Reeta Jäppinen

# Käyttöliittymän automaatiotestaus ketterän kehityksen ohjelmistoprojektissa

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tietotekniikka

Insinöörityö

3.5.2015

Metropolia

| | |
|---|---|
| Tekijä(t)<br>Otsikko<br><br>Sivumäärä<br>Aika | Reeta Jäppinen<br>Käyttöliittymän automaatiotestaus ketterän kehityksen ohjelmistoprojektissa<br>32 sivua + 1 liite (ei julkaistu)<br>3.5.2015 |
| Tutkinto | Insinööri (AMK) |
| Koulutusohjelma | Tietotekniikka |
| Suuntautumisvaihtoehto | Ohjelmistotekniikka |
| Ohjaaja(t) | Yliopettaja Auvo Häkkinen |


Insinöörityössä luotiin yritykselle selainpohjaisen sovelluksen käyttöliittymätason automaatiotestaus Selenium-testausohjelmiston avulla ja luotuun testauskokonaisuuteen liittyvä testausdokumentaatio.

Työssä tehtiin käyttöliittymätestausta järjestelmän määrittelyissä olevien käyttötapausten perusteella. Tarkoituksena oli luoda nauhoittaen käyttäjän toimia toistavia skriptejä, joiden avulla voitiin regressiotestata käyttöliittymän kautta sovelluksen toimivuutta ketterässä kehityksessä usein tapahtuvien versiojulkaisujen yhteydessä. Testausskriptit koostuivat pääasiassa käyttäjän toimista käyttöliittymän eri osissa. Lisäksi käyttäjän toimien ohessa testattiin näkymien rakennetta, muotoiluja, komponenttien sijaintia suhteessa oletettuun sekä niiden olemassaoloa näkymissä.

Testitapausten nauhoitus oli haastavaa. Alkuperäisten määrittelyjen perusteella nauhoitettuihin testitapauksiin jouduttiin tekemään parannuksia, jotta testitapausten avulla saatiin testattua oleellisimmat asiat versiosta toiseen. Testausohjelmiston käyttö oli helppoa, mutta testiskriptien vaiheet vaativat paljon suunnittelua. Testaustyökalun ominaisuuksien mukaan testausdokumentaatiota parannettiin testitapaus kerrallaan.

Automatisoitu käyttöliittymätestaus osoittautui tehokkaaksi tavaksi testata sovellusta. Nauhoitusten tekeminen vei kuitenkin aikaa ja nauhoitettujen testitapausten ylläpito koettiin haasteeksi tulevaisuudessa. Testiskriptien testauskattavuudessa päädyttiin kompromissiin suhteessa alkuperäiseen suunnitelmaan, haluttiin testata vain oleellisimmat perustoiminnot versiojulkaisun testausprosessissa.




| | |
|---|---|
| Avainsanat | käyttöliittymätestaus, automaatiotestaus, ketterän kehitys, Selenium |

| | |
|---|---|
| Author(s)<br>Title<br><br>Number of Pages<br>Date | Reeta Jäppinen<br>UI Testing in Agile Software Project<br><br>32 pages + 1 attachment (not published)<br>3 May 2015 |
| Degree | Bachelor of Engineering |
| Degree Programme | Information Technology |
| Specialisation option | Software Engineering |
| Instructor(s) | Auvo Häkkinen, Principal Lecturer |

This final year thesis produced an automated user interface (UI) testing using the Selenium web browser automation tool. The testing documentation was improved based on the tests created.

The UI tests were based on use cases described in the system documentation. The purpose was to create test script records that simulated user actions in the interface and run version regression testing using those scripts. In agile development, regression testing was needed every time when a new version was published. The test scripts mainly consisted of user actions in the interface and in addition the scripts also tested the structure of the page, styles, component visibility and position in different views.

The recording of the test cases was challenging. The tests recorded based on the original test documentation needed improvements after the recording. Only the applicable functions needed testing when publishing new versions. The test automation tool was easy to use but the test cases needed a lot of planning before recording. The test documentation was improved and completed based on the features of the automation tool.

Automated UI testing was an efficient way to test the system although the recording of the test cases was time consuming and updating those tests in the future was considered a challenge. Test coverage and accuracy of the scripts was reduced, and a decision was made to test only the main functions when releasing a new version.

| | |
|---|---|
| Keywords | UI testing, test automation, Agile Development, Selenium |

## Sisällys

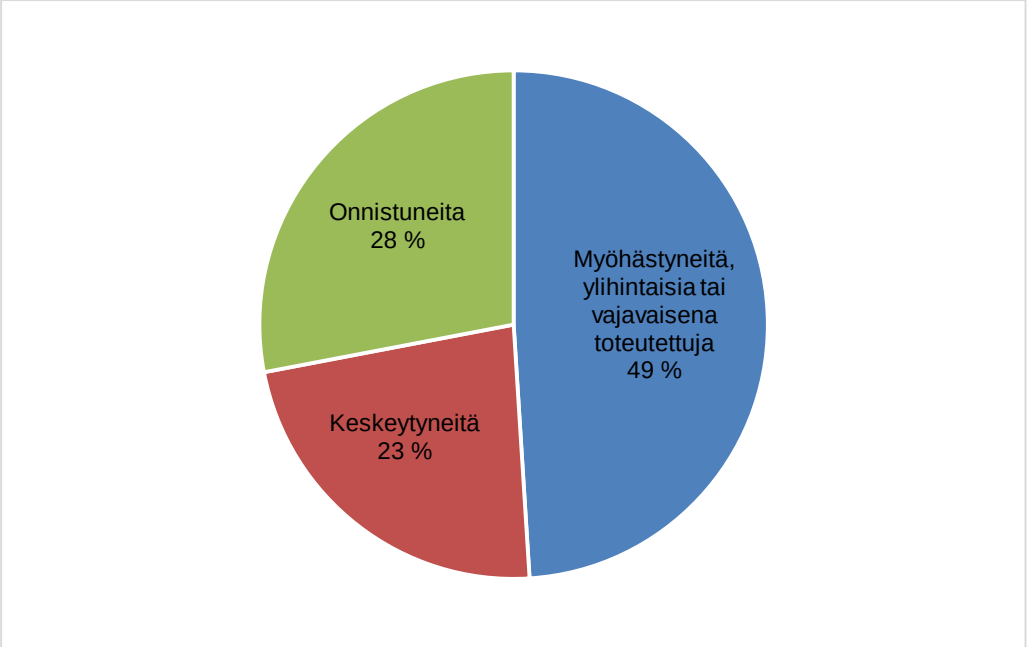
# 1 Johdanto

Tämän insinöörityön tarkoituksena oli tutkia käyttöliittymätestausta ketterän ohjelmistokehityksen menetelmillä ja toteuttaa yritykselle ohjelmistotuotteen käyttöliittymätestauksen suunnitelma ja toteutus. Suunnitelma sisälsi testausdokumentaation muutokset prosessin sopivaksi sekä parhaiden työskentelytapojen selvittämisen testiprosessin vakioimista varten. Näiden lisäksi insinöörityössä tuli suorittaa käyttöliittymätestausta määritellyllä ohjelmistolla siten, että tuotettua koodia voidaan käyttää tulevaisuudessa sovelluksen testauksessa.

Ohjelmistotestauksen tavoitteena on löytää ohjelmistossa, järjestelmässä tai koodissa olevat virheet mahdollisimman varhaisessa vaiheessa. Ohjelmistokehityksen eri työvaiheissa tapahtuu kussakin omanlaistaan testaamista joko kehittäjän tai testaajan toimesta manuaalisti tai luodun testauskoodin avulla. Testauksen automatisointi vähentää ihmistyövoiman työpanosta huomattavasti. Testien luominen vaatii tarkan määrittelyn, jotta testiautomaation tekemän testauksen voidaan luottaa olevan riittävän kattava syntyneiden virheiden paikallistamiseen. Testauksen haaste onkin siinä, että virheetöntä koodia ei ole olemassa. Testaus on käytännössä ikuinen prosessi ohjelmistokehityksen ja koodin luomisen ohessa ja sitä tulee kehittää koko ajan luodun koodin mukaisesti. Lisäksi paradoksi testauksessa on se, että on mahdotonta testata kaikkea. Esimerkiksi käyttäjän tekstikenttään syöttämien merkkiyhdistelmien täydellinen testaus ja testitapausten luominen olisi mahdotonta, sillä eri kielien merkistöt, numerot ja lisämerkit tuottavat käsittämättömän määrän merkkiyhdistelmiä. Tuolloin testeihin tuleekin määritellä testattavaksi ne testitapaukset, jotka sisältävät oikeanlaiset raja-arvot ja että ne ovat järkeviä testata käyttötapausta ajatellen.

Testaustyö ohjelmistoprojektissa on usein haastavaa. Testattavaa on paljon ja testien suunnittelu ja toteuttaminen on haastavaa jatkuvasti kehittyvässä ohjelmistossa. Testaus on kuitenkin erittäin tärkeää, sillä usein testattava ohjelmisto on myyntituote tai asiakastilaus, joka halutaan toimittaa asiakkaalle tilauksen alussa määriteltyjen aikarajojen puitteissa määrittelyjen mukaan toimivana. Jussi Pekka Kasurinen (1, s. 16 - 17) kuvaa tekstissään sitä, miten ohjelmistoprojektien kompastuskivinä ovat venyvät työvaiheet, ylioptimistisesti suunnittelut aikataulut tai rahan loppuminen. Aikataulun venyessä testaus on se työvaihe, jonka resursseja vähennetään, jotta projekti saadaan mahdollisimman pian

päätökseen ja asiakkaan käyttöön. Kuvassa 1 nähdään tutkimusten kautta saatua tietoa siitä, kuinka haasteellista on viedä ohjelmistoprojekti onnistuneesti loppuun saakka. Monesti projekti kohtaa matkan aikana ylitsepääsemättömiä haasteita, jonka takia projektia ei voida viedä loppuun saakka ja se keskeytetään.




