Haaga-Helia
ammattikorkeakoulu Oy

Testiautomaation parhaat työkalut ja käytännöt

Marko Nenonen

Opinnäytetyö

Tietojenkäsittelyn koulutusohjelma

2017

Haaga-Helia ammattikorkeakoulu Oy

Tiivistelmä

Tekijä(t)

Marko Nenonen

Koulutusohjelma

Tietojenkäsittely

Opinnäytetyön otsikko	**Sivu- ja liitesivumäärä**
Testiautomaation parhaat työkalut ja käytännöt	36 + 8

Tämän opinnäytetyön tavoitteena oli selvittää, onko Trimico Oy:n LogiPlan-tiimin käyttämissä testiautomaatioratkaisuissa tai testiautomaatioon liittyvissä käytännöissä parantamisen varaa. Ajatus aiheesta sekä toimeksiantajasta sai alkunsa keväällä 2016 tämän työn kirjoittajan suorittaessa työharjoittelunsa kyseisessä tiimissä testiautomaation parissa. Opinnäytetyö on toteutettu keväällä 2017.

Opinnäytetyö on jaettu kahteen osaan, tietoperusta jossa selvitetään kirjallisuuslähteitä sekä asiantuntijahaastatteluita hyväksi käyttäen parhaat työkalut ja käytännöt, sekä itse tutkimusosa, jossa verrataan tietoperustan tuloksia Trimicolla käytössä oleviin. Tutkimuksessa tehdyt haastattelut on suoritettu sähköpostitse.

Tietoperustassa on ensin esitelty hyödyllisiä käytäntöjä liittyen testien käyttöönottoon, ylläpitoon sekä niiden ajamiseen. Tämän jälkeen on verrattu kahta suurinta automaatio-frameworkia käyttöliittymän testaamiseen, Seleniumia sekä Robotia. Tämän jälkeen on vielä esitelty sekä vertailtu kahta suurinta jatkuvan integraation palvelua, Jenkinsiä sekä Travis CI:ta.

Tutkimuksen tuloksina löydettiin muutamia kehityskohteita. Suurimpana Selenium-testien todella pitkä ajoaika, joka on korjattavissa teknisillä muutoksilla. Myös LogiPlan-tiimin sisäisistä käytännöistä löytyi parantamisen varaa. Testejä ei olla osattu vaatia kaikkien uusien ominaisuuksien yhteydessä, testejä ei olla korjattu välittömästi, kun hajoaminen on huomattu. Pääosin kuitenkin Trimicolla on testiautomaatio hyvällä mallilla.

Asiasanat

Testiautomaatio, Testaus, Selenium, Robot, Jenkins, Travis CI

Sisällys

1 Johdanto

Jokainen ohjelmistokehitysryhmä testaa tuotteitaan, mutta tuotteeseen jää siitä huolimatta aina ohjelmointivirheitä. Manuaaliset testaajat pykivät löytämään ne ennen kuin tuote julkaistaan, mutta useimmiten ainakin osa ohjelmistovirheistä päätyy tuotantoon asti. Pahimmassa tapauksessa ohjelmointivirheet uusiutuvat, vaikka ne olisivat jo kerran löydetty ja korjattu. Testiautomaatio on paras tapa lisätä ohjelmistotestauksen tehokkuutta, tarkkuutta sekä kattavuutta. (Smartbear, 2016.)

Tämän työn toimeksiantajana toimiva Trimico Oy on vuonna 2004 perustettu yritys joka työllistää noin 40 henkeä kolmella paikkakunnalla. Trimico tarjoaa asiakkailleen laadukkaita sekä joustavia ohjelmisto- ja järjestelmäratkaisuja kuljetus- ja logistiikka-alalle. Trimico haluaa pitää tuotteissaan korkeaa tasoa ja näin ollen se on panostanut, sekä haluaa vielä tulevaisuudessakin panostaa tuotteidensa huolelliseen testaamiseen. Tästä syystä Trimico Oy oli kiinnostunut toimimaan tämän opinnäytetyön toimeksiantajana.

Tämän opinnäytetyön tavoitteena on selvittää parhaat mahdolliset työkalut sekä käytännöt ja verrata niitä Trimicolla LogiPlan-kehitystiimissä käytössä oleviin, sekä näiden perusteella löytää keinoja tehostaa heidän testiautomaatiotaan entisestään.

Tämä opinnäytetyö tulee pääosin käsittelemään web-ohjelmiston käyttöliittymän testaamista ja niiden testien automatisointia, sekä testiautomaatioon liittyviä hyödyllisiä käytäntöjä.

Opinnäytetyö tullaan toteuttamaan etsimällä aiheesta ensin mahdollisimman paljon tietoa kirjallisuuslähteistä sekä asiantuntijahaastatteluista. Haastateltavina tässä vaiheessa toimivat VALA Group Oy:ltä laadunvarmistuksen liiketoimintajohtaja Teemu Pesonen, sekä samaisesta yrityksestä testiautomaatioasiantuntija Dragos Guberna. Kirjallisuuslähteinä toimivat pääosin ajankohtaiset web-julkaisut.

Kun tietoperusta on saatu kokoon, tullaan haastattelemaan työn toimeksiantajaa Trimico Oy:n LogiPlan-kehitystiimiä heidän käyttämistään testiautomaatioratkaisuista, joita sitten verrataan tietoperustassa löydettyihin tuloksiin. Tässä vaiheessa tullaan haastattelemaan LogiPlan-tiimissä toimivaa teknistä testaajaa sekä ohjelmistokehittäjää. Näiden haastatteluiden pohjalta tullaan arvioimaan Trimicon testiautomaatioratkaisuja ja etsimään

mahdollisia parannusehdotuksia. Kaikki haastattelut tullaan tekemään sähköpostitse, jotka tulevat työn loppuun luottamuksellisina liitteinä.

2 Testiautomaatio kirjallisuuden ja asiantuntijahaastatteluiden perusteella

Manuaalisessa testauksessa ihminen istuu tietokoneen äärellä ja käy tarkkaan läpi ohjelmiston eri toiminnallisuuksia, antaa erilaisia syötteitä, sekä vertaa niitä odotettuihin tuloksiin. Otetaan esimerkiksi keskikokoinen ohjelmistoprojekti, jossa on noin 500 selaimella ajettavaa testitapausta, näistä jokaisen manuaaliseen suorittamiseen kuluu keskimäärin 5 minuuttia aikaa, näin ollen vie koko ohjelmiston testaus yhden ihmisen suorittamana noin viikon työajan, ja tämäkin vasta käyttäen yhtä selainta. Mikäli testaamiseen vielä käytettäisiin ulkopuolista konsulttia, tulisi yksi testiajo maksamaan noin 2000 euroa. (Eficode, 2016.)

Tavallisen selainpohjaisen testin automatisointiin kuluu maksimissaan kaksi tuntia, eli 500 testitapauksen automatisointiin kuluisi enimmillään noin tuhat tuntia, vaikka testiautomaatioasiantuntija olisi tuplasti manuaalista testaajaa kalliimpi, jäisi automatisoinnin kustannukset edelleen alle sadantuhannen euron, joka vastaa alle kahtakymmentä kokonaista testiajoa käsin. (Eficode, 2016.) Puhumattakaan jatkuvasti säästettävästä ajasta. Automatisoituna käyttäen esimerkiksi Seleniumia (kts. www.seleniumhq.org) testiajo kestäisi noin 2-4 tuntia riippuen testien monimutkaisuudesta.

Testiautomaatiota on jo pitkään pidetty elintärkeänä suurille ohjelmistokehitysorganisaatioille, mutta aina ei ole järkevintä automatisoida kaikkia testitapauksia. On tilanteita, joissa manuaalinen testaaminen on järkevämpää, kuten esimerkiksi, jos sovelluksen käyttöliittymään tulee suuria muutoksia lähitulevaisuudessa, tällöin automatisointi ei ole järkevin tapa, sillä testit jouduttaisiin kuitenkin kirjoittamaan uudestaan muutosten tullessa. Toisinaan ei vain yksinkertaisesti ole aikaa rakentaa testiautomaatiota, jos sovelluksella on hyvin tiukka määräaika ja testiautomaatiota ei ole vielä tarjolla, on manuaalinen testaus luultavasti automaatiota parempi ratkaisu. (Selenium.org, 2017.)

Mihin automaatio sitten soveltuu? Testiautomaation kasvu on tehnyt ohjelmistokehityksestä nopeampaa. Lisäksi, automaatio mahdollistaa regressiokierrosten paremman kontrollin ja mittaamisen. Erityisesti monet avoimen lähdekoodin automaatio-frameworkit sekä työkalut ovat mahdollistaneet aivan uudenlaiset mahdollisuudet yrityksille aloittaa testiautomaatio juuri heidän yritykselleen räätälöidyllä tavalla. (Pesonen, 2016.)

Testiautomaatiosta on ohjelmistokehityksessä selkeitä hyötyjä:

- Nopea palaute kehittäjille
- Automaatiota voidaan pitää koko kehitystyön selkärankana, se mahdollistaa kehittäjille ydintoiminnallisuuksien luomisen ilman pelkoa, että se rikkoo jotain
- Ohjelman kehityssykli on paremmassa kontrollissa
- Pitkässä juoksussa kulujen vähentäminen jatkuvalla automatisoidulla regressiotestauksella

Ennen testiautomaation aloittamista, on kuitenkin hyvä huomioida ainakin seuraavat asiat:

- Alustava investointi on suurempi kuin manuaalisessa testauksessa
- Testiautomaatio nojaa manuaaliseen testaamiseen, automaatio käyttää testitapauksia jotka on määritelty ja niiden ominaisuudet testattu testaustiimin toimesta
- Kaikkea ei voi, eikä kannata automatisoida.
- Automaatio on ohjelmiston tarkistamista varten, se testaa vain mitä olet määritellyt
- Automatisointiin kuluu aikaa
- Automaatioon siirtyminen vaatii yritykseltä suuren muutoksen, yrityksen tulee olla tarpeeksi kypsä tähän

2.1 Parhaat käytännöt

Jotta testiautomaatiosta saataisiin kaikki mahdollinen hyöty irti, on syytä kiinnittää tiettyihin seikkoihin huomiota jo ennen ensimmäisten testien kirjoittamista. Kaikkein tärkein vinkki yrityksille on suunnitella testiautomaatio jo etukäteen hyvin, sekä valita oikeat ihmiset, oikeilla taidoilla vastaamaan siitä. Testiautomaatio on kallis investointi ja se voi mennä pahasti pieleen, mikäli sitä alkavat hoitaa väärät ihmiset joilla ei ole riittävää tietotaitoa asiasta. (Guberna, 2017.) Tiimin on syytä olla myös alusta asti samalla sivulla käytettävistä tekniikoista sekä tiimin yhteisistä käytännöistä liittyen testiautomaatioon.

2.1.1 Testien käyttöönotto

Lähtökohtaisesti automaatio on hyvä ottaa käyttöön sovelluksiin, joihin tehdään jatkuvaa kehitystyötä, eli testauskierroksia tulee paljon. Automaation ROI (Return of investment) on suurimmillaan silloin, kun automatisoidut testit ajetaan useimmiten. Tällöin kustannukset yhtä automatisoitua testitapausta kohden tippuu ja tuottavuus kasvaa pitkässä juoksussa. (Pesonen, 2017.)

Testiautomaatio on helpointa ja tehokkainta ottaa käyttöön jo projektin alkuvaiheessa, aina kun ohjelmistoon luodaan uusia ominaisuuksia, automatisoidaan niille samalla testit. Näin varmistutaan tulevaisuudessa tehtävien muutosten yhteydessä siitä, etteivät ne hajota mitään muita, jo aiemmin ohjelmoituja toiminnallisuuksia.

