Sauli Flinkman

Web-sovellusten testaaminen Selenium-testaustyökalun avulla

Tietotekniikan kandidaatintutkielma

26. huhtikuuta 2016

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Sauli Flinkman

Yhteystiedot: sauli.flinkman@gmail.com

Työn nimi: Web-sovellusten testaaminen Selenium-testaustyökalun avulla

Title in English: Testing Web Applications with Selenium Testing Tool

Työ: Kandidaatintutkielma

Sivumäärä: 26+0

Tiivistelmä: Eräs tapa testata web-sovelluksia on automaattinen testaus, jota tässä tutkielmassa tarkastellaan. Testaaminen on web-sovelluksen elinkaaressa tärkeä osa, sillä web-sovelluksilta vaaditaan nykyään paljon niin käytettävyyden, turvallisuuden kuin tehokkuudenkin osalta. Tutkielmassa käydään läpi automaattisen testauksen perusperiaatteita ja esitellään web-sovellusten automaattiseen testaukseen käytettävää työkalua nimeltä Selenium. Selenium rakentuu useasta eri komponentista ja tämä tutkielma perehtyy erityisesti näistä kahteen: Selenium IDE-ohjelmointiympäristöön sekä WebDriver-rajapintaan.

Avainsanat: Selenium, ohjelmistotestaus, automaattinen testaus, web-sovellus, käyttöliittymä

Abstract: One way to test web applications is automated testing which this thesis examines. Software testing is an important part of life cycle of any web application because so much emphasis is nowadays put on their usability, safety and efficiency. This thesis presents common principles of web application testing and a tool used for automated testing of web applications, Selenium. Selenium consists of multiple different components but this thesis focuses specifically on two of them: Selenium IDE and WebDriver interface.

Keywords: Selenium, software testing, automated testing, web-application, user interface

Kuviot

Kuvio 1. Selenium IDE -käyttöliittymä 10

Sisältö

1 JOHDANTO 1

2 WEB-SOVELLUKSET JA NIIDEN TESTAAMINEN 3
2.1 Ohjelmistotestaus 3
2.2 Web-sovellusten testaaminen 4
2.2.1 Funktionaalinen testaus 5
2.2.2 Automaattinen testaus 6

3 SELENIUM 9
3.1 Selenium IDE 9
3.1.1 Käyttöliittymä 9
3.1.2 Selenium-komennot 11
3.1.3 Testitapauksen luonti 12
3.2 Selenium WebDriver 12
3.2.1 Alustus 13
3.2.2 Käyttöliittymäelementtien paikantaminen 13
3.2.3 Testaaminen 14

4 YHTEENVETO 19

LÄHTEET 21

1 Johdanto

Ohjelmistotestaus on aikaavievä, mutta kuitenkin tarpeellinen prosessi ohjelmiston laadun selvittämiseksi sekä ohjelmistosta löytyvien virheiden löytämiseksi ja korjaamiseksi. Koska web-sovellukset ovat nykyään niin yleinen tapa tuoda joku palvelu ihmisten käyttöön, niiden laadulle on asetettava korkeat vaatimukset heti alusta pitäen. Silti web-sovellusten kehittäminen on usein nopea prosessi, joka tarkoittaa, että ohjelmistosta julkaistaan uusia versioita tiheään tahtiin ja testaaminen saattaa jäädä puutteelliseksi.

Web-sovellusten testaaminen tapahtuu työkaluilla, joilla voidaan testata monia web-sovelluksen toiminnan kannalta tärkeitä ominaisuuksia. Työkalujen avulla rakennetaan testitapauksia, jotka tarkastavat, että joku pieni osa ohjelmaa toimii kuten on tarkoitus. Testitapausten rakentaminen vie aikaa ja aivan kaikkea web-sovelluksen toimintaan liittyvää on vaikea joko ajan tai rahan puutteen vuoksi testata. On kuitenkin tärkeä saavuttaa korkea kattavuus testeille, jotta ohjelmistosta saataisiin eliminoitua mahdollisimman paljon virheitä. Haasteena on siis luoda testitapaukset, jotka tarkastaisivat web-sovelluksen toiminnan mahdollisimman perusteellisesti niin, että virheen ilmetessä korjauskustannukset jäisivät niin alhaisiksi kuin mahdollista.

Yksi mahdollisuus testien kattavuuden lisäämiseksi on automaattinen testaaminen, joka on ohjelmistotestaukseen liittyvien toimintojen automatisointia. Näihin toimintoihin kuuluvat testiskriptien ajo ja kehitys, testeihin liittyvien vaatimusten todennus ja automaattiseen testaamiseen suunniteltujen testausohjelmistojen käyttö (Collins ja Lucena 2012). Helposti toistettavat testitapaukset voivat olla työläitä toistaa manuaalisesti, jolloin voi olla järkevää käyttää automaattiseen testaamiseen soveltuvaa ohjelmistoa, kuten esimerkiksi Seleniumia.

Selenium on yksi web-sovellusten testaamiseen käytettävä palvelu. Se koostuu useasta automaattiseen testaamiseen käytettävästä komponentista, joihin tässä tutkielmassa perehdytään tarkemmin. Selenium-testaustyökaluun kuuluvat muun muassa Selenium IDE, WebDriver ja RC, jotka ovat kukin työkaluja automaattiseen testaamiseen. Selenium RC on tosin jo vanhentunut ja sen korvaajaksi tuli Selenium WebDriver, joka tunnetaan myös nimellä Selenium 2 (Selenium 2016). Tutkielmassa käsitelläänkin web-sovellusten automaattista testaamista

Selenium IDE:n ja WebDriver-rajapinnan avulla, tavoitteena käydä läpi web-sovellusten testaamisen käytänteitä ja komponenttien toimintaa.

Selenium IDE ja WebDriver ovat kumpikin työkaluja automaattiseen testaamiseen, mutta metodit testien luomiseen ja ajamiseen ovat erilaisia. Selenium IDE:n toiminta perustuu testien nauhoittamiseen ja toistamiseen, mikä on yksi tapa testata graafisen käyttöliittymän toimintaa, eikä se vaadi tietoa web-sovelluksen kehitykseen käytetyistä teknologioista. Toinen tapa luoda automaattisia testejä ovat WebDriver-rajapinnan ohjelmoidut testit, jotka tarjoavat mahdollisuuden luoda monimutkaisempia testitapauksia, mutta vastaavasti vaativat enemmän osaamista testien luomisesta sekä ohjelmoinnista yleisesti.

