Haaga-Helia
ammattikorkeakoulu Oy

Ohjelmistotestaus ja Robot Framework -testausautomaatiotyökalun käyttöönotto

Tiina Anttila

Opinnäytetyö
Tietojenkäsittelyn koulutusohjelma
2020

Haaga-Helia ammattikorkeakoulu Oy

23.1.2017

Tekijä(t)

Tiina Anttila

Koulutusohjelma

Tietojenkäsittelyn koulutusohjelma

Opinnäytetyön otsikko

Ohjelmistotestaus ja Robot Framework -testausautomaatiotyökalun käyttöönotto

Sivu- ja liitesivumäärä

50 + 51

Opinnäytetyön otsikko englanniksi

Software testing and Robot Framework test automation tool

Tässä toiminnallisessa opinnäytetyössä tehtiin ohjelmistotestaukseen testiautomatisoinnin käyttöönotto. Automatisointityökaluna oli Robot Framework, joka on avoimen lähdekoodin avainsanaohjattu automaatioviitekehys. Työssä otettiin käyttöön Robot Framework ja tehtiin selain- ja rajapintatestit ohjelmistokehityksessä käytettäviksi. Työn tuloksena on liitteinä asennusohje ja testien tekemisen, ajamisen sekä päivittämisen ohje.

Opinnäytetyön teoriaosuudessa on esitelty ohjelmistokehitys, ohjelmistotestaus ja testausautomatisaatio. Ohjelmistokehityksessä laajenee ketterän kehityksen käyttö ja siten laajenee myös testausautomatisaation käyttö. Testausautomatisaatiosta on pyritty kuvaamaan miksi ja milloin sitä kannattaa käyttää sekä milloin sitä ei kannata tehdä. Lisäksi on esitelty testausautomaatiokehyksiä.

Työn toiminnallisessa osuudessa on esitelty työssä käytetty Robot Framework -automatisointityökalu. Työssä on kuvattu tarvittavat asennukset, selain- ja rajapintatestien tekeminen, niiden ajaminen ja Robot Frameworkin raportit.

Asiasanat

Ohjelmistotestaus, testausautomaatio, Robot Framework

Sisällys

1 Johdanto

Ohjelmistotestaus on prosessi, tai sarja prosesseja, joka on suunniteltu varmistamaan, että koodi tekee mitä se on suunniteltu tekemään sekä käänteisesti varmistamaan, ettei koodi tee mitään tarkoittamatonta. Ohjelmiston pitäisi toimia ennustettavasti ja johdonmukaisesti eikä tuottaa yllätyksiä käyttäjille. (Myers, Badgett & Sandler 2012, 2.)

Testiautomatisointi on muuttumassa välttämättömyydeksi ohjelmistokehityksessä, sillä kun sovellukset ja järjestelmät muuttuvat monimutkaisemmiksi ja laajemmiksi ei pelkkä manuaalitestaus enää riitä. Testiautomatisoinnin tärkeyttä nostaa myös ketterän kehityksen laajeneminen, sillä vaikka testiautomatisoinnista on hyötyä perinteisessäkin kehityksessä, on se välttämätöntä ketterässä kehityksessä. (Fewster & Graham 2012, luku 1.)

Työn tavoitteena on oppia tekemään testiautomaatio, jolla voidaan testata ohjelmistoa selain- ja rajapintatestein. Automatisointityökaluna on Robot Framework, joka on avoimen lähdekoodin avainsanaohjattu automaatioviitekehys. Robot Frameworkillä tehdään selaintesteihin ja rajapintatesteihin omat testit. Testien pitää olla nopeasti ajettavissa, helposti päivitettävissä eri aineistolla käytettäväksi sekä testauksen pitää olla nopeampaa kuin saman alueen käsittävän manuaalitestauksen.

Työssä esitellään ohjelmistokehityksen periteinen ja ketterä kehitysmalli, käydään läpi testaus näissä molemmissa ja tarkastellaan testausautomaatiota erikseen. Tavoitteena on ymmärtää, miten testausautomaatio tarvitaan ja voidaan hyödyntää ohjelmistokehityksessä. Työn tuotoksina tulee Robot Framework -automatisointityökalun ja tarvittavien ohjelmien asennusohje sekä yleisohje testien tekemiseen, ajamiseen ja ylläpitoon. Työllä ei ole toimeksiantajaa ja testeissä käytetään opinnäytetyötä varten American Airlines -verkkosivua sekä Tilastokeskuksen StatFin-tilastotietokantaa avoimen rajapinnan kautta.

Opinnäytetyön luvussa 2 käydään läpi ohjelmistokehitystä ja esitellään perinteistä ja ketterää kehitystä. Luvussa 3 esitellään ensin testaus yleisesti ja kerrotaan, miksi se on tärkeää. Tämän jälkeen esitellään testaus perinteisessä kehityksessä ja lopuksi ketterässä kehityksessä. Luvussa 4 keskitytään testausautomaatioon: mitä se on ja miksi sitä kannattaa tehdä sekä esitellään automatisointikehyksiä. Luvussa 5 tarkastellaan Robot Framework-automatisointikehystä ja käydään läpi sen sekä liittyvien ohjelmien asentaminen, testien tekeminen ja ajaminen sekä raportointi. Työn tuloksina on Robot Frameworkin asennusohje sekä ohje testien tekoon, ylläpitoon ja ajamiseen. Nämä sisällytetään työhön liitteinä. Lopuksi luku 6 sisältää pohdinnan.

Lyhenteet ja käsitteet

Ajuri	Ohjelmistokomponentti tai työkalu, jolla hallitaan tai kutsutaan testattavaa järjestelmää tai komponenttia (FiSTB 2015). Selainajurilla hallitaan selainta.
Editori	Ohjelma, jolla muokataan tiedostoja. Ohjelmistokehityksessä voidaan käyttää tekstieditoreita koodin kirjoittamiseen.
IDE	Integrated development environment (IDE) on integroitu kehitysympäristö, jota esimerkiksi kehittäjät käyttävät ohjelmistoa kehittäessä (Kasurinen 2013, 85)
Kirjasto	Kokoelma luokkia, ohjelmia ja/tai aliohjelmia eli rutiineja, joita voidaan käyttää ohjelmien kehityksessä ja suorituksessa. Rutiinit tai moduulit ovat käteviä usein käytettyjen rutiinien tallentamisessa. (Webopedia 2020)
Komentokieli	Korkean tason ohjelmointikieli, skripti. Tätä tulkataan lennossa etukäteen kääntämättä toisin kuin esimerkiksi Javaa, mikä on käännettävä ohjelmointikieli. Käytetään yleensä ohjelmoimaan toimintoja silloin kun toiminnallisuus on jo valmiina. Komentokieliä ovat esimerkiksi Python ja JavaScript. (Downey 2016, luku 1)
Ohjelmointirajapinta	Application Programming Interface (API) eli ohjelmointirajapinta määrittelee, miten eri ohjelmistot voivat tarjota palveluita tai tietoja muille tietojärjestelmille tai sovelluksille. Rajapinta voi olla datarajapinta tai toiminnallinen rajapinta. (JUHTA 2018)
PATH	Hakupolku eli käyttöjärjestelmien hyödyntämä ympäristömuuttuja, jossa määritetään suoritettavien ohjelmien hakemistoja (Open Group 2018).

Testaus	Prosessi, tai sarja prosesseja, joilla arvioidaan vastaako ohjelmiston toiminta vaatimuksia, osoitetaan ohjelmiston soveltuvan suunniteltuun käyttöönsä sekä löydetään virheitä. (FiSTB 2015.)
Testausautomatisaatio	Ohjelmistojen käyttö testaamiseen tai testauksen tukitoimiin. Testausautomatisaatioita voidaan käyttää testauksen suunnitteluun, hallintaan, testitapausten suorittamiseen ja/tai tulosten analysointiin. (FiSTB 2015.)
Testauskehys	Kokoelma hyviä käytäntöjä, työkaluja ja kirjastoja, jotka helpottavat testaamista. (Sofware Testing Help 2020.)
Virheenjäljitys	Debugging eli virheenjäljitys. Kehittäjän tekemää työtä, jossa etsitään virheen löydyttyä korjausta varten syytä virheen esiintymiselle. (Haikala & Mikkonen 2011, 205)

2 Ohjelmistokehityksen vaiheet ja menetelmät

Tässä käydään läpi ohjelmistotuotantoa ja ohjelmistokehityksen vaiheita, sillä nämä vaikuttavat suoraan testaukseen. Kappaleessa esitellään ensin ohjelmistotuotanto lyhyesti, jonka jälkeen käydään läpi perinteisen ja ketterän kehitysmallien perusteet.

2.1 Ohjelmistotuotannon esittely

Ohjelmistotuotanto voidaan määrittää kattamaan tietokoneohjelmistojen yleisiä rakennustekniikoita ja -työkaluja sekä rakentamisen periaatteita ja menettelytapoja. Käsite ohjelmisto kattaa tietokoneohjelman yhdessä sen dokumentaation kanssa ja järjestelmä kattaa ohjelmiston ja laitteiston yhdessä muodostaman kokonaisuuden. (Haikala & Mikkonen 2011, 11.)

Ohjelmistotuotannon tarkoitus on laajasti tulkiten asiakkaan odotukset kohtuullisessa määrin täyttävien tietokoneohjelmien tuottaminen niin, että aikataulu ja kustannukset ovat riittävissä määrin ennustettavissa. Ohjelmistotuotanto kattaa lähes kaikki vaiheet ohjelmistotyöstä eli suunnittelun, määrittelyn, toteutuksen, laadunvarmistuksen ja testauksen. Lisäksi tuotantoprosessin ohjelmistotyöhön liittyvät osa-alueet kuuluvat ohjelmistotuotantoon, esim. laatujärjestelmä, projektin hallinta sekä dokumentointi ja tuotteenhallinta. (Haikala & Mikkonen 2011, 12.)

Ohjelmistokehityksen epäonnistuneita esimerkkejä on paljon, esimerkiksi Iso-Britannian verojärjestelmässä ollut virhe, jonka vuoksi käyttäjä näki edellisen käyttäjän ansiotulot. Vastaavasti pieni muutos sähköntuottajan laskutusjärjestelmässä pimensi suuren kaupungin kokonaan Yhdysvalloissa. Ohjelmistojen epäonnistumiset johtavat usein rahan, ajan ja/tai maineen menetykseen sekä voivat aiheuttaa vammoja ja kuolemia. Esimerkeistä on huomattavissa joko testauksen puuttuminen tai vääränlainen testaus. (Morgan, Samaroo, Thompson & Williams 2015, luku 1.)

Ohjelmistotestaus määritetään perinteisesti ohjelman suorittamisella tehtäväksi suunnitelmalliseksi virheiden etsinnäksi. Testaus jaetaan seuraaviin työvaiheisiin: suunnittelu, testiympäristön luominen, testien suorittaminen sekä tuloksien tarkastelu. Suunnitteluun kuuluu testisuunnitelman ja testitapausten tekeminen. Testauksella saadaan esiin ohjelman virheitä, mutta ohjelman virheettömyyttä sillä ei pysty osoittamaan. Testaukseen kannattaa panostaa, mutta hyvistä testituloksista huolimatta kannattaa olla skeptinen ohjelman toimivuuden suhteen. Virhe määritetään poikkeamaksi spesifikaatiosta, joita yleisimmin

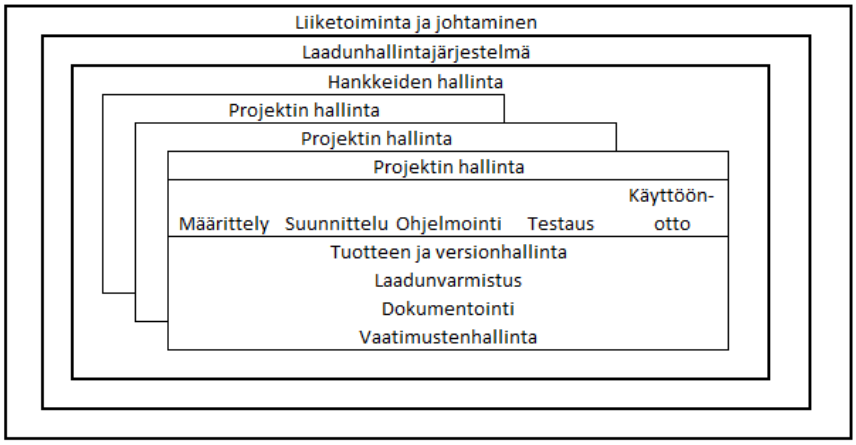
ovat tekninen ja toiminnallinen määrittely. Ilman näitä johdonmukainen testaus on perinteisesti olla siten mahdotonta. Määrittelyiden ja siten virheiden tulkinnoissa voi kuitenkin esiintyä eriäviä mielipiteitä: asiakkaan virhe voi olla toimittajan ominaisuus. (Haikala & Mikkonen 2011, 205-206.)

Virheiden löytämisen ja korjaamisen hintaa on vaikea arvioida ja tämä vaihtelee eri projekteissa, mutta yleisesti pätee se, että kustannukset nousevat sitä korkeammaksi mitä pidemmälle kehityksessä on edetty. Kustannuksien havainnollistamiseen voidaan käyttää eskalointimallia (kuva 1), jossa projektin etenemisen myötä virheen aiheuttamien suhteellisten kustannusten hinta nousee. (Morgan ym. 2015, luku 1.)

Virheen löytämisen vaihe	Suhteellinen kustannus
vaatimusmäärittely	$1
koodaaminen	$10
integraatiotestaus	$100
järjestelmätestaus	$1000
hyväksyntätestaus	$10000
tuotanto	$100000

Kuva 1. Virheen aiheuttaman kustannuksen eskaloituminen (mukaillen Morgan ym. 2015, luku 1)

Ohjelmistojen kehitysmalleja on liki rajaton määrä eri lähestymistapoja, joista yksinkertaisimmillaan vain ohjelmoidaan ja kasvatetaan siitä asiakkaan mieleistä kokonaisuutta. Näissä tapauksissa asiakas on yleensä itse koodin kirjoittaja. Monimutkaisempia ohjelmia kehitettäessä mukaan tulee paljon muutakin kuin ohjelmointia. Yrityksissä toiminnan lähtökohta kaikelle on liiketoiminta. Yrityksen laadunhallintajärjestelmässä kuvataan toimintaprosessit ylätason liiketoimintatasolta tässä huomioitaviin ohjelmistojen hankintaan ja toteutukseen asti. Projektit ovat usein osa laajempaa kokonaisuutta. Ohjelmistoprojektiin liittyy yleensä vähintään määrittely, suunnittelu, ohjelmointi, testaus sekä käyttöönoton ja ylläpidon asioita. Lisäksi projektiin usein kuuluu kautta projektin jatkuvia tukitoimia, kuten projektinhallinta, versionhallinta, vaatimustenhallinta ja dokumentointi. Tätä yrityksen toiminnan kokonaisuutta mallinnetaan kuvassa 2. (Haikala & Mikkonen 2011, 29-30.)

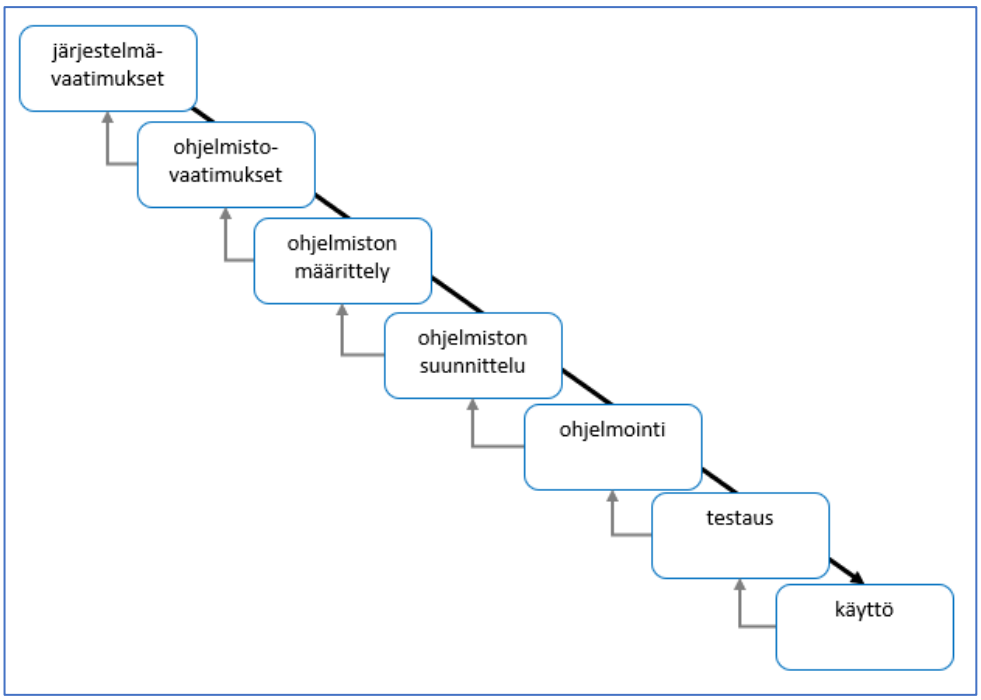


Kuva 2. Projektikokonaisuus (mukaillen Haikala & Mikkonen, 2011, 29-30.)

Kehitysmallit jaetaan yksinkertaistettuna vaiheittaisiin eli vaiheesta toiseen siirtyviin, joita tässä kutsutaan perinteiseksi kehitykseksi, sekä iteratiivisesti toimiviin, joita nimitetään tässä yleisesti ketteräksi kehitykseksi. (Crispin & Gregory 2008, luku 1.)

2.2 Perinteinen kehitysmalli

Ohjelmistokehityksen alusta on huomattu tarve työprosesseille ja ne ovat yleensä vaiheistettu seuraavasti: asiakastarpeiden määrittely, ohjelman suunnittelu, testaus ja käyttöönotto. Tästä keskeinen malli on 1970 Winston Roycen julkaisema vesiputousmalli, jossa yhtenä tärkeänä ominaisuutena on myös iterointi taaksepäin. Tässä kehitysmallissa siirrytään vaiheesta toiseen kuvan 3 mukaisesti. (Haikala & Mikkonen 2011, 36-37.)



Kuva 3. Vesiputousmalli (mallintaen Haikala & Mikkonen 2011, 37)

Vesiputousmalli on hyvä teoreettinen kehitysmalli, sillä siinä kuvataan ohjelmistokehityksen vaiheet ja malli pakottaa käymään läpi mitä tarvitaan ennen seuraavaan vaiheeseen siirtymistä. Taaksepäin siirtyminen tarvittaessa antaa mahdollisuuden päivittää edellisiä vaiheita uusien asioiden selvitessä. Vesiputousmalli on suhteellisen jäykkä, sillä se perustuu tuotantolinjan tapaiseen vaiheesta toiseen siirtymiseen. (Dooley 2011, luku 2.)

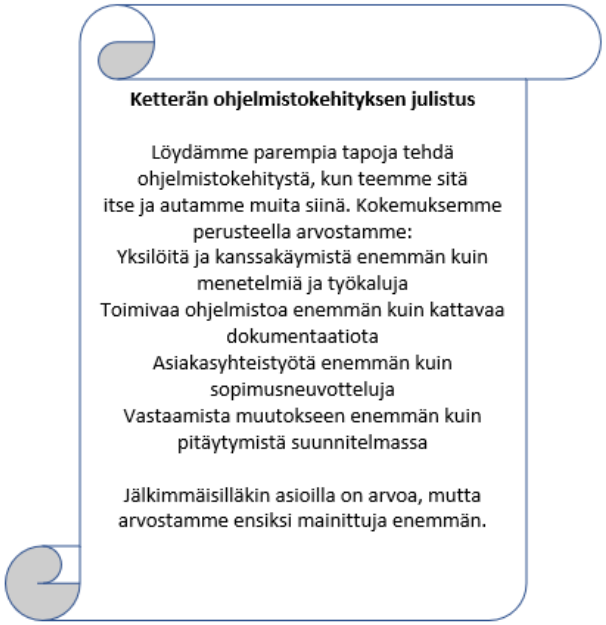
Perinteisen kehityksen ongelmia voidaan havainnollistaa käyttäen esimerkkinä vuoden kehityssykliä. Kehitysprojektin etenemissuunnitelma yleensä sisältää tuotteeseen lisättäviä ominaisuuksia tulevaisuudessa ja lisäominaisuudet aikataulutetaan niin, että tuotteesta tulee esimerkiksi vuoden välein uusi versio. Tässä on kuitenkin omat ongelmansa, sillä esimerkiksi kuluvassa syklissä opittuja asioita kyetään hyödyntämään vasta seuraavassa syklissä, jolloin moni asia on ehtinyt unohtua ja henkilöstöäkin vaihtua. Lisäksi riskit realisoituvat todennäköisesti syklin loppupuolella testausvaiheessa, jolloin voi olla jo myöhäistä tai hyvin kallista reagoida niihin saman syklin sisällä. Eli riskien hallinta on monimutkaista ja riskit ovat ennalta-arvaamattomia. Samaan tapaan myöhässä tulevat asiakaspalaute sekä vaatimusten tarkentuminen. Vaatimukset yleensä muuttuvat ja tarkentuvat pitkän syklin aikana ja reagointi aikataulussa pysyen on pitkässä syklissä mahdotonta. Pahimmillaan järjestelmä on jo valmistuessaan vanhentunut. Testaus tehdään myöhäisessä vaiheessa, jolloin projektin testaukselle varaama aika on usein liian vähäinen ja virheiden korjaus sekä uudelleentestaus vie aikaa. Testaajien on harvoin mahdollista tutustua järjestelmään ennen testauksen aloitusta. Loppuvaiheessa selviää myös, onko määrittely ja suunnittelu ollut riittävää, sillä totuus paljastuu usein toteutuksen ja testauksen aikana. Viimeisenä ongelmana on kärsimättömät asiakkaat: Kärsimättömyys nostaa päätään etenkin projektin myöhästyessä, jollei mitään konkreettista näy työn edistymisestä. (Haikala & Mikkonen 2011, 40-42.)

Perinteisessä mallissakin voidaan siirtyä niin sanottuun iteratiiviseen eli inkrementaaliseen kehitysmalliin, jossa sykli kestää esimerkiksi kuukauden. Vaatimukset voidaan aikatauluttaa eri iteraatioihin ja jokaisessa iteraatiossa voidaan esitellä mitä siinä otetaan toteutettavaksi. Tämän lisäksi myös asiakastoimitukset voidaan jakaa esimerkiksi puolen vuoden välein tapahtuviksi asiakasjulkaisuiksi. Ja jos vaatimusten kartoittaminen ja tarkentaminen tehdään jatkuvana prosessina iteraatio pari kerrallaan sekä hyväksytään etteivät vaatimukset ole tiedossa tarkalla tasolla koko projektille, niin ollaan jo hyvin lähellä ketterän kehityksen mallia. Tässä on kuitenkin oma hintansa, sillä syklin lyhentyessä testauksen ja jatkuvan uudelleen testaamisen tarve lisääntyy ja testiautomaation tarve kasvaa. Vaarana

on myös tuotteen dokumentaation ja arkkitehtuurin rapautuminen. Iteratiivisempaan suuntaan siirryttäessä tarvitaan erinomaista tuotteen- ja vaatimustenhallintaa. (Haikala & Mikkonen 2011, 42.)

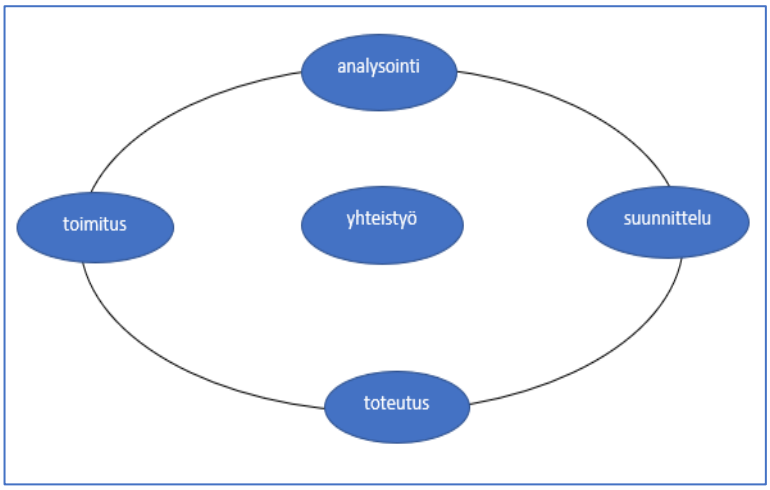
2.3 Ketterä kehitysmalli

Ketterän kehityksen voi määritellä vuonna 2001 julkaistun ketterän ohjelmistojulistuksen kautta, joka esitetty kuvassa 4. Näitä arvoja käyttäen pyrkimys on toimittaa pienissä osissa mahdollisimman paljon liiketoiminta-arvoa erittäin lyhyellä toimitussyklillä. (Crispin & Gregory 2008, luku 1)



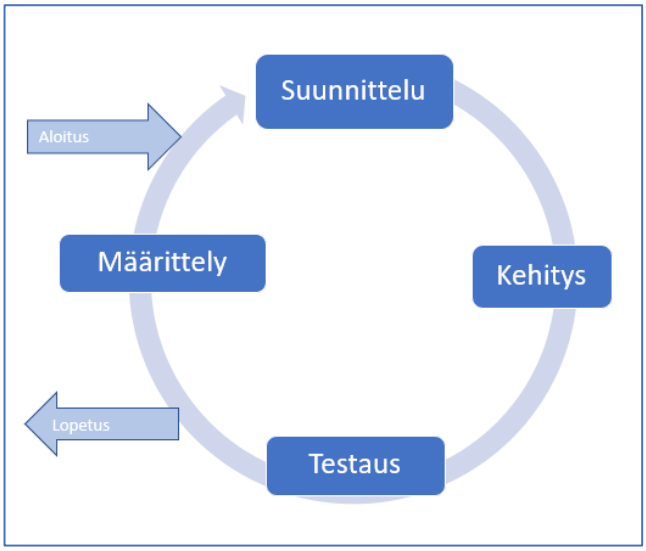
Kuva 4. Ketterän ohjelmistokehityksen julistus (mukaillen Agilemanifesto.org 2001 ja Crispin & Gregory, 2008, luku 1)

Ketterä kehitys kattaa valikoiman eri kehitysmalleja ja perustuu iteratiiviseen lähestymistapaan ohjelmistokehityksessä. Näin tuote rakennetaan pienissä paloissa, joista jokainen lisää ominaisuuksia edellisissä iteraatioissa kehitettyihin asioihin. Ketterässä kehitystavassa keskitytään tuottamaan toimivaa ohjelmistoa alkuvaiheista lähtien ja tämä pyritään toimittamaan asiakkaalle niin pian kuin mahdollista. Näin asiakas saa toimivan tuotteen aiemmin kehitysputkessa, mutta pystyy myös antamaan palautetta tuotteen ominaisuuksista ja elementeistä. Näin ketterällä mallilla pyritään täyttämään asiakkaan tarpeet paremmin. Kuvassa 5 kuvataan yksinkertainen ketterän kehityksen prosessi. Tässä tärkeässä osassa on yhteistyö, jonka ympärillä toimii analysointi, suunnittelu, toteutus ja toimitus jatkuvana. (Coleman, Walsh, Cornanguer, Forgács, Kakkonen & Sabak 2017, 2.)



