Juha-Matti Niiranen

TESTAUKSEN AUTOMATISOINTI ROBOT FRAMEWORKILLA

Case 2M-IT Oy

Opinnäytetyö
Tietojenkäsittely

2018

Kaakkois-Suomen
ammattikorkeakoulu

Kaakkois-Suomen ammattikorkeakoulu

Tekijä/Tekijät	Tutkinto	Aika
Juha-Matti Niiranen	Tradenomi (AMK)	Toukokuu 2018

Opinnäytetyön nimi

Testauksen automatisointi Robot Frameworkilla
Case 2M-IT Oy

52 sivua
1 liitesivu

Toimeksiantaja

2M-IT Oy

Ohjaaja

Janne Turunen

Tiivistelmä

Opinnäytetyössä toteutettiin toimeksiantajan 2M-IT Oy:n tuottaman sähköisen asiointipalvelu Hyviksen yksikkötestauksen automatisointi Robot Frameworkilla. Opinnäytetyössä käsiteltiin testauksen automatisoinnin toteuttaminen tarvittavista ohjelmistojen asennuksista aina kirjoitettujen testien testiajojen tulosten tarkasteluun asti.

Opinnäytetyön teoriaosuudessa selvitetään testauksen yleisiä piirteitä osana ohjelmistotuotantoa, testauksen tasoja, eri testausmenetelmiä sekä testauksen hyötyjä. Teoriaosuudessa käsitellään myös itse testauksen automatisointia sen hyötyjen ja ongelmien osalta sekä myös automatisoinnin eri työkaluja.

Opinnäytetyön Case-osuudessa pyrittiin käsittelemään testauksen automatisoinnin toteutuksen eri vaiheet niin selkeästi, että lukija pystyisi itsekin halutessaan automatisoimaan Robot Frameworkilla. Tämä osuus ajoittui noin neljän viikon pituiseen testauksen automatisoinnin pilotointivaiheeseen. Case-osuutta pohjustettiin kertomalla testauksen nykytilanteesta 2M-IT:llä sekä automatisoinnin alkuvalmisteluista. Itse automatisoinnin toteutustyö aloitettiin tarvittavilla ohjelmistoasennuksilla ja niiden käyttöönotolla. Tässä osuudessa käytiin lisäksi läpi muun muassa automaatioprojektin rakennetta, testien kirjoittamista ja ajamista Robot Frameworkilla sekä testitulosten tarkastelua. Tämän osuuden lopussa kerrottiin havaituista ongelmatilanteista sekä niiden mahdollisista ratkaisuista. Case päättyi testauksen automatisoinnin pilotointivaiheen onnistumisen arviointiin.

Opinnäytetyössä toteutettu testauksen automatisointi täytti toimeksiantajan asettamat pilotointivaiheen tavoitteet. Pilotointivaiheessa korostettiin testejä kirjoittaessa niiden uudelleenkäytettävyyttä sekä pyrittiin siihen, että sama testitapaus toimisi eri testiympäristöissä.

Asiasanat

ohjelmistotuotanto, laadunvalvonta, testaus, yksikkötestaus, automaatio, Robot Framework

South-Eastern Finland University of Applied Sciences

Author (authors)	Degree	Time
Juha-Matti Niiranen	Bachelor of Business Administration	May 2018

Thesis title

Test automation with Robot Framework
Case 2M-IT Oy

52 pages
1 page of appendices

Commissioned by

2M-IT Oy

Supervisor

Janne Turunen

Abstract

The objective of the thesis was to implement test automation of the unit testing of Hyvis with Robot Framework. Hyvis is an electronic customer service produced by 2M-IT that commissioned this thesis. This thesis described the implementation of the test automation from the installation of the necessary software to reviewing the results of the test runs of written tests.

The theoretical part of the thesis mainly focused on describing testing and test automation in general. This included testing as a part of software production, different levels of testing, different testing methods and the benefits of proper testing. Test automation was considered from the perspective of its benefits and problems. Lastly, the theoretical part of the thesis covered a few tools used for test automation.

The case part of the thesis focused on explaining the different steps of test automation so thoroughly that the reader could potentially automate testing with Robot Framework, too. The case part of the thesis covered a four-week long test automation pilot phase. This part began by explaining the current state of 2M-IT's software testing. The implementation of the test automation began by its initialization, by installing necessary software and the deployment of this software. This part of thesis also went through the structure of an automation project, writing and executing written test cases with Robot Framework, and viewing the test results. In addition, the case part of the thesis described some of the encountered problems and possible solutions to these problems. Lastly, the case part ended by evaluating the success of the test automation pilot phase.

The test automation implemented in the thesis met the objectives set by the commissioner for the pilot phase. Reusability of test cases was emphasized during the pilot phase. This ensured that the same test case would work in different test environments.

Keywords

software production, quality control, testing, unit testing, automation, Robot Framework

SISÄLLYS

1 JOHDANTO 5

2 TESTAUS YLEISESTI 6

2.1 Testaus osana ohjelmistotuotantoa 6

2.2 Testauksen tasot 10

2.3 Testausmenetelmät 12

2.3.1 Esitestaus 12

2.3.2 Kehitysvaiheen testausmenetelmät 14

2.3.3 Julkaisuvaiheen testausmenetelmät 16

2.4 Testauksen hyödyt 18

3 TESTAUKSEN AUTOMATISOINTI JA SEN TYÖKALUT 20

3.1 Automatisoinnin hyödyt 20

3.2 Automatisoinnin ongelmat 22

3.3 Automatisoinnin eri työkalut 24

4 CASE 2M-IT OY: TESTAUKSEN AUTOMATISOINNIN TOTEUTTAMINEN ROBOT FRAMEWORKILLA 26

4.1 Testauksen nykytilanne 2M-IT:llä 27

4.2 Automatisoinnin alkuvalmistelut 27

4.3 Asennukset ja käyttöönotto 29

4.4 Automaatioprojektin rakenne ja sen merkitys 33

4.5 Testien kirjoittaminen Robot Frameworkilla 35

4.6 Testien ajaminen ja testitulosten tarkastelu 41

4.7 Ongelmatilanteet 45

4.8 Testauksen automatisoinnin onnistumisen arvioiminen 47

5 PÄÄTÄNTÖ 48

LÄHTEET 50

LIITTEET

Liite 1. Asennuslinkit

1 JOHDANTO

Opinnäytetyöni aiheena on 2M-IT Oy:n tuottaman sähköisen asiointipalvelu Hyviksen yksikkötestauksen automatisointi Robot Frameworkia hyödyntäen. Opinnäytetyöni koostuu teoriapohjaisesta selvityksestä testauksesta ja sen automatisoinnista sekä myös itse testauksen automatisoinnin rakentamisesta 2M-IT:n tapauksessa.

Opinnäytetyötäni aloittaessa toimeksiantajani oli Medi-IT Oy, mutta yhtiö fuusioitui Medbit Oy:n kanssa 1.3.2018 alkaen muodostaen uuden yhtiön, 2M-IT Oy:n. Kyseessä on Suomen suurin sosiaali- ja terveydenhuollon tietoteknisiä palveluja tuottava julkisomisteinen yhtiö. Yhtiön toiminta kattaa 11 maakuntaa ja 12 sairaanhoitopiiriä. 2M-IT tarjoaa sosiaali- ja terveydenhuollon ICT-palveluja kolmella osa-alueella: tietotekniikkapalvelut, ratkaisupalvelut sekä sovelluspalvelut. (2M-IT Oy s.a.)

Opinnäytetyöni toisessa luvussa käsitellään testausta yleisesti. Luvussa keskitytään etenkin testaukseen osana ohjelmistotuotantoa sekä testauksen eri tasoihin ja menetelmiin. Lisäksi selvitän testauksesta saatavat mahdolliset hyödyt.

Kolmas luku keskittyy kokonaan testauksen automatisointiin. Luvussa käydään läpi muun muassa automatisoinnin hyötyjä ja ongelmia. Tämän lisäksi luvussa kerrotaan eri automatisoinnin työkaluista ja niiden toimintaperiaatteista.

Opinnäytetyöni neljännessä luvussa paneudutaan itse 2M-IT:n tapaukseen, eli testauksen automatisoinnin toteuttamiseen Robot Frameworkilla. Aluksi selvitetään testauksen nykytilannetta 2M-IT:llä. Tämän jälkeen selvitetään, mitä alkuvalmisteluja oli aiheellista tehdä testauksen automatisointia ajatellen. Itse automatisoitujen testien kirjoittamista varten oli tehtävä eri ohjelmistojen asennuksia sisältäen myös ohjelmistojen käyttöönoton. Suorat asennuslinkit sisällytettiin liitteeseen 1. Alkuvalmistelujen ja tarvittavien asennusten jälkeen luvussa käydään läpi Robot Frameworkin automaatioprojektin perusrakennetta, sekä testien kirjoittamista ja niiden ajamista. Luku pitää sisällään myös oman osionsa testiajon tulosten tarkastelusta. Luvun lopuksi keskitytään havaittuihin

ongelmatilanteisiin ja niiden ratkaisuihin. Edellä mainittujen kokonaisuuksien perusteella pystyttiin myös arvioimaan testauksen automatisoinnin onnistumista yleisesti.

2 TESTAUS YLEISESTI

Ohjelmistotestaus on työtä, jolla pyritään varmistamaan toteutettavan ohjelmistotuotteen olevan määrittelyn ja vaatimusten mukainen. Tämän lisäksi ohjelmistotestauksella varmistetaan, että kaikki valmiiksi saadut ominaisuudet toimivat niin kuin pitää, mielellään ilman virheitä. (ISTQB Exam Certification s.a.)

Jussi Pekka Kasurinen kirjassaan ’Ohjelmistotestauksen käsikirja’ (Kasurinen 2013, 10) tiivistää ohjelmistotestauksen hienosti yhteen lauseeseen: ”Varmistetaan, että tehdään oikeaa tuotetta ja että tuote on tehty oikein”.

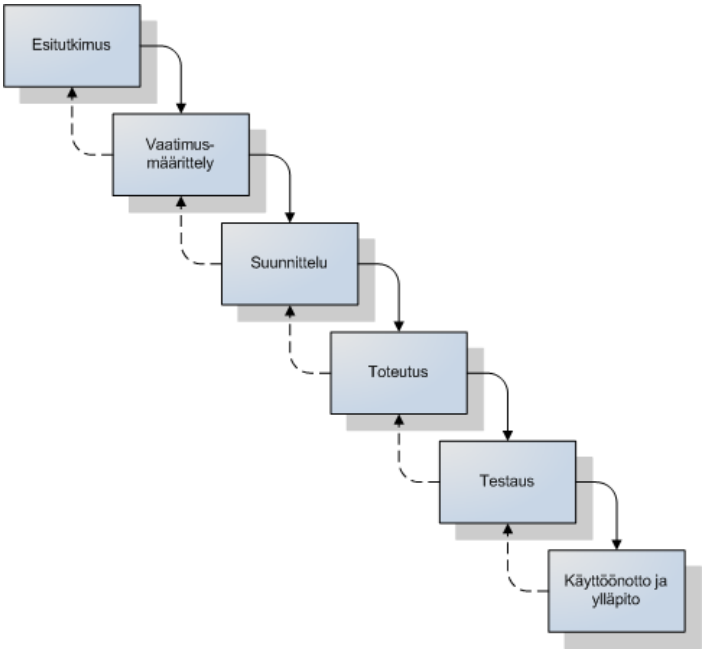
Ohjelmistotestaus on olennainen osa ohjelmistotuotannon kokonaisuutta. Ohjelmointityöhön tai tekniseen kirjoittamiseen verrattuna testaus on laajempi kokonaisuus, sillä testaaja voi työssään joutua muun muassa kirjoittamaan koodia, dokumentoimaan havaintojaan tai esimerkiksi haastattelemaan potentiaalisia ohjelmiston käyttäjiä. Testaus ei siis tarkoita pelkkää testitapausten ajamista, niin kuin monesti saatetaan ajatella. Testaajan eri työtehtävät riippuvat hyvin paljon tehtävistä testeistä. (Kasurinen 2013, 10.)

2.1 Testaus osana ohjelmistotuotantoa

Testaus on yksi merkittävimmistä tekijöistä ohjelmiston onnistumisen kannalta. Jos testaus on ollut puutteellista tai vajavaista ohjelmiston elinkaaren aikana, siitä johtuvat yritykselle koituvat tappiot voivat olla valtavia kuten voidaan havaita vuonna 2002 tehdyn tutkimuksen perusteella. Tässä tutkimuksessa (Tassey 2002) havainnollistetaan vajavaisen tai puutteellisen testauksen aiheuttaneen yhdysvaltalaisille ohjelmistotaloille 21,2 miljardin dollarin tappiot ja tuotannonmenetykset. Samaisen tutkimuksen (Tassey 2002) mukaan summa nousee 59,5 miljardiin dollariin ottaessa huomioon myös asiakkaille koituneet tappiot, tuotannonmenetykset ja vahingot. (Kasurinen 2013, 11.)

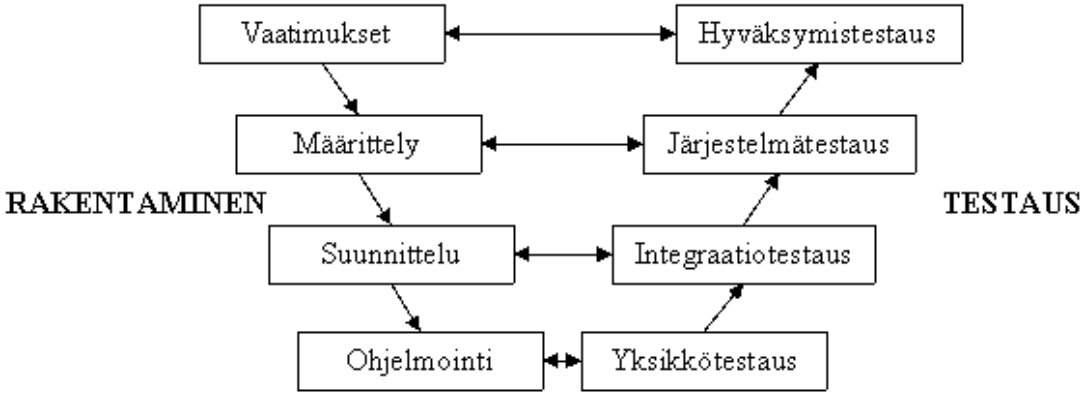
Ohjelmistoprojektien kokonaisbudjetista testaukselle varataan normaalisti noin 25–65 %. Suomessa tämä luku asettuu keskimäärin 27 % tietämille. Prosenttijakaumasta voidaan havaita, että testaus on yleensä projektin kallein työvaihe (Kasurinen 2013, 12). Mikäli projektissa on varattu testaukselle laaduntakaamisen näkökulmasta riittämätön määrä resursseja, ei kaikkia ohjelmiston vikoja tulla todennäköisesti löytämään ennen ohjelmiston viemistä tuotantoon. Esimerkiksi jos yrityksellä on varattuna testaukselle 75 % optimaalisesta resurssimäärästä, löydetään todennäköisesti neljä viidesosaa ohjelmiston vioista. (Kasurinen 2013, 12.)

Ohjelmistotuotantoa voidaan kuvata eri malleilla. Tällaisia malleja ovat muun muassa vesiputousmalli, V-malli, RUP sekä Scrum. Ohjelmiston testaus ilmenee näissä malleissa kussakin eri tavalla. Kaikkein yksinkertaisin ja helpoiten ymmärrettävä malli, vesiputousmalli, tarkoittaa ohjelmistotuotannossa käytännössä lineaarista kaaviota ohjelmistotuotannon eri vaiheista. Vesiputousmallin vaiheita on yleensä 5–7, riippuen mallin kuvauksen tekijästä. Esimerkiksi Kasurinen on kuvannut kirjassaan vesiputousmallin viisivaiheiseksi, kun taas kuvassa 1 se on kuvattu kuusivaiheiseksi. Vesiputousmallin vaiheita ovat esitutkimus ja vaatimusten määrittely, suunnittelu, toteutus, testaus, käyttöönotto ja ylläpito. Voidaan todeta, että testausta toteutetaan tässä mallissa vain yhdessä kohtaa, mikä ei ole kovin järkevää laajemmissa ohjelmistoprojekteissa. Testauksessa havaittuja puutteita on vaikeaa lähteä korjaamaan, jos virhe on sattunut jo esimerkiksi suunnittelussa. (ISTQB Exam Certification s.a.) Vesiputousmalli on esitetty hieman eroavalla tavalla Ilkka Haikalan ja Jukka Märijärven kirjassa Ohjelmistotuotanto (2004, 37), sillä heidän esittämässä vesiputousmallissa testausta ja muita tarkastustoimenpiteitä tehdään jokaisessa vesiputousmallin vaiheessa, eikä pelkästään testauksen omassa osuudessa.



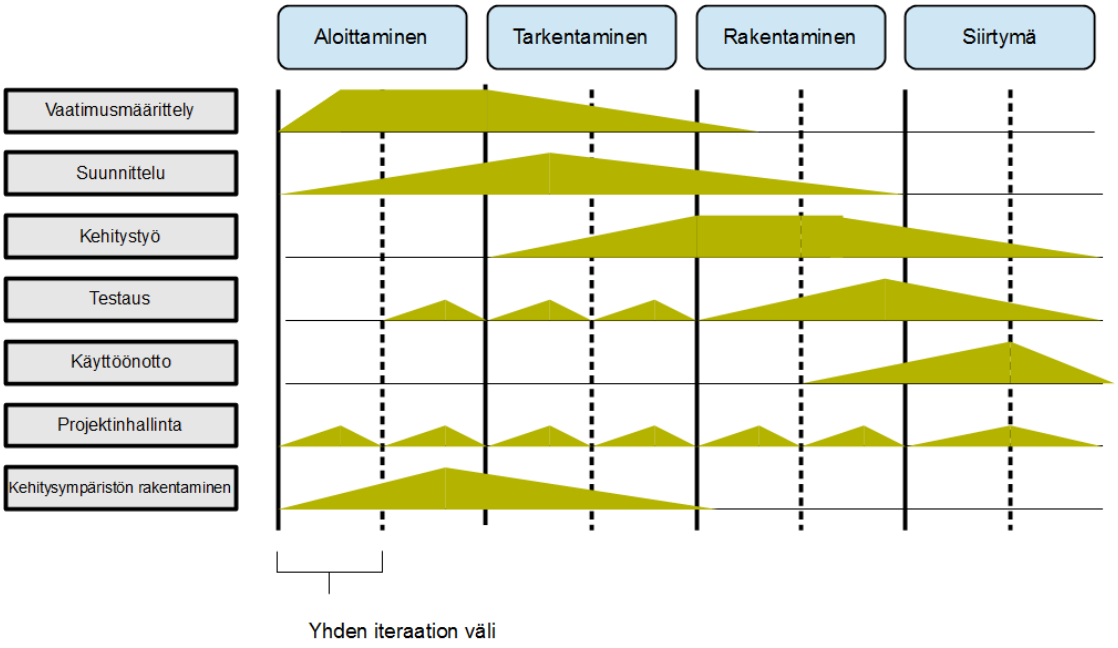
Kuva 1. Ohjelmistotuotannon vesiputousmalli (Ahonen 2010)

Toinen malli on niin sanottu V-malli. V-mallissa testataan jokainen ohjelmistotuotannon vaihe erikseen. V-mallin vaiheet ovat samat kuin vesiputousmallissakin. Toteutustyö tarkistetaan yksikkötestauksella, suunnittelu integraatiotestauksella, määrittelyn paikkansapitävyys järjestelmätestauksella sekä järjestelmän yleiset vaatimukset hyväksymistestauksella (kuva 2). Jos kaikista mainituista testauksen tasoista saavutetaan projektia ajatellen tarpeeksi tyydyttävä tulos, voidaan ohjelmisto viedä tuotantoon ja siirtyä sen ylläpitoon. Niin V-mallissa kuin vesiputousmallissakin ja muissa vastaavissa malleissa aloitetaan testaus liian myöhään. Tämä johtaa siihen, että mahdollisia löydettyjä vikoja on entistä vaikeampaa korjata jälkikäteen ilman, että sillä olisi merkittävää vaikutusta ohjelmistoprojektin onnistumiseen. (Haikala & Märijärvi 2004, 288-290.)