Kuva 1. Ohjelmistoprojektien toteumat (1, s.17).

Ohjelmistoprojektin loppuunsaattaminen vaatii koko työryhmän panosta. Tehokkaat työskentelytavat yhdistettynä ammattitaitoon ovat perusedellytyksiä ohjelmistokehityksessä. Testaus koetaan monesti vähiten tuottavaksi vaiheeksi projektin aikana, joten sen vaatima työpanos pyritään usein minimoimaan. Monissa yrityksissä onkin herätty testauksen automatisointiin ohjelmiston eri tasoilla, jotta voitaisiin todeta ohjelmistoa testatuksi edes jossain määrin ja kustannuksia saataisiin vähennettyä korvaamalla miestyövoimaa koneilla.

Tämän insinöörityön yhteistyöyrityksen tarpeena oli saada automatisoitu käyttöliittymätestaus tuotteeseen, jotta testausta tekevien ohjelmistokehittäjien käyttämä aika manuaalitestauksessa saataisiin minimoitua. Lisäksi toisena tärkeänä tarpeena koettiin testitapausten testauksen määrittely ja vakiointi. Aiempi ohjeistus testitapausten testaamiseen ei ollut yksikäsitteisen selkeä, se aiheutti testauksessa vaihtelua annettaessa tes-

taajille liikaa tulkinnanvaaraa testien toteutuksessa. Automaatiotestausta ei ollut aiemmin käytetty yrityksen ohjelmistoprojekteissa, joten yrityksessä ei ollut vakiintunutta käytäntöä testauksen toteutukselle. Tavoitteena oli myös selvittää, miten työlästä automaatiotestien rakentaminen ja ylläpito olisi sekä saada näkökulma siihen, onko automaatiotestaus kannattavaa ohjelmistoprojektissa siinä mittakaavassa, minkä verran sille oli resursseja asetettu.

Insinöörityön alkuosa luku 2 käsittelee ketterää kehitystä ja testausta. Ketterän kehittämisen pääpiirteet ja tapa työskennellä luovat haasteita testauksen toteuttamiselle. Tämän jälkeen luku 3 kertoo erilaisista testauksen tekniikoista ja menetelmistä, joilla ohjelmisto voidaan testata. Luvussa 4 kuvataan käytännön työskentelyä aiheen parissa, testauksessa käytettyjä työkaluja ja esitellään esimerkkitestitapaus. Luku 5 tekee yhteenvedon insinöörityön toteutumisesta niin käytännön kuin teoriaosuuden osalta.

# 2 Ketterä ohjelmistokehitys

Ohjelmistokehityksen työskentelytapa vaikuttaa paljon testausprosessiin ja testauksen suunnitteluun. Ketterä ohjelmistokehitys ja monella ohjelmistoprojektilla käytössä oleva Scrum-menetelmä pitävät kehitysprosessin koko ajan aktiivisena ja reaktiivisena siten, että ensisijaisesti pyritään kehittämään toimivaa ohjelmistoa suoralla viestinnällä, avoimuudella ja nopealla reagoinnilla muutoksiin. Scrum on monimutkaisten tuotteiden kehittämiseen ja ylläpitoon tarkoitettu viitekehys, jonka on kehittänyt 1990-luvulla Ken Schwaber ja Jeff Sutherland. Tämä työskentelytapa vaikuttaa myös testaukseen ja sen ylläpidettävyyteen, koska muutoksia tuotteeseen ja sitä kautta uusia testitapauksia tulee jokaisen sprintin aikana.

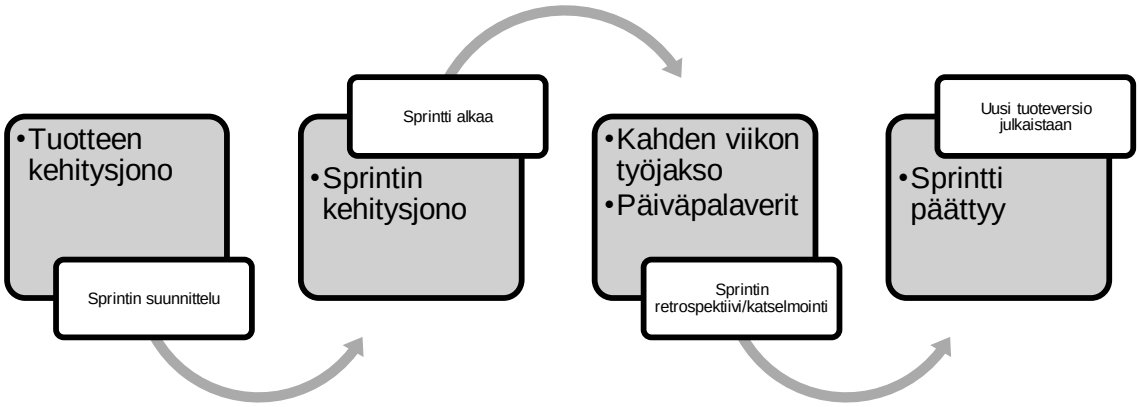
## 2.1 Ketterä ohjelmistokehitys ja Scrum

Scrum-viitekehyksen mukaan toimiminen perustuu Scrumin roolien ja sääntöjen noudattamiseen. Scrumin säännöt on kuvattu tarkkaan sen määrittelydokumentaatiossa (5), joka on saatavilla monille kielille käännettynä, myös suomeksi. Suomenkielisessä oppaassa kuvataan Scrumin pääpiirteet ja säännöt seikkaperäisesti. Scrum-projektissa on tiimi, jonka jäsenillä on käytössä kolme roolia: Tuotteen omistaja, Scrum-mestari ja Kehitystiimi. Tuotteen omistaja päättää viime kädessä kaikesta ja tekee päätökset. Scrum-

mestari varmistaa, että tiimi toimii Scrum-sääntöjen mukaisesti ja pystyy työskentelemään tehokkaasti. Lisäksi Scrum-masterin tehtävänä on johtaa päivittäisiä palavereja, johon tiimi osallistuu. Tiimi koostuu kaikista henkilöistä, josta ovat tekemässä projektia. Tiimi toimii hyvässä yhteistyössä ja yhteisvastuullisesti siten, että tehtävät jaetaan kunkin tiimin jäsenen vahvuuksien ja osaamisen mukaan.

Ohjelmistoa kehitetään yhdestä neljään viikkoa kestävissä kehitysjaksoissa eli sprinteissä, jonka aikana tuotetta kehitetään täydellisemmäksi ja valmiimmaksi. Jokaisen sprintin päätteeksi julkaistaan entistä parempi, valmiin tuotteen määritelmän omaava ja toimivampi versio tuotteesta. Jos kuitenkin koetaan, että sprintin aikana tehty kehitystyö on jäänyt kesken tai ohjelmistossa huomataan virheitä, uuden version julkaisua voidaan lykätä tuotteen omistajan päätöksellä seuraavan sprintin jälkeen. Kullekin sprintille valitaan sprintin suunnittelupalaverissa tuotteen kehitysjonosta ja edellisen sprintin työjonosta mahdollisesti jääneitä töitä tuoteomistajan päätöksellä työn alle niiden prioriteetin mukaan. Tiimin jäsenet sitoutuvat sprintin aikana heille osoitettujen ominaisuuksien valmistumiseen, toisaalta myös sprintin työmäärä lyödään lukkoon sprintin alussa, jolloin kukin jäsen tietää työmääränsä tulevan sprintin ajan. Sprintin aikana työskentelyä ja sen etenemistä katselmoidaan päivittäisten tiimipalaverien avulla, jonka aikana kukin tiimin jäsen kertoo lyhyesti edellisen työpäivän sisällön, sen mitä aikoo tehdä tänään ja onko työn tekemiselle jotain ongelmia. Sprintin lopussa järjestetään katselmointipalaveri, jossa tiimin jäsenet esittelevät tuoteomistajalle aikaansaannoksensa. Katselmoinnin jälkeen sprintti päättyy ja uusi alkaa. Kunkin sprintin päättyessä pyritään julkaisemaan toimiva versio tuotteesta, joka on edellistä parempi ja täydellisempi.

Sprintin elinkaari on kuvattu yksinkertaistettuna kuvassa 2 Scrumin kuvauksen mukaisesti.




Kuva 2. Sprintin eri vaiheet tuotteen kehitysjonosta sprintin aloitukseen ja tuoteversion julkaisuun asti (1, s. 28; 5).

Jokainen sprintti on rakenteeltaan samanlainen. Seuraava alkaa kun edellinen päättyy. Kullekin Scrumin määritelmässä kuvatulle palaverille on määritetty sisältö ja kesto, jota tulee noudattaa. Sprintin kehitysjonoon viedyt ominaisuudet toteutetaan siten, että kehitysjonoon ei saa tehdä muutoksia tai lisäyksiä sprintin aikana, jotta sprintin päämäärä ja tavoite voidaan saavuttaa suunnitellusti.

