvan integroinnin piiriin Jenkinsille, pystyy siellä tarkastelemaan graafin avulla, kuinka suuri osa testeistä on mennyt läpi sekä tutkia mahdollisia virheitä yllä mainituilla tavoilla. Jos Jenkinsillä käy katsomassa vaikkapa viimeisen viiden ajon tulokset, voi saada sellaisen kuvan, että kaikki testit ovat epäonnistuneet. Ainakin minun projektissa on vain yksi suoritettava testiluokka VHAWebDriverStories ja tällöin jos yhdessäkin testissä on virhe, niin koko testin tulos on virheellinen. Näin sen kuuluukin toimia, mutta esim. arvioitaessa kehitettävän sovelluksen tilaa tulee tarkastella tarkemmin graafeista sitä, kuinka suuri osa testissä ajetuista skenaarioista on mennyt läpi. Kaikki virhetilanteet on tietenkin selvitettävä.

#### 4.6 Ongelmia

JBehave-testejä ajettaessa voi tulla vastaan mystisiä ongelmia, joille ei tunnu olevan mitään selitystä. Välillä joku testi menee läpi, välillä ei. Itselläni kävi niin, että useaan otteeseen sain vain HTTP-virheen 500, joka tarkoittaa yleistä virhettä palvelimella. Syyksi paljastui se, että Java yrittää kysyä web driveria tekemään asioita selaimella vähän liian nopeaan tahtiin, jolloin selain ei "pysy perässä".

Tämän ongelman sain ratkaistua sillä, että lisäsin pienen viiveen niihin AbstractPage.java-luokan metodeihin, joissa selaimella siirrytään sivulta toiselle.

Lisäksi kerran kävi niin, että web driver ei löytänyt jotain sivulla näkyvää HTML-elementtiä, vaikka se löysi vierestä toisen elementin. Ihmetystä aiheutti se, että sivu oli täysin staattinen ja kahta sen eri elementtiä haettiin id-attribuutin perusteella. Toinen

löytyi ja toinen ei. Sitten taas, kun ajoin testin uudelleen, molemmat elementit löytyivät. Tähän löytyi myöhemmin syyksi se, että olin tehnyt kahteen eri steps-luokkaan samannimisen @When-annotaation, jolloin tarinan rivi kytkeytyi välillä eri steps-luokan metodeihin.

## 5 Käyttöliittymätestien automaatio

Jotta tehdyistä käyttöliittymätesteistä olisi mahdollisimman paljon hyötyä, niiden suorittaminen kannattaa viedä osaksi suoritettavaa jatkuvaa integraatiota, mikäli tällainen mahdollisuus on. Omassa projektissani pystyin käyttämään Jenkinsiä tähän tarkoitukseen.

BDD:ssä tuotettavien automaattisten käyttöliittymätestien suorittaminen on huomattavasti hitaampaa kuin esimerkiksi yksikkötestien, joten niiden suorittaminen ei välttämättä ole järkevää aina kun versionhallintaan tulee muutoksia. Varsinkin jos useampi henkilö kehittää yhtä sovellusta voi käydä niin, että kaikkia käyttöliittymätestejä ei ehditä ajaa ennen kuin seuraava testien käynnistämiskäsky tulee. [19.]

### 5.1 Jenkins

Jenkins on avoimen lähdekoodin web-pohjainen työkalu jatkuvan integroinnin aikaansaamiseksi. Sen nimi muutettiin Hudsonista Jenkinsiksi, kun Oracle sai Sun Microsystemsin ostaessaan oikeudet Hudson-tavaramerkkiin, eikä Oraclen haluttu vaikuttavan tuotteen kehitykseen.

Jenkinsillä voidaan monitoroida ohjelmistojen integrointia, yksikkötestien suorittamista, sekä siihen voidaan liittää vaikkapa koodianalysaattoreita. Jenkinsiin voidaan konfiguroida rutiineja, jotka määrätyin väliajoin käyvät vaikkapa hakemassa versionhallinnasta uusimmat koodit, kääntävät ne, ajavat yksikkötestit ja julkaisevat ne esimerkiksi kehityspalvelimelle. Myös automaattisten käyttöliittymätestien ajamiseen voidaan tehdä oma rutiini.