Projektin alussa saattaa tuntua turhauttavalta käyttää miltei yhtä paljon aikaa testien suunnitteluun ja kirjoittamiseen kuin itse toiminnallisuuden koodaamiseen, mutta se aika maksaa itsensä takaisin useaan otteeseen tulevaisuudessa.

Testiautomaatiota harkitessa kannattaa ensimmäiseksi tutustua saatavilla oleviin työkaluihin ja valita sopivin juuri teidän projektin tarpeisiin. Mikäli projekti on kirjoitettu C sharpilla, ei ole mitään syytä valita sellaista automaatiotyökalua, joka ei tue C sharpia. Uuden ohjelmointikielen opiskeleminen vain testaamista varten vie turhaa aikaa ja se kannattaa välttää ottamalla käyttöön tuttua kieltä tukeva työkalu. Kuten Pesonen haastattelussaan mainitsee, on automaation käyttöönotossa tekniset haasteet huomattavasti yksinkertaisempia kuin organisaation prosessien sekä kulttuurin muuttaminen. On siis hyvä pitää mielessä, mikäli organisaation prosessit sekä kulttuuri eivät tue automaation käyttöönottoa, niin suurella todennäköisyydellä automaatio ei tule toimimaan pitkässä juoksussa.

Kun kyseessä on jo vanhempi ohjelmisto ja se on pääosin ylläpito vaiheessa, on syytä pohtia hieman tarkemmin, onko testien automatisointi kannattavaa. Kuinka suuret kustannukset nykyisestä laadunvarmistuksesta tulee, ja kuinka pitkä elinkaari tuotteella on? Testiautomaatiota käyttöönottaessa on hyvä myös varmistua siitä, että tuotteen jatkokehitystä sekä ylläpitoa tekevät henkilöt ovat motivoituneita oppimaan testiautomaatiota sekä ylläpitämään testitapauksia osana muuta ylläpito- ja kehitystyötä. (Eficode, 2016.)

2.1.2 Testien ylläpito

Kun testiautomaatio on otettu käyttöön, ja testitapauksia on luotu, tulee testejä ylläpitää yhtä lailla kuin ohjelmiston koodiakin. Mikäli koodia muutettaessa huomataan jonkun testin epäonnistuvan, on asiaan syytä tarttua heti tarkistamalla, onko kyseessä oikea ohjelmointivirhe ohjelmistossa vai vaan esimerkiksi jonkin elementin siirtyminen tai uudelleennimeäminen joka rikkoo testin, mutta ei varsinaisesti ole ohjelmointivirhe. Jos rikkoutuneisiin testeihin ei tartuta heti, on niitä tulevaisuudessa aina vaikeampi ja vaikeampi korjata, kun ei tiedetä minkä muutoksen yhteydessä testi on mennyt rikki. Tämä johtaa myös siihen, että testin saattaa korjata joku muu kuin testin rikkonut henkilö, tällöin palaute miksi testi on hajonnut ei kantaudu oikealle henkilölle asti ja samat virheet toistuvat todennäköisesti myös tulevaisuudessa. Mikäli testien annetaan olla rikki pidempään, on vaarana, että rikkoutuneet testit kasaantuvat ja niiden korjaaminen alkaa tuntumaan ylitsepääsemättömältä ja näin ollen kyseenalaistetaan koko testiautomaation järkevyys.

2.1.3 Testien ajaminen

Testit on hyvä ajaa pienimpienkin muutosten jälkeen, kun testit on kerran automatisoitu, tulee niistä ottaa mahdollisimman paljon irti ja varmistua jatkuvasti, ettei ohjelmistoon ole jäänyt ohjelmointivirheitä.

Testiautomaatio antaa meille nopean palautteen tapahtuneesta regressiosta. Jotta nopeaan palautteeseen päästäisiin, testien tulee olla ajettavissa nopeasti, tämä muodostuu usein ongelmaksi testien määrän kasvaessa. Tämä on ongelma varsinkin, kun ollaan tekemisissä käyttöliittymän kanssa, jossa ohjelmiston kanssa kommunikointi on hitaampaa. Hyvä tapa nopeuttaa testien ajamista on laittaa ne ajautumaan rinnakkain, eikä niinkään jonossa. Tämä mahdollistaa ajoaikojen pysymisen alhaisina. (McCarthy, 2016.)

2.2 Web-ohjelmistojen käyttöliittymän automatisoitu testaaminen

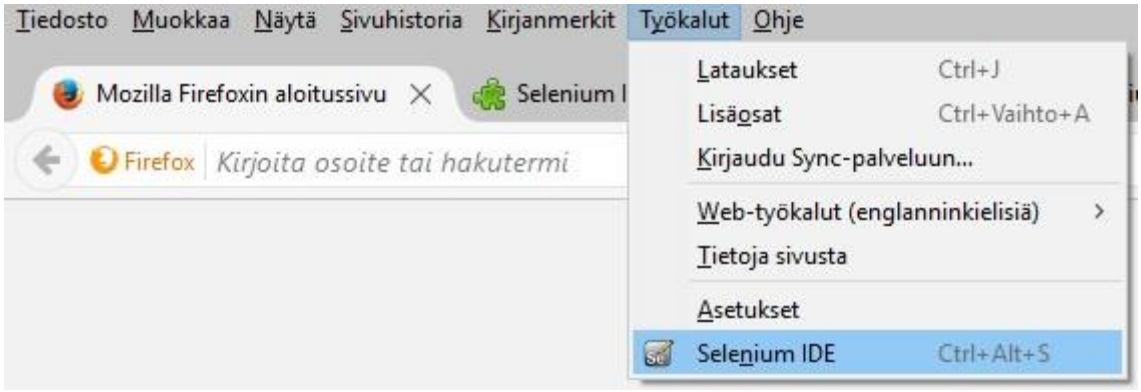
Web-ohjelmistojen käyttöliittymän testaamiseen on olemassa useita valmiita frameworkeja, useille eri ohjelmointikielille. Toiset ovat hyvinkin nopeasti käyttöön otettavissa, toiset taas vaativat suuremman työn toimiakseen. Alla esitelty suosituimmat web-ohjelmistojen käyttöliittymän testauksessa käytetyt frameworkit.

2.2.1 Selenium IDE & WebDriver

Selenium IDE (Integrated Development Environment) on suoraan Firefoxiin asennettava liitännäinen, jota tulisi käyttää sen yksinkertaisuuden takia vain testiprototyyppien luomiseen, eikä niinkään kokonaisratkaisuna monimutkaisten testien luomiseen sekä ylläpitoon. (Guru99, 2015.)

Selenium IDEn vahvuudet ovatkin sen nopea asentaminen, yksinkertainen käytettävyys ja ulkoasu sekä testien nopea ajaminen. Huonoihin puoliin kuuluu mm. että se tukee vain Firefoxia. Odottaminen, Selenium IDE koittaa toimia niin nopeasti, että testattava sivu ei välillä pysy perässä ja testi kaatuu siihen. Luotettavuus, IDEllä luodut vähänkin monimutkaisemmat testit epäonnistuvat jopa 20% ajasta, vaikka ne olisi kirjoitettu oikein. (Yarn, 2016.)

Selenium IDEn asentaminen on todella yksinkertaista, tarvitsee vain mennä osoitteeseen www.seleniumhq.org/downloads, josta löytyy linkki joka asentaa liitännäisen suoraan Firefoxiin. Tämän jälkeen Firefox tulee käynnistää uudelleen ja Selenium IDEn pitäisi löytyä työkalut-valikosta (Kuva 1).



Kuva 1. Selenium IDE löytyy työkalut-valikosta

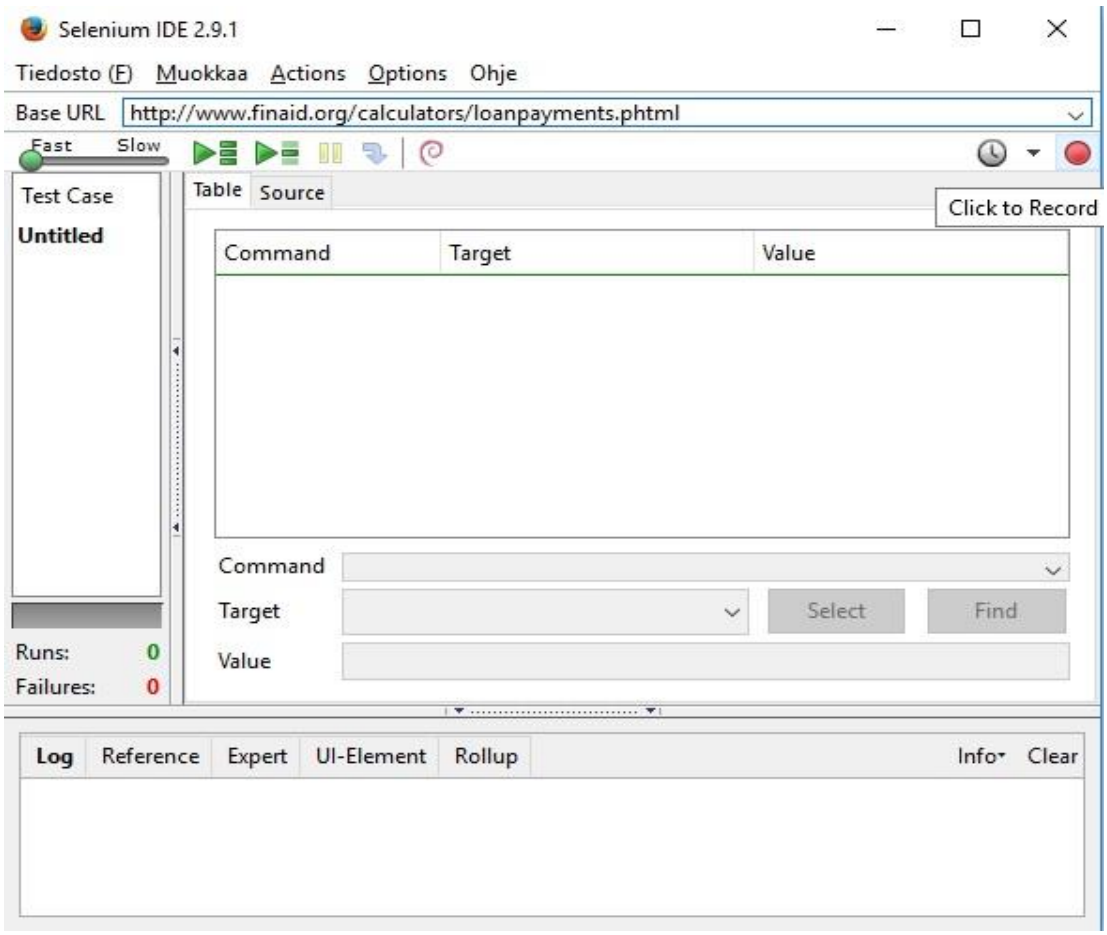
Tästä klikkaamalla päästään luomaan testiä. Luodaan esimerkiksi testi, jossa selain menee osoitteeseen http://www.finaid.org/calculators/loanpayments.phtml ja syöttää lainalaskuriin tiedot:

- Loan Balance: 10000
- Interest Rate: 10%
- Loan Term (Years): 5
- Minimum Payment: 100$

Tietojen syötön jälkeen testi klikkaa calculate-näppäintä, sekä tarkistaa että saadaan seuraavat tulokset:

