Markus Tuominen

GRAAFISTEN KÄYTTÖLIITTYMIEN REGRESSIOTESTAUKSEN AUTOMATISOINTI

JYVÄSKYLÄN YLIOPISTO
TIETOJENKÄSITTELYTIETEIDEN LAITOS
2014

TIIVISTELMÄ

Tuominen, Markus Olavi
Graafisten käyttöliittymien regressiotestauksen automatisointi
Jyväskylä: Jyväskylän yliopisto, 2014, 76 s.
Tietojärjestelmätiede, pro gradu –tutkielma
Ohjaaja: Sakkinen, Markku

Testaaminen on järjestelmäkehityksen elintärkeä osa. Järjestelmien jatkuva monimutkaistuminen ja erityisesti graafisten käyttöliittymien lisääntyminen johtaa siihen, että laadunvarmistuksen merkitys kasvaa edelleen ja lisäksi siitä tulee entistä haastavampaa ja kalliimpaa. Keskimäärin yli puolet ohjelmistokehitysprojektien kokonaiskustannuksista syntyy testauksesta. Testauksen automatisoinnilla pyritään pienentämään testauksesta aiheutuvia kustannuksia ja samalla tehostamaan testauksen toimintaa.

Aina ei ole kuitenkaan järkevää automatisoida testausta. On voitava perustella, milloin testaamisen automatisointi on kannattavaa. Regressiotestauksen yhteydessä näin usein on, sillä testausta suoritetaan useita kertoja käyttäen samoja testitapauksia. Regressiotestauksen tarkoitus on varmistaa, että ohjelmisto toimii muutosten jälkeen oikein. Sen vuoksi regressiotestausta tulisi suorittaa mahdollisimman usein, mielellään jokaisen muutoksen jälkeen. Manuaalisesti työmäärä on kuitenkin niin iso, että sen suorittaminen ei ole järkevää automatisoimatta.

Tässä pro gradu -tutkielmassa luodaan kirjallisuuteen perustuva katsaus ja selvitetään, millaisia malleja, viitekehyksiä ja tekniikoita on esitetty graafisten käyttöliittymien regressiotestauksen automatisointiin. Tarkemmin esitellään viitekehyksistä Daily Automated Regression Tester (DART) ja automatisointityökaluista Selenium IDE, jotka valittiin myös esimerkkiprojektimme graafisten käyttöliittymien regressiotestauksen automatisointiin.

Tutkielman empiirisessä osuudessa raportoidaan tämä kokeellinen tutkimus, jossa luotiin malli, jota pilotoitiin esimerkkiprojektimme. Koska organisaatiomme on testauksen automatisoinnin saralla testauksen kypsyysmallin alimmalla tasolla, emme voineet käyttää DARTia suoraan, vaan siitä piti luoda organisaatiollemme sopiva malli. Tätä mallia hyväksi käyttäen saimme automatisoitua esimerkkiprojektimme graafisen käyttöliittymän regressiotestauksen. Testauksen kohteena tässä pilotissa oli suuremman järjestelmän yksi osa: pikalinkkisovellus. Tutkimuksen perusteella voitiin vetää johtopäätös, että graafisten käyttöliittymien automaattinen regressiotestaus on mahdollista ottaa organisaatiossamme käyttöön melko pienin ponnistuksin mutta jatkotutkimus on kuitenkin vielä tarpeellista maksimaalisen hyödyn saamiseksi.

Asiasanat: graafinen käyttöliittymä, GUI, testiautomaatio, regressiotestaus, Selenium IDE

ABSTRACT

Tuominen, Markus Olavi
Automating the regression testing of graphical user interfaces
Jyväskylä: University of Jyväskylä, 2014, 76 p.
Information Systems, Master's Thesis
Supervisor: Sakkinen, Markku

Testing is an essential part of the system development process. The continuous evolution of the systems and especially the increasing number of graphical user interfaces emphasize the importance of quality assurance. It also becomes more expensive and more challenging. On the average testing generates more than 50 per cent of the total expenses of a system development project. Automating testing processes is a means to lower the expenses and simultaneously enhance the effectivity of testing.

It is not always reasonable to automate the testing. It must be justified when test automation is worthwhile. Usually this is the case with regression testing because the same test cases are executed several times. The purpose of regression testing is to prove that the software works as intended after there has been a modification in the code. Therefore regression tests should be executed as often as possible, preferably after every modification. However, running tests manually is very laborious so it is not reasonable to do so without automation.

In this Master's Thesis, using a literature review, I examine what kind of models, frameworks and techniques there are to support automating regression testing of graphical user interfaces. I discuss especially Daily Automated Regression Tester (DART) and Selenium IDE which were adopted for piloting the automated regression testing of graphical user interfaces in our sample project.

In the empirical section of the study I report this pilot project. Because our organization is on the lowest level of the Testing Maturity Model we could not adopt the DART framework as such but we needed to modify it to create a version suitable for our project. Using this model we were able to automate the regression testing of the graphical user interface in our sample project. The application under test was part of a bigger system: an application used to create quick links in the system. Based on the results of the study it is clear that it is possible to adopt automated regression testing of graphical user interfaces in our organization with quite little effort. However, regardless of the results I cannot declare that this study is enough to adopt the automation process organization-wide but more investigation is needed. However the results suggest that more investigation is worthwhile.

Keywords: graphical user interface, GUI, test automation, regression testing, Selenium IDE

KUVIOT

TAULUKOT

SISÄLLYS

1 JOHDANTO

Testaaminen on järjestelmäkehityksen elintärkeä osa (mm. Burnstein, Suwanassart & Carlson, 1996). Järjestelmien jatkuva monimutkaistuminen ja kriittisyyden kasvu johtaa siihen, että laadunvarmistuksen merkitys kasvaa edelleen ja lisäksi siitä tulee entistä haastavampaa ja kalliimpaa (Bertolino, 2007). Nyrkkisääntönä voidaan pitää, että noin puolet järjestelmäkehitykseen kuluvasta ajasta (Myers, Badgett & Sandler, 2012) ja yli puolet kokonaiskustannuksista käytetään testaamiseen (Memon, 2002). Tehokas ohjelmistotestaus voi käyttää jopa kaksi kolmasosaa koko ohjelmistokehitysprojektin resursseista (Hackner & Memon, 2008). Testauskuluja voidaan vähentää käyttämällä uudelleen vanhoja testitapauksia ja testaamalla vain muuttuneet ominaisuudet (Leung & White, 1990). Lisäksi on yleisesti tunnettu tosiasia, että ongelmien korjaaminen kehityksen alkuvaiheessa maksaa vähemmän kuin niiden korjaaminen myöhemmin (Dustin, Rashka & Paul, 2008).

Web-sovellukset tuottavat yhteiskuntaamme erilaisia kriittisiä palveluita aina yksityiseltä kaupalliselta puolelta julkiselle hallinnolliselle puolelle sekä terveydenhuoltoon. Web-sovellusten leviäminen laajalle ja useiden eri käyttäjien joukkoon luo painetta sovellusten kehittäjille tuottaa laadukkaita järjestelmiä. (Leotta, Clerissi, Ricca & Tonella, 2013). Web-sovellukset käyttävät graafisia käyttöliittymiä, joten niiden testaaminen ja laadun varmistaminen on tärkeää. Graafisten käyttöliittymien testaaminen on kuitenkin haastavampaa kuin perinteisten käyttöliittymien johtuen mm. niiden tapahtumapohjaisesta luonteesta. Lisäksi graafisia käyttöliittymiä voidaan käyttää paljon monimutkaisemmin ja se lisää virheiden mahdollisuutta (Gerrard, 1997).

Testauksen automatisointi on yksi yritys pienentää testauksesta aiheutuvia kustannuksia. Testauksen automatisoinnilla voidaan vähentää manuaalista työtä ja sitä kautta työstä aiheutuvia kustannuksia sekä myös säästää aikaa. Testauksen automatisointi on yksi erittäin oleellinen tapa parantaa investoinnin tuottoprosenttia (return of investment, ROI) (Mittal & Acharya, 2003). Testauksen automatisointi ei kuitenkaan ole helppoa johtuen testaamisen laajasta alueesta, sillä testauksen skaala vaihtelee yksikkötesteistä koko järjestelmän testaukseen ja lisäksi voidaan suorittaa myös ajonaikaista tai suorituskykytestausta.

Testauksen automatisoinnin etuna on myös se, että se vähentää manuaalisen testauksen mahdollisia virheitä (Dustin ym., 2008). Automaattinen testaaminen mahdollistaa lisäksi uusia testausmuotoja, kuten esimerkiksi tuhannen käyttäjän yhtäaikaisen käytön simuloinnin. Testaamisen automatisointiin onkin kehitetty useita erilaisia apuvälineitä eri vaiheisiin ja tarpeisiin. Välineiden kehitys sekä aihealueen tutkimus käy edelleen vilkkaana.

Toiminnallisten testien automatisointi on yksi perinteisistä ongelmista ohjelmistokehitysprojekteissa. Yksikkötestien automatisointi on ottanut aimo harppauksia eteenpäin mutta toiminnallista testausta tehdään pääsääntöisesti yhä manuaalisesti. Erityisesti web-sovellusten testauksen automatisointi on ollut haastavaa johtuen mm. monikerroksisesta arkkitehtuurista, erilaisista selaimista ja web-teknologioista, kuten JavaScriptistä. (Holmes & Kellogg, 2006).

Testaukseen ylipäätään ja erityisesti sen automatisointiin tuovat lisähaastetta graafiset käyttöliittymät (GUI, graphical user interface). Graafisissa käyttöliittymissä on erilaisia objekteja, joiden kanssa käyttäjä voi olla vuorovaikutuksessa. Nämä objektit voidaan järjestää ruudulle lukemattomilla eri tavoilla: niiden koko ja sijainti voi vaihtua ja lisäksi käyttäjä voi valita objekteja käytännössä missä järjestyksessä tahansa. (Dustin ym., 2008; Hackner & Memon, 2008). Tämä lisää edelleen haastetta testauksen automatisoinnille, sillä esimerkiksi perinteiset hiiren ja näppäimistön tallentamiseen perustuvat automatisointityökalut eivät välttämättä osaa ottaa objektien sijaintien eroa huomioon. Automaattisessa testauksessa, kun järjestelmän käyttöliittymä muuttuu, ei aiemmin kehitetyistä ja käytetyistä testitapauksista välttämättä ole enää hyötyä. Joko niitä pitää muokata tai ne ovat täysin käyttökelvottomia käyttöliittymän muuttuessa (Memon & Soffa, 2003). Myös regressiotestien testitapauksia pitää päivittää tarvittaessa.

Testauksen keskiössä on testaaja ja testaustiimi. Nykyisin vallalla olevien ketterien kehitysmenetelmien käyttö vaatii myös ketterän testaajan. Testaaja osana ketterää tiimiä on aktiivinen tiedon välittäjä ja vastaanottaja. Käytännössä tämä tarkoittaa aktiivista osallistumista palavereihin ja suunnittelukokouksiin sekä tiedonkulkua edistävien käytäntöjen kehittämistä. Testaajan tulee sekä itsenäisesti että muun tiimin kanssa etsiä tietoa testattavasta ohjelmistosta ja selvittää, mitä kannattaa testata. Ketterällä testaajalla on hyvä olla teknistä osaamista ja kokemusta muun muassa mustalaatikkotestauksesta, testien automatisoinnista, skriptien kirjoittamisesta ja tietokannoista. (Crispin, 2006b).

Tässä pro gradu –tutkielmassa luodaan katsaus kirjallisuuteen ja esitellään, millaisia malleja ja tekniikoita on esitetty graafisten käyttöliittymien regressiotestauksen automatisointiin. Lisäksi tutkielman empiirisessä osuudessa raportoidaan kokeellinen tutkimus, jossa projektissamme pilotoidaan graafisten käyttöliittymien regressiotestauksen automatisointia. Projektissamme käytettävä malli luodaan kirjallisuudesta löytyneitä malleja ja tekniikoita yhdistellen siten, että kokonaisuudessaan uusi malli sopii hyvin sekä meidän projektiimme että organisaatiomme tämänhetkiseen tilaan testauksen kypsyystasolla. Tutkielman tulosten perusteella tavoitteena on kehittää mallia edelleen kattamaan suurempi osa projektin regressiotestauksesta ja lopulta ottaa graafisten käyttö-

liittymien automaattinen regressiotestaus käyttöön myös muissa vastaavanlaisissa projekteissa.

Tutkielmani tutkimusongelma on *Mitä kirjallisuudessa kerrotaan graafisten käyttöliittymien automaattisesta regressiotestauksesta?* sekä empiirisessä osuudessa *Kuinka yrityksessäni (ja erityisesti nykyisessä projektissani) voidaan ottaa käyttöön ja käyttää automaattista regressiotestausta, kun kyseessä on graafisella käyttöliittymällä varustettu ympäristö?* Osaongelmana voidaan pitää seuraavaa: *Mitkä kirjallisuudessa esitetyistä malleista ja teknologioista ovat soveltuvia ja miten niitä voidaan yhdistellä palvelemaan yritystäni?*

Tämä tutkielma rakentuu siten, että luvussa 2 pureudutaan testaamiseen yleisellä tasolla sekä esitellään tutkimuksen kannalta keskeiset käsitteet. Luvussa käydään läpi testauksen prosessi sekä erotellaan uudelleentestaus regressiotestauksesta. Luku käy läpi myös graafisten käyttöliittymien testauksen erityispiirteitä sekä ottaa katsauksen testauksen automatisointiin.

Luvussa 3 esitellään kirjallisuudesta löytyneitä malleja ja tekniikoita, joiden pohjalta empiirisen osuuden malli rakennetaan. Luvussa esitellään testauksen kypsyystasomalli sekä tämän tutkimuksen kannalta keskeinen viitekehys DART (Daily Automated Regression Tester) sekä automaattisten testien suorittamiseen valittu työkalu Selenium IDE.

Luvussa 4 aloitetaan tutkielman empiirinen osuus. Tässä luvussa esitellään lyhyesti organisaatiomme sekä projektimme, jossa automaattisten regressiotestien pilotointi tehdään. Lisäksi luvussa rakennetaan kokeellisessa tutkimuksessa käytettävä malli, joka pohjautuu DARTiin mutta jota on yksinkertaistettu organisaatioomme sopivaksi. Tätä mallia käytetään yhteen osaan toteuttamaamme järjestelmää. Tutkimukseen valittu osa on pikalinkkien hallintasovellus, jonka kautta luodaan ja hallitaan käyttäjien henkilökohtaisia sekä yhteisiä pikalinkkejä sekä järjestelmän sisään että ulkopuolelle.

Luvussa 5 suoritetaan testitapausten luonti ja suoritus käyttäen Selenium IDE –työkalua. Luku aloitetaan täysin puhtaalta pöydältä alkaen aina Seleniumin IDEn asentamisesta ja esittelystä aina testien kirjoittamiseen ja suorittamiseen asti. Suorituksen yhteydessä käydään läpi yhtä testitapausta tarkemmin ja selitetään sen toimintalogiikkaa.

Luku 6 tarkastelee testien suorittamisesta saatuja tuloksia sekä pohtii niistä saatavia johtopäätöksiä. Johtopäätöksissä tarkastellaan, onko esittämäni malli sopiva tai muuten mahdollinen organisaatiolleni tai projektilleni sekä oliko graafisten käyttöliittymien automaattisen testauksen implementoinnista mitään hyötyä. Lisäksi luvussa pohditaan, mitä tutkimuksen suorittamisesta opittiin ja otetaan katsaus jatkotutkimuskohteisiin ja pohditaan testauksen automatisoinnin kehitystä organisaatiossamme.

Luku 7 kokoaa yhteen tutkielman pääkohdat. Tutkielman liitteeksi on annettu tutkielmassa tarkasteltu esimerkkitestitapaus taulukkomuotoisena sekä osittainen esimerkkiesitys HTML-muotoisesta esityksestä.

2 YLEISTÄ TESTAAMISESTA SEKÄ KÄSITTEIDEN ESITTELY

Tässä luvussa käsitellään ohjelmistojen testaamista yleisellä tasolla sekä esitellään tutkimuksen kannalta keskeisiä käsitteitä. Luvussa esitellään yleinen testausprosessi, graafisten käyttöliittymien erikoispiirteitä testauksen näkökulmasta sekä myös testauksen automatisointia. Lisäksi esitellään regressiotestaus ja sen erityispiirteitä verrattuna muuhun testaukseen. Luvun lopussa on sanasto tutkielmassa käytettävistä ja testaukseen liittyvistä oleellisista termeistä.

2.1 Mitä ohjelmistotestaus on?

Ohjelmistotestauksen perimmäinen tarkoitus on varmistaa, että ohjelmisto toteuttaa sille asetetut vaatimukset. Se siis antaa tukea päätökselle, onko ohjelmisto riittävän hyvä ja toimiva julkaistavaksi tuotantoon. Ohjelmistotestauksella voidaan osoittaa, että ohjelmistossa on virheitä, mutta sillä ei voida koskaan osoittaa, että siinä ei olisi virheitä (Haikala & Märijärvi, 2003). Vikojen havaitseminen vasta ohjelmiston julkaisun jälkeen vaikuttaa suoraan kustannuksiin mutta samalla se osoittaa sovelluksen tuottajan kyvyttömyyteen havaita vikoja ennen julkaisua (Mittal & Acharya, 2003).

Ohjelmistojen testaaminen on erittäin laaja alue. Se käsittää koko skaalan lyhyen ohjelmakoodin testaamisen ja laajan järjestelmän kokonaistestauksen välillä. Ja lisäksi samaan alueeseen voidaan vielä liittää myös käytönaikainen testaaminen, kuten esimerkiksi suorituskykytestaaminen. Testaamisen näkökulma voi myös vaihdella rajusti järjestelmävaatimusten validoinnista syötteiden validointiin ja ajonaikaisen kuorman tai käytettävyyden arviointiin. (Bertolino, 2007.).

Useat ohjelmistokehittäjät ovat kokeneet sen turhautumisen tunteen, kun käyttäjät raportoivat ohjelmistovirheestä. Huolellisen, tuhansia lauseita ja satoja muuttujia sisältäneen testaamisen jälkeenkin lopulliseen ohjelmistoon päätyy

vikoja, joita ei ole testaamisen yhteydessä havaittu. James Whittaker (2000) esittää mahdollisiksi syiksi seuraavat:

1. Käyttäjä suorittaa testaamatonta koodia
2. Käyttäjä suorittaa koodia eri järjestyksessä kuin testauksessa
3. Käyttäjä syöttää arvojoukon, jota ei ole testattu
4. Käyttäjän suoritusympäristöä ei ole testattu

Lisäksi tässä kohtaa voidaan huomioida IEEE:n standardin mukaiset kaksi laadullista näkökulmaa: kuinka hyvin järjestelmä, komponentti tai prosessi toteuttaa sille asetetut (1) tietyt vaatimukset sekä (2) asiakkaan ja käyttäjän odotukset (Robinson, 2009). Nämä eivät aina ole yhteneviä, ja osa järjestelmän vaatimuksista voikin olla sellaisia, joita ei ole määritetty missään vaiheessa (Douglas, 2010).

Ei ole lainkaan tavatonta, että aikarajoituksista johtuen kehittäjät julkaisevat testaamatonta koodia. Laajoissa ohjelmistoissa on lisäksi mahdotonta testata jokaista mahdollista järjestystä tai syötteitä, joita käyttäjät voivat ohjelmistolle antaa. Tämä ongelma on erityisesti graafisissa käyttöliittymissä. Testaajien onkin tehtävä ratkaisuita siitä, mitkä syötteet testataan, ja joskus nämä valinnat voivat olla vääriä. On myös mahdollista, että kehittäjät eivät tunne loppukäyttäjien ympäristöä, jossa koodia suoritetaan. Ja vaikka ympäristö olisikin tiedossa, voi olla mahdotonta rakentaa täysin vastaavaa ympäristöä testausta varten. (Whittaker, 2000).

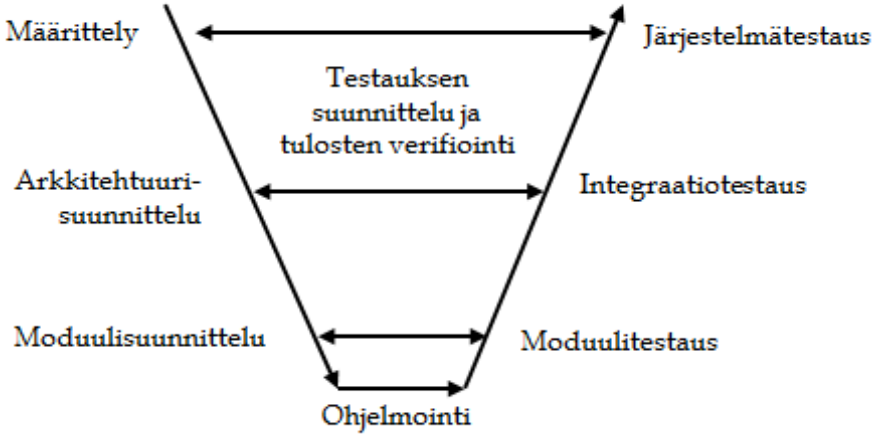
Burnsteinin ym. (1996) mukaan testaaminen on ohjelmistokehitysprosessissa kriittinen komponentti ja samalla myös yksi haastavimmista ja kalleimmista aktiviteeteista. Toisaalta se tarjoaa vahvan tuen laadukkaan ohjelmiston kehittämiselle. James Bachin (1999) kokemuksen mukaan testausprojektit epäonnistuvat jopa vielä useammin kuin kehitysprojektit, sillä usein organisaatiot eivät käytä testaamiseen samaa määrää ammattitaitoa tai resursseja kuin ohjelmistokehityksen muihin vaiheisiin ja välineisiin.

2.2 Testauksen tasot ja lähestymistavat

Testauksen tasoja voidaan kuvata V-mallin avulla (Haikala & Märijärvi, 2003). Malli kuvaa ohjelmiston tuotantoprosessin vaiheita ja niitä vastaavia testaustasoja (KUVIO 1). Testaustasoja ovat *yksikkötestaus* eli *moduulitestaus* (*unit testing*), *integraatiotestaus* (*integration testing*) ja *järjestelmätestaus* (*system testing*). Lisäksi voidaan erottaa hyväksymistestaus, regressiotestaus ja erilaiset käytettävyyteen ja suorituskykyyn liittyvät testaukset sekä esimerkiksi tietoturvaan liittyvät testaukset.

Moduulitestauksen suorittaa yleensä moduulin tekijä itse ja siinä testattavana kohteena on yksittäinen moduuli. Tuloksia verrataan moduulisuunnittelun tuloksiin, tavallisesti tekniseen määrittelydokumenttiin (Haikala & Märijärvi, 2003).

Integraatiotestauksessa testataan eri moduulien yhteistoiminta. Pääpaino on eri moduulien rajapintojen toiminnan testaamisessa. Moduuleja voidaan integroida keskenään joko kokoavasti (bottom up) eli alimman tason moduuleista ylöspäin tai jäsentävästi (top down) eli ylätasolta alemmille tasoille. Integraatiotestauksen tuloksia verrataan yleensä tekniseen määrittelyyn.



KUVIO 1 Ohjelmistotestauksen V-malli (Haikala & Märijärvi, 2003, s. 287)

Järjestelmätestauksessa testataan koko järjestelmää ja tuloksia verrataan määrittelydokumentaatioon eli ohjelmiston toiminnalliseen määrittelyyn. Järjestelmätestauksen suorittajien pitäisi olla mahdollisimman riippumattomia testaajia.

Testitapauksia valittaessa voidaan ongelmaa lähestyä lasilaatikkotestauksen (white box testing) tai mustalaatikkotestauksen (black box testing) näkökulmasta. Lasilaatikkotestauksessa käytetään hyväksi tietoa ohjelman rakenteesta ja mustalaatikkotestauksessa spesifikaatiota ilman tietoa ohjelman toteutuksesta. Mitä ylemmällä tasolla V-mallissa ollaan, sitä todennäköisemmin valitaan mustalaatikkotestaukseen perustuvia testitapauksia. (Haikala & Märijärvi, 2003).

2.3 Testausprosessi

Ohjelmiston testausprosessi koostuu erilaisista vaiheista, joiden avulla testaus voidaan suorittaa alusta loppuun asti. Testausprosessiin on olemassa erilaisia määritelmiä. Työvaiheiksi määritetään seuraavat (mm. Whittaker, 2000; Haikala & Märijärvi, 2003):

(1) Mallinnetaan ohjelmiston ympäristö.
(2) Valitaan testiskenaariot.
(3) Luodaan testiympäristö.
(4) Suoritetaan testiskenaariot.
(5) Tarkastellaan tuloksia.

(6) Mitataan testausprosessi.

Lisäksi the Fundamental Test Process –määritelmän mukaan koko prosessia tuetaan ja hallitaan erillisellä suunnittelun ja hallinnan prosessilla (Spillner, Linz & Schaefer, 2011). Burnstein ym. (1996) listaavat, että hyvässä (kypsässä) testausprosessissa tulee olla seuraavat ominaisuudet:

(1) Selkeästi määritetty ja dokumentoitu testauspolitiikka, joka ulottuu koko organisaatioon. Politiikkaa käytetään ja sitä tuetaan kaikilla organisaatiotasoilla aina johdosta lähtien.
(2) Selkeästi määritetty elinkaari vaiheineen ja aktiviteetteineen. Testauksen elinkaari on integroitu ohjelmiston elinkaareen ja se sisältää testauksen koko alueen aina testauksen suunnittelusta testausdokumentteihin asti. Testauksen elinkaarta käytetään jokaisessa projektissa.
(3) Selkeästi määritetty testauksen suunnitteluprosessi, joka on käytössä koko organisaatiossa.
(4) Itsenäinen testausryhmä, jonka tehtävät on tarkkaan määritetty, ja jonka toiminnalle on johdon tuki. Testaajille tarjotaan koulutusta ja heitä motivoidaan.
(5) Testausprosessin kehittämisryhmä, jonka tehtävänä on kehittää testausprosessia edelleen.
(6) Määritetyt testauksen mittarit, joiden perusteella testausprosessia voidaan kehittää.
(7) Työkalut, jotka helpottavat testaamista.
(8) Testausprosessin kontrollointi- ja seurantaprosessit, joita hoitavat testauspäälliköt. Heidän tehtävänään on toimia, jos ongelmia ilmenee, sekä arvioida testauksen suorituskykyä ja kapasiteettia.
(9) Tuotteen laadunhallinta, jonka mukaisesti on määritetty mm. testaamisen lopettamiskriteerit.

2.3.1 Ohjelmiston ympäristön mallintaminen

Testaajan tehtävänä on simuloida ohjelmiston ja ympäristön välinen vuorovaikutus. Testaajien on siten kyettävä tunnistamaan kaikki rajapinnat sekä syötteet, jotka voivat kulkea näiden rajapintojen läpi. Tämä vaihe voi olla koko testausprosessin haastavin vaihe, sillä mahdollisia tiedostomuotoja, kommunikaatioprotokollia ja kolmannen osapuolen ohjelmistorajapintoja voi olla lukuisia. Whittaker (2000) luettelee neljä yleistä rajapintaa.

Ensimmäinen rajapinta on ihmisrajapinta (Human interface), joista yleisin on graafinen käyttöliittymä (graphical user interface, GUI), mutta myös komentorivipohjaiset käyttöliittymät ovat edelleen käytössä.

Toinen rajapinta on ohjelmistorajapinta (software interface), jonka avulla ohjelmisto kommunikoi esimerkiksi käyttöjärjestelmän ja tietokantojen kanssa. Testaamisen näkökulmasta tässä rajapinnassa on ongelmana odottamattomat palvelut. Testaaja voi esimerkiksi varmistaa, että sovellus pyytää käyttöjärjes-

telmää tallentamaan tiedoston ja että käyttöjärjestelmä tallentaa sen, mutta voi jättää varmistamatta, antaako ohjelmisto käyttäjälle virheilmoituksen, jos tiedoston tallennus ei onnistu esimerkiksi levyn täyttymisen tai rikkoutumisen vuoksi.