## 5.2 CI-ympäristön käyttöönotto

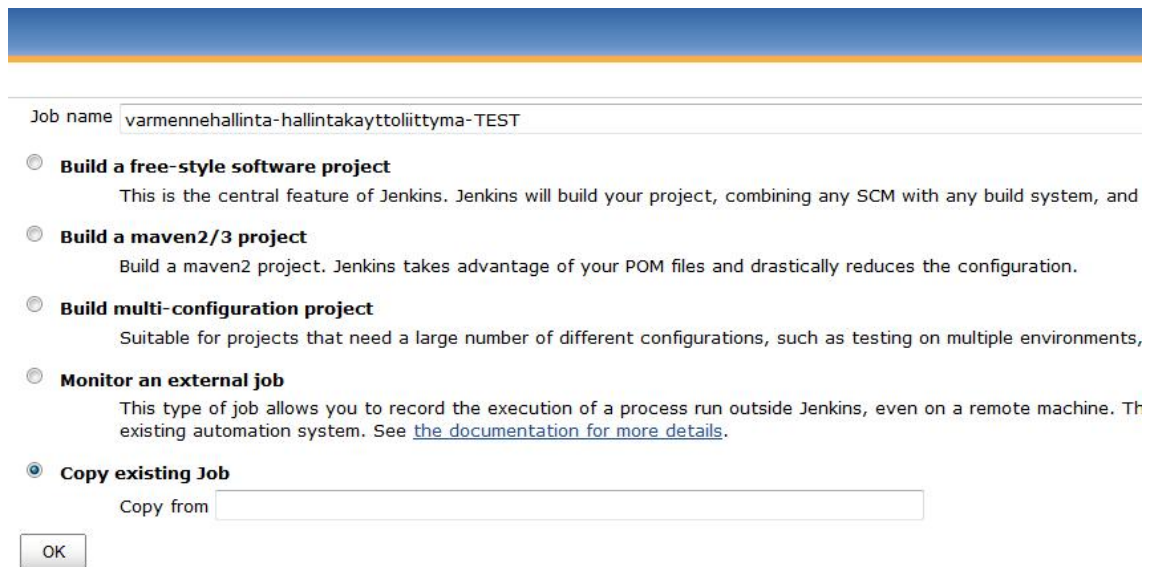
Jenkinsin asennusta ja konfigurointia ei tässä työssä käydä läpi, mutta siitä löytyy ohjeita internetistä [20]. Keskitymme seuraavaksi sellaisen rutiinin tekemiseen, joka hakee ajastetusti uusimmat koodit versionhallinnasta ja suorittaa käyttöliittymätestit määritetyssä ympäristössä.

Valitaan Jenkinsin etusivulta New job.



Kuvio 9. Uuden rutiinin luominen

Määritetään seuraavaksi rutiinin nimi ja tyyppi. Se voidaan luoda tyhjästä tai kopioida pohjaksi joku olemassa oleva rutiini. Sitten painetaan OK.



Job name

☐ **Build a free-style software project**  
This is the central feature of Jenkins. Jenkins will build your project, combining any SCM with any build system, and

☐ **Build a maven2/3 project**  
Build a maven2 project. Jenkins takes advantage of your POM files and drastically reduces the configuration.

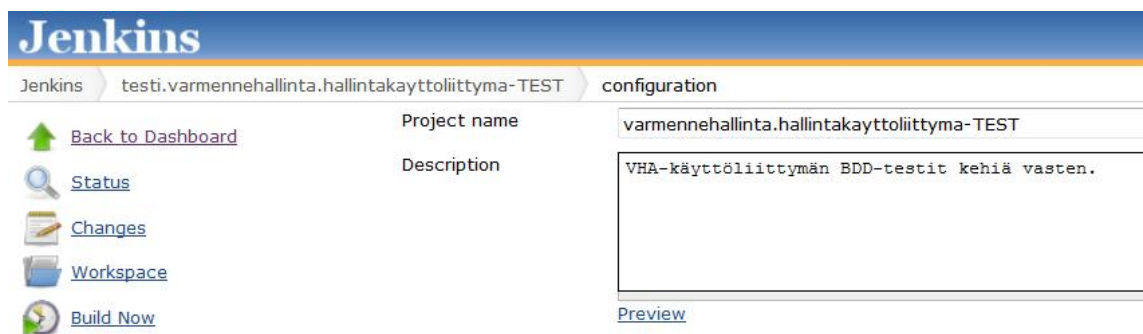
☐ **Build multi-configuration project**  
Suitable for projects that need a large number of different configurations, such as testing on multiple environments,

☐ **Monitor an external job**  
This type of job allows you to record the execution of a process run outside Jenkins, even on a remote machine. Th existing automation system. See [the documentation for more details](#).