Kuva 5. Mallinnus ketterän kehityksen mallista (mukaillen Coleman ym. 2017, 2)

Iteratiivisessa kehitysmallissa (kuva 6) vaatimuksien ei tarvitse olla tarkat ennen ohjelmoinnin aloitusta. Tarkkojen vaatimusten sijaan tuotteesta tehdään työversio vaiheissa tai iteraatioissa, josta nimitys tulee. Jokainen iteraatio kattaa toteutettavan osan vaatimusten tarkennuksen eli määrittelyn, suunnittelun, kehitystyön ja testauksen. (Morgan ym. 2015, luku 2)

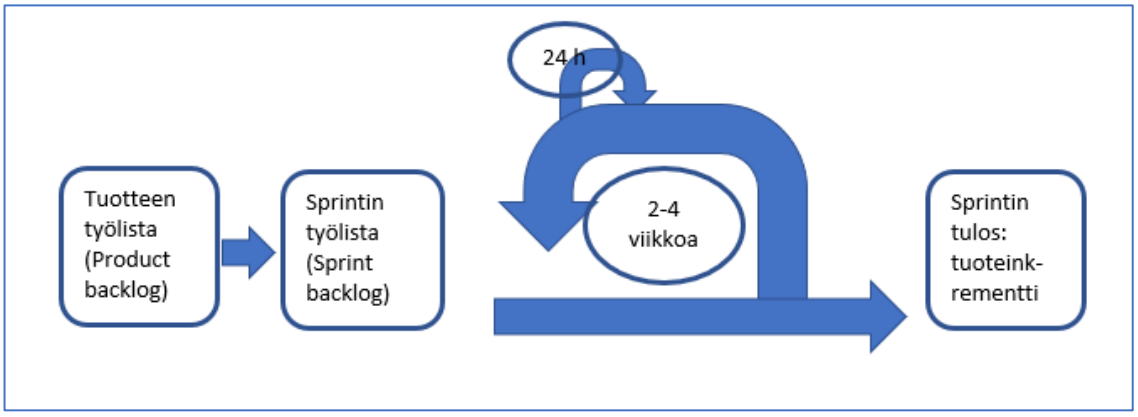


Kuva 6. Iteratiivinen kehitysmalli (mukailtuna Morgan ym. 2015, luku 2)

Tämän tyyppisessä kehityksessä projektille määritellään aikataulu ja hinta, joiden puitteissa suunnitellaan iteraatiota. Iteraatioille määritetään omat aikataulunsa ja hintansa sekä tavoitteet. Kunkin iteraation loppuvaiheessa päätetään seuraavan iteraation sisältö ja näitä jatketaan, kunnes lopputuloksena on toimiva järjestelmä. Tässä tavassa heikkoutena on testausnäkökulmasta dokumentaation puuttuminen. Tätä kompensoimaan voidaan käyttää myös testivetoista kehitystapaa, jossa toiminnalliset testit kirjoitetaan ensin, sitten

ohjelmoidaan ja testataan koodi. Lisäksi muutoksia voidaan tehdä ilman näiden dokumentointia, minkä takia tärkeää on hyvä muutoksen hallinta. (Morgan ym. 2015, luku 2.)

Ketterän kehityksen aikaikkunat ovat lyhyempiä kuin perinteisessä mallissa suunnittelulle sekä kehitys- ja testausvaiheille. Yleensä sprintti tai iteraatio kestää kahdesta neljään viikkoa ja sprintin lopputuloksen voi yleisesti ottaen viedä tuotantoon. Kuvassa 7 kuvataan ketterän kehityksen sprintti käyttäen Scrum-menetelmää mallina, jossa tuotteen työlistalta valitaan sprintin työlista ja toteutetaan tämä sprintissä. Scrum-menetelmään kuuluvat myös päivittäiset tapaamiset. (Coleman ym. 2017, 46.)



Kuva 7. Scrum sprintti (Coleman ym. 2017, 47)

Ketterän kehityksen lähestymistapa testaukseen on yksi merkittävistä eroista perinteiseen kehitysmalliin. Ketterässä mallissa tiimi tekee yhteistyötä ja testaaja on mukana julkaisujen ja iteraatioiden suunnittelussa antaen panoksensa käyttäjätarinoihin, hyväksyntäkriteereihin ja arvioihin. Iteraation käynnistyessä testaaja on mukana heti ja testaa käyttäjätarinoita sitä mukaa kuin kehittäjät koodaavat niitä. (Coleman ym. 2017, 47.)

Ketterän kehityksen hyvinä puolina on hyvä tiedonkulku kehittäjien ja testaajien kesken, sillä he työskentelevät tiiviisti yhdessä ja palaute on jatkuvaa. Pienissä tiimeissä dokumentaatiota ei tarvita paljon. Turhaa työtä jää tekemättä, sillä iteraatiossa saadun asiakkaan palautteen myötä elää vaatimusmäärittely ja voidaan priorisoida ja tarkentaa tuotteen kehityslistaa. (Penmetsa 2016, 31.)

Ketterissä kehitysmalleissa on haasteita saada niitä skaalattua suurille ryhmille ja organisaatioille. Priorisointilistan luonti kaikille projekteille pieniin tarinoihin jaettuna, on järkälemäinen työ ja sitä on vaativa ylläpitää. Ketterässä kehityksessä toteutus alkaa mahdollisimman varhaisessa vaiheessa prosessia ja joskus käytetään liian vähän aikaa suunnitteluun ja arkkitehtuuriin. Tästä syystä asioita saatetaan joutua tekemään uudestaan. Joskus

toteutusten sisään leivottuja riippuvuuksia on vaikea tunnistaa ja voi aiheuttaa ongelmia myöhemmin. (Penmetsa 2017, 31-32.)

Käytännössä suosituinta vaikuttaa olevan yhdistää ketteriä lähestymistapoja vesiputousmallin metodeihin. Organisaatiot yrittävät ylläpitää jatkumoa käytössä olleisiin vesiputousmalleihin luomalla hybridimalleja. Ketteristä kehitysmalleista Scaled Agile Framework (SAFe) on noussut suosiossa selkeästi 2017-2018. Suosiota selittää se, että SAFe käyttää ketterien kehitysmallien toimivia kehyksiä ja yhdistää niihin perinteisten mallien raportoinnin ja hallinnan ominaisuuksia. (Capgemini, Microfocus & Sogati 2019, 23-24.)

3 Testaus

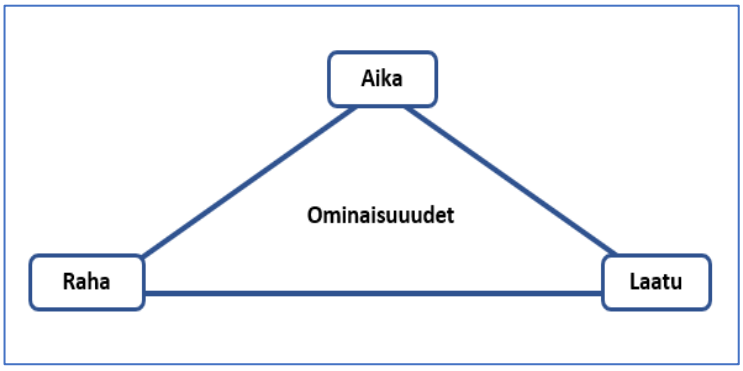
Ohjelmistotestaus on suunnitelmallista virheiden etsintää, jota tehdään suorittamalla ohjelmaa sen osaa. Testaukseen voidaan sisällyttää ohjelman laatua mittaavia ja parantavia menetelmiä. (Haikala & Mikkonen 2011, 205.)

Laadunvarmistamisen ja testaamisen trendit kulkevat käsi kädessä ohjelmistokehityksen muutosten kanssa. Ketterän kehityksen ja DevOps:n suosio nousevat sitä mukaa kun IT-tiimit näkevät ne tärkeinä työkaluina ylläpitämään digitaalisen transformaation tahtia. Yksittäisille tiimeille tarjotaan paremmin mahdollisuuksia valita omat metodit ja tekniikat kehitykseen sekä laadunvarmistukseen ja testaukseen. Kiihtyneeseen julkaisujen tahtiin yhdistettynä sovellusympäristöjen kasvava monimutkaisuus nostaa riskiä vakavien virheiden osalta. Lisäksi nykyään asiakkailla on nollatoleranssi virheille tai huonolle suorituskyvylle ja ohjelmiston huono laatu voi aiheuttaa brändin heikkenemisen ja korjauskulut voivat olla korkeita. (Capgemini ym. 2019, 8)

3.1 Testauksen esittely

Ideaalimaailmassa ohjelma testattaisiin kaikin mahdollisin tavoin. Tämä ei kuitenkaan ole usein mahdollista. Jopa yksinkertaisen ohjelman syötteistä ja tuotoksista tulee satoja tai tuhansia mahdollisia yhdistelmiä. Testitapausten luonti kattamaan nämä kaikki ei ole käytännöllistä. Monimutkaisen sovelluksen täydellinen testaaminen veisi liian paljon aikaa ja vaatisi liikaa resursseja ollakseen taloudellisesti kannattavaa. (Myers ym. 2012, 5.)

Kehitysprosessi tasapainottelee resurssien kanssa. Jos järjestelmä halutaan toimittaa nopeammin, se yleensä maksaa enemmän. Resurssikolmio (kuva 8) kuvaa tätä eli tasapainoteltavana on aika, raha ja laatu. Nämä vaikuttavat toisiinsa ja toiminnallisuuksiin mitä sisällytetään tai ei sisällytetä lopputulokseen. (Morgan ym. 2015, luku 1.)



Kuva 8. Kuvaa resurssikolmiota (mukaillen Morgan ym. 2015, luku 1)

Mitä aiemmin virhe havaitaan sen vähemmän sen korjaaminen maksaa. Virhe voi olla ristiriitainen vaatimus tai ongelma järjestelmäarkkitehtuurissa. Noin 56% testausvaiheessa löydetyistä virheistä juontaa juurensa vaatimusmäärittelyihin ja 27% suunnitteluvaiheeseen. (Kasurinen 2015, 62.)

Testauksessa löydetään järjestelmästä virheitä, ei osoiteta järjestelmän olevan virheetön. Testauksella voidaan myös osoittaa, että virhe on korjattu. Testauksen yleisin tavoite on löytää virheitä, joten ne suunnitellaan löytämään virheitä. Tyhjentävä, kaikenkattava testaus on mahdotonta, etenkin kun kyseessä on laaja ja monimutkainen järjestelmä. Kattavan testauksen lähestymistavassa käytetään testauksessa kaikkia mahdollisia datayhdistelmiä. (Morgan ym. 2015, luku 1.)

Testaaminen ja virheenjäljittäminen (debugging) ovat kaksi erilaista ja tärkeää toimintaa. Virheenjäljittämisessä kehittäjä etsii syytä virheelle koodissa korjatakseen sen. Testaaminen taas on järjestelmän tai sen osan systemaattista tutkintaa, jossa päämääränä on löytää ja raportoida virheistä. Testaamiseen ei sisälly virheiden korjaus, vaan tieto korjaustarpeesta välitetään kehittäjälle korjattavaksi. Testaamiseen sisältyy muutosten ja virheiden korjausten jälkeen järjestelmän tai sen osan tarkastaminen. Tehokas virheenjäljittäminen on edellytys ennen testauksen aloitusta, jotta järjestelmä tai sen osa on tarpeeksi laadukasta testaukseen. Virheenjäljitys ei kuitenkaan tuo luottamusta siihen, että järjestelmä tai sen osa täyttäisi vaatimukset täysin. Testauksessa tutkitaan tarkasta järjestelmän tai sen osan käyttäytymistä ja kaikki viat raportoidaan kehitystiimille korjattavaksi. Korjausten jälkeen testataan tarpeeksi kattavaksi se, että korjaus on toiminut. Molempia tarvitaan laadukkaaseen lopputulokseen päästäkseen. (Morgan ym. 2015, luku 1.)

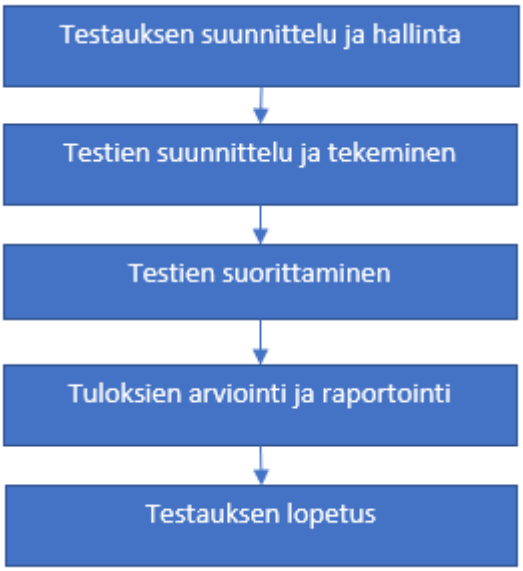
Testauksessa peruslähestymistapoja on musta laatikko -testaus ja lasilaatikkotestaus. Musta laatikko -testauksessa testataan antamalla syötteitä ja katsotaan miten käy. Tässä ei tarkastella mitä tapahtuu ohjelman sisällä, vaan tarkastetaan lopputulos vaatimusmäärittelyitä vasten. Lasilaatikkotestauksessa tarkastellaan mitä ohjelman sisällä tapahtuu testauksen aikana. Tässä testataan antamalla syötteitä ja tarkastellaan mitä tapahtuu järjestelmän sisällä. Lisäksi on harmaa laatikko -testaus, joka on yhdistelmä näitä molempia. (Kasurinen 2013, 65-68.)

Ohjelmistotestauksessa tarvitaan sekä staattista että dynaamista testausta. Staattisessa testauksessa ei suoriteta ohjelmaa tai koodia, vaan siinä käydään läpi dokumentaatiota tai koodia ajamatta sitä. Dokumentaation läpikäynti ja sieltä löytyvän virheen korjaus on huomattavasti edullisempaa kuin virheen korjaus myöhemmin. Staattisessa testauksessa käytetään esimerkiksi katselmointeja, joissa voidaan estää virheiden syntymistä poistamalla

epäselvyyksiä ja virheitä määrittelydokumenteista. Dynaamisessa testauksessa suoritetaan järjestelmää testissä testidatalla. (Morgan ym. 2015, luku 1.)

Testausprosessi käsittää valmistelevaa työtä, jossa suunnitellaan testit ja mahdollistetaan niiden ajaminen sekä testien ajon jälkeen on tulosten raportointi ja kattavuuden arviointi. Tärkeintä on päättää, mitä testauksella pyritään saavuttamaan ja asetetaan joka testille selkeät tavoitteet. Testi, joka on suunniteltu varmistamaan järjestelmän toiminnat vaatimusten mukaisiksi, eroaa testistä, jolla pyritään hakemaan mahdollisimman paljon virheitä. Testausprosessi määritetään varmistamaan, ettei kriittisiä vaiheita jää puuttumaan ja asiat tehdään oikeassa järjestyksessä. (Morgan ym. 2015, luku 1.)

Testausprosessi koostuu Morgan ym. (2015, luku 1) mukaan seuraavista osista: testauksen suunnittelu ja hallinnointi, testien suunnittelu ja tekeminen, testien suorittaminen, tuloksien arvioin ja raportointi sekä testauksen lopetus. Prosessi esitelty kuvassa 9.



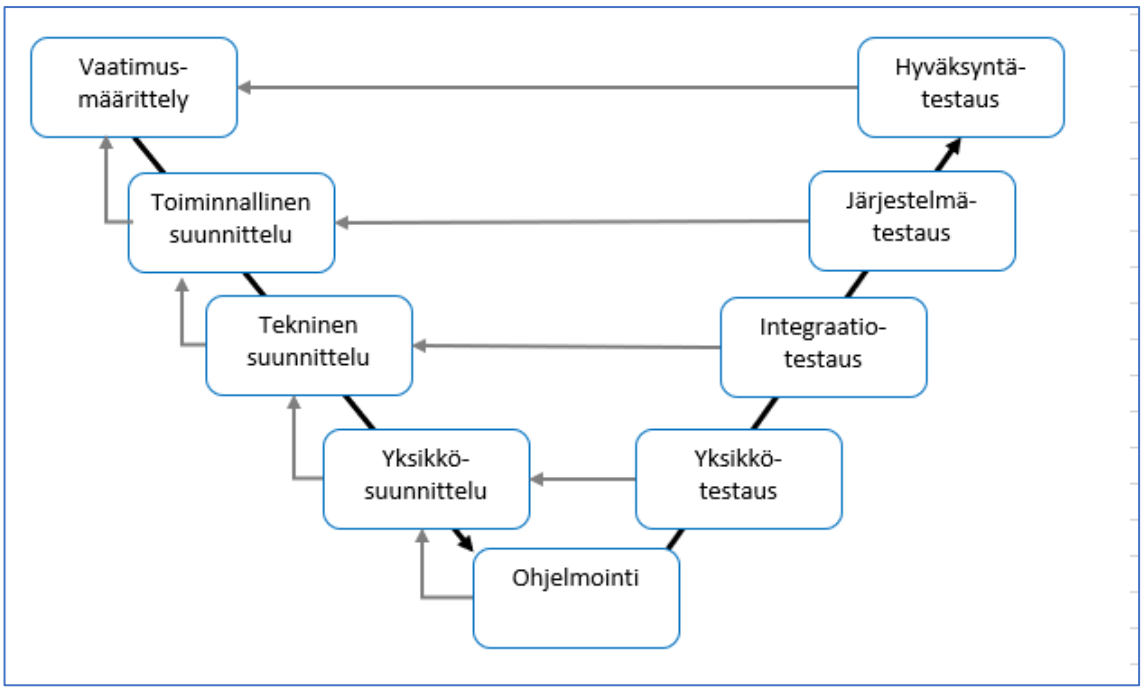
Kuva 9. Testausprosessi (mukaillen Morgan ym. 2015, luku 1)

Vähävirheisen koodin tuottaminen on kriittistä toimittamisen nopeudelle ja muuttamisen helppoudelle. Testaus ja virheiden korjauksen vaatima aika ovat suhteessa virheiden määrään. Jos virheiden määrä kaksinkertaistetaan, niin testausaika tarvitaan samassa suhteessa enemmän. Ja jos koodin virheiden määrä lisääntyy yli tietyn pisteen, ei koodista saa enää vakaata, vaan se on jatkuvassa syklissä, jossa toistuvat testaus ja korjaus loputtomasti. (Morgan ym. 2015, luku 1.)

3.2 Ohjelmistotestauksen perinteisessä kehitysmallissa

Ohjelman kehitystyön ja testauksen suhdetta havainnollistetaan usein V-mallilla, jonka mukaisesti testauksen suunnittelu tapahtuu testaustasoa vastaavalla suunnittelutasolla. Vastaavasti tulokset todetaan oikeiksi vertaamalla niitä mallissa vastaaviin dokumentteihin. Mallin mukaisesti erillisiä testaustasoja ovat yksikkötestaus (module testing, unit testing), integrointitestaus (integration testing) ja järjestelmätestaus (system testing). (Haikala & Mikkonen 2011, 206-207.)

Ideana V-mallissa on se, että kehityksen ja testauksen työt ovat yhtä tärkeitä ja toisiaan vastaavia. V-mallin V:n sakarat kuvaavat tätä. Vasen haara kuvaa vesiputousmallistakin tuttua kehitysprosessia, jossa järjestelmää kehitetään suunnittelusta ohjelmointiin. Oikea haara taas edustaa integraatio- ja testausprosesseja eli järjestelmän elementtejä kootaan isommiksi alijärjestelmiksi ja näiden toiminnallisuuksia testataan. Integraatio ja testaus päättyvät, kun koko järjestelmän hyväksyntätestaus on suoritettu. Kuva 10 kuvaa V-mallia, jossa musta nuoli kuvaa toteutusjärjestystä ja harmaa nuoli kuvaa todentamisen ja validoinnin polkua. (Spillner, Linz & Shaefer 2014, 39-40.)



Kuva 10. V-malli (mukaillen Spillner ym. 2014, luku 3.)

Spillner ym. (2014, 40-41) kertovat V-mallin vasemman haaran kehityksen tasojen kautta järjestelmän kuvattavan aina tarkentuvalla tasolla:

- **Vaatimusmäärittely**: Asiakkaan tai tulevan järjestelmän käyttäjän tarpeet ja vaatimusmäärittelyt kerätään, tarkennetaan ja hyväksytään. Siten määritellään järestelmän tarkoitus ja halutut ominaisuudet.
- **Toiminnallinen suunnittelu**: tässä vaiheessa kartoitetaan uuden järjestelmän dialogit ja toimnnallisuudet.
- **Tekninen suunnittelu**: Tässä vaiheessa suunnitellaan järjestelmän toteutus. Tämä sisältää käyttöliittymän määrittelynm järjestelmän ympäristön ja järjestelmä jaetaan pienempiin ymmärrettäviin alijärjestelmiin/kokonaisuuksiin. Jokainen alijärjestelmä on sitten kehitettävissä niin itsenäisesti kuin mahdollista.
- **Yksikkösuunnittelu**: Jokainen yksikkö kuvataan, mukaan lukien niiden tarkoitus, käyttäytyminen, sisäinen rakenne ja rajapinnat muihin alijärjestelmiin.

Virheet ovat helpointa löytää sillä tasolla missä ne esiintyvät, minkä vuoksi V-mallissa jokaista kehitystasoa vastaa testauksen taso, jotka Spillner (2014, 40-41) kuvaavat seuraavasti:

- **Yksikkötestaus**: varmistaa, että jokainen järjestelmän yksikkö vastaa määrittelyä.
- **Integraatiotestaus**: tarkistaa, että yksiköt toimivat ryhmissä määritellyllä tavalla.
- **Järjestelmätestaus**: Varmistaa, että järjestelmä kokonaisuudessaan toimii määritellysti.
- **Hyväksyntätestaus**: tarkistaa, että järjestelmä vastaa asiakasvaatimuksia, kuten määritetty sopimuksessa ja/tai vastaako järejstelmä käyttäjätarpeita ja -odotuksia.

Jokaisessa testaustasossa varmistetaan, että kehityksen tulos vastaa kyseisen tason määrittelyitä ja vaatimuksia. Tässä varmistetaan ovatko määrittelyt ja tarpeet oikein toteutettu ja vastaako tuote määrittelyä, ottamatta kantaa siihen onko tuote soveltuva tarkoitettuun käyttöön. V-mallin tärkeimmät ominaisuudet ovat, että toteutus ja testaus on erillisiä toimintoja, mutta yhtä tärkeitä. Eri testitasot erotetaan toisistaan, joissa jokaista testitasoa vastaa oma kehitystaso. (Spillner ym. 2014, 41-42.)

Spillner ym. (2014, 42) kertovat, että V-mallista voi saada sen virheellisen käsityksen, että testaus alkaa vasta järjestelmän toteutuksen jälkeen. Testitasot mallin oikeassa haarassa pitäisi oikeammin tulkita testien toteutuksen tasoina. Testien valmistelu eli suunnittelu, analysointi ja mallintaminen, alkavat aiemmin ja suoritetaan V-mallin vasemman haaran vastaavien toteutusvaiheiden aikana eikä tätä kuvata V-mallissa suoraan. V-mallin testitasojen erottelussa on päämääränä kuvata teknisesti hyvin erilaisia testaustasoja: niillä on erilaiset tavoitteet ja siten tarvitaan erilaisia metodeja ja työkaluja sekä vaaditaan henkilöstöltä erilaista tietoa ja taitoja. (Spillner ym. 2014, 42.)

Yksikkötestaus on tavallisin testausmuoto ja siinä tarkastellaan yksittäisen olion, moduulin tai funktion toimintaa toteutuksen yhteydessä, yleensä koodin ohjelmoineen henkilön toimesta. Tässä on tarkoitus varmistaa, että ohjelmoitu uusi koodi tai muutos toimii halutusti. Tässä tarkistetaan koodin virheetön kääntyminen sekä toiminnallisten ominaisuuksien toiminta erilaisin syöttein. Tätä tarkoitusta varten rakennetaan esimerkiksi joukko olio- ja

funktiokutsuja, joihin pitää tulla oikeanlainen vastaus tai toiminnon toteutus. Näin toiminta voidaan varmistaa ennen kuin komponentti viedään osaksi laajempaa ohjelmaa. Yksikkötestauksessa on sen luonteen takia paljon testitapauksia ja näissä auttaa kehittyneemmän editorityökalun tai älykkään kääntäjän avulla työskentely. Lisäksi testien koodikattavuuden arvioinnin työkalut auttavat näkemään suoraan esimerkiksi sen, mitä osia ei ole testiajettu. (Kasurinen 2013, 51-55.)