Kolmas rajapinta on tiedostojärjestelmän rajapinta (file system interface), joka ilmenee aina, kun ohjelmisto lukee tai kirjoittaa ulkopuoliseen tiedostoon. Ohjelmiston on kyettävä validoimaan sekä tiedostossa oleva sisältö että sen rakenne. Testaajan onkin käytettävä sekä valideja että epävalideja tiedostoja tämän rajapinnan testaamiseen.

Neljäs rajapinta on kommunikaatiorajapinta, joka mahdollistaa suorat yhteydet fyysisiin laitteisiin, kuten laiteohjaimiin ja sulautettuihin järjestelmiin. Nämä yhteydet vaativat kommunikaatioprotokollan. Testaajan onkin voitava testata sekä valideja että epävalideja protokollasyötteitä syöttämällä testattavalle ohjelmistolle sopivassa formaatissa olevia käskyjä ja dataa.

On myös huomattava, että testauksessa on otettava huomioon myös tapaukset, jotka eivät ole suoraan testattavan ohjelmiston hallittavissa. Esimerkiksi mitä tapahtuu, jos järjestelmä käynnistetään uudelleen kesken kommunikaation tai jos ohjelmiston käyttämä tiedosto poistetaan kesken kaiken. (Whittaker, 2000).

Ohjelmistojen koon ja monimutkaisuuden kasvaessa on pakko rajoittaa testattavia syötteitä ja arvoja. Tämä vaikeutuu merkittävästi, jos mahdollisia arvoja on useita ja niillä on vaikutusta toisiinsa joko suoraan tai esimerkiksi järjestyksen mukaan. Syötteet voivat saada erilaisia merkityksiä riippuen siitä, missä järjestyksessä ne annetaan. (Whittaker, 2000).

2.3.2 Testiskenaarioiden valinta

Monissa ohjelmistoympäristön malleissa on ääretön määrä testiskenaarioita, ja jokainen kuluttaa aikaa ja rahaa. Niinpä onkin tärkeää voida valita järkevästi sopiva osajoukko. Huolellinen testitapausten valinta voi johtaa pienessä testausajassa huomattavasti parempaan tulokseen kuin satunnaisesti valituilla testeillä suoritettu pitkäkestoinen testaus (Haikala & Märijärvi, 2003).

Suositeltavaa on pyrkiä tutkimaan tyypilliset käyttötapaukset, eli ne, joita käyttäjät todennäköisemmin käyttävät. Näin ollen saadaan löydettyä ainakin kaikkein tärkeimmät virheet. Desikan Srinivasan ja Ramesh Gopalaswamy (2008) ehdottavat, että valinnassa otetaan hyöty aiemmasta tiedosta ja valitaan ne testitapaukset, jotka ovat aiemmin tuottaneet eniten virheitä. Tavallisesti testitapausten valintaan käytetään myös yhtä tai useampaa kattavuuskriteeriä.

Yleisimpiä lasilaatikkopohjaisia kattavuuskriteerejä ovat lausekattavuus, päätös- eli haarakattavuus, polkukattavuus, ehtokattavuus ja moniehtokattavuus. Mahdollisia suorituspolkuja on kuitenkin yleensä liian monta tai jopa ääretön määrä. Siksi täyden polkukattavuuden sijasta käytetään esim. McCabe'in kantapolkujen (basis path) kattavuutta ja tietovuohon (data flow) perustuvia kriteerejä. Testauksessa voidaan käyttää myös virheiden kylvämistä (error seeding), jolloin testitapausjoukon pitäisi tunnistaa kaikki kylvetyt virheet.

Mustalaatikkopohjaisessa testauksessa mahdollisten syötteiden määrä on melkein aina liian suuri katettavaksi. Siksi testitapausten valinnassa käytetään ekvivalenssiositusta (equivalence partitioning) ja raja-arvoanalyysiä (boundary value analysis).

Syötesarjojen testaamiseen voidaan valita testitapausjoukkoja esimerkiksi valitsemalla testitapausjoukko, jossa kutsutaan jokaista mahdollista käyttöliittymäelementtiä (ikkuna, valikko, painike jne.) tai testitapausjoukko, joka sisältää todennäköisimmät tavallisen käyttäjän suorittamat toimintosarjat.

2.3.3 Testiskenaarioiden suorittaminen ja arviointi

Sopivien testiskenaarioiden valinnan jälkeen testaaja muuttaa skenaariot ajettavaan muotoon, joko koodiksi, jolloin tietokone suorittaa koodin automaattisesti tai joukoksi ohjeita, jolloin ihminen suorittaa testattavat toimenpiteet itse. Koska testien suorittaminen vaatii runsaasti työtä ja on virhealtista, pyritään tämä vaihe automatisoimaan mahdollisimman pitkälle. Tähän vaiheeseen onkin olemassa lukuisia työkaluja.

Testiskenaarioiden arviointi on suorituksesta saatujen tulosten vertaamista odotettuihin tuloksiin. Mikäli odotetut tulokset ja suorituksesta saadut tulokset eroavat toisistaan, on löytynyt vika. Vertailussa käytetään usein testioraakkelia. Odotetut tulokset saadaan esimerkiksi määrittelydokumentaatioista. Ilman spesifikaatiota onkin mahdotonta testata, sillä tulosten oikeellisuutta ei voida todeta (Haikala & Märijärvi, 2003). Tämän tutkielman empiirisessä osuudessa tarkastellaan juuri tämän vaiheen automatisointia.

2.3.4 Testausprosessin mittaaminen

Testaamisen mittaamiseen liittyy paljon lukuja: kuinka monta riviä olemme testanneet, kuinka monta syötettä, kuinka monta virhettä olemme löytäneet, kuinka monta kertaa testimme ovat menneet läpi ja niin edelleen. Lukujen arviointi on kuitenkin haasteellista, sillä emme voi välttämättä tietää, onko jokin luku hyvä vai huono. Esimerkiksi onko se hyvä asia, että olemme löytäneet lukuisia vikoja? Whittakerin (2000) mukaan se voi olla kumpi tahansa. Hyvän asian siitä tekee se, että testaus on ollut tehokasta. Mutta toisaalta se voi olla myös huono asia, sillä löydösten perusteella vaikuttaisi siltä, että ohjelmistossa on todella paljon vikoja, ja vaikka olemme löytäneet niitä paljon, niitä voi olla saman verran vielä lisää.

Testaamisen mittaaminen on kuitenkin tärkeää, sillä sen perusteella päätetään, voidaanko testaaminen lopettaa ja onko tuote valmis julkaistavaksi. Käytännössä varsinkin hyväksymistestaamista voidaan jatkaa kunnes aika tai rahat loppuvat. Testaamisen loppukriteerit tuleekin määrittää testaussuunnitelmassa esimerkiksi kumulatiivisena löydettyjen virheiden määränä tai vaikkapa kattavuusmitoilla. (Haikala & Märijärvi, 2003).

2.4 Regressiotestaus

Ohjelmiston ylläpitovaiheessa sovelluksiin tehdään muutoksia, parannuksia ja korjauksia sekä poistetaan mahdollisesti ylimääräisiä toiminnallisuuksia. Näiden muutosten vuoksi on mahdollista, että osa aiemmin toiminutta kokonaisuutta ei toimi enää muutosten jälkeen (mm. Duggal & Suri, 2008). V-malliin sisältymätön ohjelmistotestauksen vaihe on regressiotestaus, jonka tarkoituksena on juuri varmistaa, että ohjelmistoon tehdyt muutokset eivät ole vaikuttaneet mihinkään muuhun toiminnallisuuteen ohjelmistossa. Regressiotestauksen tarkoitus on siis varmistaa, että aiemmin testatut ominaisuudet toimivat edelleen, vaikka ohjelmistokoodi on muuttunut. Toisin sanoen aiemmin läpäistyjen testitapausten on mentävä läpi myös ohjelmakoodin muutoksen jälkeen. (mm. Leung & White, 1990).

Regressiotestaus tulee suorittaa aina, kun ohjelmistokoodia on muokattu, jotta voidaan varmistaa, ettei tehdyistä muutoksista ole haittaa ohjelmiston toimintaan. Käytännössä regressiotestaamiselle on kaksi vaihtoehtoista tapaa: (1) suoritetaan kaikki testitapaukset tai (2) suoritetaan vain osa testitapauksista (Rothermel & Harrold, 1996). Ohjelmistojen koon kasvaessa jokaisen muutoksen jälkeen koko järjestelmän testaaminen on hyvin raskasta työtä ja näin ollen sitä ei välttämättä ole järkevää tehdä koko järjestelmän laajuudessa (Leung & White, 1991). Lisäksi järjestelmän koon kasvaessa myös testien määrä kasvaa, ja muutosten aiheuttamina osa testeistä ei ole enää merkityksellisiä. Sen vuoksi testitapauksia pitää hallita ja pyrkiä saamaan mahdollisimman järkevä testijoukko, jolla järjestelmä kannattaa testata muutosten aiheuttamia ongelmia vastaan (Harrold, Gupta & Soffa, 1993).

Tästä syystä regressiotestaukselle kannattaakin etsiä jonkinlainen alue kriittisistä toiminnoista, jotka testataan muutosten yhteydessä. Regressiotestiksi kannattaa valita ne testitapaukset, jotka ovat aiemmin tuottaneet eniten häiriöitä (Srinivasan & Gopalaswamy, 2008). Kirjallisuudessa on esitetty erilaisia valintatekniikoita sopivien testitapausten löytämiseen. W. Eric Wong, J. R. Horgan, Saul London ja Hira Agrawal (1997) mainitsevat esimerkiksi muutosperusteisen (modification-based) valintatekniikan, jossa regressiotestien testitapauksiksi pyritään valitsemaan mahdollisimman tehokas osajoukko. Tämän testijoukon valintaan Wong ym. ehdottivat käytettäväksi minimisaatiota (minimization) ja priorisointia (priorization). Minimisaatiolla tarkoitetaan minimitestitapausjoukkoa, jolla toiminnallisuus voidaan varmistaa. Priorisoinnilla puolestaan ajetaan mahdollisimman paljon testejä aloittaen tärkeimmistä toiminnallisuuksista. Gregg Rothermel ja Mary Jean Harrold mainitsevat tutkimuksessaan (1994) lisäksi kattavuuteen perustuvat menetelmät (coverage methods) sekä turvalliset menetelmät (safe methods). Kattavuuteen perustuvassa menetelmässä valitaan testitapauksiksi kaikki testitapaukset, jotka liittyvät muuttuneeseen osaan. Turvallinen menetelmä puolestaan tarkoittaa, että alkuperäisestä testitapausjoukosta valitaan kaikki sellaiset testitapaukset, jotka voivat löytää häiriön muutetusta ohjelmistosta (Willmor & Embury, 2005).

Leung ja White (1991) kuitenkin huomauttavat, että selektiivinen regressiotestaus tulee kaikkien testien suorittamista edullisemmaksi vain siinä tapauksessa, että sopivien testitapausten valinnan sekä niiden suorittamisen aiheuttama kustannus jää pienemmäksi kuin kaikkien testitapausten suorittaminen.

Regressiotestaus voidaan jakaa kahteen erilaiseen testaukseen (Leung & White, 1989): Progressiivinen (progressive) regressiotestaus tarkoittaa alkuperäisten testien suorittamista muuttunutta sovellusta vasten. Sen sijaan korjaava (corrective) regressiotestaus tarkoittaa regressiotestausta, jossa ohjelmistokoodin lisäksi vaatimukset tai määritykset ovat muuttuneet. Tällöin myös testitapauksia tulee muuttaa, jotta ne testaavat järjestelmää muuttuneiden vaatimusten mukaisesti.

Koska regressiotestaus on raskasta työtä, se tulee erittäin kalliiksi, jos sitä ei automatisoida (Haikala & Märijärvi, 2003). Regressiotestaus sopiikin hyvin automatisoinnin kohteeksi luonteensa vuoksi, sillä regressiotestauksessa toistetaan samoja asioita useaan kertaan. Automatisoinnin kohteena voi testien suorittamisen lisäksi olla esimerkiksi testitapausten luonti tai muokkaus. Koodimuutoksen jälkeen testaajan tulee suorittaa vanhoja testejä uudelleen, jotta mahdolliset muutoksesta johtuvat virheet voidaan havaita (Marick, 1998). Tällaisissa tapauksissa automatisoinnista on merkittävää hyötyä, sillä muutoksia tulee usein hyvin paljon, minkä vuoksi testien ajokertoja on useita.

2.5 Graafisten käyttöliittymien testaamisen erityispiirteitä

Graafisia käyttöliittymiä voidaan toteuttaa lukuisilla eri tavoilla ja niiden toiminta voi vaihdella merkittävästikin, mikä aiheuttaa testaamiselle erityisiä haasteita (Hackner & Memon, 2008). Lisäksi graafiset käyttöliittymät luovat ohjelmistokoodin päälle vielä yhden kerroksen, joka voi itsessäänkin jo sisältää virheitä integraatiovirheiden lisäksi. Atif Memonin, Adithya Nagarajanin ja Qing Xien (2005) mukaan nykyisin 45–60 prosenttia ohjelmistokoodista koskee graafisia käyttöliittymiä. Graafisten käyttöliittymien suosion kasvaessa niiden testaamisen tarve on myös lisääntynyt, joskin niiden testaamisen tutkiminen on vielä verrattain tuore alue. Graafisten käyttöliittymien testaamiseen ei voida käyttää perinteisiä tekniikoita, vaan ne vaativat omat, erikoistuneet välineensä (Memon, Soffa & Pollack, 2001). Lee White (1996) antaa graafisten käyttöliittymien testauksen erilaisuudesta esimerkiksi tilanteen, jossa syöte on interaktiivinen ja tulos voi olla tapahtuma tai jokin graafinen ominaisuus. Graafisten käyttöliittymien mukana on tullut uusia virheitä sekä monimutkaisuutta ja niitä on vaikeampi testata (Gerrard, 1997). Graafisten käyttöliittymien testaamattomuus heijastuu kuitenkin myös ohjelmiston kokonaislaatuun.

Memonin ym. (2005) mukaan on olemassa käytännössä kolmenlaista automaattista graafisten käyttöliittymien testaamista päivittäisten versioiden (*nightly build*) yhteydessä: Yleisin niistä on valitettavasti se, että graafista käyttöliittymää ei testata lainkaan. Se aiheuttaa mahdollisia ongelmia ohjelmiston laatuun tai aiheuttaa kallista käyttöliittymätestausta myöhemmin. Toinen tapa on

luoda erillinen testikehys, joka kutsuu ohjelmistokoodia ikään kuin kutsu tulisi käyttöliittymältä. Tämän menetelmän käyttäminen vaatii kuitenkin mahdollisesti merkittäviä muutoksia sovelluksen arkkitehtuuriin eikä se myöskään testaa sovellusta loppukäyttäjän näkökulmasta. Kolmas tapa on automatisoida testaus käyttämällä erilaisia työkaluja, joilla voidaan tallentaa ja toistaa graafiseen käyttöliittymän käyttöä. Näillä työkaluilla ei kuitenkaan päästä täydellisiin tuloksiin mutta niistä on kuitenkin selvästi apua esimerkiksi savutestauksessa. Tämän tutkimuksen empiirisessä osuudessa käytetäänkin juuri tällaista tapaa graafisten käyttöliittymien testaamisen automatisointiin.

Graafisen käyttöliittymän testaamisessa voidaan havaita lisäksi sellainen ongelma, että sen alla olevaan koodiin ei välttämättä päästä käsiksi. Testauksessa onkin mahdollista, että ohjelmiston yhteyksien löytäminen pelkän käyttöliittymäkerroksen avulla voi olla haastavaa. Marick (1998) mainitsee esimerkkinä tapauksen, jossa käyttöliittymä pysyy samana kuin ennen mutta taustakoodiin tuleva muutos muuttaa ohjelmiston toimintaa. Voi kuitenkin olla, että toiminnallisuuden päälle rakennettu testi menee edelleen hyväksytysti läpi mutta tosiasiassa ohjelmisto toimiikin virheellisesti muutoksen jälkeen.

Testitapauksia voi olla hankala luoda manuaalisesti kattavasti, sillä graafisten käyttöliittymien tapahtumapohjaisen luonteen vuoksi mahdollisten tapausten ja polkujen määrä kasvaa merkittävästi. Hacknerin ja Memonin (2008) mukaan testitapausten luontitekniikat vaativatkin niihin soveltuvia työkaluja mutta niiden koko potentiaalin hyödyntäminen voi olla hankalaa. Memon ym. (2001) esittelevät erään graafisten käyttöliittymien tapahtumaohjattuun luonteeseen perustuvan suunnitellun kriteerien valintamenetelmän. Valintamenelmä perustuu käyttöliittymästä tunnistettuihin komponentteihin ja niiden välisiin suhteisiin sekä polkuihin. Memon ym. käyttivät tätä menetelmää myöhemmin GUITAR-viitekehyksessään (GUI Testing frAmewoRk), jota tarkastellaan lähemmin osiossa 3.3.

2.6 Testauksen automatisointi

Testaaminen voidaan automatisoida jollakin tasolla. Naina Mittalin ja Ira Acharyan (2003) mukaan on hyvä pyrkiä automatisoimaan mahdollisimman paljon mutta on kuitenkin otettava huomioon automatisoinnin käyttöönotosta aiheutuvat kulut. Niinpä automatisointi kannattaa ottaa käyttöön niissä tilanteissa, joissa testaaminen kuluttaa eniten aikaa ja muita resursseja. Erityisesti silloin, jos julkaistavaan sovellukseen voidaan odottaa tulevan ylläpitotehtäviä, kuten korjauksia, päivityksiä tai muutoksia, regressiotestaus kannattaa automatisoida.

Testauksen automatisointiin on erilaisia perusteita ja siinä on myös huomioitava ihmisen ja koneen väliset erot. Tietokone voi havaita testauksessa asioita, jotka ihmiseltä jäisivät mahdollisesti kokonaan huomaamatta tai niiden havaitseminen veisi paljon aikaa. Marick (1998) mainitsee esimerkkinä pitkät desimaaliluvut, joissa virhe voi piillä vaikkapa seitsemännessä desimaalissa.

Kone havaitsee virheen nopeasti suurestakin joukosta mutta ihminen voi tehdä testatessaan itsekin virheen ja olla havaitsematta ongelmaa.

Toisaalta ihminen voi havaita sellaisia virheitä, joita kone ei havaitse. Tällaisia voivat olla esimerkiksi käytettävyysongelmat. Koneelle ei ole merkitystä, onko käyttöliittymän ponnahdusdialogi ruudulla vai ruudun ulkopuolella, mutta ihminen havaitsee virheen heti, mikäli dialogi ei ole näkyvissä. Ihmiset eivät myöskään toimi aina samalla tavalla, joten uudelleenyrityksistä ja eri sekvensseistä johtuen voi tulla ilmi sellaisia virheitä, joita kone, joka suorittaa testit aina samalla tavalla, ei havaitse.

Manuaalisen testaamisen ongelmana voi olla se, että virheen löytyessä ongelmaa ei enää saada toistettua, mikäli testitapauksen vaiheita ei ole kirjattu ylös, sillä häiriön ilmetessä ei välttämättä enää muisteta, miten virhetilanteeseen on päädytty. Automaattisten testien kanssa tätä ongelmaa on harvoin, sillä näissä testeissä kone suorittaa testit aina samalla tavalla ja lisäksi usein testien eteneminen voidaan tarkistaa jälkikäteen vaihe vaiheelta. Mittal ja Acharya (2003) esittävätkin yhdeksi manuaalisen testauksen ongelmista tulosten järjestelmällisen kirjaamisen. Lisäksi manuaalinen testaaminen on heidän mukaansa yleensä epäjärjestelmällistä.

Marick (1998) väittää, että mikäli organisaatiossa tai projektissa testauksen kypsyys on tarpeeksi korkealla tasolla, voi olla taloudellisempaa, että kehittäjät suorittavat automaattiset testit itse todetakseen löytyneen ongelman kuin se, että erillinen testaaja kirjoittaisi löytyneestä ongelmasta virheraportin. Marick kuitenkin jatkaa, että näin tehokas testauksen automatisointi sekä se, että kehittäjien välineet testien suorittamiseen ovat tarpeeksi hyvässä kunnossa, on hyvin harvinaista.

Automaattisten testien kirjoittamisessa on Marickin (1998) mukaan lisäksi muistettava, että testien kirjoittaminen automatisoinnin näkökulmasta voi vähentää testin toimivuutta virheiden löytämisessä. Regressiotestauksen näkökulmasta tällä ei kuitenkaan ole niin suurta merkitystä, sillä sen tarkoituksena ei ole löytää uusia virheitä, vaan varmistaa, että ohjelmisto toimii edelleen samalla tavalla kuin ennenkin.

Dianxiang Xun, Weifeng Xun, Bharath K Bavikatin ja W. Eric Wongin (2012) mukaan web-pohjaisten sovellusten graafisten käyttöliittymien automaattiseen testaamiseen käytetään tavallisesti jotakin tallenna-ja-toista-työkalua. Eräs tällainen työkalu on Selenium IDE, jota käytetään tämän tutkielman empiirisessä osassa graafisten käyttöliittymien automatisointiin.

2.6.1 Testauksen automatisoinnin perusteleminen

Testitapauksen automatisointi ei ole aina järkevää, sillä usein se voi olla kalliimpaa kuin vastaavan testitapauksen suorittaminen manuaalisesti. Marick (1998) esittää seuraavat kolme kysymystä, joiden avulla automatisointipäätös on helpompi toteuttaa:

(1) Kuinka paljon enemmän testitapauksen automatisointi ja suorittaminen kerran maksaa verrattuna sen suorittamiseen kerran manuaalisesti?
(2) Automaattisella testillä on rajattu elinikä. Missä tilanteissa tämä testitapaus todennäköisesti muuttuu käyttökelvottomaksi?
(3) Testin elinaikana, kuinka todennäköistä on, että se löytää lisää virheitä niiden lisäksi, jotka se löysi ensimmäisellä kerralla?

Nämä kysymykset auttavat päättämään testauksen automatisoinnista mutta regressiotestauksen näkökulmasta näistä ainoastaan toinen kysymys on merkittävä. Kysymys yksi menettää merkityksenä siinä, että testitapaus suoritettaisiin vain kerran ja kysymys kolme siinä, että regressiotestauksessa myöhemmillä suorituskerroilla pyritään varmistamaan, että uusia virheitä ei löydy. Näin ollen kysymykset voitaisiin muuttaa regressiotestaukseen paremmin sopiviksi seuraavasti:

(1) Kuinka paljon enemmän tai vähemmän testitapauksen automatisointi maksaa verrattuna sen suorittamiseen manuaalisesti aina ohjelmiston muuttuessa?
(2) Automaattisella testillä on rajattu elinikä. Missä tilanteissa tämä testitapaus todennäköisesti muuttuu käyttökelvottomaksi?
(3) Testin elinaikana, kuinka todennäköistä on, että sen avulla voidaan varmistua, että ohjelmistokoodin muutos ei ole luonut aiemmin toimineeseen ohjelmistoon lisää virheitä?

Todennäköisesti vastaus ensimmäiseen kysymykseen muuttuu nyt negatiiviseksi, jolloin automatisointi olisi kannattavampaa kuin manuaalinen testaaminen. Marickin (1998) mukaan erityisesti graafisten käyttöliittymien tapauksessa, jos testitapauksen automatisointi suoritetaan kirjoittamalla skriptejä, automatisointi voi olla useita kertoja kalliimpaa kuin saman testin suorittaminen manuaalisesti. Tallenna-ja-toista-työkalujen avulla laadittavat testitapaukset sen sijaan voivat olla huomattavasti edullisempia toteuttaa mutta niiden elinikä voi olla lyhyt. Gerrard (1997) lisää, että myös regressiotestien automatisoinnissa kannattaa pyrkiä tekemään testeistä sellaisia, joilla voidaan löytää virheitä myös kehityksen alkuvaiheessa, jolloin niiden hyötyjä voidaan korostaa. On kuitenkin muistettava, että regressiotestaukseen käytettävät testitapaukset ovat ohjelmiston osan valmistuessa edelleen valideja.

Regressiotestien elinkaaren päättyminen on epätodennäköisempää kuin tavallisten automaattisten testien, sillä muuttuva ohjelmistokoodi tehdään ohjelmiston muihin osiin. Näin ollen regressiotestit toimivat todennäköisemmin myös muutoksen jälkeen suoraan.

Marick (1998) mainitsee, että mikäli automaattisia testiskriptejä kirjoitetaan jo ennen kuin ohjelmisto on valmis testattavaksi, siihen käytettyä aikaa ei voida pitää ylimääräisenä, sillä tällöin ei tarvitse tasapainotella manuaalisten testien ja automaattisten testien välillä. Tällöinkin tulee kuitenkin muistaa, että testien toimintaan saattamiseen voi kulua aikaa ohjelmiston valmistuttua.

Gerrard (1997) suosittelee nyrkkisäännöksi Pareton periaatteen mukaisesti, että automatisoimalla 20 prosenttia testeistä saavutettaisiin 80 prosentin hyöty. Tämä tarkoittaa, että ei käytetä aikaa siihen, että kirjoitetaan vähän käytettäviä monimutkaisten testitapausten skriptejä, vaan keskitytään kirjoittamaan paljon käytettäviä yksinkertaisempia skriptejä.

Mira Kajko-Mattsson, Marcus Jonson, Saam Koroorian ja Fredrik Westin (2004) suosittelevat, että mikäli organisaatiolla on käytössään päivittäiset versiot, eli sovellus paketoidaan ja asennetaan säännöllisin väliajoin, testauksen pitää olla automaattista, jotta päivittäisten versioiden käytöstä saadaan hyöty irti.

2.6.2 Regressiotestien automatisointi

Regressiotestien tarkoitus on varmistaa, että ohjelmisto toimii oikein myös sen jälkeen, kun ohjelmiston jotakin osaa on muutettu. Nykyisin käytetyimmät testauksen automatisoinnin työkalut ovat tallenna-ja-toista-tekniikan sovelluksia, jossa testitapaukset tallennetaan samalla, kun testaaja suorittaa testiä manuaalisesti. Tämän jälkeen testi voidaan toistaa automaattisesti siten kuin testaaja sen aiemmin teki (Memon, 2002). Yleisiä ovat myös skriptaamiseen perustuvat testigeneraattorit, jotka suorittavat käyttöliittymän testaamisen ohjelmoitujen ohjeiden perusteella.

Regressiotestien kattavuus pitää päättää niitä suunniteltaessa. Regressiotestit voivat olla esimerkiksi savutestejä, joilla varmistetaan, että kriittiset ohjelmiston osat toimivat. Näitä testejä voidaan suorittaa automaattisesti aina uuden ohjelmistovedoksen (build) julkaisun jälkeen. Regressiotestien kirjoittamisen yhteydessä kannattaa huomioida niiden kustannukset suhteessa hyötyihin. Memonin (2002, 2007) mukaan juuri tehokkaimman testijoukon valinta on erityisen haastavaa. Toisaalta testaussyklin tehostaminen ja manuaalisesta, toistuvasta samojen asioiden testaamisesta johtuvan turhautumisen vähentäminen ovat panostamisen arvoisia. Lisäksi on hyvä muistaa, että automaatti jaksaa aina suorittaa kaikki testitapaukset väsymättä läpi vaikka kuinka monta kertaa.