## 2.2 Testaus ketterässä kehityksessä

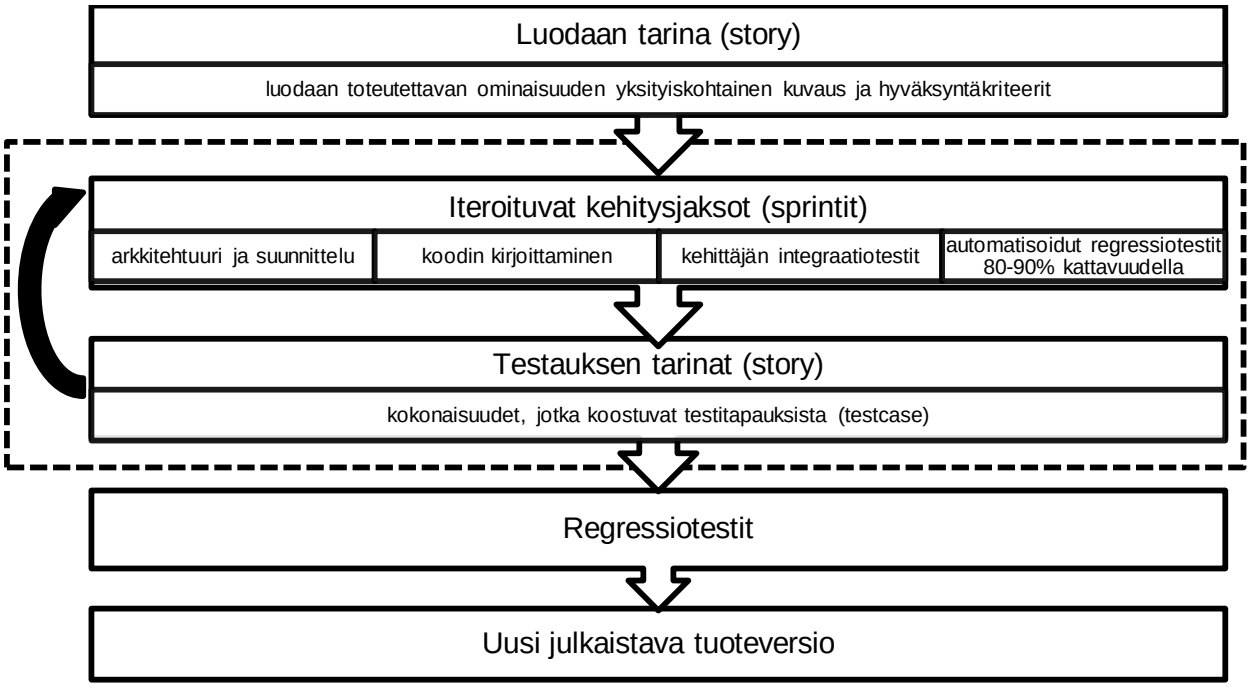
Testaus ketterässä kehitystavassa on haasteellista. Sainio (2, s.98 - 99) ja Vuori (5) mainitsevat materiaalissaan vahvuuksien lisäksi myös paljon haasteita ja ongelmia. Samoja ongelmia tuli vastaan myös insinöörityön käytännön osuutta tehdessä. Suuri vastuu testauksessa on tiimin kehittäjäjäsenillä: heidän tulee itse kirjoittaa yksikkö- ja integraatiotestit, testata toteuttamansa uudet ominaisuudet ja varmistaa, että ne toimivat ilman ongelmia. Tämä menettelytapa ei kuitenkaan poista sitä todennäköisyyttä, etteikö virheitä olisi. Tämän vuoksi tuote tulee testata myös muidenkin kuin kehittäjien toimesta. Käyttöliittymätestaus on yhdistelmä järjestelmä-, hyväksyntä- ja regressiotestausta eli testataan, että järjestelmä toimii kokonaisuudessaan oikein uusien ominaisuuksien kanssa ja että myös vanhat toiminnot toimivat.

Testauksen suunnittelussa kehittäjien ja testaajien tulee tehdä tiivistä yhteistyötä tarinan (story) ja sprintin suunnittelupalaverissa. Tarinat ovat lyhyitä, yksinkertaisia kuvauksia uudesta toiminnasta käyttäjän näkökulmasta eli miten käyttäjä haluaa järjestelmän toi-

mivan saadakseen halutun lopputuloksen. Kehitettävien ominaisuuksien käyttäjätarinoihin tulee kirjata mahdollisimman tarkasti uusien ominaisuuksien hyväksyntäkriteerit. Näistä kriteereistä muodostuvat testitapaukset, joiden avulla uusia ominaisuuksia voidaan testata.

Näitä hyväksyntäkriteerejä voidaan testata joko käsin tai automatisoidusti, riippuen uuden ominaisuuden toteutuksesta. Toteutettavien uusien ominaisuuksien kokonaisuudet tulee pitää riittävän pieninä, jotta niille voidaan kirjoittaa automatisoidut testit tai tehdä manuaalitestaus sprintin aikataulun puitteissa. Sprintin lopussa julkaistava tuoteversio tulisi olla testattu kaikkien sprintin kehitysjonoon otettujen uusien ominaisuuksien osalta mutta iteratiivinen työskentelytapa asettaa suuria haasteita automatisoitujen testien rakentamiselle sprintin aikana. Tuote kehittyy mahdollisesti isoillakin hyppäyksillä eteenpäin, joten yhden sprintin aikana on mahdotonta rakentaa kaikille uusille ominaisuuksille automaatiotestejä ennen sprintin päättymistä. Usein testiautomaatiota käytetäänkin siihen, että sprintin aikana testataan vanhojen ominaisuuksien toimimista ja uusien ominaisuuksien täydellisemmät testit tulevat jäljessä ja kehittyvät sitä mukaa, kun testattava ominaisuus hahmottuu testaajalle paremmin. Ketterässä testauksessa voidaankin puhua ad hoc - testauksesta, joka käytännössä tarkoittaa testausta ilman tarkkaa suunnitelmaa. Uusien ominaisuuksien testaus etenkin alkuvaiheessa on kokeilevaa testausta.

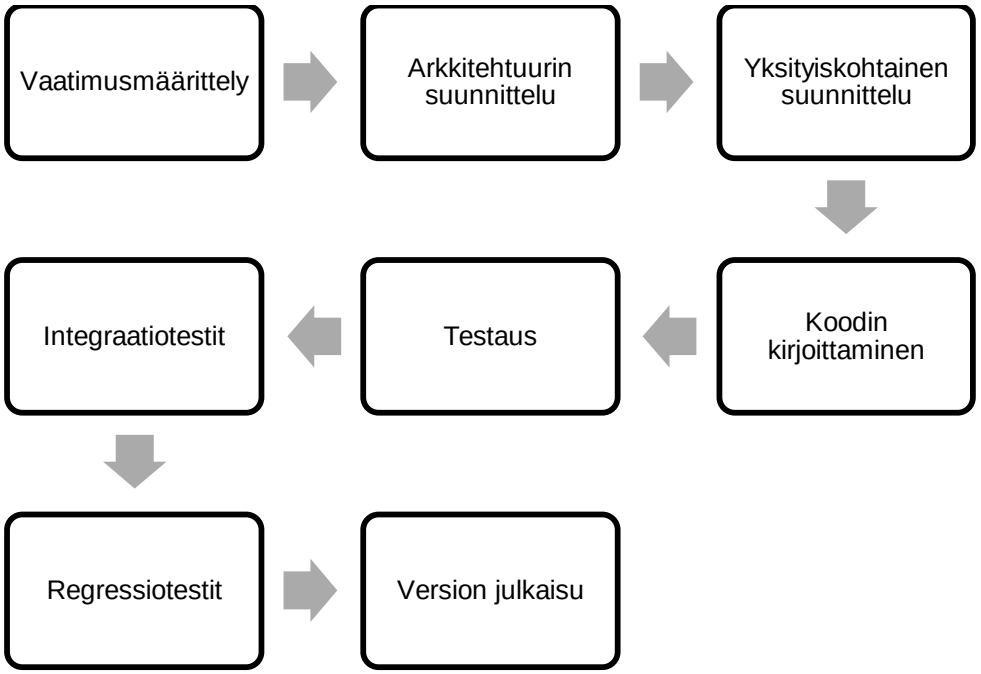
Walkden (7) esittelee artikkelissaan ketterän kehityksen lähestymistavan kehitettävien toimintojen tarinoiden testaukseen ja prosessiin etenemiseen sprinttien aikana, joka on kuvattuna kuvan 3 kaaviossa.




Kuva 3. Prosessikuvaus Scrum-kehityksen ja testauksen kannalta sprintin eri vaiheissa (6).

Ketterässä kehityksessä luodaan tarinan (story) kautta kehitettävä ominaisuus, jolle määritellään yksityiskohtaisesti määrittelyt sekä hyväksyntäkriteerit. Tämän jälkeen kehitettävä ominaisuus otetaan työn alle sprintin tai sprinttien ajaksi, jonka aikana suunnitellaan ja kehitetään ominaisuutta samalla, kun sovelluksen toimivuutta varmistetaan aiemmin luoduilla automaatiotesteillä. Kehittäjä vastaa sprintin aikana tapahtuvat integraatiotestauksesta siten, että uusi ominaisuus toimii moitteettomasti olemassa olevan koodin kanssa kehittäjän omassa kehitysympäristössä. Kun kehitettävä ominaisuus on valmis edes joiltakin osin, luodaan siitä testausta varten tarina, johon koostetaan testattavat testitapaukset. Testitapaukset ovat käytännössä alkuperäisen määrittelyn yhdistämistä hyväksyntäkriteerien kanssa käytännön testausmenetelmät muistaen. Testausprosessin aikana tulee huomioida se, että sprintin aikana kaikkea ei voida testata automaatiotestauksen avulla. Uudet, kehitettävät ominaisuudet tulee testata käsin ja julkaistava versio testataan ennen julkaisua regressiotestein, jotta varmistutaan vanhojen toimintojen toimivuudesta.

Perinteinen lähestymistapa ohjelmiston kehitykseen ja testaukseen kuvataan seuraavassa kuvan 4 kaaviossa.




Kuva 4. Prosessikuvaus kehitettäessä ja testattaessa uusia ominaisuuksia ei-ketterillä kehitysmenetelmillä (6).