Integrointitestauksessa yhdistetään osia ja muodostetaan kokonaisia osajärjestelmiä. Pääasiallinen tutkinnan kohde on rajapintojen toimivuus. Testitulosten vertailu on yleisimmin tekniseen määrittelyyn. Tätä testausvaihetta edistetään yksikkötestauksen kanssa eikä välttämättä tarkastella erikseen. Integrointi etenee joko kokoavasti alhaalta ylöspäin tai päinvastaisesti ylhäältä alaspäin eli osittavana integrointina. (Haikala & Mikkonen 2011, 207-208.)

Järjestelmätestaukseen siirrytään, kun komponentit on yksikkötestattu rakennusvaiheessa ja koottu integraatiotestauksessa kokonaisuudeksi. Järjestelmätestaus on yleisnimike kokonaiselle järjestelmälle tehtävälle testaukselle eikä sinällään testaustavan nimi. Tässä vaiheessa tavoitteena on varmistaa, että järjestelmä toimii kokonaisuutena sekä toteuttaa tavoitteet. (Kasurinen 2013, 56-57.)

Haikala & Mikkonen määrittävät järjestelmätestauksen niin, että koko järjestelmä on tarkastelussa. Testauksessa tehdään vertailua vaatimusmäärittelyyn, käyttöohjeeseen, toiminnalliseen määrittelyyn sekä muuhun tehtyyn asiakasdokumentaatioon. Myös ei-toiminnallisuksia ominaisuuksia voidaan testata tässä, kuten esimerkiksi kuormitustestausta eli kestääkö järjestelmä oletetun määrän kuormitusta tai enemmän. Tässä voi myös testata toimiiko asentaminen odotetusta, kestääkö järjestelmä oletetun määrän kuormitusta sekä tehdä käytettävyystestausta. (Haikala & Mikkonen 2011, 205-209.)

Järjestelmätestaus on tarpeen, sillä yksikkö- ja integraatiotestausten tulokset eivät vastaa järjestelmän tuotantokäyttöä. Järjestelmätestauksessa vastaavasti keskitytään järjestelmän käyttäytymiseen tuotantoa vastaavassa ympäristössä. Testaus hoidetaan yleensä kehityksestä erillään olevan tiimin toimesta. Tästä on hyötynä objektiivisempi järjestelmän arviointi, vaatimuksiin perustuen eikä koodiin. (Morgan ym. 2015, luku 2.)

Hyväksymistestaus on V-mallin viimeinen työvaihe ja sen tarkoituksena on osoittaa järjestelmän täyttävän vaatimusmäärittelyssä asetetut vaatimukset. Tämä on järjestelmän virallinen tarkastelu, minkä normaalisti mennessä onnistuneesti läpi asiakas hyväksyy tuotteen valmistumisen ja lakiteknisesti ohjelmisto siirtyy asiakkaalle. Hyväksymistestauksessa on

tavallisempaa kuin järjestelmätestauksessa, että ohjelmisto käytetään kohdeympäristössä. (Kasurinen 2013, 57.)

Hyväksyntätestauksen päätarkoitus on osoittaa järjestelmän vastaavuus asiakkaan vaatimuksiin sekä operationaalisiin ja ylläpidollisiin prosesseihin. Hyväksyntätestaus voi esimerkiksi kartoittaa järjestelmän valmiutta käyttöönottoon ja käyttöön. Hyväksyntätestaus on usein asiakkaan tai järjestelmän käyttäjien vastuulla, vaikkakin projektin jäseniä voi olla myös osallisia. (Morgan ym. 2015, luku.)

Hyväksyntätestaus sisältää yleisimmin seuraavia Morgan ym. mukaan käyttäjien hyväksyntätestauksen, operationaalisen hyväksyntätestauksen, sopimuksellisen ja säännösten hyväksyntätestauksen ja alpha- ja betatestauksen. Ensimmäisessä käyttäjien edustajat testaavan järjestelmää varmistaakseen, että se täyttää liiketoimintatarpeet. Operationaalisesta hyväksyntätestauksesta kutsutaan usein myös operationaalisen valmiuden testaamiseksi. Tämä sisältää prosessien ja menettelytapojen tarkastamisen, jotta järjestelmää voidaan käyttää ja ylläpitää. Tarkasteltavia asioita on esimerkiksi varmuuskopiointimahdollisuudet, ongelmista toipumisen menettelyt, ylläpitomenetelmät ja turvallisuusprotokollat. Sopimuksellinen ja säännösten hyväksyntätestauksessa varmistetaan sopimukseen mahdollisesti dokumentoidut järjestelmän hyväksyntäkriteerit tai joidenkin alojen vaatimat hallinnollisten, laillisten tai turvallisuusstandardien täyttymiset. Alphatestauksessa operationaalinen järjestelmä testataan kehittyjän alustalla oman henkilökunnan toimesta ennen kuin julkaistaan ulkoisille asiakkaille. Tässäkään ei testaukseen liity kehitystiimi. Betatestauksessa järjestelmää testataan joukolla asiakkaita, jotka käyttävät tuotetta omista sijainneistaan ja antavat palautetta ennen kuin järjestelmä julkaistaan. Tätä kutsutaan usein kenttätestaamiseksi. (Morgan ym. 2015, luku 2.)

Mitä pidemmälle V-mallissa on ehditty testaustasoilla, sen kalliimpaa korjaus on. Jos virhe havaitaan ja korjataan järjestelmätestauksessa, se vaikuttaa todennäköisesti useisiin komponentteihin. Myös muut komponentit kannattaisi testata siltä varalta, että jää huomaamatta jokin muutostarve. Tämän jälkeen pitäisi tehdä mieluiten kokonaan uusiksi järjestelmätestaus. Virheiden korjaus saattaa poikia uusia virheitä, mikä hankaloittaa korjausprosessia. Tätä uudelleentestausta kutsutaan regressiotestaukseksi huolimatta testaustasosta. (Haikala & Mikkonen 2011, 208.)

Perinteisessä kehitysmallissa keskitytään varmistamaan, että kaikki vaatimukset on toteutettu tuotteeseen. Jos kaikki ei ole valmista alkuperäiseen julkaisupäivään mennessä, julkaisua yleensä siirretään. Kehitystiimin ei ole yleensä mahdollista vaikuttaa mitä ominai-

suuksia julkaisu sisältää, tai kuinka niiden tulisi toimia. Kehittäjät erikoistuvat yleensä johonkin koodin ominaisuuteen. Testaajat tutkivat vaatimusmäärittelyitä kirjoittaakseen testisuunnitelmat ja odottavat että heille toimitettaan työ testattavaksi. (Crispin & Gregory 2008, luku 1.)

Haikala & Mikkonen nostavat esiin yleisen periaatteen, että toteuttajat jäisivät testaustyön ulkopuolelle. Toteuttajat yleensä keskittyvät ohjelman niin osiin, jotka tiedetään toimiviksi ja siten löydettyjen virheiden määrä jää ulkopuolisen testaajan virheiden löytömäärää pienemmäksi. Käytännössä kehittäjät tekevät yksikkötestauksen ja välillä integrointitestauksen. Ulkopuoliset testaajat aloittavat työnsä siten järjestelmätestausvaiheessa. Luonnollista on, että kehittäjä kokeilee järjestelmää eri syöttein, mutta tätä ei lasketa varsinaiseksi testaukseksi. (Haikala & Mikkonen 2011, 209.)

3.3 Ohjelmistotestaus ketterässä kehityksessä

Ketterässä kehityksessä Cline (2015, 221) kuvaa testaajan roolin koodin laadun suojelijaksi eikä mikään etene ilman että testaaja hyväksyntää tuotteen valmiudesta. Vaikka kehittäjä tekee parhaansa alatasolla yksikkötestauksessa estääkseen virheiden pääsyn koodiin, testaaja varmistaa tuotteen käyttäytymisen ylätasolla varmistaakseen, että versio vastaa vaatimuksia ja sovittuja laatustandardeja. Siten testaajan ja kehittäjän tulkinnat lisäävät yhden lisätason varmistamaan, että tuote käyttäytyy kuten käyttäjä tarvitsee. (Cline 2015, 221.)

Testaajat saavat syötettä liiketoiminta-analyytikolta (vaatimukset ja käyttäjäkokemus tuotokset) ja kehittäjiltä (suunnittelu, koodi ja yksikkötestit) ja tuottavat syötettä takaisin tiimille (virheet korjaukseen analyytikolle, kehittäjälle tai testaajalle) ja ketterän kehityksen projektipäällikölle (testien tulokset raportointiin). (Cline 2015, 221.)

Ohjelmistokehityksen käytännöt korostavat kahta strategiaa virheiden poistamiseksi: usean tason testaus ja koodin tarkastelu. Testausta suositellaan tehtäväksi usein, jotta jokainen ohjelmistoiteraatiosta on toimiva versio. Ketterät menetelmät suosittavat jatkuvaan integraatiotestaukseen ja useisiin käännöksiin. Ketteriin menetelmiin kuuluu hyväksyntätestaus asiakkaiden kassa, sekä testitapauksin että asiakkaan fokusryhmien kautta. Ketteriin menetelmiin kuuluu myös koodin tarkastelu. Ketterissä menetelmissä kehittäjät saattavat käyttää vähemmän muodollista testausdokumentaatiota ja luottavat enemmän kehitystiimin testaamiseen kuin erillisiin testaustiimeihin, mutta he ovat yhtä omistautuneita virheiden poistamiseen. (Morgan ym. 2015, luku 1.)

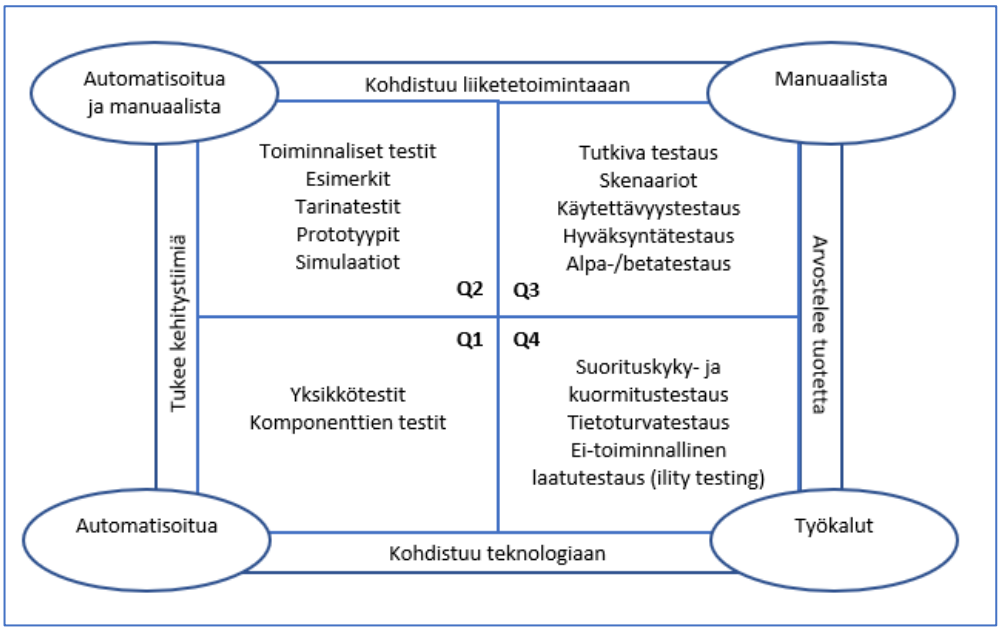
Iteraation aikana on kahdenlaista testaajaa: käyttöliittymätestaaja ja integraatiotestaaja. Yksi ihminen voi toimia molemmissa rooleissa, mutta näissä tarvitaan hieman erilaisia taitoja. Käyttöliittymätestaaja varmistaa, että käyttöliittymä on tehty oikein ja tuotteella on oikea ulkoasu ja tuntuma, mikä on subjektiivinen termi käyttäjän holistiselle tuotteen käyttökokemukselle. Käyttöliittymätestaaja varmistaa, että käyttöliittymä (näytöt, raportit ja kaikki mitä käyttäjä näkee) toimii kuten odotettu, sisältää oikean tiedon oikeaan aikaan ja siirtyy seuraavalle näytölle kuten odotettu. Integraatiotestaaja taas varmistaa, että tuote toimii kokonaisuutena kuten vaatimuksissa määritellään. Integraatiotestaaja käy läpi koko käyttäjätarinan: ohjelmointitaitoja vaaditaan automatisoitujen integraatiotestien kirjoittamiseen. Integraatiotestaajat ovat yleensä tottuneita kehitysympäristöihin, sillä automatisointitestaustyökalut, ovat usein samoja kuin mitä kehittäjät käyttävät yksikkötestauksessa. Ketterässä kehityksessä testaus tapahtuu iteraation aikana ja hyväksyntätestaus tapahtuu käyttäjädemolla tai version julkaisu vaiheessa eikä sen enempää toiminnallista testausta tarvita. Iteraation aikana ajetaan neljänlaisia testejä:

- Yksikkötestit: kehittäjän tekemät ja ajamat
- Integraatiotestit: käyttöliittymä ja toiminnalliset testit, testaajien tekemät ja ajamat
- Regressiotestit: ajetaan joka buildissä eli käännöksessä
- Laatutestit: kokoelma testejä, joilla varmistetaan standardien täyttyminen

(Cline 2015, 221-223.)

Regressiotestaus on yleistermi uudelleentestaukselle. Käytännössä kun muutetaan toimivan järjestelmän osaa, niin regressiotestauksella varmistetaan järjestelmän jatkuva toiminta. Tätä voidaan tehdä myös järjestelmän vaiheittain kehittämisessä, jossa lisätessä toiminnallisuuksia, varmistetaan että aiemmin tehty järjestelmä toimii edelleen. Eli varmistetaan että kaikki toimii edelleen eikä muutos ole tuonut uusia ongelmia esimerkiksi liittyvän komponentin päivityksen yhteydessä tai kehityshaarojen yhdistämisessä aiemmin korjatut virheet ovat poistuneet uudesta versiosta. Regressiotestauksessa voidaan hyödyntää hyvin automatisointia, sillä samoja testejä tarvitaan todennäköisesti monesti ja voidaan toistaa projektin aikana useasti. (Kasurinen 2013, 68-70)

Ketterän kehityksessä voidaan käyttää ketterän kehityksen nelikenttää tarkastelemaan projektissa tarvittavan testauksen ja testaustyyppien kattavuutta. Nelikentässä testaus jaetaan kehitystiimiä tukeviin testeihin ja tuotetta arvosteleviin testeihin sekä liiketoimintaan ja teknologiaan kohdistuviin testeihin. Tämä esitellään kuvassa 11. (Crispin & Gregory 2008, luku 6.)



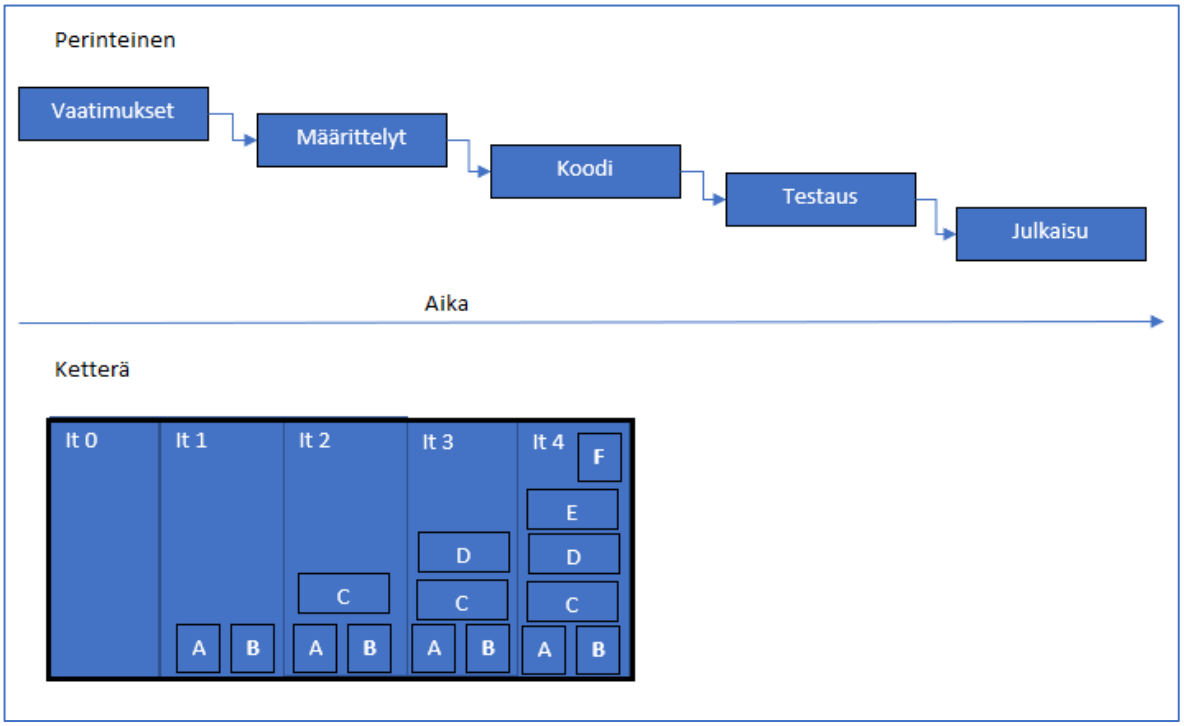
Kuva 11. Ketterän testauksen nelikenttä (mukaillen Crispin & Gregory 2008, luku 6)

Kenttien numeroinnit eivät kuvaa missä järjestyksessä testejä tehdään, vaan niiden ajoitus riippuu esimerkiksi projektin riskeistä, tavoitteista ja luonteesta. Nelikentän vasemmanpuoleiset kentät ovat kehitystiimiä tukevia tuotteen kehityksessä. Tämä on enemmän vaatimusten määrittelyä ja suunnittelun apuja kuin mitä perinteisesti ajatellaan testauksesta. Kenttä Q1 edustaa testivetoista kehitystä ja kohdistuu teknologiaan. Testivetoisessa kehityksessä testien kirjoittaminen auttaa kehittäjiä suunnittelemaan koodinsa hyvin ja nämä ovat automatisoitavissa. Kenttä Q2 tukee kehitystiimiä korkeammalla tasolla. Nämä liiketoimintaan kohdistuvat testit määrittävät ulkoista laatua ja asiakkaan haluamia ominaisuuksia. Näistä osa on automatisoitavissa ja osa on manuaalista testausta. (Crispin & Gregory 2008, luku 6.)

Nelikentän oikeanpuoleiset kentät pyrkivät arvioimaan tuotetta. Kenttä Q3 kohdistuu liiketoimintaan ja arvioi vastaako ohjelmisto odotuksia. Tässä yritetään testata järjestelmää oikean käyttäjän näkökulmasta ja tämä on pääosin manuaalista testausta. Kenttä Q4 kohdistuu teknologiaan ja yrittävät arvioida tuotteen ominaisuuksia, kuten suorituskykyä ja tietoturvaa. Testauksessa käytetään pääosin testauksen työkaluja ja yksi näistä on automatisointi. (Crispin & Gregory 2008, luku 6.)

Ketterän projektin lyhyisiin iteraatioihin siirtyminen voi olla hankalaa etenkin isommille organisaatioille. Ketterät tiimit työskentelevät läheisesti liiketoiminnan kanssa ja heillä on tarkka ymmärrys vaatimuksista. He keskittyvät tuotettavaan arvoon ja heillä voi olla paljonkin vaikutusmahdollisuutta iteraatioiden priorisoinneissa. Testaajat eivät odota työtä, vaan he etsivät tapoja avustaa koko kehityssyklin ajan. (Crispin & Gregory 2008, luku 1)

Perinteistä ja ketterää kehitysmallia vertaillessa (kuva 12), huomaa että perinteisessä mallissa testaus on lähellä loppua omana osionaan. Ketterässä kehityksessä otetaan soveltuva osa ohjelmistosta käsittelyyn, joka koodataan ja testataan. Osa voi olla toiminnallisuus tai kehitysaihio. Kun osan toiminta on varmistettu, niin koko tiimi siirtyy seuraavaan osan työstöön. (Crispin & Gregory 2008, luku 1.)



Kuva 12. Perinteisen vaiheittaisen ja ketterän kehityksen mallien vertailu (mukaillen Crispin & Gregory 2008, luku 1)

Black (2017, 38) nostaa pääeroiksi ketterän ja perinteisen kehitysmallin testauksessa testaajien mukana olon kautta koko julkaisun ja jatkuvan testauksen läpi iteraation. Lisäksi ketterässä kehityksessä korostetaan valmennusta, yhteistyötä ja parityöskentelyä sekä joustavuutta muutoksiin. (Black 2017, 38.)

Ketterät metodit eivät tarjoa varsinaisia ratkaisuja, mutta ovat hyviä ongelmien huomaamisessa varhaisessa vaiheessa. Ketterässä kehityksessä testitiimin pitää testata jokainen iteraatio kuten se olisi julkaisuversio ja integroida se lopputuotteeseen. Jatkuvan integraatiotestauksen tuloksena testitiimin on täytynyt suunnitella ja rakentaa testi-infrastruktuuri sekä ketterä nopeatahtinen kehitys ja julkaisu sekä julkaisutyökalut. Ketterän kehityksen testauksen onnistumiseen vaikuttaa tiimin kyky automatisoida testiympäristön luonti, julkaisuehdokkaan validointi ja testimetriikan raportointi. (Penmetsa 2016, 29)

3.4 Testaustyökalut

Testauksen työvälineet ovat tavallisimmin samat kuin itse ohjelmistokehityksen työvälineet. Kehitysympäristö ja havaittujen virheiden raportoimiseen käytetyt kommunikointivälineet ovat yleensä tärkeimmät testaajien työvälineet. Testausautomaation järjestelmät, yksikkötestauksen työkalut ja testitapausten hallinnointivälineet ovat tavallisimpia työkaluja testauksessa suomalaisissa ohjelmistokehityksen organisaatioissa. Lisäksi käytössä on esimerkiksi vikojen raportoinnin järjestelmiä, testitapauksien suunnittelun ja kehityksen työkaluja, lähdekoodianalysaattoreita, simulaattoreita ja testiraporttigeneraattoreita. Työkalujen valintaan vaikuttaa se, mikä teollisuuden ala on kyseessä. Testaajien työvälineet voivat olla osa valmista työkalupakettia, esimerkiksi mobiilisovellusten rakentamiseen on ohjelmistokehityspaketteja missä on sekä kehityksen että testaamisen työkalut. (Kasurinen 2013, 84-85.)

Kasurinen (2013, 85-90) on poiminut testaustyön oleellisia työvälineitä ja jakaa ne testauksen tekemisen ja testauksen hallinnan työkaluihin. Testauksen tekemisen työkalut ovat kehitysympäristöt, testausautomaation työkalut, vikatietokannat, tyngät, analysaattorit ja dokumenttipohjat. Testauksen hallinnan työkaluihin kuuluvat testitapausten hallintaympäristöt, laadunvalvontatyökalut, ohjelma-arvioinnit, koodikatselmoinnit, tarkastukset ja viestintävälineet. (Kasurinen 2013, 85-90.)

Kehitysympäristöillä, kuten Eclipse, voidaan hallita staattinen testaus ja yksikkötestaus. Testausautomaation työkalut ovat usein organisaatiossa itse tehtyjä testaustyökaluja, joita käytetään erityisesti komponenttien tai rajapintojen toimintaa testaamaan. Testausautomaatiolle on eniten käyttöä regressiotestauksessa sekä varmennettaessa laatua ja kuormituksen kestävyyttä. Vikatietokannat ovat yleensä ohjelmistotalojen kaupallisia järjestelmiä. Vikatietokantaan kehittäjät, testaajat ja koekäyttäjät voivat raportoida virheet. Tiedot voidaan koota raportteihin ja käyttää esimerkiksi työlistana. Tyngät (test stubs) ovat yksinkertaistettuja tai lyhyitä sijaiskomponentteja, jotka mahdollistavat ohjelman komponenttien testaamisen matkimalla oikealle komponentille toista komponenttia. Näin voidaan yksikkö- tai integraatiotestauksessa osajärjestelmän testaamista tai korvata muutoin raskas operaatio. Analysaattorit ovat testauksen apuvälineitä, joilla käydään läpi järjestelmän lähdekoodia tai komponentteja. Näin tutkitaan ja tarkastetaan tunnettujen ongelmien esiintymistä järjestelmässä. Esimerkiksi lähdekoodianalysaattorilla voidaan tarkista komponenttien lähdekoodia, kuten onko se syntaksinmukaista. Ja dokumenttipohjat ovat testauksen tehtävistä syntyy dokumentaatiota, joten dokumenttipohjat auttavat pitämään dokumentaation hyvänä ja yhtenäisenä. Esimerkiksi lomakepohja, joka täytetään suoritettaessa testitapausta. (Kasurinen 2013, 85-90.)

Työkalujen kategorisointiin on erilaisia tapoja, esimerkiksi kategorisointi tarkoituksen mukaan, perustestiprosessin ja niiden ohjelmiston elinkaaren toimintojen mukaan, tukemansa testityypin mukaan, työkalun lähteen mukaan (jaettu tai avoin lähdekoodi, ilmainen tai kaupallinen), käytetyn tekniikan mukaan tai sen mukaan kuka niitä käyttää. (Morgan ym. 2015, luku 6.)

Morgan ym. (2015, luku 6) käyttävät ISTQB:n Foundation Level kanssa samaa luokittelua, jossa jaotellaan työkalun tukeman toimintatyypin mukaan kuuteen kategoriaan. Ensimmäisessä on testauksen ja testien hallinnan työkalut, joita käyttävät useimmiten testaajat. Näihin kuuluvat testauksen hallintatyökalut, havaintojen hallintatyökalut, vaatimusten hallinnantyökalut sekä kokoonpanon hallintatyökalut. (Morgan ym. 2015, luku 6.)