Koska automaattiset regressiotestit voidaan suorittaa päivittäin, voidaan mahdollisesti syntyneet virheet löytää nopeammin. Virheen synnyttänyt koodimuutos on helpompi löytää, sillä muutoksen tiedetään syntyneen edellisen testiajon jälkeen. Mikäli testit on kirjoitettu hyvin, ne voidaan ajaa usealla eri sekvenssillä eri ajokerroilla, jolloin mahdollisia virheitä voidaan löytää tehokkaammin. Automaattisilla testeillä voidaan myös saavuttaa parempi tehokkuus satunnaistamisella kuin manuaalisilla testeillä. (Marick, 1998). Erilaisia sekvenssejä voidaan luoda helposti ja nopeasti esimerkiksi tapahtumavuokaaviomallilla (event-flow model) (Memon, 2007), joka on osa DARTia ja GUITARia. DART ja GUITAR esitellään tarkemmin kohdissa 3.2 ja 3.3.

Regressiotesteissä käytettäviä testitapauksia on silloin tällöin korjattava, mikäli testit eivät ole enää käyttökelpoisia. Testitapausten korjaaminen uusien luonnin sijaan voi olla halvempaa, varsinkin jos käytössä on menetelmä, joka korjaa testitapaukset automaattisesti (Memon & Soffa, 2003). Erityisen tärkeää tämä on tyypillisessä tapauksessa, jossa kehittäjät kehittävät graafisia käyttöliit-

tymiä nopeasti ja regressiotestitapauksien on mukauduttava jatkuvasti käyttöliittymien muutoksiin (Memon, 2002).

2.6.3 Graafisten käyttöliittymien automaattisen testaamisen ongelmia

Graafisten käyttöliittymien testaamisen automatisointiin liittyy lukuisia ongelmia. Marick (1998) antaa esimerkiksi tilanteen, jossa käyttäjän tulee syöttää puhelinnumero. Mikäli puhelinnumero on ennen kysytty esimerkiksi tekstikenttänä ja testi on kirjoitettu sitä varten, se ei enää toimi, mikäli puhelinnumero annettaisiinkin hiirellä virtuaalista graafista puhelimen näppäimistöä klikkailemalla. Tällöin automaattinen testi ei toimi enää käyttöliittymän muuttuessa, vaikka syötettävä data olisikin sama. Leotta ym. (2013) mainitsevatkin yhdeksi ongelmaksi nimenomaan graafisten käyttöliittyminen nopean muuttumisen.

Graafisten käyttöliittymien regressiotestauksen automatisointia varten voi olla järkevää ottaa tällaiset tilanteet huomioon ja esimerkiksi rakentaa tuotekohtainen testikirjasto, joka hallitsee käyttöliittymän käytön. Tällöin testitapaus voisi olla esimerkiksi ”try 0401234567”. Testikirjasto huolehtii sen syöttämisestä joko tekstikenttään tai graafisella näppäimistöllä riippuen siitä, kumpi on käytössä. Testikirjaston etuna on lisäksi se, että usein testitapauksiin ei tarvitse tehdä muutoksia, vaikka käyttöliittymä muuttuu, vaan muutosten tekeminen testikirjastoon riittäisi. (Marick, 1998). Näin ollen testi toimii riippumatta siitä, millaiseen käyttöliittymään se syötetään.

Memon (2002) mainitsee ongelmaksi sen, että vaikka esimerkiksi tallenna-ja-toista-tekniikalla voidaan suorittaa monimutkaisiakin automaattisia testejä, niiden laatiminen on erittäin työlästä. Nykyisten ja monimutkaisten käyttöliittymien kohdalla tämä on erityisen haasteellista, sillä kaikkia mahdollisia reittejä ei voida ottaa huomioon. Toisaalta yksi tallenna-ja-toista-tekniikalla luotava testitapaus vastaa kustannuksiltaan suunnilleen yhtä manuaalista testiä. Kun testi on kerran tallennettu, se voidaan suorittaa aina uudelleen niin kauan kuin testitapaus on validi. Leotta ym. (2003) lisäävät, että pelkästään tallenna-ja-toista-tekniikalla luotavat testitapaukset ovat yleensä hauraita ja menevät helposti käyttökelvottomiksi jo pienilläkin käyttöliittymän muutoksilla.

Graafiset käyttöliittymät ovat erittäin epävakaita (Marick, 1998). Gerrard (1997) mainitsee graafisten käyttöliittymien ongelmaksi sen, että ne tuovat testaamiseen uusia ulottuvuuksia, jotka pitää ottaa testauksessa huomioon. Esimerkiksi graafisten käyttöliittymien tapahtumaohjattu luonne (event-driven) tekee testaamisesta vaikeaa ja lisää virheiden mahdollista määrää, sillä kaikkien mahdollisten polkujen läpikäynti ei ole mahdollista. Ohjelmoijan on voitava hallita suuria määriä mahdollisia tilanteita, joissa tapahtuma voidaan kutsua (esim. käyttäjä klikkaa hiirellä ruudun eri kohtia). Koska kaikkien mahdollisten tilanteiden huomioonottaminen on haasteellista, virheiden määrä kasvaa. Automatisoinnilla kuitenkin voidaan parantaa tätä, sillä kone voi käydä useampia polkuja läpi lyhyemmässä ajassa. Mikäli polut saadaan vielä luotua automaattisesti (esim. DART ja GUITAR), voidaan päästä hyvinkin kattaviin testeihin.

Koska automaattiset testit suoritetaan koneellisesti, voi jokin virhe mennä automaattiselta testiltä läpi, vaikka ihminen havaitsisi sen heti virheeksi. Tällainen voisi olla esimerkiksi tilanne, jossa ponnahdusikkuna piirtyy ruudun ulkopuolelle eikä ihminen voisi sitä sen vuoksi nähdä. Koneelle se ei kuitenkaan ole ongelma, vaan se käsittelee dialogin normaalisti. Näin selvä virhe jää havaitsematta.

Marick (1998) kertoo havainneensa, että manuaalisella testaamisella havaittiin esimerkiksi hiiren kursorin outo käyttäytyminen sitä liikuteltaessa. Ongelman tarkemmassa tutkimuksessa havaittiin virhe, jota automaattinen testi ei löytänyt. Gerrard (1997) lisää testauksen ongelmaksi myös objektien väliset synkronoinnit. Tällöin testaajan, oli se sitten automaatti tai ihminen, tulee voida ottaa huomioon, että esimerkiksi valintaruudun aktivointi saattaa muuttaa muiden objektien toimintaa. Lisäksi on huomioitava, että usein samat valinnat voidaan tehdä monella eri tavalla, näppäimistöllä, hiirellä tai toimintopainikkeilla.

Memonin (2002) mukaan jo graafisten käyttöliittymien kehittämisessä tulisi huomioida niiden yhdenmukaisuus. Näin voitaisiin välttyä pahimmilta sudenkuopilta.

2.7 Sanasto

Tässä aliluvussa esitellään lyhyesti termejä, joita tutkimuksessa on käytetty. Ellei toisin ole mainittu, suomennokset on otettu ISTQB:n (International Software Testing Qualifications Board) testaussanastosta (ISTQB, 2007).

Virheet
Ohjelmistokoodin yhteydessä puhutaan usein bugeista, kun ohjelma ei toimi halutulla tavalla. Termi bugi ei kuitenkaan kata ohjelmistotestauksessa löydettyjen ongelmien koko kirjoa. Ongelmat voidaan jakaa kolmeksi eri termiksi niiden merkityksen ja syyn mukaan:

(1) *Virhe* (*error*): Ihmisen toiminta, joka tuottaa väärän tuloksen.
(2) *Vika/bugi* (*fault/defect/bug*): Komponentissa tai järjestelmässä oleva virhe, joka voi aiheuttaa sen, että komponentti tai järjestelmä ei pysty suorittamaan siltä edellytettävää toimintoa, esim. virheellinen lauseke tai muuttujan määrittely. Jos virhe kohdataan suorituksen aikana, se voi aiheuttaa komponentin tai järjestelmän häiriön.
(3) *Häiriö* (*failure*): Ohjelmiston poikkeama odotetusta toimituksesta, palvelusta tai tuloksesta. Häiriö esiintyy suorituksen aikana.

Yhteenvetona voidaan todeta, että ihminen tekee *virheen*, esimerkiksi tulkitsee vaatimusmäärittelyä virheellisesti. Tämän seurauksena ohjelmakoodiin ilmestyy *vika*, joka voidaan havaita suorituksen aikana *häiriönä*.

Testipeti
Testipeti (*test bed*) on ohjelmisto tai laite, jonka avulla voidaan korvata puuttuvaa laitteistoa tai ohjelmistoa jonkin osakokonaisuuden testaamisessa. Testipetiä voidaan kutsua myös testikehykseksi.

Testattava ohjelmisto
Testattava ohjelmisto (*Application under test*, AUT) on testauksen kohteena oleva ohjelmisto tai sen osa.

Testijoukko
Testijoukko (*test suite*) on komponentin tai järjestelmän testaamisessa käytettävä usean testitapauksen joukko, jossa edellisen testin jälkiehtoja käytetään usein seuraavan testin esiehtoina. Testijoukosta voidaan käyttää myös termiä testisetti tai testitapausjoukko.

Testitapaus
Testitapauksella (*test case*) tarkoitetaan syötearvojen, suorituksen esiehtojen, odotettujen tulosten ja suorituksen jälkiehtojen muodostamaa kokonaisuutta, joka on muodostettu tiettyä tavoitetta tai testauksen kohdetta varten, esim. tietyn ohjelmapolun testaukseen tai vaatimustenmukaisuuden varmistamiseksi.

Uudelleentestaus
Uudelleentestaus suoritetaan silloin, kun ohjelmistosta löydetty virhe on korjattu. Virheen korjaamisen jälkeen suoritetaan epäonnistuneet testitapaukset uudelleen ja mikäli ne menevät uudelleentestauksessa läpi, virhe on korjattu ja ohjelmisto toimii kuten pitääkin. On myös mahdollista, että testitapauksia on muokattava vastaamaan korjattua toiminnallisuutta erityisesti silloin, jos vaatimukset ovat päivittyneet.

Uudelleentestaus eroaa regressiotestauksesta siten, että sillä varmistetaan, että aiemmin havaittu ja raportoitu virhe on korjattu. Regressiotestauksella puolestaan varmistetaan, että korjaus ei vaikuttanut muuhun toimintaan sovelluksessa tai koko järjestelmässä. Uudelleentestaamisen yhteydessä tuleekin aina suorittaa myös regressiotestausta.

Testioraakkeli
Testioraakkelilla (*test oracle*) tarkoitetaan ohjelmaa, jolla voidaan verrata testituloksia oikeisiin tuloksiin. Perinteisesti testioraakkeli suoritetaan testien valmistuttua ja tällöin oraakkeli vertaa testauksessa saatuja tuloksia odotettuihin tuloksiin. Mikäli tulokset eroavat toisistaan, testi on epäonnistunut. Testioraakkeli voidaan jakaa informaatio- ja proseduraaliseen osaan. Informaatio-osa kuvaa odotettuja tuloksia ja proseduraalinen vertaa niitä testauksessa saatuihin tuloksiin. Memon, Banerjee ja Nagarajan (2003a) osoittavat tutkimuksessaan, että testioraakkelilla ja sen valinnalla voi olla merkittävä vaikutus testauksen kokonaiskustannuksiin.

Tapahtumaohjattu ohjelmisto
Yksi graafisten käyttöliittymien testaamiseen liittyvistä suurimmista ongelmista johtuu *tapahtumaohjatuista ohjelmistoista* (*event-driven software*). Tapahtumaohjatulla lähestymistavalla tarkoitetaan ohjelmointia, jossa ohjelman suoritus määräytyy erilaisten tapahtumien perusteella. Koska graafisten käyttöliittymien ohjelmoinnissa käytetään usein juuri tapahtumaohjattua ohjelmointia, ohjelmoijan on voitava hallita suuria määriä mahdollisia tilanteita, joissa tapahtuma voidaan kutsua (esim. käyttäjä napsauttaa hiirellä ruudun eri kohtia). Koska kaikkien mahdollisten tilanteiden huomioonottaminen on haasteellista, virheiden määrä kasvaa. Samalla myös kaikkien mahdollisten tapahtumakutsujen testaaminen on mahdotonta. (Gerrard, 1997).

Savutestaus
Savutestaus (*smoke testing*) on testauksen muoto, jossa järjestelmän kriittiset osat testataan pintapuolisesti sen varmistamiseksi, että järjestelmässä ei ole mitään selvää ongelmaa. Savutestauksessa ei siis testata jokaista järjestelmän osaa, vaan keskitytään lähinnä siihen, että sovellukset esimerkiksi lähtevät käyntiin eivätkä näytä heti virheilmoituksia ja että järjestelmä näyttää päällisin puolin olevan toimintakunnossa. Mm. Ian Molyneauxin (2009) mukaan termi on alun perin lähtöisin komponenttiteollisuudesta, jossa komponentin testaus meni läpi, jos siitä ei tullut savua, kun se kytkettiin järjestelmään.

2.8 Yhteenveto

Tässä luvussa käsiteltiin testausta yleisellä tasolla sekä esiteltiin siihen liittyviä käsitteitä ja sanastoa. Luvussa todettiin, että ohjelmistotestauksella pyritään varmistamaan, että toteutettu järjestelmä tai sovellus on valmis julkaistavaksi eli että se toimii halutulla tavalla. Testauksella ei kuitenkaan voida todistaa ohjelmiston virheettömyyttä. Ohjelmistotestaus on kallista ja voi käsittää yli puolet ohjelmistoprojektin kustannuksista. Graafisten käyttöliittymien monimutkaistuminen aiheuttaa testaamiselle lisäksi omat haasteensa. Niiden tapahtumaohjatun luonteen vuoksi mahdollisten käsittelypolkujen määrä kasvaa hyvin nopeasti niin suureksi, että kaikkien mahdollisten polkujen testaaminen on mahdotonta. Testauksen kalleudesta johtuen ei olekaan tavatonta, että testausta ei tehdä niin hyvin kuin pitäisi, vaan se jätetään vähemmälle huomiolle ohjelmointitehtävien viedessä huomiota. Testaus on kuitenkin laadukkaan ohjelmiston perusedellytys.

Regressiotestausta tulee suorittaa aina, kun ohjelmistoon on tehty muutos ja sen tarkoituksena on varmistaa, että ohjelmisto toimii samalla tavalla myös niiltä osin, joihin muutosta ei ole tehty. Regressiotestaamisen ei siis ole tarkoitus löytää enää uusia häiriöitä vaan varmistaa, että järjestelmä ei ole mennyt rikki siihen tehtyjen muutoksien vuoksi.

Testaamisen kustannuksia voidaan pyrkiä madaltamaan automatisoinnin avulla. Automatisointi sopii erityisen hyvin regressiotestaukseen, sillä siinä samoja testitapauksia suoritetaan useita kertoja. Kaikkia testitapauksia ei tarvitse käyttää regressiotestaukseen, mutta iso osa muuhun testaukseen käytetyistä testitapauksista sopii myös siihen. Testitapausten valinta onkin yksi suurimmista ongelmista regressiotestauksen tehokkuuden parantamisessa. Testauksen automatisointi tulee manuaalista testaamista edullisemmaksi vain silloin, jos testitapausten valinnasta aiheutuva kustannus on pienempi kuin valittujen testien automaattinen suorittaminen. Lisäksi on huomioitava, että regressiotestien ylläpitäminen vaatii resursseja ja aiheuttaa kustannuksia.

Graafisten käyttöliittymien automaattiseen testaamiseen on kehitetty erilaisia työkaluja, joista tallenna-ja-toista-tyyppiset ohjelmat ovat eniten käytettyjä. Niiden avulla testien tekeminen on edullista sekä nopeaa. Näin tehdyt testitapaukset ovat kuitenkin hauraita eivätkä yleensä kestä suuria käyttöliittymämuutoksia. Jos testejä kuitenkin tehdään regressiotestaukseen, voidaan näiden testien ajatella olevan riittäviä, koska oletuksena on tällöin nimenomaan se, että muutoksia ei tapahdu juuri näiden testien vaikutusalueelle. Tämän tutkielman empiirisessä osuudessa käytetään testauksen automatisointiin juuri tällaista tallenna-ja-toista-työkalua, mutta itse testitapaukset tehdään kuitenkin manuaalisesti kirjoittamalla testirivejä skriptillä, sillä näin testit saadaan varmistamaan monimutkaisempia tapauksia.

3 KIRJALLISUUDESSA ESITETTYJEN MALLIEN JA TEKNIIKOIDEN ESITTELY

Tässä luvussa esitellään kirjallisuudessa esitettyjä malleja ja tekniikoita, joiden avulla automaattista ohjelmistotestausta voidaan suorittaa. Kirjallisuuskatsauksen perusteella valitaan tutkielman empiirisen osuuden kokeelliseen tutkimukseen sopiva malli ja käytetään sitä pilotoimaan projektimme tuottaman järjestelmän yhden osan graafisen käyttöliittymän automaattinen regressiotestaus.

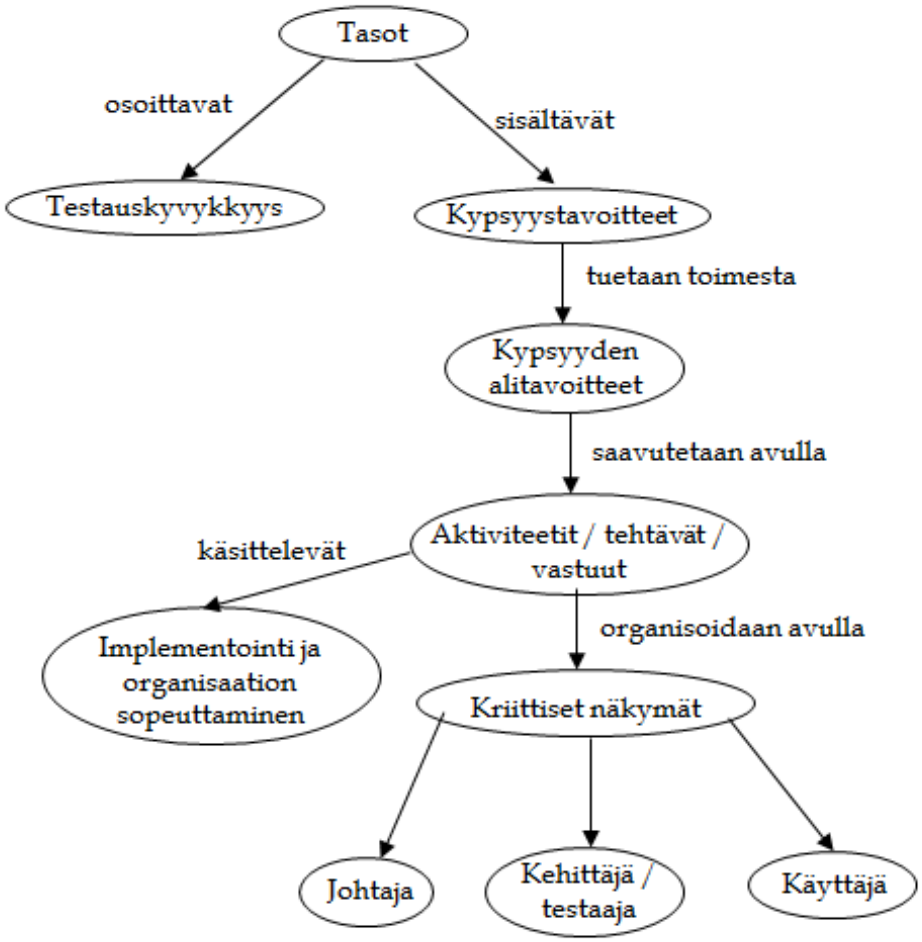
3.1 Testauksen kypsyysmalli

Ilene Burnstein, Taratip Suwanassart ja Robert Carlson (1996) kehittivät testauksen kypsyysmallin, Testing Maturity Modelin (TMM). Sen avulla voidaan määrittää organisaation kypsyysaste testauksen osalta. Malli perustuu vastaavaan prosessien kypsyysmalliin Capability Maturity Modeliin (CMM) mutta käsittää testauksen prosessit. Malli keskittyy nimenomaan testausprosessiin, jonka parantuminen voi parantaa vastaavasti ohjelmistojen laatua.

Malli tarjoaa kuvauksen jokaisesta kypsyystasosta sekä hyviä käytäntöjä jokaiselle tasolle. Lisäksi malli tarjoaa organisaatioille työkaluja omien prosessiensa arvioimiseen ja parantamiseen. Mallissa on viisi tasoa: *alkutilanne* (*initial*), *vaihemäärittely* (*phase-definition*), *integraatio* (*integration*), *hallinnointi ja mittaaminen* (*management and measurement*) sekä *optimointi, ennaltaehkäisy ja laadunvarmistus* (*optimization/defect prevention and quality control*). Mallin mukaan regressiotestauksen automatisointityökalut tulevat mukaan tasolla 2.

Kuvio 2 esittää testauksen kypsyysmallin viitekehyksen. Viitekehykseen on määritetty kullekin kypsyystasolle kypsyystavoitteet, niiden alitavoitteet sekä aktiviteetit, tehtävät ja vastuut. Kypsyystavoitteet näyttävät jokaisen tason avainprosessit, jotka organisaation on hallittava seuraavalle tasolle pääsemiseksi. Jokaisella kypsyystavoitteella on sitä tukevat alitavoitteet, jotka ovat tarkempia kuin perustavoitteet, ja ne määrittelevät rajoitukset ja laajuuden sekä tasolla vaadittavat saavutukset. Alitavoitteet saavutetaan aktiviteettien, tehtävien ja

vastuiden kautta. Aktiviteetit ja tehtävät määrittävät jokaisella tasolla vaadittavat tehtävät. Näiden tehtävien vastuut jaetaan johtajien, kehittäjien ja testaajien sekä käyttäjien kesken. Viitekehyksessä nämä ryhmät on määritetty kriittisiksi näkymiksi.



KUVIO 2 TMM:n viitekehys (Burnstein ym., 2006, s. 583)

Johtajan tehtäviä ovat testauksen laadun kehittämistehtäviin sitoutuminen ja niiden mahdollistaminen. Kehittäjän ja testaajan tehtävät puolestaan käsittävät teknisiä aktiviteetteja ja tehtäviä, joiden käyttäminen perustuu kypsille testauskäytännöille. Käyttäjille jää laatuun liittyvät aktiviteetit, kuten vaatimusten analysointi sekä käytettävyys- ja hyväksymistestaus.

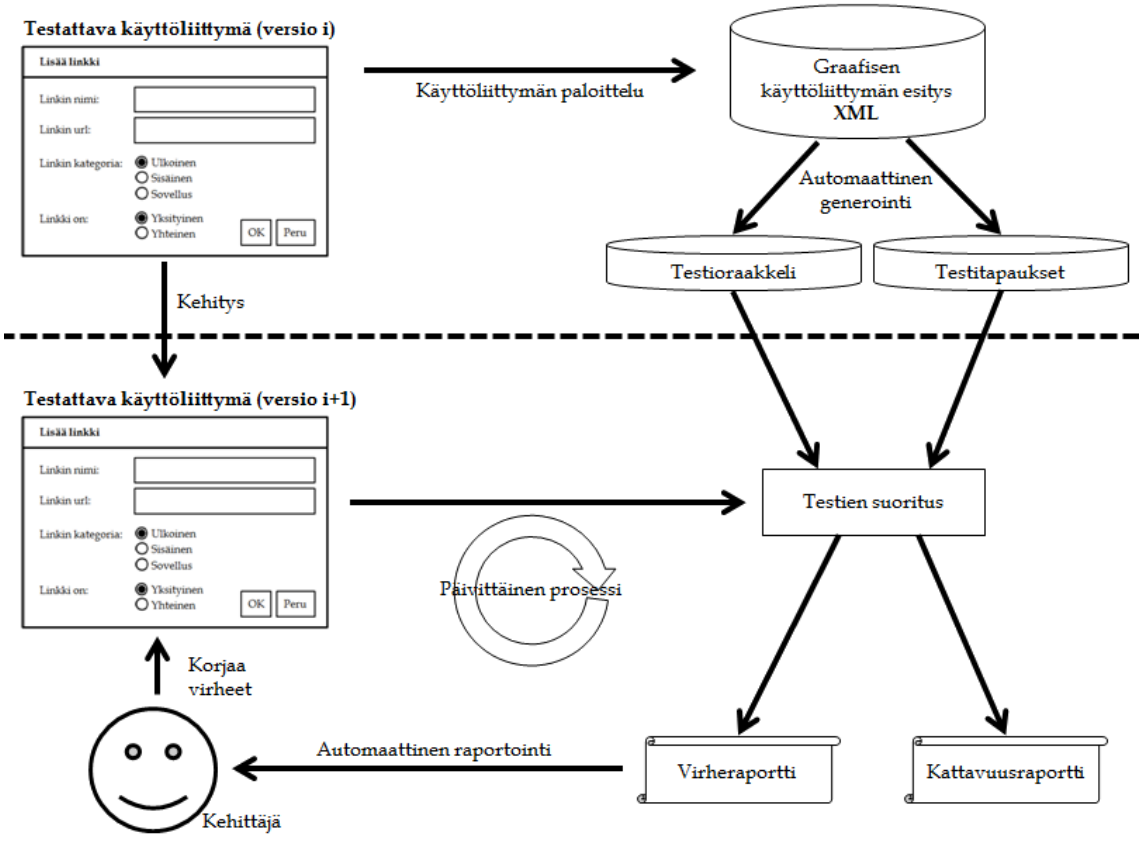
3.2 Daily Automated Regression Tester

Atif Memon on esitellyt yhdessä Ishan Banerjeen, Nada Hashmin ja Adithya Nagarajanin (2003b) kanssa Daily Automated Regression Testerin (DART), joka on viitekehys, jolla voidaan automatisoida graafisten käyttöliittymien testausprosessi. Prosessi sisältää kaiken aina automaattisesta graafisen käyttöliittymän

paloittelemisesta, testitapausten luonnista testioraakkelin luontiin ja hajonneiden testitapausten korjaamiseen ja uudelleensuorittamiseen.

DARTin kantava ajatus on koko testausprosessin automatisointi. Memon (2002) ilmaisee huolensa siitä, että manuaalisella testaamisella ei voida saavuttaa tarpeeksi hyvää kattavuutta, vaikka itse testien suorittaminen olisikin automaattista, jos testitapausten suunnittelu ja päivittäminen kuitenkin on manuaalista.

Kuviossa 3 on esitetty DARTin korkean tason prosessi. Katkoviivan yläpuolella on kertaluontoinen alkumuodostelma ja alapuolella iteratiivinen, päivittäinen savutestausprosessi (Memon, Nagaran & Xie, 2005). Alkumuodostelmassa testattava käyttöliittymä analysoidaan ja siihen luodaan automaattisesti testioraakkeli sekä testitapaukset. Ohjelmiston kehityksen jälkeen iteraatiovaiheessa suoritetaan testit sekä lähetetään kehittäjille virhe- ja kattavuusraportti. Kehittäjä korjaa löytyneet viat tai tekee muita muutoksia ohjelmaan. Tämän jälkeen prosessi jatkuu uudella testikierroksella.