Kehitettävä ominaisuus suunnitellaan ja dokumentoidaan alussa huolellisesti vaatimusmäärittelyjen avulla. Tämän jälkeen kehittäjä ryhtyy kehittämään ominaisuutta dokumentaation mukaisesti ja kehittää ominaisuutta niin kauan, kunnes se on valmis. Tämän jälkeen ominaisuus testataan huolellisesti ja testausprosessin jälkeen julkaistaan versio. Jos testeissä huomataan virheitä, prosessi alkaa alusta ja uusi versio julkaistaan vasta, kun ohjelmistoa on kehitetty eteenpäin eli korjattu virheet ja näiden virheiden osalta kaikki testit ovat menneet läpi. Tällainen kehitystapa on helppo ja selkeä testauksen kannalta. Testaaja pystyy osallistumaan alkuvaiheen määrittelyihin aktiivisesti siten, että testitapausten määrittelyt koko ohjelmiston osalta voidaan suunnitella jo alkuvaiheessa. Kun ohjelmisto on valmis testattavaksi, testejä aletaan luomaan ja ajamaan. Testaus on tämän kaltaisessa kehityksessä aivan oma työvaiheensa, johon käytetään sen verran aikaa kuin on tarvetta. Tällaisessa tavassa työskennellä kehitys voi olla kuitenkin melko hidasta ja vaiheittaista, koska koko prosessin tulee edetä alusta loppuun ennen kuin se voi taas alkaa alusta.

# 3 Käyttöliittymän testaus, menetelmät ja tekniikat

Sovellusta testatessa tulee määritellä, mitä sovelluksen osaa testataan sekä miten ja millaista testaamista järjestelmälle tehdään. Valittavat menetelmät riippuvat paljolti siitä, missä työvaiheessa ohjelmistokehityksen projektissa ollaan. Menetelmiä ja lähestymistapoja on monia. Seuraavissa luvuissa esitellään insinöörityön ohjelmiston kehitysvaiheeseen liittyvät käytössä olevat testauksen käytännöt. Yleisesti testausmenetelmää valittaessa tulee ensin määritellä, onko kyse dynaamisesta vai staattisesta testauksesta tai onko mahdollista käyttää molempia. Tämän jälkeen mietitään tarkemmin, mitä testauksen tekniikoita ja lähestymistapoja käytetään. Kasurinen (1, s. 65 – 70), Lehto (2, s. 54 – 80) ja Pohjolainen (3, s.18 – 22) käsittelevät näitä teoksissaan. Seuraavissa luvuissa esitellään käyttöliittymätestauksen sekä testausmenetelmien ydinasioita.

## 3.1 Staattinen ja dynaaminen testaus

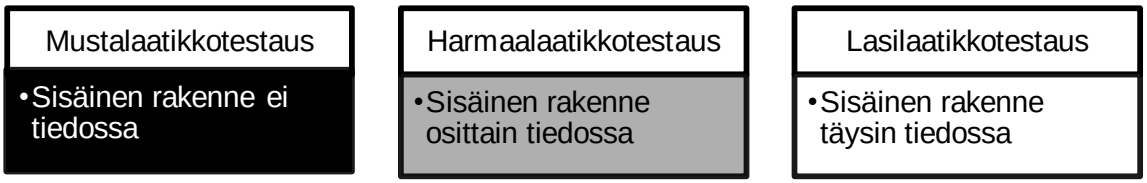
Testauksen tehokkuutta ja laatua ajatellen on tärkeää, että testaaminen aloitetaan mahdollisimman aikaisessa vaiheessa eli jo silloin, kun toteutettavia ominaisuuksia määritellään tai suunnitellaan tuotteen kehitysjonon tai sprintin suunnittelupalaverissa. Käytännön vaiheessa oleva testaus käsittää testauksen ohjelmaa ajettaessa kehitysympäristössä tai testattaessa julkaistavaa versiota. Näitä erilaisia testaustapoja kutsutaan staattiseksi tai dynaamiseksi testaukseksi. Sainio (2, s. 54 – 62) käsittelee staattista ja dynaamista testausta kattavasti.

Staattisen ja dynaamisen testauksen erot ovat selkeät. Staattisessa testauksessa katselmoidaan eli testataan testattavaa sovellusta siihen liittyvän dokumentaation, ohjelmistomallin ja määrittelyn perusteella ilman, että ohjelmaa varsinaisesti suoritetaan. Staattista testausta voidaan suorittaa suunnitteluvaiheessa tai sprintin aikana joko tarinoita luotaessa tai koodikatselmuksen kautta. Dokumenttien lisäksi tuotettua koodia voidaan tarkastella ja etsiä siitä virheitä, esimerkiksi muuttujia ilman arvoa tai virheitä koodin syntaksissa. Staattista testausta on ennaltaehkäisevää testausta, sillä virheet pyritään löytämään jo suunnitteluvaiheessa ennen kuin varsinaista ohjelmointityötä on suoritettu.

Dynaaminen testauksessa testataan sovellusta sitä käytettäessä. Käyttöliittymätestaus on dynaamista testausta, lähestymistapoja voi olla monia riippuen testaajan ammattitaidosta ja aiemmasta kokemuksesta. Testauksessa voidaan karkeasti noudattaa kahta tekniikkaa: lasilaatikkotestaus (glass-/white-box testing) ja mustalaatikkotestaus (black-box testing). Lisäksi voidaan käyttää näiden kahden strategian yhdistelmää, jota sanotaan harmaalaatikkotestaukseksi (gray-box testing). Nämä laatikkotestaustekniikat esitellään seuraavassa kappaleessa 3.3 Laatikkotestaustekniikat.

## 3.2 Laatikkotestaustekniikat

Dynaamisessa testauksessa käytettävät laatikkotekniikat eroavat toisistaan kuvassa 5 kerrotulla tavalla.




Kuva 5. Laatikkotestauksen eri käsitteet (2, s. 61)

Rakenteellisesta lasilaatikkotestauksessa testaus tapahtuu pintaa syvemmältä siten, että testaajalla on tiedossa kooditasolta asti ne tapahtumat, joita tulee testata. Testaus perustuu ohjelman rakenteeseen siten, että testitapaukset testaavat jopa yksittäisen funktioiden tasolle asti ohjelman eri haaroissa sen toimintaa ja logiikkaa. Lasilaatikkotestauksessa halutaan varmistua siis siitä, että ohjelman tapahtumat kooditasolla on toteutettu oikein. Lasilaatikkotestauksen avulla ohjelmasta saadaan havaittua virheet käyttöliittymäkerroksen alta businesslogiikan puolelta. Testejä luotaessa testaajan on tunnettava hyvin ohjelmiston rakenne ja käytetty ohjelmointikieli, jotta lasilaatikkotestausta pystytään käyttämään strategiana. Tätä testaustapaa käytetään yleensä ohjelmiston kehitysvaiheessa sprintin aikana rakennettaessa uusia toiminnallisuuksia ja kirjoitettaessa koodia. Käytännössä se tarkoittaa kehittäjän luomien yksikkötestien ajamista ja testaamista kehitysympäristössä.

Funktionaalisessa eli mustalaatikkotestauksessa testaajalle annetaan testattavaksi ohjelma, joka testataan sovelluksen määritysten perusteella. Testien läpimenoa arvioidaan

järjestelmän odotetun tai halutun vastauksen perusteella. Testaaja testaa järjestelmässä olevan toiminnon, esimerkiksi syöttää tekstikenttään käyttäjän etunimen ja tallentaa sen, ja arvioi testin läpimenoa sen mukaan, mitä ohjelmiston määrittelydokumenttiin on kirjoitettu ohjelmiston toimimisesta käyttäjän etunimen tallennustilanteessa. Jos nimi tallentui oikein ilman virheilmoituksia ja käytön estävää virhetilannetta, testin voidaan arvioida menneen läpi. Testaus käyttöliittymätasolla vaatii testaajalta vähemmän järjestelmän tuntemusta ja koodin ymmärtämistaitoja sillä esimerkiksi tietokanta- ja kooditasolla tapahtuvia virheitä ei pystytä tällä menetelmällä havaitsemaan, jos ne eivät aiheuta käyttäjälle eli testaajalle järjestelmän käytössä virhetilanteita.

Harmaalaatikkotestaus on näiden kahden erilaisen testausstrategian yhdistelmä, joka on kattavin tapa testata etenkin insinöörityön aiheeseen liittyviä selainpohjaisia sovelluksia. Harmaalaatikkotestaus ottaa kantaa järjestelmän taustalla tapahtuviin toimintoihin (back-end) kuin myös käyttöliittymätasolla (front-end) oleviin toimintoihin ja näiden molempien tasojen yhteistyöhön. Näitä molempia tasoja ja niiden yhteistyötä sekä liittymiä muihin laitteistoihin ja käyttöjärjestelmiin testaamalla testausprosessista saadaan riittävän kattava, koska pelkästään käyttämällä joko lasilaatikko- tai mustalaatikkotestausstrategiaa testaus jää liian suppeaksi eikä huomioi riittävän tarkasti järjestelmässä toisaalla olevia testattavia osa-alueita.

### 3.3 Tutkiva testaus ja regressiotestaus

Edellisessä luvussa mainittujen tekniikoiden lisäksi sekä Kasurinen (1, s. 74), Sainio (2, s. 80 – 82) että Pohjolainen (3, s. 22) mainitsevat kokeilevan eli tutkivan testauksen (explotarory testing) sekä regressiotestauksen. Nämä testausmenetelmät kuvaavat parhaiten insinöörityön aikana tehtyä testausta ja liittyvät ketterään kehitystapaan, josta kerrotaan tarkemmin seuraavassa kappaleessa.