Tutkielma on systemaattinen kirjallisuuskatsaus, jonka ensisijaisena tehtävänä on kertoa, mikä on Selenium ja mitä tarkoitetaan web-sovellusten automaattisella testaamisella. Tutkielman 2. luvussa käsitellään web-sovellusten testaamista. Siinä määritellään aluksi ohjelmistotestauksen perusasioita ja edetään käsittelemään web-sovellusten testaamista. Samalla tuodaan esille funktionaalisen ja automaattisen testaamisen käsitteet. Luku 3 tarkastelee kahta edellä mainittua Selenium-työkalun komponenttia, niiden ominaisuuksia sekä käytänteitä testitapausten luontiin. Lukuun 4 on koottu omia ajatuksia automaattisesta testaamisesta, johtopäätöksiä tutkielman annista sekä mahdollisia jatkotutkimuksen aiheita.

2 Web-sovellukset ja niiden testaaminen

Tässä luvussa tarkastellaan web-sovellusten testaamiseen olennaisesti liittyviä asioita. Luvussa tuodaan esille testaamiseen liittyvää termistöä ja Selenium-työkalun toiminnan ymmärtämisen kannalta tärkeitä perusasioita. Luvussa määritellään aluksi ohjelmistotestaukseen yleisesti liittyviä asioita, jonka jälkeen tarkastellaan web-sovellusten testaamisen käytänteitä. Lisäksi määritellään termit funktionaalinen- ja automaattinen testaus.

2.1 Ohjelmistotestaus

Ohjelmistotestaus on olennainen asia ohjelmiston kehityksessä. Testaamisen tarkoituksena on havaita ohjelmistossa esiintyviä virheitä ja löytää niille ratkaisu. Ilman testaamista ei ole varmuutta siitä, että ohjelmisto toimii juuri niin kuin pitääkin kaikissa tilanteissa (Altaf ym. 2015).

Tärkeänä osana ohjelmistotestauksessa ovat testitapaukset. Testitapaus on toimenpide, joka sisältää ohjelmistolle annettavat syötteet ja näiden perusteella odotetut tulokset (Wu 2012). Testitapauksen kohteena on joku ohjelmiston osa, usein funktio tai esimerkiksi käyttöliittymän painikkeen toiminnallisuuden tarkastaminen. Wu (2012) esittää, että tyypillisesti testitapaus voidaan näyttää tauluna, jossa on seuraavat tiedot:

- ohjelmistoa koskeva vaatimus, johon testitapaus liittyy
- ennakkoehdot, jotka kuvaavat ohjelmiston tilan ennen kuin testitapauksen voi suorittaa
- vaiheet, jotka kuvaavat testitapauksen suorituksen vaiheet
- odotetut tulokset eli ohjelmiston tila testitapauksen suorittamisen jälkeen.

Suunnitellessa testitapauksia on tärkeää muistaa, että testien rakentaminen vie yleensä aikaa pois ohjelmiston kehitykseltä. Tärkeää olisi siis saada testeille mahdollisimman korkea kattavuus niin, että olennaisimmat, ohjelmiston toiminnan kannalta tärkeimmät testitapaukset laitetaan etusijalle. Bertolino ja Marchetti (2005) esittävät, että testitapaukset tulisi valita huomioiden testien tehokkuus, kustannukset sekä toteuttamiskelpoisuus.

Koska testien tekeminen ei ole ilmaista, on siis tärkeää muodostaa tasapaino kustannusten,

ajankäytön sekä testitapausten perusteellisen tarkastelun suhteen (Bertolino ja Marchetti 2005). Jotta parhaaseen mahdolliseen tulokseen päästäisiin, on keksitty useita eri strategioita sekä testitapausten valintaan, että testien toteuttamiseen. Testitapaukset voidaan valita esimerkiksi ohjelmiston vaatimusmäärittelyn ohjaamana, ohjelmakoodin kattavuuden maksimoimiseksi tai niin, että löytyisi mahdollisimman paljon ohjelmakoodissa esiintyviä virheitä (Bertolino ja Marchetti 2005).

Testien toteuttaminen tehdään testitapausten valintaan käytetyn metodin pohjalta (Bertolino ja Marchetti 2005). Toteuttamiseen liittyy olennaisesti testaustyökalun valinta, joka voi vaikuttaa merkittävästi testitapausten luomiseen. Jotkut ohjelmistot saattavat vaatia suuren datamäärän testaamista, kun taas jossain toisessa ohjelmistossa graafisen käyttöliittymän toiminnallisuuksien tarkistaminen voi olla tärkeää. Oikeanlaisen testaustyökalun valinta voi siis helpottaa työtaakkaa työläiden tai vaikeasti testattavien ohjelmiston osien kanssa (Bertolino ja Marchetti 2005).

2.2 Web-sovellusten testaaminen

Web-sovellukset ovat hajautettuja järjestelmiä eli ne koostuvat yleensä monista eri osista, jotka kommunikoivat keskenään. Web-sovellukset voidaan kuvata asiakas-palvelin-mallilla, joka perustuu asiakkaan ja palvelimen väliseen tiedon vaihtamiseen. Ne voidaan myös esittää arkkitehtuurin avulla, missä web-sovellus jaetaan useaan eri tasoon. (Fasolino, Amalfitano ja Tramontana 2013).

Eräs tällainen malli on kolmitasoarkkitehtuuri, jonka Marston (2012) esittää artikkelissaan. Tässä mallissa web-sovellus kuvataan arkkitehtuuriksi, joka rakentuu esitys-, logiikka- ja datatasosta.

- **Esitystasolla** ovat kaikki käyttöliittymäelementit, jotka näkyvät käyttäjälle.
- **Logiikkatason** muodostavat toiminnot, jotka kontrolloivat kaikkia web-sovelluksen toiminnallisuuksia. Toiminnallisuuksiin voi kuulua esimerkiksi tiedon validointi. Tämän tason toiminnot kytkeytyvät vahvasti sekä esitystasolta, että datatasolta saatuun informaatioon.
- **Datatasolla** ovat kaikki operaatiot, jotka tekevät kyselyitä web-sovelluksen tietokan-

taan sopivan rajapinnan kautta.

Arkkitehtuurin lisäksi web-sovelluksen toimintaan vaikuttavat moni sen kehitystyön aikana tehty valinta. Laitteisto, käyttöjärjestelmä, palvelimet, ohjelmointiympäristöt ja web-sovelluksen luontiin käytetyt ohjelmointikielet vaikuttavat kukin osaltaan lopputulokseen. Nämä valinnat mahdollistavat monimuotoisen, responsiivisen ja dynaamisen web-sovelluksen luonnin (Fasolino, Amalfitano ja Tramontana 2013). Näiden ominaisuuksien takia myös web-sovellusten testaaminen on haastavaa.

