Tuomo Salmi

Testiautomaatio ja Robot Framework

Metropolia Ammattikorkeakoulu

Insinööri (AMK)

Tieto- ja viestintätekniikka

Insinöörityö

30.10.2018

Metropolia

Tekijä Otsikko Sivumäärä Aika	Tuomo Salmi Testiautomaatio ja Robot Framework 33 sivua 30.10.2018
Tutkinto	insinööri (AMK)
Tutkinto-ohjelma	Tieto- ja viestintätekniikka
Ammatillinen pääaine	Ohjelmistotuotanto
Ohjaajat	Lehtori Simo Silander Managing Consultant Tuomas Peurakoski

Tämän insinöörityön tavoitteena on toteuttaa regressiotestiuudistus, jossa siirretään vanhoja Javalla tehtyjä testejä nykypäivään Robot Frameworkille. Työ toteutettiin työskennellessäni Sogeti Finland Oy:ssä. Aluksi kerron teoriaa ja taustoja työhön, jonka jälkeen kerron projektista, jonka toteutimme asiakastyönä keväällä ja kesällä 2018.

Teoriaosuudessa käydään ensin läpi testauksen pääpiirteitä, joihin sisältyy testaamisesta saatavat hyödyt ja miksi sitä tehdään sekä automaattisen ja manuaalisen testaamisen eroja. Tämän jälkeen käydään läpi testien kattavuutta ja sitä, mitä ovat testaamisen eri vaiheet sekä mitä eri tekniikkoja voidaan käyttää testaamisessa.

Robot Framework -osuudessa keskitytään siihen, kuinka Robot Frameworkilla syntaksi toimii sekä siihen, miten sillä luodaan testejä. Samaan osuuteen sisältyy projektivaiheessakin olellisena osana oleva web-sivujen testaaminen, joka toteutetaan Selenium2Libraryn avulla.

Avainsanat	testiautomaatio, robot framework, testaus

Metropolia

Author Title Number of Pages Date	Tuomo Salmi Testiautomaatio ja Robot Framework 33 pages 30. October 2018
Degree	Bachelor of Engineering
Degree Programme	Information and Communications technology
Professional Major	Software Engineering
Instructors	Lecturer Simo Silander Managing Consultant Tuomas Peurakoski

The goal of this thesis is to do a regression test renewal where I transfer old Java-tests to Robot Framework. The work was done while I was working at Sogeti Finland Oy. In the beginning I'm going to tell about theory and give background to the subject, after which I'm going to tell about a project that we delivered to a customer during the spring and summer of 2018.

In the theory part I am first going to go through the main features of testing. That includes knowledge about benefits that are gained from testing, why is testing done and differences between automatic and manual testing. After that I will write about test coverage, what are the different phases of testing and which different techniques can be used in testing.

In the Robot Framework-part I focus in how Robot Framework's syntax works and how one does testing with it. In the same part is included webpage testing which is also an essential part of the project phase. The webpage testing is done using Selenium2Library.

Keywords	test automation, robot framework, testing

Sisältö

Lyhenteet

1 Johdanto 1

2 Testaus pääpiirteissään 1

2.1 Miksi testata? 2

2.2 Manuaalinen testaaminen 3

2.3 Automaattinen testaaminen 3

2.3.1 Automaattisen testaamisen edut 3

2.3.2 Mitä automatisoida 5

2.4 Testien kattavuus 5

2.4.1 Prosessi 6

2.4.2 Päätöspisteet 7

2.4.3 Data 8

2.4.4 Ulkoasu 9

3 Testaamisen tasot 9

3.1 Yksikkötestaus 9

3.2 Integraatiotestaus 10

3.3 Järjestelmätestaus 11

3.4 Hyväksymistestaus 11

4 Testaustapoja 12

4.1 Black-box-testaus 12

4.2 White-box-testaus 13

4.3 Gray-box-testaus 14

5 Robot Framework 15

5.1 Avainsanat 15

5.2 Testien luominen 16

5.3 Webbisivutestaus 18

5.4 Hyväksymistestivetoinen ohjelmistokehitys 20

6 Käytännön toteutus asiakkaan projektissa 21

6.1 Alkutoimenpiteet ja ympäristön asentaminen 21
6.1.1 Pilvityöasema 21
6.1.2 Pythonin asentaminen 22
6.1.3 pip 23
6.1.4 Robot Framework 24
6.1.5 RIDE 24
6.1.6 Projektinhallinta Jira:ssa 24
6.1.7 git 24
6.1.8 Muut asennettavat ohjelmat 25
6.2 Testien toteuttaminen 25
6.3 Jenkins-ajojen konfiguroiminen 28
6.4 Tulosten tulkinta 29
6.5 Dokumentaatio ja palaute 31
7 Yhteenveto ja pohdinta 32
Lähteet 34

Lyhenteet

ATDD	Acceptance test-driven development. Hyväksymistestivetoinen ohjelmistokehitys.
DCOT	Data Combination Testing. Datayhdistelmätestaustekniikka, jolla testataan ohjelman toiminnallisuutta.
HTML	Hypertext Markup Language. Standardoitu kuvauskieli, jolla voidaan luoda internetsivuja.
RF	Robot Framework. Python-pohjainen kehys testien kirjoittamiseen.
ROI	Return of Investment. Sijoitetun pääoman tuottoaste.
TDD	Test-driven development. Testivetoinen ohjelmistokehitys, jossa luodaan yksikkötestit ennen ohjelman koodia.

1 Johdanto

Tämän insinöörityön aiheena on automaatiotestaus sekä Robot Frameworkin käyttö automaatiotestauksessa. Aloitin työni Sogeti Finland Oy:ssä helmikuussa 2018 ja tulen kertomaan työssäni asioista, joita olen koulutuksessani oppinut. Sogeti Finland tarjoaa Suomessa IT-palveluita testauksen, laadunvalvonnan sekä kyberturvallisuuden parissa. Maailmanlaajuisesti Sogeti-konsernilla on työntekijöitä noin 26000 monissa Euroopan maissa, Yhdysvalloissa sekä Intiassa.

Insinöörityön tavoitteena on oppia testaamisen perusteita sekä käyttämään työssäni Robot Frameworkia työkaluna automaatiotestaukseen. Suurimpana oppimisen lähteenä työssäni tulee olemaan Sogetin järjestämä Test Automation Academy, jossa koulutetaan automaatiotestausta. Kaikkia insinöörityössäni käsittelemiäni asioita on käyty läpi edellä mainitussa koulutuksessa. Tämän lisäksi aion syventää tietämystäni etsimällä lisätietoa verkkojulkaisuista sekä kirjoista.

Aloittaessani en tiennyt testaamisesta vielä paljoakaan, mutta nopeasti edenneen koulutuksen myötä pääsin jo pian ensimmäiseen projektiini mukaan. Tässä työssä kerron juuri tuosta ensimmäisestä projektista, jossa olin mukana. Asiakkaan nimeä ei voida mainita tietoturvasyistä. Projektissa toteutetaan automaattisia testejä web-selainpohjaiselle käyttöliittymälle Robot Frameworkilla.

2 Testaus pääpiirteissään

Ohjelmiston testaaminen on tärkeää ja iso asia tuotteen kehitystä, mutta se voi olla myös asia, josta halutaan karsia, jos esimerkiksi budjetti tai aika tulee vastaan. Mitä siis tapahtuu, jos testaaminen laiminlyödään tai tehdään puolivillaisesti? Pahimmassa tapauksessa siitä voi koitua muun muassa mittavia taloudellisia vahinkoja yritykselle ja huonoa julkisuutta.

Hyvänä esimerkkinä puutteellisesta testaamisesta voidaan pitää Ariane 5 -raketin laukaisua vuonna 1996, kun raketin hallittavuus menetettiin sen ajautuessa lentämään väärään suuntaan ja se jouduttiin räjäyttämään vain noin 40 sekuntia lähdöstä. Tähän katastrofaaliseen tilanteeseen johti yksinkertainen ohjelmistovirhe, sillä Ariane 5 -

raketissa käytettiin samaa ohjelmistoa, kuin sitä edeltäneessä Ariane 4:ssä. Kävi ilmi, että epäonnistumisen syy oli ohjelmistovirhe inertiaalisuunnistusjärjestelmässä. Tarkemmin kerrottuna syynä oli 64-bittinen liukuluku, joka liittyy raketin horisontaaliseen nopeuteen suhteessa alustaan, ja tuo luku muunnettiin 16-bittiseksi kokonaisluvuksi. Numero oli suurempi kuin 32767, joka on suurin tallennettava 16-bittinen kokonaisluku, joten muuntaminen epäonnistui. [1.]

Testaaminen oli siis Arianen tapauksessa toteutettu huolimattomasti tai mahdollisesti jopa ei ollenkaan, ja hinta siitä oli erittäin suuri. Miljardeja kehitystyötä sekä miljoonien arvosta mukana ollutta rahtia räjähti taivaalle. Henkilövahingoilta kuitenkin säästyttiin laukaisun ollessa miehittämätön.

2.1 Miksi testata?

Ohjelmistotestauksessa sen nimen mukaisesti testataan jotain ohjelmaa, ohjelmistoa, nettisivua, tietokantaa jne. Mutta mitä hyötyä siitä on? Ohjelmiston kehittäjäthän ovat jo testanneet tekemäänsä koodia esimerkiksi yksikkötestein, joten eikö se silloin ole valmis ja testattu jo? Harvemmin näin on.

Ensisijaisesti useammalle tulee tietysti mieleen virheiden löytäminen ohjelmasta. Tämä on toki tärkeää, sillä silloinhan virheet tulevat tietoon, ne kirjataan ylös ja tieto niistä menee ohjelmiston tekijöille, jolloin he voivat korjata kyseiset virheet. Mutta tärkein asia, jota testaamisella pohjimmiltaan haetaan, on laadun osoittaminen testatussa tuotteessa; kukaan ei halua julkaista markkinoille tai asiakkaalleen huonolaatuista ohjelmistoa. Tuotteen tulee vastata siihen ennalta asetettuihin tavoitteisiin ja vaatimuksiin, ja testaamalla saadaan nuo asiat selville. Kun virheet löydetään ajoissa, vähentää se vahinkojen ja kulujen määrää verrattaen siihen, että se löytyisi vasta tuotannossa. Testaaminen tuo uskottavuutta ja luotettavuutta testattavaan ohjelmistoon. Siinä missä laatu on karkeasti määritelty olevan ”vaatimusten ja odotusten saavuttaminen”, antaa testaus tietoa laadusta. Se tarjoaa näkemystä esimerkiksi riskeihin, jotka ovat mukana, kun hyväksytään heikompaa laatua. Tieto laadusta ja näkemys riskeihin ovat testauksen päätavoitteita. [2.]

2.2 Manuaalinen testaaminen

Manuaalisessa testaamisessa testataan siten, että testaaja käy läpi testin vaiheet itse manuaalisesti ilman automaatiotyökaluja. Siinä testaajan on ajateltava loppukäyttäjän näkökulmasta käydessään läpi ohjelman eri osia ja ennalta suunniteltuja testejä [3]. Tarkoituksena on löytää virheitä ja bugeja, jonka jälkeen ne raportoidaan eteenpäin kehittäjille.

Manuaalista testaamista on tehty jo vuosia, ja on totta, että automaatio ei voi kokonaan korjata manuaalista testaamista, eikä täysin automoitu prosessi ole edes mahdollista toteuttaa. On myös alueita, joissa manuaalinen testaaminen on parempi vaihtoehto. Esimerkiksi ohjelman satunnainen läpikäyminen saattaa tuottaa varsinkin kokeneelta testaajalta nopeita tuloksia. Parasta manuaalinen testaus on tilanteisiin, jossa samaa asiaa ei tarvitse testata montaa kertaa putkeen tai kovin usein, tutkivaan testaamiseen sekä käytettävyyden testaamiseen [4]. Mutta manuaalinen testaaminen vie aikaa ja on mahdollisesti jopa yksitoikkoisen puuduttavaa, mikä voi altistaa testaajan olemaan huolimaton virheiden havaitsemisessa. Paljon aikaa viedessään se on myös kallista.