☒ **Copy existing Job**  
Copy from

Kuvio 10. Uuden rutiinin nimi ja tyyppi

Nyt uusi rutiini on luotu ja avautuu konfiguraatiosivu. Täytetään loput tiedot paikalleen, jotta se saadaan tekemään jotain järkevää. Ensimmäisenä laitetaan rutiinin kuvaus (kuvio 11).



Jenkins **testi.varmennehallinta.hallintakayttoliittyma-TEST** configuration

[Back to Dashboard](#) [Status](#) [Changes](#) [Workspace](#) [Build Now](#)

Project name

Description

[Preview](#)

Kuvio 11. Rutiinin kuvaus

Sitten määritetään, kuinka paljon vanhoja buildeja säilytetään. Seuraavan esimerkin mukaisilla asetuksilla säilytetään kymmenen viimeisintä buildia.

☒ Discard Old Builds

Days to keep builds

if not empty, build records are only kept up to this number of days

Max # of builds to keep

if not empty, only up to this number of build records are kept

Kuvio 12. Vanhojen buildien säilytys.

Määritetään versionhallinnan kansio, josta testit haetaan. Tässä tapauksessa kansio on projektin test-moduuli (kuvio 12). Lisäksi määritetään niin, että jobi poistaa aina ensin vanhat koodit ja hakee ne sitten uudelleen versionhallinnasta komennolla "svn checkout".

**Source Code Management**

☐ CVS  
☐ None  
☒ Subversion

Modules

Repository URL

Local module directory (optional)

[Add more locations...](#)

Check-out Strategy

Delete everything first, then perform "svn checkout". While this takes time to execute, it ensures that the workspace is in the pristine state.

Repository browser

[Advanced...](#)

Kuvio 13. Lähdekoodien sijainti

Ao. kuviossa määritetään, millä selaimella testit ajetaan. Tulee huomioida, että Jenkinspalvelimella tulee olla asennettuna haluttu selain ja sen web driver. Samalle riville määritetään myös, että mihin osoitteeseen siirrytään, kun testit käynnistetään. Itselläni tähän tulee sovelluksen etusivu. Tässä vaiheessa voidaan myös määrittää, mitä testejä ajetaan. Eli jos story-tiedostoissa on määritetty meta-tageja, voidaan niiden mukaan ajaa, tai jättää ajamatta testejä. Kuvassa näkyvällä `-Dmeta.filter="*-kayttajatunnus"` – optiolla ajetaan kaikki muut tarinat, paitsi se, jonka meta –tagina on `@kayttajatunnus`.

Build	
Root POM	pom.xml
Goals and options	-U clean verify -Dop-integration-test -Durl="http://..." -Dbrowser=firefox -Dmeta.filter="**-kayttajatunnus"

Kuvio 14. Sovelluksen aloitussivun ja käytettävän selaimen määrytykset.

Lopuksi jobi ajastetaan. Jenkinsissä ajastukseen liittyvät määrytykset ovat enemmän tai vähemmän kryptisen näköisiä, mutta asian niillä saa hoidettua. Tämä jobi käynnistyy nyt joka yö 02.30 (kuvio 15). Kentän vieressä on ikoni, jota klikkaamalla löytyy tietoa ajastuksen syntaksista.

Sitten rutiini tallennetaan.

Build Triggers	
<input type="checkbox"/>	Build whenever a SNAPSHOT dependency is built
<input type="checkbox"/>	Build after other projects are built
<input type="checkbox"/>	Trigger builds remotely (e.g., from scripts)
<input checked="" type="checkbox"/>	Build periodically
Schedule	30 2 * * *
<input type="checkbox"/>	Poll SCM

Kuvio 15. Jenkins jobin ajastus

### 5.3 Testien suorituksen analysointi

Edellisessä luvussa määritimme testit suoritettavaksi joka yö klo 02.30. Testirutiini kannattaa ajastaa niin, että ennen sitä suoritetaan sellainen rutiini, joka kokoaa päivän aikana mahdollisesti usean kehittäjän tuottamat koodit uudelleen ja asentaa ne testipalvelimelle, jossa sitten tekemämme testit tullaan ajamaan. Testijobin konfiguraatioissa voisimme määrittää, että aina ennen kyseessä olevaa jobia suoritetaan haluttu jobi. Ks. seuraava kuvio. Ensin ajettavia jobeja voi olla useita ja ne tulee erottaa pilkulla.