Web-sovelluksia kehitetään usein nopealla aikataululla ja koska ne esiintyvät nykyään kaikkialla, niiltä odotetaan korkeaa laatua (Leotta, Clerissi, Ricca ja Tonella 2013). Web-sovellusten nopean kehitystyön takia on usein vaikeaa testata ohjelmistoa sen kehitysvaiheessa. Tämä on yksi syy siihen, että web-sovellusten testaamista on pyritty tehostamaan monilla eri menetelmillä (Leotta, Clerissi, Ricca ja Tonella 2013).

Testaaminen on yksi yleisimmin käytetyistä tekniikoista tutkia web-sovelluksen laadukkuutta. Laadukkuuteen liittyy web-sovelluksissa olennaisesti niiden turvallisuus, tehokkuus, käyttövarmuus ja helppokäyttöisyys (Fasolino, Amalfitano ja Tramontana 2013). Laatu voidaankin määritellä ohjelmistolle asetettujen vaatimusten noudattamiseksi (Wu 2012).

Web-sovelluksia voidaan testata automaattisesti tai manuaalisesti riippuen paljolti testattavasta ohjelmistosta. Jos ohjelmistoa aktiivisesti kehitetään, on mahdollista, että on halvempi ratkaisu tehdä testaaminen manuaalisesti. Tosin Montvelisky ja Bhamare (2015) esittävät katsauksessaan, että jopa 86 prosenttia heidän tekemäänsä kyselyyn vastanneista yrityksistä hyödyntää automaattista testausta ainakin jossain määrin. Vaikka niin suuri osa yrityksistä käyttää automaattista testausta, nämä testit kattavat kuitenkin vain osan kaikista tehdyistä testeistä. Vain noin neljännesosalla yrityksistä automaattiset testit kattavat yli puolet kaikista tehdyistä testeistä (Montvelisky ja Bhamare 2015).

2.2.1 Funktionaalinen testaus

Funktionaalinen testaus on ohjelmistolle asetettuihin vaatimuksiin perustuvaa testaamista. Funktionaaliseen testaamiseen on esitetty useita eri lähestymistapoja:

- **Lasilaatikkotestauksessa** keskitytään lähdekoodin arviointiin ja ohjelmiston sisäisten ominaisuuksien testaamiseen (Bertolino ja Marchetti 2005).
- **Mustalaatikkotestaus** on melkolailla vastakohta lasilaatikkotestaukselle. Siinä ohjelmiston lähdekoodilla ei ole merkitystä, vaan testaaminen tehdään arvioimalla ohjelmiston toimintaa samalla, kun sitä käytetään (Fasolino, Amalfitano ja Tramontana 2013).
- **Regressiotestauksella** tarkoitetaan muuttuneen ohjelmiston testausta. Kun ohjelmakoodiin tulee muutos, sen pitää yhä täyttää sille asetetut vaatimukset ja läpäistä testit. (Bertolino ja Marchetti 2005).

Montvelisky ja Bhamare (2015) esittävät, että 75 % heidän tekemäänsä kyselyyn vastanneista yrityksistä hyödyntää automaattista testausta nimenomaan funktionaalisessa- ja ohjelmakoodin muutoksiin perustuvassa regressiotestauksessa. Funktionaaliset vaatimukset ovatkin tärkeä osa web-sovelluksen testaamista, sillä käyttäjän ja web-sovelluksen välinen vuorovaikutus tapahtuu käyttöliittymäelementtien avulla. On myös väitetty, että funktionaalinen testaaminen on vaihtoehto web-sovelluksen testaamisessa vain silloin, kun myös automaattisia testaustyökaluja käytetään apuna testaamisessa (Leotta ym. 2015).

2.2.2 Automaattinen testaus

Automaattinen testaaminen on nykyään suosittu tapa testata ohjelmistoja. Suosio näkyy esimerkiksi automaattisen testaamisen käytön lisääntymisessä. Erään kyselyihin perustuvan tutkimuksen mukaan vuosien 2015–2016 välisenä aikana automaattisten testitapausten osuus kaikista testitapauksista on kasvanut 28 %:sta 44 %:iin tutkimukseen osallistuneilla yrityksillä (Capgemini ja Sogeti 2015).

Automaattinen testaaminen voidaan määritellä ohjelmistotestaukseen liittyvien toimintojen automatisoinniksi. Näihin toimintoihin kuuluvat testiskriptien ajo ja kehitys, testeihin liittyvien vaatimusten todennus ja automaattiseen testaamiseen suunniteltujen testausohjelmistojen käyttö. (Collins ja Lucena 2012).

Leotta ym. (2015) esittävät, että web-sovelluksen kehityksen ollessa nopeata funktionaalinen testaaminen on vaihtoehto vain, jos testaaminen on suoritettu automaattisten testaus-

työkalujen avulla. Nämä työkalut pystyvät toteuttamaan testit nopeammin kuin ihminen ja ilman valvontaa, minkä seurauksena aikaa sekä rahaa säästyy. Ongelmana automaattisessa testaamisessa on kuitenkin testattavan ohjelman kehityksen vaikutus luotuihin testitapauksiin. Yleensä funktionaaliset testit liittyvät käyttöliittymän elementteihin ja web-sovelluksen kehitysvaiheessa käyttöliittymän osat monesti vaihtavat paikkaa, nimeä tai toiminnallisuutta. Tämä tarkoittaa, että jo luodut automaattiset testit eivät enää välttämättä toimi halutulla tavalla. Testitapauksia tulee siis päivittää, ja tämä voi olla aikaa vievä operaatio, sillä testitapaukset tulee manuaalisesti korjata (Collins ja Lucena 2012).

Web-sovellusten automaattinen testaaminen voidaan jakaa kahteen eri lähestymistapaan. Nämä lähestymistavat ovat nauhoita ja toista -testaaminen ja ohjelmoidut testit (Leotta, Clerissi, Ricca ja Tonella 2013).

Nauhoita ja toista -testaaminen perustuu käyttäjän tekemien toimenpiteiden nauhoittamiseen. Kun käyttäjä navigoi web-sivulle, täyttää lomakkeen tietoja tai esimerkiksi painaa jotain web-sivulla olevaa painiketta nauhoituksen aikana, nämä toimenpiteet tallentuvat skriptitiedostoon. Kun nauhoitus on valmis, skriptiä voidaan jälkeenpäin ajaa ilman, että testaajan tarvitsee itse olla aktiivisesti mukana testaustilanteessa.