2.3 Automaattinen testaaminen

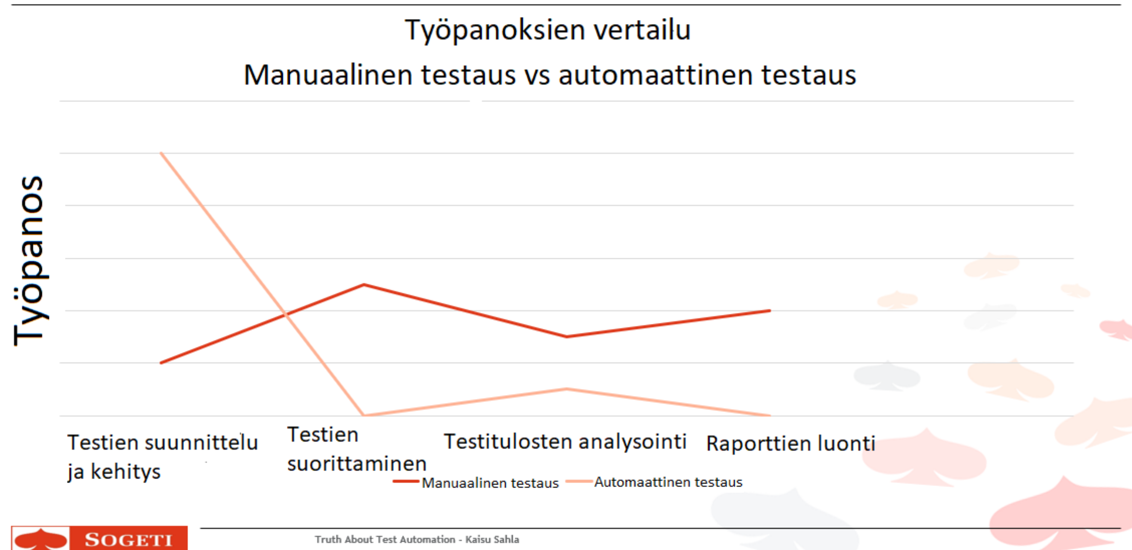
Automaattisessa testauksessa, vastoin kuin manuaalisessa, testaaja käyttää automaatiotyökaluja, ohjelmia ja skriptejä, jotka tekevät ohjelman testaamisen ilman käyttäjän syötteitä. Periaatteessa voidaan sanoa, että automaattista testausta on kaikki testaaminen, jossa käytetään jotain ohjelmaa apuna. Sillä voidaan suorittaa muun muassa yksikkötestausta, regressiotestausta, suorituskykytestausta ja tietoturvatestausta.

2.3.1 Automaattisen testaamisen edut

Automoitu testi tekee jokaisella ajolla varmasti saman asian, kuin se teki edellisellläkin kerralla. Manuaalisessa taas on aina olemassa mahdollisuus inhimillisiin virheisiin esimerkiksi väsymyksen tai keskittymisen herpaantumisen vuoksi. Jos testejä halutaan ajaa usein, on automaatio tähän erinomainen ratkaisu. Testejä voidaan pyörittää hyvin vähäisillä resursseilla vaikka ympäri vuorokauden, kun taas manuaalinen testaaja tekee päivän aikana testausta kolmasosan vuorokaudesta. Automatisoitaessa voidaan

automoida jo manuaalisesti tehtyjä testejä, tai toteuttaa testejä, joita ei ole mahdollista tehdä manuaalisesti [5].

Automaattinen testaus mahdollistaa myös jatkuvan integraation, jatkuvan toimittamisen sekä jatkuvan käyttöönoton. Jatkuvassa integraatiossa automatisoidut testit suoritetaan aina, kun versionhallintaan on tullut muutoksia. Jatkuvassa toimittamisessa varmistetaan, että koodi on nopeasti ja turvallisesti otettavissa käyttöön toimittamalla jokainen muutos tuotannon tapaiseen ympäristöön ja varmistamalla tiukalla automatisoidulla testauksella, että liiketoimintasovellukset ja -palvelut toimivat odotetusti [6]. Lisäksi pidemmissä ja isommissa projekteissa automaatiolla voidaan saada aikaan korkeampi ROI. Alkuperäinen sijoitus on suurempi kuin manuaaliseen testaukseen lähdettäessä, mutta kulut eivät enää kasva suhteessa niin paljon kuin manuaalisessa testauksessa ja takaisin saatu arvo ja hyöty on pitkäaikaista. Tämä myös tarkoittaa, että automatisointiin tarvitaan todennäköisesti vähemmän testaajia, kuin manuaaliseen testausprojektiin; yksi testaaja voi kirjoittaa useaan kohteeseen testejä samalla, kun suoritetaan muita testejä.



Kuva 1. Havainnollistava kuva työpanoksen määrästä projektin eri vaiheissa manuaali- ja automaatioprojekteissa [5].

Kuvassa 1 nähdään, että automaattisessa testauksessa vaadittava alkupanos on suurempi kuin manuaalisessa testaamiessa. Tämän jälkeen vaadittavan työpanoksen määrä kuitenkin laskee manuaalista testausta alemmalle tasolle.

Automaatio ei poista testaajien tai manuaalisen testaamisen tarvetta. Parhaimmillaan automaatio sekä manuaalinen testaus täydentävät toisiaan.

2.3.2 Mitä automatisoida

Aivan kaikkea ei ole kannattavaa eikä järkevää automatisoida, joten on hyvä miettiä, mitä automatisoidaan. Ensimmäisiä automatisoituja testejä yksikkötestauksen jälkeen ovat niin kutsutut smoke-testit, jossa testataan uusimmalla koontiversiolla kriittisimmät perustoiminnot, joiden pohjalta voidaan päättää, tehdäänkö syvempää testausta. Tämä menetelmä takaa virheiden aikaisen löytyvyyden ja paljastaa osat, joissa virheitä eniten esiintyy.

Toinen yleinen automatisoitava asia ovat regressiotestit. Niissä testataan sovelluksen yleisimpiä ominaisuuksia, jotka tulevat pysymään siinä jatkossakin, joten niiden on täten toimittava joka kerta. Käytännössä tämä tarkoittaa sitä, että aina, kun ohjelmistoon tehdään muutoksia, niin regressiotestit kertovat, menikö uusimmassa muutoksessa mitään rikki vai toimivatko kaikki edelleen halutulla tavalla.

Kuten aiemmin mainittua, usein toistettavat testit ovat suurimpia automatisoitavia asioita. Paljon datan syöttämistä vaativissa testeissä (kirjautuminen, lomakkeiden täyttö jne.) on automatisoinnista paljon hyötyä, sillä se on paljon nopeampaa kuin manuaalisesti saman tekeminen. Tämä pätee myös testeihin, joissa lähetetään ohjelmalle useita eriä, jotka sisältävät paljon dataa. Lisäksi suorituskykytestaus on yksi automatisoitavia testauksen osa-alueita.

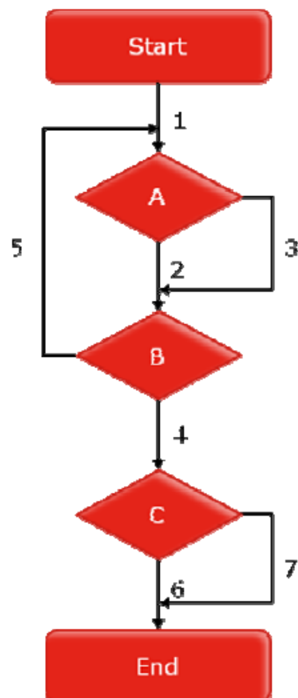
2.4 Testien kattavuus

Usein testaamista suunnitellessa pyritään mahdollisimman hyvään testikattavuuteen. Testikattavuudella tarkoitetaan testien kattavuutta ohjelman kokonaisuudesta. Tämä voi tarkoittaa esimerkiksi lauseita, funktioita, ehtoja tai polkuja. Yleinen harhaluulo on, että parasta olisi aina tähdätä 100 %:n kattavuuteen, mutta se ei pidä paikkaansa [7]. Täydellistä kattavuutta on hiemankaan isommissa projekteissa lähes mahdotonta toteuttaa, sillä se veisi paljon aikaa sekä rahaa. Olisi yksinkertaisesti liian työlästä käydä jokainen ohjelman mahdollinen kohta yksitellen läpi. Tätä varten on testien suunnitteluun

kehitetty tekniikoita, joita voidaan hyödyntää, kun halutaan hyvä kattavuus testeille. Ne voidaankin jakaa neljään eri kattavuusryhmään.

2.4.1 Prosessi

Prosessissa muodostetaan ohjelman tai prosessin kulusta prosessidiagrammi, josta katsotaan erilaisia mahdollisia polkuja, joita tapaus voi kulkea. Tätä kattavuustyyppiä kutsutaan poluksi, ja sen toteuttaminen on mahdollista, kun järjestelmä on kuvailtu ehdoin ja siitä on mahdollista muodostaa alusta loppuun kulkevia polkuja.



Kuva 2. Esimerkki piirrettävästä kaaviosta, jossa on kuvattu toiminto alusta loppuun [2].

Kuvalle 2 voidaan antaa esimerkkinä korttipelin tyyppinen laskemisohjelman tynkä. Alussa annetaan kahden kortin arvot. Kohdassa A tarkastetaan, onko ensimmäinen kortti yli 7. Jos on, niin ohjelma tulostaa ’Suuri’. Jos kortti on alle 7, niin ohjelma tulostaa ’Pieni’. Kohdassa B voidaan tarkistaa, onko annettu kummallekaan yli 14 arvoa, ja jos on, niin palataan alkuun syöttämään arvot. Kohdassa C tarkistetaan, ovatko arvot samoja, ja jos ovat, niin tulostetaan ’Pari’.

2.4.2 Päätöspisteet

Päätöspisteisiin sisältyvät ehtolauseet, kuten AND ja OR. Jokainen ohjelma sisältää niitä, ja ne määrittelevät, mihin suuntaan ohjelma seuraavaksi lähtee. Esimerkiksi, jos käyttäjä antaa syötteen X, mennään lopputulemaan Y. Näiden ehtojen pohjalta voidaan suunnitella esimerkiksi Condition coverage tai Decision coverage -mallin ratkaisu. Condition coveragessa jokaisen ehdon mahdollinen lopputulema käydään läpi. Tähän voidaan ottaa esimerkkinä tilanne, jossa lippuihin saa alennuksen, jos on opiskelija tai eläkeikäinen. Tällöin ehtolause on `IF opiskelija OR ikä >= 65 THEN alennus`. Jotta saadaan kaikki ehdot käsiteltyä, täytyy tehdä kaksi testiä. Toisessa testissä on tällöin opiskelija=true ja ikä<65, ja toisessa opiskelija=false ja ikä>65. Kummassakin edeltäneessä testissä lopputulema olisi sama, eli ne kummatkin johtaisivat alennukseen. Decision coveragessa taas jokaisen päätöksen lopputulema käydään läpi. Edelliseen esimerkkiin tämä saadaan toteutettua siten, että tehdään kaksi testitapausta; toisessa lopputulemaksi halutaan saada alennus ja toisessa ei alennusta. Lisäksi nämä kummatkin voidaan yhdistää esimerkiksi Condition / Decision coverage -mallin ratkaisuksi.