Toisessa kategoriassa on staattisen testauksen työkalut, joita käyttävät yleensä kehittäjät. Näihin kuuluvat katselmointityökalut, staattiset koodin analysointiohjelmat, koodin tarkastajat sekä mallinnustyökalut. (Morgan ym. 2015, luku 6.)

Kolmantena on testimäärittelytyökalut, joita käyttävät yleensä testaajat. Tähän kategoriaan kuuluu testien suunnittelutyökalut, skriptien luontityökalut, testioraakkelit ja testidatan valmistelutyökalut. Testioraakkelin avulla määritellään odotettuja tuloksia testeille. Testidatan valmistelutyökalun käyttäjiä voi olla muitakin kuin testaajat. (Morgan ym. 2015, luku 6.)

Neljännessä kategoriassa on testien ajamisen ja virheiden jäljityksen työkalut, joita käyttävät testaajat ja kehittäjät. Yleensä testaajien työkaluja ovat testien suoritustyökalut, kehittäjien käytössä testikehykset, testipedit sekä kattavuuden mittaustyökalut ja molempien käytössä testivertailijat. Lisäksi tietoturvatestauksen asiantuntijoilla on tietoturvatestaustyökalut. (Morgan ym. 2015, luku 6.)

Viidennen kategorian työkaluilla eli suorituskyvyn ja monitoroinnin työkaluilla on useita käyttäjiä (ei vain yhtä ryhmää) ja työkaluja ovat monitorointityökalut ja datalaadun arviointityökalut. Tähän kategoriaan kuuluu myös kehittäjien käyttämät dynaamisen analyysin työkalut, joilla saadaan ajonaikaista tietoja ohjelmistokoodin tilasta. Lisäksi kategoriaan kuuluvat suorituskykytestaus- ja kuormitustyökalut, joita käyttävät suorituskykytestauksen asiantuntijat. Myös käytettävyystestaustyökalut kuuluvat tähän ja niitä käyttävät vastaavasti käytettävyysasiantuntijat. (Morgan ym. 2015, luku 6.)

Kuudes ja viimeinen kategoria on muut työkalut ja siihen kuuluu kaikki muut työkalut, joita voidaan testauksessa hyödyntää. Esimerkiksi laskentataulukot, tekstinkäsittelyohjelma, sähköposti, varmuuskopiointi ja palauttamisen apuohjelmat, SQL ja muut tietokannan kyselytyökalut, projektin suunnittelutyökalut, virheenjäljitystyökalut (debugging tools). (Morgan ym. 2015, luku 6.)

Työkalut ovat olennaisia ketterän kehityksen projekteille. Ketterän kehityksen projektit ovat vaativia, sillä näissä on usein tapahtuva tuotteen toimitus, projekti voi muuttua paljon ja asiakkaat edellyttävät korkealaatuisia tuotteita. Näissä tärkeitä työkaluja esimerkiksi regressiotestauksen näkökannalta ovat testien suorituksen työkalut ja integraatiotyökalut. Jotkin ketterän kehityksen mallit myös edellyttävät työkaluja toimiakseen tehokkaasti. (Coleman ym. 2017, 154.)

Coleman ym. (2017, 156-170) jakavat työkalut ketterän kehityksen näkökulmasta seuraaviin:

- tehtävien hallinnan ja jäljityksen työkalut
- viestintä ja tiedon jakamisen työkalut
- ohjelmiston kehityksen ja jakelun työkalut
- kokoonpanon hallinnan työkalut
- testien suunnittelun, teon ja suorittamisen työkalut
- pilvi- ja virtualisointityökalut

Yleensä kun testaaja puhuu testityökalusta, on kyseessä työkalu, jolla automatisoidaan testien ajoa. Nämä vaihtelevat yksikkötestaustason työkaluista, toiminnallisuuksien varmistamisen tasolle ketterissä projekteissa sekä järjestelmätestauksen tasolla. Mikä tahansa testien suorituksen työkalu tekee vähintään kaksi asiaa eli lähettää syötteitä ja muita tapahtumia testattavalle asialle sekä taltioi asian vastaukset, myös mahdolliset kuvakaappaukset, vastinajat tai resurssien käytön. Monessa tapauksessa työkalu myös vertailee tulosta odotettuun tulokseen ja raportoi vertailun. (Black 2016, luku 7.)

4 Testausautomatisaatio

Dustin ym. (2009, 4) määrittävät automatisoidun ohjelmistotestauksen: ohjelmistotekniikan käyttäminen kautta ohjelmistotestauksen elinkaaren, jonka tarkoitus on parantaa testauksen tehokkuutta. Testiautomatisoinnin päätarkoitus on suunnitella ja toteuttaa automatisoitu testaus ja uudelleentestaus valmiudet, joilla pystytään tehostamaan testausta. Joka onnistuessaan vähentää tarvittavia kustannuksia, aikaa ja resursseja. Vaikka kaikki testit voidaan periaatteessa automatisoida, niin kannattaa laskea missä määrin on kustannustehokasta automatisoida. (Dustin ym. 2009, 4.)

Testausautomaatio on testaustapa, jossa rakennetaan automaattityövälineitä testien suorittamiseksi. Tavoitteena on valita testit, joita tehdään usein, ja tehdä niiden ajamiseen nopeasti väline. Näin vapautetaan testaajia muihin töihin eikä heidän tarvitse testata joka käännöksen jälkeen perustoimintoja. Automattitestit voivat myös esimerkiksi yön aikana käydä läpi tarkastuksia, joiden tulokset voi aamulla käydä läpi. Kasurinen nostaa automaatiolle otollisimmiksi kohteiksi moduulien rajapinnat sekä yksikkötestauksen yksittäisten moduulien tarkastukset. Myös yleiseksi automatisointikohteeksi nostetaan käyttöliittymän tarkistukset. Esimerkiksi toimintojennauhoitusohjelmilla voidaan luoda toimenpidesarja, joka voidaan ajaa läpi yhä uudestaan ja varmistaa että lopputulos vastaa haluttua. Usein kuvitellaan testausautomaation korvaavan käsin tehtävän testauksen, vaikka automatisointi vain täydentää käsin testausta. Testausautomaation käyttö vie myös resursseja ja ylläpitoa eikä automaation avulla usein säästetä sellaisia summia mitä kuvitellaan. Joissain tapauksissa testausautomaation käyttäminen ei aina ole kannattavaa. (Kasurinen 2013, 76-79.)

Automatisointi tehokkaasti toteutettuna ei ole pelkästään nauhoita ja toista -metodia, vaan sen pitäisi tukea eri ohjelmointikielillä toteutettuja sovelluksia, voidaan ajaa useilla tietokoneilla, käyttävät eri käyttöjärjestelmiä tai ovat eri alustoilla, on graafinen käyttöliittymä tai vastaavasti ei ole graafista käyttöliittymää sekä käyttävät eri internet protokollia kuten TCP/IP, DDS jne. (Dustin ym. 2009, 5.)

4.1 Miksi automatisoida

Ohjelmistokehitys on monimutkainen aktiviteetti, sillä sitä kehittää ihmiset ihmisille. Ohjelmistotestaus on jatkuva koneisto laadun parantamiseksi vaiheesta toiseen kautta ohjelmistotuotannon. Siksi testausta tarvitaan jokaiseen toimitettavaan tuotteeseen. Toimittavan tuotteen laadussa on jokaisessa vaiheessa epävarmuuksia. Nykyiset ohjelmistot ovat monimutkaisia. Ne ovat yleensä yhdistelmä kerroksia, osia, kehyksiä, joita on kehitetty eri

organisaatioissa. Järjestelmien odotetaan toimivan saumattomasti erilaisilla resursseilla kuten kaistanleveyksillä, tehoilla, prosessointitehoilla jne. (Moiz 2017, 67-68.)

Testiautomatisoinnin tarkoitus on tehostaa testausta ja vähentää testaukseen tarvittavaa työmäärää. Jos kehitetään ohjelmistoa testaamaan ohjelmistoa, niin sen pitää olla mahdollisimmin virtaviivainen. Tarkoituksena on vähentää tarvittavaa ja virhealtista ihmisvuorovaikutusta ja automatisoida testiautomatisointia. Esimerkiksi yritykset käyttävät usein aikaa kehittääkseen testauskehystä alusta alkaen, vaikka kaikki tarpeet täyttäviä valmiita kehyksiä löytyy valmiina: vapaasti saatavilla olevien avoimen lähdekoodin ratkaisuja, joissa on tarvittavat ominaisuudet uudelleenkäyttöön ja helposti integroitavissa lisäominaisuuksia, joilla säästää aikaa ja rahaa. Dustin ym. suosittelevat vahvasti testiautomatisoinnissa harkitsemaan avoimen lähdekoodin komponentteja suunnitellessa ja kehittäessä testiautomatisoinnin kehyksiä (frameworks). (Dustin ym. 2009, 8-9.)

Toistuvasti suoritettavaa käsin testausta edullisemmaksi voi tulla testausautomaation ylläpito. Automatisointia kannattaa harkita silloin, kun testiä toistetaan 4-20 kertaa projektin aikana. Yksikkötestaus, integraatiotestaus ja savutestit ovat testausautomaatiolle otollisia, sillä testejä voidaan käyttää aina uudestaan regressiotestauksen yhteydessä. Automaation tarkoitus on varmistaa, että jo toimineet osat toimivat edelleen, eikä löytää uusia virheitä. Automaatiotestausta kannattaa myös soveltaa kuormitustestauksessa. (Kasurinen 2013, 76.)

Automatisoidun testin tekeminen vie enemmän aikaa ja maksaa enemmän kuin ´testin tekeminen käsin, mutta automatisoidun testin uudelleen ajaminen on nopeampaa ja halvempaa kuin uudelleen testaaminen käsin. Automatisointiin kannattaa harkita niitä testejä, joita voidaan toistaa monesti ilman muutoksia tai korkeintaan pienillä muutoksilla. Testitapauksien kriittisyyden määrittely auttaa nostamaan automatisoitavien testien prioriteetin paremmin esiin. Ja automatisointiin pitää varata aikaa ja resursseja ylläpitoa varten projektin ulkopuolelta. (Kasurinen 2013, 76-79.)

Automatisoinnin käyttämisen syitä on myös manuaalitestauksen hitaus ja sen virheenmahdollisuus, sillä usein käsin ajettavia testejä toistettaessa virheen tekemisen mahdollisuus lisääntyy. Automatisointi myös vapauttaa ihmisiä muihin töihin ja automatisoiduista regressiotesteistä saa hyvän turvaverkon. Automatisoiduista testeistä saa nopeasti sekä usein palautetta ja testit toimivat ketterässä kehityksessä osana dokumentaatiota. Lisäksi automatisointi voi tuottaa hyvää vastinetta investoinnille. (Crispin & Gregory 2008, luku 13.)

Fewster & Graham huomauttavat automatisoinnin tärkeyden nousevan ketterän kehityksen maailmassa. Ketterän kehityksen jatkuva integraatio tarkoittaa testiautomatisointia, sillä regressiotestejä ajetaan päivittäin jollei useamminkin. Testiautomatisoinnin pitää pystyä myös muuttumaan samoin kuin ketterän kehityksen, joten testiarkkitehtuuri on kriittisempi. Testiautomatisointi on perinteisessä ja ketterässä kehityksessä menestyvää, mutta ketterä kehitys ei voi onnistua ilman automatisointia. (Fewster & Graham 2012, Reflections on the Case Studies.)

Black kertoo suurimmalle osalle automatisoinnin tarkoitusperän olevan säästää rahaa. Säästö tulee testien suorituksen pienemmästä kustannuksesta, minkä pitää olla tarpeeksi merkittävä maksaakseen takaisin työkalun hankinnan, testien kehityksen ja ylläpidon. Ylläpidon kustannukset ovat usein automatisointiyritysten kaatumisen takana. Osaavien teknisten testianalyytikoiden pitäisi osata rakentaa ylläpidettäviä testikehyksiä ongelmien välttämiseksi. (Black 2016, luku 7.)

Testiautomatisointi on sovellettavissa ja hyödyllistä myös julkaisuvalmistelussa tai tuotannossa. Esimerkiksi tuotantojulkaisujen hyväksyntätestauksen osan tai kokonaan automatisoinnissa säästetään testausaikaa ja kustannuksia. Testiautomatisointi auttaa testaamaan nopeasti tuotannon korjauksia, auttaa etsimään tuotannon ongelmakohtia, tukee korjauksen testausta sekä tuotannon järjestelmien ongelmien raportointia. Automatisointi tukee myös arvioimaan, kuinka hyvin noudatetaan standardeja kuten IEEE, OMG, ISO. (Dustin ym. 2009, 16-18.)

4.2 Milloin ei kannata automatisoida

Aina automatisointi ei tuo hyötyä. Lewis, Dobbs & Veerapillai (2017, luku 35) nostavat kolme syytä automatisoinnin käyttämättä ottamiseen: ensimmäinen on automatisoinnissa tarvittava syvä oppimiskäyrä, toinen automatisoinnin vaatima kehitystyö ja kolmas ylläpidon kustannukset.

Oppimiskäyrä voi olla ongelma siitä syystä, että perinteiset testien automatisointityökalut ovat pohjimmiltaan erikoistuneita ohjelmointikieliä, mutta parhaat testaajat ovat sovellusasiantuntijoita eivätkä ohjelmoijia. Tämän takia testiautomatisoinnin opettelu voi viedä kuukausia, jollei vuosia. Ohjelmistotoimittajat tietävät tämän ja pyrkivät ratkaisemaan ongelman ”nauhoita ja toista” -ohjelmilla, joilla voidaan nauhoittaa testejä ja toistaa niitä myöhemmin. Nauhoitetut testit kuitenkin epäonnistuvat helposti, sillä näissä ei ole virheenhallintaa tai logiikkaa ja siten pienikin muutos voi aiheuttaa epäonnistumisia. Lisäksi nauhoita/tallennus metodi yhdistää testiskriptin ja datan, mikä ei ole uudelleen käytettävää

eikä modulaarista. Näiden ylläpito ei ole helppoa ja koska testausta tehdään uudestaan ohjelman muututtua, automatisointi pitäisi myös muuttaa. (Lewis ym. 2017, luku 35.)

Muita syitä automatisoinnin epäonnistumiseen on esimerkiksi epärealistiset odotukset: IT-ala on surullisenkuuluisa siitä, että siinä tartutaan uusiin teknologioihin ajatellen sen korjaavan kaiken. Ihmisluontoon kuuluu optimismi uuteen teknologiaan. Toimittajien myyjät antavat ruusuisen kuvan heidän työkalutarjonnastaan. Lopputulos on odotukset, jotka eivät usein vastaa todellisuutta. Myös testausprosessin puute on syy olla ottamatta testiautomatisointia käyttöön, sillä edellytyksenä on järkevä manuaalisen testauksen prosessi. Hyvien testauskäytäntöjen ja standardien puute on haitallinen testiautomaatiolle. Automatisoidut testaustyökalut eivät automaattisesti löydä vikoja, jollei testitapaukset ole selvästi määriteltyjä. (Lewis ym. 2017, luku 35.)

Automatisointi ei sovellu käytettävyystestaukseen, kertaluontoiseen testaukseen eikä suunnitelmattomaan tutkivaan testaukseen. Ja jos järjestelmä muuttuu nopeasti joka testikierrokselle, vietetään enemmän aikaa regressiotestaustyökalun ylläpitoon kuin mitä se on väärti. (Lewis ym. 2017, luku 35.)

Automatisointi voi luoda myös väärää turvallisuuden tunnetta. Vaikka automatisoidut testit on onnistuneesti suoritettu, se ei takaa sitä, että kaikki virheet olisi löydetty. Tämä oletus luo väärää turvallisuuden tunnetta. Automatisaatio on yhtä hyvä kuin testitapaukset ja testille annettu aineisto. Lisäksi automatisointityökalut voivat sisältää myös virheitä. (Lewis ym. 2017, luku 35.)

Aina testityökalu ei ole kustannustehokas ratkaisu organisaatiolle, esimerkiksi kun tarkastellaan kustannuksia vasten saatavia hyötyjä. Lisäksi yrityksen kehityskulttuuri ei välttämättä ole valmis testaustyökalun käyttöön, sillä voi puuttua tarvittavat taidot ja sitoutuminen pitkäjänteiseen laatuun. Jos testaus pitää tehdä tietyssä aikajaksossa, niin testaustyökalu ei välttämättä ole käyttökelpoinen. Automatisointi vie aikaa oppia, ottaa käyttöön ja sisällyttää kehittämismenetelmiin. (Lewis ym. 2017, luku 35.)

Fewster & Graham (2012, Reflections on the Case Studies) nostavat automatisoinnin onnistumisessa esiin sen, että automatisoinnilla on tarpeen olla selkeät tavoitteet. Pitää myös eriyttää testauksen ja testiautomatisoinnin tavoitteet, sillä testiautomatisoinnin kohteena voi olla regressiotestaus, joka yleensä varmistaa, ettei mitään ole mennyt rikki. Näin voidaan mitata, onko automatisointi täyttänyt tavoitteensa eikä ole riskiä budjetin katoamisesta siitä syystä. (Fewster & Graham 2012, Reflections on the Case Studies.)

Crispin & Gregory (2008, luku 13) nostavat esiin automatisoinnin käyttöönottamatta jättämisen syiksi seuraavia: kehittäjien asenne, syvä oppimiskäyrä, alun investointitarve, jatkuvassa muutoksessa elävä koodi, vanhat järjestelmät, pelko ja vanhat tavat. Perinteiseen kehitysmalliin tottuneilla kehittäjille testaus voi olla etäinen asia eivätkä siten näe tarvetta automatisoinnille, vaikka käyttäisivät automatisointia yksikkötestaustasolla. Muuttuvat koodi estää myös testiautomatisointia, sillä se vaatisi jatkuvaa ylläpitoa. Testien automatisointi käyttöliittymän kautta on yleensä riskialtista, sillä käyttöliittymä muuttuu usein kehityksen aikana, minkä takia yksinkertainen nauhoitus ja toisto tekniikka on harvoin hyvä valinta ketterään projektiin. Jos myös tausta elää kehityksen aikana, niin tämä voi tarkoittaa automatisoinnin olevan kallista ja aikaa vievää myös esimerkiksi rajapintojen kautta. Tässä voi auttaa testikoodin rakentaminen sovelluksen tarkoitukseen soveltuvaksi eikä vain toteutukseen nojaten. Vanhat järjestelmät ja aiemmin tehty koodi tuottavat ongelmia, sillä on helpompi saada automatisointi toimimaan uutta koodia testatessa, jonka arkkitehtuurissa on huomioitu testaustarve. Aiemmin kirjoitettuun koodiin on vaikeampi tehdä automatisointia. Testiautomatisointi vaikuttaa pelottavalta ja sen käsitetään vaativan testaajilta ohjelmointitaitoja, minkä vuoksi automatisointia voidaan vastustaa. Lisäksi vanhat tutut tavat ovat syy pysyä vanhassa mallissa, vaikka uusi olisi tarpeen. (Crispin & Gregory 2008, luku 13.)

4.3 Automatisoinnin käyttöönotossa huomioitavaa

Kaikkea ei pysty eikä kannata automatisoida. Dustin ym. (2009, 27) arvioivat omaan kokemukseensa perustuen, että noin 40-60% testeistä voidaan ja pitäisi automatisoida. Jokainen projekti pitää kuitenkin arvioida erikseen. Ennen testien automatisointia kannattaa arvioida mitä automatisointi voi tuottaa takaisin tuottona. Automatisoinnin vaikutuksen arvioinnissa tärkeää on tarkastella mitä alueita ja kuinka paljon testauksesta voidaan automatisoida, mikä on odotettu vähennys testiaikaan ja aikatauluun, mikä on odotettu lisäys testikattavuuteen ja laatuun sekä mitä muita vaikuttavia asioita löytyy. (Dustin ym. 2009, 27.)

Fewster & Graham (2015, Reflections on the Case Studies) nostavat esiin kaksi tekijää onnistuneeseen testiautomatisointiin, joista ensimmäinen on johdon tuki automatisaatiolle, kuten realististen tavoitteiden asetanta sekä riittävien ja soveltuvien resurssien käyttöön saaminen tavoittaakseen sijoituksen tavoitellut tuoton (ROI eli Return on Investment). Toinen on hyvä tekninen arkkitehtuuri automaatiolle, eli oikea abstraktion taso, jolla voidaan saada joustavuutta ja muokkautuvaisuutta pitäen kustannukset hallinnassa kaikille automaation osille. (Fewster & Graham 2015, Reflections on the Case Studies.)

Hallinnollinen tuki on tärkeää, sillä yksilö voi aloittaa kokeilun itse, mutta automaatiota kun halutaan laajentaa, tarvitaan parhaan hyödyn saamiseksi tukea organisaation hallinnolta. Hallinnollinen tuki voi olla rahoitusta, ajan antamista pilottiprojektiin ja testipuolen arkkitehtuurin kehittämiseen, kiinnostusta automaation kuulumisiin sekä kustannusten ja hyötyjen seuraamista. Johdon on myös tärkeä tiedostaa mihin automaatiolla kyetään ja mitä tulosten saavuttamiseksi tarvitaan. Automatisoinnin kehittäjien ja ylläpitäjien täytyy kommunikoida selkeästi ja usein tavoilla, jota johtajat ymmärtävät. (Fewster & Graham 2012, Reflections on the Case Studies.)

Testiarkkitehtuuri on tärkeää, sillä testaajien odotetaan käyttävän testityökalua, mikä voi aiheuttaa ongelmia silloin kun testaajat eivät ole kehittäjiä. Testien suorittamisen automatisointiin on (ainakin) kaksi ison tason abstraktion tasoa: erotetaan työkalu ja työkalujohdannainen tieto rakenteellisesta testiympäristöstä ja erotetaan testi rakenteellisesta testipuolesta. Testien automatisoija on vastuussa näiden erottamisesta ja jos tätä ei toteuteta hyvin, on automatisointi kallista ylläpitää ja vaikea käyttää. (Fewster & Graham 2012, Reflections on the Case Studies.)

Automatisoinnista saa eniten irti silloin, kun se otetaan mukaan kehityksen alkupuolella, esimerkiksi yksikkötestauksesta lähtien, toisin kuin vain loppupuolen hyväksymistestauksessa. Testejä pystyy hyödyntämään pitkin elinkaarta ja mitä aiemmin virhe huomataan, sen edullisempi se on korjata. Aikataulu vaikuttaa automatisoinnin laajuuteen, mitä enemmän aikaa on käytössä, sen kattavampi automatisointi voidaan tehdä. Aikataulu vaikuttaa myös testien kattavuuteen ja analyysien yksityiskohtaisuuteen jne. Tehokas automatisointi sopeutuu toteutuksessa testattavan kohteen tekniikan vaatimuksiin. Automatisoinnin työkaluja voidaan käyttää esimerkiksi Windowsilla tai Linuxilla, reaaliaikaisessa tai staattisessa ympäristössä jne. Mitä monimutkaisempi testattava kohde on, sen vaativampi on automatisoinnin kehitys. Testattavan kohteen kriittisyys ja riskien määrä vaikuttaa myös liittyviin automatisointitarpeisiin. Tavoite on saada automatisoitua erilaisia ohjelmia ilman, että jokaista varten joudutaan kehittämään uusi testauskehys sekä käytetään prosesseja, jotka ovat helposti otettavissa käyttöön tai sopeutettavissa eri organisaatioon tai ohjelmaan. (Dustin ym. 2009, 6-8.)

Testiautomatisointi lisää monimutkaisuutta testauksen alkuun, sillä ennen kuin päätetään ottaa automatisointitestityökalu käyttöön, pitää harkita useita tekijöitä. Automatisointitarpeet kannattaa käydä läpi ja pohtia onko testityökalu soveltuva. Joskus markkinoilta ei löydy automaatiotarpeita täyttävää työkalua ja testausohjelmisto ja -kehys pitää tehdä talossa. Lisäksi kannattaa tarkastella automatisointitestausta tukevan aineiston saatavuutta. Aineiston laatu ja variaatiot kannattaa hahmotella ja suunnitella kuin aineisto saadaan tai

miten sen voi kehittää. Kannattaa panostaa testiskriptien modulaarisuuteen ja uudelleenkäytettävyyteen. Automatisoitu testaus on omanlaisensa kehitysprojekti ja sillä on oma mini kehityssyklinsä. Testisuunnittelussa tulee huomioida myös testauksen kehityksen elinkaaren tukemista, joka on käynnissä yhtä aikaa ohjelmiston kehityksen kanssa. Menneisyydessä testien kehittäminen oli hidasta, kallista ja työtä vievää. Käyttämällä testien hallinnoinnin ja automatisoinnin työkaluja, vie testiprosessien luonti ja muokkaus vain murto-osa entisestä. (Dustin ym. 2009, 28.)

Dustin ym. (2009, 91) listaa viisi pääasiaa ottaessa automatisointia käyttöön:

- tunne tarpeesi
- kehitä automatisointitestauksen strategia
- testaa automatisointiohjelmiston testikehys
- seuraa edistystä jatkuvasti ja säädä tarvittaessa
- toteuta automatisoinnin prosessit
- laita oikeat ihmiset oikeaan projektiin – tiedä tarvittavat taidot.

Automatisoinnin toteutus vaatii oman kehityksen elinkaarin, missä on testivaatimukset, automatisoinnin kehyksen design ja rakennus, automatisoidun testin implementointi ja varmistus sekä suoritus. Jos automatisointi tehdään kunnolla, se tuottaa uudelleen käytettävissä olevia ja laajennettavia kehyksiä ja komponentteja. Tässäkin ohjelmistokehityksen parhaat käytännöt pätevät. (Dustin ym. 2009, 6.)