KUVIO 3 DARTin prosessi (Memon ym., 2005, s. 30)

Taulukossa 1 (Memon ym., 2003, s. 2) on esitetty tarkemmalla tasolla DARTin vaiheet ja niihin liittyvät roolien mukaiset tehtävät. Ensimmäisessä askeleessa kehittäjä tunnistaa testauksen kohteena olevan sovelluksen (AUT, application under test). Tämä sisältää lähdetiedostot sekä ajotiedostot. Toisessa askelessa DART analysoi automaattisesti käyttöliittymän rakenteen ja paloittelee sen

(GUI ripping) käymällä kaikki ikkunat ja niiden objektit ja ominaisuudet läpi ja tallentamalla puretun rakenteen XML-tiedostoon. Kolmannessa askeleessa kehittäjä varmistaa rakenteen ja tekee siihen tarvittavat lisäykset ja muutokset. Muutokset tallennetaan siten, että ne osataan ottaa automaattisesti huomioon AUT:n seuraavissa versioissa. Neljännessä askeleessa DART käyttää GUIn rakennetta generoidakseen automaattisesti tapahtumavuokaavioita ja integraatiopuun. Näitä käytetään myöhemmissä vaiheissa automaattiseen testitapausten luontiin sekä testien kattavuuden määrittämiseen. Viidennessä askeleessa kehittäjälle esitetään matriisi M(*i,j*), missä *i* on käyttöliittymäkomponentti ja *j* on testitapauksen pituus. M(*i,j*)=N tarkoittaa, että N kappaletta *j*:n pituisia testitapauksia voidaan suorittaa komponentille *i*. Kuudennessa askeleessa kehittäjä luo uuden matriisin M'(*i,j*), johon määritetään, kuinka monta *j*:n pituista testitapausta komponentille *i* tulee suorittaa.

TAULUKKO 1 Kehittäjän ja testaajan tehtävät DARTissa

Vaihe	**Askel**	**Kehittäjä/testaaja**	**DART**
Tunnistaminen	1	Tunnista testauksen kohteena oleva sovellus (AUT)	
Analyysi	2		Paloittele AUT:n graafinen käyttöliittymä
	3	Varmista ja muokkaa rakennetta	
Testien soveltuvuuden määrittely	4		Luo tapahtumavuokaaviot ja integraatiopuu
	5		Luo matriisi M
	6	Määritä M'	
Testien generointi	7		Generoi testitapaukset
	8		Generoi odotetut tulokset
Muutokset	9	Muuta AUT:ta	
Regressiotestaus	10		Instrumentoi koodi
	11		Suorita testitapaukset ja vertaa odotettuihin tuloksiin
	12		Luo suoritusraportti
	13		Luo kattavuusraportti
	14		Lähetä raportit sähköpostilla
Analysointi ja uudelleenluonti	15	Tutki raportit ja korjaa viat	
	16	Muokkaa M':ää tarvittaessa	
	17		Generoi lisätestitapauksia
	18		Generoi lisää odotettuja tuloksia

Tämän jälkeen DART käyttää automaattista testitapausgeneraattoria tuottamaan testitapaukset askeleessa seitsemän. Kahdeksannessa askeleessa oraakkeligeneraattori luo automaattisesti odotetut tulokset testattavan ohjelmiston seuraavaan versioon. Yhdeksännessä askeleessa kehittäjä muokkaa testattavaa oh-

jelmistoa. Kymmenennessä askeleessa käyttöjärjestelmä tai esimerkiksi versionhallintajärjestelmä käynnistää DARTin, joka puolestaan käynnistää testattavan ohjelman. DART instrumentoi koodin automaattisesti, jotta testauksesta saadaan muun muassa kattavuustietoja. Askeleella 11 testitapaukset suoritetaan ja niistä saatuja tuloksia verrataan odotettuihin tuloksiin. Askeleella 12 luodaan suoritusraportti, jossa testitapaukset on määritetty joko onnistuneiksi tai epäonnistuneiksi. Kolmannellatoista askeleella luodaan kaksi erilaista kattavuusraporttia: (1) lausekattavuusraportti, josta käy ilmi, kuinka monta kertaa jokainen lause on suoritettu ja (2) tapahtumakattavuusraportti, joka esitetään matriisina C(*i,j*). C(*i,j*) = N' kertoo, että N' kappaletta j:n pituisia testitapauksia on suoritettu komponentille i. C on muodoltaan vastaava matriisin M' kanssa, joten niiden vertailu on helppoa. Askeleella 14 DART lähettää testitulokset kehittäjille.

Viidennellätoista askeleella kehittäjät tutkivat raportit ja korjaavat löytyneet viat sekä tutkivat epäonnistuneet testitapaukset. Mikäli epäonnistuminen johtui virheellisestä odotetusta tuloksesta, testioraakkeli korjaa testin automaattisesti. Mikäli epäonnistuminen johtui testattavaan ohjelmistoon tehdystä muokkauksesta, testitapaus ei ole enää validi ja se poistetaan. Kuudennellatoista askeleella kehittäjät tunnistavat uusia testattavia kohtia käyttöliittymästä käyttäen apunaan kattavuusraportteja. He muokkaavat M':ää vastaavasti. Askeleilla 17 ja 18 generoidaan uudet testitapaukset ja oraakkelin tiedot.

3.3 GUI Testing Framework

GUI Testing frAmewoRk (GUITAR) on viitekehys, jonka tarkoituksena on luoda automaattisesti sopivia testitapauksia (Hackner & Memon, 2008). GUITARin arvokkain komponentti on graafisen käyttöliittymän palasiin hajoittava GUI Ripper. Se tutkii käyttöliittymää hierarkkisesti luoden integraatiopuun käyttöliittymän elementeistä sekä niiden välisistä suhteista. Ripper aloittaa pääikkunasta ja käy kaikki lapsi-ikkunat läpi sekä tutkii niiden GUI-elementit, kuten painikkeet ja tekstikentät sekä niiden ominaisuudet, kuten fontin ja värin ja vielä lisäksi niiden arvot, kuten "Arial" ja "sininen". Integraatiopuun luomisen jälkeen objekteja voidaan muokata ja testata. Tämä komponentti vastaa Daily Automated Regression Testerin (DART) askelta kaksi.

GUITARin toinen komponentti on EFG (event-flow graph) Generator, eli tapahtumavuokaavioiden generaattori. Tapahtumavuokaavioiden tehtävänä on esittää kaikki mahdolliset käyttöliittymän objektien väliset tapahtumat annettuna aikana. Hacknerin ja Memonin (2008) mukaan manuaalisena tehtävänä käyttöliittymäelementtien välisten suhteiden määrittäminen voi olla testauksen raskain tehtävä ajallisesti. Tapahtumavuokaavioiden luonnin jälkeen niitä voidaan käyttää testitapausten generoinnin pohjana, jolloin testitapauksia voidaan luoda hyvin pienellä vaivalla ja kustannuksilla. Tämä komponentti tuottaa DARTin askeleen neljä tehtävät.

Kolmas komponentti GUITARissa on testitapausten generaattori (Test Case Generator), jonka tehtävänä on luoda testitapauksia käyttäen pohjana tapahtumavuokaavioita ja integraatiopuuta. Testitapaukset luodaan käyttämällä joko solmunvalintaa (node selection) tai kaarenvalintaa (edge selection). Solmunvalinnassa testitapausten generaattori valitsee sattumanvaraisen tapahtuman tapahtumavuokaaviosta ja sen jälkeen selvittää sen saavuttamiseen vaadittavat komponenttien manipulaatiot. Kaarenvalinnassa suoritetaan ensin solmunvalinta yhdelle kaaren solmuista ja sen jälkeen lisätään saatuun polkuun kaaren toinen solmu. Näin saadaan tulokseksi informaatiota, jossa on komponentin nimi sekä tieto siitä, miten komponenttia tulee manipuloida, jotta ohjelmassa päästään valittuun tapahtumaan.

GUITAR hallitsee myös kattavuusraporttien laatimisen, joten sillä voidaan seurata myös suoritettujen koodirivien määrää. GUITARin tuottamat testitapaukset on myös mahdollista muuntaa automaattisesti jfcUnit-testeiksi.

3.4 Automaattisten testien elinkaarimalli

Elfriede Dustin, Jeff Rashka ja John Paul (2008) esittelivät kirjassaan ATLM:n (Automated Test Life Cycle Methodology), joka on metodologia, jonka tarkoitus on tukea automaattisen testauksen implementointia. Metodologia on monivaiheinen prosessi ja siinä on kuusi komponenttia. Se tukee toimintoja, joiden avulla voidaan päättää, tarvitaanko automaattista testausta ylipäätään, eli onko sen käyttöönotolle perusteita. Prosessi käsittelee testauksen *kokonaissuunnittelun* (*planning*), *analysoinnin* (*analysis*), *suunnittelun* (*design*), *kehityksen* (*development*), *suorituksen* (*execution*) ja *hallinnan* (*management*).

3.5 Web-testauksen työkaluja

Automaattiseen web-testaukseen on olemassa useita eri työkaluja. Taulukkoon 2 on koottu niistä muutama. Tässä tutkimuksessa päädyttiin kuitenkin käsittelemään ainoastaan kahta ehkä tunnetuinta ja käytetyintä testaustyökalua Selenium IDEä ja Robot Frameworkia sekä lyhyesti jUnitia ja jfcUnitia, jotka ovat käytössä erityisesti Java-kehityksessä. Muita web-testaustyökaluja ovat mm. Canoo WebTest, HttpUnit ja Hewlett Packardin omistama QuickTest Professional.

JUnit on viitekehys, jonka avulla Javalla kirjoitetun koodin yksikkötestit voidaan automatisoida. jfcUnit on JUnitin laajennos, jolla testauksen piiriin saadaan mukaan myös Javan omia GUI-ominaisuuksia, jotka sisältyvät Javan Swing-kirjastoon. JfcUnitin ongelmana on se, että jotta GUI-objektia voidaan testata jfcUnitilla, sen tulee olla näkyvissä ruudulla. (Hackner & Memon, 2008). Tämä rajoittaa työkalun käyttöä erityisesti silloin, jos pitää testata piilotettuja toiminnallisuuksia.

Testityökalut, joilla testejä voidaan tallentaa tallenna-ja-toista-toimintoa käyttämällä, ovat melko edullisia käyttää. Näiden ohjelmien haittapuolena on kuitenkin testien lyhytikäisyys. Todennäköisesti testejä ei voida enää käyttää, kun käyttöliittymää muutetaan (Marick, 1998).

TAULUKKO 2 Web-testaustyökalujen ominaisuuksia

Työkalu	**Automatisointi**	**JavaScript-tuki**	**testien kirjoittaminen ennen koodia**
Canoo WebTest	kyllä	huono	
HttpUnit	kyllä	huono	
jUnit/jfcUnit	kyllä	lisäosalla	kyllä
QuickTest Professional	kyllä	kohtalainen	
Robot Framework	kyllä	vaatii Seleniumin	kyllä
Selenium	kyllä	hyvä	kyllä

3.6 Selenium IDE

Selenium IDE on avoimen lähdekoodin työkalu selainpohjaiseen testaamiseen. Selenium-projektin käynnisti alun perin ThoughtWorks mutta nykyisin sillä on laaja ja aktiivinen kehittäjä- ja käyttäjäkunta. Toisin kuin monet muut automaattista web-testausta tarjoavat työkalut Selenium IDE tukee JavaScriptiä. Lisäksi se tarjoaa automatisoidun testipedin web-kerrokselle. (Holmes & Kellogg, 2006).

Seleniumia käytetään kirjoittamalla testitapaukset yksinkertaisiksi skripteiksi, joita työkalu sitten ajaa suoraan internet-selaimessa. Selenium IDE käyttää JavaScriptiä ja iframeja, jotta testimoottori saadaan upotettua selaimeen. Tämä mahdollistaa samojen testiskriptien käyttämisen eri selaimilla ja eri ympäristöissä. Lisäksi avoimen lähdekoodin ratkaisu mahdollistaa omien laajennosten kirjoittamisen sekä muiden kirjoittamien laajennosten käyttöönoton.

Testien kirjoittamiseen käytetään HTML-taulukoita ja ne ajetaan suoraan web-selaimessa. Käyttäjä näkee reaaliaikaisesti testien tilanteen ja värikoodattuna testin läpäisyn tai epäonnistumisen. Testeissä voidaan käyttää perustoimintoja, kuten avaa (osoite), klikkaa (käyttöliittymäobjektia) tai syötä arvo (tekstikenttään). Selenium IDEssä on lisäksi muun muassa tallenna-ja-toista-toiminto, jossa voidaan rakentaa testit tallentamalla manuaalinen testaus ja toistaa ne sitten automaattisesti.

3.6.1 Selenium IDEn käyttöönotto

Seleniumin käyttöönotto on helppoa, sillä asennukseen tarvitaan vain Mozilla Firefox –selaimeen asennettava laajennos, jonka voi ladata Selenium-projektin internet-sivuilta. Asennuksen jälkeen Selenium on valmiina käytettäväksi ja

käyttäjän tarvitsee ainoastaan alkaa kirjoittaa ja suorittaa testejä (Holmes & Kellogg, 2006).

3.6.2 Testien kirjoittaminen Selenium IDEllä

Seleniumilla voidaan viitata käyttöliittymäobjekteihin selaimen DOM-puun (Document Object Model) avulla, joten testit voidaan kirjoittaa käyttämällä esimerkiksi elementtien nimiä, id-arvoja tai xpathia. Holmesin ja Kelloggin (2006) mukaan testeissä kannattaa suosia id-arvoja, sillä ne ovat suorituskykyisempiä kuin esim. xpath. Sen vuoksi itse ohjelmakoodiin pitäisi pyrkiä määrittämään objekteille yksilölliset id:t aina kun mahdollista.

Selenium mahdollistaa testitapausten yhdistämisen testijoukoksi (test suite), jonka testitapaukset voidaan ajaa perätysten. Näin voidaan esimerkiksi tehdä yhden ominaisuuden testaamiseen liittyvä testijoukko, joka voidaan suorittaa aina tarvittaessa.

Holmes ja Kellogg (2006) kuitenkin huomauttavat, että testien yhdistäminen voi olla haitaksi, sillä se lisää redundanssia ja vähentää testaamisen joustavuutta ja ylläpitoa. He suosittelevatkin testien pitämistä niin itsenäisinä kuin mahdollista ja niiden refaktoroimista mahdollisimman paljon.

3.6.3 Seleniumin ongelmia

Holmes ja Kellogg (2006) mainitsevat Seleniumin käytössä kolme ongelmaa: (1) testien organisointi on ikävää, (2) testien kirjoittaminen HTML-muodossa on luonnotonta ja (3) muuttujien käyttäminen yleiskäyttöisissä testeissä on haasteellista. Ilman muuttujien käyttöä testit puolestaan eivät olisi helposti siirrettävissä, sillä esimerkiksi elementtien nimet ja id:t voivat muuttua. Näin ollen jokaiselle testikohteelle pitäisi kirjoittaa erilliset testitapaukset, mikäli niillä on eri nimet kuin toisilla kohteilla. Selenium IDEssä ei kuitenkaan tarvitse käyttää HTML-muotoa testien kirjoittamiseen, sillä työkalussa testitapaukset voidaan syöttää suoraan graafiseen käyttöliittymään.

Lisäksi testit voivat näyttää virheellistä epäonnistumista, mikäli vaikkapa napin nimi on muuttunut. Käyttöliittymä toimii käyttäjälle mahdollisesti täysin samalla tavalla kuin ennenkin mutta testi ei enää mene läpi. Vaikka Seleniumin testit onkin mahdollista kirjoittaa ennen varsinaisen toteutuksen kirjoittamista, Holmes ja Kellogg (2006) havaitsivat, että se voi aiheuttaa ongelmia testin läpäisyn suhteen, sillä toteutuksessa käytettävät nimet eivät välttämättä ole vielä tiedossa tai esimerkiksi käytetyn komennon pitääkin olla *clickAndWait* eikä vain *click*.

3.7 Robot Framework

Robot Framework on python-pohjainen avoimen lähdekoodin testauskehys, joka on tarkoitettu erityisesti hyväksymistestausten automatisointiin sekä hyväksymistestivetoiseen kehitykseen. Kehys on laitteisto- ja teknologiariippumaton, joten se soveltuu moniin eri ympäristöihin. Kehys käyttää avainsanatyyppistä lähestymistapaa. Kehyksen tekijänoikeudet omistaa Nokia Siemens Networks. Robot Frameworkissa on myös tuki Seleniumin käyttöön. (Nokia Siemens Networks, 2012).

3.7.1 Robot Frameworkin käyttöönotto

Robot Frameworkin asennus vaatii taustalle Pythonin, ja mikäli käytössä on Javalla kirjoitettuja testikirjastoja, vaaditaan Pythonin lisäksi Jython. Robot Framework voidaan asentaa esivaatimusten täytyttyä sekä Windowsille että UNIX-pohjaisiin järjestelmiin. Jani Koskelan opinnäytetyössä (2012) Robot Frameworkin asentaminen ja käyttöönotto havaittiin melko suoraviivaiseksi ja helpoksi toimenpiteeksi.

3.7.2 Testien kirjoittaminen Robot Frameworkilla

Robot Frameworkin ydin ei itsessään tiedä testattavasta sovelluksesta mitään, vaan kaikki testit tapahtuvat testikirjastojen kautta. Robot Framework käyttää testien kirjoittamiseen taulukkopohjaista syntaksia ja tuottaa helposti luettavia tuloksia HTML-muodossa. Koska Robot Framework käyttää avainsanapohjaista lähestymistapaa, jossa testitaulukoista poimitaan avainsanojen perusteella toimintaa, voidaan testitapausten luonnissa käyttää luonnollista englannin kieltä. Esimerkki testitapaustaulukosta on esitetty kuviossa 4. (Nokia Siemens Networks, 2009).

Testitapaus	Toiminto	Argumentti	Argumentti
Käyttäjä voi luoda tilin ja kirjautua sisään	Create Valid User	antti	S4la5Ana
	Attempt to Login with Credentials	antti	S4la5Ana
	Status Should Be	Kirjattu sisään	
Käyttäjä ei voi kirjautua väärällä salasanalla	Create Valid User	anna	S4la5Ana
	Attempt to Login with Credentials	anna	salasana
	Status Should Be	Pääsy evätty	

KUVIO 4 Robot Frameworkin testitapaustaulukko (Nokia Siemens Networks, 2009).

Avainsanoja on kolmenlaisia: sisäänrakennetut avainsanat ovat aina käytettävissä, erikseen lisätyistä kirjastoista voidaan saada kirjastokohtaisia avainsanoja ja lisäksi voidaan luoda kokonaan omia avainsanoja.

Testitapaukset kirjoitetaan testitapaustiedostoihin, joista luodaan automaattisesti testijoukko, joka sisältää kyseisen testijoukon testitapaustiedostot. Testijoukoista ja testitapauksista voidaan muodostaa hierarkkinen rakenne, jossa on testijoukkoja ja niiden alitestijoukkoja. Lisäksi kehykseen kuuluu testikirjastoja, jotka sisältävät alimman tason avainsanat, resurssitiedostot, jotka sisältävät ylemmän tason avainsanat, sekä muuttujatiedostot, joiden avulla voidaan luoda muuttujia. (Nokia Siemens Networks, 2011).

Toisin kuin Seleniumissa (Holmes & Kellogg, 2006) Robot Frameworkissa muuttujien käyttäminen testauksessa on erittäin toimivaa. Muuttujia kannattaakin käyttää, mikäli testataan erilaisia syötteitä. Niinpä argumenttina voidaankin käyttää kovakoodatun S4la5Ana-tekstin sijaan muuttujaa ${salasana}. Erikseen määritetyssä tiedostossa voidaan sitten määrittää, mitä muuttuja sisältää missäkin tapauksessa.

3.8 Yhteenveto

Tässä luvussa esiteltiin kirjallisuudessa esitettyjä malleja ja tekniikoita testauksen automatisointiin. Testauksen kypsyysmalli on ylätason malli, jonka alle muita malleja voidaan niputtaa. Memonin ym. (2003b). DART on pitkälle viety viitekehys graafisten käyttöliittymien automaattiseen testaamiseen. Se sisältää kaiken aina automaattisesta graafisen käyttöliittymän paloittelemisesta, testitapausten luonnista testioraakkelin luontiin ja hajonneiden testitapausten korjaamiseen ja uudelleensuorittamiseen.

Hacknerin ja Memonin (2008) GUITAR-viitekehys puolestaan sijoittuu DARTin alkuvaiheisiin, jossa graafinen käyttöliittymä paloitellaan automaattisesti komponenteiksi, luodaan niistä tapahtumavuokaavio sekä integraatiopuu ja lopuksi luodaan testitapaukset. Nämä toiminnat ovat hyvin laajoja ja manuaalisesti suoritettuina veisivät suuren määrän aikaa. Näiden lisäksi GUITAR voi vielä tuottaa kattavuusraportteja ja muuttaa testitapaukset jfcUnit-yhteensopivaan muotoon automaattisesti suoritettaviksi.

Testien suorittamisen työkaluista tässä luvussa esiteltiin Selenium ja Robot Framework. Molemmat ovat tunnettuja ja käytettyjä automaattisia web-testaustyökaluja. Tämän tutkielman kokeellisen tutkimuksen työvälineeksi valittiin Selenium IDE sen helpon käyttöönoton ja käytettävyyden vuoksi. Lisäksi sen syntaksi sopii myös ohjelmointitaidottomalle testaajalle, joten sen avulla on helppo ja nopea päästä alkuun vain selaimen lisäosan asentamalla.

4 PROJEKTISSAMME KÄYTTÖÖNOTETTAVAN MALLIN JA TEKNIIKOIDEN YHDISTELMÄN RAKENTAMINEN

Tässä luvussa esitetään tutkielman empiirisessä osuudessa käytettävä projekti sekä rakennetaan empiirisessä tutkimuksessa käytettävä viitekehys. Empiirinen osuus suoritetaan kokeellisena tutkimuksena, jossa pyritään selvittämään, miten rakennettu malli voidaan ottaa käyttöön projektissamme sekä mahdollisesti myöhemmin koko organisaatiossamme. Rakennettava malli pohjautuu kirjallisuudessa esitettyihin malleihin ja tekniikoihin, joiden avulla graafisten käyttöliittymien automaattista regressiotestaamista voidaan tehdä.

4.1 Tutkimuksen motiivi

Tutkimuksen motiivi perustuu työhöni, jossa kehitän ja testaan mm. yritysten sähköisiä työpöytiä ja intranetejä. Yhdistävä tekijä näissä ovat erityisesti graafiset käyttöliittymät ja regressiotestaus. Työssäni olen huomannut, että regressiotestaukseen kuluu suunnattomasti aikaa ja lisäksi samojen asioiden testaaminen uudelleen ja uudelleen turruttaa nopeasti. Näin ollen regressiotestauksen teho heikkenee ja se puolestaan altistaa järjestelmät virheille, joita ei havaita tarpeeksi ajoissa. Lisäksi projektissamme on käytössä päivittäiset versiot, ja kuten Kajko-Mattsson ym. (2004) mainitsivat, testauksen automatisointi on edellytys tällaisessa tilanteessa, jotta mahdolliset virheet saadaan havaittua nopeasti ja säännöllisesti.

Lisäksi testattavat ympäristöt ovat hyvin laajoja kokonaisuuksia, ja olenkin projektissamme havainnut, että pieni ja harmittomalta vaikuttava muutos voi hajottaa toiminnan järjestelmän toisessa osassa. Tämän vuoksi järjestelmää tulisi testata jokaisen muutoksen jälkeen huolellisesti. Tämän työn helpottamiseksi vaaditaan tehokasta automatisointia, sillä tämän mittaluokan järjestelmää ei voida testata manuaalisesti jokaisen muutoksen jälkeen. Sen vuoksi onkin tärkeää automatisoida prosessista niin paljon kuin mahdollista. Tässä tutkiel-

massa automatisointia ei voida ottaa käyttöön koko testausketjun laajuudelta, kuten esimerkiksi DARTissa, mutta itse testien suorittaminen voidaan automatisoida sopivien työkalujen avulla.

Tässä tutkielmassa ei huomioida järjestelmäkehityksen ohjelmointiosuuden testejä, kuten yksikkötestejä tai integraatiotestejä, vaan keskitytään käsittelemään järjestelmän toiminnallisia vaatimuksia ja testausta loppukäyttäjän ja hyväksyntätestauksen näkökulmasta. *Hyväksyntätestauksessa* (*acceptance testing*) ei ole oleellista, mitä sovelluksessa tapahtuu pellin alla, vaan se, että käyttäjä voi tehdä sovelluksen käyttöliittymän kautta haluttuja asioita (Bruns, Kornstädt & Wichmann, 2009). Tämä on projektissamme myös aikaa vievin vaihe ja sen vuoksi juuri se on tärkeää automatisoida.

4.2 Tutkimusongelma ja osaongelmat

Tutkielmani kokeellisen tutkimuksen osuuden tutkimusongelma on: *Kuinka yrityksessäni (ja erityisesti nykyisessä projektissani) voidaan ottaa käyttöön ja käyttää automaattista regressiotestausta, kun kyseessä on graafisella käyttöliittymällä varustettu ympäristö?* Osaongelmana voidaan pitää seuraavaa: *Mitkä kirjallisuudessa esitetyistä malleista ja teknologioista ovat soveltuvia ja miten niitä voidaan yhdistellä palvelemaan yritystäni?* Tutkielman johtopäätöksessä tarkastellaan, onko esittämäni malli järkevä tai muuten mahdollinen organisaatiolleni tai projektilleni sekä oliko graafisten käyttöliittymien automaattisen testauksen implementoinnista mitään hyötyä ja mitä tutkimuksen suorittamisesta opittiin.

4.3 Organisaatiosta ja projektista

Organisaatiomme on suomalainen sähköisen liiketoiminnan ratkaisuja tarjoava yritys, jossa työskentelee noin 220 henkilöä ja jonka liikevaihto on noin 40 miljoonaa euroa. Organisaatiollamme on toimipaikkoja Suomen lisäksi myös Ruotsissa ja Puolassa.

Tämän tutkimuksen empiirisen osuuden tutkimuksen kohteeksi on valittu projekti, jossa päivitetään suomalaisen, maan mittakaavassa merkittävän kokoisen organisaation intranet-järjestelmä uudelle alustalle sekä päivitetään yleisilmettä ja lisätään uusia toiminnallisuuksia. Voitaisiin puhua jopa koko järjestelmän versiopäivityksestä. Päivitettävällä järjestelmällä on useita tuhansia käyttäjiä. Tämän projektin intranet-järjestelmä on kehittynyt pääasiallisesta viestinnällisestä järjestelmästä sähköistä työpöytää muistuttavaksi järjestelmäksi, jossa on muun muassa erilaisia hallinnallisia ja yhteisöllisiä toiminnallisuuksia.

Järjestelmä on rakennettu IBM WebSphere –portaaliratkaisun päälle. Järjestelmä käyttää pääasiallisesti IBM Web Content Management –sisällönhallintajärjestelmää mutta sovelluksia toteutetaan pääosin Javalla sekä JavaScriptin jQuery-laajennoksen avulla. Tässä tutkielmassa tarkastellaan sovel-

lusta, jonka käyttöliittymä on toteutettu HTML:llä ja jQuerylla. Tarkastelun kohteeksi valittu sovellus on kokonaisen järjestelmän yksi toiminnallisuus, pikalinkkien hallintasovellus, jonka perustoiminnallisuus ei juurikaan muutu mutta ulkoasu päivittyy projektin aikana.