JOS vaurioiden määrä < 3 **JA** vaurioiden kokonaishinta < 1000€ **NIIN** alennus = 15%

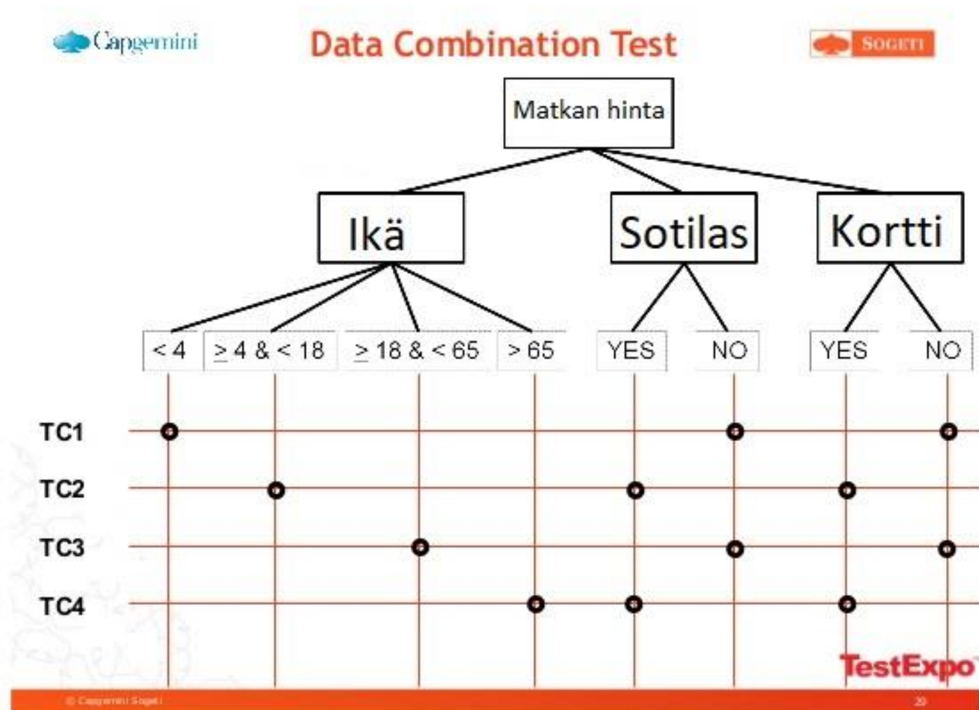
A **Vaurioiden määrä < 3**	B **Vaurioiden kokonaishinta < 1000€**	R **Lopputulos**
true	true	**15% alennus**
false	false	**Ei alennusta**

Kuva 3. Condition / Decision coverage -esimerkki [8].

Kuten kuvassa 3. näkyy, kannattaa tässä lähestymistyylissä laittaa taulukkoon ehdot (kuvassa A ja B) sekä lopputulos (kuvassa R). Esimerkissä käydään jokainen päätöksen lopputulos läpi, eli tuloksena saadaan sekä 15%:n alennus sekä ei alennusta ollenkaan. Lisäksi käydään molemmat ehdot, eli vähemmän kuin kolme vauriota sekä vaurioiden määrä alle 1000 €.

2.4.3 Data

Data-luokalla tarkoitetaan ohjelmalle syötettävää dataa. Nämä voivat olla esimerkiksi käyttäjän antamia arvoja. Tässä keskeisimmässä roolissa ovat raja-arvot sekä ekvivalenssiluokat. Data-luokasta hyvänä esimerkkinä voidaan ottaa Data combination Testing (DCot). Siinä valitaan ohjelmaan syötettävistä arvoista sellaisia, jotka ovat keskenään vuorovaikutuksessa sekä vaikuttavat ohjelman antamaan lopputulokseen. Tämän jälkeen niistä piirretään kaavio, josta saadaan pääteltyä tarvittavat testitapaukset.



Kuva 4. Esimerkki Data Combination Testin kaavasta [9].

Esimerkissä on tuotettu tapauksia, jossa toteutuu vähintään kaikki ehdot kertaalleen. Tämän jälkeen tulee kaavaa täyttää arvojen keskinäisillä yhdistelmillä. Tällöin halutaan lisätä yhdistelmiä, kuten ikä>65 ja sotilas=ei, ikä>=18&<65 ja sotilas=kyllä. Tästä syntyy yhteensä kahdeksan testitapausta, kun yhdistetään kaikki mahdolliset parit. Kaikki ehdot yhdistämällä (kuva 4) saadaan neljä testitapausta.

Mitä tyyliä kannattaa soveltaa mihinkin testiin, riippuu tarpeesta, kuinka syväluotaavaa testaamista halutaan. Esimerkiksi Data Combination -mallin jokaisen arvon täyttämällä

saadaan nopeasti käyttöön otettava tekniikka, joka ei kata aivan niin paljoa, ja jokaisella yhdistelmällä taas saadaan keskitason kattavuus [9].

2.4.4 Ulkoasu

Tässä kattavuusryhmässä käsitellään järjestelmän ei-funktionaalisia ominaisuuksia, kuten käyttäjäystävällisyyttä, suorituskykyä ja soveltuvuutta. Tässä vaiheessa voidaan tehdä muun muassa heuristista arviointia, jossa mitataan tuotteen käytettävyyttä. Heuristisessa arvioinnissa testaaja noudattaa usein muistilistaa, jonka pohjalta käy läpi ohjelman käytettävyyttä. Muistilistana voi käyttää esimerkiksi Jakob Nielsenin kehittämää heuristisen arvioinnin muistilistaa, johon sisältyy kymmenen tarkistettavaa asiaa. Niihin kuuluvat muun muassa järjestelmän hallinnan ja vapauden säilyminen käyttäjällä, mentaalisen kuormituksen minimointi sekä se, kuinka järjestelmä palautuu virhetilanteista. [10.]

3 Testaamisen tasot

3.1 Yksikkötestaus

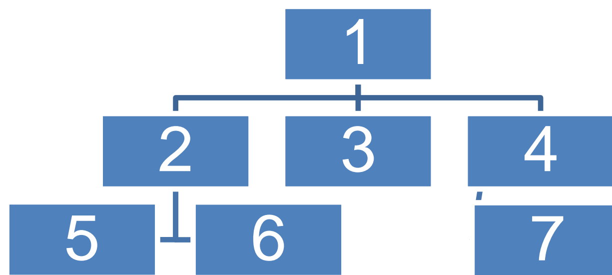
Yksikkötestaus on tuttua kaikille, jotka ovat ohjelmistojen kehitykseen koskaan tutustuneet. Siinä testataan yksittäin ohjelman pienimpiä osia, kuten funktioita, luokkia ja rajapintoja. Tarkoituksena on osoittaa eri osien toimivan oikealla tavalla. Usein yksikkötestit ovat samojen henkilöiden tekemiä, jotka ovat tehneet myös itse ohjelman koodin. Ensimmäisinä testeinä toimivat yksikkötestit ovat tärkeitä ohjelman kannalta, sillä niillä voi ennaltaehkäistä paljon tulevia ongelmia.

TDD eli Testivetoinen kehitys (eng. Test-driven development) on hyväksi todettu tekniikka, jossa testit tehdään ennen varsinaista ohjelman koodia ja vasta näiden pohjalta lähdetään kehittämään ohjelmaa. Testit toteutetaan siten, että ne epäonnistuvat aluksi ja koodi kirjoitetaan siten, että testit menisivät läpi. Tämä pakottaa kehittäjän ottamaan huomioon kaikki mahdolliset syötteet ja tulosteet. Kun ohjelmaan lisätään koodia testivetoisessa kehitystavassa, on valmiina monen testin paketti, joka voidaan suorittaa milloin tahansa, jotta saadaan selville, toimiiko lisätty osa. Tämä tarkoittaa täten myös sitä, että virheiden paikantaminen helpottuu huomattavasti. [11.]

3.2 Integraatiotestaus

Nimensä mukaisesti integraatiotestauksen pääperiaate on se, että siinä tuodaan yhteen aikaisemmin luotuja ohjelman osia. Tämän pohjalta koitetaan löytää virheitä, joita voi syntyä komponenttien liitoksissa. Aikaisemmassa vaiheessa ohjelman yksittäisiä komponentteja on testattu yksikkötestien muodossa, joten ei tiedetä, toimivatko ne yhdessä niin kuin on suunniteltu. Näin saadaan selville, että eri osat toimivat yhdessä sekä keskenään, että osana koko järjestelmää.

Integraatiotestauksessa voidaan käyttää eri lähestymistapoja, kun mietitään testauksen toteuttamista. Näitä ovat muun muassa bottom up, top down, big bang sekä sandwich -lähestymistavat. Bottom up -tavassa aloitetaan alimman tason toisista riippumattomattomista komponenteista, jotka testataan ensin ja tämän jälkeen siirrytään ylöspäin komponentteihin, jotka ovat riippuvaisia ensin testatusta.



Kuva 5. Bottom up -lähestymistapa.

Esimerkissä (kuva 5) edettäisiin siis siten, että ensiksi testattaisiin kohdat 5, 6, 7 ja 3, jonka jälkeen edettäisiin kohtiin 2 ja 4. Viimeisenä tulisi 1, sillä se on kaikkein aiempien komponenttien toimimisesta riippuvainen.

Top down -tavassa toimitaan päinvastoin kuin bottom up -tavassa. Siinä testataan ensimmäiseksi olennaisimman päätoiminnon toiminnallisuus, eli tämä vastaisi kuvassa 2 laatikkoa 1. Tällöin tarvitaan stubit, eli testityngät, kohdille 2, 3 ja 4, jotka esittävät tai simuloivat oikeiden komponenttien toimintaa. Myöhemmässä vaiheessa, kun siirrytään testaamaan kohtia 2 ja 4, tarvitaan ensinnäkin niiden oikeat komponentit sekä stubit

kohtiin 5 ja 6. Tämä lähestymistapa mahdollistaa esimerkiksi aikaisen prototyypin valmistumisen.

Big bang -tavassa nimensä mukaan kaikki järjestelmän komponentit tuodaan yhteen kerralla. Tämä tarkoittaa sitä, että testausvaiheessa kaikki komponentit ovat jo valmiita. Se sopii lähinnä pienempiin projekteihin, sillä esimerkiksi virheiden löytäminen ja niiden paikantaminen on vaikeampaa kuin tavoissa, joissa edetään komponentti kerrallaan. Kaikille komponenteille, joita pidetään eristyksissä ennen niiden integrointia, täytyy toteuttaa oma testausympäristö [15].

3.3 Järjestelmätestaus

Järjestelmätestauksella tarkoitetaan täysin valmiin ja integroidun ohjelmiston testaamista. Se ei kuitenkaan välttämättä tarkoita koko kokonaisuutta, kun puhutaan valmiista ohjelmistosta, sillä se voidaan liittää osaksi vielä suurempaa kokonaisuutta. Järjestelmätastaus onkin näiden vastaavien isojen integroitujen osien testaamista, jolloin täten varmistetaan suuren kokoonpanon toimivuus. Se kuuluu mustalaatikkotestauksen piiriin, sillä siinä tarkastellaan ohjelmaa käyttäjän näkökannalta. Mustalaatikkotestausta käsitellään enemmän luvussa 4.1.

Järjestelmätestauksessa on monia eri testauksen tyyppejä, joita käytetään prosessin aikana. Näihin kuuluvat muun muassa käytettävyystestaus, regressiotestaus, toiminnallisuuden testaus sekä rasitustestaus. Monista eri tavoista valitaan tarpeellisuuden, ajan sekä rahan pohjalta kullekin projektille sopiva tapa.

Järjestelmä voi koostua esimerkiksi mobiililaitteessa ajettavasta applikaatikosta, sekä erillisellä palvelimella pyörivästä serveriohjelmistosta. Testaaja käynnistää mobiiliapplikaatiosta jonkin toiminnan ja verifioi serveriohjelmistolta tulevan vastauksen. [16.]

3.4 Hyväksymistestaus

Hyväksymistestausvaiheessa tuotetta tarkastellaan siitä näkökulmasta, täyttääkö se kaikki sille määritetyt vaatimukset eli onko tuote sellainen, kuin on alun perin sovittu.

Tässä vaiheessa ei ole tarkoituksena enää löytää virheitä tai bugeja, sillä tuotteen tulisi olla jo valmis. Tämä vaihe käydään usein läpi yhdessä asiakkaan kanssa.

4 Testaustapoja

Testausmetodeja mietittäessä voidaan jakaa testaaminen pääpiirteissään kahteen tai kolmeen kategoriaan. Niitä ovat black-box, white-box sekä gray-box testing. Näistä kaksi ensin mainittua ovat keskeisimmässä asemassa olevat.