Testausautomatisaatiossa noudatettavia parhaita käytäntöjä on esimerkiksi ”älä toista itseäsi” eli vältetään asioiden tekemistä useampaan kertaan. Siten kun testattavassa järjestelmässä muuttuu jokin, päivitys riittää yhteen osioon. Testillä on hyvä olla vain yksi tarkoitus, mikä tekee siitä kevyemmän virheenjäljityksen näkökulmasta sekä helpomman muuttaa, jos liiketoimintasääntö muuttuu. Testin ymmärtämisen kannalta on hyvä erottaa koodi testistä. Testeillä on hyvä tehdä valmistelut ja lopetukset, jotta testejä voi ajaa toistuvasti. Testien on hyvä olla itsenäisesti ajettavissa eli ei tarvitse ajaa testejä tietyssä järjestyksessä niiden toimimiseksi oikein. Kannattaa käyttää yleisiä testausstandardeja (myös nimeämisessä), mikä mahdollistaa testien ymmärtämisen yleisimmin sekä jaetun koodin /testin omistajuuden. Voi myös hyödyntää yksikkötestauksen AAA-mallia (Arrange-Act-Assert pattern) eli testiin rakennetaan kolme vaihetta: valmistelu, suoritus ja varmennus. Esimerkiksi valmisteluvaiheessa luodaan objekti ja asetetaan sille arvot, suoritetaan testi jollakin metodilla ja varmennetaan että odotetut tulokset toteutuivat. (Crispin & Gregory 2015, 240-241.)

Parhaan työkalun löytäminen organisaatioon vaatii tarkan ymmärryksen ongelmasta, jota ratkotaan, sekä riippuu tarpeesta ja sen vaatimuksista. Kun ratkaistava ongelma on selvillä, on mahdollista aloittaa työkalujen arviointi. Löytyy kaupallisia ja avoimen lähdekoodin

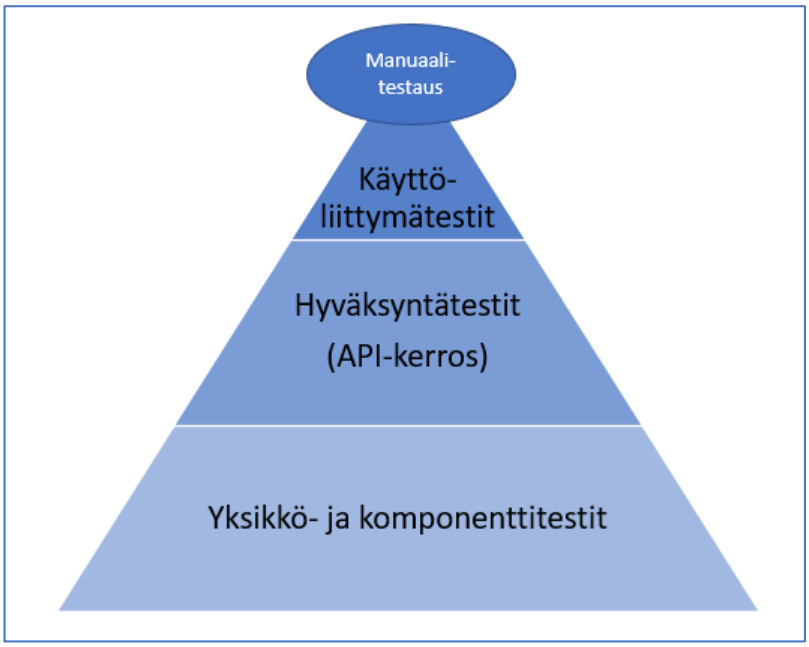
ratkaisuja, tai voi rakentaa oman. Kaupallisissa ratkaisuissa on etunsa, jotka sisältävät julkaistut ominaisuuksien kehityspolut, tuen ja vakiintuneisuuden (todellisen tai kuvitellun). Ohjelmiston toimittajalta ostamisessa on omat huonot puolensa, kuten toimittajaan lukittuminen, toimimattomuus toisten tuotteiden kanssa, huono vaikutusmahdollisuus kehityspolkuun sekä lisenssimaksut ja -rajoitteet. Näistä osa voi päteä myös avoimen lähdekoodin projekteihin, mutta yhä useampi yritys siirtyy siihen suuntaan. Avoimenlähdekoodin työkalujen hyviä puolia ovat: ei lisenssimaksuja, -ylläpitoa tai -rajoituksia, ilmainen ja tehokas vaikkavin vaihteleva tuki, toimivuus eri alustojen yli, muokattavissa tarpeisiin, ratkaisut suhteellisen kevyitä, eivätkä nämä ole sidottu toimittajaan. Miettien nykyistä laajaa avoimen lähdekoodin toimintaa ja valikoimaa, ei kannata rakentaa koodia, joka on jo olemassa ja on jo kenttätestattu. (Dustin ym. 2009, 91-92.)

Usein ymmärretään väärin automatisoinnin kulut, mitkä käsitetään työkalun hankintakuluiksi eli avoimen lähdekoodin työkalun osalta kuluja ei ajatella syntyvät. Kuluja syntyy, sillä on tarpeen panostaa kehitykseen ja ideointiin, hyvän testiarkkitehtuurin suunnitteluun ja kehittämiseen jne. Sijoituksen tuottoaste tehdään usein automatisointiprojektin aluksi. Tässä on huomioitava testien teon ja ajamisen lisäksi niiden ylläpito, testien analysointi ja parannus sekä muu ajankäyttö, mikä on pois manuaalisesta testauksesta. Muita huomioitavia asioita on testien suurempi kattavuus, lyhyempi aika julkaisuun ja parempi luotettavuus. Nämä eivät usein esiinnyt heti alussa ja näitä on vaikeahko määrittää. Kun automatisointi on vakiintunut, on tärkeä tarkastella ovatko tavoitteet saavutettu, joten alun tarkastelua on hyvä tehdä säännöllisin väliajoin ja viestiä tästä rahoittavalle johdon tasolle. (Fewster & Graham 2012, Reflections on the Case Studies.)

4.4 Mitä kannattaa automatisoida

Yleisimmät testaustyypit, joita voidaan helpoiten automatisoida ovat yksikkötestaus, regressiotestaus, toiminnallinen testaus, turvallisuustestaus, suorituskykytestaus, kuormitustestaus, rinnakkaistestaus ja koodin kattavuuden testaus (Dustin ym. 2009, 15).

Automatisoinnin optimoinnissa auttaa ketterän testauksen automatisointipyramidi, joka kuvassa 13 (Crispin & Gregory 2008, luku 14).



Kuva 13. Ketterän testauksen automatisointipyramidi (mukaillen Crispin & Gregory 2008, luku 14)

Pyramidin perusta sisältää yksikkö- ja komponenttitestauksen ja se tukee muuta kokonaisuutta. Tässä kerroksessa on suurin osa automatisoiduista testeistä ja ne on toteutettu yleensä samalla ohjelmointikielellä kuin testattava järjestelmä. Näistä saa myös nopeimmin palautetta ja se tekee näistä arvokkaita, sillä ne tuottavat eniten arvoa sijoitukseen nähden. Tästä syytä ketterässä kehityksessä tähän kerrokseen pyritään tekemään eniten testejä. (Crispin & Gregory 2008, luku 14.)

Pyramidin keskimmäinen taso sisältää suurimman osan liiketoimintaan kohdistuvista automatisoitavista testeistä, jotka on tehty tukemaan kehitystiimiä. Nämä varmistamat, että kehitetään oikeaa asiaa. Nämä testit ovat yksikkötestausta laajempia, voivat sisältää tarina- ja hyväksyntätestejä ja pyörivät yleensä käyttöliittymän takana API-tasolla. Koska testit ohittavat käyttöliittymän, näitä on edullisempaa ja nopeampaa tehdä sekä ylläpitää. Näistä saa myös nopeammin palautetta kuin käyttöliittymän kautta testatessa. (Crispin & Gregory 2008, luku 14.)

Pyramidin ylin taso edustaa tasoa, johon pitäisi panostaa vähiten automatisointia. Nämä testit tehdään yleensä käyttöliittymän kautta ja yleensä silloin kun koodi on valmista. Tästä syytä nämä ovat yleensä tuotetta arvioivia testejä. Tämän tason testien ajaminen vie huomattavasti enemmän aikaa ja koska käyttöliittymä muuttuu kehityksen aikana, niin tämän tason automatisoitujen testien ylläpito on vaativampaa. Tämän tason testit tuottavat tärkeää palautetta, mutta näiden lukumäärä kannattaa pitää minimissä. (Crispin & Gregory 2008, luku 14.)

Pyramidin kärkeen on tuotu erikseen manuaalinen testaus edustamaan sitä, että manuaalista testausta aina, kuten tutkivaa testausta. Regressiotestauksesta suurimman osan kannattaa olla automatisoituna ketterässä kehityksessä tai manuaalitestauksesta ei saada suurinta hyötyä sijoitukseen nähden. (Crispin & Gregory 2008, luku 14.)

4.5 Paljonko automatisoidaan

Kasurinen (2013, 12) kertoo että noin 10% testeistä automatisoidun testaustyöstä sille tasolle, ettei siihen tarvita testaajaa. Testityö on monessa paikassa käsityötä, huolimatta sen kalleudesta. Jotkut yritykset ovat kuitenkin ilmoittaneet automatisoineensa testaustaan jopa 90%. (Kasurinen 2013, 12.)

ISTQB:n Worldwide Software Testing Practices Report 2017-18 -julkaisussa kerrotaan, että automatisointia hyödyntämättömiä yrityksiä on yhä paljon 21,2 prosenttia. Lähes puolet vastanneista kertoo automatisoineensa liki 20 prosenttia. Testiautomaatiota eniten tehdään autoteollisuudessa 59,2 prosentin kattavuudella ja tietoliikenne-, media- ja viihdeteollisuudessa 27,1 prosenttia. (ISTQB 2019.)

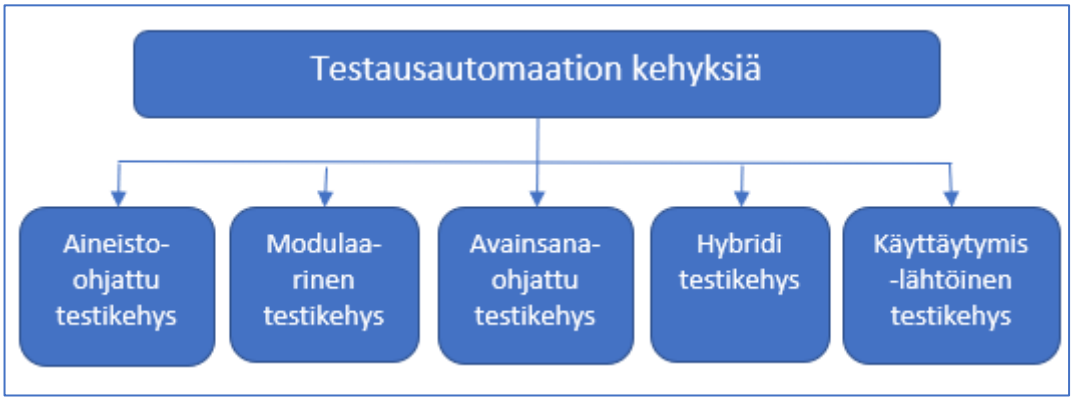
Capgemini ym. (2019, 10) kertoo, että vaikka suurempi määrä organisaatioista hyötyy testiautomaatiosta, sitä käytetään edelleen noin 16% testauksesta automatisoidaan.

Regressiotestejä automatisoidaan eniten 86%, jonka jälkeen on usein toistetut 46%, kuormitus (load) 29%, suorituskykytestit (performance) 29%, ristiin selaimia testaavat 29% (QASymphony & Techwell, 2019, 5).

4.6 Testausautomaation kehyksiä

Yksinkertaistaen kehys on ‘rakenteellinen’ yhdistelmä erilaisia ohjeita, ohjelmointistandardeja, käsitteitä, prosesseja, käytäntöjä, projekti hierarkioita, modulaarisuutta, raportointijärjestelmää, testiaineiston syötteitä jne., joista muodostuu testauksen perusta. Käyttäjä voi hyödyntää näitä automatisoidessaan. Kehyksien hyödyt voivat olla erilaisia kuten skriptien tekemisen helppous, skaalautuvuus, modulaarisuus, ymmärrettävyys, prosessin määrittely, uudelleen käytettävyys, kustannukset, ylläpito jne. Etujen hyödyntämiseksi kehittäjiä kehotetaan käyttämään yhtä tai useampaa testiautomaation kehystä. Etuna on myös yhtenäinen tapa automatisoida saman projektin eri kehittäjien kanssa. Testauskehys ei ole sovelluskohtainen eli sitä voidaan käyttää minkä tahansa sovelluksen kanssa huolimatta testattavan sovelluksen ominaisuuksista, kuten arkkitehtuurista. Kehyksen pitäisi olla skaalautuva ja ylläpidettävä. (Sofware Testing Help 2020.)

Lewis ym. (2017, luku 28) nostavat neljäksi yleisimmäksi tekniikaksi aineisto-ohjatun, modulaarisen, avainsanaohjatun ja hybridin testikehyksen. Rajkumar (2018) lisää tähän käytösvetoisen testikehyksen (behaviour based). Testikehykset esitelty kuvassa 14.



Kuva 14. Yleisimmät testikehykset (mukaillen Lewis ym. 2017 ja Rajkumar 2018)

Aineisto-ohjattu testikehys käyttää syötteitä ja tulosteita, jotka luetaan aineistotiedostoista, kuten esimerkiksi CVS-, Excel- tai tekstitiedostoista. Tässä lähestymistavassa on keskityttävä sekä testiskripteihin että testiaineistoon. Modulaarinen testikehys tarkoittaa pienien ja itsenäisten testiskriptien tekemistä, joista voidaan koota laajempia testattavan sovelluksen toimintoja ja osioita edustavia testejä. Avainsanaohjatussa testikehyksessä määritetään avainsana eri ruuduille, toiminnoille ja liiketoimintakomponenteille. Testaus tapahtuu näiden määritettyjen avainsanojen kautta. Testiaineisto ja -toiminta skriptataan testiautomatisointityökalulla. Yleisin käytössä oleva kehys on hybriditestikehys, joka on yhdistelmä edellisiä kehyksiä. Tässä yhdistetään niiden vahvuudet ja pyritään lievittämään heikkouksia. (Lewis ym. 2017, luku 28.)

Käyttäytymislähtöinen testikehys on luotu mahdollistamaan projektin kaikkien jäsenien aktiivinen osallistuminen, kuten liiketoiminta-analyytikkojen, kehittäjien ja testaajien. Kehyksen myötä ohjelmoinnin osaaminen ei ole välttämätöntä. (Rajkumar 2018.)

5 Robot Framework

Tässä työssä tehtiin automatisoitu testisetti, jota voidaan käyttää monipuolisesti ohjelmistokehityksen eri vaiheissa. Automatisointikeinoksi valittiin Robot Framework ja tehtiin testit verkkosivujen testaamiseen käyttöliittymän ja rajapinnan kautta. Robot Frameworkin käyttöönottoon on tehty asennusohjeet (liite 1) ja käyttöohje (liite 2), joissa katetaan tarvittavat asennukset sekä testien tekeminen, ajaminen ja päivittäminen.

Käyttöönotto perustui pääosin Robot Framework User Guide -käyttöohjeeseen (Robot Framework 2020) ja Automation step by step -sivuston Youtube-opetusvideoihin (Raghav 2020).

Ensin esitellään Robot Framework tarkemmin. Tämän jälkeen käydään läpi asentaminen, testien tekeminen sekä testien ajaminen ja raportointi.

5.1 Robot Framework esittely

Robot Framework on avoimen lähdekoodin automaatioviitekehys, jota voidaan käyttää testiautomatisointiin ja ohjelmistorobotiikkaan. Se on käyttöjärjestelmäriippumaton eikä sitä ole sidottu tiettyihin sovelluksiin vaan se on käytettävissä lähes minkä tahansa muun työkalun kanssa. (Robot Framework 2020.)

Robot Framework on avainsanaohjattu testikehys, jonka toiminnallisuuksia voidaan laajentaa Python- ja Java-kirjastoin. Robot Framework ydin on toteutettu Python-ohjelmointikielellä. Sen kanssa toimimaan on kehitetty paljon avoimen lähdekoodin kirjastoja ja työkaluja. Robot Frameworkin automatisointi- ja testikyvykkyydet tulevat kirjastojen tuottamien avainsanojen kautta. Kirjastoja voi myös tehdä itse. Robot Frameworkin arkkitehtuuri on modulaarinen, kuvassa 15 esiteltynä. (Robot Framework 2020.)

Kuva 15. Robot Frameworkin arkkitehtuuri (mukaillen Robot Framework 2020)

Kirjastoissa on määritetty avainsanoja ja näitä voi tehdä myös itse ylemmälle tasolle. Tekemällä testitapaus ylätason avainsanoilla ja määrittämällä näille toiminnallisuuden joukon kirjastojen avainsanoista, pysyy testitapaus selkeämpänä ja on helposti luettavissa. Robot Framework käsittää tekstin yhdeksi kokonaisuudeksi, kunnes tehdään väli tabulaattorilla. Esimerkiksi ”Open Browser” on yksi avainsana ja sille annetaan muuttujat tabulaattorille erotettuna. Kuva 16 on esimerkkikuva Robot Test suite -tiedostosta eli testisetistä. (Robot Framework 2020.)

```
*** Settings ***
Documentation    Simple example using SeleniumLibrary.
Library          SeleniumLibrary

*** Variables ***
${LOGIN URL}     http://localhost:7272
${BROWSER}       Chrome

*** Test Cases ***
Valid Login
    Open Browser To Login Page
    Input Username    demo
    Input Password    mode
    Submit Credentials
    Welcome Page Should Be Open
    [Teardown]    Close Browser

*** Keywords ***
Open Browser To Login Page
    Open Browser    ${LOGIN URL}    ${BROWSER}
    Title Should Be    Login Page

Input Username
    [Arguments]    ${username}
    Input Text    username_field    ${username}

Input Password
    [Arguments]    ${password}
    Input Text    password_field    ${password}

Submit Credentials
    Click Button    login_button

Welcome Page Should Be Open
    Title Should Be    Welcome Page
```

Kuva 16. Selaintestiesimerkki (mukaillen SeleniumLibrary 2020)

Robot Framework testisetti koostuu seuraavista osioista:

Settings
Variables
Test Cases tai ***Tasks***
Keywords
Comments

Settings-osiossa voidaan Documentation-kohtaan kirjoittaa testisetin metadataa, kuten kuvaus testistä. Tässä myös tuodaan käyttöön käytettävät kirjastot tai tiedostot. Variables-osiossa kirjoitetaan testien käyttämät muuttujat. Test Cases eli testit tai Tasks eli tehtävät ovat toisilleen vaihtoehtoisia osioita. Testeissä kirjoitetaan testin nimi ja sen alle testistepit avainsanojen kautta. Lisäksi voidaan kirjoittaa omia avainsanoja Keywords-osioon, jolloin testiin voidaan käyttää omia ylemmän tason avainsanoja testin vaiheina ja ne ovat tarkemmalla tasolla kuvattu Keywords-osiossa. Lisäksi testisettiä voidaan kommentoida Comments-osiossa. (Robot Framework 2020.)

Robot Framework luo testitapauksista oletuksena raportti- ja logitiedostot HTML-tiedostoina. Raportista käy selkeästi ilmi testistatistiikat, kuten onnistuneiden ja epäonnistuneiden testien suhteen sekä kauan testeissä on kestänyt. Log-tiedostoon muodostuu jokaisen ajetun testin testistepistä statistiikkaa avainsanakohtaisesti. Se mahdollistaa porautua testin yksityiskohtiin. (Robot Framework 2020.)

5.2 Asentaminen

Ennen asennusta kannattaa käydä läpi tarvittavat ohjelmat sekä niiden yhteensopivuudet. Tässä Robot Frameworkin kanssa on otettu käyttöön Robot Framework syntaksin ymmärtävä Eclipse-pohjainen RED-editori. Testit voidaan kirjoittaa myös tekstieditorilla kuten Notepad, mutta testien tekemistä helpottaa syntaksin tuntevan editorin käyttö. RED-editorin hyvänä puolena on myös virheenjäljitys (debugging) toiminto.

Asennettavat ohjelmat, kirjastot ja ajurit Windows 10 käyttöjärjestelmälle:

- Python 3.8.2
- Robot Framework 3.1.2
- SeleniumLibrary 4.3.0
- RESTinstance 1.0.2
- ChromeDriver 81.0.4044.69
- Java SE Development Kit 14 (2020-03-17)
- Eclipse IDE 2020-03 (4.15.0)
- RED 0.9.3

Asentamiseen on hyvät ohjeet Robot Frameworkin ja Eclipsen sivuilla. Näistä käy ilmi myös tarvittavat edellytykset ohjelmien pyörittämiseen, kuten Python ja Java tarpeet sekä ohjaukset näiden sivuille. Robot Frameworkin sivuilta löytyy lista kirjastoista ja näiden toiminnallisuuksista sekä asentamisesta.

Robot Frameworkin toimimiseksi on otettu käyttöön Pythonin uusin versio 3.8.2. Tämä ajettiin pääkäyttäjänä ja lisättiin Python PATH-ympäristömuuttujaan. Pythonin asennuksen yhteydessä asennettiin pip eli Pythonin paketinhallintajärjestelmä. Pip:n avulla asennettiin Robot Framework 3.1.2 ja SeleniumLibrary 4.3.0 komentorivin kautta komennolla ”pip install --upgrade robotframework-seleniumlibrary”. Samaan tapaan on asennettu API-kirjasto RESTinstance 1.0.2 komennolla ”pip install --upgrade RESTinstance”. Näissä tehdään samalla päivitys uusimpaan versioon. SeleniumLibrary soveltuu selaintestaukseen ja sen lisäksi tarvitaan selaimelle ajuri, esimerkiksi ChromeDriver on Chromen ajuri. Koska käytössä on Chromen versio 81, asennettiin yhteensopiva ChromeDriver 81.0.4044.69. Ajuri on lisätty PATH:iin, jotta Robot Framework pystyy käyttämään sitä.

Valittu editori RED on Eclipse-pohjainen ja sen käyttö edellyttää Java SE Development Kit (JDK) sekä Eclipsen asennusta. JDK:sta asennettiin uusin versio 13.0.2. Eclipsen uusin RED 0.9.3 kanssa yhteensopiva versio on Eclipse IDE 2020-03 (4.15.0). Eclipsen installer asentaa Eclipsen uusimman version, joten Eclipsen sivuilta haetaan halutun version tiedosto ja puretaan se haluttuun paikkaan. Eclipsen Marketplace-osion kautta lisättiin RED versio 0.9.3. Liitteessä 1 on kuvattu tarvittavat asennukset tarkemmalla tasolla.

Ohjelmien asennusohjeet ja edellytykset kannatti käydä katsomassa etukäteen ja tehdä valinta käytettävistä ohjelmista tarkistetun tiedon valossa. Esimerkiksi Firefoxin ajuri vaatisi lisäksi Microsoft Visual Studio redistributable runtime -ohjelmiston. Lisäksi vaatimukset kannatti tarkistaa sen takia, että kaikki versiot eivät toimi toistensa kanssa. Vaikka koneella oli suositeltu tai edellytetty ohjelma, kannatti varmistaa sen versio ja tarvittaessa päivittää se. Kannatti myös etukäteen päättää etukäteen, minne tekee asennukset, esimerkiksi työn tarkoituksiin toimi parhaiten työpöydälle Tools-kansio sekä Workspace-kansio Eclipsen työtiloille.

Liitteessä 1 on asennusohje, jossa on kuvattu näiden asentaminen yksityiskohtaisella tasolla sekä PATH-ympäristömuuttujan lisäys.

5.3 Testien tekeminen

Testien tekeminen tuli aloitettua harjoitusmielessä yksinkertaisesta selaintestistä, jossa pääsi kokeilemaan erilaisia toiminnallisuuksia. Robot Frameworkin käyttöönotto sinällään oli helppo ohjeiden perusteella, mutta ilman etukäteistuntemusta työkalusta ja testiautomatisoinnista, joutui työn käytännössä tekemään kahdesti. Kokonaisuus alkoi muodostua vasta tehdessä, kun työkalun mahdollisuudet aukesivat. Testailun jälkeen oli hyvä käydä vielä kerran läpi tarvittavat testit ja mikä hierarkia projektiin sopi.

Testien tekeminen on aloitettu luomalla ensin uusi Eclipsen Workspace eli työtila, jonne projekti ja testit tallentuvat. Käyttöön on otettu Eclipsen RED:n mukanaan tuoma Robot perspektiivi ja perustettu Robot-projekti. Projektin alle pystyy luomaan haluamansa kansiorakenteen testeille, raporteille ja logeille. Projektin luonnin jälkeen projektille on tehty Robot test suite -tiedosto (eli testisetti) ”.robot”-muotoisena.

Tiedostoa pystyy käsittelemään tarvittaessa esimerkiksi tekstieditoreilla kuten Notepad. Robot Frameworkin raportit, logit ja kuvakaappaukset tulevat oletuksena projektin kansioon, joten erityylisille testeille on järkevä tehdä erilliset projektit. Tähän työhön perustettiin lopulta selkeyden vuoksi kaksi Robot-projektia: ensimmäinen selaintesteille ja toinen rajapintatesteille.

Selaintestien teossa käytettiin SeleniumLibrary-kirjastoa ja sen avainsanoja. Kuvassa 17 on selaintestauksen opettelua varten tehty testisetti, jossa yksinkertainen selaintesti American Airlines -sivustolle.

```
*** Settings ***
Documentation     Chrome test example using SeleniumLibrary.
Library           SeleniumLibrary

*** Variables ***
${START URL}    https://www.americanairlines.fi/intl/fi/index.jsp
${BROWSER}    Chrome
${DELAY}    1
${FROM}    NYC
${TO}    ORD

*** Test Cases ***

TravelSearch
    Open Browser To Start Page
    Accept Cookies
    Input From    ${FROM}
    Input To    ${TO}
    Input Day From    15.6.2020
    Input To    22.6.2020
    Search
    Results Page Should Be Open
    [Teardown]    Close Browser

*** Keywords ***
Open Browser To Start Page
    Open Browser    ${START URL}    ${BROWSER}
    Set Selenium Speed    ${DELAY}
    Title Should Be    Airline Tickets and Airline Reservations from American Airlines | aa.com

Accept Cookies
    Click Button    optoutmulti_button
    Set Selenium Speed    ${DELAY}

Input From
    [Arguments]    ${FROM}
    Input Text    reservationFlightSearchForm.originAirport    ${FROM}

```

Kuva 17. Yksinkertainen selaintesti

Settings-osioon on kirjoitettu lyhyt kuvaus testisetistä. Tämä on hyödyllinen etenkin silloin, jos testisettejä on useampia samantyylisiä. Silloin hyvän nimeämisen lisäksi pystyy tämän kuvauksen perusteella erottamaan testisettejä. Settings-osiossa on myös tuotu käyttöön SeleniumLibrary-kirjasto.