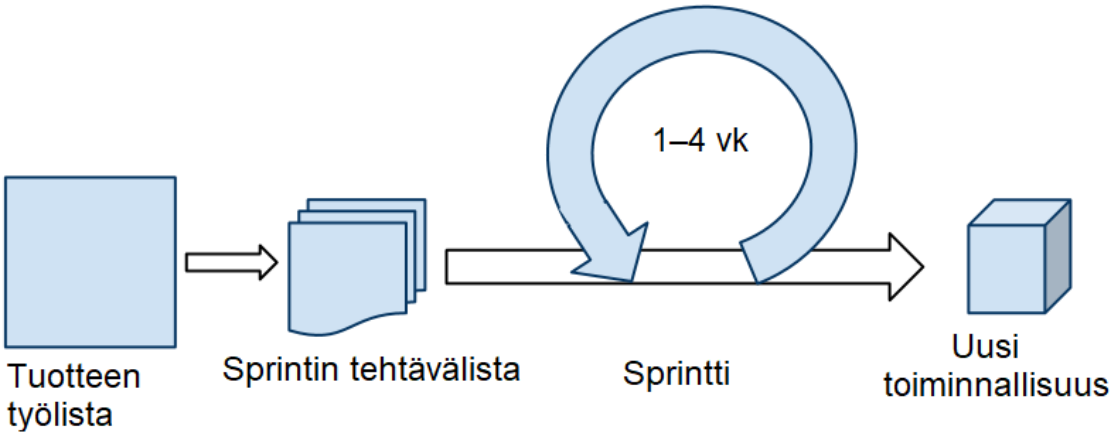
Kuva 2. Ohjelmistotuotannon V-malli (Soberit Aalto-yliopisto s.a)

Edellä mainituissa ohjelmistotuotannon malleissa ongelmaksi havaittu testauksen myöhäinen aloittaminen on otettu huomioon RUP (Rational Unified Process) -prosessimallissa. Siinä hyödynnetään muissakin malleissa nähtyä vaiheittaista kehitystapaa sekä varsinkin jatkuvaa laadunvalvontaa, eli testausta. RUP-mallissa testataan koko projektin ajan, minkä vuoksi osataan reagoida nopeasti mahdollisiin löydettyihin virheisiin ja vikoihin projektin eri vaiheissa. RUP-mallille tyypillistä on projektin jakaminen neljään päätyövaiheeseen, joissa kussakin ohjelmaa kehitetään pienemmissä vaiheissa (kuva 3). Testaus sisältyy jokaiseen neljään päätyövaiheeseen. (Kasurinen 2013, 15-16.)



Kuva 3. RUP-malli (Kasurinen 2015)

Neljäs, sekä myös ehkä suosituin malli on Scrum. Siinä ohjelmistotuotannon projekti etenee säännöllisissä sykleissä eli sprinteissä (kuva 4). Sprintti on kehitysjakso, jonka aikana tuotetaan ohjelmiston tietty osa, tai pienemmissä projekteissa koko ohjelmisto (Sininen Meteoriitti 2013).



Kuva 4. Scrum-malli (Purojärvi 2010)

Tällaisessa ohjelmistoprojektin jaottelussa sprinteiksi hyvää on se, että voidaan keskittyä yhteen kehitettävään ohjelmiston osaan kerrallaan. Tällöin myös testauksessa voidaan keskittyä pelkästään sen hetkisen sprintin aikana kehitettävään osaan, jolloin voidaan varmistua osan virheettömästä toimivuudesta osana ohjelmistoa.

2.2 Testauksen tasot

Testauksen eri tasot ovat havaittavissa aikaisemmin mainitsemassani V-mallissa. Tasoihin luetaan yksikkötestaus, integrointitestaus, järjestelmätestaus ja hyväksymistestaus. Kukin testauksen taso käsittelee tiettyä kokonaisuutta ohjelmistosta.

Yksikkötestaus eli moduulitestaus tarkoittaa ohjelmiston yksittäisen komponentin eli moduulin testaamista. Yksikkötestauksesta vastaa yleensä kyseisen komponentin kehittäjä. Yksikkötestaamista varten saatetaan joutua luomaan niin sanottuja testipetejä, jotta voidaan simuloida oikeaa ohjelmiston ympäristöä, jossa kyseinen komponentti tulee toimimaan. Testipeti koostuu ajuri- ja tynkämoduuleista (engl. driver and stubs) (Pohjolainen 2003, 15). Yksikkötestauksessa havaitusta virheestä voidaan suoraan päätellä, että jotain on pielessä moduulissa. Tällöin viallinen moduuli ei päädy osaksi ohjelmistoa, jolloin sen havaitseminen ja korjaaminen olisi vaikeampaa ja kalliimpaa. (Kasurinen 2013, 51-52.)

Integrointitestaus tarkoittaa ohjelman eri komponenttien eli moduulien, keskenään toimivuuden testaamista. Tässä vaiheessa saatetaan vielä käyttää ajuri-

ja tynkämoduuleja simuloimaan jotakin toista moduulia, jonka kanssa haluttua moduulia testataan. Integrointitestauksessa ei kuitenkaan ole tarkoitus testata vielä koko ohjelmiston toimivuutta. (Pohjolainen 2003, 16)

Integraatio voi edetä eri tavoin. Alhaalta ylöspäin (bottom up) menetelmässä aletaan yhdistelemään matalimman tason moduuleja ja testataan niiden toimivuus yhdessä, ennen kuin lisätään taas uusi yksikkötestattu moduuli kokonaisuuteen. Tällä menetelmällä havaitaan helposti matalantason virheitä, jotka ovat tällöin myös helppoja korjata. (Haikala & Märijärvi 2004, 290.)

Ylhäältä alaspäin (top down) menetelmässä ohjelmiston kokoaminen ja testaus aloitetaan järjestelmähierarkian huipulta ja edetään kohti rautatason toimintoja. Tämän menetelmän avulla saadaan aikaisemmin parempi yleiskuva ohjelmiston toimivuudesta sekä myös tietoa mahdollisesti puuttuvista ohjelmiston toiminnoista. (Kasurinen 2013, 55.)

Näiden kahden menetelmän yhdistelmä on niin sanottu voileipätestaus (sandwich testing), eli integraatio tapahtuu molemmista päistä, järjestelmähierarkiasta ja matalimmista rautatason toiminnoista lähtien. Hyötynä kyseisessä menetelmässä on ajuri- ja tynkämoduulien tarpeen väheneminen muihin menetelmiin verrattuna. Haittapuolena tosin saattaa olla edessä olevat vaikeudet sovittaa kaikki ohjelmiston moduulit yhteensopiviksi toistensa kanssa. (ProfessionalQA 2018.)

Neljäs sekä ehkä hieman harvemmin käytetty menetelmä on niin sanottu kertarysäys (big bang). Kertarysäyksessä ohjelmiston moduulit yhdistetään toisiinsa kirjaimellisesti kaikki kerralla. Tällöin myös testattavaksi tulee koko ohjelmisto kaikkine moduuleineen. Integrointi on helppo toteuttaa kertarysäyksenä, mutta virheiden havaitseminen voi olla vaikeaa. Kertarysäys voidaan tehdä vain siinä tapauksessa, kun kaikki ohjelmiston moduulit ovat yksikkötestattuja, eli tällöin ohjelmiston testaaminen kokonaisuutena viivästyy huomattavasti. (Pohjolainen 2003, 20.)

Järjestelmätestauksella viitataan aikaisemmin mainitsemani V-mallin kolmanteen tasoon. Järjestelmätestaukseen voidaan siirtyä, kun ohjelmiston moduulit

ovat yksikkötestattuja ja varmistettu myös toimivaksi toistensa kanssa integraatiotestauksella. Tästä voidaan todeta, että järjestelmätestaus tarkoittaa kokonaisen ohjelmiston testaamista. Yksikkö- ja integraatiotestauksessa tarvitut ajuri- ja tynkämoduulit karsitaan tässä kohtaa pois, eli testataan ohjelmiston oikeilla moduuleilla kokonaisuutena. Järjestelmätestauksen tärkein tavoite on varmistaa, että ohjelmisto toimii kokonaisuutena sekä täyttää sille asetetut tavoitteet (Haikala & Märijärvi 2004, 290). Toiminnallisten osuuksien testaamisen lisäksi voidaan myös tehdä kuormitustestausta, luotettavuustestausta, asennustestausta sekä käytettävyystestausta. Näistä varsinkin kuormitustestaukseen olisi syytä panostaa, jotta voidaan varmistua ohjelmiston toimintakyvystä eri vallitsevissa olosuhteissa ja kuormassa. (Pohjolainen 2003, 16.)

Hyväksymistestaus on V-mallin viimeinen eli neljäs vaihe. Hyväksymistestaukselle on tyypillistä, että se suoritetaan kohdeympäristössä, kun taas alemman tason testaukset (järjestelmätestaus jne.) suoritetaan yleensä testi- tai kehitysympäristössä. Sen tavoitteena on osoittaa, että ohjelmisto on riittävän korkealaatuinen ja että se täyttää sille vaatimusmäärittelyssä asetetut vaatimukset ja tavoitteet. Hyväksymistestaus on viimeinen tarkastus ennen kuin valmis ohjelmisto luovutetaan asiakkaalle ja siirretään käyttöönottoon. Asiakkaan hyväksyttyä tämän viimeisen testauksen tulokset, valmis ohjelmisto siirtyy lakisääteisesti asiakkaan omistukseen. (Kasurinen 2013, 57.)

2.3 Testausmenetelmät

Ohjelmistoa testataan projektin eri vaiheissa eri menetelmien mukaan. Menetelmät vaihtelevat sen perusteella, ollaanko projektissa esituotannossa, kehitysvaiheessa vai valmiin ohjelmiston julkistuksen alla. Testausmenetelmät eri projektin vaiheissa voidaan esittää myös aikaisemmin mainitsemani V-mallin avulla. (Kasurinen 2013, 61.)

2.3.1 Esitestaus

Esitestauksella viitataan ohjelmistoprojektin esituotannossa tapahtuvaan testaukseen. Esitestaus ei varsinaisesti tarkoita ohjelmiston testaamista testitapauksia hyödyntäen, vaan lähinnä testauksen suunnittelua projektin eri vaiheita ajatellen. Esituotannon aikana voidaan myös työstää valmista ohjelmistoa kuvaava prototyyppi, jota sitten testataan. Tällöin voidaan havaita ajoissa

jo perustoiminnallisuuksiin liittyvät viat, joiden korjaaminen käy edullisesti. Perusperiaate on se, että mitä myöhemmin ohjelmiston kehitystä vika löydetään, niin sitä kalliimpaa sen korjaaminen on.

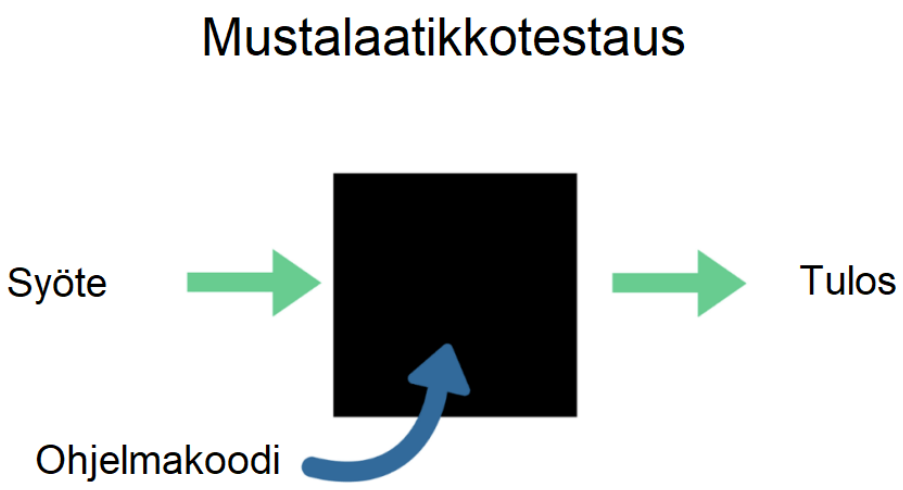
Eräs suosituimmista prototyypeistä, mitä työstetään esituotannon aikana, on käyttöliittymä. Käyttöliittymäsuunnittelulla pyritään tekemään ohjelmiston näkyvästä osasta eli käyttöliittymästä, mahdollisimman yksinkertainen ja helppokäyttöinen itse käyttäjälle. Kun testauksella otetaan kantaa jo näin varhain itse ohjelmiston käytettävyyteen, niin voidaan vielä tehdä käyttöä helpottavia muutoksia ohjelmistoon ilman suurempia taloudellisia kustannuksia. Kasurinen (2013, 63) kirjoittaa kirjassaan, että prototyypit auttavat myös tunnistettaessa ohjelmiston käyttötapauksia, joiden testaamiseen olisi syytä panostaa jatkossa. Täten prototyypit ovat oiva apu testauksen suunnittelussa.

Testauksen suunnittelu on toinen esitestauksen työmuoto. Yksi testauksen suunnittelun vaiheista on ensimmäisten testitapausten laadinta, jonka apuna käytetään ohjelmistoprojektin vaatimusmäärittelyä. Vaatimusmäärittelyssä on kuvattu ohjelmiston vaadittavat toiminnallisuudet ja rajoitteet, joiden perusteella alustavien testitapausten laatiminen on mahdollista (Kasurinen 2013, 63). Esimerkiksi viestinvälitysohjelmistolle on oleellista se, että viestien välittäminen on mahdollista, joten viestin lähettämisestä ja vastaanottamisesta voidaan laatia testitapaukset. Toinen testauksen suunnittelun kohde on mahdollisen testiympäristön suunnittelu ja rakentaminen sekä työkalujen valinta. Testiympäristön tulisi mallintaa valmiin ohjelmiston normaalia käyttöympäristöä. Tällöin testiympäristössä havaitut viat voidaan todeta myös normaalista käyttöympäristöstä. Lisäksi testiympäristöä hyödynnetään uusien toiminnallisuuksien testaamiseen ennen viemistä tuotantoon normaalissa käyttöympäristössä. Testauksen työkalujen valinnassa tulee kiinnittää huomiota niiden käytettävyyteen testaajien parissa. Pelkästään tiettyjen työkalujen käyttöönotto voi olla oma prosessinsa, joten harkintaa valinnan suhteen tulisi käyttää. Lopuksi suunnittelussa tulisi myös harkita itse testaajien valitsemista (Kasurinen 2013, 64). Ketkä ovat oikeita työhön? Kuka testaa mitäkin osaa ohjelmistosta? Kenelle raportoidaan havainnoista? Nämä ovat kysymyksiä, joita tulisi miettiä suunnitelmassa. Esitestauksen työvaiheisiin kuuluu olennaisesti testaajien valitseminen sekä mahdollinen kouluttaminen järjestelmän ympäristöön, testauksen työkaluihin ja muihin testauksen käytäntöihin.

2.3.2 Kehitysvaiheen testausmenetelmät

Kehitysvaiheen testausmenetelmiä ovat muun muassa mustalaatikkotestaus (black box testing), lasilaatikkotestaus (white box testing), harmaalaatikkotestaus (grey box testing) sekä regressiotestaus.

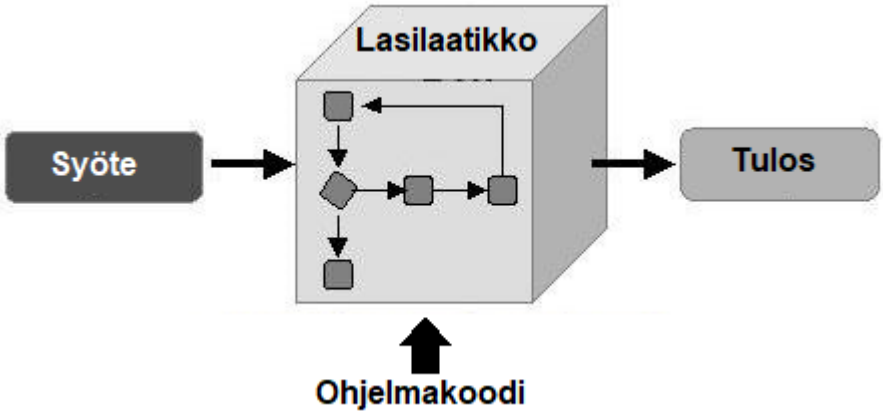
Mustalaatikkotestaus on se testauksen menetelmä, joka useimmille tulee mieleen testauksesta. Siinä ohjelmistoa tai sen osaa testataan antamalla syötteitä ja tarkastamalla niiden tuottaman lopputuloksen (kuva 5). Mustalaatikkotestauksessa ei siis kiinnitetä huomiota, miten itse ohjelmisto tai sen osa on toteutettu, vaan tarkastetaan, että eri syötteet toimivat vaatimusten ja rajoitteiden puitteissa sekä että lopputulos on oikein (Haikala & Märijärvi 2004, 291). Tämä menetelmä on siitä hyvä, että sitä voidaan hyödyntää missä tahansa testauksen vaiheessa, kunhan vain on olemassa jokin toiminnallisuus mitä testata (Kasurinen 2013, 65). Pentti Pohjolainen (2003, 18) toteaa pro gradu -tutkielmassaan: ”Mitä ylemmäs V-mallin oikeassa haarassa siirrytään, sitä enemmän testaus muuttuu mustalaatikkotestaukseksi.”



Kuva 5. Mustalaatikkotestaus (Testlio 2017)

Lasilaatikkotestauksessa tarkastellaan syötteiden ja lopputuloksen lisäksi myös itse ohjelmiston sisäistä toimintaa (kuva 6). Menetelmässä tarkastellaan, miten ohjelmisto reagoi annettuihin syötteisiin ja muodostaa niistä jonkin lopputuloksen. Esimerkiksi ohjelmiston muodostaessa virheellisen lopputuloksen, voidaan tarkastella ohjelmiston itse toteutusta etsiessä vikaa. Lasilaatikkotes-

tauksessa kuitenkin vaaditaan testaajalta perinpohjaista tietämystä järjestelmästä ja osaamista ohjelmointityöstä tutkiessa lähdekoodia. Ilman tietämystä ja osaamista, ei testaaja voi olla täysin varma siitä, että toimiiko ohjelma oikein vai ei. (Kasurinen 2013, 67.)



Kuva 6. Lasilaatikkotestaus (Kipp 2016)

Harmaalaatikkotestauksessa yhdistetään molempien, mustalaatikkotestauksen ja lasilaatikkotestauksen, hyvät puolet. Menetelmällä tarkastetaan, että ohjelmistoprojektin vaatimusmäärittelyssä asetetut vaatimukset ja rajoitteet toteutuvat ohjelmistossa. Lisäksi tarkastetaan, että itse lähdekoodi on kunnossa. Harmaalaatikkotestaus on kyseessä, kun esimerkiksi testataan verkkokaupan toimivuutta; sen sisältöä voidaan tarkastella lähdekoodin tasolla, mutta itse maksujärjestelmä on hankittu muualta, joten sen toimivuutta voidaan testata vain syötteiden ja lopputuloksen puolesta. (Kasurinen 2013, 68.)