4.4 Tutkimuksen lähtötilanne

Projektissamme käytetään ketterää kehitystä ja sovellamme siinä Scrumia. Lähtötilanteessa projektissamme toimitaan siten, että ensin toteutetaan ominaisuuksia, jotka sen jälkeen testataan, jotta voidaan varmistaa, että ne toimivat halutulla tavalla. Ominaisuuksia voidaan kuvata Scrumin mukaisesti käyttäjätarinoiksi (user story). Pääasiallisesti testauksen suorittaa kehitystiimin jäsen, joka ei kuitenkaan ole ollut kyseisen ominaisuuden ohjelmoinnissa mukana. Tällä testaajalla ei myöskään ole ohjelmointitaustaa vaan hän tarkastelee sovellusta enemmänkin loppukäyttäjän näkökulmasta. Testatessaan testaaja tarkentaa testitapauksia kehittäjän toimittamaan testitapauslistaukseen ja tarvittaessa laatii uusia testitapauksia joidenkin toiminnallisuuksien testaamiseksi. Tästä voi aiheutua se, että testaaja löytää sellaisia ongelmia, jotka kehittäjän mielestä ovat ominaisuuksia (*feature*) tai eivät kuulu kyseisen tehtävän testauksen fokukseen. Tässä vaiheessa löydetyt havainnot, jotka eivät ole testattavan ohjelmiston näkökulmasta virheitä mutta vaativat kuitenkin tarkennuksia, selvitetään asiakkaan kanssa, ja tarvittaessa muutetaan sovellusta ja testitapauksia vastaamaan asiakkaan tarkentamia vaatimuksia tai otetaan myöhemmin jatkokehitykseen. Koska tässä projektissa kyseessä oli jo olemassa oleva sovellus, joka siirrettiin uuteen ympäristöön ja jota muutettiin vain vähän, vaatimukset olivat jo hyvin selvät eikä vaatimustenhallinnasta aiheutunut ongelmia.

Holmes ja Kellogg (2006) ovat saaneet omissa tutkimuksissaan hyviä tuloksia siten, että tarinan hyväksymistestitapaukset kirjoitetaan ennen kuin ominaisuutta aletaan toteuttaa. Tällä tavalla voidaan fokusoida paremmin tarinan sisältö sekä varmistaa, että se läpäisee hyväksytysti sille asetetut vaatimukset. Samalla vältytään mahdollisilta keskusteluilta siitä, onko testaajan löytämä poikkeama vika vai ominaisuus (Whittaker, 2000).

Myös Lisa Crispin (2006a) tukee testien kirjoittamista ennen toiminnallisuuden ohjelmointia, eli testivetoista kehitystä (test driven development, TDD). Crispinin mukaan tämä menetelmä vähentää selvästi virheiden määrää koodissa, helpottaa niiden korjausta sekä tuottaa laadukkaampaa koodia. Lisäksi nämä testit voidaan lisätä yhteiseen testikantaan, josta ne voidaan suorittaa aina, kun ohjelmistoon tehdään muutos.

TDD on suunnattu yksikkötestaukseen, mutta hyväksymistestaukselleki on oma versionsa: behavioral-driven development (BDD). BDD on TDD:n laajennos, jossa käyttäjätarinat kirjoitetaan tiettyyn tietokoneen ymmärtämään muotoon kuitenkin niin, että myös ohjelmistokehityksen ulkopuoliset ihmiset voivat lukea ja ymmärtää niitä normaalisti. BDD:n avulla on mahdollista ymmärtää järjestelmän vaatimukset paremmin sekä saada helpotettua myös do-

kumentointia (Agile Alliance, 2013). BDD:n avulla hyväksymistestitapausten laatiminen on helpompaa, sillä vaatimukset on kirjattu valmiiksi sopivassa muodossa. Tässä projektissa ei käytetty kumpaakaan menetelmää, vaan kaikki testit kirjoitettiin vasta sen jälkeen, kun toteutus oli valmis.

Seleniumin käytöstä regressiotestauksen apuvälineenä saadaan tukea Holmesin ja Kelloggin (2006) tutkimuksesta. Siinä kehittäjille jää mahdollisuus muokata ja refaktoroida koodia, sillä Seleniumilla suoritetut automaattiset testit pitävät huolen siitä, että aiemmin läpäistyjen testien epäonnistuttua ne havaitaan välittömästi. Lisäksi heidän tutkimuksensa osoitti, että kun Seleniumin testit näyttivät vihreää sprintin lopuksi, se sai kehittäjät luottamaan siihen, että nopeasti muuttuva sovellus on vakaa.

Regressiotestausta projektissamme suoritetaan satunnaisesti. Pääasiallisesti regressiotestaus suoritetaan savutestauksena, mikäli järjestelmään tehdään laajempi muutos taustapalveluihin ja aina ennen tuotantojulkaisua. Muutoin regressiotestaus on lähinnä muiden ominaisuuksien testauksen yhteydessä tehtävää testausta aiemmin jo toimineissa kokonaisuuksissa. Järjestelmällistä regressiotestausta ei suoriteta, sillä se on liian kallista ja aikaa vievää. Mikäli kokonaisuus on tarpeeksi erillinen muusta järjestelmästä, sitä ei välttämättä regressiotestata lainkaan edes julkaisuiden yhteydessä, sillä kyseistä toimintoa ei käytetä muiden ominaisuuksien testaamiseen. Kokemus on kuitenkin osoittanut, että laatu kärsii, mikäli näissä tapauksissa regressiotestausta ei tehdä. Näin ollen projektissamme tarvitaan tehokkaampaa regressiotestausta ja sen vuoksi regressiotestauksen automatisointia on päätetty alkaa tutkia ja ottaa käyttöön testauksen laadun parantamiseksi.

4.5 Kokeellisessa tutkimuksessa käytettävä malli

Tässä vaiheessa rakennan projektiimme graafisten käyttöliittymien regressiotestaukseen automaattiseksi ainoastaan testien suorittamisen. DARTin tai GUITARin mukaiset muut automatisoinnin vaiheet jätän myöhemmäksi, mikäli tästä tutkimuksesta saadut tulokset rohkaisevat automatisoinnin jatkokehittämiseen. Perusteluna tälle rajaukselle esitän sen, että koska projektissamme ja organisaatiossamme ollaan testauksessa ja ennen kaikkea sen automatisoinnissa vasta testauksen kypsyysmallin alimmalla tasolla, kannattaa automatisoinnin pohja rakentaa vankaksi ennen kuin automatisointia lisätään.

Rakennan tutkimuksen mallin kuitenkin ottaen huomioon kirjallisuudessa esitettyjä huomioita graafisten käyttöliittymien automaattisesta regressiotestauksesta, jotta automatisointia on myöhemmin helpompi lähteä kehittämään viitekehysten ollessa tuttuja. Tässä tutkimuksessa muun muassa käyttöliittymän paloittelu, testitapausten valinta ja kirjoittaminen hoidetaan manuaalisesti. Testien suorittaminen kuitenkin automatisoidaan. Jo pelkästään tästä oletetaan saatavan hyötyä sekä työmäärällisesti, kalenteriajallisesti että laadullisesti. Automatisointia verrataan manuaaliseen testaukseen suorittamalla samat testita-

paukset sekä automatisoidusti että manuaalisesti ja selvittämällä niihin kuluvat ajat.

Automatisoinnin välineeksi päätin valita aiemmin esitetyistä työkaluista Selenium IDEn. Seleniumin valintaan vaikuttivat kirjallisuudesta löydetyt hyvät kokemukset sekä se, että organisaatiossamme on jo valmiiksi hieman kokemusta Seleniumista automaattisten suorituskykytestien alueelta. Lisäksi Robot Frameworkin käyttäminen projektissamme vaatisi toimiakseen Selenium-liitännäisen joka tapauksessa. Lisäksi Seleniumin käyttöönotto on äärimmäisen helppoa ja sen syntaksi on helppoa myös testaajille, joilla ei ole ohjelmointikokemusta.

Käytän tutkimuksessani mallia, jossa on testausprosessin kaikki kuusi vaihetta. Vaiheen 1 eli testattavan ympäristön mallintamisen rajoitan koskemaan ainoastaan sovelluksen graafista käyttöliittymää ja sen suoria toiminnallisia ominaisuuksia. Vaiheeseen 4 eli testiskenaarioiden suorittamiseen toteutan automatisoinnin Selenium IDEllä. Muut vaiheet, kuten testiskenaarioiden valinnan suoritan manuaalisesti vielä tässä tutkielmassa. Koska kyseessä on vain pieni osa järjestelmästä, ei tutkielman alkupuolella esitettyjä regressiotestien testitapauksien valintamenetelmiä tarvita. Kaikki tämän tutkielman yhteydessä rakennettavat testitapaukset voidaan ajaa automaattisesti ilman huolta resurssien loppumisesta. Vaiheen 6 eli testausprosessin mittaamisen suoritan vertaillen tuloksia nykyiseen manuaaliseen testaamiseen. Pääasiallisena mittarina käytetään aikaa, joka testausprosessiin kuluu manuaalisesti. Mikäli tulokset mahdollistavat, mittarina voidaan myös käyttää löytyneiden virheiden määrää. Koska kyseessä on regressiotestaus, virheiden määrän pitäisi olla hyvin pieni riippumatta siitä, suoritetaanko testaus automaattisesti vai manuaalisesti.

Testausprosessin aikana arvioidaan myös jatkuvasti testitapausten käytettävyyttä ja sitä, voisiko niitä parantaa. Testitapausten valinta suoritetaan käyttöliittymän luonteen vuoksi tarkastelemalla erilaisia polkuja. Polkujen kriittisyyttä arvioidaan niiden käyttömäärän perusteella. Käyttömäärätieto tulee projektin sisältä kokemuksen kautta sekä käyttötapauksista, eikä sitä tarkastella tässä tutkimuksessa sen enempää. Sen sijaan esimerkiksi lausekattavuuteen liittyvät valintaperusteet jätetään tässä tutkimuksessa pois. Mahdolliset käyttäjävirheet tai virheelliset syötteet rajataan pois tässä tutkielmassa tarkasteltavien testitapausten määrän pitämiseksi kohtuullisena. Lisäksi tarkastelun lähtökohtana on varmistaa, että kriittinen toiminnallisuus pysyy ehjänä. Painotan tärkeämmäksi sen, että sovellus toimii oikein, jos käyttäjä toimii oikein, enkä sitä, että sovelluksen toiminta huomioisi käyttäjien virheellisen toiminnan. Jälkimmäinen on myös tärkeää, mutta tässä tutkielmassa se rajataan edellä mainituilla syillä pois.

Koska DART antaa hyvän viitekehyksen koko testausprosessin automatisoinnista, käytän sitä hyväkseni, vaikka suoritankin suurimman osan askeleista manuaalisesti. Käyttämässäni riisutussa mallissa, joka on esitetty taulukossa 3, sovelletaan DARTin askeleita 1, 2, 7, 8, 11, 12, 15 sekä tarvittaessa askeleita 17 ja 18. Muut askeleet on jätetty tässä vaiheessa pois, sillä painoarvo on saada varsinainen testien suorittaminen automatisoitua. Lisäksi organisaatiomme tes-

tauksen kypsyysmallin mukainen taso ei vielä mahdollista koko testausprosessin automatisointia tässä projektissa. Myös se, että osa askelista tehdään manuaalisesti, tekee osan askelista turhiksi. Taulukossa on esitetty askelten järjestysnumerot alkuperäistä DARTia vastaavasti.

TAULUKKO 3 Tutkielmassa käytettävä DARTin pohjalta laadittu riisuttu malli

Vaihe	**Askel**	**Testaaja**	**Selenium IDE**
Tunnistaminen	1	Tunnista testauksen kohteena oleva sovellus (AUT)	
Analyysi	2	Paloittele AUT:n graafinen käyttöliittymä	
Testien generointi	7	Luo testitapaukset	
	8	Luo odotetut tulokset	
Regressiotestaus	11		Suorita testitapaukset ja vertaa odotettuihin tuloksiin
	12		Luo suoritusraportti
Analysointi ja uudelleenluonti	15	Tutki raportit ja raportoi viat kehittäjille	
	17	Luo lisää testitapauksia	
	18	Luo lisää odotettuja tuloksia	

4.6 Testattava käyttöliittymä

DARTin askeleessa 1 tunnistetaan testauksen kohteena oleva sovellus. Tässä tutkielmassa se on projektimme järjestelmässä oleva toiminnallisuus, jossa käyttäjä voi luoda linkin haluamalleen sivulle, joka voi olla järjestelmän sisällä tai ulkopuolella.

Askeleessa 2 DART paloittelee graafisen käyttöliittymän osiin. Tässä tutkielmassa tätä vaihetta ei vielä automatisoida, vaan testauksen automatisoinnin näkökulma on testien suorittamisen automatisoinnissa. Tämä vaihe tehdään tässä tutkielmassa siis manuaalisesti. Tutkittava sovellus koostuu erilaisista osioista ja niihin liittyvistä käyttöliittymistä. Sovellus sisältää (1) alasvetovalikon, joka listaa käyttäjän listalle valitsemat pikalinkit, (2) linkkien valintanäkymän, jonka avulla voidaan selata omia ja yhteisiä linkkejä sekä siirrellä niitä yhteisten linkkien luettelon ja oman linkkiluettelon välillä. Lisäksi se sisältää (3) linkkien luonti- ja muokkausnäkymän, jolla uusien linkkien luonti sekä vanhojen muokkaus ja poisto on mahdollista.

Tutkitaan ensin perustapausta eli linkkien luontinäkymää. Vaihtoehtoisia syöttötapoja on kolme:

(1) Valitaan halutulta sivulta painike, joka luo linkin lähes automaattisesti kyseiselle sivulle ja lisää sen linkkien listaan. Käyttöliittymässä kysytään, millä nimellä linkki halutaan näytettävän listassa. Oletuksena an-

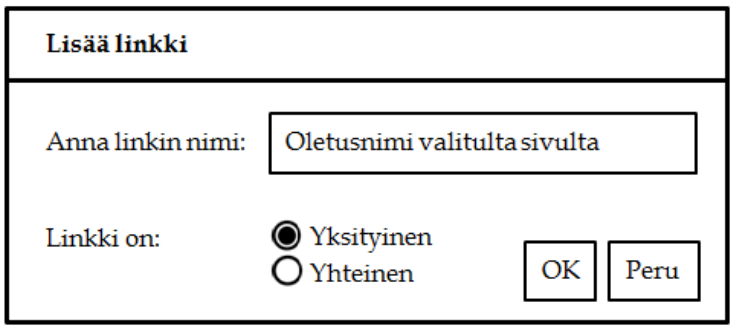
netaan sivun otsikko. Lisäksi erikoiskäyttäjillä on mahdollisuus tehdä linkistä yhteinen, jolloin se ilmestyy kaikkien haluttujen käyttäjien saataville erilliseen valmiiden linkkien listaan.

(2) Valitaan linkkilistauksen hallintasovelluksesta painike, jonka avulla voidaan luoda uusi linkki. Käyttöliittymässä kysytään, halutaanko linkistä luoda ulkoinen, sisäinen vai sovelluslinkki. Lisäksi erikoiskäyttäjillä on mahdollisuus tehdä linkistä yhteinen.

(a) Mikäli linkistä tehdään ulkoinen tai sovelluslinkki, syötetään kohdesivun tai sovelluksen URL-osoite sille varattuun kenttään.

(b) Mikäli linkistä tehdään sisäinen, avautuu käyttöliittymään puurakenne, jolla voidaan navigoida sivustohierarkiassa halutulle sivulle ja valita se linkin kohteeksi.

Edellä mainituista kolmesta syöttövaihtoehdosta ensimmäinen on pikatoiminto ja se on myös käyttöliittymältään yksinkertaisin. Pikatoiminnon käyttöliittymä on kuvattu kuviossa 5. Mikäli käyttäjä ei ole erikoiskäyttäjä, näkymässä ei ole muuta kuin linkin nimeä kysyvä kenttä, joka on jo valmiiksi täytetty sivun otsikolla, sekä OK- ja Peru-napit. Jo tästä näkymästä on kuitenkin eroteltavissa useita eri testitapauksia: (1) oletusnimeksi tulee automaattisesti sen sivun nimi, josta linkin luontinäkyvä avattiin, (2) oletusnimen voi vaihtaa (3) Linkki on –valinta näkyy dialogissa ainoastaan silloin, jos käyttäjä on erikoiskäyttäjä, (4) Peru-nappi sulkee dialogin tekemättä muutoksia ja (5) OK-painike tallentaa muutokset tietokantaan huomioiden käyttöliittymässä tehdyt valinnat varmistaen tietysti syötettyjen tietojen oikeellisuuden. Lisäksi meidän projektissamme on vaatimus, jonka mukaan (6) ikkunaa tulee voida siirtää raahaamalla. Kaikkia kohtia ei voida testata samalla kertaa, joten testit on suoritettava useammalla kierroksella, jotta kaikki eri yhdistelmät saadaan testattua. Lisäksi on mahdollista, että käyttöliittymäkomponenttien klikkaaminen eri järjestyksessä voi vaikuttaa niiden toimintaan, joten sekin pitää ottaa huomioon, ei välttämättä kuitenkaan enää regressiotestauksessa.



KUVIO 5 Linkin lisääminen pikavalinnalla (erikoiskäyttäjän näkymä)

Taulukosta 4 nähdään, että tämän käyttöliittymän eri yhdistelmiä (klikkausten järjestystä huomioimatta) on kahdeksan kappaletta. Taulukossa ei myöskään ole vielä tässä vaiheessa huomioitu, että valittaessa ”Yhteinen” käyttöliittymään avautuu lisävalintoja. Lisäksi yksinkertaistamisen vuoksi on oletettu, että Peru-

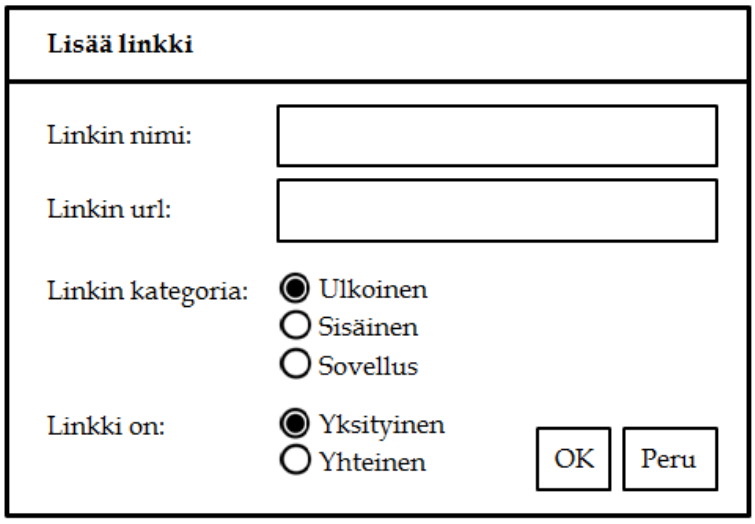
painike toimii aina oikein riippumatta siitä, mitä muita valintoja käyttöliittymässä on tehty.

TAULUKKO 4 Pikaluontinäkymän käyttöliittymän mahdolliset yhdistelmät

Käyttäjä	Linkin nimi	Linkki on	Toiminto
Perus			Peruutus
Perus	Oletus		OK
Perus	Muokattu		OK
Erikois			Peruutus
Erikois	Oletus	Yksityinen	OK
Erikois	Oletus	Yhteinen	OK
Erikois	Muokattu	Yksityinen	OK
Erikois	Muokattu	Yhteinen	OK

Näiden testitapausten lisäksi tulee varmistaa, että (1) linkki on luotu oikein ja että (2) se toimii. On myös varmistettava, että (3) yksityisen linkin tapauksessa sitä eivät näe muut kuin linkin tekijä. Yhteisen linkin tapauksessa on varmistettava, että (4) linkin voivat nähdä ja valita omalle pikalinkkilistalleen ne käyttäjät, jotka linkkiin on määritetty teknisiksi tai hallinnollisiksi vastuuhenkilöiksi tai käyttäjiksi.

Linkin lisääminen pikatoiminnon avulla on kuitenkin sovelluksen tekniseltä kannalta katsottuna erikoistapaus. Perustapauksena voidaan pitää linkin lisäämistä erillisen hallintanäkymän kautta, jossa linkille voidaan määrittää enemmän vaihtoehtoisia arvoja. Linkin luonti hallintanäkymän kautta on huomattavasti monimutkaisempaa kuin pikatoiminnon kautta. Näiden seikkojen vuoksi olen valinnut tämän tutkielman tarkastelun kohteeksi nimenomaan hallintanäkymän tapaukset. Hallintapaneelin lisäämisnäkymä on esitetty kuviossa 6.



KUVIO 6 Linkin lisääminen hallintapaneelin kautta

Linkin lisäämiskäyttöliittymä koostuu ponnahdusdialogista, jossa on radiopainike-valinnat linkin kategorialle (ulkoinen, sisäinen ja sovellus) sekä erikoiskäyttäjien tapauksessa sille, tehdäänkö linkistä ainoastaan itselle näkyvä vai yhteinen. Yhteisen linkin luonnin yhteydessä käyttöliittymään ilmestyy välilehtiä, joilla tehdään pikalinkin kannalta merkittäviä lisävalintoja. Välilehdiltä voidaan valita linkille sekä tekniset että hallinnolliset vastuuhenkilöt ja käyttäjät, jotka voivat valita linkin omalle pikavalintalistalleen. Dialogissa on edellä mainittujen kenttien lisäksi myös lyhyt tekstikenttä linkin nimeä varten. Riippuen kategoriavalinnasta dialogissa näkyy joko lyhyt tekstikenttä URL-osoitetta varten tai vaihtoehtoisesti painike, jolla voidaan avata sivustohierarkiapuu linkin kohteen valintaa varten.

Muutaman kentän lisääminen käyttöliittymään aiheuttaa mahdollisten yhdistelmien määrän kasvamisen. Taulukossa 5 on esitetty hallintapaneelin kautta luotavan linkin käyttöliittymän mahdolliset kombinaatiot. Tässä taulukossa ei ole kuitenkaan huomioitu yhteisen linkin luomisen aiheuttamia lisävalintoja erillisillä välilehdillä, vaan ainoastaan yhden näkymän tapaukset, eli erilaiset linkin perusmääritysten vaihtoehdot. Myöhemmin tehtävissä testitapauksissa on kuitenkin huomioitu myös lisävalinnat, jotta testit saadaan vietyä läpi kokonaisuudessaan automatisoidusti.

Kuten taulukosta näemme, erilaisten linkinluontimahdollisuuksien määrä ei aina kasva eksponentiaalisesti, koska joidenkin valintojen tekeminen voi poistaa toisia valintoja käytöstä. Pikalinkin luontinäkymän kautta linkkiä luotaessa vaihtoehtoja oli kahdeksan ja hallintapaneelin kautta 14. Vaihtoehtoja tuli lisää alle puolet, vaikka luontinäkymään oli lisätty tekstikenttä sekä kolmivaihtoehtoinen radiopainikeryhmä. Esimerkiksi linkin valinta ulkoiseksi poistaa linkin oletusnimen mahdollisuuden. Vaikka tässä tapauksessa niin ei olekaan, on kuitenkin mahdollista, että valintojen lisääminen käyttöliittymään kasvattaisi testitapausten määrää eksponentiaalisesti, ja tällöin testien automatisoinnin arvo kasvaa entisestään.

TAULUKKO 5 Hallintapaneelissa luotavan linkin käyttöliittymän mahdolliset yhdistelmät

Käyttäjä	**Linkin nimi**	**Kategoria**	**Linkki on**	**Toiminto**
Perus				Peruutus
Perus	Oletus	Sisäinen		OK
Perus	Muokattu	Sisäinen		OK
Perus	Muokattu	Ulkoinen		OK
Perus	Muokattu	Sovellus		OK
Erikois				Peruutus
Erikois	Oletus	Sisäinen	Yksityinen	OK
Erikois	Muokattu	Sisäinen	Yksityinen	OK
Erikois	Muokattu	Ulkoinen	Yksityinen	OK
Erikois	Muokattu	Sovellus	Yksityinen	OK
Erikois	Oletus	Sisäinen	Yhteinen	OK
Erikois	Muokattu	Sisäinen	Yhteinen	OK
Erikois	Muokattu	Ulkoinen	Yhteinen	OK
Erikois	Muokattu	Sovellus	Yhteinen	OK

Tässä tutkielmassa päädyttiin jättämään DARTin askel 4 eli tapahtumavuokaavioiden ja integraatiopuun luonti väliin, sillä projektissamme ei ole käytössä GUI Ripperiä ja manuaalisesti tapahtumavuokaavioiden laatiminen olisi liian työlästä. Lisäksi tässä tutkielmassa ei pyritä luomaan automaattisia testejä käyttöliittymän kaikkien mahdollisten elementtien kombinaatioiden testaamiseen, vaan enemmänkin erilaisten käyttötapauspolkujen läpikäyntiin. Tässä tutkielmassa ei siis ole tarkoituksena varmistaa automaattisin testein, että käyttäjän klikatessa Ulkoinen-valintapainiketta hän voi tämän jälkeen valita jonkin toisen kategorian tai vaikka Ulkoinen-valinnan uudelleen. Sen sijaan tässä pyritään rakentamaan regressiotestaus kriittisempään toiminnallisuuteen, jossa käyttäjän valitessa Ulkoinen-kategoria hän kykenee viemään linkin luonnin loppuun asti. Tutkielma jatkaa DARTin askeleesta kaksi suoraan testitapausten luontiin askeleeseen seitsemän käyttäen lähtökohtana yllä esitettyä mahdollisten yhdistelmien taulukkoa.

4.7 Yhteenveto

Tässä luvussa aloitettiin tutkielman empiirisen osuuden kokeellinen tutkimus esittelemällä tutkimuksen motiivi ja lähtötilanne. Tutkielmassa tarkasteltavan graafisten käyttöliittymien regressiotestauksen automatisoinnin pilotoinnin kohteeksi päädyttiin valitsemaan jo osittain valmis sovellus, joka tässä tarkasteltavassa projektissa muutettiin vastaamaan uutta konseptia ja siirrettiin toimimaan uuden alustaversion päälle. Kohteeksi valittiin pikalinkkien hallintasovellus, sillä sen tiedettiin aiheuttaneen ongelmia aiemmin kehityksen aikana. Lisäksi sovellus on tarpeeksi monimutkainen, jotta testauksen automatisoinnin vaikutusta voidaan tutkia tarkemmin. Linkkien hallintasovellus oli siis jo valmis ja testattu toimivaksi manuaalisesti aiemmin, mutta projektissamme sovellus siirrettiin toiseen ympäristöön päivitetyn alustan päälle sekä tehtiin joitakin tyylillisiä ja konseptiin liittyviä muutoksia. Sovelluksen perustoiminnallisuus kuitenkin pysyi samana. Toiminnallisuuteen tehtiin kuitenkin sen verran muutoksia, että regressiotestaus oli tarpeellista.

Automaattisilla testeillä voidaan varmistua, että toiminnallisuus toimii kokonaisuutena myös muutosten jälkeen eivätkä ne ole rikkoneet jo aiemmin toiminutta toteutusta. Tällä sovelluksella on ollut taipumusta mennä epäkuntoon joidenkin muutosten jälkeen, joten regressiotestausten automatisointi sopii tälle sovellukselle erityisen hyvin.

Tämän sovelluksen automaattiseen regressiotestaukseen valittiin 14 testitapausta, jotka esitettiin taulukossa 5. Testauksen viitekehykseen rakennettiin DARTista riisuttu malli, jossa osa automaattisista tehtävistä suoritetaan manuaalisesti ja lisäksi osa askelista on poistettu kokonaan, koska niiden suorittaminen manuaalisesti ei ole järkevää ja seuraavien vaiheiden automatisoinnin puuttuminen ei mahdollista niiden järkevää käyttöä.

5 TESTITAPAUSTEN KIRJOITTAMINEN JA SUORITTAMINEN SELENIUM IDELLÄ

Tässä luvussa esitetään testitapausten kirjoittaminen ja suorittaminen Selenium IDEllä. Luvussa käsitellään Selenium IDEä ja sen syntaksia yleisesti aina asennuksesta lähtien ja lopulta jatketaan tutkielman tapaustutkimuksen automatisointiprosessia DARTin askeleesta 7.

5.1 Selenium IDEn käyttöönotto

Tutkimusta varten tehty Selenium IDEn asennus sujui helposti ja nopeasti. Latauslinkki (http://release.seleniumhq.org/selenium-ide/1.10.0/selenium-ide-1.10.0.xpi) löytyi SeleniumHQ:n internet-sivuilta ja asennus suoritettiin suoraan selaimen liitännäiseksi. Selaimena oli testien aikaan käytössä Mozilla Firefoxin versio 18.0 ja Seleniumista IDE:n versio 1.10.0. Selenium IDE oli käytettävissä alle viidessä minuutissa asennuksen aloittamisesta, mutta koko asennusprosessiin latauslinkin etsimisineen sekä dokumentaation lukemiseen kului puoli tuntia.