4.1 Black-box-testaus

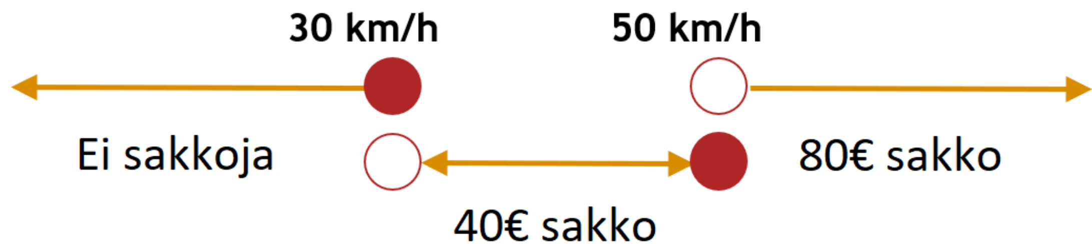
Black-box-testauksella (suom. mustalaatikkotestaus) tarkoitetaan ohjelman testaamista siten, että testaaja ei pääse käsiksi ohjelman sisältöön, kuten koodiin. Tällöin keskitytään pelkästään ohjelmalle annettaviin syötteisiin sekä ohjelman antamiin tuotoksiin. Testaaja siis tietää, miten ohjelman tulisi toimia, mutta ei sitä, miten se on toteutettu. Tämä on havainnollistettu kuvassa 6.

Kuva 6. Havainnollistava kuva mustalaatikkotestauksesta, jossa musta laatikko esittää sovellusta [17].

Saadun tulosteen jälkeen verrataan sitä odotettuun lopputulokseen ja täten päätellään, toimiiko sovellus, kuten on haluttu tai oletettu.

Yksi yleisimpiä metodeita, kun kehitetään mustalaatikkotestejä, on annettavien syötteiden jakaminen ekvivalenssiluokkiin. Tällöin mahdolliset syötteet jaetaan luokkiin siten, että niiden antamien tulosteiden voidaan olettaa olevan samankaltaisia. Tämän jälkeen jatketaan yhdistelemällä eri syötteitä, joista on luotu ekvivalenssiluokat, ja näin

syntyy testitapauksia. Jos ohjelmalla on paljon syötettäviä arvoja, voidaan käyttää boundary value -testausta eli raja-arvojen perusteella testausta. Tällöin määritellään raja-arvot ekvivalenssiluokkien välillä ja milloin ollaan siirrytty luokasta toiseen. Tällä metodilla saadaan rajattua testien lukumäärää paljon.

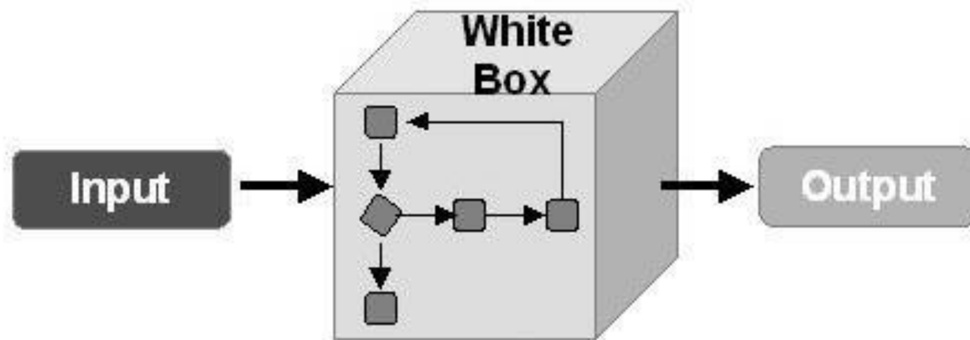


Kuva 7. Esimerkki raja-arvojen määrittämisestä [8].

Kuvassa 7 on havainnollistettu raja-arvojen määrittelyä, kun mietitään ylinopeussakkoja. Tällöin tarkistetaan ekvivalenssiluokkien arvoissa, joissa luokka vaihtuu, että ohjelma toimii oikein. Esimerkiksi halutaan, että autoilijan ajaessa 30 km/h ei tule sakkoa, mutta taas välillä 31-50 km/h tuleekin 40 €:n sakko. Kun testataan raja-arvoilla ja niiden viereisillä arvoilla, ja on todettu, että ne toimivat oikein, voidaan olettaa, että luokkien sisällä olevat arvot toimivat myös, eikä niitä tarvitse tällöin erikseen testata. Tämä vähentää huomattavasti testien määrää.

4.2 White-box-testaus

White-box-testaus (suom. lasilaatikkotestaus) on täysin päinvastainen kuin edellä käsitelty black-box-testaus. Siinä ohjelman testausta suunnitellaan ja toteutetaan siten, että testaajalla on pääsy sovelluksen lähdekoodiin. Kuvassa 8 on havainnollistettu tämä tilanne. Tämä tarkoittaa sitä, että näin voidaan toteuttaa perusteellisempaa testausta kuin mustalaatikkotestauksella. Siksi sen tavoitteena onkin päästä mahdollisimman suureen testikattavuuteen, eli tilanteeseen, jossa mahdollisimman moni toiminto olisi testattu.



Kuva 8. White-box-testauksen kaava [18].

Houkuttelevan kuuloisella white-box-testauksella on myös haittapuolensa. Se voi olla hyvinkin aikaavievää sekä kallista. Kun joudutaan palkkaamaan ohjelmointitaitoisia testaajia, jotka osaavat lukea ja tulkita koodia, nousevat kulut nopeasti korkeiksi. Lisäksi, jos koodikanta muuttuu usein, tulee automatisoiduista white-box-testeistä turhia, jolloin ne kaipaavat uudelleen tekemistä. [19.]

White-box-testauksessa tehdään testejä perustuen ohjelman lähdekoodiin, on testien suunnittelussa hyvä miettiä käytettäviä tekniikoita myös siltä kannalta. Esimerkiksi voidaan käyttää Statement coverage -metodia, jolloin tavoitteena on, että jokainen ohjelman lause käydään läpi ainakin kerran. Toinen metodi, jota voidaan käyttää, on Path coverage, jossa pyritään siihen, että kaikki ohjelman polut käydään läpi. Poluilla tarkoitetaan kaikkia ohjelman ehtolauseista syntyviä haaroja.

4.3 Gray-box-testaus

Gray-box-testaus on harvemmin käytetty termi, ja se jätetään usein mainitsematta, kun puhutaan laatikkomallista. Sen periaatteena on yhdistellä edellä mainittuja white-box- ja black-box-malleja. Siinä testaajalla on osittainen tieto ohjelman sisällöstä, kuten tietorakenteista sekä algoritmeista, testien suunnittelua varten mutta itse testaaminen tapahtuu mustalaatikkomallin mukaisesti. [20.]

Kuva 9. Gray-Box-testaus.

Gray-box-testausmallissa testaaminen tapahtuu niin sanotusti käyttäjän näkökulmasta.

5 Robot Framework

Automaatiotestaukseen on olemassa hyvin paljon työkaluja ja kehyksiä eri käyttötarkoituksiin. On olemassa kaupallisia ja hyvin hintavia sekä täysin ilmaisia ja avoimen lähdekoodin kehyksiä ja työkaluja. Robot Framework on näistä jälkimmäiseen kategoriaan kuuluva. Se on Python-pohjainen kehys, jonka kehittäminen alkoi alun perin vuonna 2005 Nokia Networksilla pohjautuen Pekka Klärckin diplomityön tutkimuksen aikana nousseeseen prototyyppiin. Vuonna 2008 se julkaistiin avoimen lähdekoodin ohjelmana kaikkien käytettäväksi. [22.] Nykyään se on suosittu työkalu testauksen saralla, ja monet yritykset Suomessa käyttävät sitä testeissään. Suurimpina näistä mainittakoon esimerkiksi Nokia, Finnair, Kone sekä ABB. Muualla maailmassa Robot Frameworkia käyttävistä yrityksistä mainittakoon muun muassa yhdysvaltalainen kaapeli-tv- sekä internet-yhteyksiä tarjoava Spectrum, Yhdysvaltain merivoimien tutkimuslaitos United States Naval Research Laboratory sekä venäläinen vakuutusyhtiö AlfaStrakhovanie Group. [23.]

5.1 Avainsanat

Robot Frameworkilla tehdyt testit perustuvat avainsanoihin eli keywordeihin. Tämä tarkoittaa sitä, että testeissä käytetään avainsanoja, joiden takana on itse logiikka. RF:lle on käytettävissä paljon valmiita sisäänrakennettuja sekä ladattavia kirjastoja, joissa on tarpeellisia avainsanoja. Käyttäjä voi myös itse halutessaan luoda omia avainsanoja ja kirjastoja Pythonilla tai Javalla.

```
import os
import csv
class CSV(object):
    def lue_csv_nimi(self, filedir):
        dir = os.listdir(filedir)
        for file in dir:
            if file.endswith('.csv'):
                return file

    def lue_csv_tiedosto(self, filename):
        data = []
        with open(filename, 'rb') as csvfile:
            reader = csv.reader(csvfile)
            for row in reader:
                data.append(row)
            return data
```

Esimerkkikoodi 1. Esimerkki omasta kirjastosta

Esimerkkikoodi 1:ssä on tehty kirjasto, jossa on kaksi avainsanaa: lue_csv_nimi ja lue_csv_tiedosto. Kirjaston käyttöönoton jälkeen niitä voi käyttää kutsumalla niitä nimellä, esimerkiksi:

```
*** Settings ***
Library    CSV

*** Test Cases ***
My Test
    ${tiedostonimi}=    Lue Csv Nimi    C:/downloads/
    ${tiedostodata}=    Lue Csv Tiedosto    C:/downloads/${tiedostonimi}
```

Esimerkkikoodi 2. Oman kirjaston ja avainsanojen käyttöönotto sekä käyttö.

Esimerkkikoodissa 2 luetaan ensin muuttujaan ${tiedostonimi} tiedoston nimi sille asetetusta polusta. Tämän jälkeen luetaan haetun nimen perusteella tiedoston sisältö muuttujaan ${tiedostodata}.

5.2 Testien luominen

Robot Frameworkissa data koostuu neljästä eri taulukosta, joiden sisään määritetään asetukset, muuttujat, testit ja avainsanat.

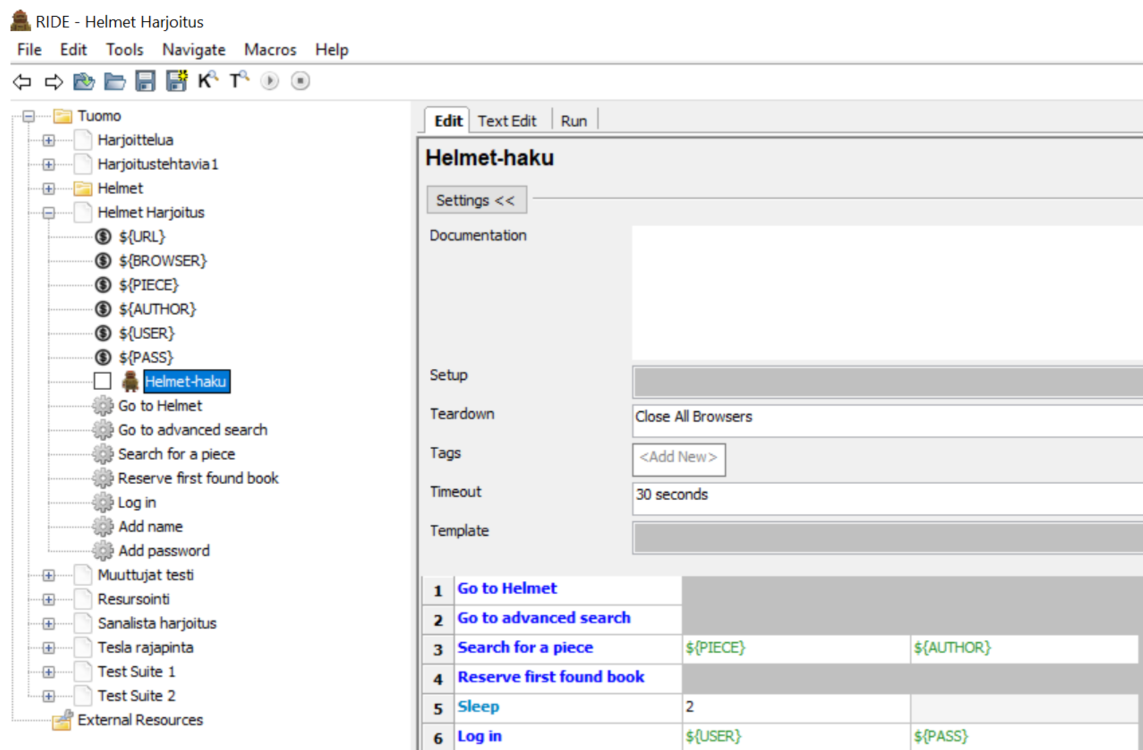
Taulukko 1. Eri tiedostotaulut Robot Frameworkissa [12].