**Build Triggers**
☐ Build whenever a SNAPSHOT dependency is built

☒ Build after other projects are built

Projects names

varmennehallinta.hallintakayttoliittyma-POLLER

Multiple projects can be specified like 'abc, def'

☐ Trigger builds remotely (e.g., from scripts)

☒ Build periodically

Schedule

30 2 \* \* \*

☐ Poll SCM

Kuvio 16. Jobia ennen ajetaan aina ensin määritettävä jobi.

Kun automaattiset testit määritetään ajettaviksi öisin, nämä rutiinitestit eivät kuormita testiympäristöä silloin, kun esimerkiksi testaajat tekevät työtään etsiessään sovelluksesta kompleksisempia virhetilanteita. Lisäksi kun kehittäjä tulee seuraavana päivänä töihin, hänellä on aina tuore tieto sovelluksen tilasta ja mahdollisista muutosten aiheuttamista ongelmista. Omasta mielestäni tällä on myös psykologisesti positiivinen vaikutus siihen, että on helpompi aamulla heti tarttua työhön, kun ensimmäisenä katsoo, miten testien suoritus on onnistunut. Jos testeissä on havaittu virheitä, ryhtyy korjaamaan niitä.

#### 5.4 Automatisoinnin haasteita

Käytännön elämän sovelluksissa on usein niin, että käyttäjäryhmiä voi olla useita. Vaikka käyttöoikeuksien hallinta itsessään on jo haastava asia, niin testien automatisoinnille se aiheuttaa vielä omat lisähaasteensa. Testien kannalta olisi kohtalaisen yksinkertaista, mikäli käyttäjän tunnistaminen tapahtuisi sovellukseen kirjautumalla, jolloin testin alussa tai jopa kesken testien voitaisiin vaihtaa kirjautunutta käyttäjää selainta ohjaamalla. Toteutetun sovelluksen tapauksessa käyttäjän tunnistaminen tapahtuu työasemakirjautumisen perusteella, joten web driverin avulla ei ole mahdollista vaihtaa kirjautuneen käyttäjän tietoja.

Työasemalta testejä ajaessa voisi tehdä niin, että kirjautuu työasemalle sopivilla tunnuksilla, jotta eri testit saa ajettua. Jenkins-ympäristössä homma meneekin haasta-

vammaksi, koska palvelimen kirjautunutta käyttäjää ei pysty vaihtelevaan miten satuu. Tätä ongelmaa ei ole vielä ratkaistu.

Lisähaasteita aiheutuu myös siitä, että toteutetussa sovelluksessa on toimintoja, jotka pystyy tekemään vain kahden henkilön silmien kautta. Eli ensin jonkun henkilön täytyy tehdä toiminto, jonka sitten joku muu henkilö voi hyväksyä. Tällaisen testitapauksen osalta tulee harkita, kannattaako sitä yrittää automatisoida. Automatisoidun testin vaatima työmäärä on mielestäni melko suuri suhteessa testitapauksen tekemiseen tarvittava aika, joten yksittäistapauksissa kannattaa mieluummin testata tällaiset testitapaukset manuaalisesti, ellei käytettävissä ole sopivia työkaluja asian helpottamiseksi.

## 6 Menetelmän analysointi

### 6.1 Työmäärä

Automaattisten testien tekeminen lisää kehittämisen kustannuksia hetkellisesti, vaikka todennäköisesti automatisoiduista testeistä on suuri hyöty jo kehitysvaiheessa. Olen huomannut, että ajan kuluessa - vaikka olen toteuttanut koko sovelluskoodin yksin - en enää muista, mihin kaikkeen mikään koodi vaikuttaa.

Ennen kaikkea hyöty tulee ylläpitovaiheessa. Automaattisia testejä on tehty jo pitkään erilaisilla työkaluilla, mutta erityisesti jos joku muu kuin ohjelman kehittäjä toimii sen ylläpitäjänä, on suuri hyöty, että automaattitestit voidaan ajaa kehitysympäristössä heti kun ohjelma käännetään. Näin kehittäjän ei tarvitse odottaa, että ohjelma viedään esim. testausympäristöön, jolloin mahdolliset muun koodin rikkoutumiset havaittaisiin. Jos projektissa tehdään kattavasti JBehave-testit, ei muulle automaatiolle ole välttämättä tarvetta.