Ensimmäisen järkevän ja toimivan testitapauksen kirjoittamiseen kului aikaa puolitoista tuntia. Tässä ajassa on huomioitava se, että kirjoittaminen aloitettiin täysin puhtaalta pöydältä ja se sisälsi myös Seleniumin toimintaan ja komentoihin perehtymistä. Lopullisissa, tässä tutkielmassa esitettävissä testitapauksissa ei tästä ensimmäisestä testitapauksesta kuitenkaan ole yhtään osaa, sillä testiä parannettiin myöhemmin huomattavasti ja se kirjoitettiin käytännössä kokonaan alusta komentojen ja ymmärryksen kasvettua.

Selenium IDE:n käyttöönottoon ja ensimmäisen testin kirjoittamiseen ja suorittamiseen riitti tässä tapauksessa alle kaksi tuntia, vaikka minulla ei ollut työkalusta lainkaan aiempaa kokemusta. Myöhempien testitapausten kirjoittaminen oli huomattavasti nopeampaa, sillä työkalu oli jo tutumpi ja logiikka paremmin hallussa. Selenium IDEn oppimiskäyrä lähti erittäin jyrkästi nousuun heti alussa ja sen vuoksi sen käyttöönotto sekä testitapausten kirjoittaminen oli

alusta asti nopeaa. On kuitenkin mainittava, että jäljempänä esitetyt testitapaukset eivät valmistuneet lopulliseen muotoonsa yhdellä kertaa, vaan niitä testattiin kirjoittamisen aikana useita kertoja ja täydennettiin sekä lisättiin logiikkaa. Jatkossa on kuitenkin mahdollista hyödyntää aiempaa kokemusta sekä osia aiemmista testitapauksista uusien testien luontiin ja näin ollen vähentää vaadittavaa työtä entisestään.

Selenium IDE tarjoaa testitapausten kirjoittamiseen lisäksi tallenna-ja-toista-toiminnallisuuden. Tässä tutkielmassa sitä ei kuitenkaan hyödynnetä, sillä testitapauksista on tarkoitus saada hieman monimutkaisempia kuin mihin tallenna-ja-toista-työkalu suoraan kykenee. Lisäksi tässä tutkielmassa testattavan sovelluksen erikoisluonne ja toteutuksen rakenne vaikeuttavat huomattavasti tallenna-ja-toista-työkalun käyttöä. Työkalu on kuitenkin hyvä apu pienien testien tekemiseen. Sillä voidaan myös rakentaa laajemmankin testitapauksen runko ja sitten vain täydentää vajaiksi jääneet osat manuaalisesti.

Selenium IDEssä on kuitenkin rajoituksena se, että se ei toimi suoraan Microsoftin Internet Explorer –selaimella. Projektissamme kohdeselaimina ovat juuri Internet Explorer 7 ja 9. Testien automatisointi Firefox-selaimen liitännäisellä ei tämän seikan vuoksi ole täydellinen valinta, vaan testit pitää suorittaa erikseen manuaalisesti Internet Explorerilla. Tätä en pidä kuitenkaan liian suurena ongelmana, sillä toiminnallisuuden kriittiset kohdat voidaan testata kuitenkin automaattisesti Firefoxilla ja mahdollisten selainriippuvaisten ongelmien tällä kohdalla käyttää manuaalista testausta. Lisäksi käyttöliittymässä voi graafisesti olla huomattaviakin eroja eri selainten välillä ja nämä erot on testattava erikseen manuaalisesti.

5.2 Selenium IDEn käyttöliittymä

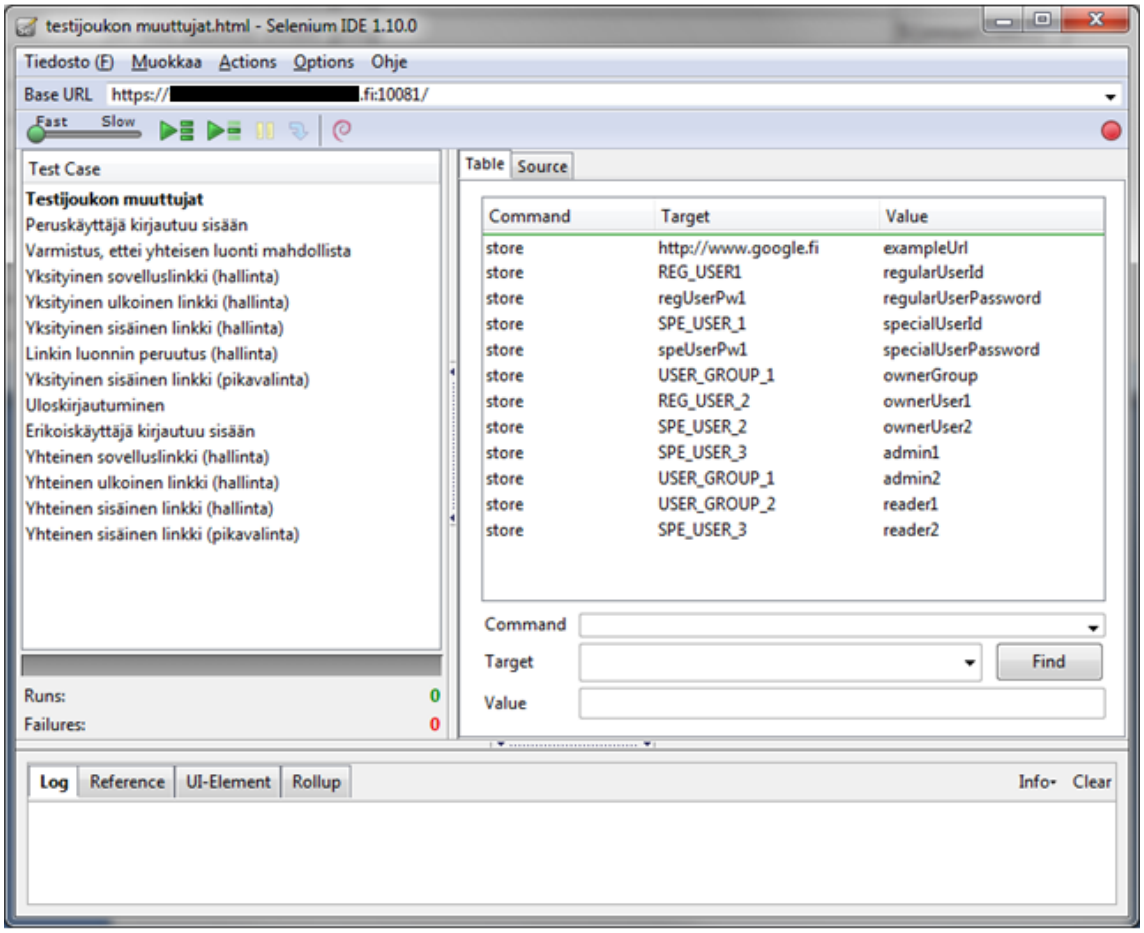
Selenium IDE käynnistetään selaimen työkaluriville ilmestyvästä painikkeesta. Käyttöliittymä on melko yksinkertainen koostuen toimintopainikkeista, testitapauslistasta, testitapauksen hallintanäkymästä ja konsolista, jossa näytetään mm. ajon aikaisia viestejä ja tuloksia. Kuviossa 7 on esitetty Selenium IDEn käyttöliittymä, jossa on näkyvillä luotuja testitapauksia sekä muuttujajoukon testitapauksen askelten määritys.

Editorissa näytetään vasemmalla testijoukon testitapaukset. Valittu testitapaus näytetään yksityiskohtaisesti oikealla. Testitapaus koostuu yhdestä tai useammasta käskystä ja testijoukko kahdesta tai useammasta testitapauksesta. Kuvasta nähdään, että tässä tutkielmassa käsiteltävä testijoukko koostuu useista testitapauksista, jotka on johdettu järjestelmän vaatimuksista.

Kuviossa 7 on esillä testijoukon ensimmäinen testitapaus. Se ei kuitenkaan ole varsinainen testitapaus, vaan siinä tehdään muuttujatallennuksia muita testijoukon testitapauksia varten. Näin ollen testit voidaan kirjoittaa muuttujien avulla yleiskäyttöisemmiksi. Vaikka Holmes ja Kellogg (2006) raportoivatkin havainneensa Seleniumin käytössä ongelmia muuttujien kanssa, tässä tutkimuksessa muuttujien käyttö oli suoraviivaista. Muuttujia kuitenkin käytettiin

vain syötearvojen tallentamisessa ja siihen tarkoitukseen ne sopivat oikein hyvin.

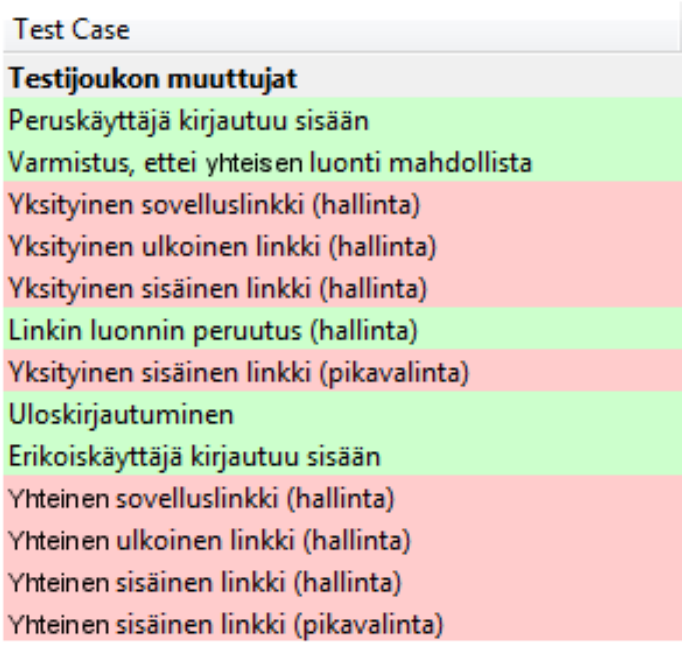
Tässä tutkimuksessa muuttujien tallennustestitapauksessa muuttujiin on tallennettu pikalinkkien luontiin käytettävä URL-osoite sekä erilaisten käyttäjien (peruskäyttäjä ja erikoiskäyttäjä) käyttäjätunnuksia ja salasanoja. Näitä käyttäjiä voidaan nyt käyttää testitapauksissa ja testeihin saadaan vaihtelua vain muuttamalla tähän testitapaukseen toiset käyttäjät. On kuitenkin huomattava, että tässä tapauksessa roolit ovat kuitenkin oleellisia ja testit onkin rakennettu siten, että erikoiskäyttäjäksi määritetyn käyttäjän on oltava erikoiskäyttäjä tai muuten testi epäonnistuu. Kuvassa olevat käyttäjätiedot on muutettu tätä raporttia varten geneerisiksi eivätkä ne vastaa suoraan todellisia käyttäjätunnuksia.



KUVIO 7 Selenium IDEn peruskäyttöliittymä

Mikäli testi menee läpi, se näytetään Selenium IDEn käyttöliittymässä vihreällä värillä, kun taas läpäisemätön ajo näytetään punaisella. Jos testitapauksessa on yksikin testi, joka ei mene läpi, koko testitapaus merkitään epäonnistuneeksi. Kuviossa 8 on havainnollistettu ajettujen testitapausten läpäisy. Punaisella merkityissä testitapauksissa on vähintään yksi testi jäänyt läpäisemättä. Testitapaus voidaan avata tarkempaan tarkasteluun, jolloin voidaan tarkastella testi-

tapauksen suoritusta yksittäisten askelten mukaan. Kaikki varsinaiset testit värikoodataan myös testitapauksen sisällä, jolloin voidaan helposti nähdä, mikä tai mitkä testit ovat tässä testitapauksessa jääneet läpäisemättä. Jos kokonainen testitapaus on vihreä, tarkoittaa se samalla myös sitä, että kaikki siinä olevat testit ovat menneet läpi eikä testin suorituksen tarkempaan tarkasteluun ole tarvetta.



KUVIO 8 Testien läpäisy havainnollistetaan värikoodauksella.

Testien kirjoittaja voi tehdä testitapauksiin myös kommentteja sekä tulostuksia helpottamaan testien tarkastelua suorituksen jälkeen. Seuraavassa esimerkissä näin on tehty ja niistä kerrotaan tarkemmin testien yksityiskohtaisemman läpikäynnin yhteydessä.

5.3 Esimerkki: Yhteisen sovelluslinkin luonti hallintapaneelin kautta

Kehittämämme linkkisovelluksen yksi vaatimus on se, että erikoiskäyttäjän on voitava luoda yhteinen linkki sovellukseen. Tarkastellaan tämän vaatimuksen testitapausta ”Yhteinen sovellus (hallinta)”, jonka testitapaus on esitetty kokonaisuudessaan taulukkomuotoisena esityksenä liitteessä 1. Taulukossa on kolme saraketta, jotka vastaavat Seleniumin testien syntaksia. Ensimmäinen sarake on *command* (komento), johon määritetään Seleniumin kielen, *selenesen*, mukainen komento. Keskimmäinen sarake on *target* (kohde), johon määritetään esimerkiksi html-elementti, jota komento koskee. Viimeinen sarake on *value* (arvo),

johon määritetään nimensä mukaisesti jonkin arvo, esimerkiksi tekstikenttään syötettävä arvo.

Selenium IDEssä voidaan käyttää työkalun omien komentojen lisäksi myös JavaScript-käskyjä kirjoittamalla *target*-kentän arvoksi *javascript{komennot;}*. Testitapausta tarkasteltaessa havaitaankin, että ensimmäisen rivin käsky on Seleniumin oma *echo*-komento, joka tulostaa tietoa käyttöliittymän tulostusalueelle. *Target*-kentässä arvona onkin sitten jo käytetty JavaScript-kieltä *javascript{startTime=new Date(); startTime.getTime();}* tulostamaan tieto nykyisestä kellonajasta. Tätä tietoa käytetään testin ajon yhteydessä selvittämään testitapauksen suoritukseen kulunut aika. Vastaava komento on testitapauksen lopussa.

echo	javascript{startTime=new Date(); start-Time.getTime();}	

Varsinainen testitapaus alkaa tallentamalla satunnainen linkin otsikko muuttujaan nimeltä *generatedTitle*. Tätä arvoa käytetään myöhemmin testitapauksessa. Tallennus tehdaan *store*-komennolla. Tällä rivillä on *target*-arvona JavaScript-kielinen käsky, jolla satunnainen nimi luodaan. *Value*-kentän arvoksi annetaan sen muuttujan nimi, johon nimi tallennetaan. Muuttujan tallennuksen yhteydessä kannattaa kiinnittää huomiota muuttujan kirjoitusasuun, sillä sen syntaksi poikkeaa siitä, kun tallennettua muuttujaa käytetään myöhemmin. Lisäksi luotu nimi tulostetaan *echo*-komennolla lokiin helpottamaan mahdollisesti tehtävää manuaalista tarkistusta.

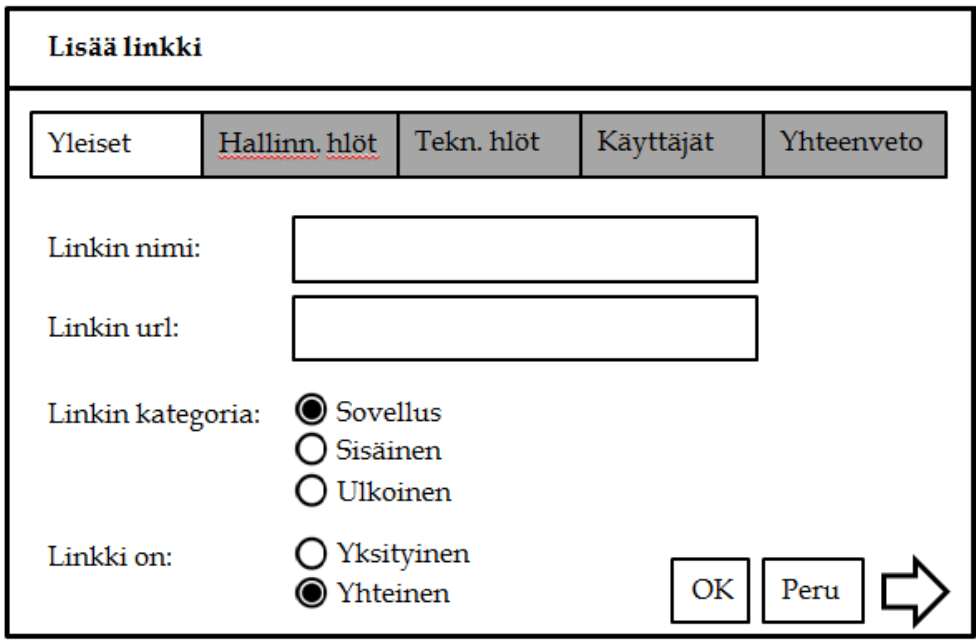
store	javascript{"ASeleniumTest" + Math.floor(Math.random()*1000);}	generatedTitle
echo	${generatedTitle}	

Tallennuksen jälkeen navigoidaan linkin hallintaan käyttäen hyväksi *open*- ja *click*-komentoja. *Open*-komennolla voidaan avata jokin web-sivu URL-osoitteen avulla ja *click*-komennolla simuloida hiirennapsautusta jonkin sivulla olevan elementin kohdalta. Tässä tapauksessa on käytetty id-arvoja, sillä niiden voidaan olettaa olevan yksilöllisiä koko sivulla. Lopulta on navigoitu uuden linkin luonnin näkymään (KUVIO 6). Tässä näkymässä linkin nimeksi annetaan *type*-komennolla testitapauksen alussa arvottu merkkijono ja osoitteeksi testijoukon ensimmäisessä testitapauksessa määritetty URL. Muuttujan nimen käytössä on nyt erilainen kirjoitusasu kuin tallennuksen yhteydessä: *${generatedTitle}*, kuten aiemmin todettiin.

open	/wps/myportal/xxxxx/etusivu	
click	shortcuts	
click	id=open-shortcut-manager	
click	id=shortcutManagementCreateNew	
type	id=shortcutUrlField	${exampleUrl}
type	id=shortcutNameField	${generatedTitle}

Koska nyt testataan yhteisen linkin luontia, käytetään jälleen *click*-komentoa ja id-arvoa (tässä tapauksessa *visibility2*) valitsemaan Yhteinen-radiopainike. Koska valittaessa yhteisen linkin luonnin radiopainike käyttöliittymään tulee lisää valintoja välilehtien muodossa (KUVIO 9), valitaan seuraavaksi luontivelhossa seuraavalle välilehdelle ohjaava nuolipainike oikeasta alakulmasta (*click id=shortcutWizardNext*).

click	id=visibility2	
click	id=shortcutWizardNext	



KUVIO 9 Yhteisen linkin luonnissa dialogissa näytetään useita välilehtiä

Toiminnallisuuden vaatimuksena on, että avautuvaan välilehteen voidaan määrittää linkin hallinnolliset henkilöt. Lisäksi vaatimuksena on, että hallinnollinen henkilö voi olla vain käyttäjä, ei ryhmä. Varmistetaan ensin, että kenttään ei voida antaa ryhmää, joten syötetään käyttäjähakuun aiemmin erillisessä muuttujien tallentamista varten luodussa testitapauksessa tallennettu muuttuja *${ownerGroup}*. Linkkienluontisovelluksessa syötetään kaikki käyttäjät ja ryhmät erillisellä käyttäjienhakutoiminnolla, jossa on syötettyjen merkkien mukaisesti tuloksia suodattava valintalistaus. Selenium IDE vaatii sen vuoksi hieman normaalia monimutkaisempaa toiminnallisuutta: ensin pitää käyttää normaalisti *type*-komentoa ja tämän lisäksi vielä *typeKeys*-komentoa, jolla simuloidaan käyttäjän näppäimenpainalluksia. Ensimmäinen komento lisää kenttään halutun tiedon ja toisen avulla saadaan tekstinsyöttötunnistustoiminnallisuus käyttöön ja sitä kautta valittavissa olevien käyttäjien lista näkyviin.

type	id=add-shortcut-owner-group	${ownerGroup}
typeKeys	id=add-shortcut-owner-group	${ownerGroup}
pause	2000	

Tässä vaiheessa tulee testitapauksen ensimmäinen varsinainen testi. *VerifyText-Present*-komennolla varmistetaan, että arvona annettu teksti ”Ei hakutuloksia, tarkista hakutermi.” näkyy ruudulla. Tässä luodaan samalla DARTin askeleen 8 mukainen odotettu tulos. Koska hallinnollinen henkilö ei saa olla ryhmä, ryhmän syöttämisen ei pitäisi tuottaa hakutuloksia, vaan tieto siitä, että hakutuloksia ei löydy. Tässä testissä on kuitenkin huomioitava sen luotettavuus ja kattavuus. On nimittäin mahdollista, että (1) ryhmä löytyy mutta sovellus näyttää silti ilmoituksen, että hakutuloksia ei löytynyt (sovellus näyttää ilmoituksen virheellisesti), tai (2) testattavaa ryhmää ei löydy mutta muita hakuehtoihin osuvia ryhmiä löytyy (jolloin testin vaatimaa ilmoitusta ei näytetä). Näissä tapauksissa testi menee läpi, vaikka vaatimus ei välttämättä täyttyisikään. Tässä tapauksessa testi on kuitenkin jätetty tällaiseksi, sillä ongelman mahdolliset haitat on arvioitu sen verran pieniksi, että tämä on hyväksyttävää. Joka tapauksessa testin tulos värikoodataan Selenium IDEn käyttöliittymässä joko vihreällä tai punaisella sen mukaan, meneekö testi läpi vai ei.

verifyTextPresent	Ei hakutuloksia, tarkista hakutermi.	

Edellisessä kohdassa käytettiin testiin *verifyText*-komentoa. Seleniumissa on *verifyn* lisäksi myös kaksi muuta mahdollista tapaa tehdä testejä: *assert* ja *waitFor*. *Assert*in käyttäminen testiin lopettaa testin suorittamisen heti virheen sattuessa, kun taas tässä käytetty *verify* jatkaa testin suorittamista ja kirjaa virheen lokiin. *WaitFor* odottaa määritellyn ajan jonkin ehdon täyttymistä ja päästää testin läpi, mikäli ehto täyttyy valitun ajan kuluessa tai on täyttynyt jo aiemmin. Tämä testitapa sopii hyvin esimerkiksi AJAX-toteutuksiin (Selenium, 2014.). Koska tässä tutkimuksessa tarkoituksena on ajaa pitkiä sarjoja automaattisia testejä, on *verify* ainoa oikea vaihtoehto, sillä testien suorittamisen pitää jatkua virheestä huolimatta.

Seuraavaksi testataan, että kenttään voidaan lisätä käyttäjiä. Kenttään lisätään käyttäjä muuttujista *${ownerUser1}* ja *${ownerUser2}*. Valitut käyttäjät näytetään käyttöliittymässä ja ne voidaan haluttaessa poistaa klikkaamalla niiden yhteydessä olevaa poistokuvaketta. Kuten kuvio 7 osoittaa, käyttäjät näihin muuttujiin on valittu siten, että toinen on peruskäyttäjä (*REG_USER_2*) ja toinen erikoiskäyttäjä (*SPE_USER_2*). Lisäysten jälkeen toinen käyttäjistä poistetaan ja sen jälkeen varmistetaan, että poistettua käyttäjää ei enää löydy valittujen käyttäjien listalta komennolla *verifyTextNotPresent* käyttäen arvona *${ownerUser2}*.

type	id=add-shortcut-owner-group	${ownerUser1}
typeKeys	id=add-shortcut-owner-group	${ownerUser1}
pause	2000	

mouseOver	css=.ui-menu-item > a	
click	css=button.add-this-button	
type	id=add-shortcut-owner-group	${ownerUser2}
typeKeys	id=add-shortcut-owner-group	${ownerUser2}
pause	2000	
mouseOver	css=.ui-menu-item > a	
click	css=button.add-this-button	
click	css=#selected-shortcut-owners > li + li img	
verifyTextNotPresent	{ownerUser2}	

Vastaavasti menetellään myös ylläpitäjien ja lukijoiden välilehdillä. Ylläpitäjien testissä varmistetaan lisäksi se, että lisätyn käyttäjän tai ryhmän on oltava eri-koiskäyttäjäryhmän jäsen. Testi suoritetaan varmistamalla, että *${regularUserId}*-muuttujalla haettaessa ei löydetä hakutuloksia.

click	id=shortcutWizardNext	
type	id=add-shortcut-admin-group	${regularUserId}
typeKeys	id=add-shortcut-admin-group	${regularUserId}
pause	2000	
verifyTextPresent	Ei hakutuloksia, tarkista hakutermi.	
type	id=add-shortcut-admin-group	${admin1}
typeKeys	id=add-shortcut-admin-group	${admin1}
pause	5000	
mouseOver	css=.ui-autocomplete + .ui-autocomplete a	
click	css=.ui-autocomplete + .ui-autocomplete a button.add-this-button	
type	id=add-shortcut-admin-group	${admin2}
typeKeys	id=add-shortcut-admin-group	${admin2}
pause	3000	
mouseOver	css=.ui-autocomplete + .ui-autocomplete a	
click	css=.ui-autocomplete + .ui-autocomplete a button.add-this-button	
click	css=#selected-shortcut-admins > li + li + li img	
verifyTextNotPresent	${admin2}	
click	id=shortcutWizardNext	
type	id=add-shortcut-user-group	${reader1}
typeKeys	id=add-shortcut-user-group	${reader1}
pause	2000	
mouseOver	css=.ui-autocomplete + .ui-autocomplete + .ui-autocomplete a	
click	css=.ui-autocomplete + .ui-autocomplete + .ui-autocomplete a button.add-this-button	
type	id=add-shortcut-user-group	${reader2}
typeKeys	id=add-shortcut-user-group	${reader2}
pause	2000	
mouseOver	css=.ui-autocomplete + .ui-autocomplete + .ui-autocomplete a	
click	css=.ui-autocomplete + .ui-autocomplete + .ui-autocomplete a button.add-this-button	

click	css=#selected-shortcut-users > li + li +li img	
verifyTextNotPresent	${reader2}	

Luontivelhon viimeinen välilehti on yhteenvetosivu, jossa näytetään tehdyt valinnat. Testissä käytetään *verifyTextPresent*- ja *verifyTextNotPresent*-komentoja. Testi on puutteellinen sikäli, että testin läpäisyyn riittää se, että tarkistettava teksti löytyy (tai ei löydy) sivulta. Testi ei ota kantaa siihen, löytyykö teksti oikeasta paikasta, eli esim. onko omistajaksi määritetty käyttäjä listattu omistajien alle vai esim. lukijaksi. Itse asiassa etsittävä teksti voi olla vaikka jossakin toisessa sovelluksessa, jos se kuitenkin on samalla HTML-sivulla. Lisäksi tässä tapauksessa tulee rajoituksia testitapausten muuttujien valintaan: samaa käyttäjää tai ryhmää ei voida tallentaa eri muuttujiin, sillä ne voivat olla ristiriidassa testin suorituksessa. Esimerkiksi jos käyttäjä *REG_USER1* on tallennettu sekä muuttujiin *${ownerUser1}* että *${reader2}*, testi epäonnistuu, sillä testitapaus sisältää seuraavat testit: (1) *verifyTextPresent* arvolla *${ownerUser1}* ja (2) *verifyTextNotPresent* arvolla *${reader2}*. Näistä toinen on aina epätosi, jos näiden muuttujien arvot ovat samat.

click	id=shortcutWizardNext
pause	2000
verifyTextPresent	${exampleUrl}
verifyTextPresent	${specialUserId}
verifyTextPresent	${ownerUser1}
verifyTextPresent	${admin1}
verifyTextPresent	${reader1}
verifyTextNotPresent	${ownerUser2}
verifyTextNotPresent	${admin2}
verifyTextNotPresent	${reader2}

Seuraavaksi linkki tallennetaan tietokantaan. Nyt pitää varmistaa, että linkki on tallentunut oikein, joten automaattinen testi etsii sen hallintatyökalun tallennettujen linkkien listasta ja avaa sen infonäkymän, joka vastaa käytännössä linkin luonnin yhteenvetosivua mutta ainoastaan lukumuodossa, jossa muutoksia ei voida tehdä. Jos infosivu avautuu, linkki on tallentunut oikein, sillä se löytyy linkkien hallintatyökalusta. Infonäkymässä varmistetaan samat tiedot kuin luontivaiheen yhteenvetovälilehdellä.

click	id=button_save	
pause	2000	
click	css=li:has(a:contains(${generatedTitle})) > a	
pause	5000	
verifyElementPresent	css=#selectedContainer a.title:contains(${generatedTitle})	
click	css=a:contains(${generatedTitle}) +a	
pause	1000	
verifyTextPresent	${exampleUrl}	
verifyTextPresent	${specialUserId}	
verifyTextPresent	${ownerUser1}	

verifyTextPresent	${admin1}	
verifyTextPresent	${reader1}	
verifyTextNotPresent	${ownerUser2}	
verifyTextNotPresent	${admin2}	
verifyTextNotPresent	${reader2}	

Seuraavaksi on varmistettava, että linkki näkyy valittuna omien pikalinkkien listauksessa. Testi sulkee nyt pikalinkin luontisovelluksen, navigoi omien linkkien listaukseen ja varmistaa, että listauksesta löytyy testitapauksen alussa määritetty muuttuja *${generatedTitle}*.

click	id=shortcutInfoBack
pause	1000
click	id=shortcutManagementClose
open	/wps/myportal/xxxxx/etusivu
click	shortcuts
pause	1000
verifyTextPresent	${generatedTitle}

Lopuksi vielä poistetaan luotu linkki ja varmistetaan, että linkki on poistunut.

click	id=open-shortcut-manager
pause	2000
click	css=a.title:contains(${generatedTitle}) + a + a
pause	1000
chooseOkOnNextConfirmation	
click	id=shortcutManagementDelete
verifyConfirmation	Oletko varma, että haluat poistaa oikotien ${generatedTitle}?
pause	2000
click	id=shortcutManagementClose
pause	10000
open	/wps/myportal/xxxxx/etusivu
click	shortcuts
pause	3000
verifyTextNotPresent	${generatedTitle}

Aivan viimeisenä testitapauksessa on vielä lokiin tulostavat lauseet, joiden avulla saadaan selville testin suorittamiseen kulunut aika. Tässä käytetään testin alussa ja lopussa tallennettujen aika-arvojen erotusta.

echo	javascript{endTime=new Date(); endTime.getTime();}	
echo	javascript{"Test completed: "+ (endTime-startTime)/1000+" seconds";}	

Testitapauksessa on useita *pause*-komentoja erilaisilla arvoilla. Nämä on lisätty testiin sitä varten, että testi odottaa esim. tietokantojen päivittymisen tai sivun lataukset, ennen kuin suorittaa tehtäviä. Taukoja luovia komentoja kannattaa kuitenkin käyttää vain tarvittaessa, sillä jokainen taukokomento pidentää testin suorittamiseen kuluvaa aikaa. Pienillä testijoukoilla sillä ei kuitenkaan ole suurta merkitystä mutta ajankäytön tehokkuuden tarve kasvaa sitä mukaa mitä suurempia kokonaisuuksia testataan.