Taulu	Käyttötarkoitus
Settings (asetukset)	1) Kirjastojen, resurssitiedostojen ja muuttujatiedostojen tuonti 2) Metadatan määrittäminen testisarjalle ja testeille
Variables (muuttujat)	Muuttujien määrittäminen
Test Cases (testit)	Testien luonti käytettävissä olevista avainsanoista
Keywords (avainsanat)	Avainsanojen luonti alemman tason avainsanoista

Settingsissä eli asetuksissa voidaan tuoda muuttujia, resurssitiedostoja sekä kirjastoja. Muuttujat listataan Variables-kohdan alle, testit Test Caseihin sekä avainsanat Keywordsien alle. Varsinkin jälkimmäisenä mainittu Keywords on erittäin hyödyllinen, sillä sinne voi testaaja luoda omia avainsanojaan muista avainsanoista. Esimerkiksi yksi käyttäjän määrittämä avainsana voi avata selaimen, kirjautua sisään ja suorittaa haun sivulla. Se on hyvä siinä mielessä, että testissä tämä näkyy yhtenä rivinä, jossa on avainsana ja argumentit, kun taas ilman käyttäjän tekemää avainsanaa siinä olisi useampi rivi koodia.

RF:llä testejä voidaan muokata plain text -muodossa helposti kaikilla tekstieditoreilla. Tässä muodossa se onkin suosituin tapa käyttää Robot Frameworkia. Aiemmin mainitut tiedostotaulut merkitaan plain text -muodossa kolmella tähdellä kumminkin puolin taulun nimeä: esim. *** Settings ***. Erotellessa avainsanoja niille annetuista argumenteista, käytetään välissä neljää välilyöntiä. Tämä käytäntö voi olla sekavaa, jos yhdellä avainsanalla on paljon argumentteja. Sen vuoksi on myös mahdollista erotella avainsanat ja argumentit toisistaan pystyviivoin. Esimerkkikoodi 2:ssa on käytetty välilyöntejä eroteltaessa avainsanoja ja argumentteja toisistaan. Plain textiä suosivalle riittää periaatteessa mikä tahansa teksiteditori, mutta myös monelle ohjelmointiympäristölle on omat Robot Framework -liitännäisensä. Muun muassa Eclipselle, PyCharmille sekä Atomille löytyvät liitännäiset. Ne toimivat lähinnä syntaksin korostamiseen.

Toinen vaihtoehto on käyttää robottitestien tekemiseen tehtyä editoria, joka tässä tapauksessa on RIDE.



Kuva 10. RIDE:n perusnäkymä, jossa yksi testi auki.

RIDE:ssä testit kirjoitetaan taulukon näköisiin riveihin, josta testit luetaan aina rivin ensimmäisestä solusta lähtien. Siinä saadaan myös määritettyä testitapausten lisäksi asetukset, muuttujat sekä avainsanat. Kuvassa 10 näkyvät käyttäjän itse tekemät avainsanat hammasrattaina, muuttujat dollarimerkkeinä ja testit punaisena hahmona. Asetukset saadaan laitettua oikeanpuolisesta Settings-painikkeesta. Esimerkiksi kirjastot saadaan liitettyä Test Suite -tasolla asetuksien kautta testeihin mukaan, jolloin niissä olevia avainsanoja voidaan käyttää.

5.3 Webbisivutestaus

Koulutusjakson aikana opiskelin paljon nettisivujen testausta RF:llä. Web-sovellukset ovat jatkuvasti kasvava sovellusten alue, kun yhä useampi sovellus päätetään toteuttaa web-pohjaisena. Tähän voidaan nähdä syynä esimerkiksi paremman yhteensopivuuden takaaminen ja vähäisemmän laitteiston suorituskyvyn vaatiminen verrattuna perinteiseen työpöytäsovellukseen. Tämä tarkoittaa myös sitä, että web-sovellusten testaaminen ja testien automatisointi onkin hyvin suuri osa-alue ohjelmistotestauksessa.

Selenium2Library

Koska testaukseni kohde oli web-sovellus, tarvitsin kirjaston, joka on tehty juuri tähän tarkoitukseen. Selenium2Library on web-sovellusten testausta varten tehty kirjasto Robot Frameworkille, ja se käyttää Selenium-työkalua. Selenium on yksi käytetyimmistä automaatiotyökaluista tällä hetkellä. Vuonna 2016 tehdyssä kyselyssään Zephyr raportoi, että kaikista haastatteluun vastanneista yhtiöistä 55 % kertoi käyttävänsä automaatiotyökalunaan Seleniumia. Täytyy tosin myös muistaa se, että suurin osa yrityksistä käyttää useampaa kuin yhtä työkalua, ja kyselyn mukaan näin tekee 75 % vastaajista. [13.]

Seleniumilla voidaan etsiä HTML-elementtejä sivulta ja näille voidaan tehdä toimenpiteitä. Voidaan myös tutkia, tapahtuuko odotettu lopputulos, kun tehdään jotain asiaa.

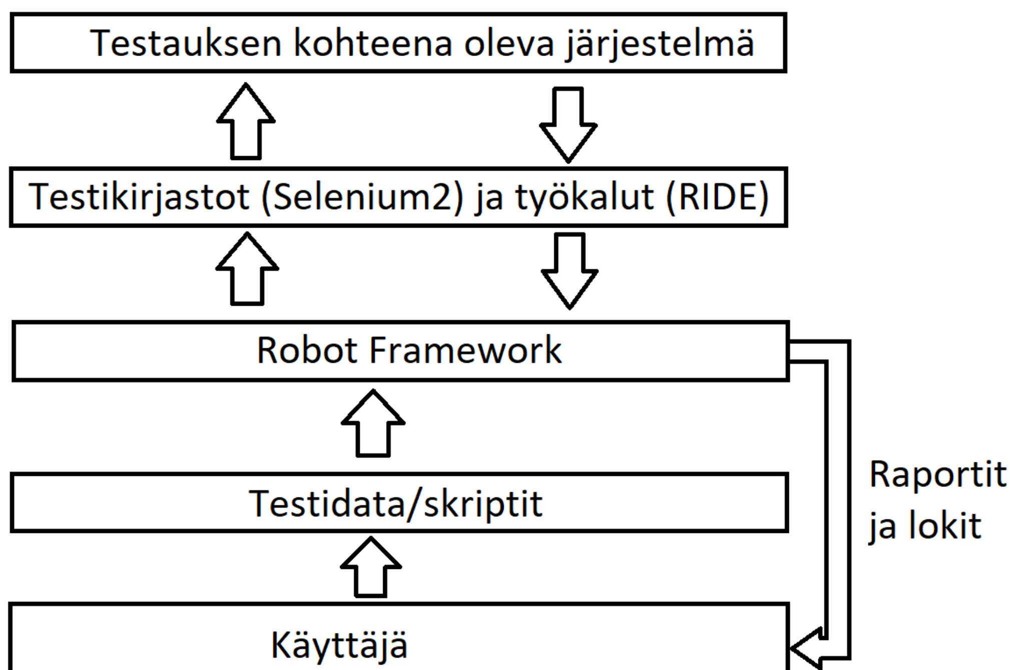
```
Avaa selain Helmet-osoitteeseen
    Open Browser    http://www.google.fi    chrome
    Maximize Browser Window
    Wait Until Page Contains Element    //li[@class="search"]//a
    Click Element    //li[@class="search"]//a
    Wait Until Page Contains Element    //input[@id="keyword"]
    Input Text    //input[@id="keyword"]    Tieto- ja viestintätekniikka
    Click Element    //button[@id="search-submit"]
    Wait Until Page Contains Element    //div[@class=”tx-indexedsearch-whatis”]
    Page Should Contain Element    //td[@class=”tx-indexedsearch-title ti-
tle”][1]//a[contains(text(),”Tieto- ja viestintätekniikka”)]
```

Esimerkkikoodi 3. Esimerkki Selenium2Libraryn avainsanoista. Avainsana sisennetyssä koodissa vasemmalla, joiden jälkeen niille annetut argumentit eroteltuna neljällä välilyönnillä.

Esimerkkikoodissa nähdään Seleniumin avainsanojen käyttöä ja HTML-elementtien hakemisesta. Selaimen avaamisen jälkeen odotetaan, että sivu sisältää jonkin halutun elementin (`Wait Until Page Contains Element`). Tätä käytetään siihen tarkoitukseen, että sivu on ladannut kyseisen elementin, jolloin sitä voidaan käyttää. Tässä esimerkissä odotettiin, että Metropolia.fi-sivuston haku-ominaisuus on käytettävissä, jolloin siihen syötettiin teksti ’Tieto- ja viestintätekniikka’ ja painettiin hakupainiketta (`Input Text` sekä `Click Element`). Tämän jälkeen odotettiin, että odotettu tulos on tullut näkyviin, mikä tässä tapauksessa on rivi, jossa on linkki otsikolla ’Tieto- ja viestintätekniikka’. Jos haluttu otsikkorivi löytyy, testi menee läpi ja muissa tapauksissa ei.

Seleniumia voidaan käyttää kaikilla suosituimmilla selaimilla, joka mahdollistaa monipuolisen testaamisen. Kullekin selaimelle on olemassa oma webdriverinsa, jonka

lataamalla saadaan Selenium toimimaan halutulla selaimella. Selenium-WebDriver tekee suoria pyyntöjä selaimelle käyttäen kunkin selaimen natiivia tukea automaatiolle. Kuinka nämä kutsut tehdään ja mitä ominaisuuksia ne tukevat, riippuvat selaimesta, jota käytetään. [14.]



Kuva 11. Testien kulku havaillistettuna.

Kuvassa 11. nähdään, että käyttäjä luo testiskriptit Robot Frameworkille, joka käyttää apunaan sen omia avainsanoja sekä ulkopuolisia kirjastoja, kuten Seleniumia. Kun järjestelmältä saadaan vastaus, palautuu se generoituna HTML-dokumenttina käyttäjälle.

5.4 Hyväksymistestivetoinen ohjelmistokehitys

Robot Framework on myös hyvin soveltuva hyväksymistestivetoiseen kehitykseen, jossa koko tiimi työskentelee yhdessä päästäkseen haluttuun lopputulokseen. Tämä perustuu asiakkaan, kehittäjien sekä testaajien väliseen kommunikaatioon. Kaikki lähtee siitä, että asiakas kertoo ensin, mitä hän haluaa tehdä tuotteella, mistä saadaan koostettua niin sanottuja User storyjä tai käyttäjätarinoita.