Tutkivassa testauksessa luodaan tai muutetaan testaussuunnitelmaa samalla, kun testausta tehdään. Tutkivassa testauksessa luotetaan siihen, että testaaja oppii testattavasta ympäristöstä jatkuvasti lisää testausprosessin aikana ja tämän kautta pystyy kehittämään testaussuunnitelmaa. Tutkiva testaus perustuu suunnitelmaan, jota noudatetaan ja johon tarpeen mukaan lisätään testausprosessin edetessä esimerkiksi testauk-

sessa huomattuja uusia testattavia ominaisuuksia, joita ei ole aiemmin huomattu dokumentoida. Tutkivassa testauksessa kokeneesta testaajasta on hyötyä, sillä testaustapa vaatii kurinalaisuutta ja johdonmukaisuutta vapaamuotoisesta toteutustavastaan huolimatta. Tutkiva testaus jakaa mielipiteitä vapaamuotoisuutensa takia, joko sitä arvostetaan tai sitä pidetään ajan ja resurssien tuhlauksena. Kuitenkin tämä testaustapa on melkeinpä välttämätön osa ketterän kehitys ohjelmistotestausta, koska uusia ominaisuuksia kehitetään jatkuvasti ja niiden testausta tulee suorittaa jatkuvasti sprinttien aikana, vaikka ne olisivat vielä osittain keskeneräisiä. Tutkivan testauksen avulla on mahdollista löytää työn alla kehitettävästi ominaisuudesta uusia kehityskohtia tai jopa muutoksia määrittelyyn.

Regressiotestausta tulisi suorittaa jatkuvasti eli aina, kun jotain muuttuu ohjelmassa ja sen komponenteissa. Sen avulla varmistutaan siitä, että ohjelma toimii edelleen muutosten jälkeen. Regressiotestauksella kuvataan kaikkia testaustoimenpiteitä, jotka suoritetaan sitä varten, että varmistutaan uuden version toimivuudesta. Regressiotestauksella varmistetaan kehitettävän sovelluksen laatu ja esimerkiksi se, että jo korjatut virheet eivät esiinny uudestaan sovelluksessa komponenttien muutosten jälkeen. Sprinttien aikana tehdyt virheenkorjaukset ja kehitystyö vaativat regressiotestejä välittömästi korjauksen jälkeen ja aina ennen uuden version julkaisemista (kuva 2).

### 3.4 Käyttöliittymätestaus ja sen automatisointi

Käyttöliittymän merkitys kaikkien sovelluksien käyttökokemuksen kannalta on merkittävä. Selkeästi rakennettu, käytettävyydeltään hyvä käyttöliittymä on sovelluksen oikeanlaisen käytön kannalta tärkeää. Käyttäjän tulee kokea, että ohjelmaa on helppoa ja loogista käyttää. Arto Kangas käsittelee kandidaatintyössään (4) reaaliaikaista käyttöliittymätestausta ja kuvailee sen hyötyjä ja haasteita. Käyttöliittymää voidaan testata sen käytettävyyden kannalta mutta sen avulla voidaan myös tehokkaasti testata se, että ohjelma toimii karkealla tasolla ja vastaa määrittelyjä.

Käyttöliittymätestauksen avulla voidaan todentaa, että järjestelmän perustoiminnot toimivat. Määrittelydokumentaatioon perustuva käyttöliittymätestaus testaa tehokkaasti järjestelmälle määritellyt käyttötapaukset käyttöliittymän toimiessa rajapintana taustalla tapahtuville toiminnoille. Käyttöliittymätestauksen avulla ei kuitenkaan pystytä kattamaan

koko ohjelman testausta, koska taustalla tapahtuvia kooditason virheitä ei voida välttämättä todentaa käyttäjän toimien kautta. Sovelluksen taustalla tapahtuvia toimintoja tulee testata yksikkö- ja integraatiotestein, jotta testaus olisi kattavaa.

Käyttöliittymää testatessa tulee määritellä, mitä kaikkea on oleellista testata. Kaikkea näkyvää tuskin on järkevä testata kaikissa näkymissä jokaisen käyttäjän toimen ja näkymän vaihtumisen jälkeen, joten testien tekijän ja laatijan on tärkeää hahmottaa se, mitä kaikkea tulee testata, jotta järjestelmän toimivuus saadaan varmistettua. Lisäksi liian pikkutarkka testaaminen vie aikaa sovelluksen ydintoimintojen testaamiselta, joten testien hyvä suunnittelu ja valmistelu ovat tärkeitä kustannustehokkuuden ja ajankäytön kannalta. Käyttöliittymän kannalta oleellista on testata se, että käyttäjälle näytetään tarvittavat komponentit (menuvalikot, painikkeet, navigointi), sivujen ja näkymien välillä liikkuminen toimii oikein sekä todennetaan tiedon siirtyminen molempiin suuntiin eli se, että tietokantaan tallentuu tietoa ja sieltä haetaan tietoa oikein. Käyttöliittymän testaaminen ei ole monimutkaista mutta kattavien testitapauksen määrittely ja testaaminen on aikaa vievää. Tulee kuitenkin muistaa, että automaatio ei poista manuaalisten testien tarvetta, eikä tarvetta testaajille, jotka osaavat analysoida testien tuloksia, eivätkä automatisoidut testit ole verrattavissa manuaalisiin testeihin. Automaatiota tuleekin käyttää manuaalisen testauksen jatkeena silloin, kun sen avulla voidaan tehdä jotain, mihin ei manuaalisesti pystyttäisi. (2, s. 120.)

Käyttöliittymän automaatiotestauksen avulla testausprosessi saadaan automatisoitua siten, että järjestelmän perustoiminnot saadaan testattua ja siten varmistettua, että ohjelma toimii oikein ainakin nauhoitettujen testitapausten osalta. Testien toistettavuus tuo testausprosessiin laatua ja vakautta, koska testejä voidaan ajaa milloin vain ilman, että se vaatii erillistä testaajaa suorittamaan testausta. Testien toistettavuus on tärkeää etenkin ketterässä ohjelmistokehityksessä, jossa kehittäminen tapahtuu nopeasti ja uusia ominaisuuksia luodaan jatkuvasti. Sprinttien aikana voi olla tarpeellista suorittaa automaatiotestausta useaan kertaan regressiotestauksen periaatteella, jotta voidaan todeta järjestelmän eheys uutta ominaisuutta kehitettäessä.

# 4 Tuotteen käyttöliittymätestauksen suunnittelu ja toteutus

Insinöörityön tarkoituksena oli toteuttaa yritykselle sovelluksen testaus käyttöliittymätasolla. Kyseessä on SaaS -palveluna (Software as a Service) toimitettava henkilöstönhallinnan sovellus, jota asiakkaat käyttävät internetselaimen kautta verkossa. Sovellus sisältää seuraavat kokonaisuudet:

- henkilöstönhallinta (henkilö-, työsuhde, ja palkkatiedot)
- lomien ja poissaolojen hallinta
- kehityskeskustelut
- raportointi.

Sovelluksen käyttö perustuu käyttäjärooleihin ja niiden oikeuksiin, oletusrooleja järjestelmässä on seitsemän kappaletta. Henkilöstöhallinnan sovelluksissa käsitellään arkaluontoisia tietoja (henkilötiedot, työsuhdetiedot, palkkatiedot), joiden näkyvyyden tulee olla sallittu vain tietyille rooleille tieturvan vuoksi.

Seuraavissa luvuissa kuvataan insinöörityön käytännön osuus alkuvaiheen suunnittelusta toteutukseen ja matkan aikana ilmenneisiin haasteisiin. Käytännön työtä kuvatessa työssä on esitelty yksi esimerkkitestitapaus kuvankaappauksilla havainnollistaen.

## 4.1 Lähtötilanne

Sovelluksen testaus suoritettiin aiemmin kahdella tavalla:

- Ohjelmoijat testasivat omaa koodiaan paikallisesti sitä kehittäessään yksikkö- ja integraatiotestein.
- Ennen versiopäivityksen julkaisua ohjelman kaikki toiminnot testattiin käsin kaikilla selaimilla saatavilla olevien resurssien puitteissa.

Tämä työskentelytapa koettiin puutteelliseksi testauksen laadun osalta sekä resursseja aikaa vieväksi, sillä käsin järjestelmän testaus kaikilla tuetuilla selaimilla ennen version julkistamista vei aikaa yleensä viikon verran yhdellä testaajalla. Testaukseen kulunut aika koettiin kohtuuttoman pitkäksi, koska sama aika oli poissa kehittäjän kehitystyöhön varatusta työajasta.

Käsin tapahtuvan testauksen runkona oli testidokumentaationa toimiva Excel-taulukko, johon oli kirjattu moduulikohtaisesti käyttötapaukset yhdellä virkkeellä kuvattuna (esimerkki taulukossa 1). Testitapausten ylläpito ja päivitys etenkin sovelluksen uusien ominaisuuksien myötä ei ollut järjestelmällistä ja riittävän tarkkaa, joten toimintojen perusteellinen testaus etenkin uudelta testaajalta oli dokumentaation pohjalta mahdotonta tehdä riittävän tarkasti.

Taulukko 1. Vanhan, alkuperäisen testausdokumentaation esimerkki

| **Rooli** | **Toiminto/testattava asia** |
|---|---|
| HR | Työntekijälomakkeen tiedot sekä toiminnot (muokkauslomake) |

Taulukossa 1 esimerkkitestitapauksella katettiin aiemmin kuvaus siitä, miten työntekijälomakkeen tietoja testattiin muokkaustoimintoa käyttäen HR- roolissa. Tällainen dokumentointitapa ja kuvaus olivat riittämättömiä ja liian ylimalkaisia, koska testaajalle jäi tässä tapauksessa hyvin paljon tulkinnan varaa siitä, mitä kenttiä ja miten hän työntekijälomakkeen muokkaustoimintoa testaa.