Variables-osioon on tehty muuttujia, joita voidaan käyttää eri testeissä. Tässä on esimerkiksi käytettävä selain ja testin aloituksessa käytetty verkko-osoite laitettu muuttujiksi. Hyödylliseksi kävi myös nostaa selaimen viiveaika muuttujiin, sillä tästä sitä oli helppo muuttaa kaikkiin kohtiin. Viiveaikaa tarvittiin odottamaan selaimen latautumista, sillä testiajo kaatui siihen, että testissä yritettiin syöttää tietoja kenttiin, jota ei ollut vielä latautuneina. Muuttujaa tarvittiin eri paikoissa, joten viemällä tämän muuttujiin, pystyi viivettä pidentämään kaikkiin kohtiin kerralla.

Test Cases -osioon on tehty esimerkkiin testi käyttäen American Airlines -sivustoa. Tässä testi on tehty omilla ylätason avainsanoilla, jotka on tehty Keywords-osioon SeleniumLibrary-kirjaston avainsanoja käyttäen. Näin testin sai pidettyä ymmärrettävämpänä. Testin stepiksi riitti näin esimerkiksi avainsana ”Open Browser To Start Page” ja sen toimin-

nallisuudet sai tehtyä SeleniumLibrary-kirjaston avainsanoilla Keywords-osiossa. Selaimen avaaminen myös toistui usein testeissä, joten oli järkevää tehdä tämä kerran avainsanoihin ja pystyi näin käyttämään tätä pienemmällä vaivalla useampiin testeihin. Lisäksi jos verkkosivun otsikko vaihtuu, riittää päivitys yhteen kohtaan. Testin loppuun osoittautui hyödylliseksi asettaa aina testin lopussa suoritettavaksi stepiksi selaimen sulkeminen [Teardown] -määritteellä. Muuten testin epäonnistuessa jäi selain auki. Kannatti myös tehdä testiin varmistuksia testien edetessä, esimerkiksi vastaako siirryttäessä hakutuloksiin sivun otsikko haluttua vai onko päädytty virhesivulle. Jos otsikko ei vastannut määritettyä, steppi epäonnistui ajettaessa ja näki selkeästi missä vaiheessa testi epäonnistuu. Testin teossa kannatti käydä läpi Crispin & Gregory (2015, 240-241) mainitsemia testausautomaation hyviä käytäntöjä, joista edellisten esimerkkien periaatteet poimittu.

Yhteen tiedostoon eli testisettiin saa tehtyä yhdet asetukset, muuttujat, avainsanoista koostuvat testitapaukset ja omat avainsanat. Omia avainsanoja voi myös tehdä erilliseen tiedostoon, jonka tuo testisetin käyttöön. Tässä ei tätä tehty, mutta testimäärän lisääntyessä tulee todennäköisesti lisääntymään testisettien määrä ja silloin erillinen omien avainsanojen tiedosto voi olla järkevä.

Selaintestejä oli suhteellisen helppo tehdä, sillä SeleniumLibrary-kirjaston avainsanat on kuvattu hyvin ja tässä rakennetaan samaa loogista polkua kuin käyttäjä tekisi. Etsitään siten selaimesta syöttökenttien ja painikkeiden id:t tai nimet esimerkiksi selaimen Developer tools -kautta, niin saadaan vastaavasti vietyä syöte sivulle ja siirryttyä eteenpäin. Selaimen kautta testaaminen ei ole aina tehokkain tapa testata asioita, vaan suositeltavaa on ohittaa se ja käyttää esimerkiksi rajapintoja testaukseen aina kun mahdollista (Crispin & Gregory 2008, luku 14).

Tässä rajapintatesteissä käytettiin RESTinstance-kirjastoa ja sen avainsanoja. Alla on kuvassa 18 yksinkertainen rajapintatesti.

```
1  *** Settings ***
2  Library       REST
3
4  *** Test cases ***
5  GetPopulation
6    GET      http://pxnet2.stat.fi/PXWeb/api/v1/en/StatFin?query=population
7    Output    response body   file_path=${CURDIR}/population1.json
8
```

Kuva 18. Yksinkertainen rajapintatesti

Rajapintatestauksessa testin rakentaminen vei enemmän aikaa ja vaati hyvää rajapintakuvausta, jotta testeistä tuli mielekkäitä. Hyvä lähtökohta oli käyttää avainsanaa GET, kuten

kuvassa 18, jossa haetaan avoimen rajapinnan kautta tietoa annetulla hakuehdolla Stat-Fin-tietokannasta. Tieto otetaan vastaan ja viedään json-tiedostona testikansioon. Rajapintatesteissä voidaan myös käyttää muuttujia, rajata testissä hakutuloksia sekä tehdä varmistuksia saatavasta datasta. Testeissä kannatti myös yrittää lisätä varmistuksia tehtävien asioiden väliin, esimerkiksi varmistaa, että vastaus lähetettyyn sanomaan vastasi odotettua. Näin oli helpompi yrittää jäljittää virhettä.

Testien tekemisessä oli hyvä ottaa huomioon teoriasta automatisoinnin hyvät opit ja Robot Frameworkin käyttöohjeen neuvot. Molemmissa esimerkiksi kehotetaan välttämään päällekkäisyyksiä. Pääasia on saada mahdollisesti päivittyvät ja muuttuvat asiat kirjoitettua vain kerran, jolloin päivitystarve muutoksen tullessa on vain yhteen kohtaan. Eli kannattaa tehdä pieniä kokonaisuuksia, joista voi yhdistellä testejä. Myös testiaineiston päivittäminen olisi hyvä saada mahdollisimman helpoksi päivittää. Voi olla vaikea arvioida etukäteen mitä kaikkea tulee projektin aikana muuttumaan. Testien määrän ollessa pieni, voi päivitystarpeen tullessa eteen arvioida tuleeko samankaltaisia enemmänkin ja päivittää testin rakennetta sen mukaan. Testimäärän laajentuessa tai ollessa jo alusta laaja kannattaa olla selkeä linja, minkä mukaan toimii. Itselleni mahdollisesti päivittyvät asiat kannatti viedä muuttujiin, sillä testistepeistä ja tällä rakenteella myös omista avainsanoista ei aina löytynyt tätä helposti.

Molemmissa testityypeissä on omat haasteensa testejä tehdessä. Selaimen kautta testatessa osoittautui vaikeimmaksi käyttää testiaineistoa tehokkaasti. Nyt testeissä on käytetty muuttujia suhteellisen paljon, mutta tulevaisuudessa on todennäköisesti tuotava testiaineistoa erillisen tiedoston kautta. Toistaiseksi kuitenkin Robot Frameworkin Template-toiminnolla on saanut taulukkomaisesti aineistoa käyttöön ja se toimii riittävän hyvin. Rajapintatestien tarvittavat toiminnallisuudet ovat olleet suhteellisen yksinkertaisia, mutta testeihin tarvitaan testattavan ohjelman hyvä rajapintakuvaus ja paljon kokeilua. RED:in debugging-työkalu on osoittautunut hyväksi testejä tehdessä.

Testistepeissä on myös hyvä varmistaa välillä, että testissä on tapahtunut mitä halutaan. Esimerkiksi avattaessa selain tiettyyn osoitteeseen voidaan varmistaa, että selaimen otsikko vastaa odotettua. Tai jos verkkokaupasta haetaan tiettyä tuotetta, että tulosten sivulla esiintyy tuotteen nimi. Tämä helpottaa testin epäonnistuessa sen tutkintaa. Selaintesteissä kannattaa myös tarkastella ehtiikö selaimessa tapahtua halutut asiat vai kannattaako testiin rakentaa hidastuksia. Tässä selaimen aukeamisessa sekä hakutulosten latautumisessa kesti pidempään, joten näitä varten piti tehdä joidenkin sekuntien hidastuksia testien ajon onnistumiseksi. Selaimen kautta ajettavat testit osoittautuivat odotetusti

rajapintatestejä hitaammiksi ajaa. Tässä se ei ollut suuri ongelma, mutta testien lukumäärän kasvaessa voi olla järkevä suosia siitäkin syystä rajapintatestejä. Toinen syy on se, että selaimeen todennäköisesti tulee useammin muutoksia ja nämä mahdollisesti käyvät ilmi vasta testien epäonnistuessa.

5.4 Testien ajaminen ja raportointi

Robot Framework testit voidaan ajaa editorin kautta, mutta onnistuu myös komentoriviltä. Testatessa eri koneella on riittänyt asentaa koneelle Python, Robot Framework ja tarvittavat kirjastot sekä ajurit. Testit on voinut siirtää eri koneelle tiedostoina ja pienet muutokset testeihin pystyy tekemään esimerkiksi Notepad-tekstieditorilla. Testejä tehdessä tässä on käytetty yksinomaan Eclipse/Red-yhdistelmää, sillä tässä onnistuu myös testin ajo virheenjäljitystilassa.

Testien ajossa muodostuu automaattisesti Report-, Log- ja Output-tiedostot sekä testin epäonnistuessa tallennetaan kuvakaappaus epäonnistuneesta vaiheesta. Report- ja Log-tiedostot ovat HTML-tiedostoja ja Output-tiedosto on XML-muotoinen. Nämä kolme ylikirjoitetaan uusien ajojen myötä, joten ne kannattaa poimia erilliseen kansioon ajojen jälkeen, esimerkiksi ajopäivän mukaan nimeten. Kuvakaappauksista tulee uusi tiedosto jokaisesta epäonnistumisesta, mutta on selkeämpää poimia nämä yhdessä Report- ja Log-tiedostojen kanssa. Näitä voi testisetin ajossa tulla useampia ja kuvakaappausten listasta tulee pitkä ajomäärän lisääntyessä. Report-tiedostosta löytyy testiajon statistiikka ja tämä on selkeä raportti jakaa. Log-tiedostossa on tietoa ylätasolta alaspäin yksittäisiin steppeihin asti ja tästä on enemmän hyötyä esimerkiksi epäonnistuneen stepin tutkinnassa. Molemmat käyttävät testisettien ja testien nimiä sekä Log-tiedostossa mennään avainsanatasolle. Tämän takia nimeämisten selkeys on tärkeää, sillä jos omien avainsanojen ja testien nimeämisissä on mahdollisimman kuvaava ja selkeä, niin sitä vähemmän tarvitsee Report-tiedoston sisältöä selitellä muille. Eli kannattaa huomioida myös testejä rakentaessa, että miten niiden ajot näkyvät raporteilla.

Liitteessä 2 on käyttöohje, jossa on kuvattu testien tekemisen perusteet, mallitestisetti ja testien ajamisen sekä raportoinnin ohje.

6 Pohdinta

Työssä on käyty läpi testauksen teoriaa ja automatisointia tarkemmin. Useasti huomasi, että moni asia pätee periaatteessa, mutta käytännössä toteutus riippuu projektista. Vaikka ohjelmistokehitys ja testaus ovat tuttuja entuudestaan, huomasi näihin liittyen vääriä oletuksia. Esimerkiksi perinteinen kehitys ei ole niin joustamatonta kuin annetaan ymmärtää eikä ketterä kehitys tarkoita dokumentoinnin puutetta. Kummassakin kehityksen haarassa on monia eri kehitysmalleja, joista voi toimia eri malli tai mallien yhdistelmä eri projektissa. Ketterän kehityksen yleistyttyä on ehkä vapaampaa valita projektiin käytettäviä metodeja ja työkaluja, etenkin avoimen lähdekoodin ilmaisten työkalujen yleistyttyä.

Testausautomaation käyttöönottoon liittyen tärkeimmiksi asioiksi nousee suunnitelmallisuus ja tavoitteiden selkeys. Yleistäen testausautomatisaatiosta on hyötyä ohjelmistokehityksen mallista riippumatta, mutta etenkin ketterän kehityksen malleissa testausautomatisaatio on edellytys onnistumiselle. Pienemmissä projekteissa ja ilman siihen sitoutumista testausautomaatiosta ei välttämättä saada kaikkea hyötyä irti ja huonosti toteutettuna se voi olla haitaksi projektille. Sitoutuminen testausautomatisointiin tarkoittaa yleensä siihen panostamista ajallisesti tai rahallisesti, sekä ideaalitilanteessa sen huomiointia projektin alusta lähtien. Testausautomaatio vaatii myös ylläpitoa kehityksen edetessä. (Fewster & Graham 2012, Reflections on the Case Studies.)

Tavoitteena opinnäytetyössä oli tutustua Robot Frameworkin toimintaan ja tehdä selkeät asennus- ja käyttöohjeet sekä toimiva testisetti. Alkuperäisenä tavoitteena oli tehdä vain selaintestejä, mutta työn aikana tavoite laajeni kattamaan myös rajapintatestisetin tekemisen. Testejä tehdessä huomasi käyttöliittymän ohittamisen hyödyt.

Testien tekeminen hyvin vie aikaa ja vaati alkuvaiheessa täyden uudelleen aloittamisen, sekä testit kokivat monta päivitystä. Ennen uudelleen aloittamista tehty testisetti oli sinällään toimiva, mutta testien päivitys olisi vienyt liikaa aikaa ja esimerkiksi testin hakuehtojen muuttaminen vaati testien päivittämistä useampaan paikkaan. Lisäksi testien ja projektien hierarkia piti miettiä uudestaan järkevän tekemisen ja raporttien muodostumisen kannalta. Testien valmistuttua näiden ajo on ollut huomattavasti nopeampaa kuin tehdä manuaalitestausta kattaen samat asiat. Testejä tullessa lisää huomasi, että turhaan teki ensin jokaisen testin alkuun ”avaa selain sivulle x” -toiminnon, sillä kätevämpi tehdä siitä oma palikka, jota kutsui tarvittavissa testeissä. Oli vaikeaa, mutta kannattavaa, pysähtyä pohtimaan etukäteen mitä rakennuspalikoita tarvitsee testeihin. Lisäksi muutaman harjoi-

tuskierroksen jälkeen ymmärsi paremmin Log- ja Report-tiedostoja tarkasteltua, miten testisetit, testit ja testistepit kannattaa nimetä. Tavoitteena oli saada raporteista siten helposti ymmärrettäviä.

Testisettiä pystyy hyödyntämään tämän työn testausta kattavampaan testaukseen, mutta silloin kannattaa työstää tehokkaampi tapa tuoda testiaineistoa, esimerkiksi eri hakuehtoja. Vaikka nämä ovat nyt muuttujissa, niin esimerkiksi Excel-tiedoston kautta voisi olla helpompi päivittää ja seurata käytettyjä hakuehtoja. Voisi olla hyödyllistä tehdä lisäksi erillinen muuttujatiedosto tai kirjasto, jolloin muuttuja tai avainsanat olisivat yhdessä tiedossa helposti päivitettävissä ja käytettävissä helposti eri testiseteille tai projekteille. Myös testihierarkia on hyvä pohtia testien lisääntyessä.

Robot Frameworkin testien tekeminen onnistuu ilman vahvaa ohjelmointiosaamista, mutta selaintestien tekemisessä HTML-tuntemuksesta on apua. Ja aiemmista ohjelmointikursseista on siinä mielessä hyötyä, että eri kirjastojen avainsanojen toiminnallisuuksien ymmärtäminen on helpompaa. Kirjastoja ja niiden kuvauksia kannattaa käydä läpi laajemminkin ennen testien tekoa. Oli hyödyllistä hahmotella etukäteen esimerkiksi Excel-tiedostoon mitä testejä tarvitaan ja mitä näissä pitäisi tapahtua. Sen oli hyödyllistä käydä läpi kirjastoja, kun tiesi mitä toiminnallisuuksia etsiä. Käyttöliittymäpuolen testeihin oli selkeämpi hakea toimintoja, kuten anna syöte haluttuun kenttään ja paina valittua painiketta. Rajapintatesteissä tämä oli haasteellisempaa. Rajapintojen käyttö ei välttämättä onnistu ilman jonkinlaista ymmärrystä rajapintojen toiminnasta, mutta kovin syvällistä perehtymistä tässä ei tarvinnut päästäkseen alkuun. Testauksessa käytettävän rajapinnan kuvaus oli tärkeä ja näiden testien teossa piti etsiä enemmän esimerkkejä siitä, miten testi voidaan ja kannattaa rakentaa.

Työn aikana tehdyt testisetit ovat käyttökelpoisia, mutta tulevat laajentumaan käytössä. Robot Frameworkin kokonaisuudesta ei tullut hyödynnettyä kuin pieni osa ja osaaminen laajenee jatkossa. Rajapinnoista ja rajapintatestauksesta on hyötyä ja näistä olisi mielenkiintoista tehdä jatkotutkimusta.

Lähteet

Agilemanifesto.org 2001. Luettavissa: https://agilemanifesto.org/iso/fi/manifesto.html. Luettu 18.4.2020.

Black, R. 2016. Advanced Software Testing - Vol. 1. Rocky Nook. Santa Barbara, USA.

Capgemini, Microfocus & Sogati 2019. World Quality Report 2017-2018. Luettavissa: https://www.qasymphony.com/landing-pages/report-the-evolution-of-test-automation/. Luettu 10.2.2020.

Cline, Alan. 2015. Agile Development in the Real World. How and why to use Agile Methods. Apress. New York.

Crispin, L & Gregory, J. 2008. Agile Testing: A practical Guide for Testers and Agile Teams. Addison-Wesley Professional. Upper Saddle River, NJ. Luettavissa: https://learning.oreilly.com/library/view/agile-testing-a/9780321616944/?ar. Luettu: 6.4.2020.

Crispin, L & Gregory, J. 2015. More Agile Testing: Learning Journeys for the Whole Team. Addison-Wesley. Upper Saddle River, NJ.

Coleman, G., Walsh, M. Cornanguer, I. Kakkonen, K. & Sabak J. 2017. Teoksessa Black, R. (toim.). Agile testing foundations: an ISTQB foundation level Agile tester guide. BCS, The Chartered Institute for IT. Swindon.

Dooley, J. 2011. Software Development and Professional Practice. Apress. New York. Luettavissa: https://learning.oreilly.com/library/view/software-development-and/9781430238010/?ar. Luettu: 19.4.2020.

Downey, A. 2016. Think Python. O'Reilly Media. Sebastopol. Luettavissa: https://learning.oreilly.com/library/view/think-python-2nd/9781491939406/?ar. Luettu 25.4.2020

Fewster, M & Graham, D. 2012. Experiences of Test Automation. Case Studies of Software Test Automation. Addison-Wesley. Upper Saddle River, NJ. Luettavissa: https://learning.oreilly.com/library/view/experiences-of-test/9780132776608/. Luettu: 15.3.2020.

Finnish Software Testing Board (FiSTB) 2015. ISTQB:n testaussanasto v.2.3 Suomi – Englanti. Luettavissa: http://www.fistb.fi/sites/fistb/files/liitteet/istqb_sanasto_2015-04-30%202.3%20FI-ENG.pdf. Luettu: 7.11.2019.

Gauf, B., Garrett, T. & Dustin, E. 2009. Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality. Addison-Wesley. Upper Saddle River, NJ.

Haikala, I & Mikkonen, T. Ohjelmistotuotannon käytännöt. 2011. Talentum. Helsinki.

International Software Testing Qualifications Board (ISTQB) 2019. Worldwide Software Testing Practices Survey 2017-18. Luettavissa: https://www.istqb.org/references/surveys/istqb%C2%AE-worldwide-software-testing-practices-survey-2017-18.html. Luettu: 1.10.2019.

Julkisen hallinnon tietohallinnon neuvottelukunta (JUHTA) 2018. JHS-suositukset, JHS 166 Julkisen hallinnon IT-hankintojen yleiset sopimusehdot. Liite 10. Luettavissa: http://docs.jhs-suositukset.fi/jhs-suositukset/JHS166_liite10/JHS166_liite10.html. Luettu 25.4.2020.

Kasurinen, J. 2013. Ohjelmistokehityksen käsikirja. Docendo. Jyväskylä.

Lewis, W., Dobbs, D. & Veerapillai, G. 2017. Software Testing and Continuous Quality Improvement. Taylor & Francis Group, LLC. Boca Raton. Luettavissa: https://learning.oreilly.com/library/view/software-testing-and/9781351722209/. Luettu: 15.2.2020.

Moiz, S. 2017. Uncertainty in Software Testing. Teoksessa Mohanty, H. Mohanty, J & Balakrishnan, A. (toim.). Trends in Software Testing. Springer. Singapore.

Morgan, P., Samaroo, A., Thompson, G. & Williams, P. 2015. Teoksessa Hambling, B. (toim.). Software Testing: An ISTQB-BCS Certified Tester Foundation guide. BCS Learning & Development Limited. Swindon. Luettavissa: https://learning.oreilly.com/library/view/software-testing-/9781780172996/. Luettu: 4.4.2020.

Myers, L., Badgett, T. & Sandler, C. 2012. The Art of Software Testing. John Wiley & Sons. Hoboken.

The Open Group 2018. Base Specifications Issue 7. Luettavissa: https://pubs.opengroup.org/onlinepubs/9699919799/. Luettu 25.4.2020.

Penmetsa, J. 2017. Agile Testing. Teoksessa Mohanty, H. Mohanty, J & Balakrishnan, A. (toim.). Trends in Software Testing. Springer. Singapore.

QASymphony & Techwell 2019. The Evolution of Test Automation – Results from the 2018 QASymphony and Techwell Survey. Luettavissa: https://www.qasymphony.com/landing-pages/report-the-evolution-of-test-automation/. Luettu 10.2.2020.

Raghav, P. Automation Step by Step. Luettavissa: https://automationstepbystep.com/. Luettu: 1.9.2019.

Rajkumar, S. 2018. Types of Test Automation Frameworks. Luettavissa: https://www.softwaretestingmaterial.com/types-test-automation-frameworks/. Luettu 29.8.2019

Robot Framework 2019. Robot Framework. Luettavissa: http://robotframework.org/. Luettu 29.8.2019.

SeleniumLibrary 2020. SeleniumLibrary. Luettavissa: https://github.com/robotframework/SeleniumLibrary/. Luettu 16.4.2020.

Software Testing Help 2020. Most Popular Test Automation Frameworks with Pros and Cons of Each –Selenium Tutorial #20. Luettavissa: https://www.softwaretestinghelp.com/test-automation-frameworks-selenium-tutorial-20/. Luettu: 22.3.2020.

Spillner, A., Linz, H. & Schaefer, H. 2014. Software Testing Foundations. Rocky Nook. Santa Barbara.

Webopedia 2020. What is Library? Webopedia definition. Luettavissa: https://www.webopedia.com/TERM/L/library.html. Luettu 25.4.2020.

Liitteet

Liite 1. Asennusohjeet

Robot Framework asennusohje

Versio 1

25.4.2020

Sisällys

Versionhallinta

Päivä	Versio	Kuvaus	Tekijä
10.10.2019	0.1	Ensimmäinen versio	Tiina Anttila
13.2.2020	0.2	Täydennys	Tiina Anttila
5.4.2020	0.3	Asennuksen testaus ja täydennys	Tiina Anttila
25.4.2020	1.0	Viimeistelty versio	Tiina Anttila

1 Ohjelmat ja johdanto

Asennettavat ohjelmat Windows 10 käyttöjärjestelmälle:

- Python 3.8.2
- Robot Framework 3.1.2
- SeleniumLibrary 4.3.0
- RESTinstance 1.0.2
- ChromeDriver 81.0.4044.69
- Java SE Development Kit 14 (2020-03-17)
- Eclipse 2019-09 (4.13)
- RED 0.9.3

Jos yksi tai useampi ohjelmista on jo käytössä, on hyvä varmistaa käytössä olevien versioiden olevan yhteensopivia ja tarvittaessa päivittää ne.

Robot Framework edellytykset löytyvät asennusohjeista:
https://github.com/robotframework/robotframework/blob/master/INSTALL.rst
Robot Framework edellyttää Python asennusta ja verkkosivujen testaamiseen tarvitaan SeleniumLibrary sekä valitun selaimen ajuri.

Eclipse edellytykset löytyvät asennusohjeista:
https://wiki.eclipse.org/Eclipse/Installation
Eclipse on java-pohjainen sovellus ja edellyttää toimiakseen Java Runtime Environment (JRE) toimiakseen. Jos Eclipsellä on tarkoitus koodata javaa, niin JRE:n sijaan kannattaa käyttää Java Development Kit (JDK).

RED asennustiedot ja edellytykset:
http://nokia.github.io/RED/help/first_steps/download_install.html
RED asennusohjeissa kerrotaan, että RED on testattu ja julkaistu Eclipse 2019-09 (v4.13) kanssa, joten ei kannata ladata uusinta versiota Eclipsestä.