6 Käytännön toteutus asiakkaan projektissa

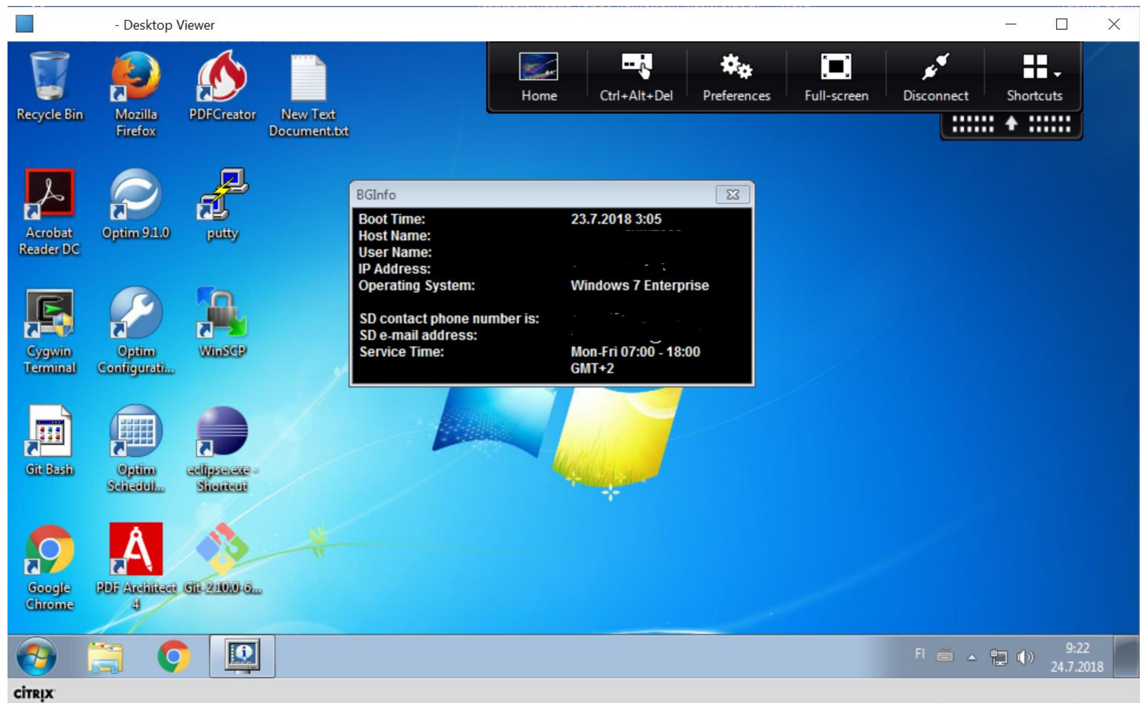
Projektin tavoitteena oli siirtää asiakkaan vanhoja automaatiotestejä nykypäivään Robot Frameworkille, kun ne aiemmin oli suoritettu Javalla. Tällöin testit ovat helpompilukuisia myös muille käyttäjille, sillä Robot Framework on avainsanapohjainen. Lisäksi testit ovat helpommin ylläpidettäviä ja hallitavia. Uusia testitapauksia tai testejä ei siis suunniteltu lainkaan, vaan yksinkertaisesti käytettiin vanhoja testejä pohjana. Kun testit olivat valmiina, niistä kirjoitettiin ylös parannuksia, muokkaamista tai poistamista vaativia asioita.

6.1 Alkutoimenpiteet ja ympäristön asentaminen

Ennen kuin töihin pääsee varsinaisesti käsiksi, on asennettava tietokone ja ympäristöt kuntoon. Asennuksissa tuli olla tarkkana, sillä versionumeroilla oli suurta merkitystä yhteentoimivuuden kannalta.

6.1.1 Pilvityöasema

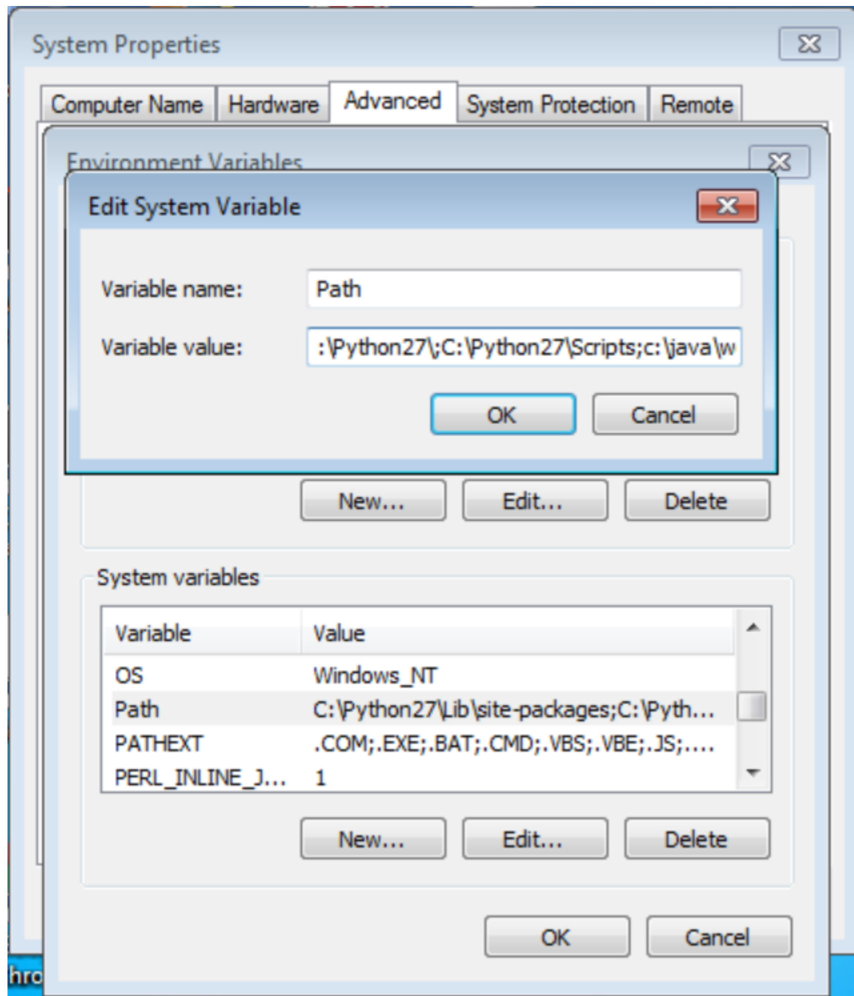
Tässä projektissa käytettiin asiakkaan pilvipalveluna tarjoamaa virtuaalityöasemaa, joten täysin uutta konetta ei tarvittu. Tällöin kehittäjät voivat käyttää palvelua omalta tietokoneeltaan käsin. Pilvipalveluna toimi Citrix-pilvipalvelu, johon oli asennettu työasemaksi virtuaalinen Windows 7 -käyttöjärjestelmä. Kirjautuminen työasemalle tapahtuu internetselaimen kautta, jossa työstetään henkilökohtaiset asiakkaalta saadut tunnukset. Kun kirjautuminen on vahvistettu mobiilivarmenteella, aukeaa Citrix StoreFront, josta etsitään Desktops-välilehti ja klikataan haluttua työasemaa, jolloin asema aukeaa käyttöä varten.



Kuva 12. Virtuaalikoneen työpöydän näkymä avattaessa.

6.1.2 Pythonin asentaminen

Robot Framework on pythonpohjainen, joten on ladattava python osoitteesta https://www.python.org/downloads/. Versioksi valitaan uusin 2.7 Python-versio, sillä ainakaan kirjottamisen hetkellä editoriksi valittu RIDE ei toimi Python 3:lla. Asennetaan Python C:n juureen ja lisätään C:\Python27\; sekä C:\Python27\Scripts; Windowsin ympäristömuuttujiin.



Kuva 13. Python Windows 7:n ympäristömuuttujissa.

6.1.3 pip

pip on paketinhallintajärjestelmä Pythonille, jolla voidaan ladata, päivittää ja poistaa Pythonilla kirjoitettuja ohjelmistoja. Sen pitäisi tulla Pythonin latauksen mukana valmiiksi asennettuna, mutta sen voi myös ladata itse osoitteesta https://bootstrap.pypa.io/get-pip.py, josta tallennetaan tiedosto get-pip.py. Tämän jälkeen avataan Windowsin komentorivi lataushakemistossa ja asennetaan pip komennolla `python get-pip.py`. Komennolla pip --version, voidaan tarkastaa nykyinen versio.

```
C:\Python27>pip --version
pip 9.0.3 from c:\python27\lib\site-packages (python 2.7)
```

Kuva 14. pip-version tarkastaminen.

pip tai joku muu paketinhallintajärjestelmä on hyvä ladata, jotta Python pakettien lataaminen ja asentaminen on helpompaa.

6.1.4 Robot Framework

Tärkeimmän osuuden eli Robot Frameworkin asentaminen tapahtuu pip-komennolla `pip install robotframework==x.x.x`. Perässä olevalla parametrilla x.x.x saadaan määritettyä haluttu ohjelmistoversio esimerkiksi ”robotframework==3.0.1”.

6.1.5 RIDE

RIDE on editori, joka on tarkoitettu Robot Frameworkin testien kirjoittamiseen. Sen lataaminen tapahtuu nopeasti pip:llä komennolla `pip install robotframework-ride`. Ennen tätä täytyy kuitenkin asentaa wxpython osoitteesta https://sourceforge.net/projects/wxpython/files/wxPython/, jotta RIDE toimii. Robot Frameworkille on olemassa myös muita editoreja, mutta tähän projektiin valittiin RIDE, sillä se oli entuudestaan tuttu, ja se on ollut käytössä myös muissa asiakkaan RF-projekteissa.

6.1.6 Projektinhallinta Jira:ssa

Sogetilla yleisesti käytössä oleva projektinhallintatyökalu Jira valittiin myös tähän projektiin käyttöön. Se on hyvä työkalu ketterään kehitykseen, missä voidaan luoda projekteja, backlogeja, sprinttejä, raportteja yms. Tässä kyseisessä projektissa ei kuitenkaan toteutettu kuin yksi iso sprintti, sillä asiakkaan kanssa oli sovittu, että heidän kanssaan käydään läpi sovitut järjestelmät kokonaisuudessaan, kun deadline on lähestymässä. Muutoin projektin manageri piti asiakasta tietoisena kehityksestä, minkä lisäksi oli myös Skypessä järjestettyjä kokouksia. Jira:ssa olleeseen sprinttiin kirjattiin kunkin testauksen alla olleen järjestelmän testit, josta me testien kehittäjät pystyimme seuraamaan, miten ne edistyvät.

6.1.7 git

Versionhallintaohjelmistona oli git-palvelu, johon päivitettiin uudet testit. Siellä oli myös entuudestaan vanhat Javalla tehdyt testit, joten ne otettiin sieltä omalle koneelle. RF-

testejä varten oli tehty tälle projektille uusi haara, jolloin työmme ei mennyt ristiriitaan muiden projektien kanssa.

6.1.8 Muut asennettavat ohjelmat

Yllämainittujen pakollisten ohjelmien lisäksi on hyvä asentaa Java JDK 1.8.1 ja jokin Java-editori, kuten esimerkiksi Eclipse. Tällöin vanhojen Javalle tehtyjen testien pyörittäminen virtuaalikoneella on mahdollista ja niitä on helppo käydä läpi. Lisäksi, kun kyseessä on internetselainpohjaisia testejä, on asennettava myös selaimet Internet Explorer 11 sekä Mozilla Firefox 45. Varsinkin Mozillan versiolla on merkitystä, sillä uusimmilla Mozilla-versioilla oli ongelmia toimia Robot Frameworkissa käytetyn Selenium2Libraryn kanssa. Tämän takia oli automaattiset päivityksetkin otettava pois Firefoxin asetuksista.

6.2 Testien toteuttaminen

Ennen kuin koodeja pääsi toteuttaman Robot Frameworkille, oli luonnollisesti vanhoja Javalla tehtyjä testejä käytävä läpi, jotta tiedettiin, mitä oltiin tekemässä. Perinteiseen testausprosessiin verrattuna tämä on helpohko lähestymistapa, sillä testit ovat jo olemassa ja tiedetään, mitä halutaan testata. Eli uusien testien suunnittelua ei tarvitse miettiä, sillä on jo tiedossa, mitä testataan. Paras tapa saada selville, mitä Java-koodit testaavat, oli ajaa testejä Eclipsellä ja katsoa, mitä ne tekevät. Tietysti oli myös mahdollista vain lukea koodia ja päätellä sen perusteella, mitä missäkin kohdassa koodia tapahtuu. Tämän lisäksi apuna oli vanha dokumentaatio, jossa oli muun muassa askel askeleelta testeiltä halutut toimenpiteet. Näiden pohjalta oli myös helppo lähteä toteuttamaan omia Robot Framework -koodeja.