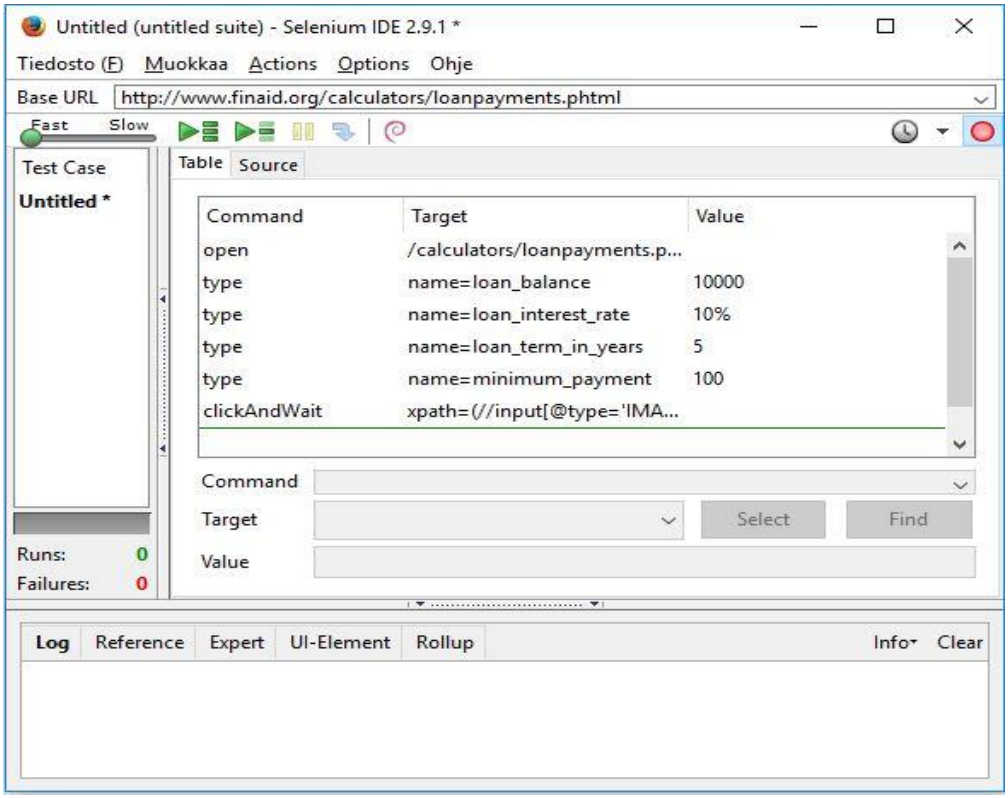
- Monthly Loan Payment: $212.47
- Number of Payments: 60
- Cumulative Payments: $12,748.23
- Total Interest Paid: $2,748.23

Testin luominen alkaa kirjoittamalla Base URL kohtaan testattava sivu, tässä tapauksessa http://www.finaid.org/calculators/loanpayments.phtml, seuraavaksi startataan nauhoitus oikealla näkyvästä punaisesta pallosta, jonka jälkeen voidaan itse syöttää selaimella halutut arvot ja painaa calculate -näppäintä ja Selenium IDE nauhoittaa kaikki tekemisemme (Kuva 2).



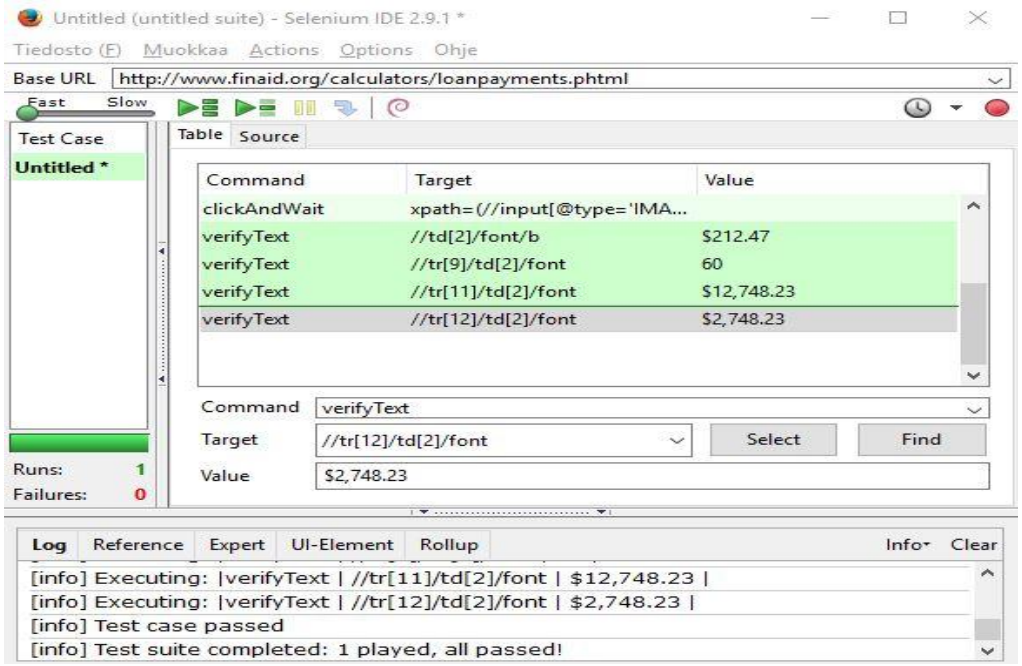
Kuva 2. Selenium IDE perusnäkymä

Tässä vaiheessa testin tulisi näyttää seuraavalta:



Kuva 3. Selenium IDE testi

Selain on nyt siis syöttänyt tarvittavat tiedot ja klikannut calculate-näppäintä ja siirtynyt seuraavalle sivulle jossa voidaan tarkistaa, että saadaan oikeat tulokset:



Kuva 4. Selenium IDE testin tarkistukset

Ylläolevassa kuvassa näkyy (Kuva 4), kuinka tarkistetaan verifyText komennolla, että oikeat arvot löytyvät oikeilta paikoilta. Targetista löytyvät arvot saadaan helposti klikkaamalla kuvassa näkyvää Select -näppäintä ja valitsemalla selaimesta oikea kohta. Kuvasta ilmenee myös, että testi on ajettu hyväksytysti. Yksinkertaisimmillaan Selenium-testitapausten luominen on näin yksinkertaista.

Käytettäessä Selenium WebDriveria, testien kirjoittaminen meneekin huomattavasti monimutkaisemmaksi ja vaatii paljon enemmän asennuksia. Selenium IDE oli saatavana vain Firefoxille, WebDrivereita taas on lähes kaikille alustoille esimerkiksi Google Chrome, Firefox, Opera, Safari ja jopa puhelimille, Android, iOS sekä Windows Phone. Nämä pluginit ovat kolmannen osapuolen tekemiä. Selenium ei myöskään ole lukittu mihinkään ohjelmointikieleen vaan se tukee ainakin seuraavia kieliä:

- Java
- C#
- Ruby

- Python
- Javascript (Node)

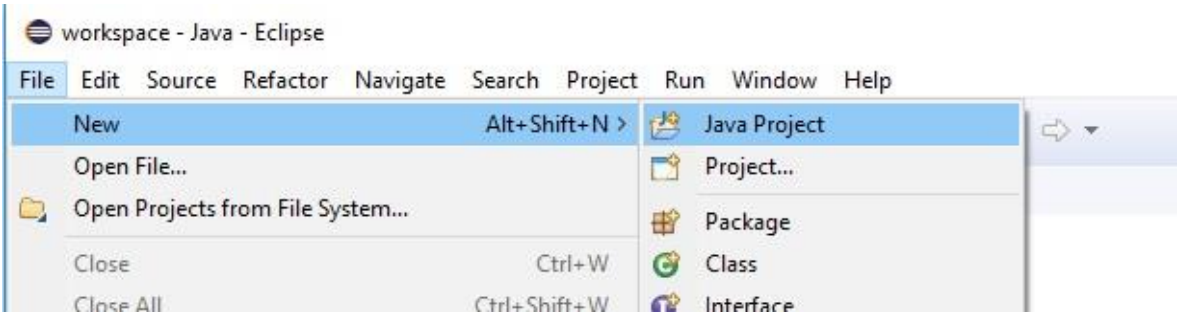
Nämä ovat Seleniumin itsensä tukemia kieliä, kolmannet osapuolet ovat tähänkin koodanneet plugineita tukemaan mm. PHP:tä ja Perliä (Selenium kotisivu). Selenium on siis hyvin monipuolinen tässä suhteessa. Selenium WebDriver soveltuu erinomaisesti web-applikaation testaamiseen silloin, kun itse ohjelmistoa kehittävät ohjelmoijat luovat samalla myös testit. WebDriverilla testien kirjoittaminen vaatii jonkinlaista osaamista ohjelmoinnin puolelta.

Seuraavassa esimerkissä kieleksi on valittu Java ja WebDriveriksi Chrome Driver.

Jotta WebDriverilla pääsee testejä kirjoittamaan, tulee työpisteellä olla asennettuna seuraavat välineet (Guru99, 2017):

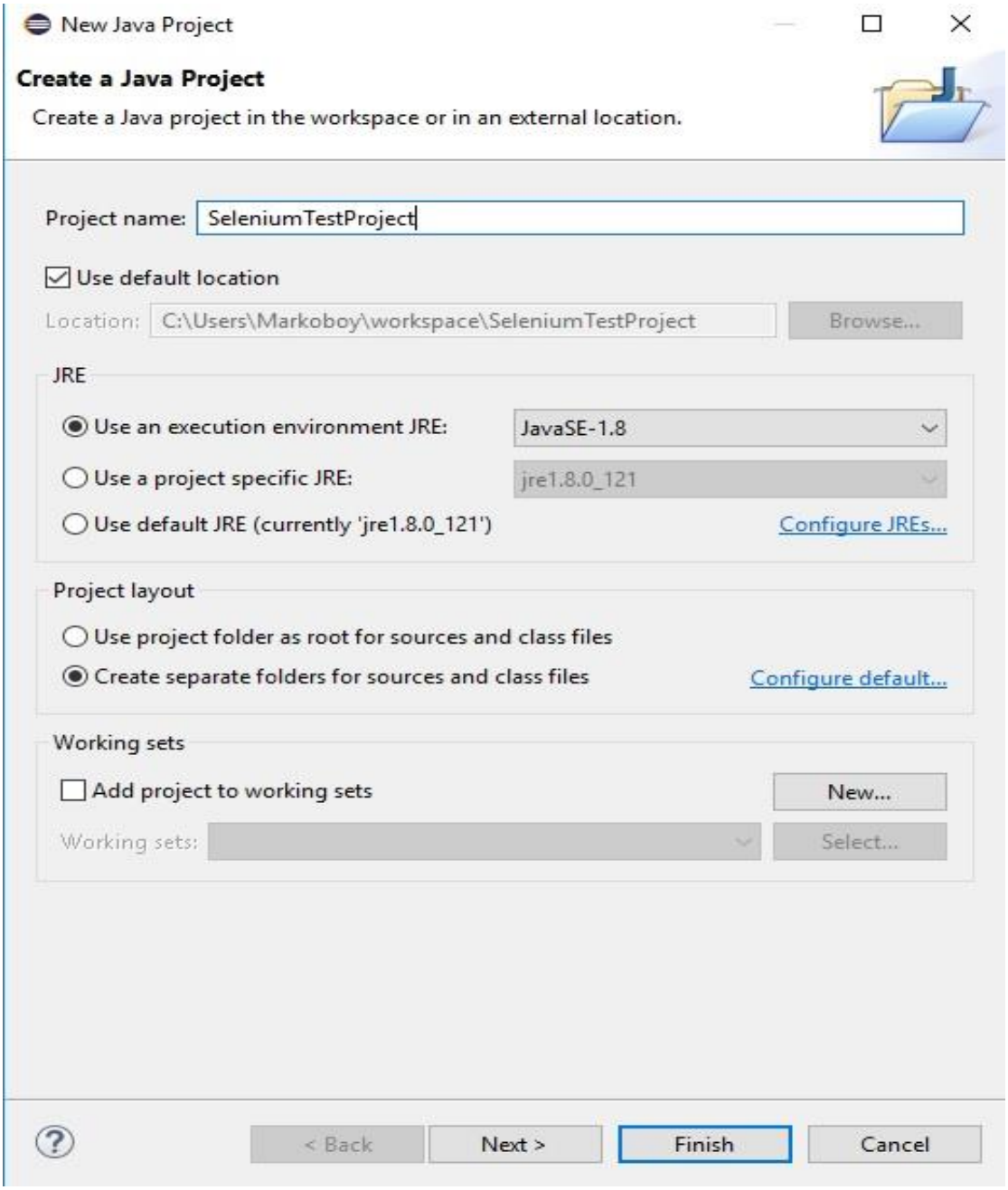
- Java Development Kit (JDK) - http://www.oracle.com/technetwork/java/javase/downloads/index.html
- Eclipse IDE - http://www.eclipse.org/downloads/
- Java Client Driver - http://seleniumhq.org/download/
- Chrome Driver - https://sites.google.com/a/chromium.org/chromedriver/downloads