Neljäs ohjelmiston kehityksen aikana käytetty testausmenetelmä on regressiotestaus. Regressiotestaus tarkoittaa käytännössä jonkin sellaisen, ohjelmiston tai sen osan, uudelleentestaamista, joka on jo aiemmin todettu toimivaksi (Kasurinen 2013, 68-69). Aina, kun jotain muutoksia tehdään jo toimivaan järjestelmän osaan, tulee se testata uudestaan regression varalta. Pentti Pohjolainen (2003, 17) toteaa pro gradu -tutkielmassaan, että regressiotestauksen rooli on tärkeä, sillä muutokset ja korjaukset ovat virhealttiimpia kuin alkuperäinen kehitystyö. Tämän lisäksi hän toteaa menetelmän sopivan hyvin laadunvarmistukseen sekä käytettäväksi etenkin ylläpidon aikana. Suuri osa testauksesta on nimenomaan regressiotestausta (Pohjolainen 2003, 17). Sekä Pohjolainen (2003, 17) että Kasurinen (2013, 70) molemmat toteavat, että reg-

ressiotestaus on oiva kandidaatti automatisoinnille, sillä regression varalta testattavien järjestelmän osien toiminnot ovat otollisia testitapauksille. Näiden testitapausten ajaminen käsin kerta toisensa jälkeen on työlästä, joten ne on järkevää automatisoida.

2.3.3 Julkaisuvaiheen testausmenetelmät

Julkaisuvaiheen testausmenetelmiä käytetään siinä projektin vaiheessa, kun ohjelmisto on jo olemassa ja toimintakykyinen, mutta ei vielä julkaisukelpoinen tai asiakkaalle luovutuskelpoinen. Tällaisia menetelmiä ovat muun muassa käytettävyystestaus, kuormitustestaus, suorituskykytestaus, savutestaus, alfa- ja beetatestaus, tutkiva testaaminen, ad hoc -testaus sekä testausautomaatio. Kyseisiä testausmenetelmiä voidaan pääsääntöisesti hyödyntää silloin, kun ohjelmisto on lähes valmis. (Kasurinen 2013, 70-76.)

Käytettävyystestauksella tarkoitetaan testaustyötä, jolla varmistetaan ohjelmiston käyttöliittymän toimivuus. Käytettävyystestausta voidaan julkaisuvaiheen lisäksi tehdä myös jo esituotannon aikana prototyyppien avulla. (Haikala & Märijärvi 2004, 291.) Mitä aikaisemmin käytettävyyttä päästään testaamaan prototyyppejä hyödyntäen, sitä enemmän siitä on hyötyä tuotekehitykselle (Avania 2009). Käytettävyystestausta voidaan toteuttaa myös oikeita potentiaalisia ohjelmiston käyttäjiä hyödyntäen. Esimerkiksi käyttäjä saatetaan laittaa täyttämään jokin lomake, ja samalla hänen silmänliikkeitään seurataan erilaisin katseenseurantamonitorein. Tällöin löydetään mahdolliset ongelmakohdat esimerkissä mainitussa lomakkeen täyttöprosessissa. Toinen hyvä esimerkki käytettävyystestauksesta on peliteollisuudesta tutut ennen itse julkaisua järjestettävät beetatestaukset. Näin pelaajat pääsevät tutustumaan peliin jo ennen varsinaista julkaisua, ja voivat samalla kokemustensa pohjalta antaa palautetta pelintekijöille.

Toinen ennen julkaisua tapahtuva testausmenetelmä on kuormitustestaus (load testing, stress testing). Edellä mainitussa peliteollisuuden esimerkissä peliä voidaan myös kuormitustestata. Tällöin testataan peliympäristön toimintakyky tietyllä kapasiteetilla. Voidaan asettaa maksimimäärä pelaajia, jotka voivat olla samaa aikaa pelissä, jolloin toivottavasti tuo maksimimäärä saavu-

tetaan ja voidaan havainnollistaa haluttua kuormaa peliympäristössä. Kasurisen (2013, 71) mukaan kuormitustestaus keskittyy kolmeen eri pääkohtaan: löytämään ohjelmistosta pullonkauloja, selvittämään ohjelmiston samanaikaisten käyttäjien maksimimäärän, sekä sen toimivuuden normaalioloissa. Mikäli ohjelmisto ei onnistu täyttämään tiettyjä kuormitukseen liittyviä vaatimuksia, on löydetty virhe, joka olisi syytä korjata ennen julkaisua. Kuormitustestaus voidaan viedä myös astetta pidemmälle, rasitustestaukseen. Rasitustestauksessa ohjelmistoa laitetaan käyttämään suurempi määrä käyttäjiä, kuin mitä suunniteltu normaali käyttäjämäärä on. Sen tavoitteena on löytää ehdoton maksimi ohjelmiston normaalille toiminnalle (Kasurinen 2013, 71). Suorituskykytestaus voidaan laskea myös kuuluvaksi kuormitustestaukseen, sillä siinä testataan ohjelmiston toimivuus vaatimusmäärittelyssä määritellyn käyttäjäkapasiteetin mukaisesti (Kasurinen 2013, 72).

Savutestaus (smoke test) on menetelmä, jolla varmistetaan testattavan ohjelmiston perustoiminteiden toimivuus. Esimerkiksi ajanvarausjärjestelmässä voidaan savutestata muun muassa rekisteröityminen, sisäänkirjautuminen sekä itse ajanvaraaminen. Ne ovat osa kyseisen järjestelmän perustoiminteista, joiden on oltava toiminnassa koko järjestelmän toimivuuden kannalta. Savutestauksella testataan perusasiat, joiden on toimittava. Jos perusasiat eivät toimi, on turhaa haaskata aikaa kattavampaankaan testaukseen. Termin ’*savutestaus’* sanotaan olevan peräisin laitteiston vastaavanlaisesta testaamisesta, jolla varmistettiin, ettei laitteisto syttynyt tuleen tai savuttanut ensimmäisellä käynnistyskerralla. (Software Testing Fundamentals s.a.)

Alfa- ja beetatestausta toteutetaan varsinkin erilaisille ohjelmistotuotteille (Software Testing Class 2015). Alfatestauksella (alpha testing) varmistetaan ohjelmiston toimivuus kokonaisuutena. Sen suorittavat yleensä kehittäjistä koostuva testaustiimi (Haikala & Märijärvi 2004, 291). Alfatestauksen tulosten perusteella voidaan täten tehdä vielä muutoksia kehittäjien toimesta, ennen kuin se viedään eteenpäin asiakkaalle tai käyttäjälle. Beetatestaus (beta testing) keskittyy oikeaan potentiaaliseen käyttäjäkuntaan. Ohjelmiston toimivuus normaalioloissa testataan sekä myös kerätään palautetta beetatestaajien kokemusten pohjalta. Beetatestauksen tulosten pohjalta voidaan puuttua muun muassa mahdollisiin toimivuusongelmiin sekä myös muuttaa tiettyjä toiminteita, jos ne ovat havaittu toimimattomiksi nykymuodossaan.

Ad hoc -testauksella tarkoitetaan täysin suunnittelematonta ja dokumentoimatonta testausta. Ad hoc -testauksessa kirjaimellisesti lähdetään satunnaisesti testaamaan eri ohjelmiston toiminteita ja katsotaan, että ohjelmisto toimii. Kasurinen (2013, 74-75) viittaa kirjassaan useaan otteeseen SWEBOK-testauskirjaan, jonka mukaan kyseinen ad hoc -menetelmä on maailman yleisin testausmenetelmä. Sitä voi tehdä käytännössä kuka tahansa ohjelmiston käyttäjä. Ainoana testausmenetelmänä käytettynä ad hoc -testaus ei ole kovinkaan suotavaa.

Tutkiva testaaminen muistuttaa paljon ad hoc -testausta, mutta siitä tehdään dokumentaatiota ja sen tavoitteena on täydentää testaussuunnitelmaa (Kasurinen 2013, 74). Tutkivassa testauksessa pyritään samanaikaisesti tutustumaan ohjelmistoon, suunnittelemaan sitä varten testit, suorittamaan suunnitellut testit sekä raportoimaan mahdollisista virheistä (Pohjolainen 2003, 22). Tutkivasta testaamisesta puhutaan myös ”testausmaailman ketteränä menetelmänä” (Kasurinen 2013, 74).

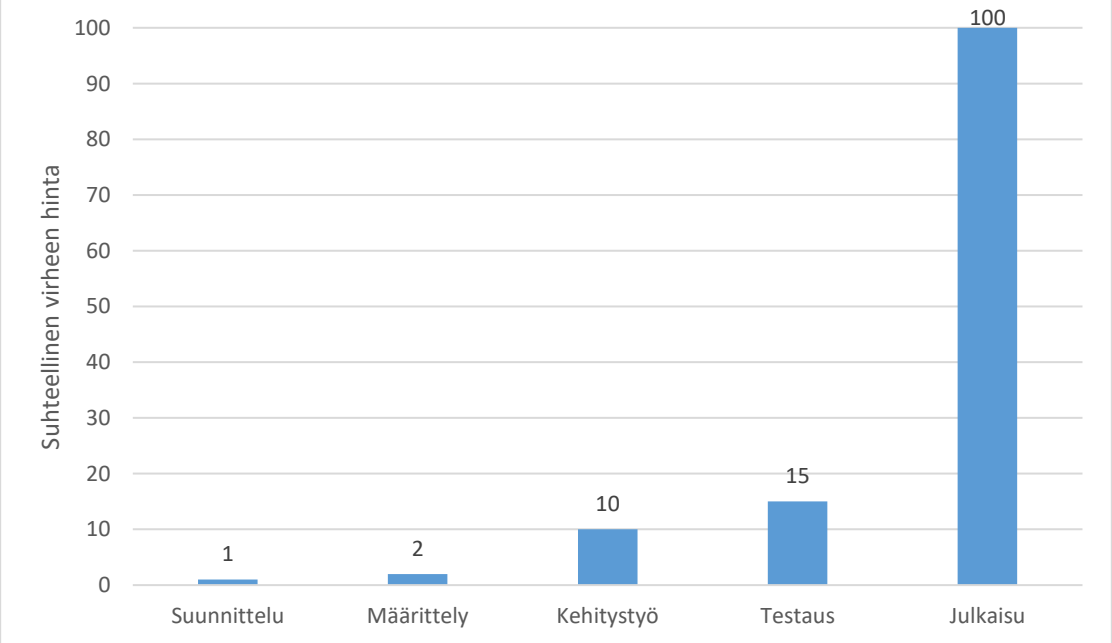
Testausautomaatio on automaattityövälineitä hyödyntävä testausmenetelmä. Sillä pyritään automatisoimaan sellaisia testitapauksia, joita joudutaan toistamaan useasti. Näin testaajat voidaan vapauttaa tekemään muita työtehtäviä. Testausautomaatiosta lisää seuraavassa luvussa.

2.4 Testauksen hyödyt

Kattavan testauksen suurimmat hyödyt keskittyvät virheiden löytämiseen mahdollisimman aikaisessa vaiheessa ohjelmistoprojektia sekä niistä aiheutuneiden kustannuksien minimointiin (Haikala & Märijärvi 2004, 40). Mitä aikaisemmassa vaiheessa ohjelmistoprojektia löydetään virhe testauksen eri menetelmin, sitä vähemmän kustannuksia siitä aiheutuu. Kasurisen (2013, 16-18) mukaan kuitenkin harmillisesti tilanne on joskus se, että projektissa leikataan kustannuksia nimenomaan testauksen osalta. Tällöin virheitä saattaa jäädä jopa valmiiseen tuotokseen, jolloin niiden korjaaminen tulee kalliiksi. Kasurinen (2013, 17) antaa esimerkin kirjassaan, että jo ohjelmistoprojektin suunnittelu-

vaiheessa tai ensimmäisessä prototyypissä havaitun vian korjaus maksaa sadasosan siihen verrattuna, että kyseinen vika havaittaisiin vasta jo asiakkaalle myydyssä tuotteessa.

Projektin eri vaiheissa toteutetun testauksen löytämien virheiden korjauskustannukset voidaan havainnoida kustannuskäyrillä (kuva 7). Jo julkaistussa tuotteessa havaittu virhe on kaikkein kalleinta (100 %) korjata. Ennen julkaisua tehtävässä järjestelmä- ja hyväksymistestauksessa havaitun vian korjaus (15 %) maksaa noin kuudesosan siitä, mitä se maksaisi jo julkaistussa tuotteessa. Kehitystyön aikana havaitun vian korjaus (10 %) on kymmenesosa verrattuna jo julkaistusta tuotteesta löydetyn vian korjaukseen. Jo ohjelmistoprojektin määrittelyvaiheessa havaitun vian korjaus (2 %) maksaa taas viidesosan siitä, mitä sen korjaus maksaa kehitysvaiheessa. Suunnitteluvaiheessa havaitun vian korjaus on ohjelmistoprojektin vaiheista kaikkein halvinta (1 %). (Kasurinen 2013, 17-18.)



Kuva 7. Testauksessa löydetyn virheen korjauksen kustannukset (Kasurinen 2013, 18)

Mahdollisimman virheettömän tuotteen kehittäminen on sen kehittäjälle myös siinä mielessä hyödyllistä, että tulevaisuudessa kyseisen yrityksen asiakaskunta saattaa kasvaa. Toisaalta, jos taas on tiedossa, että jokin yritys on aiemmin tuottanut paljon virheitä sisältäneitä tuotteita, niin uusia potentiaalisia asiakkaita voi olla vaikea löytää.

3 TESTAUKSEN AUTOMATISOINTI JA SEN TYÖKALUT

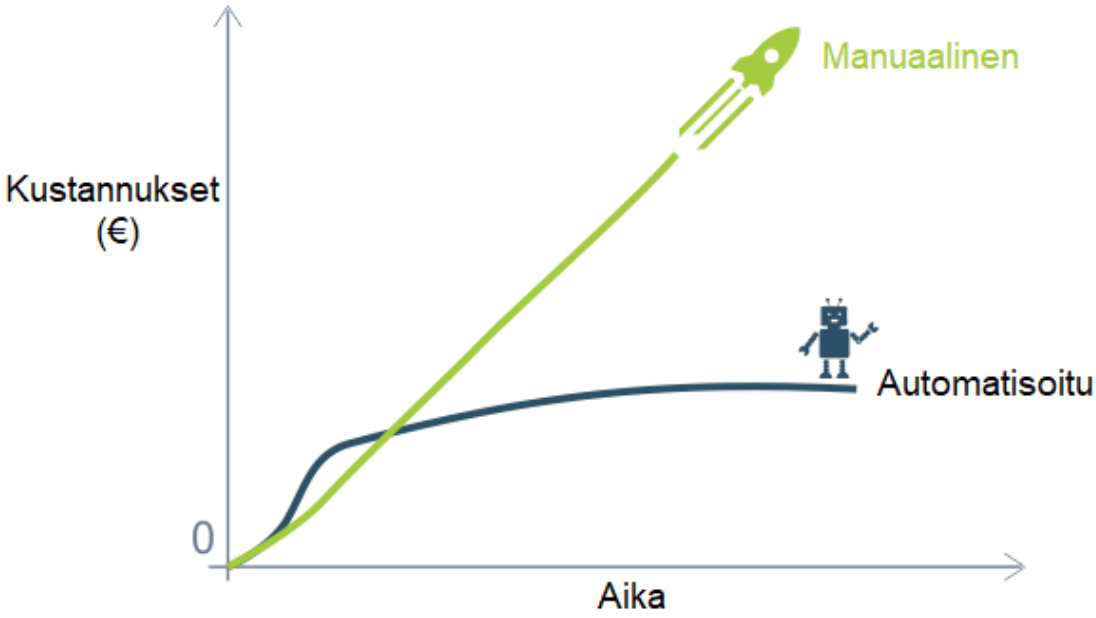
Testausautomaatio on testausmenetelmä, jossa hyödynnetään automaattityövälineitä. Testausautomaation päätavoite on automatisoida sellaisia testitapauksia, joita toistetaan useita kertoja. Tällaisia useasti toistettavia testitapauksia automatisoidessa rakennetaan niitä varten erillinen laite, joka nopeasti tarkistaa testattavat asiat. Saman testin toistaminen useasti manuaalisesti testaajan toimesta vie testausresursseja muulta, mahdollisesti vaativammalta, testaukselta. (Kasurinen 2013, 76.) Automatisoinnissa on kuitenkin hyvä muistaa, että ihminen on testausjärjestelmässä tärkein osa. Lisäksi täytyy olla toimiva manuaalinen testausjärjestelmä, ennen kuin automatisointia kannattaa edes harkita. (Pohjolainen 2003, 23.) Testausautomaatio on tarkoitettu nimenomaan jo toimiviksi todettujen asioiden testaamiseen regression varalta, ei niinkään uusien toiminnallisuuksien testaamiseen.

Otollisimpia käyttökohteita testausautomaatiolle ovat muun muassa ohjelmiston moduulien rajapinnat sekä myös yksittäisten moduulien yksikkötestauksessa testattavat asiat (Kasurinen 2013, 76). Lisäksi myös käyttöliittymän toimintojen testaaminen on oiva kohde automatisoinnille. Erilaiset toimintojennauhoitusohjelmat voidaan laittaa käymään läpi luotuja testitapauksia, joissa käsky- ja toimenpidesarjat suorittavat haluttuja testattavia asioita. Ohjelma etenee listattujen käsky- ja toimenpidesarjojen mukaan ja tekee tarkistuksia niiden mukaisesti. Esimerkiksi, jos jokin tietty ikkuna ei avaudu tai napin painallus sivulla ei tuota haluttua lopputulosta, niin ohjelma ilmoittaa epäonnistumisesta. (Kasurinen 2013, 76).

3.1 Automatisoinnin hyödyt

Testauksen automatisoinnin hyödyistä ehkä yksi suurimmista on se, että testaajien resursseja voidaan hyödyntää muihin työtehtäviin perinteisten usein toistettavien testitapausten suorittamisen sijaan. Joka kerta, kun ohjelmiston koodia muutetaan, tulisi muutoksia koskevat ohjelmiston moduulit testata muun muassa regression varalta. Tällaisten testien ajaminen manuaalisesti testaajien toimesta on kallista ja aikaa vievää. Kun useasti toistettava testitapaus on automatisoitu, voidaan sitä ajaa uudestaan ja uudestaan käytännössä

ilman minkäänlaisia lisäkustannuksia (kuva 8). (Smartbear s.a.) Pohjolainen (2003, 23) toteaa, että testitapauksen ajaminen manuaalisesti on usein nopeampaa kuin samaisen testitapauksen ajaminen automaatiota hyödyntäen. Tämä on ristiriidassa Smartbearin (s.a.) ’Why automated testing?’-artikkelin näkemyksen kanssa, sillä artikkelin mukaan automatisoidut testitapaukset ovat huomattavasti nopeampia ajaa kuin manuaaliset testitapaukset. Näistä voidaan kuitenkin tehdä johtopäätös, että testauksen automatisoinnin kannattavuus ilmenee nimenomaan testin toistojen määrän kasvaessa.



Kuva 8. Kustannuskäyrä automatisoitu testaus vs. manuaalinen testaus (Kunda 2016)

Automatisointi auttaa myös inhimillisten virheiden kitkemisessä. Kun testaaja suorittaa manuaalisesti määrättyjä testitapauksia päivästä toiseen, niin ennen pitkään testaaja tekee virheitä. Automatisoitu testitapaus suorittaa sille määrätyt käsky- ja toimenpidesarjat aina tismalleen samalla tavalla. Lisäksi jokainen suorituskerta tuottaa yksityiskohtaiset tulokset testin päätyttyä onnistuneesti tai epäonnistuneesti. (Smartbear s.a.)