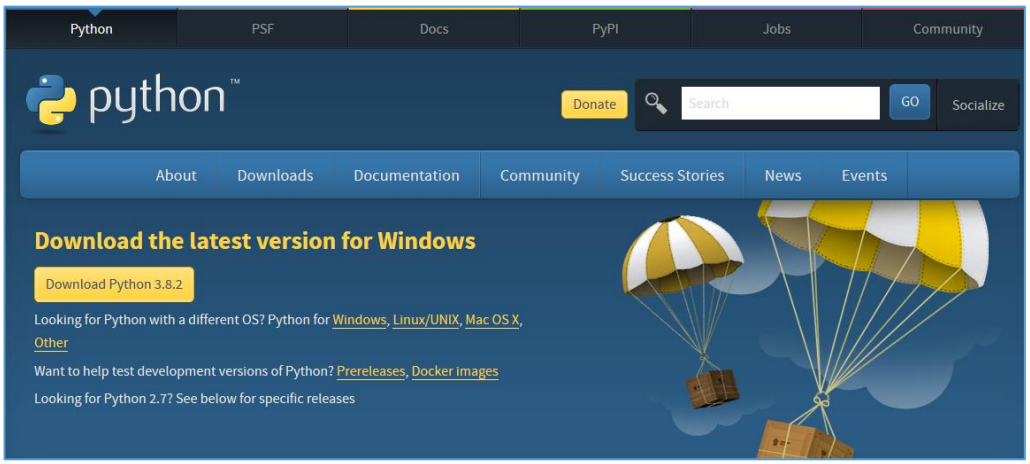
Luvussa kaksi käydään läpi ohjelmien asennus, luvussa kolme PATH-asetukset ja luvussa neljä päivitys- sekä poisto-ohjeet.

2 Asennus

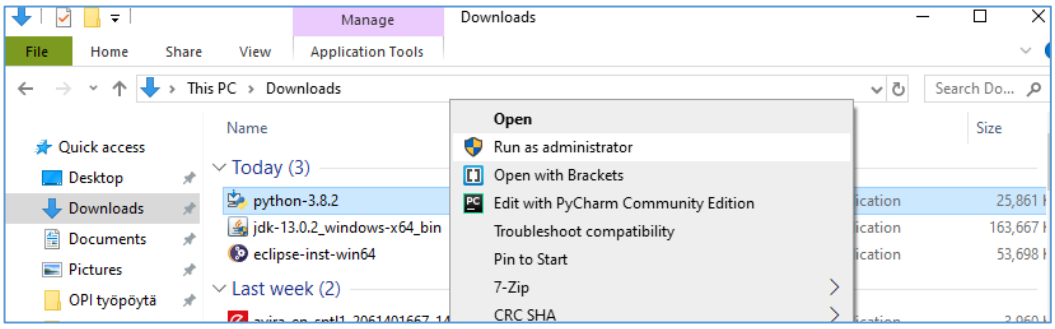
Käydään jokaisen ohjelman asennus ja näiden versioiden tarkistaminen läpi.

2.1 Python

Python löytyy Python.org -sivustolta. Sieltä valitaan etusivulla ehdotettu uusin Python 3 -versio Windowsille. Downloads-osiosta pystyy tarkastelemaan kaikkia saatavilla olevia latauksia. Python 3 -versio sai olla uusin, mutta sillä ei ollut suurta merkitystä toimintaan. Python 2 -versiota ei kannata ottaa käyttöön, sillä sen tuki loppuu 2020 aikana.

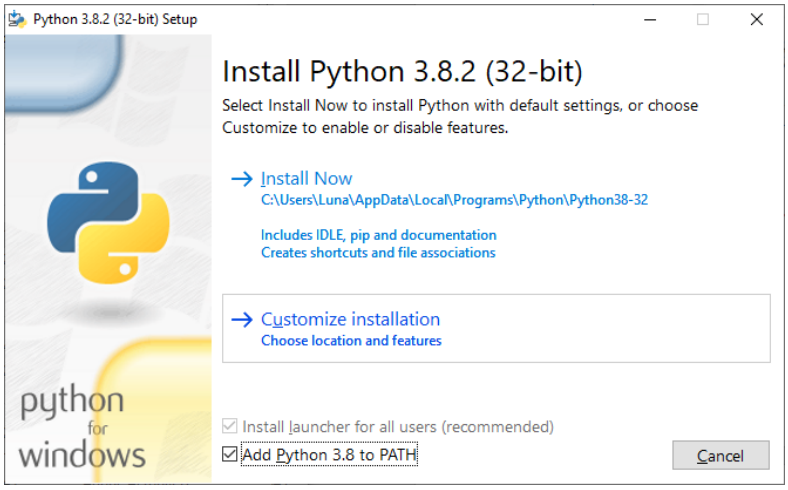


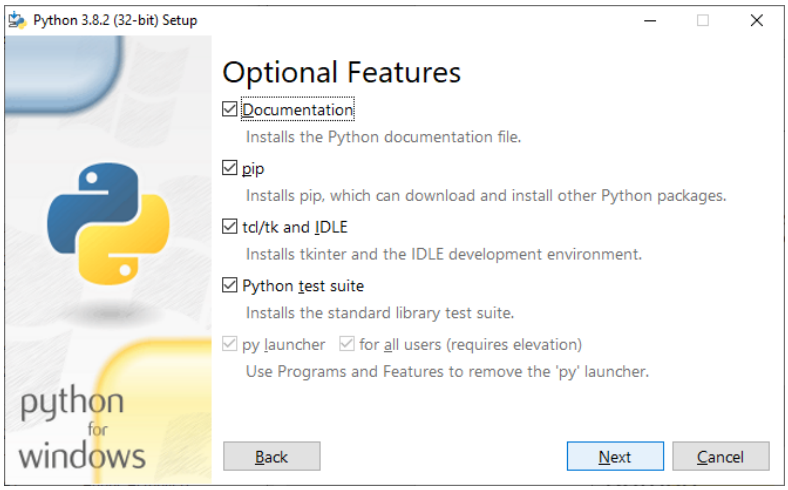
Latauksen jälkeen suoritetaan tiedosto, tarvittaessa administrator eli järjestelmänvalvojan roolissa:



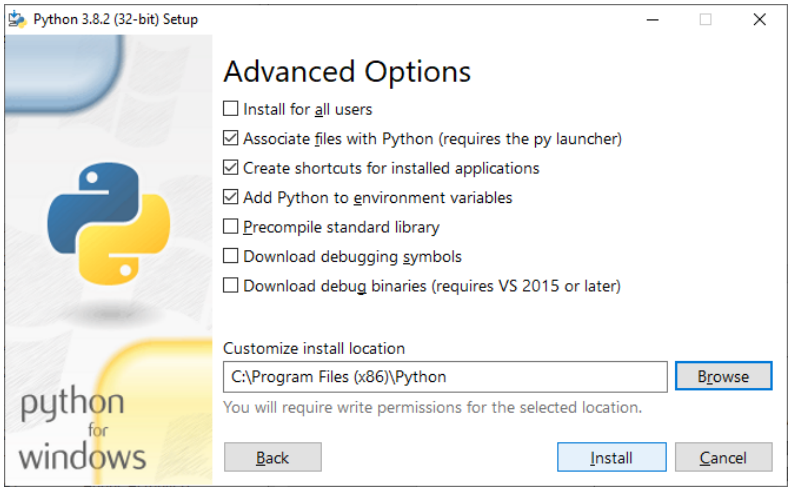
Tästä aukeaa Python installer.

Lisätään valinta Add python 3.8 to PATH -ruutuun. Tämän jälkeen voidaan valita Customize installation, sillä sitä kautta pystyy määrittämään kohdekansion. Valintoja ei muutoin tarvitse muuttaa asennuspolussa.



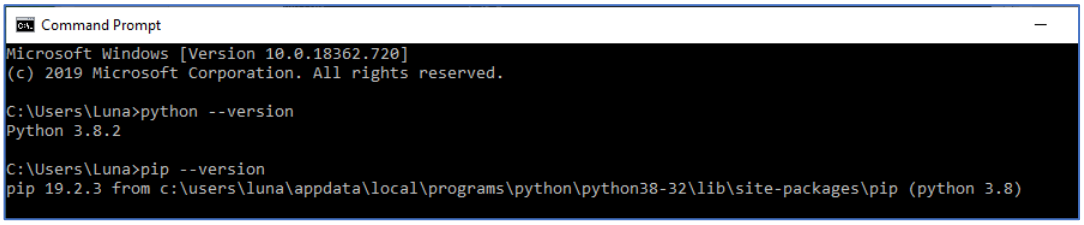


Valitaan haluttu kohdekansio Browse-painikkeen kautta ja varmistetaan että valittuna on Add Python to environment variables -valintaruutu.

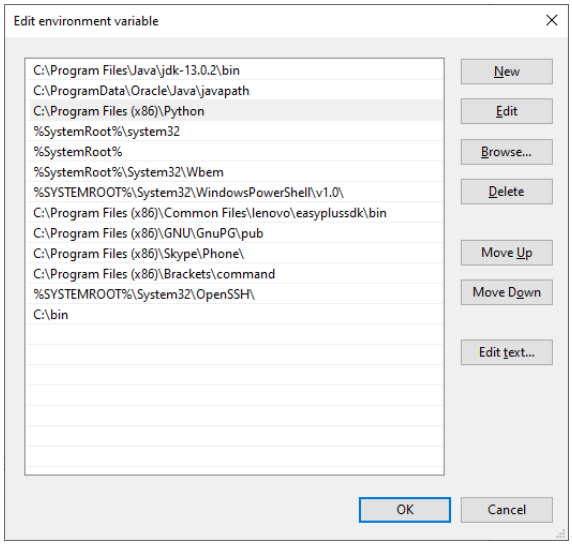


Asennuksen jälkeen voi tarkistaa löytyykö Python komentorivin kautta komennolla:

```
python --version
```



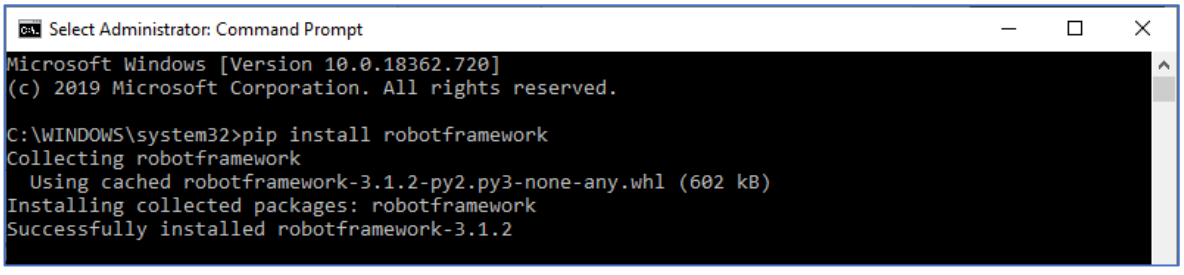
Jos komennolla ei löydy mitään, niin joko Pythonia ei ole asentunut, tai sille puuttuu PATH-ympäristömuuttuja. Python kansion voi lisätä Environment variables kautta Path-listaukseen. Tämän ohje löytyy luvusta kolme.



2.2 Robot Framework ja kirjastot

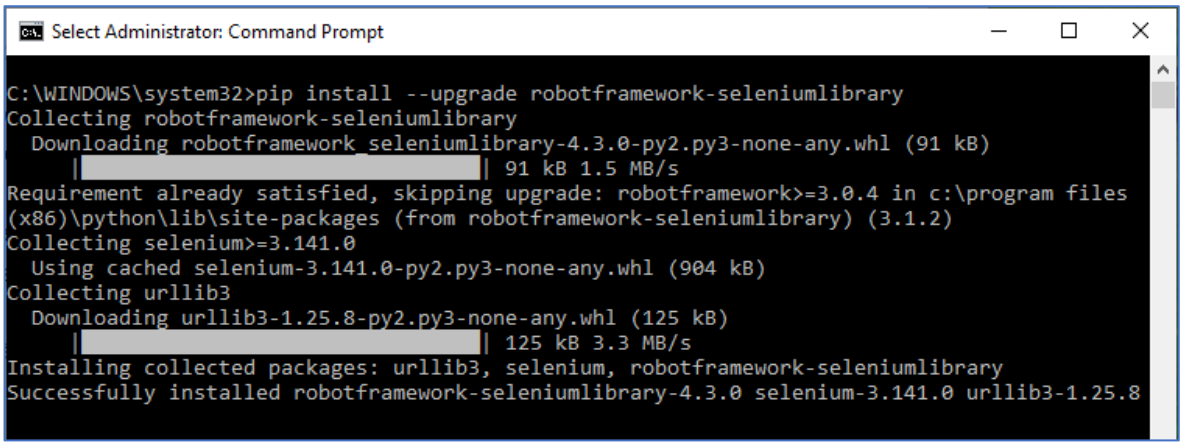
Asennus Pip kautta tapahtuu komentorivin kautta komennolla:

```
pip install robotframework
```

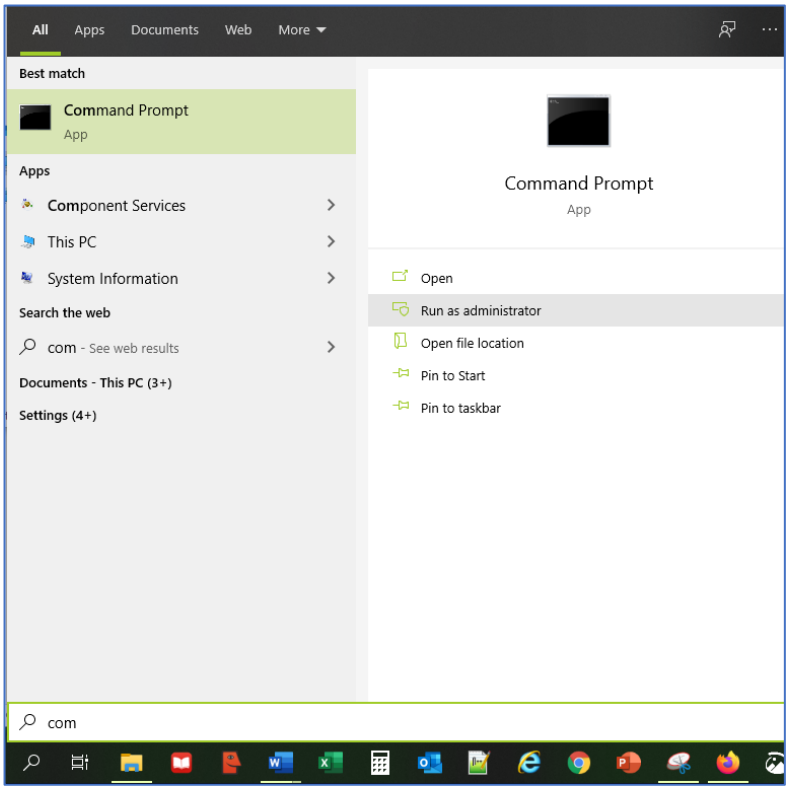


Kun asennetaan myös kirjastoja, kuten SeleniumLibrary, voidaan käyttää edellisen sijaan komentoa:

```
pip install --upgrade robotframework-seleniumlibrary
```



Jos asennus ei onnistu ja virheilmoituksessa mainitaan puuttuvista oikeuksista, suoritetaan komentorivi administrator-roolissa:

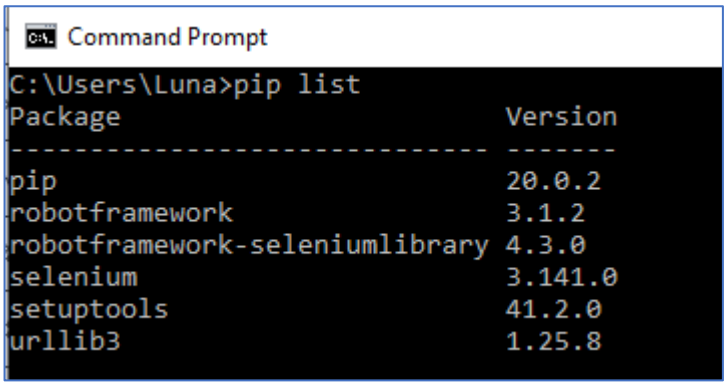


Muiden kirjastojen, kuten API-testaukseen tarvittavan RESTinstance-kirjaston asennus käy samaan tapaan. Komennolla:

```
pip install --upgrade RESTinstance
```

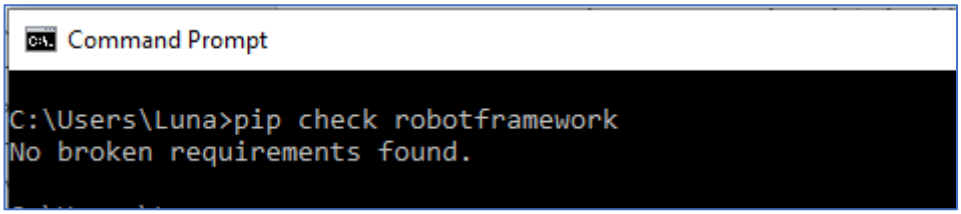
Asennuksen onnistumisen voi tarkistaa komentoriviltä esimerkiksi komennolla

```
pip list
```



Voi myös tarkistaa onko kaikki kunnossa Robot Frameworkin osalta komennolla:

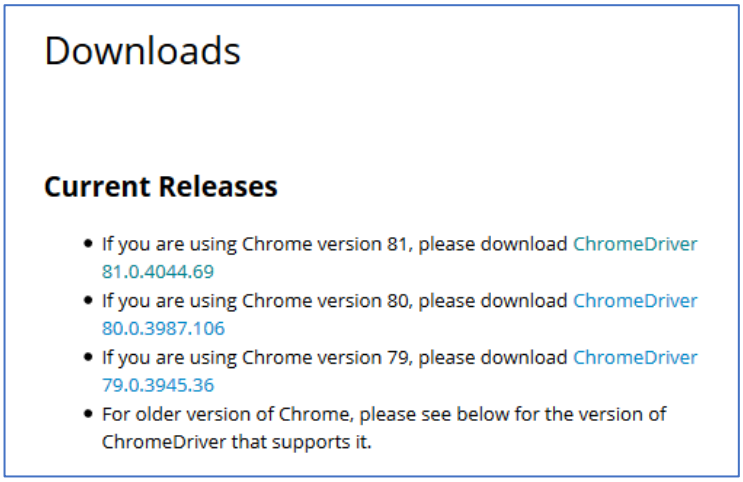
```
pip check robotframework
```

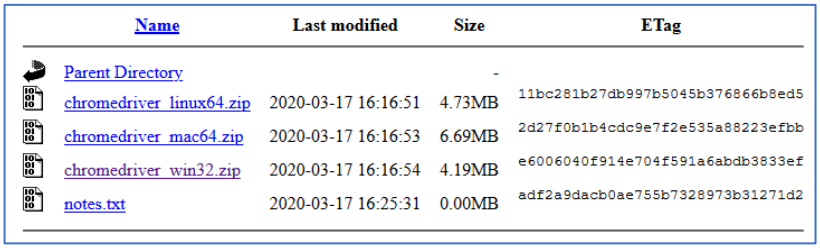


2.3 Verkkoajuri

ChromeDriver on Chromen verkkoajuri. Muille selaimille löytyy erilliset ajurit.

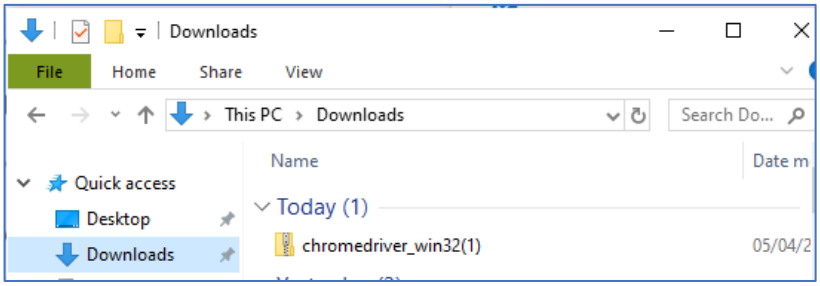
Chromen eri versioille on omat ajurinsa, joten tässä käytössä on Chromen versio 81 ja valitaan siihen sopiva ChromeDriver 81.0.4044.69.



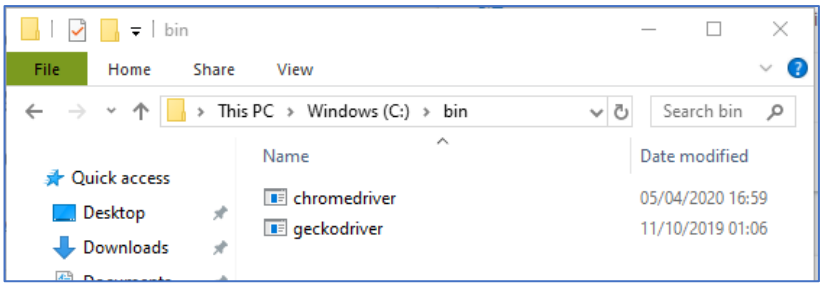


Name	Last modified	Size	ETag
Parent Directory		-	
chromedriver_linux64.zip	2020-03-17 16:16:51	4.73MB	11bc281b27db997b5045b376866b8ed5
chromedriver_mac64.zip	2020-03-17 16:16:53	6.69MB	2d27f0b1b4cdc9e7f2e535a88223efbb
chromedriver_win32.zip	2020-03-17 16:16:54	4.19MB	e6006040f914e704f591a6abdb3833ef
notes.txt	2020-03-17 16:25:31	0.00MB	adf2a9dacb0ae755b7328973b31271d2

Zip-tiedosto löytyy latauskansiosta.



Puretaan valittuun kansioon, joka on lisätty tai lisätään erikseen PATHiin esim. /usr/local/bin.

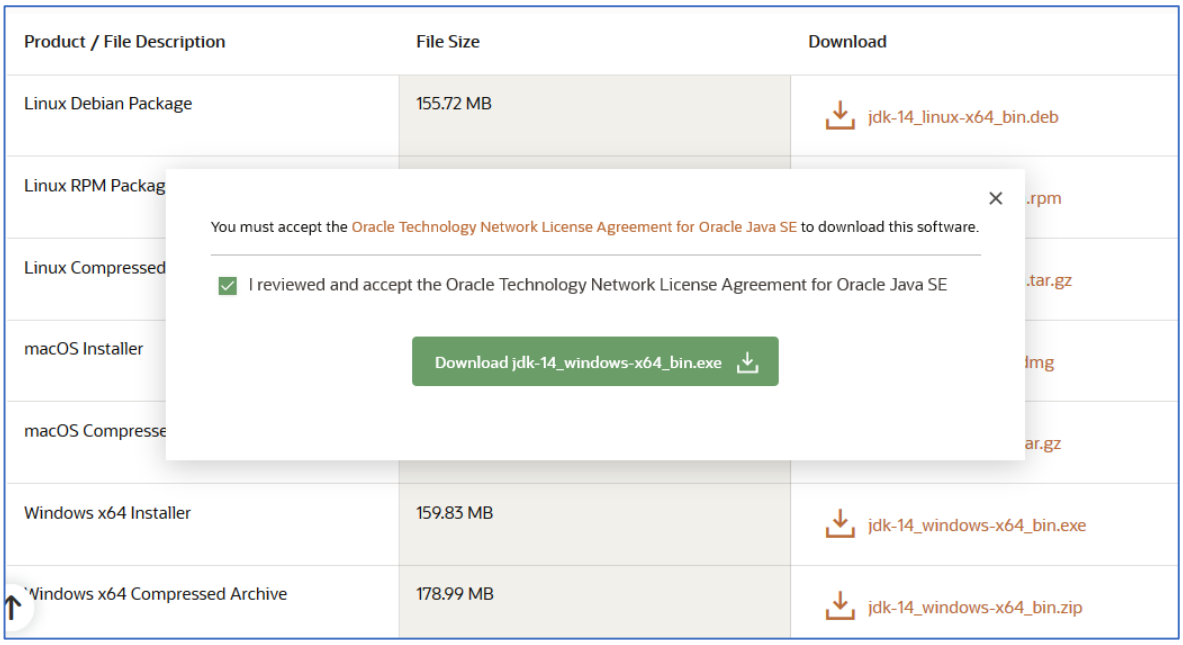


Muita ajureita voi käyttää, jos haluaa käyttää eri selainta. Esimerkiksi GeckoDriver-ajuri soveltuu Firefox-selaimelle, mutta selaimen käyttö vaatii ajurin lisäksi toimiakseen Microsoft Visual Studio redistributable runtime -ohjelmiston asennettuna koneelle.

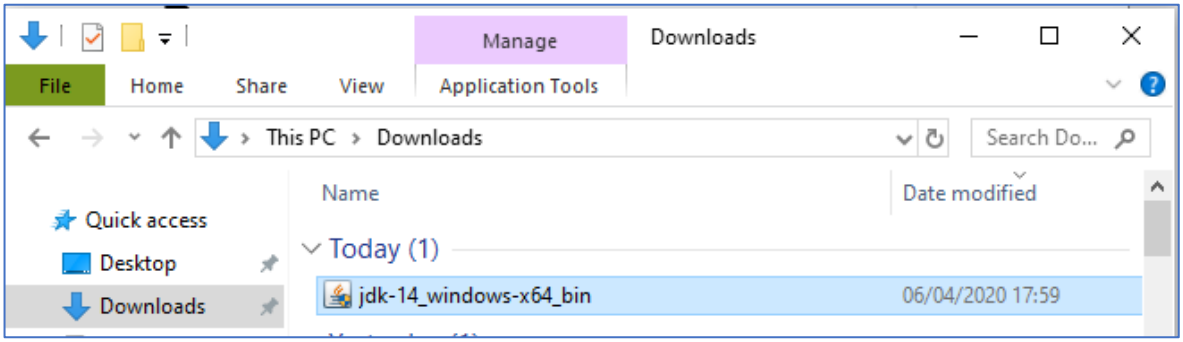
2.4 JDK

Haetaan Java Development Kit (JDK) asennettavaksi. Tämä löytyy helposti Google-hakukoneella.

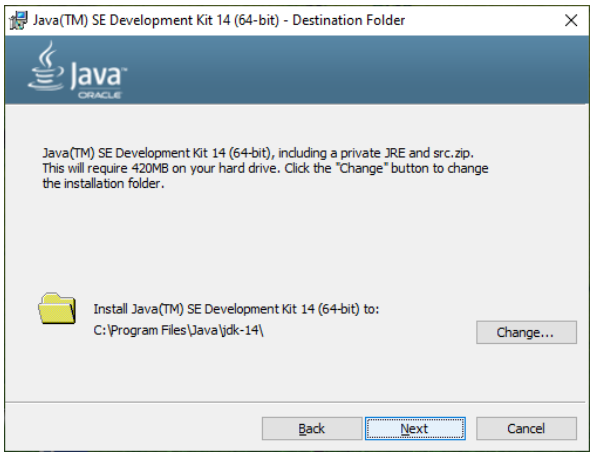
Valitaan Windows installer ja hyväksytään ehdot.