Kun tarvittavat lataukset on tehty, tulee asentaa Java Development Kit sekä Eclipse, tämän jälkeen voidaan avata Eclipse ja luoda uusi projekti (Kuva 5):



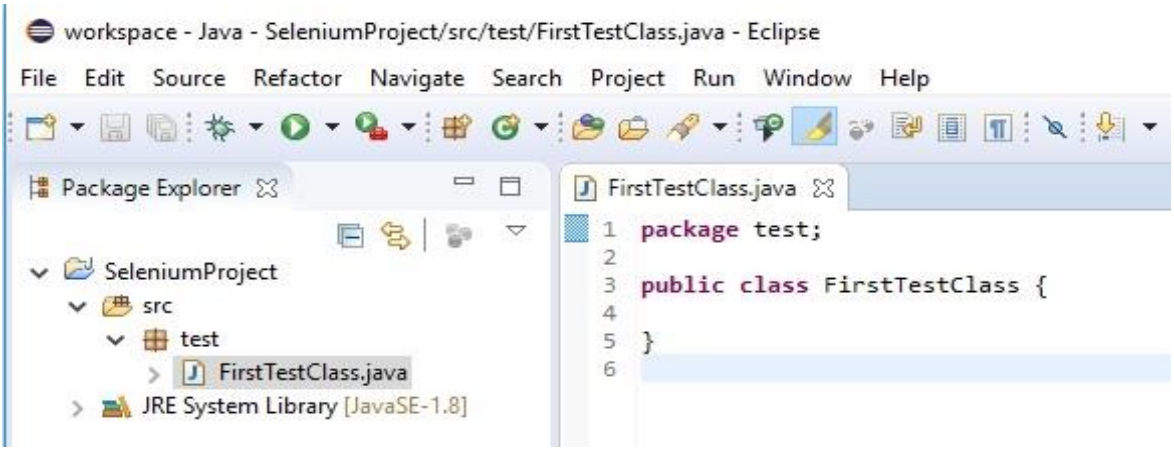
Kuva 5. Eclipse uuden projektin luominen

Tämän jälkeen aukeaa uusi ikkuna, johon voidaan syöttää projektille nimi ja jättää muut kohdat koskemattomiksi (Kuva 6):



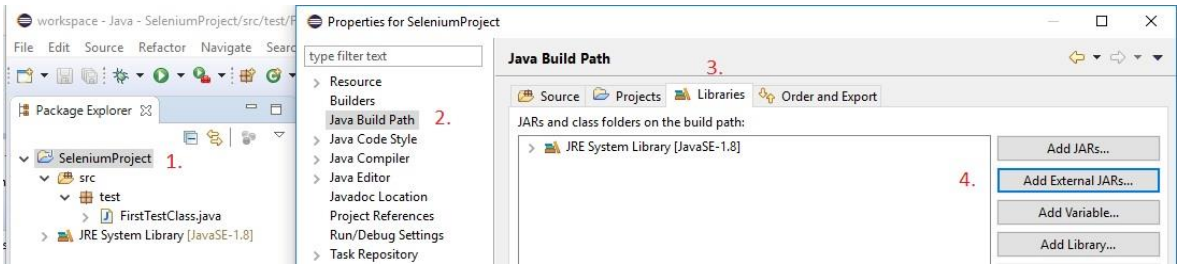
Kuva 6. Eclipse uuden projektin nimeäminen

Seuraavaksi tulee luoda projektin alle paketti ja sen alle itse testiluokka johon testi tullaan kirjoittamaan (Kuva 7):



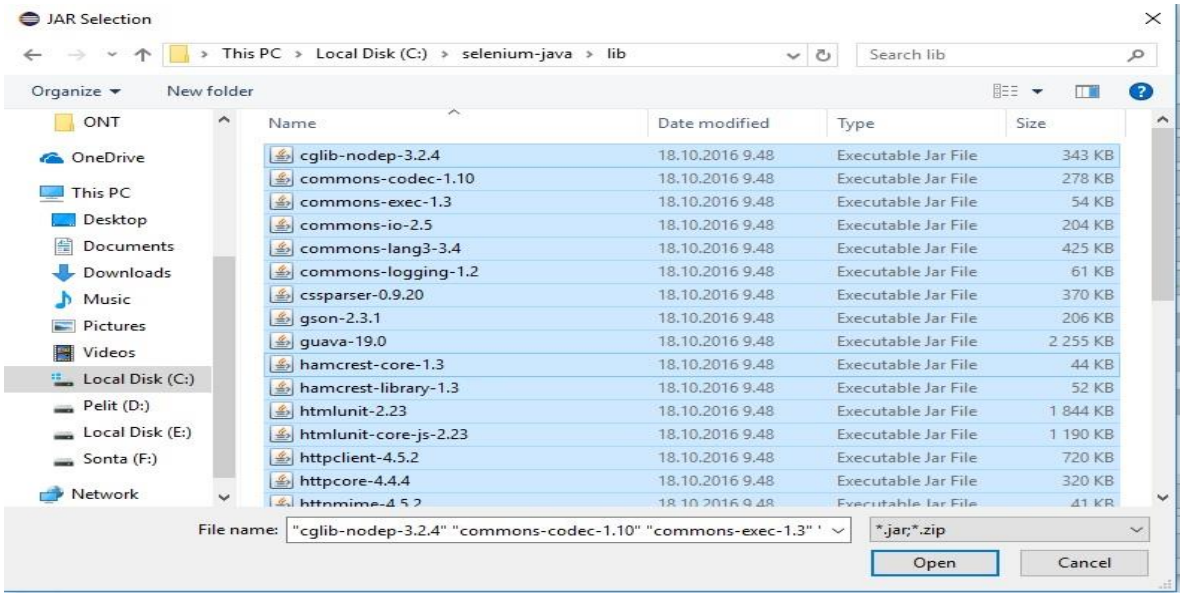
Kuva 7. Eclipse ensimmäisen testiluokka

Seuraavaksi liitetään Selenium WebDriver Javan Build Pathiin (Kuva 8):



Kuva 8. Eclipse Selenium WebDriverin lisääminen Java Build Pathiin

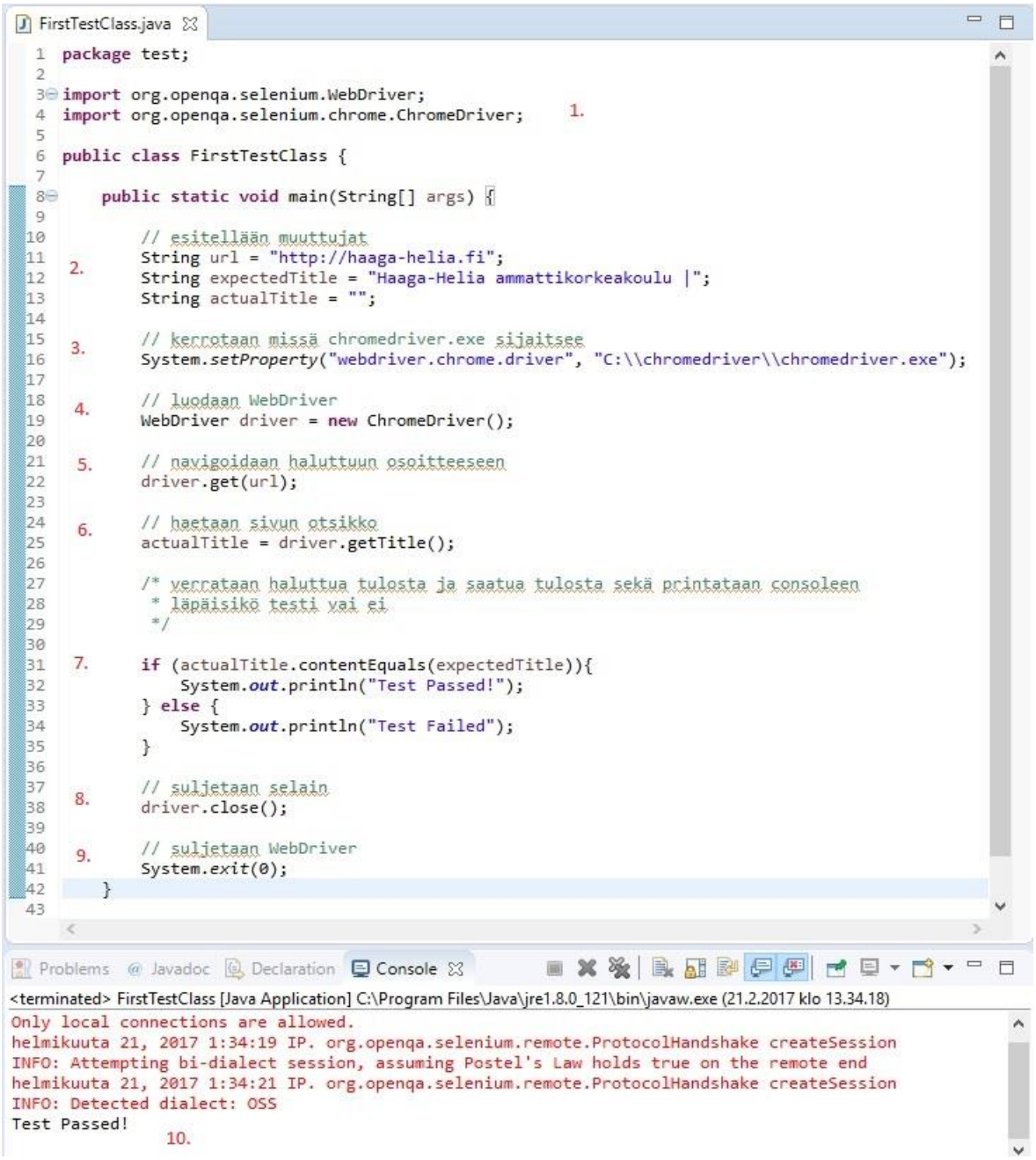
Klikatessa ”Add External JARs…” aukeaa ikkuna johon valitaan ladatusta Selenium Java Client kansiosta tarvittavat jar tiedostot, eli kaikki lib kansiosta ja sen ulkopuolelta löytyvät (Kuva 9):



Kuva 9. Tarvittavien JARien lisääminen

Asennukset ovat nyt valmiit ja voidaan siirtyä itse testin kirjoittamiseen. Esimerkkitesti selitetty auki:

1. Tarvittavat importit
2. Esitellään testissä käytettävät muuttujat
3. Kerrotaan WebDriverille missä chromedriver.exe sijaitsee
4. Luodaan WebDriver
5. Navigoidaan kohdassa 2 esiteltyyn osoitteeseen
6. Haetaan sivuston otsikko
7. Verrataan saatua otsikkoa kohdassa 2 esiteltyyn odotettuun otsikkoon, sekä tulostetaan consoleen, onko testi onnistunut
8. Suljetaan selain
9. Suljetaan WebDriver
10. Testin tulos: onnistunut



Kuva 10. Selenium WebDriver esimerkkitesti

2.2.2 Robot Framework

”Robot Frameworkin kehitys aloitettiin Nokia Networksilla vuonna 2005 ja se on noussut kymmenessä vuodessa organisaatioiden luottotyökaluksi testausvaiheessa.” (Eficode, 2016.)