Testien kattavuus kasvaa automatisoinnin myötä. Yksi suurimmista hyödyistä testien kattavuuteen liittyen on se, että ohjelmiston testaajat voidaan vapauttaa suorittamaan muita testauksia tai esimerkiksi luomaan uusia automatisoituja testitapauksia. Lisäksi ohjelmistossa harvemmin testataan pitkiä ja uuvuttavia testitapauksia manuaalisesti, jolloin kyseisten testitapausten automatisointi on yritykselle kannattavaa. (Smartbear s.a.) Testien kattavuutta voidaan

kasvattaa myös automatisoimalla sellaisia testejä, joiden suorittaminen manuaalisesti on täysin mahdotonta (Pohjolainen 2003, 24). Varsinkin, kun testataan ohjelmiston toimivuutta tietyn kuormituksen alaisena, tulee automatisointi loistamaan. Kuormituksen vaikutusta on melkeinpä mahdotonta testata manuaalisesti, sillä ohjelmiston samanaikaisia käyttäjiä ei saada tätä kautta tarpeeksi. Automatisoinnilla voidaan simuloida satojen tai jopa tuhansien samanaikaisten käyttäjien aiheuttamaa kuormitusta ohjelmistossa. (Smartbear s.a.)

Jos automatisoidut testitapaukset ovat luotu yhdenmukaisesti ajatellen kehittäjäympäristöä (DEV) ja testiympäristöä (QA), niin voidaan samoja testejä suorittaa kehittäjien ja testaajien toimesta. Tällöin kehittäjien suorittaessa automatisoituja testejä jo ohjelmiston kehitysvaiheessa, saadaan virheet kiinni nopeasti, jolloin niihin reagoiminen ja niiden korjaaminen vauhdittuu. (Smartbear s.a.) Testitapausten yhdenmukaisuus myös edesauttaa niiden yhdenmukaista toimintaa ympäristöstä riippumatta, jolloin itse testitapauksissa olevien virheiden karsiminen on helpompaa.

Edellä mainittujen hyötyjen lisäksi myös testaustiimin ajattelumaailma kehittyy ja moraali kasvaa automatisoinnin myötä. Kun testaajilla ei mene kaikki aika usein toistettavien testien suorittamiseen, jää aikaa haastavimmille ja palkitsevimmille työtehtäville. Lisäksi itse testaustyöstä vähenee rutiininomainen toistuva työ, mikä taas voi olla jaksamisen kannalta hyväksi. (Smartbear s.a.)

3.2 Automatisoinnin ongelmat

Yksi ensimmäisistä mietinnän kohteista automaatiotestausta tehdessä on se, onko testattava tapaus ylipäätään kannattavaa automatisoida. Jos tiettyä asiaa testataan lähteestä riippuen yli 4-20 kertaa, niin silloin automatisointi voi olla yritykselle kannattavaa (Kasurinen 2013, 77). Automatisoidessa tulee ottaa huomioon, että yksittäisen testitapauksen manuaaliseen testaamiseen verrattuna automatisoidussa testauksessa menee 3-10 kertaa kauemmin (Pohjolainen 2003, 7). Toisaalta automatisoinnin tuomat säästöthän syntyvät nimenomaan toistojen määrän kasvusta.

Toinen suuri ongelma liittyy ennakkoluuloihin testauksen automatisointia kohtaan. Automatisointi ei poista tarvetta manuaaliselta testaukselta, sillä manuaalisessa testauksessa on mukana muun muassa ihmisen käytöksen ennaltaarvaamattomuus ja yleensäkin havainnointikyky. Luulo, että automatisoidut testitapaukset löytävät paljon uusia virheitä, on tosiasiassa väärä. Automatisoinnin myötä saatetaan ensimmäisellä testiajolla löytää uusia virheitä, mutta sen jälkeisten testiajojen löytämien virheiden määrä on todennäköisesti pieni. (Pohjolainen 2003, 39.) Tämä johtuu siitä, että automatisoidut testitapaukset noudattavat joka kerta samaa kaavaa. Myös päinvastaisessa tilanteessa, jossa virheitä ei löydy esimerkiksi ensimmäisellä testikerralla, on haitallista ajatella, ettei ohjelmistossa olisi virheitä lainkaan. Virheiden löytymättömyys voi kertoa muun muassa testitapausten virheellisyydestä (Pohjolainen 2003, 39).

Automatisointiin voi liittyä ongelmia myös itse testityökalujen parissa. Testityökalun käyttöönotossa voi olla haasteita, minkä vuoksi testauksen automatisointia ei välttämättä saada edes toteutettua. Kesken automatisointiprojektin saatetaan myös huomata, ettei valitusta testityökalusta löydy tiettyjä ominaisuuksia ja/tai piirteitä suorittamaan jotakin haluttua toimenpidettä. Lisäksi yrityksen henkilöstön osaamattomuus testityökalun käyttöön liittyen heijastaa myös testauksen automatisoinnin onnistumiseen.

Kun automatisoituja testitapauksia on saatu työstettyä halutuista testattavista asioista, siirtyvät luodut testitapaukset ylläpitoon. Näiden testitapausten ylläpito saattaa osoittautua haasteelliseksi tai joskus jopa mahdottomaksi. (Pohjolainen 2003, 39.) Ensinnäkin ohjelmistoon tehdyt koodistomuutokset voivat vaikuttaa itse testitapauksen rakenteeseen, jolloin olisi aiheellista muuttaa testitapaus noudattamaan muutetun lähdekoodin rakennetta. Toisekseen ylläpidolle ei välttämättä ole varattu tarpeeksi resursseja, jolloin testitapaukset saattavat lakata toimimasta esimerkiksi aikaisemmin mainittujen koodistomuutosten ja niihin reagoimattomuuden johdosta.

Testauksen automatisoinnin ongelmat heijastuvat pääosin juuri yrityksen epärealistisista odotuksista automatisoinnin suhteen sekä resurssien varaamisen puutteesta. Vääriä ajatusmalleja ovat esimerkiksi se, että automatisointi ratkai-

sisi kaikki ongelmat, tai että kaikki testaaminen voitaisiin automatisoida (Kasurinen 2013, 79). Toinen pääkohta, josta automatisoinnin ongelmat juontavat juurensa, on yrityksen varaamien resurssien puute testauksen automatisointiin. Ylläpitovaihe saatetaan unohtaa, testaustyökalujen valintaan ei varata tarpeeksi aikaa ja osaamista, eikä käyttäjille anneta tarpeeksi opetusta työkalujen käyttöä ajatellen (Pohjolainen 2003, 39). Yrityksen on osattava ottaa testauksen automatisointi huomioon niin, että sille varataan tarpeeksi resursseja ja että se tukee muuta testaustoimintaa.

3.3 Automatisoinnin eri työkalut

Testausta voidaan automatisoida useita eri työkaluja hyödyntäen. Automatisoinnin eri testaustyökaluja ovat muun muassa testipetigeneraattorit, testitapausgeneraattorit, vertailijat ja testikattavuustyökalut (Haikala & Märijärvi 2004, 297). Esimerkkeinä käytössä olevista työkaluista ovat muun muassa TestComplete, Selenium, UFT (Unified Functional Testing), AutoIt sekä Robot Framework.

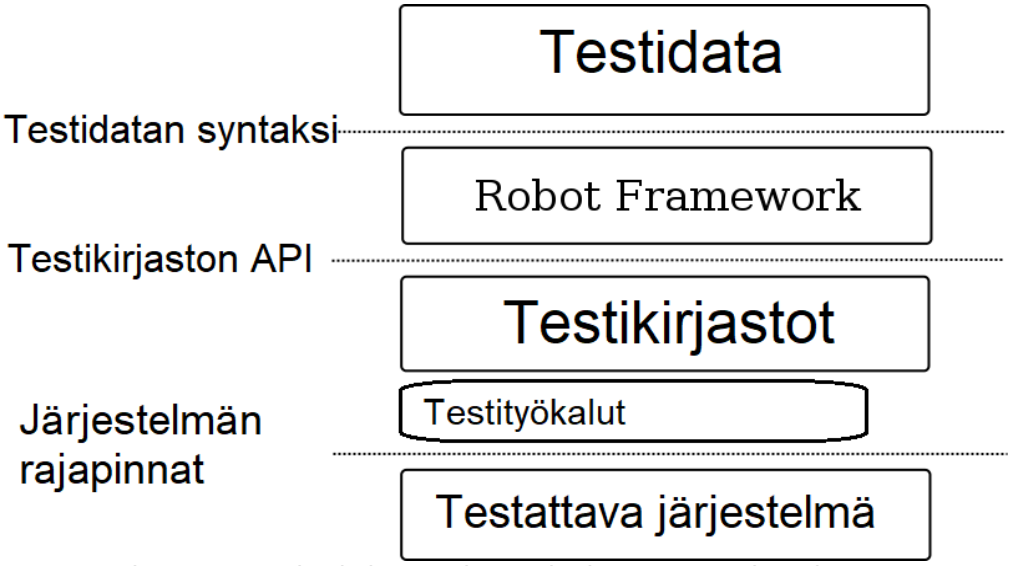
Testipetigeneraattorin (test bed generator) tarkoituksena on luoda testipeti testattavalle ohjelmamoduulille. Tälle luodulle testipedille voidaan testikuvauskielellä kuvata ajettava testi. Samaisella testikuvauskielellä voidaan myös kuvata testitapauksen halutut tulokset, jolloin myös testin tulosten tarkastelu on automatisoitu. (Haikala & Märijärvi 2004, 297.) Tähän kategoriaan voidaan laskea myös ns. ”play back” -työkalut, joilla nauhoitetaan halutun testitilanteen syötteet, jolloin testiä ajaessa toistetaan nämä samat syötteet (Haikala & Märijärvi 2004, 297). Esimerkiksi TestComplete on juuri ”play back” -työkalu.

Testitapausgeneraattorin hyödyntäminen on usein mahdotonta. Testitapausgeneraattorin idea on pyrkiä generoimaan testiaineisto esimerkiksi ohjelmiston käyttöliittymän spesifioidun tilakaavion pohjalta. Useimmiten ohjelmiston toiminnasta ei ole kuvattuna spesifioitua tilakaaviota, joten testitapausten generointi ei ole mahdollista. Mikäli testitapaus voidaan generoida automatisoidusti, voidaan myös testitapauksen tuloksia tarkastella automatisoidusti. (Haikala & Märijärvi 2004, 297.)

Vertailuohjelmia käytetään vertaamaan ajetun testitapauksen tulostetta aikaisempien testiajojen tulosteisiin. Haikala ja Märijärvi (2004, 297) kertovat kirjassaan, että vertailuohjelmien käyttöön liittyy ongelmia tulosteiden muuttuvien tietojen osalta, esimerkiksi muuttuva päivämäärä. Vertailuohjelmia voidaan kuitenkin hyödyntää, jos tulosteiden muuttuvia tietoja, esimerkiksi aikatietoja, ei sisällytetä tarkastukseen.

Testikattavuustyökalut, eli toiselta nimeltään testikattavuusanalysaattorit, ovat työkaluja, joilla mitataan testin kattavuutta. Testikattavuusanalysaattorien toimintaperiaate on esiprosessori, joka instrumentoi ohjelmiston moduulin koodia. Suorittaessaan testikattavuusanalysaattori mittaa testikattavuuden. Analysaattoreita voidaan myös hyödyntää mittaamaan ohjelmarivien suorituskertojen lukumäärää sekä prosessoriajan käyttöä. Testikattavuusanalysaattorien käyttöä järjestelmätestauksessa rajoittaa instrumentoinnista johtuva ohjelmakoodin koon kasvu sekä suorituksen hidastuminen. (Haikala & Märijärvi 2004, 297.)

Esimerkkeinä mainituista automatisoinnin työkaluista Robot Framework on yhdistelmä testipetigeneraattoria ja vertailuohjelmaa. Opinnäytetyöni tapauksessa toteutetaan testauksen automatisointi nimenomaan Robot Framework -työkalua hyödyntäen. Robot Framework on geneerinen testausautomaation runko, jolla voidaan automatisoida etenkin hyväksymis- ja regressiotestausta. Se hyödyntää helppokäyttöistä taulukkomuotoista testidataa sekä avainsanoja (keywords). Robot Frameworkissa testitapaus (test case) koostuu avainsanoista. Avainsanoja on sisäänrakennettuna, ja niitä voi myös käyttäjä itse rakentaa sisäänrakennettujen pohjalta. Käyttäjä voi hyödyntää verkosta löytyviä testikirjastoja (test libraries), jotka ovat luotu Pythonille tai Javalle käytettäväksi. Kuvassa 9 nähdään Robot Frameworkin toiminnan rakenne. Testiajon käynnistyksen yhteydessä Robot Framework prosessoi testidatan. Robot Frameworkin ei tarvitse tietää kohdejärjestelmää, vaan se hyödyntää testikirjastoja ollakseen vuorovaikutuksessa kohde järjestelmän kanssa. Testikirjastot taas käyttävät eri rajapintoja tai muita testityökaluja ajureina. Ajurit käskevät järjestelmää tekemään testissä määriteltyjä asioita, esimerkiksi avaamaan Edge-selaimen Googlen etusivulle. (Robot Framework s.a.)



Kuva 9. Robot Framework toiminnan rakenne (Robot Framework s.a.)

Robot Frameworkiin, sen asennukseen sekä käyttöön tullaan perehtymään tarkemmin itse 2M-IT:n tapauksessa.

4 CASE 2M-IT OY: TESTAUKSEN AUTOMATISOINNIN TOTEUTTAMINEN ROBOT FRAMEWORKILLA

Opinnäytetyöni käytännön osuus keskittyy avaamaan testauksen automatisoinnin toteuttamista Robot Framework -työkalua hyödyntäen. Automatisointi kohdistuu toimeksiantajani 2M-IT Oy:n tarjoamaan tuotteeseen Hyvikseen. Hyvis (www.hyvis.fi) on terveydenhuollon sähköinen asiointipalvelu, joka yhdistää kansalaisen ja terveydenhuollon ammattilaisen luottamuksellisesti ja turvallisesti. Hyviksen käyttäjiin tullaan viittaamaan opinnäytetyössäni termein *kansalainen* ja *ammattilainen*, helpottaakseni lukijaa ymmärtämään käyttäjien eri roolien merkityksen eri testitapauksissa.

Testaus on merkittävä osa Hyviksen tuotekehitystä. Testauksessa haluttiin ottaa mukaan automatisointi, joten alkuvuodesta 2018 aloitettiin Eficode Oy:n johdolla automatisoinnin pilotointivaihe. 2M-IT:ltä mukana oli pääasiassa minun lisäksi ohjaajani Markus Peltonen. Pilotointivaihe kesti neljä (4) viikkoa, jonka aikana oli tarkoitus perehtyä automatisoinnin työkaluihin (tässä tapauksessa Robot Frameworkiin) mahdollisimman syvällisesti, sekä myös saada tuotettua jo valmiita automatisoituja testitapauksia niin paljon kuin mahdollista. Pilotointivaiheen jälkeen automatisoinnin jatkotyöstäminen siirtyi 2M-IT:n henkilöiden, eli käytännössä minun ja Markus Peltosen vastuulle.

Tämä luku pyrkii esittelemään vaiheet 2M-IT:n testauksen nykytilanteesta aina testauksen automatisoinnin onnistumisen arviointiin. Tulen ottamaan kantaa myös automatisoinnin pilotointivaiheen etenemiseen, pääasiallisen sisällön ollessa Robot Frameworkin käyttöön keskittyvää. Tähän sisältyy muun muassa tarvittavien asennusten esittely sekä käyttöönotto, avainsanojen eli keywordien rooli, kokonaisten testitapausten (test case) rakentaminen, valmiiden testien ajaminen ja näiden ajettujen testien tulosten tarkastelu. Vaikka käytännön osuudessa kerrotaan osittain myös 2M-IT:n testaukseen liittyvästä toiminnasta, niin tavoitteenani on saada lukijan ymmärrys sille tasolle, että hän voi itsekin automaattitestata Robot Framework -työkalua hyödyntäen.

4.1 Testauksen nykytilanne 2M-IT:llä

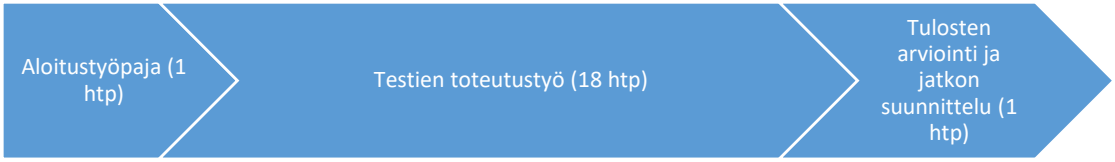
2M-IT:llä testauksesta vastaa oma testaustiimi. Se koostuu pääasiassa viidestä (5) henkilöstä. Testaustiimin kesken on jaettu vastuualueita Hyviksen eri toiminteisiin liittyen, esimerkiksi yksi vastaa lomakkeista, toinen sisällöntuotannollisista asioista jne. 2M-IT:n testaustoiminta koostuu pääosin regressio- ja savutestauksesta. Tämän lisäksi testataan tiettyihin virheisiin kohdistetut korjaukset Atlassianin Jira-tikettilistaa hyödyntäen. Jokaisen päivityksen yhteydessä tulee ajaa vähintäänkin savutestit jokaisella eri alueella, jolle päivitys on viety. Testiympäristöön eli QA-ympäristöön, viedään välillä öisin niin sanottu *nightly build,* jonka mukana tulee uusimmat virhekorjaukset. Tällöin myös jokainen nightly buildikin tulisi vähintään savutestata aamuisin. Ohjelmiston kehittäjät taas vastaavat omista testauksistaan kehitysympäristössä (DEV) ennen muutosten viemistä testiympäristöön.

Tällä hetkellä 2M-IT:llä käytännössä kaikki testaus toteutetaan manuaalisesti eli käsin testaustiimin toimesta. Automatisointi tulee auttamaan kohdistamaan testaustiimin resursseja muuhunkin kuin useasti toistettavien testien käsin ajamiseen. Tarkoituksena on nimenomaan automatisoida regressio- ja savutestauksen testitapauksia.

4.2 Automatisoinnin alkuvalmistelut

Testauksen automatisoinnin pilotointivaihe käynnistyi kickoff-palaverilla 11. tammikuuta. Kickoff-palaverissa käytiin läpi pilotointivaiheen tavoitteita ja aikataulutusta. Automatisoinnin pilotointivaihe kesti neljä (4) viikkoa (n. 20 htp per

henkilö), jonka aikana rakennettiin tekninen runko, johon oli mahdollista toteuttaa ajettavia testitapauksia. Pilotointivaihe koostui aloitustyöpajasta, testien toteutustyöstä sekä tulosten arvioinnista ja jatkon suunnittelusta (kuva 10). Aloitustyöpajassa kävimme läpi eri ympäristöt ja niihin pääsyn, eri kirjautumistavat ja niiden erot, sekä listasimme ja priorisoimme automatisoitavat testitapaukset. Testien toteutustyö käsitti suurimman osan koko pilotointivaiheesta. Pilotointivaiheen tuloksia arvioitiin ja mahdollista jatkoa suunniteltiin omassa työpajassaan. Työpajassa muun muassa todettiin pilotoinnin saavutukset ja hyödyt, listattiin kehityskohteet, joihin panostaa jatkossa, sekä arvioitiin jatkoautomatisoinnin mahdollisuuksia.