#	Type	Description	Expected Results
1	Execution Step	Login to with a user that has rights to	Tee pyyntö screen opens up.
2	Execution Step	Give the following inputs * Lähtöympäristö: * Kohdeympäristö: * Kohdeympäristö ansaintatiedot: * sotkenta: Kyllä * tunnus: any valid * Toimintokoodi: U * Pyytäjä: * Pyytävä sovellus: * Vapaa teksti: 123456 Click Lisää Pyyntörivi button	A row containing the given information is added on the bottom of the page. Pyynnön yleiset valinnat section is not editable. Henkilötunnus input is cleared, all other information remains on the screen. Lähetä pyyntö button becomes active.
3	Execution Step	Click Lähetä pyyntö button.	System sends the request.

Kuva 15. Esimerkki olemassa olleesta dokumentaatiosta.

Osa dokumentaatiosta saattoi kuitenkin olla jo muutamia vuosia vanhaa, jolloin testauksen kohteena ollut ohjelmisto oli saattanut uudistua hieman tai sen käyttämä data muuttua, joten tällöin oli katsottava Java-koodeista, mitä niissä oli tehty erilailla kuin dokumentaatiossa. Kuvasta 15. nähdään, että vanha dokumentaatio oli hyvin vajavaista ja vaikeahkolukuista.

```
1 waitForPageToLoad(waitTime);
2 assertTrue(selectFromComboByText(getDynamicSelectid("Lähtöympäristö"),
getProperty("Lahtoymparisto", getFileName())));
3 assertTrue(selectFromComboByText(getDynamicSelectid("* Kohdeympäristö"),
getProperty("Kohdeymparista", getFileName())));
4 Thread.sleep(2000);
5 assertTrue(selectFromComboByValue(getDynamicSelectid("* Toimintokoodi"),
getProperty("ToimintokoodiValue", getFileName())));
6 Thread.sleep(3000);
7 assertTrue(selectFromComboByText(getDynamicSelectid("* Henkilötietojen sot-
kenta"), getProperty("Sotkenta", getFileName())));
8 Thread.sleep(1000);
9 assertTrue(selectFromComboByText(getDynamicSelectid("* Pyytäjä"), getProp-
erty("Pyytaja", getFileName())));
10 Thread.sleep(2000);
11 assertTrue(writeToTextField(getDynamicid("Vapaa teksti"), "test"));
12 Thread.sleep(1000);
13 assertTrue(selectFromComboByText(getDynamicSelectid("Pyytävä sovellus"),
getProperty("PyytavaSovellus", getFileName())));
```

Esimerkkikoodi 4. Ote Javalla tehdystä testistä.

Esimerkkikoodissa 4 valitaan sivulla olevista valikoista arvoja, jotka vastaavat resurssitiedostosta haettua dataa. Koodiesimerkissä rivillä 9 haetaan elementtiä, joka vastaa käyttöliittymässä 'Pyytäjän' valitsemista ja Java-tiedoston nimen perusteella haetaan saman nimistä resurssitiedostoa, joka sisältää kyseiseen kohtaan annettavan arvon. Lisäksi 11 rivillä täytetään tekstiä kenttään, jonka tekstiotsikkona on Vapaa teksti.

Kuva 16. Kuvakaappaus sovelluksen osasta, johon testi syöttää tietoja.

Kuvassa 16 nähdään muun muassa esimerkkikoodista selitetyt kohdat 'Pyytäjä' sekä 'Vapaa teksti', joihin syötetään resurssitiedostossa määritetyt arvot.

Javalla tehdyissä testeissä oli pyritty siihen, että yleisesti käytössä olleet funktiot olivat yhdessä tiedostossa, josta niitä kutsuttiin testeihin. Esimerkkinä tästä on Koodiesimerkki 1:ssäkin käytetty getDynamicId(), joka hakee sille tekstinä annetun otsikon html-labelia ja sen perusteella palauttaa dynaamisen elementin id-atribuutin, jolloin testi löytää kentän ja saa syötettyä siihen tekstiä tai valittua löydetystä valikosta oikean arvon. Myös Robot Frameworkilla tehdyissä testeissä pyritään siihen, että yleisesti käytössä olevat avainsanat ovat vain yhdessä tiedostossa. Lisäksi testeissä käytettävä data on laitettu muuttujiin yhteen tiedostoon, jolloin sen ylläpito on hyvin helppoa. Tämä tarkoittaa sitä, että jos tulevaisuudessa tarvitsee muuttaa dataa, käy se nopeasti vaikka suoraan tekstieditorilla.

3	Select Lähtöympäristö	${TC01_lahtoymparisto}
4	Select Kohdeympäristö	${TC01_kohdeymparisto}
5	Select Henkilötietojen sotkenta	${TC01_henkilotietojen_sotkenta}
6	Select Toimintokoodi	${TC01_toimintokoodi}
7	Select Pyytäjä	${TC01_pyytaja}
8	Select Pyytävä sovellus	${TC01_pyytava_sovellus}
9	Fill in Vapaa teksti	${TC01_vapaa_teksti}

Kuva 17. Vastaava toiminnallisuus kuin Java-koodissa (Koodiesimerkki 4) Robot Frameworkilla toteutettuna.

Kuten kuvasta 17 näkee, on Robot Frameworkilla tehty koodi on todella helppolukuista RIDE-editorissa avattuna. Logiikka on suurimmaksi osaksi piilotettuna testistä avainsanojen alle ja avainsanoille välitettävät tiedot ovat muuttujina.

```
Fill in Vapaa teksti
    [Arguments]    ${text}
    [Documentation]    Filling in free text in its textfield
    Wait Until Element Is Visible    ${vapaa_teksti}
    Log To Console    Typing Vapaa teksti ${text} in its input field
    Input text    ${vapaa_teksti}    ${text}
```

Esimerkkikoodi 5. Fill in Vapaa teksti -avainsana avattuna tekstimuotoon.

Esimerkkikoodissa nähdään, mitä avainsana sisältää. Se käyttää Selenium2Library-kirjaston tarjoamia avainsanoja. Ensin odotetaan, että haluttu elementti on näkyvillä, jonka jälkeen syötetään elementtiin argumenttina välitetty teksti ${text}, joka on siis määritetty muuttujaksi ${TC01_vapaa_teksti} resurssitiedostossa.

Robot Frameworkilla jokaisen testisarjan ylätason asetuksiin laitettu selaimen avaaminen poisti tarpeen tehdä se jokaisen testin kohdalla yksitellen. Sulkeminen tapahtuu, kun sarjan kaikki testit on ajettu läpi. Tällöin testien tasolla asetuksiin laitettiin vain sisäänkirjautuminen, joten sekään ei näy itse koodiriveissä editorissa. Tämä käytäntö vähentää testeistä toisteisuutta ja rivien määrää.

6.3 Jenkins-ajojen konfiguroiminen

Valmiit testit laitettiin suoritettaviksi avoimen lähdekoodin Jenkins-palvelun palvelimelle. Robot Framework -testeille oli asetettu yksi kone niin sanotuksi orjakoneeksi, joka ajoi

testejä ajastetusti öisin tai manuaalisesti, kun käyttäjä niin halusi. Tämä teki testausryhmälle sekä muille projektissa mukana oleville helpoksi testien kulloisenkin tilanteen seuraamisen.

Jokaiselle testauksen kohteena olleelle sovelluksen osalle luotiin Jenkinsiin oma ympäristö. Ne konfiguroitiin Jenkinsin pipeline-skriptissä siten, että ne ajoivat määrätyssä hakemistossa (RF-testien kanssa samassa) olleen gradle-skriptin, joka taas ajoi puolestaan testit. Konfiguraatiossa pystyttiin myös asettamaan url, joka halutaan avata testeissä sekä kirjautumistiedot. Tämä on hyvin tärkeää siksi, että testiympäristöissä eri tunnuksille on asetettu eri oikeuksia, joten täten on helppoa vaihtaa kirjautumistietoja Jenkinsin kautta.

```
parameters {
    string(name: "BRANCH_SELECTOR", defaultValue: "master", description: "The branch to build
    string(name: "URL", defaultValue: "http://testisivusto.com:8000/web-ui", description: "UR
    string(name: "YMPÄRISTÖ", defaultValue: "arekaix206e", description: "Environment where te
    string(name: "KÄYTTÄJÄNIMI", defaultValue: "admin", description: "Username of admin right
    string(name: "SALASANA", defaultValue: "Salasana123", description: "Password of admin rig
}
```

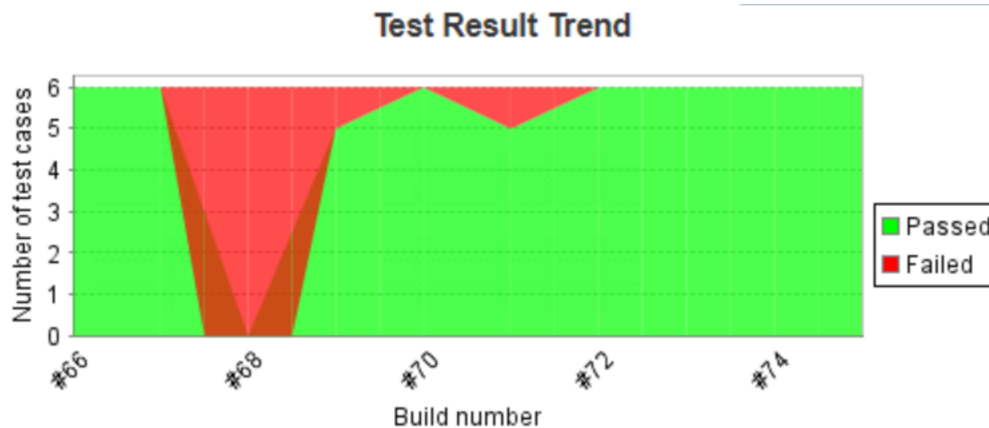
Kuva 18. Esimerkki kirjautumistietojen parametrisoinnista Jenkinsin pipeline-skriptissä.

Kuvassa 18 määritetään Jenkins-ajolle esimerkiksi versionhallinnasta haettava oikea haara.

Lopuksi asetettiin Jenkins lähettämään testien kehittäjän sähköpostiin tiedot ajosta, jos jokin epäonnistuu, jolloin kehittäjä tietää nopeasti jonkin menneen pieleen.

6.4 Tulosten tulkinta

Jokaisen Jenkins-ajon päätyttyä tuli Jenkinsiin näkyviin värikoodattu tieto siitä, menikö edellinen testi läpi kokonaan vai epäonnistuiko siinä jokin. Myös konsolisyöte oli saatavilla, josta nähtiin, mikä testi oli milloinkin ajon alla ja missä mahdolliset virheet tapahtuivat, mikä teki ongelmien selvittämisestä helpompaa. Testien trendikuvaajasta oli helppo nähdä nopeasti edellisten testiajojen lopputulokset. Tämä kertoi paljon siitä, olivatko testit vakaita vai oliko niissä paljon heittelyä lopputulosten osalta. Epävakaasti toimiva testi saattoi kertoa ympäristön toiminnallisuuden ongelmasta tai heikosti toteutetusta testistä.



Kuva 19. Testien trendikuvaaja, josta nähdään, että testit ovat olleet suhteellisen vakaita. Keskellä olevat epäonnistuneet ajot johtuvat mitä todennäköisimmin siitä, että ympäristö tai yhteydet ovat olleet alhaalla.

Asiakkaalle helpompitulkintaisia tuloksia kuin konsolisyötteet olivat Robot Frameworkin generoimat lokitiedostot sekä raportit. Jälkimmäisenä mainutusta on helppo osoittaa nopeasti ongelmakohdat. Jos jokin testi jatkuvasti epäonnistui, niin osattiin kertoa, mistä sovelluksen osasta oli kyse ja mitä testattiin.