Robot Framework on geneerinen testiautomaatio framework hyväksyntätestaukselle sekä hyväksymistestivetoiselle ohjelmistokehitykselle (Acceptance test-driven development). Se käyttää avainsanoihin perustuvaa lähestymistapaa testaukseen, joka mahdollistaa testien luomisen helposti ymmärrettävään muotoon. Avainsanat mahdollistavat myös testien kirjoittamisen sellaisille henkilöille joilla ohjelmointitaustaa ei ole, jolloin kokeneemmat ohjelmoijat voivat keskittyä uusien avainsanojen luomiseen, joka vaatii ohjelmoinnin osaamista. Avainsanastoa voi laajentaa joko Pythonilla, joka on Robot Frameworkin ydinteknologia, tai Javalla. Robot Framework on kokonaan avointa lähdekoodia.

Robot Framework on Seleniumia monikäyttöisempi siksi, että se sisältää Selenium2Libraryn, joka on sen käytetyin ulkoinen kirjasto, eli sillä voidaan käytännössä testata kaikki mitä Seleniumillakin ja paljon muuta, kuten FTP:tä, MongoDB:tä, Androidia, Appiumia ja muita. Lisäksi siihen on paljon ohjelmointirajapintoja (API), jotka auttavat tekemään siitä mahdollisimman laajennuskykyisen. (TechBeacon, 2016.)

Robot Framework on myös testiautomaatiota konsulttipalveluna tarjoavan VALA Group Oy:n käytetyin työkalu. Juurikin sen avainsanoihin perustuvan lähestymistavan takia, joka mahdollistaa useampien ihmisten tuomisen lähemmäksi testaamista, joka taas vapauttaa kokeneemmat ohjelmoijat tuottamaan lisää avainsanoja. (Guberna, 2017.)

Robot Frameworkia asennettaessa tulee ensin olla asennettuna Python, jonka voi ladata osoitteesta: https://www.python.org/ftp/python/2.7.12/python-2.7.12.msi

Pythonin asentamisen jälkeen avataan komentorivi ja siirrytään Pythonin asennuskansioon, tässä tapauksessa: *cd C:\Python27*

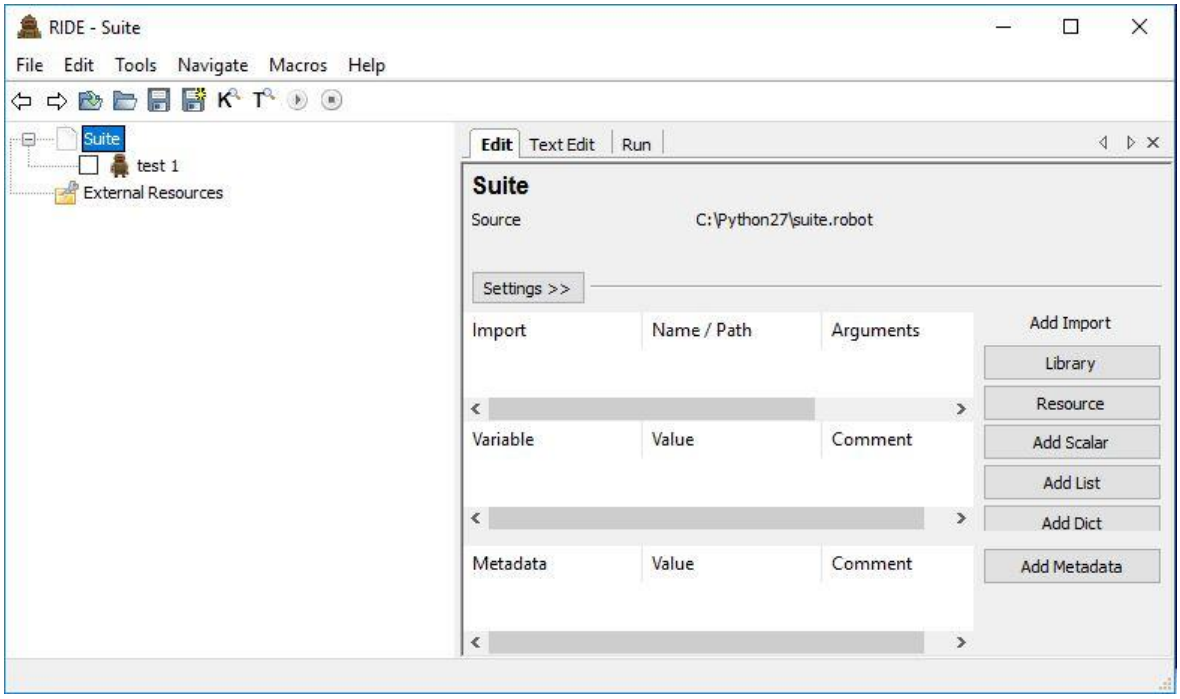
Seuraavaksi voidaan ladata sekä asentaa Robot Framework, joka tapahtuu komennolla: *python -m pip install robotframework*

Tämän jälkeen ladataan Selenium-kirjasto: *python -m pip install robotframework-selenium2library*

Sitten asennetaan Pythonille tarkoitettu simppeli editori wxPython, jota RIDE tarvitsee toimiakseen. WxPythonin löytää osoitteesta: https://sourceforge.net/projects/wxpython/files/wxPython/2.8.12.1/wxPython2.8-win32-unicode-2.8.12.1-py27.exe/download

Viimeisenä vielä asennetaan RIDE IDE: *python -m pip install robotframework-ride*

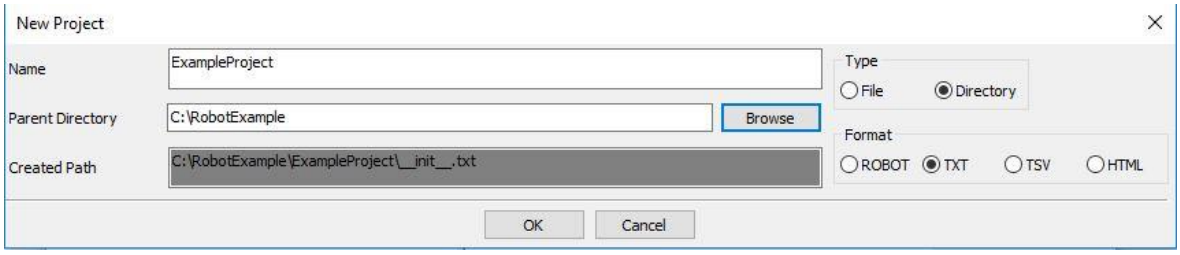
Nyt ollaan tilanteessa jossa kirjoittamalla komentoriville *python Scripts\ride.py* tulisi aueta RIDE (Kuva 11):



Kuva 11. RIDE perusnäkymä

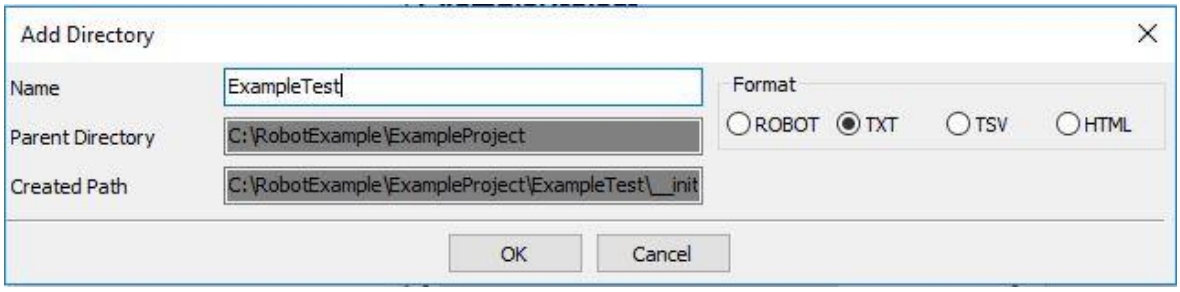
Nyt voidaan aloittaa esimerkkitestin luominen, testi tulee navigoimaan sivulle http://demo.mahara.org, syöttämään käyttäjätunnuksen sekä salasanan ja tarkistamaan että sisäänkirjautuminen onnistui. Testi ajetaan Firefoxilla.

Aloitetaan luomalla uusi projekti File:n alta (Kuva 12):



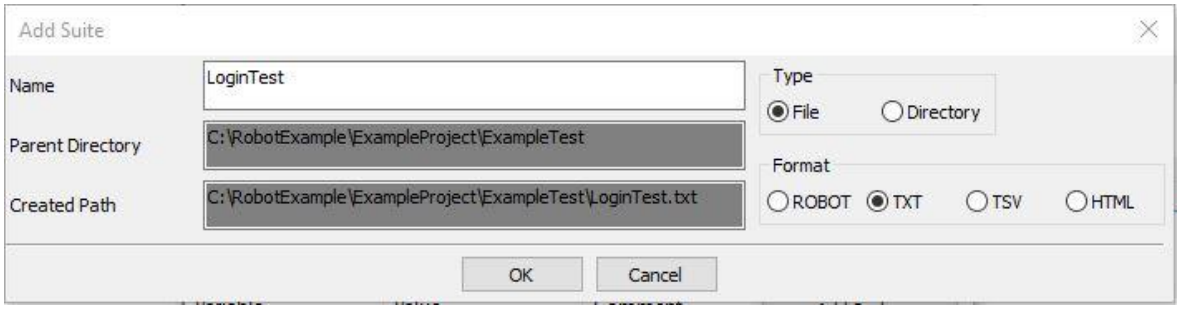
Kuva 12. RIDE uuden projektin luominen

Kun projekti on luotu, klikataan sitä hiiren oikealla näppäimellä ja valitaan New Directory (Kuva 13):



Kuva 13. RIDE uuden kansion luominen

Sitten samalla tavalla New Suite (Kuva 14):



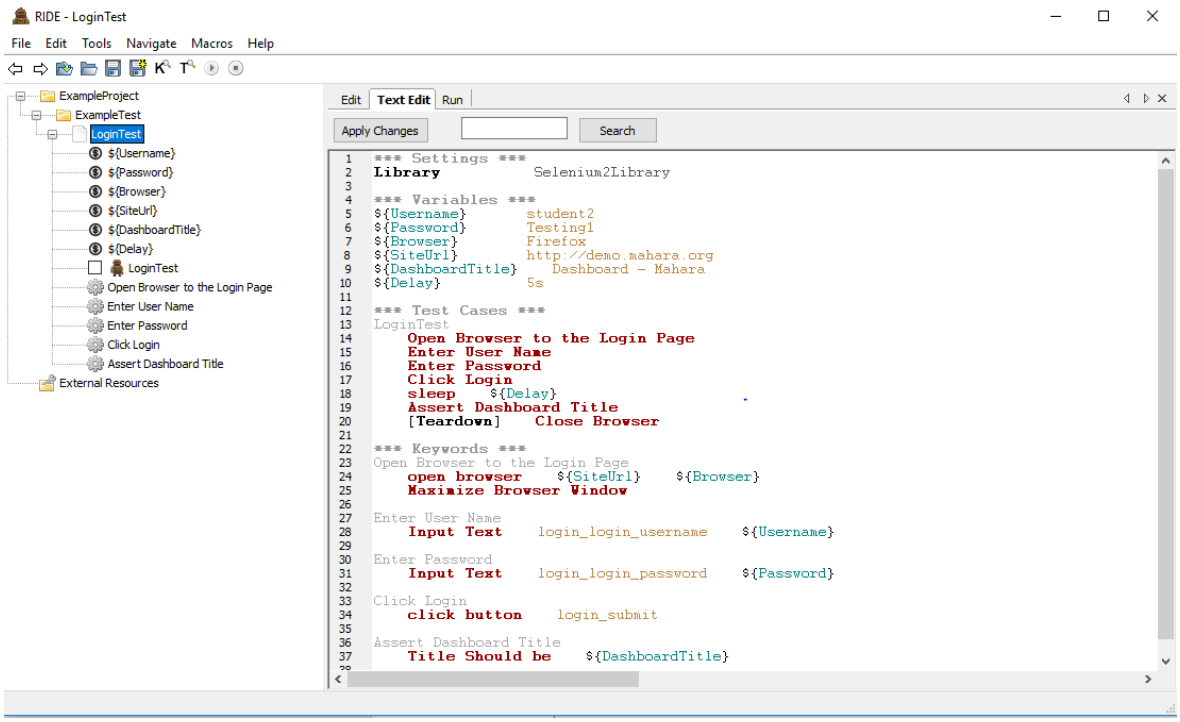
Kuva 14. RIDE uuden testiluokan luominen