5.4 Testitapausten esitysmuodot

Tässä tutkielmassa testit luotiin Selenium IDE –työkalulla käyttäen graafista käyttöliittymää. Testien käsittely voidaan hoitaa alusta loppuun muokkauksineen, poistoineen ja ajoineen graafisella käyttöliittymällä. Testit voidaan kuitenkin esittää myös taulukko- ja HTML-muodoissa sekä tuoda käyttöliittymään ajettavaksi erillisestä tiedostosta. Taulukkomuotoisesta esityksestä on esimerkki liitteessä 1. Liitteessä 2 puolestaan on esimerkki vastaavan testitapauksen esityksestä HTML-muodossa. Esityksestä on poistettu osa testiriveistä tilan säästämiseksi. Tämä esitysmuoto alkaa normaaleilla HTML:n määrityksillä ja sisältää lisäksi <*link rel="selenium.base" href="url"* /> -määrityksen, jossa URL on testeissä käytettävä perusosoite. Tätä osoitetta voidaan käyttää hyväksi suhteellisten osoitteiden kanssa, jolloin samoja testitapauksia voidaan ajaa eri ympäristöissä (esimerkiksi kehitys-, testi-, ja laadunvarmistusympäristössä) vaihtamalla vain tämän perusosoitteen arvo. Perusosoitteen määrityksen jälkeen annetaan testitapauksen otsikko *title*-elementissä. Varsinainen sisältö määritettään *body*-elementin sisässä olevaan taulukkoon. Jokainen testitapauksen rivi määritetään *tr*-elementtien sisään *td*-elementeillä siten, että ensimmäinen *td*-elementti merkitsee komentoa, toinen kohdetta ja kolmas arvoa. Kuviossa 10 on esimerkkinä rivi, jossa Googlen URL-osoite tallennetaan *exampleUrl*-nimiseen muuttujaan.

```
<tr>
  <td>store</td>
  <td>http://www.google.fi</td>
  <td>exampleUrl</td>
</tr>
```

KUVIO 10 Esimerkki testitapauksen rivin esityksestä HTML-muodossa

HTML-tiedosto loppuu taulukon sekä lopulta *body*- ja *html*-elementtien sulkemiseen. Lisäksi HTML-muotoisessa esityksessä voidaan testeihin kirjoittaa kommentteja normaalilla HTML-kommenttisyntaksilla <*!-- Kommentti* -->.

5.5 Huomioita kommennoista

Selenium IDE:ssä komentojen syöttämisessä tulee olla tarkkana, että ne menevät oikeisiin sarakkeisiin. Esimerkiksi muuttujien käytössä arvon asettaminen suoritetaan store-komennolla. Komennon kohteeksi (*target*) määritetään arvo ja arvoksi (*value*) muuttujan nimi. Lisäksi on huomattava, että store-komennon yhteydessä muuttujan nimi ilmoitetaan ilman erikoismerkkejä, kun taas siihen viitattaessa myöhemmin se merkitään ${muuttujanNimi}-merkinnällä. Seleniumin komennoista löytyy dokumentaatio Selenium-projektin www-sivuilta (Selenium, 2014).

6 AUTOMAATTISEN JA MANUAALISEN TESTAUKSEN VERTAILU

Tässä luvussa tarkastellaan testien automatisoinnin vaikutuksia testauksen laatuun sekä käytettyyn aikaan. Vertailussa huomioidaan sekä testien suunnitteluun, kirjoittamiseen että suoritukseen kuluneet ajat. Lopuksi pohditaan tutkimuksen tuloksia ja niiden vaikutuksia jatkotoimenpiteisiin organisaatiossamme.

6.1 Testien suorittaminen

Kun kaikki halutut testitapaukset on saatu kirjoitettua Seleniumiin, voidaan testijoukon suorittaminen aloittaa. Suoritukseen voidaan valita joko yksittäinen testitapaus tai kokonainen testijoukko. Selenium IDEn käyttöliittymän avulla voidaan testien etenemistä seurata reaaliaikaisesti mutta se ei ole välttämätöntä, sillä testien suorituksen jälkeen järjestelmä tuottaa automaattisesti värikoodatun listauksen testitapauksista ja niiden läpimenosta. Näin ollen testit voidaan jättää käyntiin esimerkiksi vaikka jokaisen työpäivän päätteeksi ja katsoa seuraavana aamuna testien tulos.

Tässä tutkielmassa automaattisten testien suoritusta verrataan manuaaliseen testaukseen. Vertailu tehdään tarkastelemalla testaukseen kuluvaa aikaa. Mikäli automaattisen ja manuaalisen testauksen vertailussa saadaan havaintoja häiriöistä, jotka löytyvät vain toisella testaustavalla, huomioidaan myös ne. Häiriöiden etsiminen ei kuitenkaan ole tässä tutkielmassa oleellista, sillä koska kyseessä on regressiotestaus, manuaalisten ja automaattisten testien oletetaan olevan identtisiä. Ihmisen ja koneen väliset erot voivat kuitenkin vaikuttaa häiriöiden havainnointiin ja sen vuoksi häiriöiden määrä otetaan kuitenkin mukaan tarkasteluun, jos tulokset antavat siihen aihetta. Testitapausten identtisyyden vuoksi suoritan manuaalisen testauksen itse enkä käytä manuaaliseen testaukseen toista henkilöä. Tällä pyritään myös varmistamaan se, että testaus tosiaan on identtinen sekä manuaalisesti että automaattisesti. Lisäksi manuaalisten testien ajallisen tehokkuuden halutaan olevan mahdollisimman hyvä.

Manuaalisen testauksen tehokkuus perustuu tämän sovelluksen testauskokemukseen.

Testien vertailuun on otettu 10 testitapausta, jotka on luotu ja valittu käyttäen aiemmin esitettyjen mahdollisten polkujen taulukoita apuna. Mukaan on otettu sekä peruskäyttäjän että erikoiskäyttäjän luomat eri kategorioiden pikalinkit hallinnan kautta sekä lisäksi kummankin luoma pikalinkki pikatoiminnon kautta. Lisäksi mukaan on otettu linkin luonnin peruuttaminen mutta vain peruskäyttäjällä, sillä toiminnallisuuden oletetaan toimivan identtisesti riippumatta siitä, onko kyseessä perus- vai erikoiskäyttäjä. Taulukossa 6 on esitetty testitapaukset ja niihin kuluneet ajat sekä automaattisten että manuaalisten testien osalta. Taulukossa on lisäksi esitetty aputestitapauksia, joilla testien suorittaminen mahdollistetaan. Ensimmäinen testitapaus tallentaa halutut arvot testitapauksien muuttujiin. Toinen testitapaus kirjaa peruskäyttäjän sisään. Tämä on pakollinen vaihe, jotta testin suorittaminen on mahdollista, mutta se ei kuitenkaan kuulu varsinaiseen testattavaan toiminnallisuuteen. Samoin peruskäyttäjän testitapausten jälkeen tulevat uloskirjautuminen sekä erikoiskäyttäjän sisäänkirjautuminen ovat mukana ainoastaan auttamassa testien suorittamista. On kuitenkin huomioitava, että vaikka nämä eivät kuulu varsinaisesti testattavaan kokonaisuuteen, on niillä silti aikavaikutus testin kokonaiskestoon.

TAULUKKO 6 Automaattisten ja manuaalisten testien ajon vertailu kehitysympäristössä

Testitapaus	**Automaattsen testin kesto (s)**	**Manuaalisen testin kesto (s)**
Testijoukon muuttujat	0,027	–
Peruskäyttäjä kirjautuu sisään	3,951	16
Varmistus, ettei yhteisen linkin luonti mahdollista	3,626	5
Yksityinen sovelluslinkki (hallinnan kautta)	26,233	23
Yksityinen ulkoinen linkki (hallinnan kautta)	13,764	28
Yksityinen sisäinen linkki (hallinnan kautta)	22,889	26
Linkin luonnin peruutus (hallinnan kautta)	3,414	4
Yksityinen sisäinen linkki (pikatoiminnolla)	12,200	18
Uloskirjautuminen	3,023	3
Erikoiskäyttäjä kirjautuu sisään	4,961	6
Yhteinen sovelluslinkki (hallinnan kautta)	62,646	111
Yhteinen ulkoinen linkki (hallinnan kautta)	62,902	116
Yhteinen sisäinen linkki (hallinnan kautta)	68,602	111
Yhteinen sisäinen linkki (pikatoiminnolla)	63,292	111
YHTEENSÄ	351,530	578

Kehitysympäristössä automaattisesti suoritetun testijoukon suorittamiseen kului 5 minuuttia ja 52 sekuntia. Manuaalisiin testeihin kului aikaa 9 minuuttia ja 38 sekuntia. Automaattisten testien käynnistämisen jälkeen niiden suorittamiseen ei kuitenkaan tarvittu ihmisen panosta kuin vasta tulosten tarkastelun yhteydessä. Koska tässä tapauksessa kaikki testitapaukset menivät läpi, tarvittiin automaattisten testien suorittamiseen ihmisen tekemää työtä ainoastaan puoli

minuuttia: Selenium IDEn käynnistämiseen, testijoukon tiedoston avaamiseen ja testien suorittamisen aloittamiseen.

Automaattisten testien ajat on saatu testien ympärille kirjoitetuilla aikaleimoilla. Jokaisen testitapauksen alkuun kirjoitettiin JavaScript-kielinen komento, jolla otettiin talteen testin aloitusajankohdan aikaleima. Testitapauksen loppuun puolestaan kirjoitettiin JavaScript-komento, joka tulosti lokiin testin lopussa tallennetun aikaleiman ja alussa talteen otetun aikaleiman välisen erotuksen. Manuaalisten testien kellotus on tehty manuaalisesti sekuntikellolla. Manuaalisessa testauksessa jokainen testitapaus suoritettiin erikseen ja niiden välillä oli tauko, jota ei ole huomioitu ilmoitetussa kokonaisajassa. Automaattiset testit sen sijaan jatkoivat suoritusta välittömästi edellisen testin jälkeen. Käytännön testauksessa testitapausten välillä kuluu aikaa esimerkiksi uuden testitapauksen valmisteluun. Taulukossa ilmoitetut ajat ovat lisäksi mahdollisimman tehokkaasti suoritetun testauksen tulosta. Tavallisessa testauksessa ei useinkaan päästä samaan tehokkuuteen johtuen inhimillisistä tekijöistä ja mahdollisesti laajemmista testattavista kokonaisuuksista. Tässä tapauksessa testattavat tapaukset olivat hyvin selkeästi määritellyt ja testaaja keskittyi suorittamaan testitapaukset läpi mahdollisimman tehokkaasti, sekä tämän raportin laatimista varten testikierroksia tuli myös tavallista useampia. Tulokset kuitenkin osoittavat, että automaatti hoiti testauksen nopeammin kuin kokenut ihminen, vaikka manuaaliset testitkin pyrittiin tekemään mahdollisimman nopeasti. Lisäksi tässä tapauksessa automaattitestien pitkät kestot selittyvät testitapauksien *pause*-komennoilla, ja testejä onkin mahdollista pyrkiä nopeuttamaan optimoimalla komentojen välillä tapahtuvia odotusaikoja. Tämän kestoluokan testijoukoissa se ei kuitenkaan ole tarpeellista, jos testit ajetaan esimerkiksi iltaisin.

Kokemus on osoittanut, että manuaalista testaustehokkuutta ei välttämättä voida ylläpitää kovin pitkään, joten manuaaliseen testaukseen kuluva aika tulee kasvamaan testauksen edetessä. Kuten aiemmin on jo todettu, kokemus on osoittanut myös sen, että manuaaliseen testaamiseen turtuu helposti, jos samaa asiaa pitää testata toistuvasti. Erityisesti regressiotestauksen kohdalla rasitus voi olla hyvinkin korkea, sillä sen tarkoitus on osoittaa, että jo aiemmin testattu toiminnallisuus toimii edelleen. Näin ollen testaaja voi oikoa mutkia ja päättää, että joitakin asioita jätetään testaamatta, koska ne on testattu jo niin monta kertaa aiemmin.

6.2 Tulosten tarkastelu ja johtopäätökset

Tämän tutkielman puitteissa tehdyn empiirisen tutkimuksen perusteella saatiin vertailukohtaa automaattisen ja manuaalisen testauksen välille. Tässä vaiheessa kyse oli kuitenkin vain yhdestä järjestelmän osasta ja sen vuoksi tämä tutkielma ei anna täysin kattavaa selvitystä graafisten käyttöliittymien regressiotestauksen automatisoinnista organisaatiossamme. Tutkielma kuitenkin osoittaa sen, että testitapausten laatiminen on helppoa ja se onnistuu helposti myös ilman ohjelmointikokemusta. Tätä tutkielmaa varten laatimieni testitapausten kirjoit-

taminen vei kuitenkin merkittävästi aikaa, mutta on muistettava, että käytännön kokemukseni tästä työkalusta oli tutkimuksen aloittamishetkellä lähes olematon. Testitapausten luonnin edetessä havaitsin kuitenkin selvästi, että testitapausten kirjoittaminen nopeutui ja helpottui koko ajan. Seleniumin käyttämän kielen oppiminen matkan varrella helpotti merkittävästi testitapausten kirjoittamista ja loogista rakennetta. Lisäksi jo kirjoitettujen testien käyttäminen uusien testitapausten pohjana nopeutti prosessia.

Kun jätetään pois Seleniumin käyttöönotto ja ensimmäiset harjoitustestitapaukset, tämän tutkielman testitapausten kirjoittamiseen kului aikaa yhteensä 15 tuntia. Se tarkoittaa noin yhtä tuntia jokaista testitapausta kohti. Todellisuudessa käytetty aika ei jakautunut testitapauksille tasan, vaan ensimmäisen testin kirjoittaminen kesti huomattavasti muita pidempään. Tähän oli syynä kokemuksen puute ja lisäksi toiset testitapaukset voitiin kirjoittaa kopioimalla edellinen testitapaus ja muuttamalla siitä vain oleelliset kohdat. Testien valmistumisen jälkeen niiden suorittaminen on kuitenkin käytännössä ilmaista, sillä ihmispanoksen tarve on 15 minuutin sijaan vain 30 sekuntia. Lisäksi testaus voidaan suorittaa useammin ja näin ollen saada virheitä kiinni nopeammin.

Lisäksi, kuten taulukosta 6 havaittiin, automaattisten testien suorittaminen oli manuaalista testaamista nopeampaa. Ainoastaan yksi testitapaus kesti automaattisesti suoritettuna kauemmin kuin manuaalisesti. Tämä johtui siitä, että testitapauksiin on kirjoitettu odotuskäskyjä joidenkin toiminnallisuuksien yhteyteen, jotta järjestelmä ehtii suorittaa toiminnot loppuun ennen kuin automaattinen testi suorittaa seuraavan lauseen. Testitapausjoukon ensimmäinen testitapaus, eli muuttujien ottaminen käyttöön, kesti vajaat kolme sekunnin sadasosaa eli käytännössä mitättömän ajan. Manuaalisessa testauksessa muuttujien lukeminen tapahtuu testien suorittamisen yhteydessä vaikkapa testausohjeesta tai tämän tutkielman tapauksessa testaajan omasta muistista. Mikäli muuttujia olisi hyvin paljon, se todennäköisesti vaikuttaisi manuaaliseen testaamiseen, kun taas automaattiseen testaamiseen sillä ei juurikaan olisi merkitystä.

6.3 Tulosten merkitys organisaatiollemme

Tämän tutkielman tulosten valossa väitän, että organisaatiossamme kannattaa jatkaa graafisten käyttöliittymien automaattisten regressiotestaamisen kehittämistä ja tutkimista. Vaikka tämän tutkielman testitapausten laatimiseen menikin runsaasti aikaa, sen tuloksena syntyneistä automaattisista testeistä saatiin hyötyä ohjelmiston vakauden varmistamisessa jatkuvassa integraatiossa. Vaikka itse pikalinkkitoiminnallisuuteen ei enää tehty muutoksia testien kirjoittamisen jälkeen, automaattisten testien suorittaminen havaitsi kuitenkin häiriöitä, jotka aiheutuivat muualle järjestelmään tehdyistä muutoksista. Havaitut ongelmat eivät kuitenkaan liittyneet suoranaisesti pikalinkkitoiminnallisuuteen mutta skriptivirheiden vuoksi esimerkiksi pikalinkkitoiminnallisuuden dialogit eivät erään päivityksen jälkeen avautuneet. Nämä ongelmat olisi havaittu hyvin

pienellä työmäärällä myös manuaalisella testauksella, sillä ne ilmenivät heti, kun pikalinkkitoimintoa yritti käyttää. Manuaalista regressiotestausta ei kuitenkaan tehdä projektissamme jatkuvasti johtuen käytettävissä olevista resursseista. Tässä tapauksessa ongelma tulikin ilmi heti sen synnyttyä siksi, että testit suoritettiin heti muutoksen jälkeen. Näin ei kuitenkaan olisi tehty, mikäli testien suoritus ei olisi ollut automaattista.

Pikalinkkisovellukseen on jo suunniteltu tulevaisuudessa tehtäviä muutoksia, jotka pakottavat tekemään muutoksia edellä esitettyihin testeihin. Sovellukseen tehtävä muutos on kuitenkin tässä tapauksessa melko pieni ja vaikuttaa vain pikalinkkien hallintasovelluksen logiikkaan. Tämän vuoksi sovellukseen tehtävät muutokset voidaan testata näillä jo kirjoitetuilla testitapauksella hyvin pienin muutoksin. Tässä vaiheessa arvioisin testien uudelleenkirjoittamiseen kuluvan aikaa noin puoli tuntia. Tämän jälkeen meillä on käytössämme laaja, automaattinen testipatteristo sovelluksen testaamiseen. Verrattuna kokonaan manuaaliseen testaamiseen ajan säästöksi voidaan arvioida tässä vaiheessa kolmesta viiteen tuntia. Automaattisen testaamisen ollessa vielä kypsyysmallin alimmalla tasolla, emme voi nojautua vielä pelkästään automaattiseen testaukseen. Näilläkin testeillä voimme kuitenkin jo varmistaa muuttumattomien osien toiminnan automaattisesti ja mikäli niihin tulee häiriöitä sovelluksen muutoksista johtuen, ne saadaan selville heti lähes ilman ihmispanostusta.

Uskon kuitenkin, että varsinainen hyöty regressiotestien automatisoinnista saadaan vasta sitten, kun toiminta on järjestelmällisempää ja kattaa suuremman osan toiminnallisuuksista. Kaikkien mahdollisten tapausten kirjoittaminen on liian raskasta, ellei myös testitapausten luontia automatisoida, kuten DART esittää. Tässä vaiheessa organisaatiossamme ei kuitenkaan ole mahdollisuutta lähteä testauksen automatisointiin niin korkealla tasolla, joten automatisoinnin testitapaukset tuleekin valita toisella menetelmällä. Tässä tutkielmassa on esitetty myös erilaisia vaihtoehtoja testitapausten valintaan ja ne ovatkin organisaatiossamme yksi jatkotutkimuksen aihe.

Tässä tutkielmassa esitettyjen testiesimerkkien lisäksi organisaatiossamme on tehty muutamia muitakin automaattisia testejä käyttäen Selenium IDEä. Eräässä testitapauksessa pyrittiin varmistamaan toiminnallisuuden toiminta järjestelmän uudelleenkäynnistyksen jälkeen, mikä tarkoitti manuaalisesti noin kymmenen minuutin työtä. Automaattisen testin kirjoittaminen kesti 40 minuuttia ja suorittaminen alle minuutin. Tämän esimerkin järjestelmän uudelleenkäynnistys tehdään noin kaksi kertaa kuukaudessa, joten tässä tapauksessa automaattisen testin käyttäminen maksoi itsensä takaisin kahdessa kuukaudessa ja tuo jatkossa joka kuukausi lisää ajansäästöä.

Eräässä projektissamme on käytössä kolme erillistä ympäristöä, jossa toiminnallisuus tulee testata ja todeta toimivaksi ennen kuin se voidaan julkaista tuotantoon. Automaattisten testien käyttäminen vähentää manuaalisen testauksen tarvetta jokaisessa ympäristössä sen jälkeen, kun testit on kirjoitettu yhdelle ympäristölle. Ympäristökohtaiset erot tulee ottaa huomioon, mutta tässä tapauksessa niitä on hyvin vähän ja suunnitellut automaattiset testit valikoiduille toiminnoille voidaan käyttää sellaisinaan kaikissa ympäristöissä. Erityisen hyö-

dyllisiä testit ovat tapauksissa, joissa manuaalinen testaaminen keskitetään johonkin uuteen toiminnallisuuteen samalla, kun automaatti varmistaa jo aiemmin olemassa olleiden toiminnallisuuksien toiminnan.

Kuten aiemmin todettiin, uusien testien kirjoittaminen on nopeampaa, sillä vanhoja testejä voidaan käyttää uusien pohjana. Näin ollen regressiotestauksen kattavuutta voidaan parantaa koko ajan pienemmällä työmäärällä ja samalla koko järjestelmän laatu paranee, sillä sitä testataan koko ajan enemmän ja laajemmin. Jo toteutettujen testien perusteella organisaatiollemme voisi olla edullista jopa tehdä näitä testejä omalla kustannuksella, koska siten voimme parantaa toimittamiemme järjestelmien laatua ja sitä kautta asiakastyytyväisyyttä. Samalla järjestelmän laadun parantuminen on suoraan verrannollinen takuutöiden määrään.

Automaattisten testien käyttöönoton jälkeen (tässä tarkastelujaksona on kolme kuukautta) ei havaittu merkittäviä ongelmia toiminnallisuuksissa päivittäisten regressiotestien suorituksen yhteydessä. Tämä selittyy kuitenkin sillä, että testijoukko kohdistui vain sellaisiin järjestelmän osiin, joihin ei tarkastelujakson aikana tehty muutoksia. Testit kuitenkin havaitsivat yleisiä ongelmia, jossa koko järjestelmä ei toiminut oikein ja sen vuoksi myöskään yksittäisten toiminnallisuuksien testit eivät menneet läpi. Aiemman kokemuksen perusteella voin kuitenkin sanoa, että järjestelmän osat ovat hyvin epävakaita siinä vaiheessa, kun testattaviin toiminnallisuuksiin tehdään säätöä. Ensimmäinen tulikoe tähän on tulossa, kun pikalinkkisovellusta päivitetään. Joka tapauksessa tästä automatisoinnista on jo ollut hyötyä sen vuoksi, että olemme saaneet varmistettua, että järjestelmä toimii muutoksista huolimatta. Ilman automaattisia testejä näitä varmistuksia ei olisi tehty todennäköisesti lainkaan ja tällöin mahdolliset ongelmat olisivat päässeet pahimmassa tapauksessa tuotantoympäristöön asti.

Selenium IDE on erittäin helppokäyttöinen ja nopea testitapausten luonti- ja suorittamistyökalu. Meidän organisaatiossamme rajoituksena tämän työkalun käytölle on kuitenkin sen selainriippuvuus. Koska Selenium IDE toimii ainoastaan Mozilla Firefoxin kanssa, joudutaan testaus tekemään kuitenkin vielä erikseen manuaalisesti esimerkiksi Internet Explorer –selaimella. Tästäkin huolimatta Selenium IDEn käyttö regressiotestitapausten automatisointiin vaikuttaa tulosten perusteella varteenotettavalta vaihtoehdolta. Joka tapauksessa päivittäiset regressiotestit parantavat nykyistä tilannetta huomattavasti, vaikka testaus suoritettaisiinkin vain yhdellä selaimella ja manuaalista testausta suoritettaisiin muilla selaimilla sillä tasolla kuin nykyisinkin.

Tutkimuksen lupaavat tulokset sekä ylipäätään organisaatiossamme vaikuttavat kokemukset ovat antaneet sykäyksen jatkaa testauksen kehittämisessä ylipäätään, mutta myös automatisoinnin näkökulmasta. Yksikkötestien suorittaminen on ollut automatisoitua jo pitkään, mutta nyt myös regressio- ja hyväksymistestien automatisointi on noussut ajankohtaiseksi. Tästä tutkimuksesta saadaan hyvä pohja jatkotutkimukselle ja automatisoinnin kehittämiselle.