Nauhoita ja toista -testaamisessa testitapausten luonti perustuu painikkeiden painamisen ja lomakkeiden täyttämisen lisäksi vaatimusten asettamiseen. Väittämä (engl. assertion) on suoritettava toimenpide, joka tarkistaa, että sovellus on siinä tilassa, missä pitääkin (Selenium 2016). Väittämillä voidaan esimerkiksi tarkistaa, että painettaessa tiettyä linkkiä tai painiketta siirrytään siihen URL-osoitteeseen, mihin pitääkin.

Testitapausten rakentaminen on nauhoita ja toista -työkaluilla helppoa. Koska testitapausten rakentaminen ei välttämättä vaadi minkään ohjelmointikielen taitamista, melkein kuka vain pysty rakentamaan testejä nauhoittamalla niitä (Leotta, Clerissi, Ricca ja Tonella 2013). Ongelmana nauhoita ja toista -testaamisessa on kuitenkin testitapausten sisältämät syötteet. Pienikin muutos testattavan ohjelman lähdekoodissa voi aiheuttaa monia muutoksia käyttöliittymään ja edelleen johtaa testitapausten rikkoutumiseen. (Leotta, Clerissi, Ricca ja Tonella 2013). Nauhoitetut testit ovat kuitenkin erinomainen tapa varmistaa, että tietyillä syötteillä päästään aina samaan lopputulokseen (Bruns, Kornstadt ja Wichmann 2009).

Vaihtoehtona nauhoitetuille testeille ovat ohjelmoidut testit, jotka vievät enemmän aikaa luoda, mutta testattavana olevan ohjelmiston muuttuessa ohjelmoidut testit on helpompi pitää ajan tasalla (Bruns, Kornstadt ja Wichmann 2009). Nämä testit luodaan manuaalisesti, mutta valmiita testiskriptejä pystytään ajamaan automaattisesti, aivan kuten nauhoitettujakin testejä. Ohjelmoidut testit luodaan jollain valitulla ohjelmointikielellä käyttäen apuna kirjastoja, joilla pystytään kontrolloimaan selaimen toimintaa (Leotta, Clerissi, Ricca ja Tonella 2013). Kirjastot sisältävät muun muassa komentoja, joilla voidaan tehdä samoja toimintoja kuin selaimessa. Painikkeiden painallukset, lomakkeiden täyttäminen ja muu sivulla navigointi muutetaan siis ohjelmakoodiksi. Tähän ohjelmakoodiin lisätään vielä testitapauksen kannalta olennaiset väittämät, jolloin testitapaus tulee valmiiksi.

Kun web-sovellusta testataan automaattisten testien avulla, haasteena on testitapausten rikkoutuminen ohjelmakoodin muuttuessa. Riippuen ohjelmakoodin muutoksen laadusta testitapaus joudutaan korjaamaan joko web-sovelluksen loogisen tai rakenteellisen muutoksen takia (Leotta, Clerissi, Ricca ja Tonella 2013). Looginen muutos voi olla isompi muutos web-sovelluksen ohjelmakoodissa, joka saattaa rikkoa useitakin testitapauksia. Rakenteellinen muutos on puolestaan usein pienempi, esimerkiksi sivun ulkoasun tai käyttöliittymäelementin nimen uudistus.

Suunnitellessa automaattisia testejä on hyvä arvioida nauhoita ja toista -testien ja ohjelmoitujen testien eroja. Nauhoita ja toista -testit eivät voi täysin korvata ohjelmoituja testejä, sillä ohjelmoimalla pystytään tekemään monipuolisempia testejä. Tosin kustannukset ohjelmoitujen ja nauhoitettujen testien välillä ovat erilaiset. Leotta, Clerissi, Ricca ja Tonella (2013) tutkivat kustannuksia näiden testien välillä ja päätyivät siihen tulokseen, että mitä enemmän versioita web-sovelluksesta julkaistaan, sitä halvemmaksi tulee käyttää ohjelmoituja testejä nauhoitettujen sijaan. Tutkimuksessa tarkasteltiin testien kehittämisen kustannuksia kuudessa eri web-sovelluksessa. Johtopäätöksenä testitapausten tekeminen on nauhoittamalla nopeampaa, mutta testien ajan tasalla pitäminen on työläämpää kuin ohjelmoiduissa testeissä.

3 Selenium

Tässä luvussa käydään läpi Selenium IDE:n ja WebDriver-rajapinnan toimintaa. Luvussa tarkastellaan näiden työkalujen toimintaperiaatteita ja kuinka testitapauksia luodaan kummankin työkalun avulla.

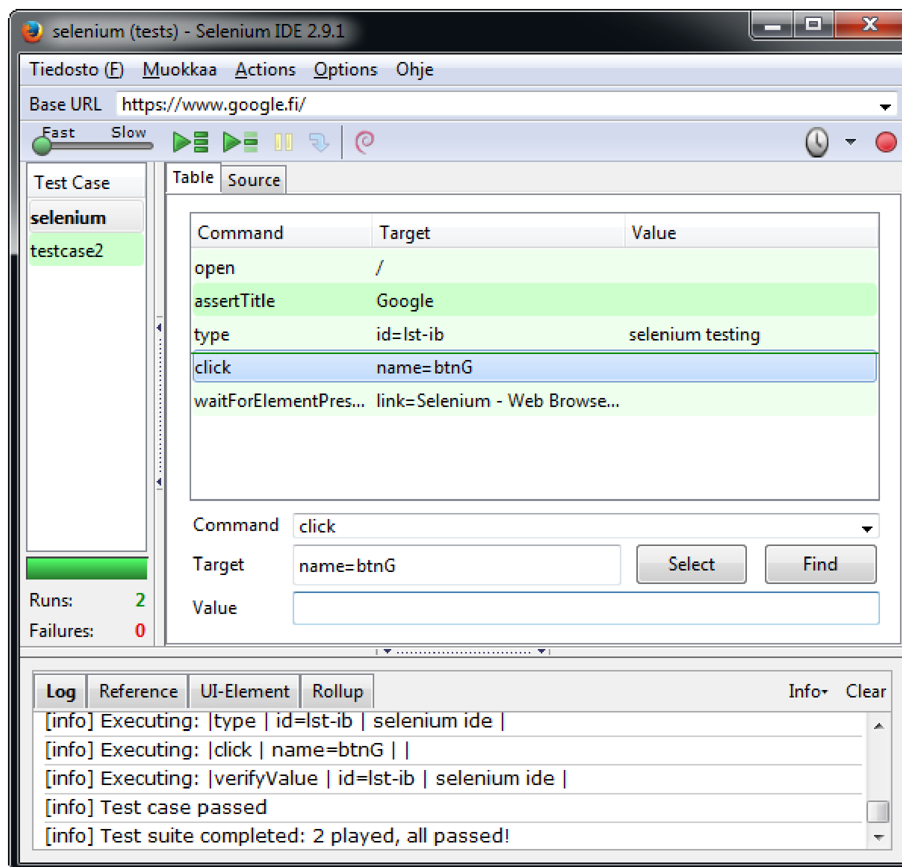
3.1 Selenium IDE

Selenium IDE on Firefoxin lisäosana toimiva nauhoita ja toista -työkalu automaattiseen testaamiseen. Selenium IDE on nimensä mukaisesti ohjelmointiympäristö, jossa on graafinen käyttöliittymä ja jolla on mahdollista nauhoittaa, muokata ja debugata testitapauksia.