## 4.2 Testauksen suunnittelu ja tavoite

Työn tärkeimmäksi tavoitteeksi määriteltiin testitapausten nauhoittaminen mahdollisimman kattavasti ja perusteellisesti siten, että nauhoitettavien testien laatu oli määrää tärkeämpi. Sen ohessa testidokumentaatiota tuli päivittää ja mahdollisesti parantaa ja miettiä parempaa tapaa hallita testitapauksia ja niiden dokumentointia ja ylläpitoa.

Ennen testauksen aloittamista järjestelmään ja sen toimintoihin tuli tutustua. Haastavaksi tilanteen teki se, että sovelluksen käytöstä ei ollut kunnollista käyttöohjetta eikä testitapauksia ollut kuvattu niin kattavasti, että odotettu tulos testitapausten osalta olisi täysin selvä ja yksikäsitteinen. Järjestelmän toimintojen läpikäynti alkuvaiheessa paljasti sen,

miten puutteellista dokumentointi testitapausten osalta oli. Testitapauksia suunnitellessa testituloksen perustuivat lähinnä olettamuksiin siitä, mikä odotettu tulos käyttäjän toiminnolla oli.

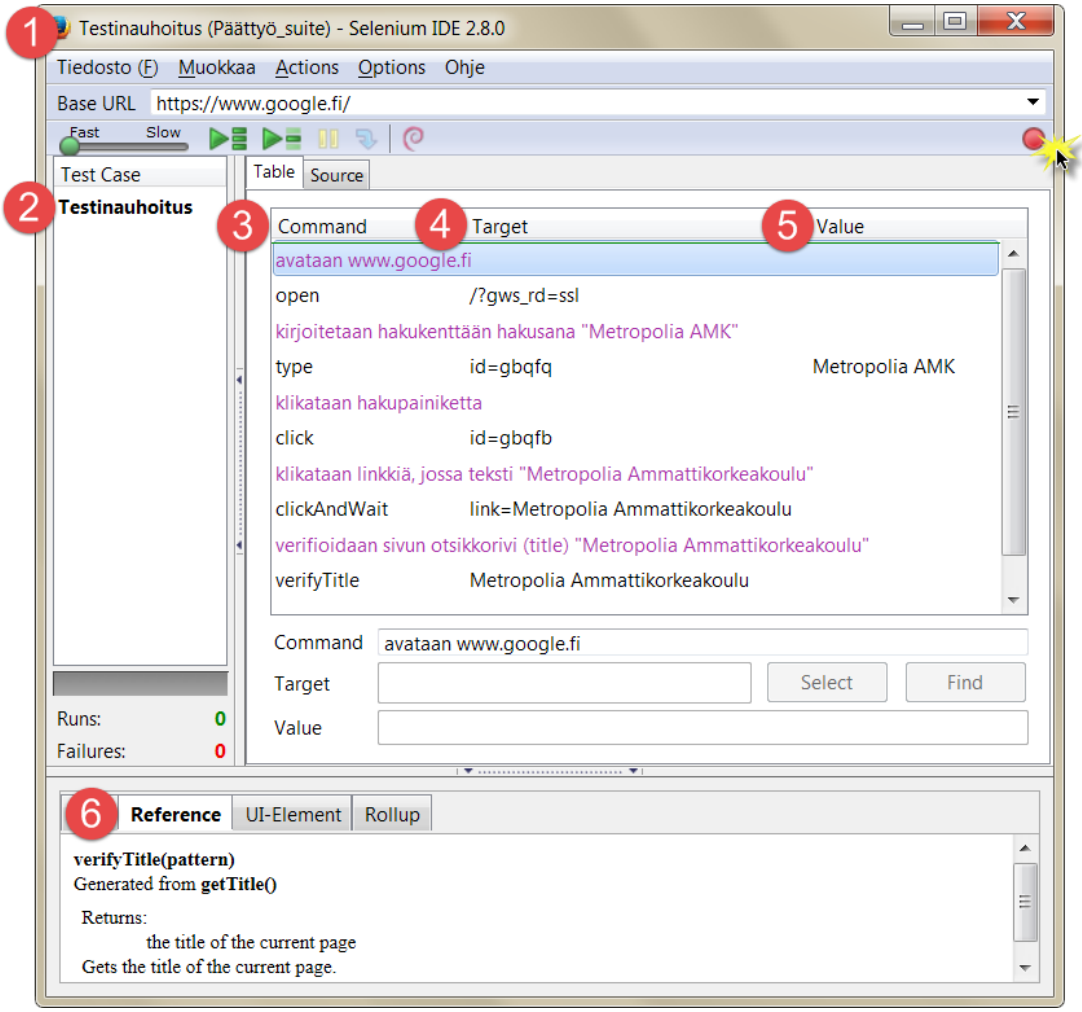
Testauksen ja etenkin testausohjelmiston osalta kaikki testitapaukset piti suunnitella ja kirjata uudelleen, jotta osoitetulla testaustyökalulla päästäisiin parhaaseen mahdolliseen testitulokseen. Testitapauksia ei ollut kirjattu riittävän perusteellisella tasolla siten, että testitapauksista kävisi vaiheittain ilmi, mitä haluttiin testata ja millä tavalla.

## 4.3 Toteutus

Testausohjelmisto insinöörityötä varten oli Selenium IDE (Integrated Development Environment), joka on Mozilla Firefox -internetselaimen ilmainen lisäosa. Selenium on yksi laajimmin käytettyjä ilmaisohjelmia www-sovellusten käyttöliittymätestauksessa. Seleniumin viimeisin päivitetty stabiili version on 2.8.0, joka oli käytössä testauksen aikana.

Selenium IDE:n avulla käyttäjä pystyy nauhoittamaan käyttöliittymässä tapahtuvia käyttäjän toimia ja toistamaan ne joko yksittäisenä testitapauksena (case) tai usean testitapauksen kokoelmana (suite). Nauhoitustilanteessa ohjelma muodostaa nauhoituksesta skriptin (script) HTML-kielellä, jonka voi halutessaan viedä (export) ja tallentaa muille kielille. Ohjelma tukee HTML:n lisäksi seuraavia kieliä: Ruby, Python, Java ja C#. Nauhoitettuja skriptejä voidaan muuntaa ja viedä verkkoajurin (webdriver) sekä sovelluskehittimien käyttöön.

Selenium IDE:n käyttöliittymä on helppokäyttöinen. Käyttöliittymä ja sen eri osat on esitelty seuraavassa kuvassa (kuva 6).




Kuva 6. Selenium IDE käyttöliittymä ja sen eri osat

1) Selainikkunan otsikkoriviltä voidaan nähdä testitapauksen nimi (Testinauhoitus) ja minkä nimiseen testitapausten kokoelmaan (Päättötyö_suite) tapaus kuuluu

2) Vasemman reunan listassa näkyy mahdollinen lista testitapauksista. Vahvennettuna oleva testitapauksen nimi on se, jota on näkyvissä taulukkonäkymässä.

Selenium luo nauhoituksesta taulukon (table), jonka rivit kuvaavat nauhoituksen eri vaiheita eli käyttäjän toimia käyttöliittymässä. Käyttöliittymän taulukossa on kolme saraketta:

3) Command, johon valitaan komento, joka suoritetaan. Komento voi olla käyttäjän toiminto käyttöliittymässä (esimerkiksi click, type) tai komento, joka suoritetaan

käyttäjän toimintojen ohessa (esimerkiksi verifyText, assertVisible, echo). Komentoja on valittavissa kymmeniä, joiden avulla käyttöliittymää voi testata laajasti. Komentoihin voidaan liittää mm. javascriptin suorittamista, joka helpottaa monimutkaisempien sivujen toiminnallisuuksien testaamista.

4) Target, johon valitaan käyttöliittymässä oleva kohde, johon komento suoritetaan. Sovellus pystyy tunnistamaan sivulla olevia HTML-sivun elementtejä elementin nimen, linkin tai tunnisteen (id:n) lisäksi CSS- tiedoston sisältämien määrittelyjen, xpath- polun tai DOM- mallin mukaisella tunnistuksella. Käyttöliittymästä voidaan siis etsiä mikä tahansa elementti, joka on näkyvissä sivulla ja sivun lähdekoodissa ja suorittaa sille komentoja.

5) Value-kenttään on mahdollista syöttää arvo, jota komennon yhteydessä halutaan käyttää kyseiselle elementille, esimerkiksi komento type vaatii tekstin, joka kirjoitetaan kohde-elementtiin.

Käyttäjän apuna on lisäksi alareunan infoalue, jonka Reference-välilehdellä on mahdollista saada valittuna olevaan komentoon määrittelytiedot. Log-välilehdellä ohjelma kirjoittaa skriptien ajon aikana tapahtumalokia, josta voidaan havaita mahdolliset virhetilanteet punaisella värillä korostettuna. Skriptin selventämiseksi käyttäjä voi kirjoittaa komentojen väliin kommentteja (korostettuna violetilla värillä), jotka helpottavat skriptin tulkintaa eri vaiheissa. Koodin kommentointia pidetään yleisesti hyvänä käytäntönä, jotta muutkin kuin koodin luoja ymmärtävät koodin suorituksen eri vaiheita

Selenium IDE:n käyttöliittymäkuvassa näkyvä testitapaus on seuraavassa taulukossa 1. Sovellus muodostaa kaikista testitapauksista automaattisesti HTML-taulukon, joka on nähtävissä Source-välilehdellä.