Menetelmää käytettäessä työmäärä aluksi on erittäin suuri, kun joudutaan luomaan perusinfrastruktuuri testien tekemistä varten. Siinä vaiheessa, kun tavalliset "embedderit" eli käyttöliittymää ohjaavat toiminnot on toteutettu, uusien testien tekemiseen tarvittava aika vähenee ja varsinkin kokeneempi kehittäjä luo tämän jälkeen perustestit

rutiinitoimenpiteenä. Näin ollen jo muutamalla regressiokierroksella aletaan saavuttaa hyötyä tehdyistä testeistä ja testausammattilaisen osaaminen voidaan keskittyä haastavampiin tilanteisiin. Puhumattakaan siitä, että useimmilla ammatti-ihmisillä tuskin riittää motivaatiota napsuttelevaan käyttöliittymällä rutiinitestitapauksia läpi kokeillen kaikki mahdolliset virhesyötteet ja syötevariaatiot.

## 6.2 Saadaan mitä halutaan

Kun ohjelman kehittämistä tarkastellaan jatkuvasti siitä näkökulmasta, että voiko sillä suorittaa halutut tarinat ja meneekö tehdyt testiskenaariot onnistuneesti läpi, kehittämisen fokus on juuri niissä asioissa, joita asiakas tarvitsee.

## 6.3 Toimivaa koodia

Kun ohjelmoijalla on käytössään kattavat testit, hän voi huolettomammin tehdä kehitystä ja refaktoroida koodia, jolloin se on toimintavarmempaa ja selkeämpää erityisesti tuleville ylläpitäjille, mutta myös itse kehittäjälle. Tämä kuitenkin vaatii osaamista ja viitseliäisyyttä myös tulevilta ylläpitäjiltä. Testeistä ei ole hyötyä, jos niitä ei ylläpidetä muun ohjelmakoodin tavoin.

# 7 Työn arviointia

Alun perin sain työpaikaltani opinnäytetyöaiheen, jossa tarkoituksena oli toteuttaa uuden käyttöliittymän tekeminen BDD:nä. Periaatteessa pelkästään käyttöliittymän toteutus olisi voinut olla aiheenani, mutta siinä ongelmaksi olisi tullut se, että itse kehitettävän tuotteen sisältö olisi liian suurelta osin salaista, jolloin opinnäytetyön kirjoittaminen ja esittely riittävän kattavasti olisi erittäin hankalaa. Näin päädyin tekemään opinnäytetyötäni kehitysmenetelmän näkökulmasta.

Projektin aloitus sujui hieman takkuisesti ja pääsin tekemään toteutusta vasta kuukausia myöhemmin, kuin olimme töissä suunnitelleet. Kun kehittäminen pääsi käyntiin ja aloin hieman myöhemmin miettiä työssä esiteltujen JBehave-testien tekemistä, törmäsin muutamiin ongelmiin, jotka viivästyttivät kehittämistä BDD:n kannalta. Itse kehi-

tystyötä pystyin tekemään, mutta käyttöliittymätestien tekeminen toisinaan unohtui pitkiksikin ajoiksi

Nyt olenkin vasta myöhemmin saanut rakenneltua perusinfrastruktuurin kasaan testien osalta ja päässyt kunnolla opettelemaan asiaa. Niinpä ihan puhtaasti BDD:nä en ole projektia pystynyt tekemään, kun olen testejä tehnyt vasta myöhemmin sovelluksen jo monelta osin toimiessa. Tässä tapauksessa jos olisin tehnyt testit jo ennen toteutustyötä, olisi joutunut aika paljon tekemään muutoksia niihin, koska käyttöliittymän toiminnallinen määrittely on hieman elänyt projektin myötä. Voisi ajatella, että kyse on huonosta projektikäyttäytymisestä tai huolimattomasta pohjatyöstä, mutta tekemälläni sovelluksella ei ole ollut varsinaista tilaajaa, joten olemme yrittäneet rakentaa ratkaisuja, jotka palvelisivat mahdollisimman monia eri tahoja ja käyttötarkoituksia. Tätä työtä kirjoittaessa työprojekti on vielä kesken, mutta joitakin osia sovelluksesta pystytään jo testaamaan automaattitesteillä.