Selenium IDE-ohjelmointiympäristöllä voidaan luoda nauhoittamalla testitapauksia, joista voitaan koota isompia, useiden testitapausten testijoukkoja (engl. *test suite*). Testejä voidaan siis ajaa yksittäin tai usean testin sarjana ja Selenium IDE mahdollistaa testejä ajettaessa muun muassa suoritusnopeuden muokkaamisen sekä aloitus- ja pysäytyspisteiden käytön.

3.1.1 Käyttöliittymä

Selenium IDE:n käyttöliittymä on Firefox-selaimen päälle avattava ikkuna, joka sisältää monia toimintoja web-sivun UI-elementtien löytämiseen ja testitapausten luomiseen ja ajamiseen. Kuviossa 1 on esitettynä työkalun käyttöliittymä, jossa on esillä yksinkertainen testitapaus.



Kuvio 1. Selenium IDE -käyttöliittymä

Käyttöliittymän yläosan päävalikosta käyttäjän on mahdollista tallentaa tai luoda uusia testitapauksia ja esimerkiksi muuttaa asetuksia, kuten tuettuja tallennusformaatteja ja liitännäisiä. Selenium IDE tallentaa testitapaukset oletuksena HTML-tiedostoiksi, mutta ne on mahdollista tallentaa muun muassa Java-, Python-, C#- ja Ruby-tiedostoiksi (Selenium 2016). Näin samoja testitapauksia on helppo hyödyntää myös muissa ohjelmointiympäristöissä ja erityisesti WebDriver-rajapinnan testeissä.

Päävalikon alla ovat painikkeet, joilla testitapausten luominen ja ajaminen käynnistetään ja jotka ovat keskeisessä roolissa Selenium IDE:n käytössä. Vasemmassa reunassa ovat kaikki yhdessä testijoukossa olevat testitapaukset ja valittuna olevan testitapauksen tiedot näkyvät käyttöliittymän keskellä.

Käyttöliittymän alaosassa sijaitsee loki, joka kirjaa testin aikana tapahtuvat toimenpiteet sekä mahdolliset virhetilanteet, joihin testaaminen saattaa pysähtyä. Virhetilanteista ilmoite-

taan myös värein: Kaikki onnistuneet testitapaukset näkyvät testijoukossa vihreällä värillä ja virheen sattuessa testitapaus näkyy punaisena.

3.1.2 Selenium-komennot

Selenium IDE:ssä testitapaus rakentuu komennoista (engl. *command*), kohteista (engl. *target*) ja niille asetettavista arvoista (engl. *value*). Komentoja kutsutaan usein *selenese*-kieleksi ja näistä komennoista rakentuva sarja muodostaa yhden testitapauksen (Selenium 2016).

Selenium-komennolla on yleensä parametrinä kohde ja mahdollisesti myös kohteelle asetettava arvo. Komennon kohteena on pääasiassa web-sovelluksessa esiintyvän käyttöliittymäelementin tunniste, esimerkiksi sen ID-arvo tai nimi. Selenium (2016) luokittelee Selenium-komennot kolmeen eri ryhmään:

- **Toimenpiteet** (engl. *Action*) ovat komentoja, jotka muokkaavat ohjelman tilaa. Esimerkiksi painikkeiden painaminen ja navigointi web-sovelluksessa lasketaan toimenpiteiksi.
- **Väittämät** (engl. *Assertion*) ovat tärkeä osa testitapausta, sillä niiden avulla määritetään testin onnistuneisuus. Käytännössä väittämien avulla vahvistetaan, että web-sovellus toimii, kuten pitääkin. Väittämät voivat tarkastaa esimerkiksi tietyn käyttöliittymäelementin olemassaolon, sijainnin tai sen sisältämän tiedon oikeellisuuden.
- **Käsittely** (engl. *Accessor*) on web-sovelluksen tilan tarkastelua ja muuttujien tallentamista myöhempää käyttöä varten. Muuttujat ovat hyödyllisiä erityisesti silloin, kun tiettyä muuttujaa tarvitaan testitapauksen tai -joukon aikana useita kertoja.

Väittämät ovat testitapauksessa ehkä olennaisin asia ja Selenium IDE sisältää kolme erilaista väittämätyyppiä: ”*assert*”, ”*verify*” ja ”*waitFor*”. Kun *assert*-komento epäonnistuu, testitapauksen suoritus pysähtyy välittömästi. Jos halutaan, että virhetilanne ei pysäytä testitapauksen etenemistä, on mahdollista käyttää *verify*-komentoa, joka listaa tapahtuneen virheen pysäyttämättä kuitenkaan testitapauksen suorittamista. Kolmas vaihtoehto, joka on hyödyllinen erityisesti Ajax-sovellusten testaamiseen, on *waitFor*. Näillä komennoilla on joku tietty ehto, jonka täyttymistä ne odottavat. Ehtona voi esimerkiksi olla tietyn käyttöliittymäelementin ilmestyminen sivulle. Jos ehto ei täyty aikakatkaisuun mennessä, testitapauksen suoritus

pysäytetään. (Selenium 2016).

3.1.3 Testitapauksen luonti

Testitapauksen voi Selenium IDE:ssä luoda joko käyttämällä ohjelmointiympäristön tarjoamaa nauhoitusta tai kirjoittamalla testin komennot *selenese*-kielen avulla manuaalisesti. Jos testitapauksen haluaa kirjoittaa itse, ovat apuna kuitenkin kuviossa 1 esiintyvät *Select*- ja *Find*-painikkeet. *Select*-painikkeella käyttäjän on mahdollista valita komennolle kohde suoraan selaimesta klikkaamalla käyttöliittymäelementtiä. *Find*-painike on puolestaan hyödyllinen silloin, kun halutaan paikantaa kohde selaimesta. Painiketta painettaessa käyttöliittymäelementti korostuu selainikkunassa, mikäli se on olemassa ja sillä hetkellä näkyvissä.