Jonka jälkeen valitaan juuri luotu testi ja syötetään Text Editoriin seuraavat syötteet (Kuva 15):

```
*** Settings ***
Library           Selenium2Library

*** Variables ***
${Username}       student2
${Password}       Testing1
${Browser}        Firefox
${SiteUrl}        http://demo.mahara.org
${DashboardTitle}    Dashboard - Mahara
${Delay}          5s

*** Test Cases ***
LoginTest
    Open Browser to the Login Page
    Enter User Name
    Enter Password
    Click Login
    sleep    ${Delay}
    Assert Dashboard Title
    [Teardown]    Close Browser

*** Keywords ***
Open Browser to the Login Page
    open browser    ${SiteUrl}    ${Browser}
    Maximize Browser Window

Enter User Name
    Input Text    login_login_username    ${Username}
Enter Password
    Input Text    login_login_password    ${Password}
Click Login
    click button    login_submit
Assert Dashboard Title
    Title Should be    ${DashboardTitle}
```

Kuva 15. Esimerkkitestin syötteet

Tämän jälkeen testin tulisi näyttää seuraavalta ja sen voi ajaa Run -välilehdeltä (Kuva 16):



Kuva 16. RIDE esimerkkitesti

Näin helposti käyttämällä vain valmiiksi annettuja avainsanoja, sekä muokkaamalla muutamaa syötettä saatiin luotua ensimmäinen Robot Framework testi joka testaa sivuston login toimintoa.

2.3 Muita automatisoitavia testejä

Käyttöliittymätestauksen lisäksi on muutamia muita testityyppejä, jotka ovat käytännöllistä myöskin automatisoida.

2.3.1 Yksikkötestit

Yksikkötestaus on ohjelmiston testaamisen taso, jossa testataan ohjelmistoon kuuluvia mahdollisimman pieniä yksittäisiä osia, eli yksiköitä. Yksikkö tarkoittaa tässä tapauksessa esimerkiksi yksittäisen olion yksittäistä metodia. Yksikkötestauksen päätavoite onkin pilkkoa ohjelmisto niin pieniin osioihin kuin mahdollista. (Hellas, 2011.)

Yksikkötestit on hyvä automatisoida, sillä niitä saattaa tulla laajassa ohjelmistossa jopa tuhansia. Yksikkötestit testaavat niin pieniä osia ohjelmistosta, että niiden avulla on helppo ongelmien ilmetessä paikantaa ongelman lähde. Kattava yksikkötestaus pääsee parhaimmilleen käytettäessä ketteriä menetelmiä, jolloin uusia buildeja tulee useita päivittäin. (Smarteducation, 2016.)

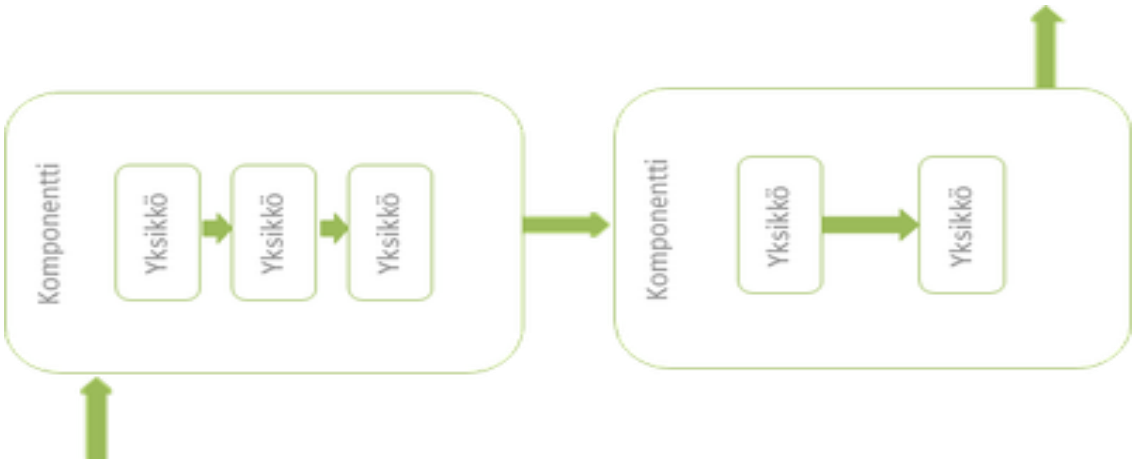
Yksikkötestaukseen on olemassa useita valmiita frameworkeja, lähes kaikille ohjelmointikielille. Näistä Javalle suosituimpana mainittakoon JUnit. Alla esimerkkitesti JUnitilla luotuna, jossa testataan yksinkertaisen laskimen, ja tarkemmin laskimen tester -metodin toimintaa (Kuva 17).

```
import static org.junit.Assert.assertEquals;

import org.junit.Test;

public class MyTests {

        @Test
        public void multiplicationOfZeroIntegersShouldReturnZero() {
                MyClass tester = new MyClass(); // MyClass is tested

                // assert statements
                assertEquals("10 x 0 must be 0", 0, tester.multiply(10, 0));
                assertEquals("0 x 10 must be 0", 0, tester.multiply(0, 10));
                assertEquals("0 x 0 must be 0", 0, tester.multiply(0, 0));
        }
}
```

Kuva 17. Esimerkki yksikkötesti (Vogella, 2016.)

2.3.2 Integraatiotestit

Integraatiotestaus on ohjelmiston testaamisen taso, jossa ohjelmiston yksiköt kootaan suuremmiksi komponenteiksi ja testataan niiden yhteistoimintaa (Kuva 18). Tällöin testeihin jää kiinni erilaiset ohjelmointivirheet kuin yksikkötesteissä. Mitä pidemmälle ohjelmistokehitysprojekti etenee, sitä enemmän ohjelmistossa on yksiköitä, sekä niistä koostuvia komponentteja. Tämä johtaa siihen, että integraatiotesteissä läpi käytävät suorituspolut lisääntyvät ja samalla ne pitenevät. Tässä vaiheessa on hyödyllistä ottaa käyttöön regressiotestaus, joka tarkoittaa jo aikaisemmin ajettujen testien uudelleen ajamista, jolloin varmistutaan siitä, etteivät uusimmat muutokset aiheuta ongelmia aiemmin toteutettujen elementtien toiminnassa. (Smarteducation, 2016.)



Kuva 18. Integraatiotestaus testaa yksikkötestausta suurempia kokonaisuuksia (Smarteducation, 2016.)

2.3.3 Suorituskykytestit

Suorituskyky on todella kriittinen tekijä web-ohjelmistojen käyttäjäkokemukselle, ja näin ollen myös menestykselle. Nykyaikana ihminen turhautuu todella nopeasti, mikäli ohjelmisto ei pysy käyttäjän toimintojen perässä ja käyttäjä joutuu jatkuvasti odottamaan. Näin ollen on tärkeää pitää huoli ohjelmiston kasvaessa, että suorituskyky myös pysyy korkeana. Tässä kohdassa mukaan astuvat automatisoidut suorituskykytestit. Suorituskykytesteillä on tarkoitus löytää ohjelmiston suorituskyvystä mahdolliset pullonkaulat, jotka hidastavat ohjelmiston toimintaa rasituksen alaisena.

Suorituskykytestit eroavat yksikkö- ja integraatiotesteistä suuresti siten, että niitä ei ole tarvetta ajaa yhtä usein, eli kaikkien muutosten jälkeen. Ne ovat järkevintä ajaa esimerkiksi ajastetulla buildilla kerran yössä.

Alla viisi yleisintä suorituskyvyn testaus työkalua: (Softwaretestinghelp, 2017.)

- WebLOAD
- LoadUI NG Pro
- Apica LoadTest
- LoadView
- Load Xen

2.4 Jatkuvan integroinnin palvelut

Jatkuva integrointi on ohjelmistokehityksen käytäntö, jossa ohjelmistotiimin jäsenet integroivat työnsä usein, yleensä jokainen vähintään kerran päivässä, joka johtaa tiimin sisällä useisiin integrointeihin yhden päivän aikana. Jokainen integraatio tarkistetaan automatisoidulla buildilla, johon kuuluu myös testit, jotta integraatiovirheet havaittaisiin mahdollisimman aikaisessa vaiheessa. Monet ohjelmistotiimit ovat havainneet, että tämä vähentää merkittävästi integraatiosta syntyviä ongelmia ja mahdollistaa yhteneväisen ohjelmiston nopeamman tuottamisen. (Fowler, 2006.)

Jotta jatkuvan integroinnin palveluita voi hyödyntää tulee ohjelmistotiimillä olla seuraavat asiat hoidossa:

- Yksi yhteinen versionhallinta käytössä
- Ohjelmiston kääntäminen tulee olla automatisoitu
- Yksikkötestit tulee ajaa automaattisesti
- Jokainen kehittäjä vie versionhallintaan päivittäin työnsä
- Integrointiympäristö, joka on identtinen tuotantoympäristön kanssa (yhtä paljon muistia, samat versiot ohjelmistoissa jne.)

Seuraava askel jatkuvan integraation kehittämiselle on tuoda CI-palvelimelle (Continous Integration) automaattinen deployaus, joka tarkoittaa siis sitä, että buildattu koodi paketoidaan ja testataan testipalvelimella, mikäli se on siis ensin läpäissyt buildissa ajetut testit. Automaattinen deployaaminen mahdollistaa muun muassa ohjelmiston demoamisen asiakkaalle koska vain, lisäksi testipalvelimella voidaan ajaa pitkäkestoisia kuormitustestejä sekä testaajien on mahdollista tehdä siellä tutkivaa testausta aina tuoreimmalle järjestelmän versiolle. (Ohjelmistotuotanto, 2012.)

2.4.1 Jenkins

Jenkins on avoimen lähdekoodin automaatiopalvelin. Se on koodattu Javalla ja se on valmis toimimaan suoraan paketista Windowsilla, Mac OS X:llä ja muilla UNIX käyttöjärjestelmillä. Jenkinsin avulla yritykset voivat nopeuttaa ohjelmistokehityksen prosessia automaation kautta. Jenkins hallitsee ja kontrolloi ohjelmistokehityksen elinkaaren kaikenlaisia prosesseja, kuten buildit, dokumentointi, testit, paketit, deployt ja paljon muuta. Jenkins

voidaan ohjelmoida tarkistamaan, onko versionhallintaan, kuten SVN tai Git, tullut muutoksia, ajamaan testit ja sen jälkeen ryhtyä toimenpiteisiin, palauttamaan ohjelmiston entiselleen, mikäli testit epäonnistuvat (rollback) tai jatkaa eteenpäin, mikäli testit menivät läpi. (Cloudbees, 2016.)