Tämän opinnäytetyön tekeminen on ollut haastavaa, mutta myös erittäin antoisaa. Ilman opinnäytetyöaihetta olisin tuskin alkanut opiskella BDD:tä. Haasteita minulle on aiheutunut siitä, että olen vasta tätä projektia tehdessä alkanut tutustua työpaikan ohjelmointikäytäntöihin ja järjestelmiin ja pelkästään kouluohjelmoinnista siirtyminen työohjelmointiin on avannut silmiä käytännön ohjelmoinnin kompleksisuuteen. Kun lisänä on ollut kaikille kohtalaisen uuden asian opettelu, niin aikaa asian parissa on joutunut kuluttamaan.

Sivutuotteena olen oppinut mm. Selenium web driverin kanssa työskennellessä nopean tavan käyttöliittymän ajamiseksi tiettyyn tilaan. Toisinaan sovelluksen saaminen haluttuun tilaan voi kestää kymmeniä sekunteja, kun täytellään lomakkeita ja tehdään valintoja. Aika ei sinänsä ole pitkä, mutta jos kehitysvaiheessa sen joutuu tekemään kymmeniä kertoja, alkaa se turhauttaa. Ensimmäisellä kerralla voi käynnistää selaimen asennetun web driver -liitännäisen ja käynnistää nauhoituksen. Kun on päässyt haluttuun vaiheeseen, voi nauhoituksen lopettaa. Tämän jälkeen kun halutaan päästä samaan tilanteeseen uudelleen riittää, että käynnistää nauhoitetun skriptin, jolloin käyttöliittymän napsutteluun tarvitsee käyttää huomattavasti vähemmän aikaa. Näitä skriptejä voi tallentaakin, mutta itse olen käyttänyt sitä vain työkaluna kullakin hetkellä, koska samalla vaivalla testiä tehdessä voi pitää nauhoituksen päällä, jolloin se tarvitsee tehdä vain kerran.

Nyt minulla on työkalupakissa kuitenkin uuden osaamisen alkeet, ja sen käyttäminen tästeden on helpompaa.

## 8 Yhteenveto

Tässä opinnäytetyössä oli tavoitteena tutustua uudehkoon ketterään kehitysmenetelmään, BDD:hen peilaten sitä aiempien menetelmien hyvien ja huonojen puolien tai puutteiden valossa. Tavoitteena oli sekä tutustua kehitysmenetelmän teoriapuoleen että toteuttaa toimiva automaattitestipatteri työn yhteydessä toteutetun käyttöliittymän testaamiseen ja viedä testien suorittaminen osaksi jatkuvaa integraatiota.

Kuten edellisessä luvussa mainittiin, työtä jouduttiin tekemään monin paikoin väärässä järjestyksessä johtuen siitä, että käyttöliittymä oli jo monelta osin toteutettu ennen kuin automaattisia käyttöliittymätestejä päästiin tekemään. Menetelmän ideana on, että testit suunnitellaan ennen kuin testattava ominaisuus toteutetaan.

Opinnäytetyön yhteydessä olen toteuttanut – ja toteutan edelleen – uutta käyttöliittymää, jonka testaamiseen on rakennettu automaattisesti suoritettavia testejä vähentämään nykyisten ja tulevien testaajien työkuormaa. Lisäksi työn yhteydessä kirjoitin tämän dokumentin, joskaan tässä ei salassapitosyistä oteta kantaa toteutetun käyttöliittymäsovelluksen toimintalogiikkaan muilta osin kuin JBehave-testien teknisten ominaisuuksien osalta.

Tämä työ voi toivottavasti olla tukena asiasta kiinnostuneille automaattitestiympäristön pystyttämiseen ja ennen kaikkea automaattisten käyttöliittymätestien tekemiseen. Pääpaino on ollut testien tekemisessä, mutta tekstissä on myös viitteitä ympäristöjen pysytys- ja konfigurointiohjeisiin.

Vaikka itse käyttöliittymäsovelluksen kehitys on vielä kesken ja projekti sen parissa jatkuu edelleen. Arvioin, että tavoitteisiin päästiin, koska osa käyttöliittymän toiminnallisuuksista pystytään testaamaan automaattisten testien avulla sekä toimivat testit on saatu vietyä osaksi jatkuvaa integraatiota. Käytännön työn kuvaukseenkin on saatu mielestäni kuvattua testien vaatiman infrastruktuurin pystyttäminen keskeisimmiltä osiltaan.