Nauhoitus alkaa, kun käyttäjä painaa punaista nauhoituspainiketta. Nauhoituksen aikana Selenium IDE tallentaa kaikki seuraavat toimenpiteet komennoiksi:

- painikkeen tai muun hyperlinkin painaminen
- tekstikenttään kirjoittaminen
- pudotusvalikon tai muun valintanapin painaminen
- käyttöliittymäelementtin klikkaaminen oikealla hiirenpainikkeella, jolloin testitapaukseen voidaan lisätä väittämiä menun kautta.

Kun nauhoitus lopetetaan, tallennetut toimenpiteet näkyvät käyttäjälle, kuten kuvion 1 esittämässä testitapauksessa. Tallennettuja toimenpiteitä voi nyt muokata, kunnes tuloksena on yksi valmis testitapaus.

3.2 Selenium WebDriver

Selenium WebDriver on ohjelmointirajapinta, joka kontrolloi automaattisissa testeissä selaimen eri toimintoja. WebDriver-rajapinnan testit luodaan manuaalisesti ohjelmointikielillä, kuten Javalla, Rubylla, Pythonilla ja C#:lla, mutta tässä luvussa esimerkit ovat Pythonilla kirjoitettuja. Toisin kuin Selenium IDE, WebDriver-rajapinta tukee useimpia eniten käytettyjä selaimia. Rajapinta tukee myös Android- ja iOS-mobiiliselaimia sekä HtmlUnit-selainta, joka on selain ilman graafista käyttöliittymää. (Selenium 2016).

3.2.1 Alustus

Yksinkertainen Selenium WebDriver-rajapinnan ja Python-kielen avulla tehty testitapaus voisi alkaa seuraavasti:

```
from selenium import webdriver

driver = webdriver.Firefox()
driver.get("https://www.google.fi/")
```

Aluksi otetaan käyttöön WebDriver-rajapinta ja sen jälkeen määritetään käytettävä selain. Tässä esimerkissä on valittu Firefox selaimeksi, minkä jälkeen *get*-komento vie käyttäjän Googlen etusivulle. Web-sivun lataaminen onkin WebDriver-rajapinnan yksi keskeisimmistä vaatimuksista.

3.2.2 Käyttöliittymäelementtien paikantaminen

Selenium Webdriver-rajapinta tarjoaa paljon komentoja ja operaatioita selaimessa navigoimiseen ja elementtien löytämiseen. Kun web-sovellusta testataan, käyttöliittymäelementit ovat siinä keskeisessä roolissa. WebDriver-rajapinnan avulla on mahdollista paikantaa käyttöliittymäelementtejä monella eri tavalla:

- ID- ja Class-attribuutit
- HTML-tagit
- Nimi-attribuutti
- Hyperlinkin teksti
- CSS-tyyli
- XPath-polku
- Javascript-kielellä tai jollain muulla ohjelmointikielellä kirjoitettu käsky. (Selenium 2016).

Kaikki edellä mainitut menetelmät perustuvat käyttöliittymäelementtien paikantamiseen *document object* -mallista (DOM). Malli on puurakenne, joka kuvaa kaikki web-sovelluksen elementit hierarkkisena puurakenteena (Leotta ym. 2015).

Selenium 2016 esittää, että tehokkain tapa käyttöliittymäelementin löytämiseen on ID-attribuutti. DOM-puu ei kuitenkaan aina sisällä ID-attribuuttia, joten joskus voi olla tarpeellista käyttää Class- tai nimi-attribuuttia tai HTML-tagia. Nämä eivät ole usein kuitenkaan uniikkeja, vaan DOM saattaa sisältää monia elementtejä, joissa on sama attribuutin arvo tai HTML-tagi. Yksi tapa valita tietty DOM-elementti ilman attribuuttien käyttämistä on käyttää XPath-kieltä, joka hakee elementin DOM-dokumentin puurakenteesta Xpath-polun avulla (Selenium 2016).

Käyttöliittymäelementtien paikantamiseen käytettävistä menetelmistä on tehty myös tutkimusta. Leotta, Clerissi, Ricca ja Spadaro (2013) tutkivat testien korjaamiseen kuluvaa aikaa, kun testitapauksissa on käytetty eri tapoja DOM-elementtien paikantamiseen. Parhaaksi menetelmäksi paikantamiseen osoittautui olevan ID-attribuutin ja hyperlinkin tekstin yhtäaikainen käyttäminen. XPath-kieltä käyttävät testitapaukset ovat tutkimuksen mukaan yli neljä kertaa hitaampia korjata kuin ID-attribuutin avulla tehdyt testitapaukset. XPath-kieli voi olla silti tarpeellinen tilanteessa, jossa käyttöliittymäelementin tarkalla sijainnilla suhteessa muihin elementteihin on merkitystä (Selenium 2016).

Selenium WebDriver-rajapinnassa paikantaminen tapahtuu esimerkiksi alla olevan Python-kielisen koodin avulla:

```
user = driver.find_element(By.ID, "username")
```

3.2.3 Testaaminen

Tässä luvussa tarkastellaan WebDriver-rajapinnan avulla testaamista, tuodaan esille joitain WebDriver-rajapinnan etuja verrattuna IDE-ohjelmointiympäristöön sekä kerrotaan perusperiaatteita testitapausten luontiin. Luvussa tarkastellaan myös, kuinka Python-ohjelmointikielellä luodaan yksinkertaisia testitapauksia.

Selenium WebDriver-rajapinnan testeissä tärkeimmät toiminnot ovat sivulla navigoiminen, käyttöliittymäelementtien valinta ja muokkaaminen sekä väittämien toteutus. Nämä kaikki toiminnot ovat helppoja tehdä niin Selenium IDE-ohjelmointiympäristön kuin WebDriver-rajapinnan avulla, mutta WebDriver antaa enemmän mahdollisuuksia dynaamisten web-sovellusten testaamiseen.

Ajax on yksi teknologia, jossa WebDriver-rajapinta on erityisen hyödyllinen. Ajax-pyyntö lähetetään palvelimelle yleensä jonkun käyttäjän tekemän toiminnon yhteydessä, jolloin websovellus päivittyy ilman, että sitä ladataan uudestaan. Koska sivun päivittyminen ei usein tapahdu välittömästi ajax-pyynnön jälkeen, Selenium-työkalulla joudutaan käyttämään luvussa 3.1.2 mainittuja *waitFor*-tyylisiä väittämiä. WebDriver-rajapintaa käyttämällä Ajax-pyyntöjen tuomat muutokset voidaan tarkistaa muun muassa implisiittisten ja eksplisiittisten odotuksien avulla (Selenium 2016).