Kuva 10. Pilotoinnin vaiheet

Testit kirjoitettiin Robot Frameworkin avulla. Pilotointivaiheessa pyrittiin kirjoittamaan mahdollisimman monta ennakkoon listattua ja priorisoitua automatisoitavaa testitapausta. Lisäksi suunniteltiin, mitä kaikkea kannattaa ja on mahdollista automatisoida.

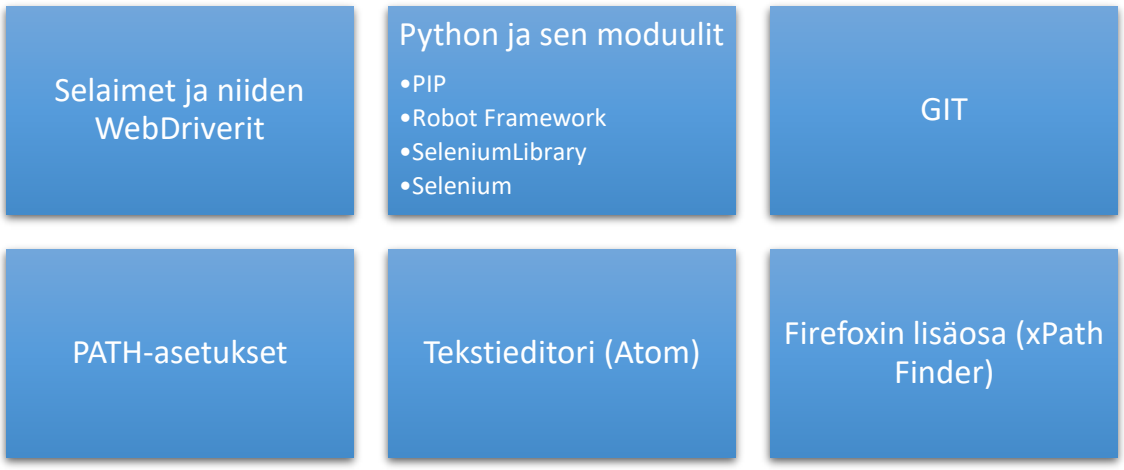
Pilotointivaiheessa asetettiin tiettyjä rajauksia ja oletuksia. Pilotointivaiheessa testien ajaminen käynnistettiin käsin komentokehotteen (Command Prompt) kautta Jenkinsin sijaan. Jenkins on webpohjainen jatkuvan integraation sovellus, jolla voidaan monitoroida toistuvasti suoritettavien tehtävien suoritusta. Avoimen lähdekoodin MIT-lisenssin omaava Jenkins on oiva työkalu automatisoida myös testijoukon ajon käynnistäminen, mutta sen hyödyntämistä mietittiin pilotointivaiheen päätyttyä. (Laitinen 2011, 18.)

Testaustiimin osalta automatisoinnin pilotointivaiheeseen päästiin mukaan 25.1 pidetyssä automatisoinnin työpajassa. Pilotointivaiheen vetäjänä toiminut Eficode Oy:n asiantuntija oli jo tähän mennessä pystynyt luomaan pohjia automatisoinnille, muun muassa tarvittavat rakenteet eri ympäristöille (DEV ja QA) sekä jo joitain valmiita testitapauksia. Jotta projekti saatiin käyntiin myös testaustiimin osalta, käytiin työpajassa läpi automatisointiin tarvittavien ohjelmistojen listaa sekä katselmoitiin jo valmiita testitapauksia ja niiden rakennetta.

Tämän lisäksi sovittiin testikäyttäjälistan luomisesta sekä tehtiin alustava lista automatisoitavista testitapauksista ammattilaisen ja kansalaisen toiminteiden osalta. Aivan työpajan lopuksi Eficode Oy:n asiantuntija demosi meille 2M-IT:n henkilöille testitapausten rakentamista ja testien ajamista. Kickoff-palaverin sekä työpajan oppeja hyödyntäen testaustiimi pystyi lähteä työstämään testauksen automatisointia itsenäisesti.

4.3 Asennukset ja käyttöönotto

Testauksen automatisoinnin toteuttamiseksi Robot Frameworkilla tarvittiin useita eri asennuksia muun muassa selaimia, web drivereita ja Python moduuleja. Kuvassa 11 nähdään kaikki läpikäytävät asennukset. Asennuksissa tuli kiinnittää huomiota siihen, että asennettiin eri ohjelmistoista sellaiset versiot, jotka vastasivat käyttöjärjestelmän versiota, eli joko 32-bittinen tai 64-bittinen. Opinnäytetyössä asennukset tehtiin Windows 10 64-bittisellä käyttöjärjestelmällä. Tarvittavat asennukset teimme yhdessä ohjaajani Markus Peltosen kanssa. Asennettujen ohjelmien ja lisäosien asennuslinkit löytyvät liitteestä 1.



Kuva 11. Robot Frameworkin tarvittavat asennukset

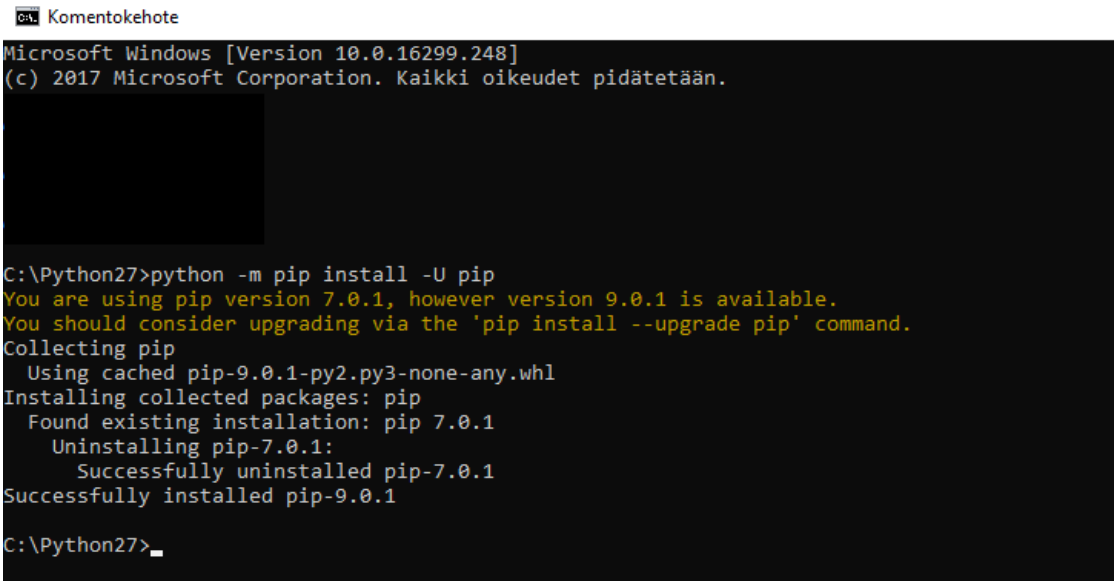
Koska tarkoituksena oli automatisoida nimenomaan verkossa toimivan sähköisen asiointipalvelu Hyviksen eri testitapauksia, niin ensimmäinen oleellinen asennettava kokonaisuus oli eri selaimet. Kaikki suosituimmat selaimet (Internet Explorer, Microsoft Edge, Google Chrome ja Mozilla Firefox) olisi hyvä sisällyttää testiajoihin, sillä eri toiminteet saattavat toimia eri tavoin selaimesta

riippuen. Tässä vaiheessa automatisointia olimme kuitenkin päättäneet keskittyä etenkin Chromeen ja Firefoxiin testeissämme havaituista IE:n ja Edgen erinäisistä ongelmista johtuen. Asensimme Chromesta ja Firefoxista viimeisimmät versiot.

Jokaiselle selaimelle, jolla halusimme automatisoituja testitapauksia ajaa, jouduimme asentamaan erikseen oman WebDriverin. WebDriver on avoimen lähdekoodin työkalu, jolla voidaan automatisoida websovelluksia eri selaimilla. WebDriver tarjoaa mahdollisuuksia muun muassa navigoida verkkosivustolle, tuottaa käyttäjän syötteitä sekä suorittaa esimerkiksi JavaScript-komentoja (Google s.a.). Chromelle asennettiin sen oma WebDriver nimeltään ChromeDriver ja versioksi viimeisin (2.3.5). Windows-käyttöjärjestelmälle löytyi vain 32-bittisen version .zip-paketti, joka ladattiin ja purettiin. Firefox-selaimen oma WebDriver on nimeltään Firefox Gecko. Firefox Gecko:sta ladattiin ja purettiin viimeisimmän version (v.0.19.1) 64-bittiselle Windowsille tarkoitettu .zip-paketti. Tässä kohtaa oli hyvä luoda C-aseman juureen kansio nimellä *webdrivers.* Kyseiseen kansioon siirrettiin kaikki ladatut ja puretut WebDriverit. Kansion merkitykseen perehdytään myöhemmin PATH-asetusten konfiguroinnin yhteydessä. Selaimia päivittäessä on hyvä muistaa päivittää myös niiden omat WebDriverit, jotta automatisointi toimii moitteettomasti.

Toinen oleellinen osa asennuksia oli Python. Python on yleiskäyttöinen, oliopohjaiseen ohjelmointiin hyvin soveltuva, avoimen lähdekoodin ohjelmointikieli, joka on saatavilla ilmaiseksi. Se pohjautuu muun muassa ABC-, Modula-3- ja Smalltalk-ohjelmointikieliin. Python on tunnettu erityisesti sen selkeydestä ja helppokäyttöisyydestä. Sitä voidaan käyttää komentokehotteen (Command Prompt/cmd) kautta. (W3Schools s.a.) Automatisointia varten Pythonista asennettiin Python 2.7.10 64-bittinen versio. Asennusvaiheessa oli hyvä muistaa lisätä Pythonin hakemistot PATH:iin. Tämä tapahtui lisäämällä ruksi kohtaan ”Add Python.exe to Path”. Tällöin asennusohjelma automaattisesti lisäsi PATH-muuttujaan Pythonin suoritettavien ohjelmien hakemistot (directories) C:\Python27 ja C:\Python27\Scripts. Muutoin jouduttaisiin lisäämään kyseiset hakemistot käsin myöhemmin. PATH on Unix-tyyppisten käyttöjärjestelmien (mm. Windows) ympäristömuuttuja, joka sisältää suoritettavien ohjelmien hakemistojen sijaintitiedon. (Java s.a.)

Seuraavana vuorossa oli Pythonin eri moduulien asennus. Kun Python oli onnistuneesti asennettu, niin aloitettiin päivittämällä PIP. PIP on työkalu, jolla voidaan helposti asentaa eri Python moduuleja. Asennetun Python version (2.7.10) mukana tuli automaattisesti PIP. Itse PIP:in päivittäminen onnistui alla olevalla Pythonin pip-komennolla komentokehotteessa (cmd) (kuva 12).



Kuva 12. Pythonin PIP-työkalun päivittäminen komentokehotteessa

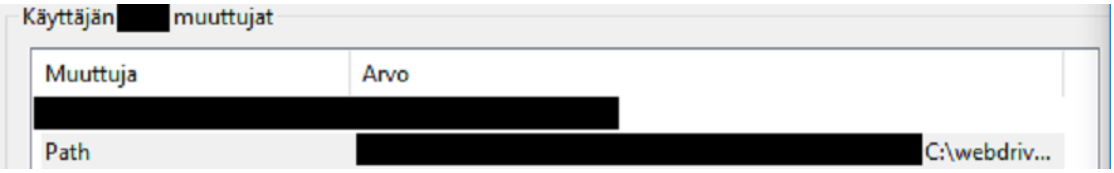
PIP-työkalulla asennettiin myös itse Robot Framework. Niin kuin aikaisemmin mainitsin, Robot Framework on avainsanoihin (keywords) perustuva testityökalu. Asennus ja päivitys uusimpaan versioon tapahtui komentokehotteessa Pythonin pip-komennolla *'python -m pip install -U robotframework'*.

Robot Frameworkia varten tarvitsimme myös testikirjaston (test library), jota se hyödyntää. SeleniumLibrary on Robot Frameworkille tarkoitettu verkkotestauskirjasto, joka hyödyntää Selenium-työkalua sisäisesti (GitHub s.a.). Kaikki testeissä käytettävät jo sisäänrakennetut avainsanat (keywords) löytyvät kyseisestä testikirjastosta. Käyttäjä voi luoda omia avainsanojaan sisällyttämällä siihen näitä valmiita avainsanoja. Avainsanoista tarkemmin omassa luvussaan. SeleniumLibrary asennettiin ja päivitettiin uusimpaan versioon komentokehotteessa Pythonin pip-komennolla *'python -m pip install -U robotframework-seleniumlibrary'*.

SeleniumLibrarya varten tarvitsimme Seleniumin. Selenium pitää sisällään joukon työkaluja ja testikirjastoja, jotka mahdollistavat verkkoselaimella testaamisen. Selenium oli helppo asentaa ja päivittää uusimpaan versioon komentokehotteessa Pythonin pip-komennolla *'python -m pip install -U selenium'*.

2M-IT:n automatisointiprojektissa versionhallinta oli toteutettu Git-versionhallintajärjestelmän avulla. Kaikki projektin sisältämät tiedostot yms. löytyivät Gitin säilytyspaikasta (repository). Projektiin tehtyjä muutoksia tallennettiin ja puskettiin (push) osissa (commit) säilytyspaikkaan. Näin projektin versionhallinta oli helppoa, kun pystyimme osittain erottelemaan eri versiot ja niiden sisällöt. 2M-IT:n tapauksessa kyseistä Git-säilytyspaikkaa ylläpidettiin Atlassianin omistaman Bitbucket hosting-palvelun kautta. Tapauksessamme Gitistä asennettiin uusin 64-bittinen versio.

Robot Frameworkin toimivuuden kannalta ehkä yksi tärkeimmistä tehtävistä asioista oli PATH-ympäristömuuttujan asetusten määrittäminen. Aikaisemmin mainitsin C-aseman juureen luotavasta *webdrivers*-kansiosta. Viimeistään tässä kohtaa kyseisen hakemiston, sekä sinne asennettavien selainten WebDrivereiden, oli löydyttävä. Määrittelimme molempiin, käyttäjätilin ympäristömuuttujiin (kuva 13) sekä järjestelmän ympäristömuuttujiin, PATH-asetuksiin luomamme hakemiston C:\webdrivers.



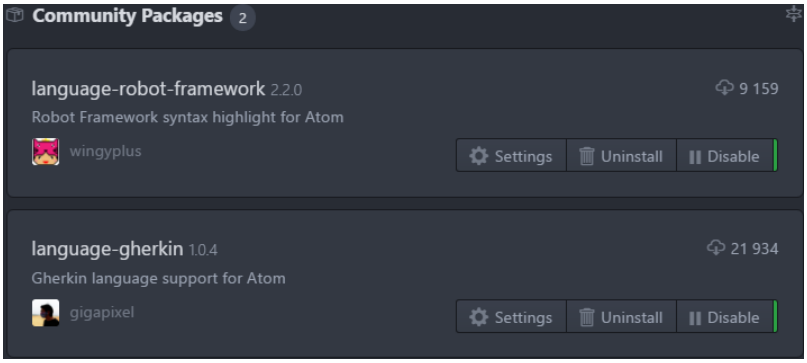
Kuva 13. Käyttäjätilin ympäristömuuttujiin Path:iin määritelty C:\webdrivers -hakemisto

Mikäli Pythonin asennuksessa olisi unohtunut laittaa ruksi kohtaan ”Add Python.exe to Path”, niin käyttäjä joutuisi lisäämään hakemistot C:\Python27 ja C:\Python27\Scripts samalla tavalla käyttäjätilin sekä järjestelmän Path-ympäristömuuttujaan kuin jo lisätty C:\webdrivers.

Robot Frameworkin SeleniumLibrary käyttää verkkosivuston elementtien tunnistukseen muun muassa tapoja: id, name, class, tag, xpath ja css. Kätevä tapa viitata elementtiin on varsinkin xpath. Xpathin löytämiseksi tarvitaan selaimeen lisäosa: Firefoxilla xPath Finder ja Chromella XPath Helper. 2M-IT:n

tapauksessa asensimme Firefoxiin xPath Finderin. Suurin osa elementtiviittauksista tehtiin nimenomaan xpathia hyödyntäen.

Robot Frameworkin ajettavat testit on kirjoitettava tekstieditorilla. Valitsimme käytettäväksi Atom-tekstieditorin. Atom on avoimen lähdekoodin, alustariippumaton, ohjelmointiin tarkoitettu tekstieditori. Se on helposti kustomoitava ja lisäksi se sisältää sisäänrakennetun paketinhallinnan (package manager). (Atom s.a.) Jotta testien kirjoittaminen kävi jouhevasti, asensimme Atomiin paketit *language-gherkin* ja *language-robot-framework.* Ensimmäinen paketti lisäsi Atomiin tuen Gherkin-kielelle. Jälkimmäinen paketti lisäsi Robot Frameworkin syntaksin korostuksia ja pätkiä. Onnistuneesti asentuneet paketit ilmestyivät Atomin ”Community Packages”-kohdan alle (kuva 14).



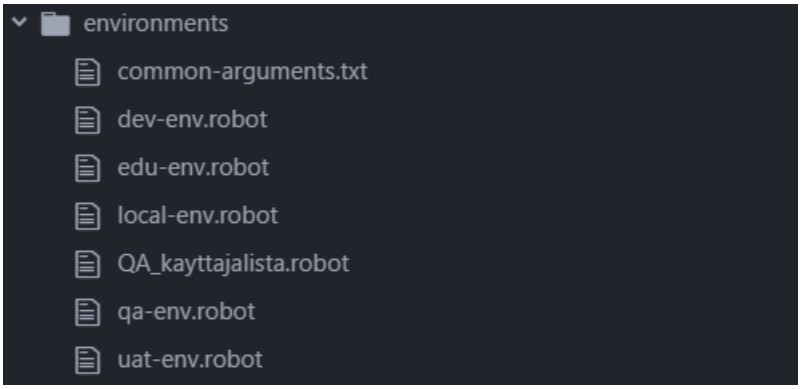
Kuva 14. Onnistuneesti asennetut language-gherkin ja language-robot-framework

Robot Frameworkin testitapausten luomiseen on olemassa myös RIDE-kehitysympäristö. RIDE:n avulla testitapausten rakentaminen on helppoa ja nopeaa. 2M-IT:n tapauksessa emme kuitenkaan ottaneet RIDE:ä alusta asti käyttöömme, vaan jätimme sen hyödyntämisen harkinnan myöhempään.

4.4 Automaatioprojektin rakenne ja sen merkitys

Ymmärtääkseen Robot Frameworkin toimintaa käyttäjällä täytyy olla ymmärrys robotiikan toimintatavoista sekä automatisointiprojektin tiedosto- ja kansiorakenteesta. Tarvittavat asennukset tehtyämme loimme perus tiedosto- ja kansiorakenteen automatisointiprojektille. Robot Frameworkin automatisoitavat testit kirjoitettiin robot-tiedostolle, joka on niin sanottu testisovellus (test suite). Testisovellus sisältää testidataa. Koska Hyvis toimii useassa eri ympäristössä (QA, DEV, EDU, UAT ja tuotanto) tarvitsimme projektiin ympäristökohtaisille testidatoille omat tiedostonsa. Siksi loimme *environments*-kansion,

joka sisälsi jokaiselle ympäristölle kohdistetun oman robot-tiedostonsa (kuva 15). Samaan kansioon sisälsimme myös *common-arguments.txt*-tiedoston (kuva 16), joka sisälsi testiajojen tuloksiin liittyviä määrittelyjä, muun muassa tuloslokin ja -raportin sijaintitiedon.