Jenkins projekti aloitettiin vuonna 2004 Kohsuke Kawaguchin toimesta. Kiitos Jenkinsin laajennuskyvyn ja aktiivisen yhteisön, Jenkins on kasvanut merkittävästi. Nykyään Jenkins tarjoaa yli 1200 lisäosaa joka mahdollistaa Jenkinsin integroimisen melkein mihin tahansa suosituista teknologioista. Se on ylivoimaisesti suosituin automaatioserveri ja sillä olikin joulukuussa 2016 jo yli miljoona käyttäjää ympäri maailmaa. (Cloudbees, 2016.)

Jenkinsin pystyttäminen vaatii hieman työtä, riippuen projektin tyypistä siihen menee muutamasta tunnista pariin päivään. Kuitenkin kun se on kerran pystytetty, on sen ylläpitäminen hyvin yksinkertaista. Jenkins on ilmainen, kun pyörität sitä omalla palvelimellasi. Mikäli oman palvelimen pystyttäminen tuottaa kuitenkin ongelmia, on sen pystyttäminen mahdollista ostaa palveluna Jenkinsiltä. (Waller, 2014.)

Jenkinsin vahvuudet:

- Ilmainen
- Muokattavuus
- Lisäosat
- Systeemin kokonaisvaltainen kontrolli

Jenkinsin heikkoudet:

- Oma palvelin, tai useampi palvelin, on pakollinen. Joka johtaa lisäkustannuksiin.
- Konfigurointiin kuluva aika. (Django Stars, 2017)

2.4.2 Travis CI

Travis CI on hyvin samanlainen kuin Jenkins, siinä on kuitenkin pieniä eroja. Travis CI on ilmainen avoimen lähdekoodin projekteille, mutta maksullinen mikäli projektisi on yksityinen. Toisin kuin Jenkins, Travis CI tarjoaa palvelimet käyttöösi. Travis CI ei vaadi ylläpitoa, koska palvelin tulee Travis CI:n puolelta, he hoitavat kaikki päivitykset. Ainut mitä käyttäjän tulee

tehdä, on ylläpitää konfiguraatiotiedostoa, jossa kerrotaan mitä työkaluja haluat käyttää. Esimerkiksi mikä PHP- tai Ruby-versio on käytössä. (Waller, 2014.)

Jenkinsiin verrattuna Travis CI on erittäin nopea laittaa pystyyn. Ainut asia mitä käyttäjän tulee tehdä, on luoda edellä mainittu konfiguraatiotiedosto.

Travis CI:n vahvuudet

- Nopeasti otettavissa käyttöön
- Yksinkertainen YAML konfiguraatio
- Ilmainen avoimen lähdekoodin projekteille
- Ei tarvitse omia palvelimia

Travis CI:n heikkoudet:

- Suhteellisen korkea hinta
- Heikko muokattavuus. (Django Stars, 2017.)

Kumpikin edellä mainituista jatkuvan integroinnin palveluista ovat toimivia ratkaisuja. Valinta on hyvin pitkälti siinä, haluatko käyttää aikaa etukäteen palvelun pystyttämiseen vai maksaa kuukausimaksua ja saada palvelun heti pystyyn. Mikäli projektisi on avointa lähdekoodia, Travis CI on paras valinta, koska pystyttämiseen ei mene aikaa ja se on ilmainen. Yksityisille projekteille paras valinta on todennäköisesti Jenkins, koska kun sen kerran itse pystyttää on sen ylläpitäminen todella helppoa. (Waller, 2014.)

2.5 Manuaalisen testauksen tulevaisuus

Onko manuaalinen testaaminen kuollut? Voiko testiautomaatio kokonaan korvata manuaaliset testaajat? Näihin kysymyksiin tulee törmättyä paljon etsiessä asiasta tietoa. Testiautomaatio on kuitenkin vain työkalu. Todella voimakas työkalu, kun se on toteutettu ja sitä hyödynnetään oikein, mutta silloinkin vain työkalu. Testiautomaatio ei korvaa manuaalisia testaajia, se mahdollistaa testaajia saamaan parempia tuloksia yhä lyhyemmässä ajassa. Testiautomaation on tarkoitus kattaa pitkäveteiset regressiotestit, jotka toistuvat todella usein, sekä esimerkiksi testit jotka vaativat todella paljon eri tietojen syöttämistä ja vievät näin ollen todella paljon aikaa. Testiautomaatio mahdollistaa testaajien keskittymisen esimerkiksi tutkivaan testaukseen, joka vaatii ihmisen luovuutta. (Guberna, 2017.)

Yksi suuri palanen manuaalisen testaajan hyödyistä, jota ei automatisoinnilla voida hoitaa, on käyttäjäkokemuksen testaaminen, joka on yksi suurimmista asioista joka määrittää mistä applikaatioista tulee oikeasti suosittuja. Tästä esimerkkinä Wolt. Markkinoilla on olemassa lukuisia muitakin ruoan kuljetus applikaatioita, mutta Woltin applikaation korkea laatu yhdistettynä miellyttävään käyttäjäkokemukseen on ollut avain Woltin suureen suosioon. (Pesonen, 2016.)

Kuten Pesonen blogissaan mainitsee, manuaalisen testaamisen rooli on muuttunut, mutta tarve on edelleen tallella. Kun regressio- ja muut aikaa vievä testit on automatisoitu, jää taitaville testaajille enemmän aikaa löytää ohjelmiston heikkouksia omien kokemustensa perusteella.

3 Testiautomaatio Trimico Oy:n LogiPlan-kehitystiimissä

Trimicolla on testiautomaatiota ollut käytössä projektien alusta lähtien, tässä luvussa tullaan selvittämään, onko nykyisissä käytänteissä parantamisen varaa, vai ollaanko jo nyt tarpeeksi hyvällä tasolla. Trimicolla on myös käytössään yksi tekninen testaaja, joka ei tällä hetkellä luo testitapauksia itse.

3.1 Nykytilanne

Nykytilanne LogiPlan-kehitystiimissä työskentelevän ohjelmistokehittäjän mukaan on seuraavan lainen.

Käyttöliittymän testaukseen käytetään Selenium frameworkia, joka on hyvin perusteltu valinta, sillä Trimicolla on käytössään vain muutama testaaja, voivat he keskittää kaiken työnsä manuaaliseen testaamiseen. Mikäli manuaalisia testaajia olisi suurempi joukko, olisi syytä harkita Robot Frameworkiin siirtymistä, sillä se mahdollistaa testitapausten kirjoittamisen myös sellaisille henkilöille, jotka eivät varsinaisesti osaa ohjelmoida. Selenium frameworkin pystyttämiseen on panostettu paljon, joten sitä ei ole järkevää tässä vaiheessa lähteä vaihtamaan.

Selenium-testien lisäksi Trimicolla on automatisoitu yksikkö-, integraatio-, suorituskyky- sekä wicket-testit. Näiltä osin testiautomaatio näyttäisi olevan kunnossa, joskin jatkuvaa kehittämistä on.

Automatisoidut testit ajetaan Jenkins-palvelimella, mikä on myöskin perusteltu valinta, sillä kyseessä ei ole avoimen lähdekoodin projekti.

Normaali työnkulku Trimicolla menee niin, että kehittäjä saa jonkun tehtävän, joko jonkin uuden toiminnallisuuden tai bugikorjauksen, kummassakin tapauksessa tehtävän toteuttamisen yhteydessä kehittäjän tulisi kirjoittaa testit jotka kattavat tehtävässä luodut tai korjatut ominaisuudet. Tätä seurataan tiimin sisällä koodikatselmoinnin yhteydessä. Tässä kohtaa kuitenkin esiintyy jonkin verran puutteita, testejä jää syystä tai toisesta kirjoittamatta, eikä niitä aina osata vaatia koodikatselmoinnissa.

Poikkeuksena normaalille työnkululle oli Selenium-testien käyttöönottovaihe, jolloin Selenium framework rakennettiin ja lisäksi sen avulla testikattavuutta haluttiin nostaa

nopeammalla aikataululla kuin yleensä. Koska Selenium frameworkia ei otettu käyttöön heti projektin alussa, oli ohjelmiston automatisoituun testikattavuuteen jäänyt aukkoja.

Trimicolla testit ajetaan kaikkien muutosten jälkeen. Automatisoidut testit voidaan ajaa omalla koneella tai dedikoidulla palvelimella. Jo ennen kuin tehtävä siirretään koodikatselmointiin, muutoksille ajetaan yksikkötestit. Mikäli yksikkötestit eivät mene läpi, muutosten eteenpäin vieminen on estetty. Kun tehtävä on koodikatselmoitu hyväksytysti, viedään se eteenpäin haluttuun haaraan versionhallintaan, tavallisesti master-haaraan. Tässä vaiheessa muutoksia vasten ajetaan yksikkötestien lisäksi myös integraatio-, sekä Selenium-testit. Yksikkö- sekä integraatiotesteistä tulokset saadaan nopeasti, mutta Selenium-testien ajossa kestää huomattavasti pidempään. Seleniumtestejä on kertynyt jo melkein 600 kappaletta ja ne ajetaan kahdella slave-koneella, joiden konfiguraatiossa on vielä parantamisen varaa. Etenkin suorituskykyä joudutaan aika-ajoin kasvattamaan. Riippuen slave-koneiden kuormituksesta Selenium-testien ajaminen kestää 2,5 tunnista, jolloin molemmat slave-koneet ovat koko ajan vapaina, 4,5 tuntiin jolloin kaikki testit ajetaan yhdellä koneella peräkkäin. Testien rinnakkain ajamiseen ja optimointiin on käytetty viime aikoina aikaa, mikä on nopeuttanut testien ajoa.

Tuloksia seurataan kehitystilassa sijaitsevista monitoreista. Pitkistä ajoajoista johtuen myös testitulosten seuranta välillä unohtuu, koska tulokset saadaan aikaisintaan parin tunnin viiveellä, on kehittäjä saattanut jo siirtyä seuraavan tehtävän pariin. Jos testituloksia ei seurata, testien epäonnistuessa niitä ei välttämättä heti korjata, joka johtaa siihen, että korjattavia testejä saattaa kertyä ruuhkaksi asti eikä tiedetä kenen muutoksista mikäkin on mennyt rikki. Tällöin kaikkien työ hidastuu sekä joku muu saattaa korjata jonkun toisen henkilön rikkoman testin, eikä ongelman aiheuttaja välttämättä koskaan saa tietoa mitä on rikkonut ja miksi, eikä oppimista tapahdu. Tällöin ongelma saattaa helposti uusiutua tulevaisuudessa.

3.2 Parannusehdotukset

Suurimpana sekä selkeimpänä ongelmana on Selenium-testien pitkä ajoaika, mihin on kehitystiimissä jo tartuttukin. Tiimillä on jo käytössään kaksi slave-konetta, joiden tehoa on pyritty kasvattamaan. Koneiden määrää voi tarvittaessa lisätä, koska slave-koneeksi kelpaa hyvin vähän vanhempikin jo kehityskäytöstä poistunut tietokone. Koneiden asennus- ja konfigurointiohjeet on kuitenkin oleellista dokumentoida, jotta uusien palvelimien konfigurointi onnistuu helposti. Ongelmia syntyy, mikäli edellisten koneiden konfiguroimista

ei ole dokumentoitu asianmukaisesti, sillä aiheesta löytyy hyvin vähän dokumentaatiota internetistä. Asioita, joita tulee ainakin huomioida slave-koneen konfiguroinnissa

- Täsmäävät Java-versiot
- Näytön resoluutio, kun käytössä on vanhoja kehityskoneita saattaa niiden näytön resoluutio olla riittämätön
- Palomuurien asetukset

Selenium-testien pitkä ajoaika heijastuu suoraan seuraavaan ongelmaan, joka on testitulosten huono seuranta. Aikaisemmin, kun testien ajaminen kesti jopa 4,5 tuntia, oli luonnollista, että kehittäjä on jo siirtynyt seuraavan tehtävän pariin ja edellisen tehtävän testien seuranta jää vähemmälle huomiolle. Tämä ei kuitenkaan ole hyvä tilanne. Rikkoontuneet testit tulisi aina korjata sama henkilö joka ne on rikkonut, jotta oppi menisi oikeaan osoitteeseen ja jatkossa samoilta virheiltä vältyttäisiin. Tilanne, jossa rikkoontuneita testejä kertyy paljon ja jossain vaiheessa yksi henkilö ottaa tehtäväkseen korjata kaikki, saattaa lyhyellä tähtäimellä tuntua tehokkaammalta, mutta pitkässä juoksussa siinä tullaan häviämään resursseja, koska samat virheet tulevat toistamaan itseään.