Eksplisiittiset odotukset määräytyvät jonkun tietyn ehdon täyttymisen mukaan (Selenium 2016). Ehtona voi olla jonkun käyttöliittymäelementin löytyminen, silmukka tai esimerkiksi pelkkä aikarajoite, joka hidastaa testitapauksen etenemistä täsmälleen tietyn ajan verran. Implisiittinen odottaminen määräytyy puolestaan asettamalla kaikille testitapauksen operaatioille aika, jonka sisällä käyttöliittymäelementin tulisi viimeistään löytyä (Selenium 2016).

Eräs toinen ominaisuus, jossa WebDriver-rajapinta on IDE-ohjelmointiympäristöön verrattuna monipuolisempi, on aineisto-ohjattu testaaminen (engl. *Data-driven testing*). Toiselta nimeltään parametrisoidut testit, aineisto-ohjatut testit ovat testitapauksia, jotka ajetaan useaan kertaan eri syötteillä (Leotta, Clerissi, Ricca ja Tonella 2013). Näiden testien data voidaan saada esimerkiksi teksti- tai CSV-tiedostosta tai tietokannasta (Selenium 2016).

Listaus 3.1. Python-testitapaus

```
from selenium import webdriver
from selenium.webdriver.common.by import By
from selenium.common.exceptions import NoSuchElementException
import unittest

class Example(unittest.TestCase):
    def setUp(self):
        self.driver = webdriver.Firefox()
        self.driver.implicitly_wait(15)
        self.base_url = "https://www.google.fi/"
        self.verificationErrors = []

    def test_example(self):
        driver = self.driver
```

```python
        driver.get(self.base_url + "/")
        self.assertEqual("Google", driver.title)
        driver.find_element_by_id("lst-ib").clear()
        driver.find_element_by_id("lst-ib").send_keys("selenium testing")
        driver.find_element_by_name("btnG").click()
        tmp = "Selenium - Web Browser Automation"
        self.assertTrue(self.is_element_present(By.LINK_TEXT, tmp))

    def is_element_present(self, how, what):
        try: self.driver.find_element(by=how, value=what)
        except NoSuchElementException as e: return False
        return True

    def tearDown(self):
        self.driver.quit()
        self.assertEqual([], self.verificationErrors)
```

Yllä on esitettynä yksinkertainen *unittest*-kehyksellä luotu testitapaus, joka voidaan käynnistää komennolla *unittest.main()*. Testitapaus vastaa pitkälti kuviossa 1 näkyvillä olevaa testitapausta, joka on Selenium IDE:n kautta tallennettu Python-ohjelmakoodiksi. *unittest*-kehys ohjaa testitapauksen suoritusta siten, että *setUp*-funktio ajetaan ennen varsinaisen testitapauksen ajamista ja *tearDown* on viimeinen toimenpide, joka ajetaan testin suorituksen jälkeen.

Ohjelmakoodi tekee seuraavat toimenpiteet:

1. valitsee Firefox-rajapinnan, asettaa 15 sekunnin implisiittisen odotuksen käyttöliittymäelementtien löytämiseen ja valitsee Googlen etusivun testitapauksen pääsivuksi
2. menee Googlen etusivulle ja *assertEqual*-väittämällä varmistaa, että siellä ollaan
3. etsii hakukentän, tyhjentää sen tekstistä ja kirjoittaa hakuehdon
4. etsii hakupainikkeen ja painaa sitä
5. etsii, että sivulta löytyy linkki, jonka otsikkona on *Selenium - Web Browser Automation*
6. odottaa maksimissaan 15 sekuntia linkin löytymistä ja jos sitä ei löydy, testitapaus epäonnistuu ja käyttäjälle tulostetaan virheiden määrä ja laatu.

Selenium IDE:n avulla luotujen testien muuttaminen suoraan WebDriver-rajapintaa käyttä-

viksi testeiksi on vain yksi vaihtoehto. Yllä oleva testitapaus voidaan helposti tiivistää kuvion 1 mukaiseksi ohjelmakoodiksi, mutta testitapausten huollon kannalta tiivistäminen ei välttämättä ole kannattavaa.

Leotta ym. (2015) esittävät tutkimuksessaan sivuobjekti-suunnittelumallin (engl. *page object pattern*), jonka tavoitteena on vähentää testien ylläpitoon liittyviä kustannuksia sekä vähentää ohjelmakoodin toisteisuutta. Mallia käytetään niin, että web-sovelluksen sivuja käsitellään ohjelmakoodissa luokkina, joiden sisään laitetaan kaikki yhtä sivua koskevat toiminnallisuudet. Nämä toiminnallisuudet ovat luokan sisällä metodeja, joita testitapauksissa kutsutaan. (Leotta, Clerissi, Ricca ja Spadaro 2013). Etuna tässä menetelmässä on erityisesti se, että testejä ja web-sovelluksen toiminnallisuuksia käsittelevä ohjelmakoodi pystytään pitämään erossa toisistaan. Kun web-sovelluksen käyttöliittymä muuttuu, testitapauksia ei välttämättä tarvitse muokata, vaan riittää, että sivuobjekteja päivitetään. (Selenium 2016).

Jos listauksessa 3.1 oleva testitapaus halutaan toteuttaa sivuobjekti-suunnittelumallilla, täytyy testitapaus jakaa useammaksi luokaksi. Alla oleva ohjelmakoodi kuvaa testitapauksen yksinkertaistettuna.

Listaus 3.2. Sivuobjekti-suunnittelumallia käyttävä testitapaus

```
driver = webdriver.Firefox()
driver.get("https://www.google.fi/")
homePage = GooglePage(driver)
searchPage = homePage.search("selenium testing")
tmp = "Selenium - Web Browser Automation"
assertTrue(searchPage.is_element_present(By.LINK_TEXT, tmp))
```

Mallin avulla luodaan *GooglePage*-luokka, joka sisältää kaikki Googlen etusivun toiminnallisuudet. Tämän sivuobjektin *search*-funktio luo *searchPage*-instanssin luokasta, jossa on kaikki hakutulossivulla esiintyvät toiminnallisuudet. Nyt kaikki web-sivun toiminnallisuudet sijaitsevat luokkien sisällä ja testitapaus hyödyntää luokkien tarjoamia funktioita.