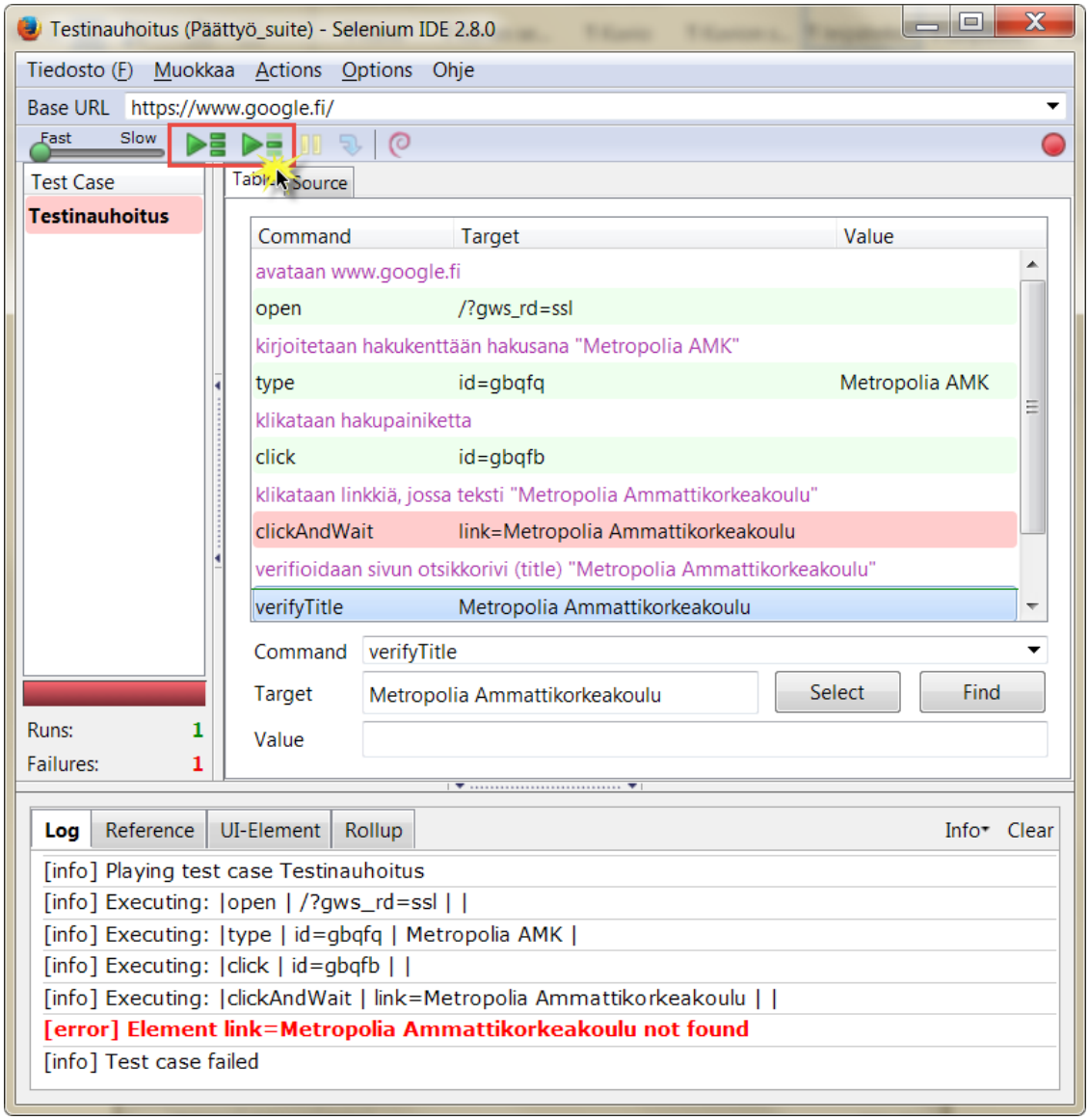
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en">
<head profile="http://selenium-ide.openqa.org/profiles/test-case">
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
<link rel="selenium.base" href="https://www.google.fi/" />
<title>Testinauhoitus</title>
</head>
<body>
```