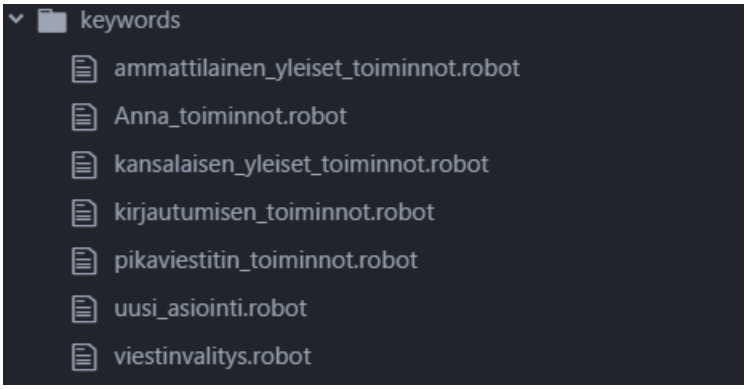
Kuva 15. Environments-kansion rakenne

Environments-kansioon lisäsimme myös ympäristökohtaisten ammattilaisten ja kansalaisten listauksen käyttäjätunnuksineen ja yhteystietoineen.

```
1  --output ./output.xml
2  --log ./log.html
3  --report ./report.html
4
```

Kuva 16. Common-arguments.txt sisältämät testien tulosten määrittelyt

Toinen kansio, jonka loimme heti aluksi, oli *keywords*-kansio. Kyseiseen kansioon pystyimme kirjoittamaan sellaisia avainsanoja, joiden käyttäminen ei ollut ympäristöriippuvaista, vaan sellaisia, jotka olivat kohdistettu johonkin tiettyyn Hyviksen toiminnallisuuteen, esimerkiksi viestinvälitykseen. Keywords-kansion alle loimme jokaiselle toiminnallisuudelle oman robot-tiedostonsa, johon kyseisen toiminnallisuuden avainsanoja kirjoitettiin (kuva 17). Esimerkiksi *viestinvalitys.robot*-tiedosto sisälsi avainsanan ”Siirry viesteihin”, jota pystyimme käyttämään eri testitapauksissa siirryttäessä viesteihin sekä ammattilaisena että kansalaisena, jokaisessa Hyviksen eri ympäristössä.



Kuva 17. Keywords-kansion sisältämät toiminnallisuuskohtaiset robot-tiedostot

Jokaisessa eri ympäristössä toimivia, yleisesti käytettäviä avainsanoja kirjoitimme *common-resource.robot*-tiedostoon. Tähän tiedostoon kirjoitimme muun muassa selaimen valitsemisen, selaimen avaamisen ja selaimen sulkemisen avainsanat. Myös usein esiintyvät näppäimet (takaisin, lähetä jne.) oli hyvä kirjoittaa kyseiseen, yleiseen käyttöön tarkoitetuista avainsanoista koostuvaan tiedostoon. Avainsanojen jaottelu eri tiedostoihin edellä mainituin tavoin edisti niiden uudelleenkäytettävyyttä.

4.5 Testien kirjoittaminen Robot Frameworkilla

Robot Frameworkin testit kirjoitettiin testisovelluksiin eli robot-tiedostoihin. Testisovellukset sisälsivät testidataa eli muun muassa asetuksia (settings), muuttujia (variables), testitapauksia (test cases) ja avainsanoja (keywords). Loimme jokaiselle testattavalle Hyviksen toiminnallisuudelle oman testisovelluksen, joka sisälsi edellä mainittuja testidatan osia. Testisovelluksen perusrakenne havainnollistetaan kuvassa 18.

Kuva 18. Testisovelluksen perusrakenne

Testisovelluksen asetuksiin pystyimme määrittelemään muun muassa käytettävät testikirjastot ja resurssit. Testikirjastoiksi määrittelimme SeleniumLibra-

ryn ja DateTimen. SeleniumLibraryn avulla saimme käyttöömme jo sisäänrakennetut avainsanat, ja DateTimen avulla pystyimme luomaan aikaleiman sekä hyödyntämään sitä eri testeissä. Resursseina käytimme tapauskohtaisesti yleiseen käyttöön tarkoitettua *common-resource.robot*-testisovellusta, ympäristökohtaisia robot-testisovelluksia sekä toiminnallisuuskohtaisia robot-testisovelluksia. Kuvan 19 esimerkissä testikirjastoiksi valitsimme juuri SeleniumLibraryn ja DateTimen, ja resursseiksi valikoitui *common-resource.robot* sekä Hyviksen QA-testiympäristön *qa-env.robot*.

testi.robot

```
*** Settings ***
Documentation   Tähän voi kirjoittaa tekstiä
Library     SeleniumLibrary
Library     DateTime
Resource    common-resource.robot
Resource    ./environments/qa-env.robot
```

Kuva 19. Esimerkkirakenne testisovelluksen asetuksista

Testisovelluksen alkuun pystyimme myös määrittelemään testeissä käytettäviä muuttujia ja niiden arvoja. Tarvitsimme Hyviksen eri ympäristöjen testausta varten etenkin ympäristökohtaisia muuttujia. Muuttujien määritteleminen helpotti uudelleenkäytettävyyttä huomattavasti. Kuvassa 20 on esimerkiksi määritelty *qa-env.robot*-testisovelluksen muuttujiin QA-testiympäristön kansalaisen kirjautumisosoite. Tietoturvasyistä johtuen kirjautumisosoite on peitetty. Luotuun muuttujaan voidaan jälkikäteen viitata suoraan, esimerkiksi avattaessa selaimen ja navigoidessa muuttujan osoittamaan osoitteeseen. Kyseinen muuttuja osoittautui projektissa erittäin hyödylliseksi, sillä eri toiminnallisuuksien testaamisessa vaadittiin aina sisäänkirjautuminen osana testitapausta.

```
*** Variables ***
${LOGIN_URL_KANSALAINEN}    https://[redacted]
```

Kuva 20. Hyviksen QA-testiympäristön kansalaisen kirjautumisosoite määriteltynä muuttujiin

Robot Frameworkin testitapausten toiminnallisuudet rakentuvat avainsanoista. Avainsanoja löytyy valmiiksi eri testikirjastoista. Käyttäjä voi myös itse luoda avainsanoja valmiiden avainsanojen pohjalta. 2M-IT:n tapauksessa hyödynsimme SeleniumLibraryn sisäänrakennettuja avainsanoja, joiden pohjalta tar-

vittaessa loimme uusia, kokoavia avainsanoja. Avainsanojen käyttäminen onnistui viittaamalla suoraan SeleniumLibraryyn, sisällyttämällä ne itse testisovellukseen tai lisäämällä ne johonkin tiettyyn robot-tiedostoon resurssiksi. Alla olevassa esimerkissä (kuva 21) nähdään, kuinka pystyimme SeleniumLibraryn sisäänrakennettujen avainsanojen avulla luomaan oman avainsanan toiminnosta ”Navigoi Hyviksen kansalaisen kirjautumissivulle QA-testiympäristössä”. Atom-tekstieditorissa käyttäjän luomat avainsanat korostuivat violetin värisinä. Asetuksissa määrittelimme, mitä resursseja kyseisessä testijoukossa käytettiin. QA-testiympäristön resurssi *qa-env.robot* sisälsi tässä esimerkissä muuttujana saman kansalaisen kirjautumisosoitteen kuin kuvassa 20. Esimerkissä luotu avainsana voisi sijaita myös resurssina *common-resource.robot*-tiedostossa, johon on viitattu asetuksissa. Kyseinen sisäänrakennettu avainsana *’Open Browser {LOGIN_URL_KANSALAINEN} Firefox’* toimisi myös itsenään ilman erikseen luotua avainsanaa.

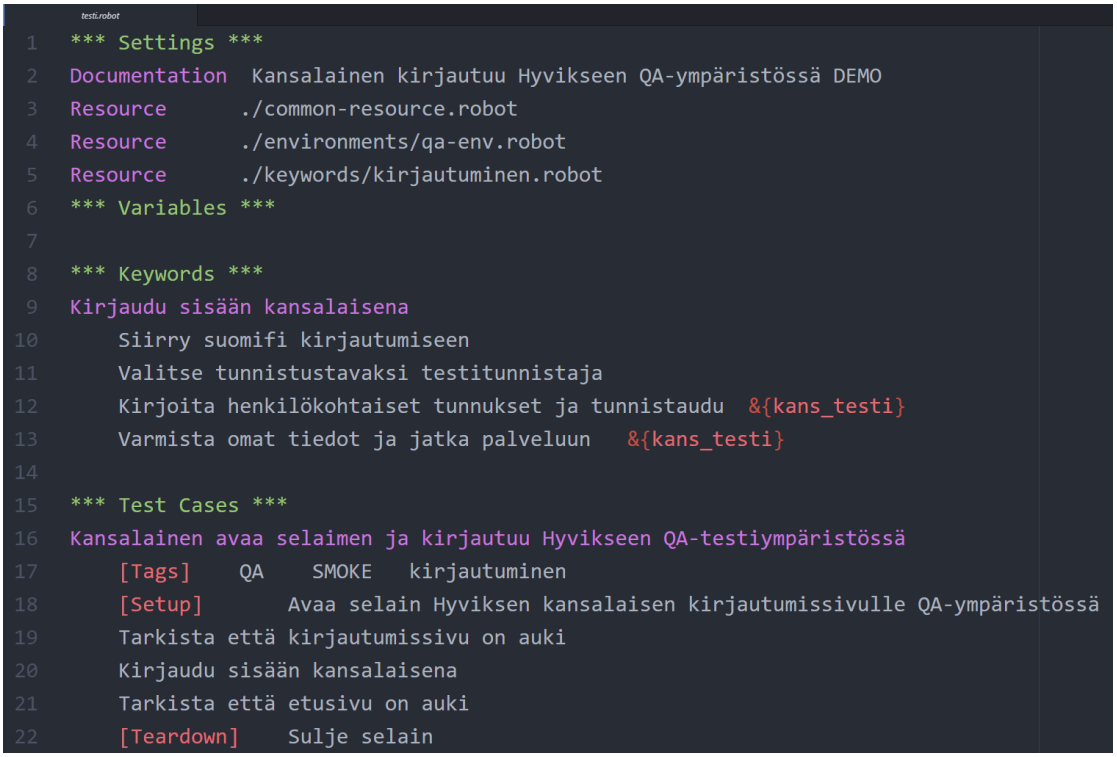
testi.robot

```
*** Settings ***
Documentation  Tähän voi kirjoittaa tekstiä
Library        SeleniumLibrary
Library        DateTime
Resource       ./common-resource.robot
Resource       ./environments/qa-env.robot
*** Variables ***

*** Keywords ***
Avaa selain Hyviksen kansalaisen kirjautumissivulle QA-ympäristössä
    Open Browser    ${LOGIN_URL_KANSALAINEN}    Firefox
```

Kuva 21. Avainsanan luominen ja sen asetukset

Avainsanat eivät yksinään riittäneet testien automatisoimiseksi. Viimeinen testisovelluksen testidatan osa, itse testitapaukset, rakennettiin edellä mainittujen asetusten, muuttujien ja avainsanojen pohjalta. Testitapauksen vaiheet rakentuivat lisäämällä siihen avainsanoja peräkkäin. Testitapausta kirjoittaessa piti huomioida myös sen omat asetukset, eli *[Tags], [Setup] ja [Teardown].* Myös testitapausten nimet erottuivat violettina Atom-tekstieditorissa.

Käyn seuraavassa esimerkissä läpi, kuinka rakensimme kokonaisen testisovelluksen toiminnosta ”Kansalainen avaa selaimen ja kirjautuu Hyvikseen QA-testiympäristössä”. Esimerkki koostuu testisovelluksista *testi.robot, kirjautuminen.robot, QA_kayttajalista.robot, qa-env.robot ja common-resource.robot.*



```
testi.robot
*** Settings ***
Documentation  Kansalainen kirjautuu Hyvikseen QA-ympäristössä DEMO
Resource       ./common-resource.robot
Resource       ./environments/qa-env.robot
Resource       ./keywords/kirjautuminen.robot
*** Variables ***

*** Keywords ***
Kirjaudu sisään kansalaisena
    Siirry suomifi kirjautumiseen
    Valitse tunnistustavaksi testitunnistaja
    Kirjoita henkilökohtaiset tunnukset ja tunnistaudu  &{kans_testi}
    Varmista omat tiedot ja jatka palveluun   &{kans_testi}

*** Test Cases ***
Kansalainen avaa selaimen ja kirjautuu Hyvikseen QA-testiympäristössä
    [Tags]    QA    SMOKE   kirjautuminen
    [Setup]       Avaa selain Hyviksen kansalaisen kirjautumissivulle QA-ympäristössä
    Tarkista että kirjautumissivu on auki
    Kirjaudu sisään kansalaisena
    Tarkista että etusivu on auki
    [Teardown]    Sulje selain
```

Kuva 22. Esimerkkitestisovelluksen *testi.robot*-rakenne

Edellä olevaan *testi.robot*-testisovellukseen (kuva 22) kerättiin kaikki testin ajamiseksi tarvittava testidata. Asetuksiin määriteltiin käytettävät resurssit. Testisovelluksessa tarvittavat muuttujat määriteltiin erikseen *common-resource.robot*-tiedostoon sekä ympäristökohtaiseen *qa-env.robot*-tiedostoon. Itse testisovelluksen avainsanoihin luotiin yksi niin sanottu ”kokoava” avainsana ’*Kirjaudu sisään kansalaisena’*, joka sisälsi kansalaisen sisäänkirjautumisen eri vaiheet omina avainsanoinaan. Kyseiset eri vaiheiden avainsanat löytyivät *kirjautuminen.robot*-tiedostosta. Itse testitapaus sisälsi tunnisteet (tags), aloituksen (setup), tarvittavat avainsanat sekä lopetuksen (teardown). Tunnisteiden avulla pystyttiin ajamaan haluttuja testejä joukkona tai yksittäin, esimerkiksi voitiin savutestata ajamalla kaikki SMOKE-tunnisteen omaavat testitapaukset. Aloituksessa määritellään jokin avainsana, joka suoritetaan ensimmäisenä testitapausajossa. Lopetuksessa taas ajetaan jokin avainsana aivan testitapausajon lopuksi. Testitapauksen sisältämät avainsanat löytyivät eri robot-tiedostoista toiminnallisuuden ja ympäristön mukaan, jotta avainsanojen uudelleenkäytettävyys oli helppoa.

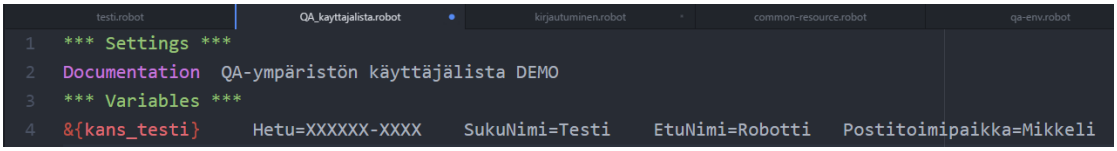
kirjautuminen.robot

```
*** Settings ***
Documentation  Kirjautumisen avainsanat
Resource       ../common-resource.robot
Resource       ../environments/QA_kayttajalista.robot
Resource       ../environments/qa-env.robot

*** Keywords ***
Avaa selain Hyviksen kansalaisen kirjautumissivulle QA-ympäristössä
    Open Browser    ${LOGIN_URL_KANSALAINEN}    ${BROWSER}
    Maximize Browser Window
    Set Selenium Speed    ${DELAY}

Tarkista että kirjautumissivu on auki
    Title Should Be    ${WELCOME_KANSALAINEN}

Siirry suomifi kirjautumiseen
    Wait Until Page Contains    Asioinnin etusivu
    Click Link                  Jatka kirjautumista tästä linkistä
    Wait Until Page Contains    Terveystietosi verkossa
    Click Element               xpath=.//*[text()[contains(., 'Kirjaudu')]]

Valitse tunnistustavaksi testitunnistaja
    Wait Until Page Contains    Valitse tunnistustapa
    Click Element               xpath=.//span[text()[contains(., 'Testitunnistaja')]]
    Wait Until Page Contains    Testitunnistaja

Kirjoita henkilökohtaiset tunnukset ja tunnistaudu
    [Arguments]     &{kayttaja}
    Wait Until Page Contains    Henkilötunnus
    Input Text                  xpath=.//input[@id='hetu_input']    ${kayttaja.Hetu}
    Click Button                Tunnistaudu

Varmista omat tiedot ja jatka palveluun
    [Arguments]     &{kayttaja}
    Wait Until Page Contains    Olet tunnistautumassa palveluun  timeout=10s
    Page Should Contain         ${kayttaja.SukuNimi}
    Page Should Contain         ${kayttaja.EtuNimi}
    Page Should Contain         ${kayttaja.Postitoimipaikka}
    Click Button                Jatka palveluun
```

Kuva 23. Esimerkkitestisovelluksen *kirjautuminen.robot* sisältämät kirjautumisen avainsanat

Testisovelluksen sisältämät kirjautumiseen liittyvät eri avainsanat löytyivät *kirjautuminen.robot*-tiedostosta (kuva 23). Kyseiset avainsanat rakennettiin valmiiden, SeleniumLibraryn sisäänrakennettujen avainsanojen pohjalta. Avainsanat sisälsivät muuttujia, jotka olivat listattuna *common-resources.robot*-tiedostoon, mikäli ne olivat yleiskäyttöisiä, ja *qa-env.robot*-tiedostoon, mikäli ne olivat QA-testiympäristökohtaisia. Avainsanojen elementtiviittauksissa hyödynsimme elementtien omien tunnisteiden (id, class, ng-model, xpath) lisäksi myös niiden sisältämää tekstiä. Kirjautumiseen vaadittiin tietyn henkilön tunnistetiedot, mistä johtuen jouduimme käyttämään argumentteja sellaisissa

avainsanoissa, joissa käyttäjäkohtaisia viittauksia ja tarkistuksia tehtiin. Argumentiksi lisättyyn *'&{kayttaja}'* pystyttiin viittaamaan *testi.robot*-tiedoston sisältämässä kokoavassa avainsanassa *'Kirjaudu sisään kansalaisena'*. Viitattava käyttäjä *'&{kans_testi}'* löytyi luomastamme *QA_kayttajalista.robot*-tiedostosta (kuva 24). Kyseiseen tiedostoon pystyimme lisäämään kaikki testeissä tarvittavat käyttäjät omine tietoineen (henkilötunnus, nimi, osoite yms.) riveittäin.



Kuva 24. *QA_kayttajalista.robot* sisältämät käyttäjien tiedot

Esimerkkitesti luotiin Hyviksen QA-testiympäristöön, joten tarvitsimme muutamia ympäristökohtaisia muuttujia: kansalaisen kirjautumisosoitteen kirjautumista varten sekä tekstimuuttujan verkkosivuston otsikon tarkistusta varten. Muuttujia tarvittiin automatisointiprojektissa useaan otteeseen, joten niiden lisääminen *qa-env.robot*-tiedostoon (kuva 25) ja niiden käyttäminen sen kautta oli järkevää. Kuvassa osa muuttujista on peitetty tietoturvasyistä johtuen.