Testien seurantaan ei kuitenkaan tekniset muutokset tuo apua, vaan siitä tulee keskustella tiimin sisäisesti ja jokaisen pitää henkilökohtaisesti pitää huoli, että korjaa omat jälkensä.

Seuraavat kehityskohdat liittyvät testikattavuuteen. Koska LogiPlan-tiimissä Selenium otettiin käyttöön kesken projektin, eikä heti alussa, jäi testikattavuuteen aukkoja. Talvella 2016 Trimico palkkasikin tämän opinnäytetyön kirjoittajan tekemään Seleniumtestejä työharjoittelun ajaksi ja täten parantamaan tuotteensa testikattavuutta. Sekään ei riittänyt kaikkien aukkojen paikkaamiseen. Onkin ymmärrettävää, että aukkoja ei lähdetä täydellisesti paikkaamaan, koska se vaatii niin paljon työtunteja. On kuitenkin pidettävä huoli, että testikattavuus nousee koko ajan ja että kaikkien uusien ominaisuuksien mukana tehdään myös asiaankuuluvat testit. Tässä asiassa on ohjelmistokehittäjän haastattelun mukaan tiimissä hieman lipsuttu, eikä testejä ole aina osattu vaatia koodikatselmointi vaiheessakaan. Tämäkin on tiimin sisäinen asia, jota ei ole järkevää lähteä korjaamaan teknisillä muutoksilla.

Kokonaisuudessaan LogiPlan-tiimin testiautomaatio on varsin hyvällä mallilla, mutta pienillä muutoksilla käytäntöihin, sekä pienillä satsauksilla tekniseen toteutukseen, saadaan vielä huomattavasti enemmän hyötyjä irti.

LogiPlan-tiimissä toimivan teknisen testaajan roolia tuskin on kannattava muuttaa myöskään. Kun käytössä on vain yksi päätoiminen testaaja, on hyvä, että hänen työpanoksensa on manuaalisen testauksen puolella. Mikäli testaajia olisi useampia, tulisi harkita sellaisten automaatiotestaus-frameworkien käyttöönottoa, joihin myös ohjelmointitaidoton pystyisi testejä kirjoittamaan, kuten edellä mainittu Robot Framework. Myöskään lisätarvetta useammalle testaajalle ei juuri tällä hetkellä ole, sillä testiautomaatio hoitaa hyvin jatkuvan regressiotestauksen. Näin ollen tekninen testaaja pystyy keskittymään täysin uusien ominaisuuksien testaamiseen, sekä löytyneiden virheiden korjausten verifiointiin. Tarvittaessa manuaalista testausta tekee myös useampi henkilö, varsinaiset kehittäjät eivät kuitenkaan manuaalista testausta tee kuin omassa kehitysvaiheessaan.

4 Arviointi

Testiautomaatio tekniikoita on lukematon määrä. Niiden valinta ei ole lainkaan yksinkertaista vaan niihin vaikuttavat useat seikat, kuten koodauskieli, lähdekoodin avoimuus, projektin tyyppi ja niin edelleen. Tämän takia opinnäytetyön aihe ja rajaus muuttuivat työn edetessä useasti. Huomasin hyvin nopeasti, että koko testiautomaation kattaminen olisi valtavan laaja aihe, joten rajasin aluetta web-ohjelmistojen käyttöliittymän testaamiseen. Tässä vaiheessa sain myös työharjoittelupaikkani mukaan toimeksiantajaksi, mikä oli erittäin positiivinen asia molempien kannalta.

Tämän opinnäytetyön tavoitteena oli selvittää parhaimmat työkalut sekä käytännöt web-ohjelmiston käyttöliittymän testaamiseen, sekä hieman sivuta manuaalisen testaajan rooliin tapahtuvia muutoksia. Näitä tuloksia oli sitten tarkoitus verrata Trimicon LogiPlan-tiimissä käytössä oleviin työkaluihin sekä käytäntöihin. Tavoitteeseen päästiin mainiosti, työkaluista ja käytännöistä löytyi hyvin tietoa web-lähteistä, jotka olivat hyvin ajankohtaisia ja niistä saatiin koottua tarvittavat tiedot tietoperustaa varten. Tämän jälkeen tietoperustaan koottuja tietoja olikin suhteellisen helppoa verrata LogiPlan-kehitystiimissä käytössä oleviin. Tuloksena saatiin, että Trimicolla ovat asiat testiautomaation suhteen hyvällä mallilla, mutta myös muutama pieni parannusehdotus löydettiin, mikä oli ehdottoman positiivinen asia opinnäytetyön hyödyllisyyden sekä onnistumisen kannalta.

Opinnäytetyön tietoperusta vaiheessa löydetyt tulokset, niin työkaluista kuin käytännöistäkin, ovat hyvin ajankohtaisia ja ne pätevät myös muiden yritysten testiautomaatioratkaisuihin. Testiautomaatio yleisestikin on ohjelmistokehityksessä hyvin ajankohtainen aihe ja siihen tullaan tulevaisuudessa panostamaan entistäkin enemmän.

Tämän opinnäytetyön tutkimukselle looginen jatko on parannusehdotusten analysoiminen Trimicon päässä, sekä mikäli ne koetaan aiheellisiksi, niiden toteutus. Myös opinnäytetyön kirjoittaja saattaa olla mukana muutosten toteuttamisessa tulevaisuudessa, mutta siitä ovat neuvottelut vielä kesken.

Tämä työ oli sen tekijälle hyvä oppimiskokemus, paljon uutta tietoa testiautomaatiosta, siihen liittyvistä työkaluista ja käytännöistä sekä testiautomaation tärkeydestä nii nyt kuin myös tulevaisuudessa. Myös projektinhallinnallisesti tämä työ opetti sen tekijälle paljon uutta, etenkin suunnitelmallisuuden sekä aikatauluttamisen tärkeydestä. Uskon näistä

tiedoista olevan apua joskus tulevaisuudessa työelämässä. Lisäksi opinnäytetyön kautta saadut uudet kontaktit alan ihmisiin ovat erittäin arvokkaita.

Tutkimuksen toteuttaminen sähköpostihaastatteluin toimi työssä mielestäni hyvin, vastaajat saivat ottaa oman aikansa vastauksien antamiseen, sekä miettiä niitä rauhassa. Jatkokysymysten esittäminen oli helppoa ja niille ilmenikin tarvetta lähes jokaisessa haastattelussa. Lisäksi sähköpostitse saadut vastaukset ovat helposti dokumentoitavissa suoraan liitteisiin.

Opinnäytetyössä kuvatun tehtävän projektinhallinnassa onnistuttiin suhteellisen hyvin siihen nähden, että kyseessä oli tekijän ensimmäinen useamman kuukauden kestävä henkilökohtainen projekti. Vaikka välillä olikin kausia, jolloin työ eteni hitaampaan tahtiin, pysyttiin alkuperäisessä aikataulussa hyvin.

Lähteet

Cloudbees, 2016. About Jenkins. Luettavissa: https://www.cloudbees.com/jenkins/about Luettu: 29.03.2017.

Django Stars, 2017. Continuous Integration. CircleCI vs Travis CI vs Jenkins. Luettavissa: https://hackernoon.com/continuous-integration-circleci-vs-travis-ci-vs-jenkins-41a1c2bd95f5#.or09rkpqt Luettu: 23.03.2017.

Eficode, 2016. Testiautomaatio-opas. Haettu: 20.02.2017 osoitteesta: http://lataa-testiautomaatio-opas.instapage.com/

Fowler, M. 2006. Continuous Integration. Luettavissa: https://martinfowler.com/articles/continuousIntegration.html Luettu: 21.03.2017.

Guberna, D. 2017. Re: Questions about testautomation, sähköpostiviesti. Vastaanottaja Marko Nenonen. Lähetetty: 16.03.2017. Viitattu: 20.03.2017.

Guru99, 2015. Introductrion to Selenium IDE Luettavissa: http://www.guru99.com/introduction-selenuim-ide.html Luettu: 27.02.2017.

Guru99, 2017. Installing Selenium WebDriver Luettavissa: http://www.guru99.com/installing-selenium-webdriver.html Luettu 21.02.2017.

Hellas, A. 2011. Yksikkötestaus luentomateriaali. Luettavissa: https://www.cs.helsinki.fi/u/avihavai/edutainment/2011/ohma/7-yksikkotestaamisesta.pdf Luettu: 05.04.2017.

McCarthy, T. 2016. Test Automation Concepts – Parallel test execution. Luettavissa: https://opencredo.com/test-automation-concepts-parallel-test-execution/ Luettu: 22.03.2017.

Ohjelmistotuotanto, Luento 7. 2012. Tietojenkäsittelytieteen laitos. Luentosarja testauksesta ketterissä menetelmissä. Luettavissa: https://www.cs.helsinki.fi/group/java/k12-ohtu/luento7.pdf Luettu: 05.04.2017.

Pesonen, T. 2016. Is manual testing dead? Luettavissa: https://www.valagroup.com/fi/2016/11/manual-testing-dead/ Luettu: 29.03.2017.

Pesonen, T. 2017. Re: Kysymyksiä testiautomaatiosta, sähköpostiviesti. Vastaanottaja Marko Nenonen. Lähetetty 16.03.2017. Viitattu: 20.03.2017.

Robot Framework, 2017. Robot Framework kotisivu. Luettavissa http://robotframework.org/ Luettu: 22.02.2017.

Selenium, 2017. Seleniumin kotisivu. Luettavissa http://www.seleniumhq.org/ Luettu: 21.02.2017.

Smartbear, 2016. Why automated testing? Luettavissa: https://smartbear.com/learn/automated-testing/ Luettu: 20.02.2017.

Smarteducation, 2016. Testauksen tasot. Luettavissa: http://smarteducation.jyu.fi/projektit/systech/Periaatteet/suunnittelun-periaatteet/testaus/testauksen-tasot Luettu: 05.04.2017.

Softwaretestinghelp, 2017. Top 15 Performance Testing Tools of 2017: Comprehensivve Load Testing Tools List. Luettavissa: http://www.softwaretestinghelp.com/performance-testing-tools-load-testing-tools/ Luettu: 05.04.2017.

TechBeacon, 2016. 6 top open-source testing automation frameworks: How to choose? Luettavissa: https://techbeacon.com/6-top-open-source-testing-automation-frameworks-how-choose Luettu 22.02.2017.

Vogella, 2016. Unit Testing with Junit – Tutorial. Luettavissa: http://www.vogella.com/tutorials/JUnit/article.html Luettu: 05.04.2017.

Waller, J. 2014. Jenkins vs Travis. Luettavissa: https://thoughts.wallproductions.com/2014/01/jenkins-vs-travis/ Luettu: 27.03.2017.

Yarn, J. 2016. Selenium IDE: The good, the bad and the ugly. Luettavissa: https://www.lucidchart.com/techblog/2016/09/13/selenium-ide-the-good-the-bad-and-the-ugly/ Luettu: 27.02.2017.

Liitteet

Liite 1. (luottamuksellinen) Sähköpostihaastattelu Teemu Pesonen, VALA Group Oy

Liite 2. (luottamuksellinen) Sähköpostihaastattelu Dragos Guberna, VALA Group Oy

Liite 3. (luottamuksellinen) Sähköpostihaastattelu ohjelmistokehittäjä, Trimico Oy