```
<table cellpadding="1" cellspacing="1" border="1">
<thead>
<tr><td rowspan="1" colspan="3">Testinauhoitus</td></tr>
</thead><tbody>
<!--avataan www.google.fi-->
<tr>
              <td>open</td>
              <td>/?gws_rd=ssl</td>
              <td></td>
</tr>
<!--kirjoitetaan hakukenttään hakusana "Metropolia AMK"-->
<tr>
              <td>type</td>
              <td>id=gbqfq</td>
              <td>Metropolia AMK</td>
</tr>
<!--klikataan hakupainiketta-->
<tr>
              <td>click</td>
              <td>id=gbqfb</td>
              <td></td>
</tr>
<!--klikataan linkkiä, jossa teksti "Metropolia Ammattikorkeakoulu"-->
<tr>
              <td>clickAndWait</td>
              <td>link=Metropolia Ammattikorkeakoulu</td>
              <td></td>
</tr>
<!--verifioidaan sivun otsikkorivi (title) "Metropolia Ammattikorkeakoulu"-->
<tr>
              <td>verifyTitle</td>
              <td>Metropolia Ammattikorkeakoulu</td>
              <td></td>
</tr>
</tbody></table>
</body>
</html>
```

Esimerkkikoodi 1. Nauhoitetun testitapauksen kuvaus HTML-kielellä

Testin rakenne toistuu samanlaisena jokaisessa nauhoitetussa testitapauksessa. Taulukosta käy ilmi, mitä komentoa on käytetty, mikä sivulla oleva komponentti on komennon kohteena ja arvo, jos sitä on tarvittu komennon yhteydessä. Kommenttirivit ovat hyvän tavan mukaiset olla olemassa eli ne helpottavat koodin eri vaiheiden tulkintaa.

Painamalla punaista nauhoituspainiketta käyttöliittymän oikeassa yläkulmassa sovellus alkaa nauhoittamaan käyttäjän toimia Firefox-selaimessa avoimena olevassa ikkunassa. Testitapauksen nauhoituksen ja tarvittavien muokkausten jälkeen testitapaus on mahdollista toistaa eli ajaa käyttöliittymän kautta (kuva 7). Tuolloin ohjelma suorittaa selaimen avoimena ja aktiivisena olevassa ikkunassa nauhoitetun skriptin.




Kuva 7. Selenium IDE -testiskriptin ajaminen

Näkymän alareunan lokista käyttäjä voi seurata ajonaikaisesti komentojen toteutumisia ja mahdollisia virheilmoituksia. Onnistuneet ajon vaiheet näkyvät taulukossa vihreällä ja

epäonnistuneet punaisella. Testitapauksia voidaan ajaa yksitellen tai muodostaa testitapauksista kokoelma (suite), jonka avulla usean erillisen testitapauksen pystyy suorittamaan halutussa järjestyksessä peräkkäin.

### 4.4 Testauksen aikana havaitut haasteet ja ongelmat

Testitapausten skriptien nauhoituksen aikana korostui testitapausten määrittelyjen tarkkuus. Testauksen suunnittelu- ja toteutusapuna ollut testaustaulukko oli erittäin epätarkka, jonka seurauksena kaikki testitapaukset tuli miettiä uusiksi ja määritellä tarkemmin samalla, kun testiskriptejä nauhoitettiin.

Testien nauhoittaminen oli helppoa ja nopeaa, mutta haasteen tehtävään toi se, miten kattavasti käyttöliittymää haluttiin testata. Järjestelmän käyttö perustui mitä suurimmalta osin käyttäjän syötteisiin ja tallennuksiin. Nauhoitus ilman lisättyjä testivälikomentoja testasi käyttäjän toimia ja niiden onnistumista nauhoitetussa järjestyksessä. Se koettiin kuitenkin puutteelliseksi testaamiseksi ja tämän johdosta testauksen kattavuutta ryhdyttiin parantamaan ja tarkentamaan etenkin näkymissä, jotka muuttuivat käyttäjän syötteiden ja tallennuksen myötä. Haluttiin varmistaa, että käyttäjän syötteet tulevat näkyviin eli tallentuvat järjestelmään.

Kattavuuden parantaminen lisää testauksen laatua. Testitapauksia määritellessä ja nauhoittaessa tuli päättää se, olisiko oleellista testata joka tallennuksen jälkeen koko näkymä ja kaikki siinä näkyvät elementit vai oliko tärkeämpää keskittyä muokattuihin osioihin ja niiden muutosten testaamiseen. Kattava testaus vaatisi enemmän työtä ja aikaa mutta toisaalta varmistaisi sen, että näkymissä olevat eri osiot (esimerkiksi otsikot, tekstit, taulukot, komponentit) tulisi testattua ja testien avulla saataisiin mahdollisesti näkymien päivittymiseen liittyvät virhetilanteet paikallistettua. Testit päätettiin toteuttaa ”puolikattavasti”, joka tarkoitti käytännössä sitä, että testattiin muokattujen osioiden päivittyminen sekä se, että näkymässä näkyi oikeat asiat oikealla paikalla. Tekstien näkyvyyden testaamisen avulla saatiin testattua se, että tieto löytyy tietokannasta ja se pystytään näyttämään käyttöliittymässä ja että näkymä on rakentunut oikein.

Ongelmaksi ja haasteeksi testitapauksia nauhoittaessa muodostui testien ylläpidettävyys. Testitapauksia oli nauhoitettu kymmeniä perustuen siihen, mitä käyttöliittymässä

näkyy ja tapahtuu. Muutokset käyttöliittymän rakenteessa, näkymissä, sivurakenteessa tai käyttötapauksessa itsessään tekisivät nauhoitetusta skriptistä käyttökelvottoman, sillä se ei enää vastaisi alkuperäistä määrittelyä eli sitä hetkeä, jolloin testitapaus oli nauhoitettu. Jos samasta käyttötapauksesta oli tehty monta erilaista testiskriptiä (esimerkiksi henkilötunnuksen syöttämisessä tapahtuvat eri käyttötapaukset erilaisilla syötevariaatioilla) niin muutokset aiheuttivat sen, että monta skriptiä oli korjattava uusilla nauhoituksilla. Ylläpidettävyyden kannalta huomattiin, että testiskriptit olisi pidettävä mahdollisimman yksinkertaisina ja testitapaukset oli pilkottava mahdollisimman pieniin osioihin, jotta yksittäiset testiskriptit saataisiin mahdollisimman pienellä työllä uudelleen käyttöön. Isot muutokset esimerkiksi käyttöliittymän rakenteessa tai komponenttien nimissä vaatisivat käytännössä testien nauhoittamisen uudelleen. Komponentit testattiin tässä tapauksessa lähinnä komponenttien tunnisteeseen perustuvaan paikantamiseen näkymissä.

Pohjolainen (3, s. 41) toteaakin tutkielmassaan automatisoitujen testien ylläpidosta seuraavasti:

> Automatisoidun testauksen ylläpitokustannukset ovat paljon korkeammat kuin manuaalisen, koska testattaessa manuaalisesti on mahdollista toteuttaa muutoksia välittömästi. Automatisoidussa testauksessa kaikki osaset on määriteltävä tavalla tai toisella. Mitään ei voida jättää työkalun huolehdittavaksi, koska sillä ei ole älyä. Ohjelmistojen ylläpito on itsestään selvyys kaikille, mutta usein ajatellaan liian vähän testien ylläpidettävyyttä. Testauspaketti sisältäen testauksen dokumentoinnin, testitapaukset ja odotetut tulokset on niin arvokas kokonaisuus, että sitä täytyisi pitää yhtä tärkeänä kuin ohjelmistoa, jota sillä testataan.

Testausdokumentaation parantaminen oli yksi osatavoite varsinaisten testausskriptien nauhoittamisen ohessa. Työtä tehdessä huomattiin, että itse dokumentaation tekeminen olisi todella iso työ, joka vaatisi todella paljon aikaa ja resursseja, joten sen osion toteuttaminen jätettiin pois.

## 4.5 Esimerkkitestitapaus

Esimerkkitestitapaus on kuvattuna liitteessä 1, joka on piilotettu julkaistusta insinöörityöstä sen sisältäessä luottamuksellisia tietoja.

# 5 Yhteenveto

Ohjelmistotestaus ketterän kehityksen ohjelmistoprojektissa on haastavaa. Nopeasti muuttuva ohjelmisto uusine toimintoineen ja ominaisuuksineen pitää testaajan jatkuvasti kiireisenä. Ketterä kehitystapa kehittää jokaisella sprintillä tuotetta paremmaksi, jolloin myös jokaisella sprintillä tulee muutoksia ja uusia testitapauksia järjestelmään, joita tulee testata. Jatkuva muutos on etenkin automaatiotestauksen ylläpidon kannalta työlästä ja muutoksien seurauksena testaus on jatkuvasti kehitystä jäljessä, koska sprinttien aikana ei voida yhtä aikaa kehittää tuotteeseen ominaisuuksia ja testejä. Ominaisuuden tulee olla valmis, jotta sitä voidaan testata. Alkuvaiheessa uusi ominaisuus kokee varmastakin parannuksia, koska staattisella testaamisella ei kaikkia suunnitteluvirheitä voida karsia ja usein vasta ensimmäisen tai toisen toteutuksen jälkeen ominaisuus alkaa toimia suunnitellusti. Vakiintuneen toimintatavan jälkeen voidaan lähteä rakentamaan automaatiotestejä. Tätä ennen ominaisuus testataan manuaalisti kokeilevalla testauksella.

Insinöörityön kohteena olleen sovelluksen testaussuunnitelmaa ja automaatiotestejä luotaessa törmättiin huonosti määriteltyihin testitapauksiin ja myös järjestelmän toimintoihin, joita ei ollut määritelty riittävän tarkasti. Kokeilevan testauksen avulla nämä ominaisuudet saatiin kirjattua kattavammin testausdokumentaatioon, jotta niille voitiin luoda automaatiotestit. Testitapausten suunnittelu ja nauhoittaminen tarkoituksenmukaisella tarkkuudella oli haastavaa. Nauhoituksia tehdessä tuli käyttötapauskohtaisesti määritellä, mitä kaikkea oli oleellista testata testitapauksessa. Liian yksityiskohtainen testaus olisi hankala ylläpitää, jos käyttöliittymässä tapahtuisi muutoksia. Toisaalta liian ympäripyöreät testitapaukset eivät testanneet riittävän kattavasti näkymiä ja niissä olevia komponentteja, tekstejä ja niiden sijainteja. Testitapauksien toistettavuus tuotteen eri versioissa korostui, haluttiin sellaisia testejä, jotka voitiin suorittaa versiosta ja mahdollisista uusista ominaisuuksista ja näkymämuutoksista huolimatta. Testitapaukset tuli pilkkoa riittävän pieniin kokonaisuuksiin ja yksinkertaisiin testitapauksiin, jotta niiden ylläpidettävyys säilyi. Mitä enemmän testitapauksia saatiin testattua, sitä kattavammin testauksen tuloksista voitiin raportoida testitapauskohtaisesti.

Haastavuudesta huolimatta huomattiin kuitenkin, että automaatiotestauksella on vahvuutensa. Automaatiotestaus vähensi manuaalista testausta huomattavasti ja tavoitteeseen päästiin ainakin ajankäytön osalta. Testejä voitiin toistaa lukemattomia kertoja ja

niitä voitiin suorittaa ajastetusti milloin tahansa. Automaatiotestauksen avulla voitiin varmistaa se tärkeä seikka, että sovellus toimii edelleen ja vanhat ominaisuudet eivät olleet menneet rikki uusia kehitettäessä. Automaatiotesteillä saatiin luotua yhdenmukaisuutta ja vakautta testausprosessiin, jonka avulla sovelluksen luotettavuutta ja laatua saatiin parannettua. Tämän insinöörityön toteutumisen myötä automaatiotestit otettiin jatkuvaan käyttöön sovelluskehityksen tueksi regressiotestejä varten ja koettiin, että testien suunnitteluun ja nauhoittamiseen käytetty aika oli ollut hyödyllistä.

Insinöörityötä aloittaessa alkuvaiheen tavoitteet muuttuivat monta kertaa. Alun optimistinen tavoite koko ohjelman kattavasta nopeasti luodusta automaatiotestauksesta muuttui vähitellen moduulikohtaisten testitapausten suunnitteluun ja sen jälkeisiin testitapauksen nauhoitukseen. Testitapauksia muodostui koko ajan lisää kokeilevan testauksen seurauksena, sillä näkemys sovelluksen toiminnasta ja testausta tarvittavista ominaisuuksista muuttui, laajeni ja parantui koko ajan työn edetessä. Tämän johdosta testitapausten nauhoitettuja tiedostoja muodostui paljon ja niiden hallinta oli haastavaa. Tulevaisuutta varten testitapausten hallinnan ja ylläpidon suunnittelu olisi seuraava kehityskohde. Selenium ei tarjoa mitään hallintatyökalua tiedostoille. Testauksen laajentuessa tiedostojen määrä kasvaa satoihin ellei jopa tuhansiin testitapauksiin, jolloin niiden hallinta tulisi suunnitella hyvin.

Insinöörityön eri vaiheissa tuli opittua paljon uutta ohjelmistokehityksen työskentelytavoista ja prosesseista. Tutustuminen ketteriin menetelmiin oli mielenkiintoista, lisäksi yrityksessä käytössä ollut tehtävienhallintaohjelmisto JIRA selvensi sitä, miten työt käytännössä jaetaan ja miten kehitys tapahtuu käytännössä. Automaatiotestauksen rakentaminen oli hyvä tapa tutustua ohjelmiston testauksen teoriaan ja käytäntöön. Työ oli itsenäistä mutta palkitsevaa, testitapaus toisensa jälkeen lisäsivät tietämystä sovelluksen käyttöliittymän rakenteesta ja sen takana tapahtuvista prosesseista.

## Lähteet


1 Kasurinen, Jussi Pekka. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo.

2 Sainio, Laura. 2009. Opinnäytetyön lukumateriaaliosa. Ohjelmistotestauksen menetelmät ja työvälineet. <https://publications.theseus.fi/bitstream/handle/10024/12297/Sainio_Laura_liite1.pdf>. Luettu 26.3.2015.

3 Pohjolainen, Pentti. 2003. Pro Gradu-tutkielma. Ohjelmiston testauksen automatisointi. <http://www.cs.uku.fi/tutkimus/Teho/PenttiPohjolainen_Gradu.pdf>. Luettu 26.3.2015.

4 Kangas, Arto. 2011. Kandidaatintyö. Reaaliaikajärjestelmän käyttöliittymätestauksen automatisointi. <http://wiki.ase.tut.fi/courseWiki/images/2/2a/Kangas-201102.pdf>. Luettu 12.5.2015.

5 Schwaber, K., Sutherland, J. (suom. Lekman, L). Heinäkuu 2013. Verkkodokumentti. Scrumin määritelmä ja pelisäännöt. <http://scrumwell.files.wordpress.com/2014/03/scrum-guide-2013-fi-v1-1.pdf>. Luettu 26.3.2015

6 Vuori, Matti. 2010. Verkkodokumentti. Ketterä testaus ja testaus ketterässä ohjelmistokehityksessä. <http://www.mattivuori.net/julkaisuluettelo/liitteet/kettera_testaus.pdf>. Luettu 26.3.2015.

7 Walkden, Michael. 2011. Agile in FDA Regulated Medical Software – Functional Testing vs Technical Testing. Verkkosivusto. <http://pathfindersoftware.com/2011/05/functional-test-vs-technical-testing/>. Luettu 26.3.2015.