7 YHTEENVETO

Tässä tutkielmassa käsiteltiin ensin testausta yleisesti sekä automatisoinnin ja regressiotestauksen näkökulmasta. Luvussa kaksi tarkasteltiin automatisoinnin perustelemista sekä graafisten käyttöliittyminen automaattisen testaamisen ongelmia. Lisäksi luvussa esiteltiin muutama oleellinen käsite testauksen laajasta kentästä. Regressiotestauksen pääasiallinen tehtävä on varmistaa, että testattava ohjelmisto toimii oikein myös sen jälkeen, kun siihen on tehty muutoksia. Uudelleentestauksesta regressiotestauksen erottaa se, että uudelleentestauksessa pyritään varmistamaan, että ohjelmistoon tehty korjaus on todella korjannut ongelman. Regressiotestaus puolestaan pyrkii varmistamaan sen, että aiemmin toimineet ominaisuudet toimivat myös tehtyjen muutoksien jälkeen. Sen vuoksi regressiotestausta tulisi suorittaa jokaisen muutoksen jälkeen. Tämä puolestaan tekee regressiotestauksesta loistavan kohteen automatisoinnille.

Automaattisten testien kirjoittaminen on kuitenkin aikaa vievää toimintaa. Sen vuoksi onkin voitava perustella, kannattaako testit automatisoida. Erityisesti graafisten käyttöliittymien kohdalla ongelmana on niiden luonne, jossa syötteitä voi tulla useasta eri suunnasta ja käyttäjällä on enemmän valtaa kuin muissa järjestelmissä. Graafiset käyttöliittymät kuitenkin ovat vahvasti läsnä yhä useammassa järjestelmässä, joten niiden testaaminen on tärkeää. Regressiotestien automatisointi on usein järkevää, sillä niitä suoritetaan usein. Näin ollen testin kirjoittamisesta syntyvät kustannukset saadaan todennäköisemmin katettua. Lisäksi automaattisella testaamisella on joitakin etuja verrattuna manuaaliseen testaamiseen. Testitapauksia voidaan ajaa käytännössä koko ajan ilman merkittävää manuaalista valvontaa. Lisäksi tietokone voi havaita sellaisia virheitä, joita ihminen ei kykene havaitsemaan ainakaan helposti. Erityisen helposti graafisten käyttöliittymien regressiotestejä voidaan tehdä tallenna-ja-toista-tyyppisillä työkaluilla.

Tallenna-ja-toista-pohjaisilla työkaluilla testien kirjoittaminen on hyvin nopeaa mutta toisaalta niiden luonteen vuoksi testien elinkaari voi olla hyvinkin lyhyt. Koska testi tekee täsmälleen sen, mitä tallennusvaiheessa on näytetty, voi esimerkiksi käyttöliittymäkomponentin paikan sijainnin muutos tehdä koko testin virheelliseksi. Sen vuoksi onkin pyrittävä ainakin tarkistamaan tällaisten

työkalujen tuottama koodi, mikäli se on nähtävissä ja muokattavissa. Tällaista työkalua on hyödynnetty myös tämän tutkielman empiirisen osuuden tekemiseen, joskin kaikki testit on kirjoitettu suoraan skriptikielellä käyttämättä työkalun tallenna-ja-toista-ominaisuutta.

Luvussa kolme esiteltiin kirjallisuudesta löytyneitä malleja ja viitekehyksiä testauksen automatisointiin. Samassa luvussa esiteltiin myös graafisten käyttöliittymien automaattiseen regressiotestaukseen sopivia työkaluja. Tutkielmani empiiriseen osuuteen valittiin käytettäväksi sovellettuna Daily Automated Regression Tester sekä suoritustyökaluksi Selenium IDE. DARTiin päädyttiin sen vuoksi, että se ottaa huomioon koko testausprosessin automatisoinnin aina graafisen käyttöliittymän paloittelusta lähtien testitapausten laadintaan ja suorittamiseen asti. Organisaatiomme testauksen kypsyystason vuoksi tässä tutkielmassa ei kuitenkaan vielä tutkittu koko prosessin automatisointia vaan keskityttiin testien suorituksen automatisointiin. Selenium IDE puolestaan valittiin sen helpon käyttöönoton ja nopean omaksumiskäyrän vuoksi. Selenium IDEllä voidaan tehdä helposti testitapauksia joko tallenna-ja-klikkaatoiminnolla, erillisellä skriptikielellä tai näiden yhdistelmällä. Selenium IDE on tarpeeksi yksinkertainen, jotta sitä voi käyttää myös hyvin vähäisellä ohjelmointikokemuksella. Toisaalta se tarjoaa kuitenkin hyvät välineet myös hieman monimutkaisempien testien laatimiseen.

Luvussa neljä aloitettiin tutkielman kokeellisena tutkimuksena toteutettu empiirinen osuus esittelemällä organisaatiotamme sekä tutkimuksen lähtötilanne ja motiivit. Lisäksi siinä esiteltiin tarkasteltavan sovelluksen käyttöliittymä ja tunnistettiin testitapauksia selvittämällä sovelluksen käyttötapauspolkuja. Tutkimuksen tavoitteena oli rakentaa malli, jota kehittämällä voidaan graafisten käyttöliittymien automaattinen regressiotestaus ottaa käyttöön koko organisaatiossamme. Tutkielman perusteella voitiin tehdä alustavia johtopäätöksiä siitä, että tällainen malli on organisaatiossamme mahdollinen, mutta yksinään sen perusteella ei mallia voida kuitenkaan ottaa vielä laajempaan käyttöön. Tutkimuksen tuloksien perusteella on kuitenkin selvästi havaittavissa, että graafisten käyttöliittymien automaattista testausta voidaan suorittaa hyvinkin helposti ja sen vuoksi sen tutkimista kannattaa jatkaa, jotta siitä tulisi rutiinia kaikissa projekteissa. Myöhemmin, mikäli tulokset ovat rohkaisevia, tavoitteena voi olla esimerkiksi koko prosessin automatisointi, kuten DART-viitekehys esittää.

Luvussa viisi rakennettiin testattavaan sovellukseen liittyvät testitapaukset Selenium IDE –työkaluun ja suoritettiin ne. Luvun alussa esiteltiin Selenium IDEn käyttöönotto ja työkalun käyttöliittymää sekä ominaisuuksia. Tämän jälkeen käytiin läpi yksi tutkielmassa käytetyistä testitapauksista tarkemmalla tasolla testitapausten logiikan ja toiminnan selkeyttämiseksi. Lopulta luvussa kuusi tarkasteltiin automaattisen testien suorittamisen tuloksia ja verrattiin niitä vastaavien testien manuaaliseen suorittamiseen. Lopuksi otettiin kantaa tulosten merkitykseen projektissamme sekä organisaatiossamme ja pohdittiin jatkotoimenpiteitä.

Jatkossa testien kirjoittaminen pyritään saamaan kehityksen yhteyteen siten, että tehtävän (issue) valmistuttua se testataan ja siihen kirjoitetaan auto-

maattiset testit. Automatisoinnin kohteena olevat testit valitaan tapauskohtaisesti. Tehtäväkokonaisuuden (story) valmistuttua tehtävien testitapaukset varmistetaan ja kootaan yhteen testitapausjoukoksi. Tämän lähestymistavan etu on se, että testeistä saadaan automaattisia samalla, kun ne suoritetaan ensimmäisen kerran. Näin ollen niillä voidaan regressiotestata automaattisesti aina tarvittaessa heti alusta alkaen ja siten mahdolliset ongelmat voidaan havaita ja korjata mahdollisimman aikaisin. Tällä lähestymistavalla testien automatisointi ei kasaannu tehtäväkokonaisuuden valmistumisen jälkeiseksi tehtäväksi ja niistä saadaan hyöty heti alusta alkaen.

Testien automatisoinnissa sovelluskehityksen aikana tai ennen sen aloittamista on kuitenkin omat ongelmansa. On mahdollista, että testejä pitää muuttaa radikaalistikin niiden jo kertaalleen valmistuttua. Tällainen tilanne voi tulla esimerkiksi silloin, jos käyttöliittymässä tehdään muutoksia myöhemmissä vaiheissa ja niiden vaikutuksesta aiemmat testit eivät enää ole valideja. Lisäksi mahdolliset muutokset HTML-rakenteessa voivat muuttaa testit toimimattomiksi. DARTin täydellisen soveltamisen kuitenkin pitäisi osata käsitellä myös tällaiset tilanteet lähes automaattisesti pienellä manuaalisella työmäärällä. Tässä tutkielmassa sitä ei kuitenkaan voida varmistaa, sillä tutkielman tavoite on selvittää ainoastaan testien suorittamisen automatisointi eikä testien automaattinen päivittäminen kuulu tämän tutkielman tarkastelun alueelle. Sovellusten ja testien huolellinen suunnittelu kuitenkin pienentävät testitapausten muutostarpeista aiheutuvia riskejä. Tarkoituksena onkin pyrkiä selvittämään sovelluksen vaatimukset ja mahdollisesti kirjoittamaan valmiit testitapaukset jo ennen kuin sovelluksen toteutus alkaa. Tällöin tarve muutoksille myöhemmin pienenee huomattavasti, kun mahdolliset ongelmatilanteet on kartoitettu jo aiemmassa vaiheessa. BDD on hyvä menetelmä näiden ongelmien ratkomiseen ja sen avulla myös järjestelmän vaatimukset saadaan esille paremmin aiemmin.

Koska Selenium IDE on käytettävissä ainoastaan Mozilla Firefox-selaimen kanssa, se ei välttämättä ole paras vaihtoehto organisaatiollemme. Sen avulla on kuitenkin helppo päästä automatisointiin mukaan ja sen käytön ohella on mahdollista jatkaa muiden työkalujen tutkimista ja siihen onkin viime aikoina ollut kasvavaa kiinnostusta organisaatiossamme. Tästä voimme päätellä, että automatisoinnin merkitys on otettu vakavasti myös meillä. Kuitenkin, kuten kirjallisuudessakin on esitetty, automatisointiin kannattaa suhtautua maltillisesti ja rakentaa sitä pala kerrallaan, sillä täysin automatisoitua prosessia ei voida ottaa käyttöön suoraan alimmalta tasolta.

LÄHTEET

Agile Alliance (2013). *Bdd.* Agile Alliance and Institut Agile. Haettu 6.1.2014 osoitteesta http://guide.agilealliance.org/guide/bdd.html

Bach, J. (1999). *Test Automation Snake Oil.* Haettu 2.2.2014 osoitteesta http://www.satisfice.com/articles/test_automation_snake_oil.pdf.

Bertolino, A. (2007). Software Testing Research: Achievements, Challenges, Dreams. Teoksessa L. C. Briand & A. L. Wolf (toim.) *Future of Software Engineering, Minneapolis, May 23–25* (s. 85–103). Los Alamitos: IEEE Computer Society.

Bruns, A., Kornstädt, A. & Wichmann, D. (2009). Web Application Tests with Selenium. *IEEE Software 26(5)*, 88–91.

Burnstein, I., Suwanassart, T. & Carlson, R. (1996). Developing a Testing Maturity Model for Software Test Process Evaluation and Improvement. Teoksessa *Proceedings International Test Conference, Washington, DC, October 20–25*. (s. 581–589). Los Alamitos: IEEE Computer Society.

Crispin, L. (2006a). Driving Software Quality: How Test-Driven Development Impacts Software Quality. *IEEE Software 23(6)*, 70–71.

Crispin, L. (2006b). Hiring a tester with an agile attitude. *Better Software 8(3)*, 16–18, 37.

Douglas, V. L. R. (2010). *A case study of non-functional requirements and continuous improvement at a national communications system contractor.* Informaatioteknologian väitöskirja. Waldenin yliopisto.

Duggal, G. & Suri, B. (2008). *Understanding Regression Testing techniques.* Proceedings of the 2nd National Conference on Challenges and Opportunities, Mandi Gobindgarh, India, March 29.
Haettu 23.2.2014 osoitteesta
http://rimtengg.com/coit2008/proceedings/SW15.pdf

Dustin, E., Rashka, J. & Paul, J. (2008). *Automated Software Testing – Indroduction, Management, and Performance* (13. painos). New York: Addison-Wesley.

Gerrard, P. (1997). Testing GUI Applications. *The 5th European Conference on Software Testing, Analysis & Review, Edinburgh, November 24–28*. Haettu 3.5.2012 osoitteesta
http://gerrardconsulting.com/sites/default/files/Techgui.pdf

Hackner, D. & Memon, A. (2008). Test Case Generator for GUITAR. Teoksessa *Companion of the 30th International Conference on Software Engineering, Leipzig, May 10–18* (s. 959–960). New York: ACM Press.

Haikala, I. & Märijärvi, J. (2003). *Ohjelmistotuotanto* (9. painos). Helsinki: Talentum.

Harrold, M. J., Gupta, R. & Soffa, M. L. (1993). Driving Software Quality: How Test-Driven Development Impacts Software Quality. *ACM Transactions on Software Engineering and Methodology 2(3)*, 270–285.

Holmes, A. & Kellogg, M. (2006). Automating Functional Tests Using Selenium. Teoksessa J. Chao, M. Cohn, F. Maurer, H. Sharp & J. Shore (toim.) *Proceeding of Agile Conference, Minneapolis, July 23–28* (s. 270–275). Los Alamitos: IEEE Computer Society.

ISTQB (2007). *ISTQB:n testaussanasto*. International Software Testing Qualifications Board. Haettu 21.7.2012 osoitteesta http://www.fistb.fi/sites/fistb.ttlry.mearra.com/files/istqb_sanasto.pdf

Kajko-Mattsson, M., Jonson, M., Koroorian, S. & Westin, F. (2004). Lessons Learned from Attempts to Implement Daily Build. Teoksessa *Proceedings of the 8th European Conference on Software Maintenance and Reengineering, Tampere, Finland, March 24–26,* (s. 137–146). Los Alamitos: IEEE Computer Society.

Koskela, J. (2012). *Automaatiotestaus ja Robot Framework – Asennus, testien kirjoittaminen sekä ylläpidettävyys*. Ohjelmistotekniikan opinnäytetyö. Jyväskylän ammattikorkeakoulu.

Leotta, M., Clerissi, D., Ricca, F. & Tonella, P. (2013). Capture-Replay vs. Programmable Web Testing: An Empirical Assessment during Test Case Evolution. Teoksessa R. Lämmel, R. Oliveto & R. Robbes (toim.) *Proceedings of the 20th Working Conference on Reverse Engineering, Koblenz, Germany, October 14–17*. (s. 272–281). Los Alamitos: IEEE Computer Society.

Leung, H. K. N. & White, L. (1989). Insights into Regression Testing. Teoksessa *Proceedings of Conference on Software Maintenance, Miami, October 16–19* (s. 60–69). Los Alamitos: IEEE Computer Society.

Leung, H. K. N. & White, L. (1990). A Study of Integration Testing and Software Regression at the Integration Level. Teoksessa *Proceedings of Conference on Software Maintenance, San Diego, November 26–29* (s. 290–301). Los Alamitos: IEEE Computer Society.

Leung, H. K. N. & White, L. (1991). A Cost Model to Compare Regression Test Strategies. Teoksessa *Proceedings of Conference on Software Maintenance, Sorrento, October 15–17* (s. 201–208). Los Alamitos: IEEE Computer Society.

Marick, B. (1998). When should a Test Be Automated? Teoksessa *Proceedings of the 11th International Software Quality Week, San Francisco, May 26–29*. San Francisco: Software Research, Inc.

Memon, A. (2002). GUI Testing: Pitfalls and Process. *IEEE Computer*, 35(8), 87–88.

Memon, A. (2007). An event-flow model of GUI-based applications for testing. *Software Testing, Verification and Reliability, 17(3)*, 135–203.

Memon, A., Banerjee, I. & Nagarajan A. (2003a). What Test Oracle Should I Use for Effective GUI Testing? Teoksessa *Proceedings of the 18th IEEE International Conference on Automated Software Engineering, Montreal, October 6–10* (s. 164–173). Los Alamitos: IEEE Computer Society.

Memon, A., Banerjee, I., Hashmi, N. & Nagarajan A. (2003b). DART: A Framework for Regression Testing "Nightly/daily Builds" of GUI Applications Teoksessa *Proceedings of the International Conference on Software Maintenance, Amsterdam, September 22–26* (s. 410–419). Los Alamitos: IEEE Computer Society.

Memon, A., Nagarajan, A. & Xie, Q. (2005). Automating regression testing for evolving GUI software. *Journal of Sofware Maintenance and Evolution: Research and Practice 17,* 27–64.

Memon, A. & Soffa, M. L. (2003). Regression Testing of GUIs. Teoksessa *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, Helsinki, September 1–5* (s. 118–127). New York: ACM Press.

Memon, A., Soffa, M. L. & Pollack, M. (2001). Coverage Criteria for GUI Testing. *ACM SIGSOFT Software Engineering Notes, 26(5)*, 256–267.

Mittal, N. & Acharya, I. (2003). An Open Framework for Managed Regression Testing. Teoksessa D. Hogrefe & A Wiles (toim.) *Proceedings of the 15th International Conference, Testcom, Sophia Antipolis, France, May 26–28* (s. 265–278). Germany: Springer-Verlag.

Molyneaux, I. (2009). *The Art of Application Performance Testing: Help for Programmers and Quality Assurance* (1. painos). USA: O'Reilly Media.

Myers, G. J., Badgett, T. & Sandler C. (2012). *The Art of Software Testing*. (3. painos). Hoboken, New Jersey: John Wiley & Sons.

Nokia Siemens Networks (2009). *Robot Framework Quick Start Quide.* Haettu 6.10.2012 osoitteesta http://robotframework.googlecode.com/svn/trunk/doc/quickstart/quickstart.html

Nokia Siemens Networks (2011). *Robot Framework User GuideVersion 2.6.2*. Haettu 6.10.2012 osoitteesta http://robotframework.googlecode.com/hg/doc/userguide/RobotFrameworkUserGuide.html?r=2.6.2

Nokia Siemens Networks (2012). *Robot Framework*. Haettu 6.10.2012 osoitteesta http://code.google.com/p/robotframework/

Robinson, W. N. (2009). Seeking Quality through User-Goal Monitoring. *IEEE Software 26(5)*, 58–65.

Rothermel, G. & Harrold, M. J. (1996). Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering 22(8)*, 529–551.

Selenium (2014). *Selenium DocumentationSelenium*. Selenium Documentation Team. Haettu 1.2.2014 osoitteesta http://docs.seleniumhq.org/docs/index.jsp

Spillner, A., Linz, T. & Schaefer, H. (2011). *Software Testing Foundations* (3. painos). Santa Barbara: O'Reilly Media.

Srinivasan, D. & Gopalaswamy, R. (2008). *Software Testing: Principles and Practice* (6. painos). Chennai: Dorling Kindersley.

White, L. (1996). Regression Testing of GUI Event Interactions. Teoksessa *Proceedings of the International Conference on Software Maintenance, Monterey, California, November 4–8*, (s. 350–358). Los Alamitos: IEEE Computer Society.

Whittaker, J. (2000). What Is Software Testing? and Why Is It So Hard? *IEEE Software 17(1)*, 70–79.

Willmor, D. & Embury, S. M. (2005). A safe regression test selection technique for database-driven applications. Teoksessa *Proceedings of the 21st IEEE In-*

ternational Conference on Software Maintenance, Budapest, Hungary, September 26–29, (s. 421–430). Los Alamitos: IEEE Computer Society.

Wong, W. E., Horgan, J. R., London, S. & Agrawal, H. (1997). A Study of Effective Regression Testing in Practice. Teoksessa *Proceedings of the 8th International Symposium on Software Reliability Engineering, Albuquerque, NM, November 2–5,* (s. 264–274). Los Alamitos: IEEE Computer Society.

Xu, D., Xu, W., Bavikati, B. K. & Wong, W. E. (2012). Mining Executable Specifications of Web Applications from Selenium IDE Tests. Teoksessa *Proceedings of the 6th International Conference on Software Security and Reliability, Gaithersburg, Maryland, June 20–22* (s. 263–272). Los Alamitos: IEEE Computer Society.

LIITE 1 TESTITAPAUS YHTEISEN LINKIN LUONTIIN HALLINTAPANEELIN KAUTTA (TAULUKKOMUOTOINEN ESITYS)

Yhteinen sovellusoikotie (hallinta)		
echo	javascript{startTime=new Date(); startTime.getTime();}	
store	javascript{"ASeleniumTest" + Math.floor(Math.random()*1000);}	generatedTitle
echo	${generatedTitle}	
open	/wps/myportal/xxxxx/etusivu	
click	shortcuts	
click	id=open-shortcut-manager	
click	id=shortcutManagementCreateNew	
type	id=shortcutUrlField	${exampleUrl}
type	id=shortcutNameField	${generatedTitle}
click	id=visibility2	
click	id=shortcutWizardNext	
type	id=add-shortcut-owner-group	${ownerGroup}
typeKeys	id=add-shortcut-owner-group	${ownerGroup}
pause	2000	
verifyTextPresent	Ei hakutuloksia, tarkista hakutermi.	
type	id=add-shortcut-owner-group	${ownerUser1}
typeKeys	id=add-shortcut-owner-group	${ownerUser1}
pause	2000	
mouseOver	css=.ui-menu-item > a	
click	css=button.add-this-button	
type	id=add-shortcut-owner-group	${ownerUser2}
typeKeys	id=add-shortcut-owner-group	${ownerUser2}
pause	2000	
mouseOver	css=.ui-menu-item > a	
click	css=button.add-this-button	
click	css=#selected-shortcut-owners > li + li img	
verifyTextNotPresent	{ownerUser2}	
click	id=shortcutWizardNext	
type	id=add-shortcut-admin-group	${regularUserId}
typeKeys	id=add-shortcut-admin-group	${regularUserId}
pause	2000	
verifyTextPresent	Ei hakutuloksia, tarkista hakutermi.	
type	id=add-shortcut-admin-group	${admin1}
typeKeys	id=add-shortcut-admin-group	${admin1}
pause	5000	
mouseOver	css=.ui-autocomplete + .ui-autocomplete a	
click	css=.ui-autocomplete + .ui-autocomplete a button.add-this-button	
type	id=add-shortcut-admin-group	${admin2}

typeKeys	id=add-shortcut-admin-group	${admin2}
pause	3000	
mouseOver	css=.ui-autocomplete + .ui-autocomplete a	
click	css=.ui-autocomplete + .ui-autocomplete a button.add-this-button	
click	css=#selected-shortcut-admins > li + li + li img	
verifyTextNotPresent	${admin2}	
click	id=shortcutWizardNext	
type	id=add-shortcut-user-group	${reader1}
typeKeys	id=add-shortcut-user-group	${reader1}
pause	2000	
mouseOver	css=.ui-autocomplete + .ui-autocomplete + .ui-autocomplete a	
click	css=.ui-autocomplete + .ui-autocomplete + .ui-autocomplete a button.add-this-button	
type	id=add-shortcut-user-group	${reader2}
typeKeys	id=add-shortcut-user-group	${reader2}
pause	2000	
mouseOver	css=.ui-autocomplete + .ui-autocomplete + .ui-autocomplete a	
click	css=.ui-autocomplete + .ui-autocomplete + .ui-autocomplete a button.add-this-button	
click	css=#selected-shortcut-users > li + li +li img	
verifyTextNotPresent	${reader2}	
click	id=shortcutWizardNext	
pause	2000	
verifyTextPresent	${exampleUrl}	
verifyTextPresent	${specialUserId}	
verifyTextPresent	${ownerUser1}	
verifyTextPresent	${admin1}	
verifyTextPresent	${reader1}	
verifyTextNotPresent	${ownerUser2}	
verifyTextNotPresent	${admin2}	
verifyTextNotPresent	${reader2}	
click	id=button_save	
pause	2000	
click	css=li:has(a:contains(${generatedTitle})) > a	
pause	5000	
verifyElementPresent	css=#selectedContainer a.title:contains(${generatedTitle})	
click	css=a:contains(${generatedTitle}) +a	
pause	1000	
verifyTextPresent	${exampleUrl}	
verifyTextPresent	${specialUserId}	
verifyTextPresent	${ownerUser1}	
verifyTextPresent	${admin1}	
verifyTextPresent	${reader1}	
verifyTextNotPresent	${ownerUser2}	
verifyTextNotPresent	${admin2}	

verifyTextNotPresent	${reader2}	
click	id=shortcutInfoBack	
pause	1000	
click	id=shortcutManagementClose	
open	/wps/myportal/xxxxx/etusivu	
click	shortcuts	
pause	1000	
verifyTextPresent	${generatedTitle}	
click	id=open-shortcut-manager	
pause	2000	
click	css=a.title:contains(${generatedTitle}) + a + a	
pause	1000	
chooseOkOnNextConfirmation		
click	id=shortcutManagementDelete	
verifyConfirmation	Oletko varma, että haluat poistaa oikotien ${generatedTitle}?	
pause	2000	
click	id=shortcutManagementClose	
pause	10000	
open	/wps/myportal/xxxxx/etusivu	
click	shortcuts	
pause	3000	
verifyTextNotPresent	${generatedTitle}	
echo	javascript{endTime=new Date(); endTime.getTime();}	
echo	javascript{"Test completed: "+ (endTime-startTime)/1000+" seconds";}	

LIITE 2 TESTITAPAUS YHTEISEN LINKIN LUONTIIN HALLINTAPANEELIN KAUTTA (OSA HTML-MUOTOISESTA ESITYKSESTÄ)

```
<?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://xxxxxdev-
sistuot.xxxxx.fi:10081/" />
  <title>Yhteinen sovelluslinkki (hallinta)</title>
</head>

<body>
  <table cellpadding="1" cellspacing="1" border="1">
  <thead>
    <tr><td rowspan="1" colspan="3">Yhteinen sovelluslinkki (hallinta)
        </td>
    </tr>
  </thead>

  <tbody>
    <tr>
      <td>echo</td>
      <td>javascript{startTime=new Date(); startTime.getTime();}</td>
      <td></td>
    </tr>

    <tr>
      <td>store</td>
      <td>javascript{&amp;quot;ASeleniumTest&amp;quot;+Math.floor(Math.random()
          *1000);}
      </td>
      <td>generatedTitle</td>
    </tr>

    <tr>
      <td>echo</td>
      <td>${generatedTitle}</td>
      <td></td>
    </tr>

    <tr>
      <td>open</td>
      <td>/wps/myportal/xxxxx/etusivu</td>
      <td></td>
    </tr>
```

...

```
<!--Varmistetaan, että peruskäyttäjää ei voida lisätä muokkaajaksi-->
    <tr>
      <td>type</td>
      <td>id=add-shortcut-admin-group</td>
      <td>${regularUserId}</td>
    </tr>

    <tr>
      <td>typeKeys</td>
      <td>id=add-shortcut-admin-group</td>
      <td>${regularUserId}</td>
    </tr>

    <tr>
      <td>pause</td>
      <td>2000</td>
      <td></td>
    </tr>

    <tr>
      <td>verifyTextPresent</td>
      <td>Ei hakutuloksia, tarkista hakutermi.</td>
      <td></td>
    </tr>




...




    <tr>
      <td>verifyTextNotPresent</td>
      <td>${generatedTitle}</td>
      <td></td>
    </tr>

    <tr>
      <td>echo</td>
      <td>javascript{endTime=new Date(); endTime.getTime();}</td>
      <td></td>
    </tr>

    <tr>
      <td>echo</td>
      <td>javascript{&quot;Test completed: &quot;+ (endTime-
          startTime)/1000+&quot; seconds&quot;;}</td>
      <td></td>
    </tr>
  </tbody>

  </table>

</body>
</html>
```