## Lähteet

1. VTT:n tutkimus ohjelmistokehityksestä.

<[http://www.vtt.fi/research/area/software\\_development.jsp](http://www.vtt.fi/research/area/software_development.jsp)>.

Luettu 12.1.2013.

2. Ohjelmistotuotanto. Verkkodokumentti. Wikipedia.

<<http://fi.wikipedia.org/wiki/Ohjelmistokehitys>>. Luettu 12.1.2013.

3. RUP. Verkkodokumentti. Wikipedia.

<<http://fi.wikipedia.org/wiki/RUP>>. Luettu 25.1.2013.

4. Suihkulähdemalli. Verkkodokumentti. Wikipedia

<<http://fi.wikipedia.org/wiki/Suihkul%C3%A4hdemalli>>. Luettu 25.1.2013.

5. Ketterä ohjelmistokehitys. Verkkodokumentti. Wikipedia.

<[http://fi.wikipedia.org/wiki/Ketter%C3%A4\\_ohjelmistokehitys](http://fi.wikipedia.org/wiki/Ketter%C3%A4_ohjelmistokehitys)>. Luettu 27.1.2013.

6. Agile software development. Verkkodokumentti. Wikipedia.

<[http://en.wikipedia.org/wiki/Agile\\_software\\_development](http://en.wikipedia.org/wiki/Agile_software_development)>. Luettu 27.1.2013.

7. Ketteryys haltuun: Scrum päähkinänkuoressa. Verkkodokumentti.

<<http://www.meteoriitti.com/fi-FI/tiedotteet/ajankohtaista/ketteryys-haltuun-scrum-pahkinankuoressa>>. Luettu 27.1.2013.

8. Agile Methodologies for Software Development. Verkkodokumentti.

<<http://www.versionone.com/Agile101/Agile-Development-Methodologies-Scrum-Kanban-Lean-XP/>>. Luettu 27.1.2013.

9. Scrum. Verkkodokumentti. Wikipedia.

<<http://fi.wikipedia.org/wiki/Scrum>>. Luettu 27.1.2013.

10. Testivetoinen ohjelmistokehitys. Verkkodokumentti. Wikipedia.

<[http://fi.wikipedia.org/wiki/Testivetoinen\\_kehitys](http://fi.wikipedia.org/wiki/Testivetoinen_kehitys)>. Luettu 27.1.2013.

11. TDD ja ATDD. Luentomateriaali. Niklas Collin.

<<http://www.cs.tut.fi/~testaus/s2010/luennot/Collin.pdf>>. Luettu 27.1.2013.

12. Test-driven development. Verkkodokumentti. Wikipedia.

<[http://en.wikipedia.org/wiki/Test-driven\\_development](http://en.wikipedia.org/wiki/Test-driven_development)>. Luettu 27.1.2013.

13. Introducing BDD. Verkkodokumentti.

<<http://dannorth.net/introducing-bdd/>>. Luettu 12.1.2013.

14. Behavior-driven development. Verkkodokumentti. Wikipedia.

<[http://en.wikipedia.org/wiki/Behavior-driven\\_development](http://en.wikipedia.org/wiki/Behavior-driven_development)>. Luettu 10.12.2012.

15. JBehave. Ohjelmointikehyksen kotisivu.

<<http://jbehave.org>> Luettu 11.3.2013.

16. Stories in your language. Verkkodokumentti.

<<http://jbehave.org/reference/stable/stories-in-your-language.html>>.

Luettu 14.4.2013.

17. JBehave configuration. Verkkodokumentti.

<<http://jbehave.org/reference/stable/developing-stories.html#configuring>>

Luettu 11.3.2013.

18. JBehave Story reporter. Verkkodokumentti.

<<http://jbehave.org/reference/stable/reporting-stories.html>> Luettu 11.3.2013.

19. Ketteryys haltun: yleisimmät ketterät käytännöt. Verkkodokumentti.

<<http://www.meteoriitti.com/fi-FI/tiedotteet/ajankohtaista/ketteryys-haltuun-yleisimmat-ketterat-kaytannot>> Luettu 13.3.2013.

20. Installing Jenkins. Verkkodokumentti.

<<https://wiki.jenkins-ci.org/display/JENKINS/Installing+Jenkins>> Luettu 15.3.2013.