Status:	2 critical tests failed
Start Time:	20180731 05:30:22.712
End Time:	20180731 05:37:42.981
Elapsed Time:	00:07:20.269
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	25	23	2	00:06:42	
All Tests	25	23	2	00:06:42	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Tests	25	23	2	00:07:20	
Tests . AUTO 01	5	5	0	00:01:42	
Tests . AUTO 02	6	5	1	00:01:38	
Tests . AUTO 03	2	2	0	00:00:45	
Tests . AUTO 04	1	0	1	00:00:33	
Tests . AUTO 05	1	1	0	00:00:25	
Tests . AUTO 06	1	1	0	00:00:17	
Tests . AUTO 07	1	1	0	00:00:21	
Tests . AUTO 08	3	3	0	00:00:46	
Tests . AUTO 09	1	1	0	00:00:17	
Tests . User and access	4	4	0	00:00:35	

Kuva 20. Esimerkki Robot Frameworkin generoimasta raportista.

Edellisestä kuvasta nähdään, että kaksi testiä 25:sta on epäonnistunut, ja ongelmakohdat voidaan nopeasti nähdä olevan kahdessa testisarjassa. Klikkaamalla

haluttua testisarjaa nähdään lisätietoa yksittäisten testien suorituksista. Lisätiedoissa myös näkee syyn sille, mikä on aiheuttanut testin epäonnistumisen.

Tests . AUTO 02 Verify valid search	User logs into as a user with rights '. Does a search and checks screen elements from . Checks that the search in works as intended when is checked.	30033	yes	PASS	
Tests . AUTO 02 Verify valid search for	User logs into as a user with rights ' Does a social security number search and checks screen elements from Checks that the search in works as intended when is checked.	30034	yes	FAIL	Keyword 'Element Should Be Visible' failed after retrying for 10 seconds. The last error was: Element '//td[@class='top-left']/./a[contains(text(),' ')]' should be visible, but it is not.

Kuva 21. Ylempänä läpi mennyt testi ja tämän alla epäonnistunut testi.

Kuvassa 10 testin epäonnistumiselle syynä on ollut elementti, jonka on odotettu olevan näkyvissä, mutta ei ole ollut.

Vielä tarkempaa tietoa yksittäisistä testeistä saa Robot Frameworkin lokitiedostojen kautta. Siellä voi purkaa jokaisen testin yhden askeleen tarkkuudelle testien tasolla. Askeleesta, jossa testi on epäonnistunut, on myös kuvankaappaus ohjelmasta. Tämä on mielestäni erittäin hyvä ja tärkeä ominaisuus, sillä se vähentää ongelmien ratkomiseen kulunutta aikaa paljon, kun testit ovat kehityksen alla.

6.5 Dokumentaatio ja palaute

Kuten kaikessa ohjelmistokehityksessä, myös näistä testeistä tehtiin dokumentaatiot. Vanhoja dokumentaatioita ei voinut enää käyttää, sillä ne eivät olleet ajan tasalla Java-koodien kanssa, joten uudet oli kirjoitettava. Dokumentaatio toteutettiin SpiraTest-palveluun, joka on työkalu testien hallintaan, vaatimusten jäljitettävyyteen sekä bugien seurantaan [21]. Sen kautta voidaan myös ajaa testejä, jolloin se luo kuvaajia ja statistiikkaa testeistä.

Dokumentaatio toteutettiin samaan tapaan kuin vanhakin dokumentaatio oli toteutettu, eli testit kirjattiin ylös askel askeleelta ja alkuun kerrottiin, mitä testin on tarkoitus tehdä. Lisäksi kerrottiin, mitä oikeuksia eli käyttäjätunnusta tulee käyttää, jotta testi toimii. Tällöin testit ovat helposti toistettavissa esimerkiksi myös manuaalisessa testauksessa.

This test does a search and checks that correct data is shown to this user.
User:
Steps:
1. Log into
2. Search with
3. Go to '
4. Check page content:
- ' ' is visible
- ' ' is visible
- ' ' is visible
- ' ' is not visible
- ' ' is not visible

Kuva 22. Esimerkki dokumentoidusta testistä Spirassa

Kuvassa 22 on ote siitä, kuinka testit dokumentointiin Spiraan. Aluksi on lyhyt kuvaus testistä, minkä jälkeen käydään testin kulkua läpi vaihe kerrallaan. Tämä näkyy kuvassa 'Steps' alla lueteltuina numeroina. Kuvasta on jouduttu sensuroimaan pois osioita, kuten esimerkiksi se, mitä odotetaan sivulta löytyvän (4. Check page content... ” is visible.).

Palautteen osalta koostimme jokaisesta ohjelmiston kokonaisuudesta oman PowerPoint-esityksen, jotka esitimme asiakkaalle viimeisessä kokouksessa eli niin sanotussa loppudemossa. Dioissa oli tiivistettynä perustietoa testeistä, kuten missä ympäristössä niitä oli ajettu, missä testit tällä hetkellä sijaitsevat ja mitkä testit epäonnistuvat. Tämän lisäksi kerroimme, mitä oli muutettu edellisiin testeihin verrattuna ja annoimme jatkokehitysideoita, jos sellaisia oli. Jos jatkokehitysideoita tai parannusehdotuksia oli, keskusteltiin niistä asiakkaan kanssa loppudemossa tai sen jälkeen.

7 Yhteenveto ja pohdinta

Tämän työn aiheena oli regressiotestiuudistuksen tuottaminen, tutkia testaamista, automaatiotestaamista sekä Robot Frameworkia. Regressiotestiuudistus toteutettiin asiakastyönä web-pohjaiseen ohjelmaan, missä siirettiin vanhat Java-testit Robot Frameworkille ja uusittiin dokumentaatiota. Työssä edettiin ensin käsittelemällä ympäristöjen asentamista, jonka jälkeen siirryttiin käytännön testausprosessin

toteutukseen. Lopuksi tutkittiin, kuinka prosessi saatiin automoitua jokaöiseen suoritukseen sekä tarkasteltiin tuloksia.

Projektin edetessä huomasin automaation tuomia hyötyjä nopeasti myös itse. Testit ajettiin joka yö asiakkaalla, ja testien tulokset olivat helposti nähtävissä ja luettavissa. Virhetilanteiden sattuessa lähetti Jenkins automaattisesti sähköpostin minulle. Mielestäni testien automatisointi tämänkaltaisessa tilanteessa on erittäin hyödyllistä, sillä kyseisten testien manuaalisesti suorittaminen olisi hyvin aikaavievää.

Testaajana aloittaminen ja Sogetilla työskentely sekä testauksen oppiminen ovat olleet hyvin silmiä avaava kokemus. Minulla ei ollut ennen Sogetille tuloa kovinkaan suurta kokemusta testaamisesta, joten en osannut täysin odottaa, mitä on vastassa, kun aloitan työni. Ehkä jopa yllätyksekseni voin todeta, että odotukseni ovat ylittyneet testaamisen osalta. Se on hyvin mielenkiintoista ja siinä on vuosiksi opittavaa laajalta alueelta.

Lähteet

1 The Explosion of the Ariane 5. 23.7.2000. Verkkoaineisto. Douglas N. Arnold. <http://www-users.math.umn.edu/~arnold/disasters/ariane.html> Luettu 27.7.2018.

2 TMAP Suite Workbook. s. 139. <https://drive.google.com/open?id=0B1mncLJKDX4DNHJhRGM1NXd4X0E>. Luettu 27.7.2018.

3 Manual Testing Tutorial for Beginners. Verkkoaineisto. Guru99. <https://www.guru99.com/manual-testing.html>. Luettu 20.10.2018.

4 Pros and Cons of Automated and Manual Testing. Maryna Zavyiboroda / RubyGarage.<https://rubygarage.org/blog/automated-and-manual-testing>. Luettu 22.10.2018.

5 Truth About Test Automation. 2018. PowerPoint-diasarja, sisäinen dokumentti. Kaisu Sahla / Sogeti Finland. Luettu 7.8.2018.

6 Continuous Delivery Vs. Continuous Deployment: What's the Diff?. 30.4.2013. Verkkoaineisto. Carl Caum. <https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff > Luettu 3.8.2018.

7 Should You Aim For 100 Percent Test Coverage? 2017. Verkkoaineisto. Erik Dietrich. <https://dzone.com/articles/should-you-aim-for-100-percent-test-coverage> . Luetu 20.10.2018.

8 TMap Suite Test Engineer. 2017. PowerPoint-diasarja, sisäinen dokumentti. Juhana Britschgi / Sogeti Finland. Luettu 12.9.2018.

9 No struggle with test design. 2015. PowerPoint-diasarja. Rik Marselis. <https://www.slideshare.net/RikMarselis/no-strugglewithtestdesignrikmarselistestexpo>. Luettu 4.10.2018.

10 04_Asiantuntija-arviot 2016. PowerPoint-diasarja. Erja Nikunen / Metropolia. <https://oma.metropolia.fi/delegate/download_workspace_attachment/1024750/04_Asiantuntija-arviot.pdf>. Luettu 25.10.2018.

11 The Role of Unit Tests in Test Automation. 2017. Verkkoaineisto. Bobby Lalvani / DevOps Zone. <https://dzone.com/articles/the-role-of-unit-tests-in-test-automation>. Luettu 2.10.2018.

12 Robot Framework User Guide. 2018. Verkkoaineisto. Robot Frameworks Foundation.<http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#test-data-tables> Luettu 8.8.2018.

13 How the World Tests. 2016. Verkkoaineisto. Zephyr. <https://cdn2.hub-spot.net/hubfs/1646055/__ASSETS__/Whitepa-pers/pdf/How_The_World_Tests_2016.pdf>. Luettu 16.8.2018.

14 Selenium Webdriver. 2018. Verkkoaineisto. Selenium Project. <https://www.seleniumhq.org/docs/03_webdriver.jsp#how-does-webdriver-drive-the-browser-compared-to-selenium-rc>. Luettu 16.8.2018.

15 Ohjelmiston testaus ja laatu. Pdf-opetusmateriaali. Seppo Räsänen / Savonia-ammattikorkeakoulu. <http://webd.savonia.fi/home/ktrasse/muut/testaus_laatu/testaus_3.pdf>. Luettu 4.10.2018.

16 Ohjelmistoprojektin sudenkuopat ja miten ne vältetään, s27. 2018. Rami Juvonen. Luettu 21.10.2018.

17 What is BLACK Box Testing? Techniques, Example & Types. Verkkoaineisto. Guru99. <https://www.guru99.com/black-box-testing.html> . Luettu 27.8.2018.

18 Ask a Security Professional: Black Box vs. White Box Series – Part Two: White Box Testing. 2016. Verkkoaineisto. Lauren Papagalos. <https://www.sitelock.com/blog/2016/06/black-box-vs-white-box-part2-sast/> . Lu-ettu 20.10.2018.

19 White-Box Testing: Pros and Cons. 2017. Verkkoaineisto. James Lee / Segue Technologies. <https://www.seguetech.com/white-box-testing-pros-cons/>. Luettu 27.8.2018.

20 Gray Box Testing. Verkkoaineisto. Software Testing Fundamentals. <http://soft-waretestingfundamentals.com/gray-box-testing>. Luettu 28.8.2018.

21 SpiraTest. Verkkoaineisto. Inflectra Corporation. <https://www.inflectra.com/Spi-raTest/> Luettu 31.7.2018.

22 DEVOPS 2016: Robot Framework-työkalun pääkehittäjä Pekka Klärck nousee lavalle. 2016. Verkkoaineisto. Eficode. <https://www.eficode.com/blogi/blogi/devops-2016-robot-framework-tyokalun-paakehittaja-pekka-klarck-nousee-lavalle> Luettu 7.8.2018.

23 Robot Framework. 2018. Verkkoaineisto. <http://robotframework.org/#users>. Luettu 21.10.2018.