Testaamista voi helpottaa sivuobjekti-suunnittelumallin ohella myös ohjelmakoodin toisteisuutta vähentämällä. Tällä tarkoitetaan WebDriver-testien suunnittelussa sitä, että usein käytetyistä Selenium-kutsuista muodostetaan apufunktioita, joilla nopeutetaan testien kirjoittamista ja vähennetään usein toistuvan ohjelmakoodin määrää (Selenium 2016). Listauksessa

3.2 esiintyvällä funktiolla *search* pyritään vähentämään toisteisuutta. Hakeminen on Googlen hakukonetta testatessa todennäköisesti sen olennaisin toiminto ja osa montaa testitapausta. Tämä funktio aluksi etsii hakukentän, täyttää sen parametrina annetulla arvolla ja sen jälkeen painaa hakupainiketta.

4 Yhteenveto

Automaattinen testaaminen on nykyään tärkeä ja melko yleinen osa monen yrityksen ohjelmistotestausta. Sen avulla ei voida välttämättä testata kaikkia ohjelmiston osia eikä se voi täysin korvata manuaalista testaamista, mutta sen avulla pystytään kuitenkin usein vähentämään testien tekemiseen, suorittamiseen ja huoltoon kuluvaa aikaa.

Web-sovelluksien testaaminen on yksi kohdealue, jossa automaattinen testaaminen on paljon esillä. Koska web-sovelluksia on helppo testata esimerkiksi nauhoitettujen testien avulla, automaattinen testaus on helppo ottaa osaksi ohjelmistotestausta. Web-sovellukset rakentuvat käyttöliittymäelementeistä, joiden toimintaa on helppo arvioida automaattisten testitapausten avulla. Kuitenkaan funktionaaliset, tehokkuuteen tai web-sovelluksen ulkoasuun liittyvät vaatimukset eivät ole ainoita tärkeitä asioita web-sovellusten testaamisessa, vaan esimerkiksi turvallisuus nähdään yhtä lailla merkittäväksi osaksi web-sovelluksen testausta.

Haasteena automaattisessa testaamisessa on testien ajan tasalla pitäminen. Sekä ohjelmoidut, että nauhoita ja toista -tyyliset automaattiset testit ovat kumpikin hyviä regressiotestauksen yhteydessä. Tosin kun testattavana oleva ohjelmisto muuttuu, osa testeistä tulee rikkoutumaan, mikä aiheuttaa työtä varsinkin automaattisten testien korjaamisessa. Automaattisten ja manuaalisesti luotujen testien välille tulisi siis löytää jonkinlainen tasapaino. Tietyt testit kannattaa siis tehdä automaattisilla testaustyökaluilla, kun taas toiset voivat olla järkevämpi luoda manuaalisesti. Tätä aihetta on hyvä tutkia lisää.

Tutkielmassa käytiin läpi automaattista testausta Selenium-työkalun avulla. Sekä Selenium IDE, että WebDriver ovat melko helppokäyttöisiä komponentteja, joilla kummallakin voi luoda kattavia testijoukkoja erilaisille web-sovelluksille. Selenium IDE loistaa erityisesti helppokäyttöisyydellään. Ohjelmointiympäristön *selenese*-kieli muodostaa suuren valikoiman komentoja, joista on helppo koota testitapauksia. WebDriver-rajapinta tarjoaa puolestaan enemmän mahdollisuuksia testitapausten luontiin, mutta vaatii myös testaajalta enemmän osaamista ohjelmoinnista.

Selenium on silti vain yksi monista web-sovellusten automaattiseen testaamiseen tarkoitetuista työkaluista. Markkinoilla on suuri määrä erilaisia automaattiseen testaamiseen tarkoi-

tettuja työkaluja ja vaikka Selenium on suosittu, vertailuja eri testausohjelmistojen välillä olisi hyvä tehdä enemmän, jotta voitaisiin valita sopiva testausohjelmisto sen tarjoamien ominaisuuksien perusteella.

Lähteet

Altaf, I., J.A. Dar, F.U. Rashid ja M. Rafiq. 2015. “Survey on selenium tool in software testing”. Teoksessa *2015 International Conference on Green Computing and Internet of Things (ICGCIoT),* 1378–1383.

Bertolino, A., ja E. Marchetti. 2005. “A Brief Essay on Software Testing”. Teoksessa *Software Engineering: The Development Process,* 393–411.

Bruns, A., A. Kornstadt ja D. Wichmann. 2009. “Web Application Tests with Selenium”. *Software, IEEE* 26 (5): 88–91.

Capgemini ja Sogeti. 2015. *World Quality Report 2015–2016.*

Collins, Eliane Figueiredo, ja Vicente Ferreira de Lucena Jr. 2012. “Software Test Automation Practices in Agile Development Environment: An Industry Experience Report”. Teoksessa *Proceedings of the 7th International Workshop on Automation of Software Test,* 57–63. AST ’12. Zurich, Switzerland: IEEE Press.

Fasolino, A.R., D. Amalfitano ja P. Tramontana. 2013. “Web application testing in fifteen years of WSE”. Teoksessa *Web Systems Evolution (WSE), 2013 15th IEEE International Symposium on,* 35–38.

Leotta, M., D. Clerissi, F. Ricca ja C. Spadaro. 2013. “Comparing the Maintainability of Selenium WebDriver Test Suites Employing Different Locators: A Case Study”. Teoksessa *Proceedings of the 2013 International Workshop on Joining AcadeMiA and Industry Contributions to Testing Automation,* 53–58. Lugano, Switzerland: ACM.

Leotta, M., D. Clerissi, F. Ricca ja P. Tonella. 2013. “Capture-replay vs. programmable web testing: An empirical assessment during test case evolution”. Teoksessa *Reverse Engineering (WCRE), 2013 20th Working Conference on,* 272–281.

Leotta, M., A. Stocco, F. Ricca ja P. Tonella. 2015. “Automated Generation of Visual Web Tests from DOM-based Web Tests”. Teoksessa *Proceedings of the 30th Annual ACM Symposium on Applied Computing,* 775–782. Salamanca, Spain: ACM.

Marston, T. 2012. *What is the 3-Tier Architecture?* Saatavilla WWW-muodossa, http://www.tonymarston.net/php-mysql/3-tier-architecture.html, viitattu 10.3.2016.

Montvelisky, J., ja L. Bhamare. 2015. *State of Testing.* Saatavilla WWW-muodossa, http://www.practitest.com/wp-content/uploads/2015/07/State_of_Testing_Survey_2015.pdf, viitattu 28.2.2016.

Selenium. 2016. *Selenium – Web Browser Automation.* Saatavilla WWW-muodossa, http://docs.seleniumhq.org/docs/, viitattu 14.2.2016.

Wu, H. 2012. "An effective equivalence partitioning method to design the test case of the WEB application". Teoksessa *2012 International Conference on Systems and Informatics (ICSAI),* 2524–2527.