Kuva 25. Ympäristökohtaiset muuttujat *qa-env.robot*-tiedostossa

Kaikissa testisovelluksissa oli resurssiviittaus *common-resource.robot*-tiedostoon. Kyseessä oli yleiset toiminnot ja muuttujat kokoava testisovellus, jota käytettiin resurssina. Siihen oli järkevää esimerkiksi määritellä käytettävät testikirjastot suoraan, niin kuin esimerkissä on määritelty SeleniumLibrary- ja DateTime-testikirjastot (kuva 26).

testi.robot | QA_kayttajalista.robot | kirjautuminen.robot | common-resource.robot

```
*** Settings ***
Documentation  A resource file with reusable keywords and variables. Utilizes
...            keywords by the imported SeleniumLibrary.
Library        SeleniumLibrary
Library        DateTime
Resource       ./environments/qa-env.robot
*** Variables ***
${BROWSER}      ${Firefox}
${DELAY}        0.2

*** Keywords ***
Tarkista että etusivu on auki
    Wait Until Page Contains   Etusivu

Sulje selain
    Close Browser
```

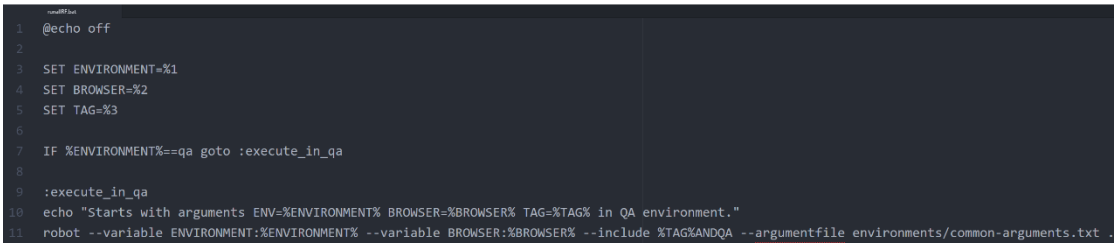
Kuva 26. Esimerkkitestisovelluksen *common-resource.robot*-tiedosto

Esimerkissä määrittelimme myös muuttujiin ajettavaksi selaimeksi Firefoxin sekä määrittelimme viivemuuttujan *'${DELAY}',* jonka avulla asetimme eri toimintojen välisen viiveen *kirjautuminen.robot*-tiedoston sisältämässä avainsanassa (kts. kuva 23). *Common-resource.robot* sisälsi myös sellaisia yleiskäyttöisiä avainsanoja, jotka toimivat ympäristöstä riippumatta. Kokonaisuutena esimerkkitestisovellus oli nyt suorittamiskelpoinen ja helposti uudelleenkäytettävä.

4.6 Testien ajaminen ja testitulosten tarkastelu

Robot Frameworkin automatisoidut testit ajettiin komentokehotteen kautta. Ajettujen testien tulokset tallennettiin vakiona XML- ja HTML-muodoissa. Testien ajamista varten loimme komentojonon eli BAT-päätteisen tiedoston. Komentojonon hyödyntäminen testien ajamisessa mahdollisti käyttäjäystävällisen tavan ajaa testejä eri muuttujilla. Testien ajamiseen liittyviä muuttujia olivat ympäristö, jossa testit ajetaan, selain sekä ajettavien testitapausten tagit. Annoimme komentojonolle nimeksi *runallRF.bat*. Robot Frameworkin testiajo saatiin käyntiin navigoimalla komentokehotteessa koko testijoukon hakemistoon (tässä tapauksessa *hyvis-rf-tests*-hakemisto) ja syöttämällä komento *runallRF.bat <ENVIRONMENT> <BROWSER> <INCLUDE TAG>.* Esimerkiksi komento *'runallRF.bat qa Firefox kirjautuminen'* ajoi QA-testiympäristössä Firefox-selaimella kaikki 'kirjautuminen'-tagilla olevat testitapaukset. Yksinään kyseisen komennon syöttäminen ei riittänyt testien ajamiseen, vaan komentojono oli rakennettava niin, että sen suorittaminen onnistui edellä mainituilla

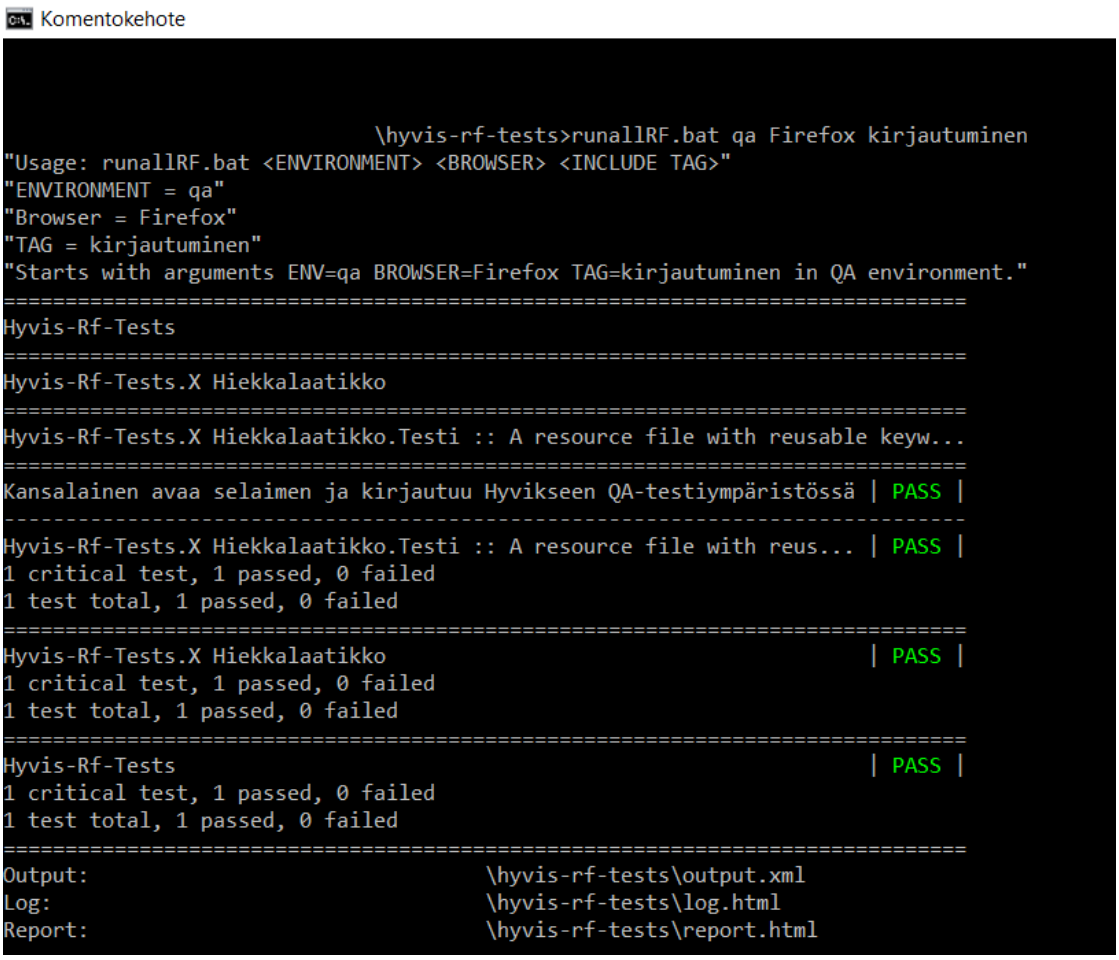
muuttujilla. Kuvassa 27 nähdään, kuinka *runallRF.bat*-komentojono rakentui. Riveillä 3–5 asetettiin komentokehotteen kutsussa komentojonon perään asetettavien muuttujien järjestys, eli ensimmäinen muuttuja tarkoittaa ajettavaa ympäristöä, toinen muuttuja tarkoittaa selainta ja kolmas muuttuja tarkoittaa ajettavien testitapausten tagia/tageja. Kuvassa 27 on esitelty vain QA-testiympäristössä ajamiseen tarvitsemamme komentojonon sisältö. Tästä johtuen rivillä 7 määritelty ’*IF %ENVIRONMENT%==qa goto :execute_in_qa’* käski komentojonoa suorittamaan ’*:execute_in_qa’*, mikäli ympäristöksi oli valittu ’*qa’*. Suoritettava komento *’:execute_in_qa’* piti sisällään tulostuksen, jolla kerrottiin aloitettavasta testiajosta valituilla muuttujilla. Kuvasta poiketen jouduimme luomaan jokaiselle testattavalle ympäristölle oman IF-ehdon ja suoritettavan komennon. Itse testiajon suorittava robotin käynnistyskomento sisälsi tiedon valitusta ympäristöstä, selaimesta, ajettavien testitapausten tagista/tageista sekä myös testiajon lopputulokseen liittyvistä sijaintitiedon määrittelyistä.



```
1  @echo off
2
3  SET ENVIRONMENT=%1
4  SET BROWSER=%2
5  SET TAG=%3
6
7  IF %ENVIRONMENT%==qa goto :execute_in_qa
8
9  :execute_in_qa
10 echo "Starts with arguments ENV=%ENVIRONMENT% BROWSER=%BROWSER% TAG=%TAG% in QA environment."
11 robot --variable ENVIRONMENT:%ENVIRONMENT% --variable BROWSER:%BROWSER% --include %TAG%ANDQA --argumentfile environments/common-arguments.txt .
```

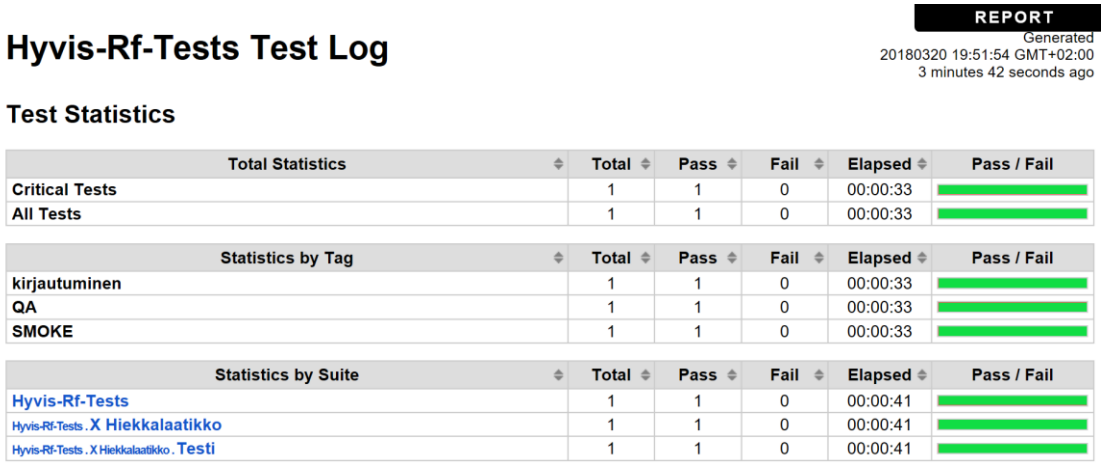
Kuva 27. *runallRF.bat*-komentojonon rakenne

Luvussa 4.5 esittelemäni esimerkkitestisovelluksen testiajon käynnistämisen voisi toteuttaa luomalla kuvan 27 kaltaisen komentojonon ja sitten suorittamalla sen komentokehotteella (kuva 28). Toinen vaihtoehtoinen tapa olisi antaa suoraan komentokehotteelle robotin oma käynnistyskomento: ’*robot –variable ENVIRONMENT:qa –variable BROWSER:Firefox –include kirjautuminen –argumentfile ./environments/common-arguments.txt’*. Komentojonon käyttäminen testiajon käynnistämiseen oli lukemattomista toistoista johtuen järkevää sekä myös erittäin käyttäjäystävällistä.



Kuva 28. Suoritetaan komentojono, joka käynnistää esimerkkitestisovelluksen ajon

Testiajon käynnistyttyä Robot Framework ajoi läpi kaikki käynnistysvaiheen määrittelyiden mukaiset testitapaukset. Yksittäinen testitapaus päätyi aina joko PASS- tai FAIL-tulokseen. Testattava testitapausten joukko päätyi myös FAIL-tulokseen, mikäli yksikin sen sisältämistä testitapauksista päätyi FAIL-tulokseen. Testiajon tulokset tallentuivat *common-arguments.txt*-tiedoston määrittelyjen mukaisesti tiettyyn tiedostopolkuun. Suoritetusta testiajosta muodostui kolme eri tiedosta: *output.xml, log.html* ja *report.html. Output.xml*-tiedosto sisälsi testiajon tulokset luettavassa XML-muodossa. Jälkimmäiset *log.html*- ja *report.html* -tiedostot pohjautuivat edellä mainittuun XML-muotoiseen *Output*-tiedostoon, ja ne olivat tutkittavissa selaimen kautta. Tutkittaessa testiajon tuloksia selaimen kautta html-tiedoston muodossa pystyimme helposti havaitsemaan testiajon eri vaiheet sekä erottelemaan PASS- ja FAIL-tulokset.



Hyvis-Rf-Tests Test Log

REPORT
Generated
20180320 19:51:54 GMT+02:00
3 minutes 42 seconds ago

Test Statistics

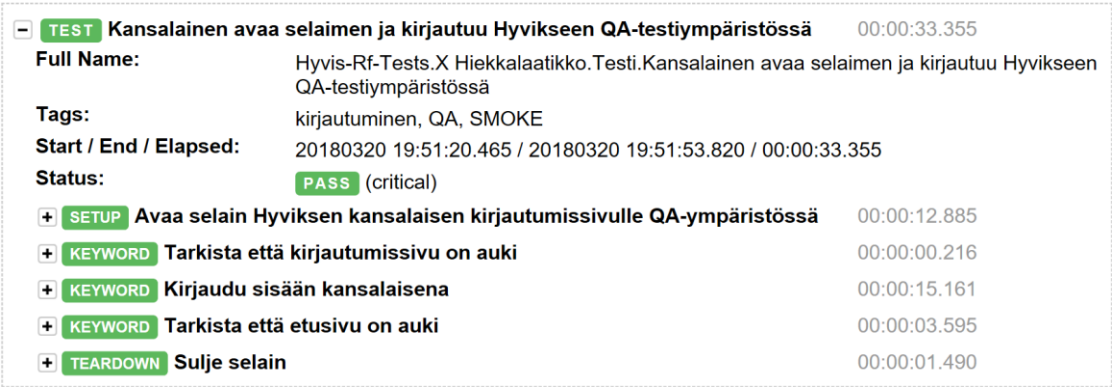
Total Statistics	Total	Pass	Fail	Elapsed	Pass / Fail
Critical Tests	1	1	0	00:00:33	
All Tests	1	1	0	00:00:33	

Statistics by Tag	Total	Pass	Fail	Elapsed	Pass / Fail
kirjautuminen	1	1	0	00:00:33	
QA	1	1	0	00:00:33	
SMOKE	1	1	0	00:00:33	

Statistics by Suite	Total	Pass	Fail	Elapsed	Pass / Fail
Hyvis-Rf-Tests	1	1	0	00:00:41	
Hyvis-Rf-Tests . X Hiekkalaatikko	1	1	0	00:00:41	
Hyvis-Rf-Tests . X Hiekkalaatikko . Testi	1	1	0	00:00:41	

Kuva 29. Esimerkkitestisovellus ajettu onnistuneesti läpi

Kuvassa 29 nähdään onnistuneen testiajon *log.html* tekemällemme kansalaisen Hyvis kirjautumisen esimerkkitestisovellukselle. Pystyimme tätä kautta tutkimaan muun muassa testiajon kestoa ja läpäistyjen testien määrää. Jokaisesta eri ajettavasta testitapauksesta näimme yksityiskohtaiset tiedot jokaisen kyseisen testitapauksen sisältämän avainsanan kohdalta (kuva 30).



TEST Kansalainen avaa selaimen ja kirjautuu Hyvikseen QA-testiympäristössä 00:00:33.355

Full Name: Hyvis-Rf-Tests.X Hiekkalaatikko.Testi.Kansalainen avaa selaimen ja kirjautuu Hyvikseen QA-testiympäristössä

Tags: kirjautuminen, QA, SMOKE

Start / End / Elapsed: 20180320 19:51:20.465 / 20180320 19:51:53.820 / 00:00:33.355

Status: PASS (critical)

SETUP Avaa selain Hyviksen kansalaisen kirjautumissivulle QA-ympäristössä 00:00:12.885

KEYWORD Tarkista että kirjautumissivu on auki 00:00:00.216

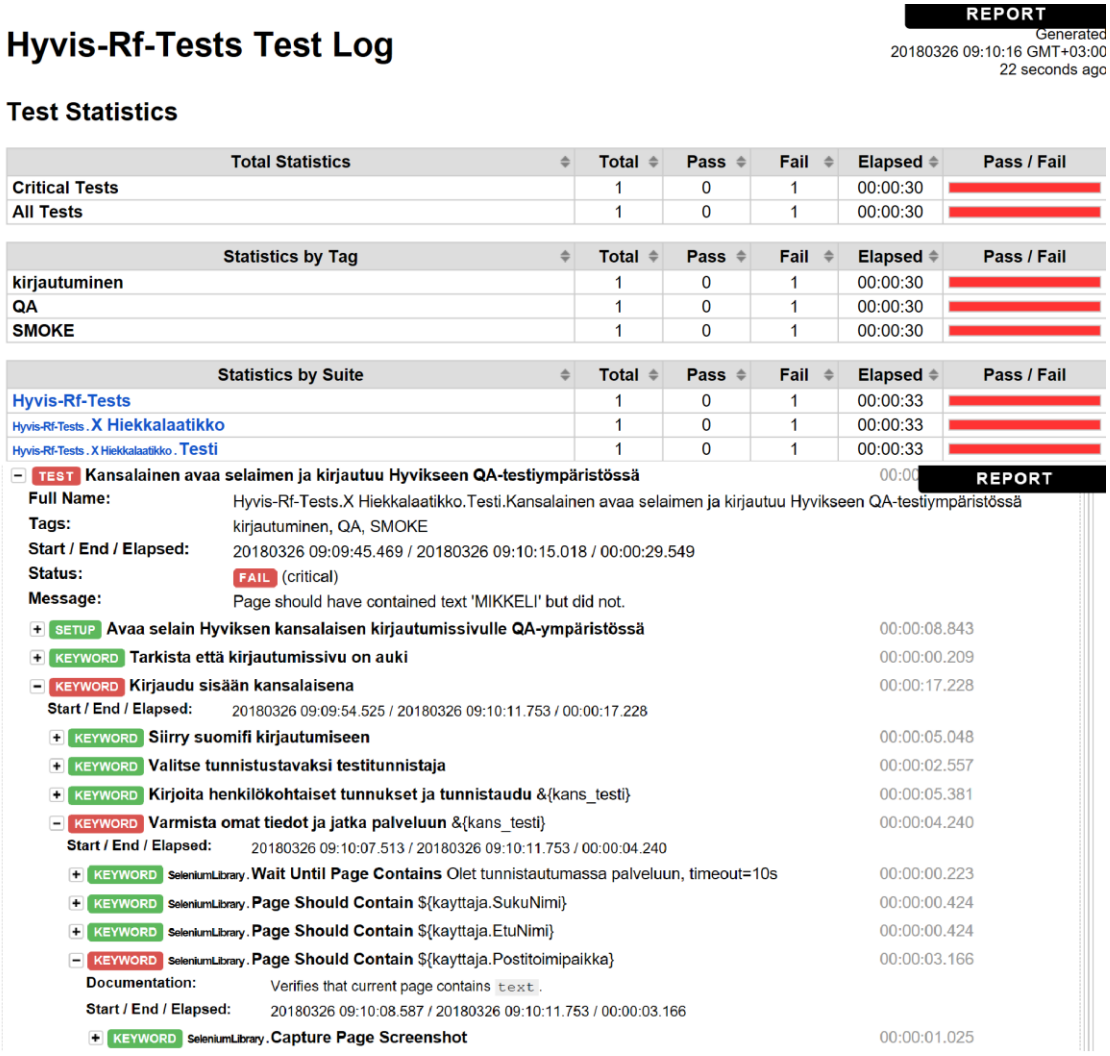
KEYWORD Kirjaudu sisään kansalaisena 00:00:15.161

KEYWORD Tarkista että etusivu on auki 00:00:03.595

TEARDOWN Sulje selain 00:00:01.490

Kuva 30. Onnistuneen testiajon yksityiskohtaiset vaiheet eroteltuna

Testiajo saattoi päätyä myös FAIL-tulokseen. FAIL-tuloksen syy oli yleensä yksinkertaisesti järjestelmän tai verkkoyhteyden hitaus, minkä takia jotkin elementit eivät latautuneet tarpeeksi nopeasti. Hitauden lisäksi syinä FAIL-tulokseen päätyneisiin testiajoihin olivat muun muassa testien sisältämät mahdolliset kirjoitusvirheet sekä virheelliset elementtiviittaukset. Myös tiettyjä selainkohtaisia virheitä esiintyi esimerkiksi sivun skrollaamiseen liittyen.



Kuva 31. Esimerkkitestisovelluksen testiajo päättynyt FAIL-tulokseen

Siinä kohtaa, jos testitapaus päätyi FAIL-tulokseen, otti Robot Framework automaattisesti kuvankaappauksen SeleniumLibraryn sisäänrakennetun avainsanan ’*Capture Page Screenshot’* avulla sivulta kohdassa, jossa virhe tapahtui. Yksityiskohtaiset testituloslokit ja kuvankaappaukset auttoivat mahdollisen ongelman selvittämisessä. Kuvan 31 tapauksessa Robot Framework ei löytänyt tunnistautumisvaiheen tietojen varmistussivulta kansalaisen postitoimipaikkaa ”Mikkeli”, joka oli testimielessä vaihdettu *QA_kayttajalista.robot* tietoihin kansalaisen oikean postitoimipaikan ollessa ”Lappeenranta”. Vaihdoksen tuloksena testi päätyi FAIL-tulokseen.

4.7 Ongelmatilanteet

Automatisoinnin edetessä törmäsimme muutamiin haastavampiin ongelmiin. Tällaisia olivat muun muassa liitetiedostojen lisääminen viesteihin, ilman tunnistetietoja oleviin elementteihin viittaaminen sekä testiajon tulosten erottelu

esimerkiksi aikaleiman mukaan. Osa ratkesi pelkästään tietyillä koodimuutoksilla, mutta osaan jouduimme harkitsemaan ylimääräisten lisäosien asennusta.

Ensimmäinen haastavampi ongelma, johon törmäsimme, oli liitetiedostojen lisääminen lähetettäviin viesteihin. Liitetiedostojen lisääminen Hyviksessä tapahtui osittain käyttöjärjestelmän toiminnallisuuksien kautta. Liitetiedoston valitseminen tietystä hakemistosta ei onnistunut Robot Frameworkia hyödyntäen, koska se on webpohjaisten sovellusten testaamiseen tarkoitettu työkalu. Pilotointivaiheessa päädyimme harkitsemaan AutoIt-työkalun asentamista, koska sillä on mahdollista automatisoida Windowsin toiminnallisuuksia, eli muun muassa juuri tiedoston valitseminen. Pyrimme mahdollisimman pitkälle välttämään ylimääräisten lisäosien asentamista, joten päätimme pilotointivaiheessa keskittyä ensin muiden Hyviksen toiminnallisuuksien automatisoidun testauksen rakentamiseen ja myöhemmin mahdollisesti palata harkitsemaan AutoIt:n käyttöönottoa.

Toinen ongelma, jonka kohtasimme, liittyi elementtiviittauksiin. Samalla rivillä ja ilman yksilöllisiä tunnistetietoja oleviin elementteihin viittaaminen osoittautui hankalaksi. Ainut tapa viitata riviin ja siinä haluttuun elementtiin saattoi olla tietty tekstinpätkä kyseisen rivin sisältämässä toisessa elementissä. Kuvassa 32 on esitetty esimerkkikoodi tapauksesta, jossa on kaksi eri riviä, joissa molemmissa on ’Valitse’-näppäin. Klikkauksen kohdistaminen halutun rivin ’Valitse’-näppäimeen onnistui hyödyntämällä kyseisen rivin sisältämän toisen elementin tekstiä. Ratkaisu ongelmaan löytyi xpathin *following-sibling-* ja *preceding-sibling* -ominaisuuksista. Kuvan 32 tapauksessa pystyimme kohdistamaan klikkauksen ylemmän rivin ’Valitse’-näppäimeen avainsanalla *’Click Element xpath=.//div[@class=’row’]/div[contains(., ’Tekstiä1’)]/preceding-sibling::button[@class=’button’]’.* Mikäli klikattava elementti sijaitsi viitattavan tekstiä sisältävän elementin jälkeen, onnistui elementtiviittaus taas *following-sibling* -ominaisuudella.

```
<div class="row">
  <button class="button">Valitse</button>
  <div class="row-text">Tekstiä1</div>
</div>
<div class="row">
  <div class="row-text">Tekstiä2</div>
  <button class="button">Valitse</button>
</div>
```

Kuva 32. Esimerkkikoodi elementeistä ilman yksilöllisiä tunnistetietoja

Kolmas ongelma, testiajon tulosten erottelu, oli edessä, kun halusimme testiajojen tuloksiin mukaan aikaleiman sekä ympäristön, jossa testit oli ajettu. Ratkaisu tähän ongelmaan löytyi luomastamme komentojonosta. Lisäämällä *runallRF.bat*-tiedostoon kuvan 33 mukaiset määrittelyt, saimme testiajojen tulosten kansiorakenteen halutunlaiseksi. Itse testiajojen tulosten kansiorakenteen määräävä *'—outputdir'* lisättiin ympäristökohtaisiin *':execute'*-komentoihin. Tällöin testiajojen tulosten kansiorakenteeksi muodostui *'./robot-results/ympäristö/päivämäärä/kellonaika/'*.

```
SET HOUR=%time:~0,2%
SET kellonaika9=0%time:~1,1%%time:~3,2%%time:~6,2%
SET kellonaika24=%time:~0,2%%time:~3,2%%time:~6,2%
SET paivamaara=%date:~4,4%%date:~7,2%%date:~10,2%
if "%HOUR~0,1%" == " " (SET kellonaika=%kellonaika9%) else (SET kellonaika=%kellonaika24%)

--outputdir robot-results/%ENVIRONMENT%/%paivamaara%/%kellonaika%
```

Kuva 33. Komentojonoon lisätyt määrittelyt testiajojen tulosten erottelemiseksi

Suurin osa automatisoinnin aikana kohtaamistamme ongelmista liittyi lähinnä hankaluuksiin uuden ohjelmointikielen kanssa. Selvisimme kuitenkin koodiin liittyvistä hankaluuksista tarpeeksi asiaa selvitettyämme eri lähteistä. Uusien ylimääräisten työkalujen hyödyntäminen automatisoidun testauksen kattavuuden kasvattamiseksi taas on kirjoitushetkellä edelleen harkinnan alla.

4.8 Testauksen automatisoinnin onnistumisen arvioiminen

Testauksen automatisoinnin pilotointivaiheen aikana saimme tehdyksi Robot Frameworkilla rungon, johon oli helppoa toteuttaa uusia kokonaisia testisovelluksia. Lisäksi saimme tehtyä muutamia valmiita toimivia testitapauksia, muun muassa ammattilaisen ja kansalaisen kirjautumiset. Pilotointivaiheessa mukana ollut Eficode Oy:n asiantuntija sai myös tehtyä pohjustuksia usealle eri Hyviksen testattavalle toiminnallisuudelle kehitysympäristössä (DEV). Näin

meidän oli 2M-IT:n puolesta helppo tarttua kyseisiin pohjustuksiin ja kääntää ne toimiviksi kokonaisiksi testisovelluksiksi QA-testiympäristöön, jossa testaajat pääosin testaavat.

Helmikuun alkupuolella pidetyssä pilotointivaiheen päätöspalaverissa totesimme pilotointivaiheen olleen onnistunut. Olimme yhtä mieltä 2M-IT:n puolella, että testauksen automatisointia kannattaa viedä pidemmälle, joten jatkoimme yhdessä ohjaajani Markus Peltosen kanssa automatisoinnin parissa työstämistä. Tavoitteisiin jatkon suhteen liittyivät olennaisesti automatisoitujen testien kattavuuden kasvattaminen uusina testisovelluksina, testien laajentaminen toimimaan myös muissa Hyviksen ympäristöissä sekä testiajojen suorittaminen virtuaalikoneen välityksellä.

Arvioidessa testauksen automatisoinnin onnistumista omien kokemusteni näkökulmasta, voin todeta, että pystyin omaksumaan Robot Frameworkin ja muiden tarvittavien ohjelmistojen perusteet pilotointivaiheen edetessä. Alussa osaamiseni esimerkiksi Pythonista ja automatisoinnin työkaluista oli käytännössä olematon. Robot Framework ja sen eri käyttömahdollisuudet alkoivat valottua minulle heti pilotointivaiheen alussa, joten itse automatisoinnin työstämisen aloittaminen oli suhteellisen ongelmatonta. Itselläni tätä edesauttoi varsinkin selkokielisen SeleniumLibraryn käyttäminen kirjastona. Koen, että Robot Frameworkin helppo käyttöönotto oli yksi pääsyistä testauksen automatisoinnin pilotointivaiheen onnistumiselle.

5 PÄÄTÄNTÖ

Opinnäytetyöni tavoitteena oli selventää, kuinka 2M-IT:n tarjoaman sähköisen asiointipalvelun Hyviksen testaus pystyttiin automatisoimaan pilotointivaiheessa. Automaattitestauksen rakentamisen lisäksi kerroin yleisesti automaattitestauksesta sekä testauksesta osana ohjelmistotuotantoa. Testauksen taustojen selvittäminen oli suhteellisen helppoa, sillä löysin paljon aiheeseen liittyvää kirjallisuutta. Muun muassa lähteenä käytetty Kasurisen ”Ohjelmistotestauksen käsikirja” sisälsi huomattavan määrän tietoa testauksesta osana ohjelmistotuotantoa. Automaattitestauksen taustojen selvittämisessä apuna olivat eri verkkolähteet sekä etenkin Pohjolaisen pro gradu -tutkielma ”Ohjelmiston testauksen automatisointi”. Testauksen automatisoinnin rakentamisesta

kerroin pääosin oman työkokemuksen sekä automatisoinnissa käytettävien kirjastojen perusteella.

Opinnäytetyön aiheeseen, eli testauksen automatisointiin Robot Frameworkilla, oli mielestäni helppo päästä sisään, sillä olin ollut 2M-IT:llä töiden puolesta mukana testauksen automatisointiprojektissa. Opinnäytetyön aloittamisen aikaan, samaan aikaan kuin testauksen automatisoinnin pilotointivaihe alkoi, minulla ei ollut juurikaan tietoa automatisoinnista ylipäänsä eikä myöskään automatisoinnin työkaluista. Koen pystyneeni oppimaan tarvittavat perustaidot testauksen automatisoinnin toteuttamiseksi neljä viikkoa kestäneen pilotointivaiheen aikana.

Aiheena testauksen automatisointi oli hyvin mielenkiintoinen ja uskon jatkossakin olevani aiheen kanssa tekemisissä etenkin töiden puolesta. Opinnäytetyötä kirjoittaessani ja aiheen taustoja selvittäessäni vasta tajusin, kuinka tärkeä osa ohjelmistotuotantoa testaus oikein on. Lyhyenä yhteenvetona voisin todeta, että ohjelmiston kattava testaus vähentää virheistä aiheutuneita kustannuksia, ja automaattitestaus lisää testauksen kattavuutta.

Automatisointia olisi ollut mielestäni mielenkiintoista viedä vielä astetta pidemmälle automatisoimalla myös testiajojen suorittaminen esimerkiksi Jenkinsillä. Pilotointivaiheessa tähän ei kuitenkaan vielä perehdytty. Minua olisi myös kiinnostanut tutkia, että mihin kaikkeen Robot Framework oikein taipuu automatisoinnissa. Lopuksi olisin halunnut selvittää, että olisiko testauksen automatisointi ollut helpompi toteuttaa jollakin toisella automatisoinnin työkalulla Robot Frameworkin sijaan. Pilotointivaiheeseen ja opinnäytetyöhöni varatut resurssit eivät kuitenkaan riittäneet selvittämään näitä kohtia, vaan mahdollinen jatkotutkimus oli jätettävä tulevaisuuteen.

LÄHTEET

2M-IT Oy s.a. Hyvinvoiva ihminen. WWW-dokumentti. Saatavissa: https://2m-it.fi/ [viitattu 5.4.2018].

Ahonen, M. 2010. Tapaustutkimus: Soveltuuko Scrum vesiputousmallin korvaajaksi yrityksen sovelluskehitysprojekteihin? PDF-dokumentti. Päivitetty 3.5.2010. Saatavissa: http://docplayer.fi/16523969-Tapaustutkimus-soveltuuko-scrum-vesiputousmallin-korvaajaksi-yrityksen-sovelluskehitysprojekteihin.html [viitattu 20.2.2018].

Atom s.a. A hackable text editor for the 21st Century. WWW-dokumentti. Saatavissa: https://atom.io/ [viitattu 8.3.2018].

Avania. 2009. Käytettävyystestaus pähkinänkuoressa. WWW-dokumentti. Päivitetty 19.4.2009. Saatavissa: https://www.avania.fi/kaytettavyystestaus-pahkinankuoressa/ [viitattu 19.2.2018].

GitHub s.a. Robot Framework – SeleniumLibrary. WWW-dokumentti. Saatavissa: https://github.com/robotframework/SeleniumLibrary [viitattu 7.3.2018].

Google s.a. ChromeDriver – WebDriver for Chrome. WWW-dokumentti. Saatavissa: https://sites.google.com/a/chromium.org/chromedriver/home [viitattu 6.3.2018].

Haikala, I. & Märijärvi, J. 2004. Ohjelmistotuotanto. 10. painos. Helsinki: Talentum.

ISTQB Exam Certification s.a. What is Software Testing?. WWW-dokumentti. Saatavissa: http://istqbexamcertification.com/what-is-software-testing/ [viitattu 7.4.2018].

ISTQB Exam Certification s.a. What is Waterfall model -advantages, disadvantages and when to use it?. WWW-dokumentti. Saatavissa: http://istqbexamcertification.com/what-is-waterfall-model-advantages-disadvantages-and-when-to-use-it/ [viitattu 19.2.2018].

Java s.a. How do I set or change the PATH system variable?. WWW-dokumentti. Saatavissa: https://www.java.com/en/download/help/path.xml [viitattu 7.3.2018].

Kasurinen, J. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo.

Kasurinen, J. 2015. Ohjelmistotestauksen perusteet. PDF-dokumentti. Saatavissa: http://docplayer.fi/7192802-Ct60a4150-ohjelmistotestauksen-perusteet-jussi-kasurinen-etu-suku-lut-fi-kevat-2015.html [viitattu 20.2.2018].

Kipp, L. 2016. Ask a Security Professional: Black Box vs. White Box Series. Part Two: White Box Testing. Blogi. Päivitetty 13.6.2016. Saatavissa: https://wpdistrict.sitelock.com/blog/black-box-vs-white-box-part2-sast/ [viitattu 20.2.2018].

Kunda, E. 2016. The Insider's View of QC for Salesforce Custom Apps. WWW-dokumentti. Päivitetty 29.6.2016. Saatavissa: https://codeswat.com/quality-control-for-salesforce-custom-apps [viitattu 22.2.2018].

Laitinen, M. 2011. Jatkuvan integraation käyttöönotto. PDF-dokumentti. Päivitetty 16.4.2011. Saatavissa: https://www.theseus.fi/bitstream/handle/10024/32462/Laitinen_Miku.pdf?sequence=1 [viitattu 5.3.2018].

Pohjolainen, P. 2003. Ohjelmisto testauksen automatisointi. PDF-dokumentti. Päivitetty 12.12.2003. Saatavissa: http://cs.uef.fi/uku/tutkimus/Teho/PenttiPohjolainen_Gradu.pdf [viitattu 14.2.2018].

ProfessionalQA. 2018. Sandwich Testing. WWW-dokumentti. Päivitetty 17.2.2018. Saatavissa: http://www.professionalqa.com/sandwich-testing [viitattu 7.4.2018].

Purojärvi, J. 2010. Lääkinnällisen laitteen ohjelmistokehitys täydennetyllä Scrum-mallilla. PDF-dokumentti. Päivitetty 18.10.2010. Saatavissa: http://docplayer.fi/7907638-Laakinnallisen-laitteen-ohjelmistokehitys-taydennetylla-scrum-mallilla.html [viitattu 26.4.2018]

Robot Framework s.a. Introduction. WWW-dokumentti. Saatavissa: http://robotframework.org/ [viitattu 26.2.2018].

Sininen Meteoriitti. 2013. Ketteryys haltuun: Scrum pähkinänkuoressa. WWW-dokumentti. Päivitetty 6.6.2013. Saatavissa: https://www.meteoriitti.com/2013/06/06/ketteryys-haltuun-scrum-pahkinankuoressa/ [viitattu 19.2.2018].

Smartbear s.a. Why automated testing? WWW-dokumentti. Saatavissa: https://support.smartbear.com/articles/testcomplete/manager-overview/ [viitattu 21.2.2018].

Soberit Aalto-yliopisto s.a. V-malli. WWW-dokumentti. Saatavissa: http://www.soberit.hut.fi/T-76.115/03-04/palautukset/groups/PPT/i1/images/testplan_vmalli.gif [viitattu 20.2.2018].

Software Testing Class. 2015. What is Alpha and Beta Testing? WWW-dokumentti. Päivitetty 9.6.2015. Saatavissa: http://www.softwaretestingclass.com/what-is-alpha-and-beta-testing/ [viitattu 7.4.2018].

Software Testing Fundamentals s.a. Smoke Testing. WWW-dokumentti. Saatavissa: http://softwaretestingfundamentals.com/smoke-testing/ [viitattu 10.4.2018].

Tassey, G. 2002. The Economic Impacts of Inadequate Infrastructure for Software Testing. PDF-dokumentti. Saatavissa: https://www.nist.gov/sites/default/files/documents/director/planning/report02-3.pdf [viitattu 11.2.2018].

Testlio. 2017. Black Box Testing. WWW-dokumentti. Päivitetty 6.12.2017. Saatavissa: https://testlio.com/wp-content/uploads/2017/12/Screenshot-2017-12-06-16.51.30-1024x529.png [viitattu 20.2.2018].

W3Schools s.a. Python Overview. WWW-dokumentti. Saatavissa: https://www.w3schools.in/python-tutorial/overview/ [viitattu 7.3.2018].

Liite 1

ASENNUSLINKIT

Python 2.7.10

- https://www.python.org/downloads/release/python-2710/

Git

- https://git-scm.com/downloads

ChromeDriver

- https://sites.google.com/a/chromium.org/chromedriver/downloads

Firefox Gecko

- https://github.com/mozilla/geckodriver/releases

XPath Helper

- https://chrome.google.com/webstore/detail/xpath-helper/hgim-nogjllphhhkhlmebbmlgjoejdpjl

xPath Finder

- https://addons.mozilla.org/en-US/firefox/addon/xpath_finder/

Atom

- https://atom.io/