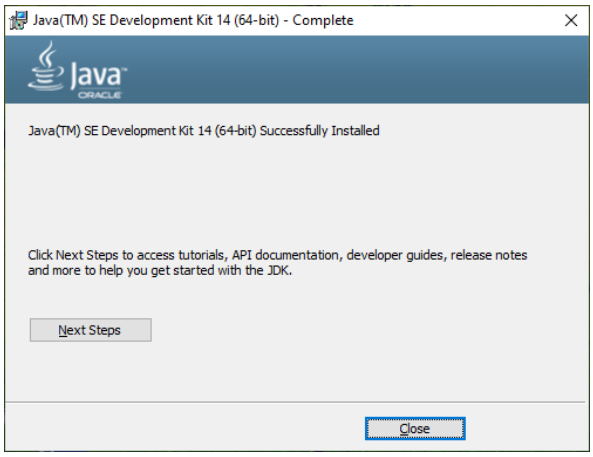


Ajetaan asennuspaketti (tarvittaessa administrator-oikeuksilla):



Hyväksytään asennuksen vaiheet:





Lisätään asennuksen jälkeen Java C:\Program Files\Java\jdk-14\bin Environmsent Variables kautta PATH:iin.

Asennuksen ja polun lisäämisen onnistumisen voi tarkistaa komentoriviltä varmistamassa esimerkiksi asennetun version komennolla:

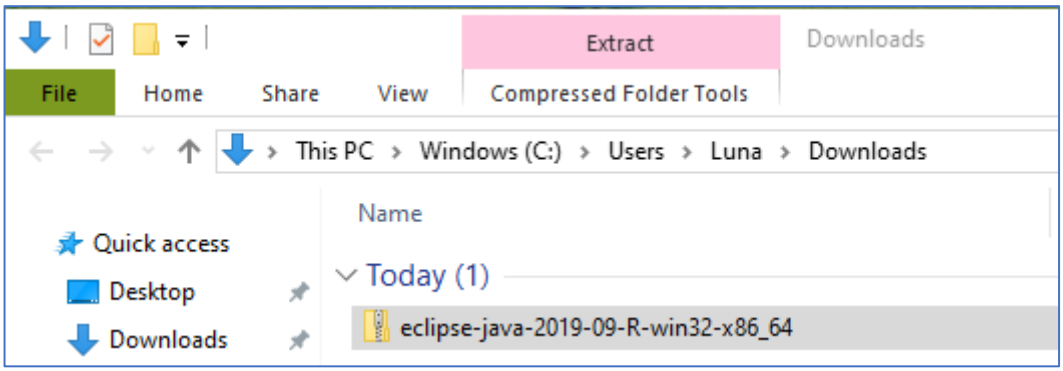
```
Java -version
```

```
C:\Users\Luna>java -version
java version "14" 2020-03-17
Java(TM) SE Runtime Environment (build 14+36-1461)
Java HotSpot(TM) 64-Bit Server VM (build 14+36-1461, mixed mode, sharing)
```

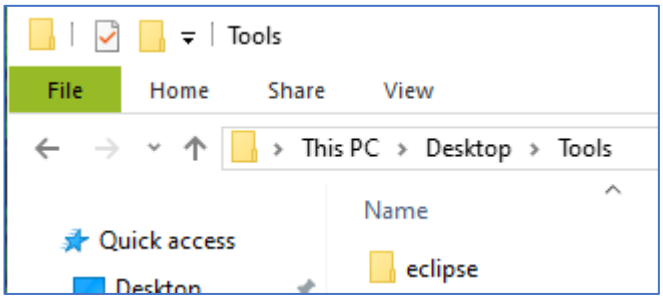
2.5 Eclipse

RED kanssa toimiva Eclipse on 2019-versio, tämä löytyy helpoiten Google-hakukoneella. Eclipse installer asentaa Eclipsestä uusimman version, joka ei ole yhteensopiva RED-editorin kanssa.

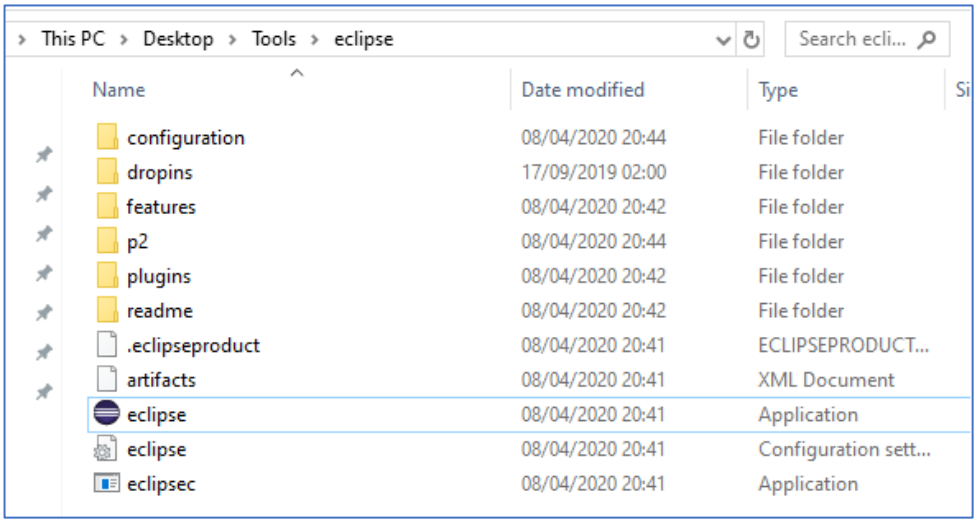
Ladataan halutun version Zip-tiedosto Windowsille:



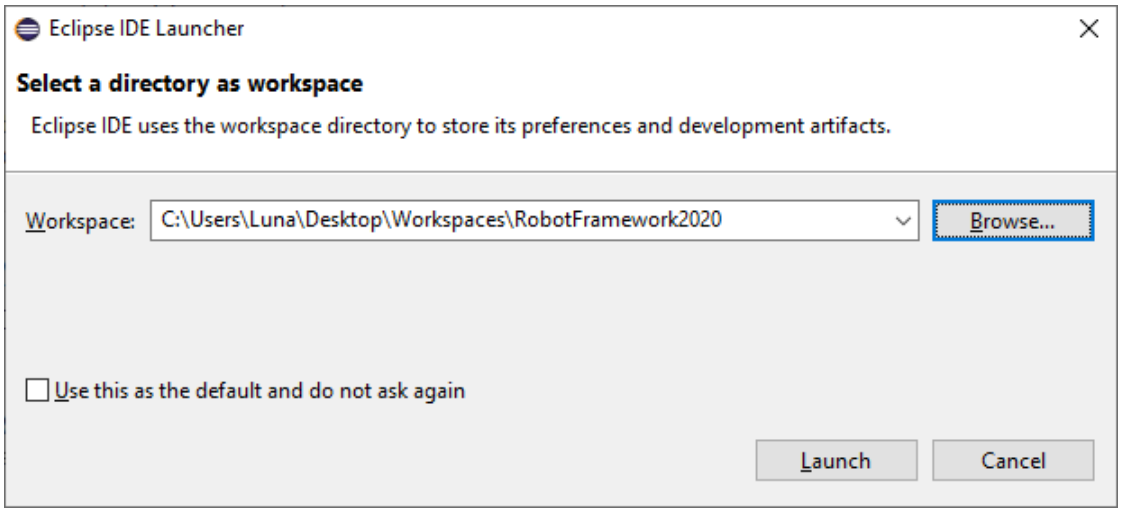
Puretaan zip-tiedosto haluttuun kansioon:



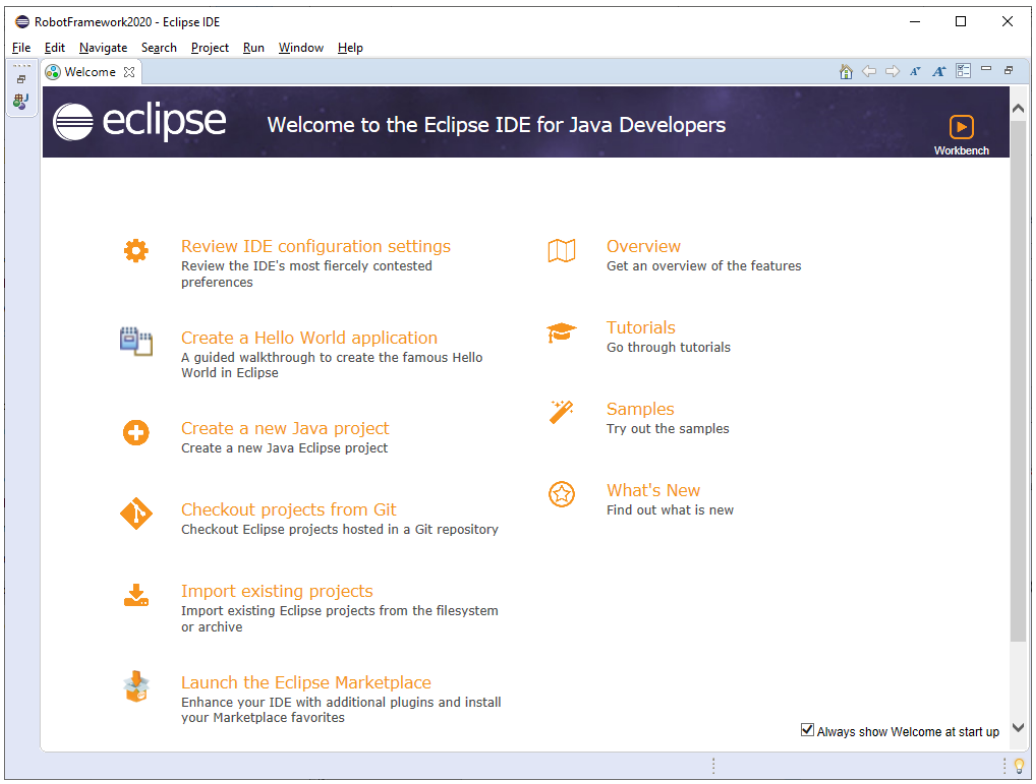
Puretusta tiedostosta voidaan käynnistää Eclipse:



Valitaan jo olemassa oleva tai luodaan uusi työtila:



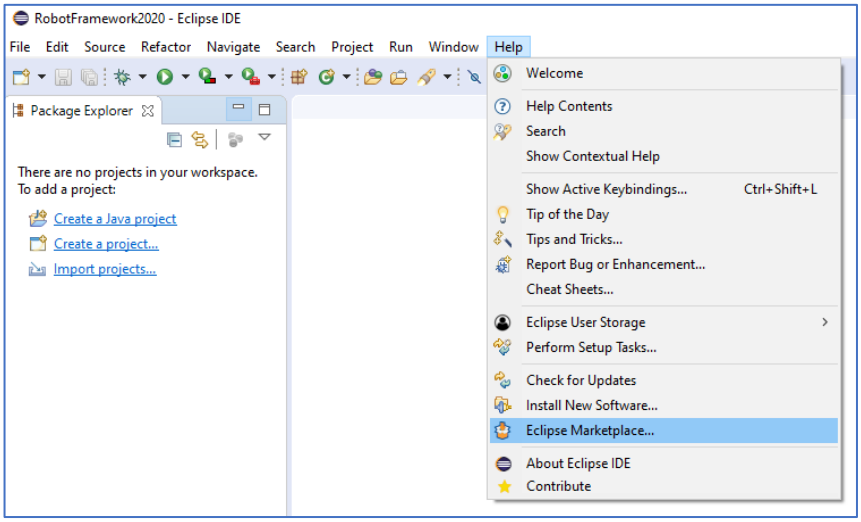
Jos on käytössä useita työtiloja, niin ei kannata valita ”Use this as default”. Jos taas käyttää yhtä työtilaa pidempiä aikoja, voi valitan haluamansa työtilan oletukseksi ja sen pystyy vaihtamaan Eclipsessä File-valikon kautta myöhemmin.



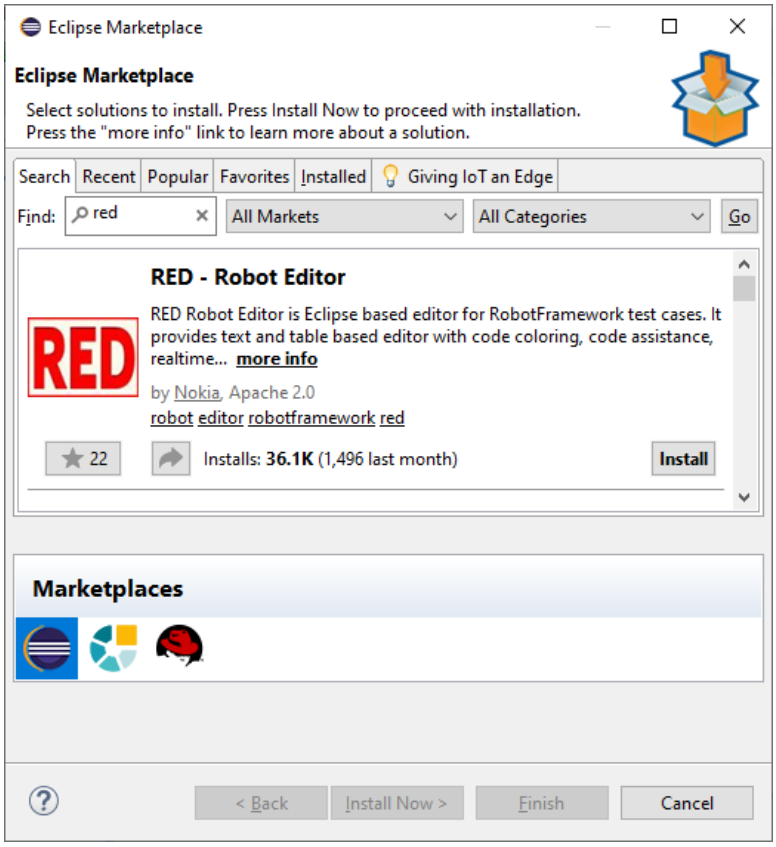
Welcome-välilehdeltä voi halutessaan ottaa valinnan pois oikealta alhaalta kohdasta Always show Welcome at start up. Tämän saa auki myöhemmin seuraavasti: Help – Welcome.

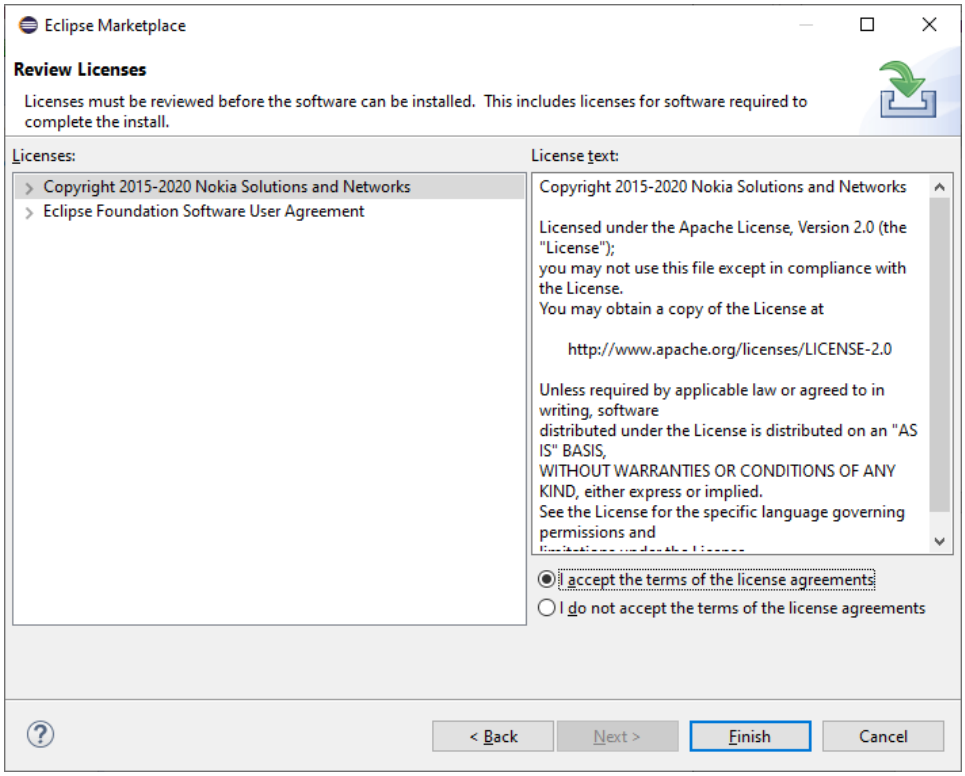
2.6 RED

Eclipsen Help-valikosta löytyy Eclipse Marketplace.

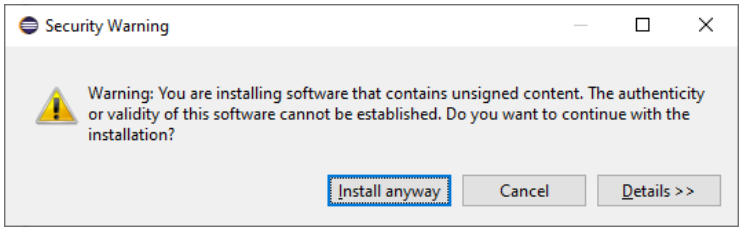


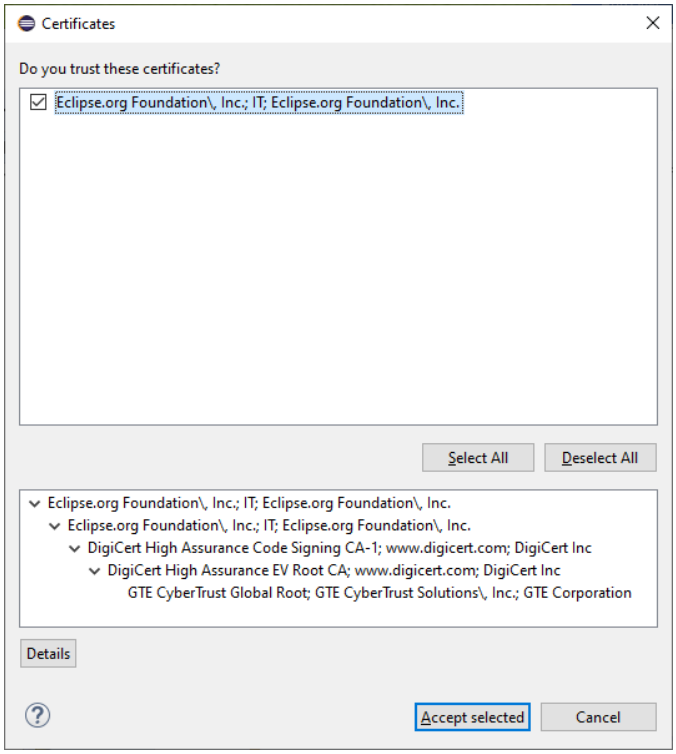
Kirjoitetaan Find-laatikkoon RED ja valitaan se hakutuloksista. Asennuksen saa tehtyä Install-painikkeesta.

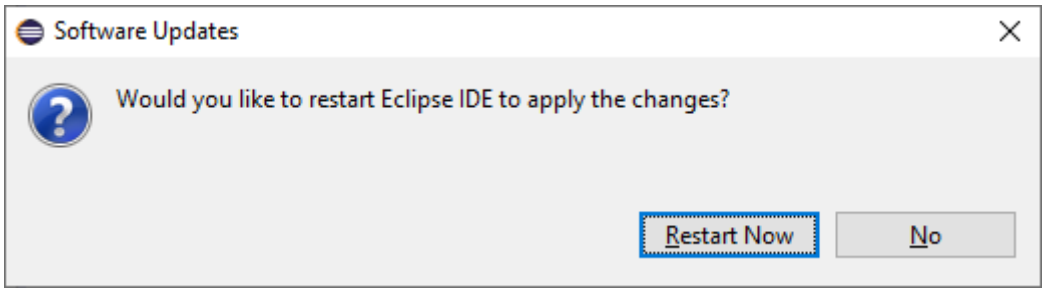




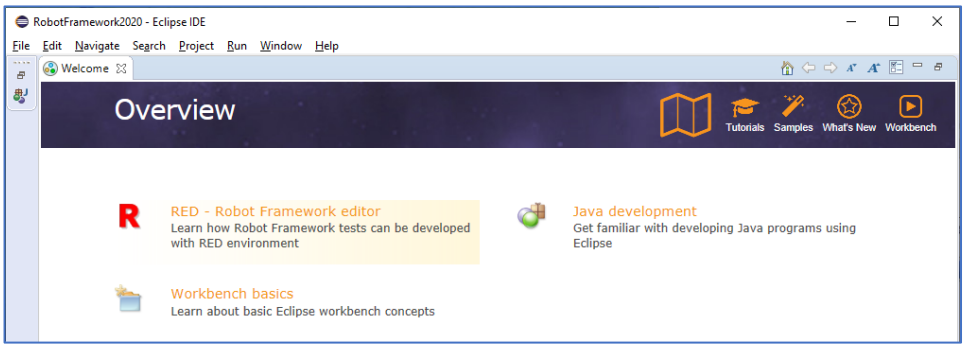
Hyväksytään ehdot ja asennuksen vahvistukset.



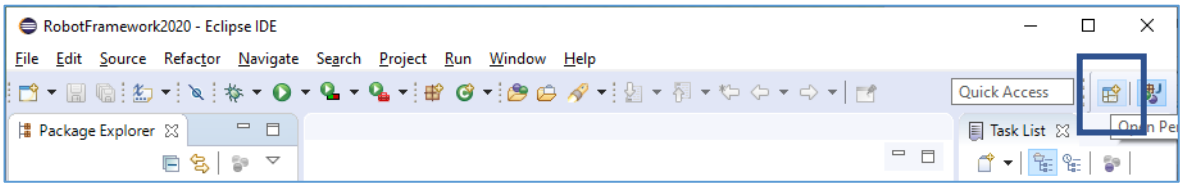


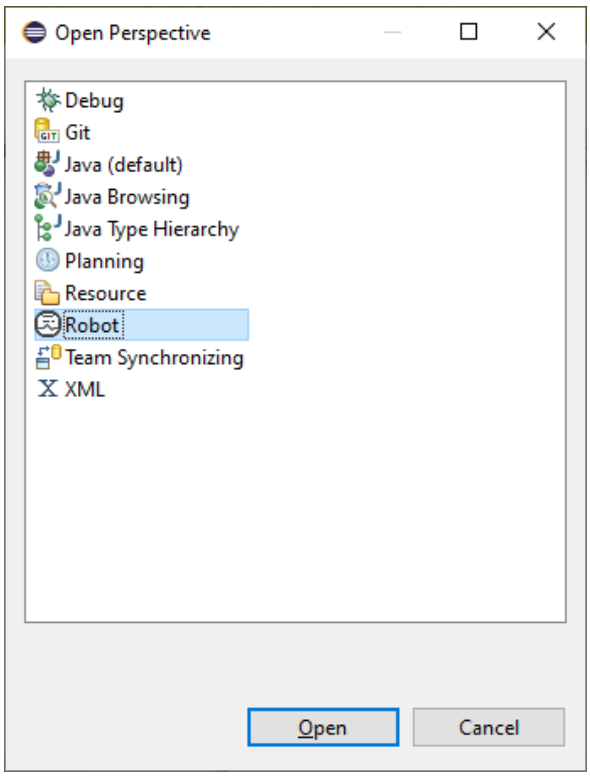


Valitaan Restart Now ja aukeavassa Eclipse Welcome-ikkunassa näkyy nyt linkki RED-ohjeisiin.



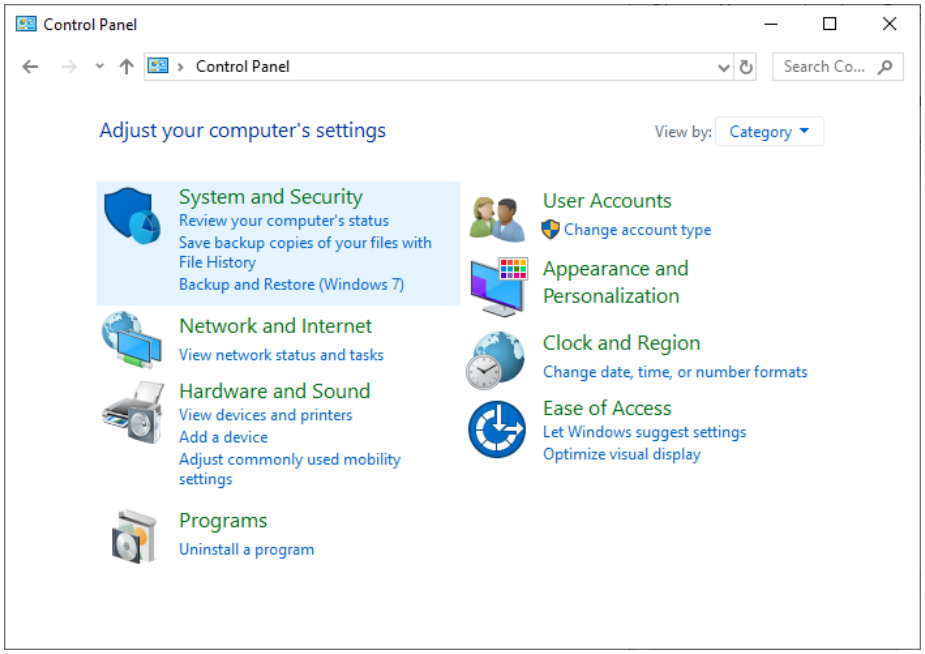
Lisäksi Robot on valittavissa Eclipsen oikeasta yläkulmasta aukeavasta Open perspective-listassa.



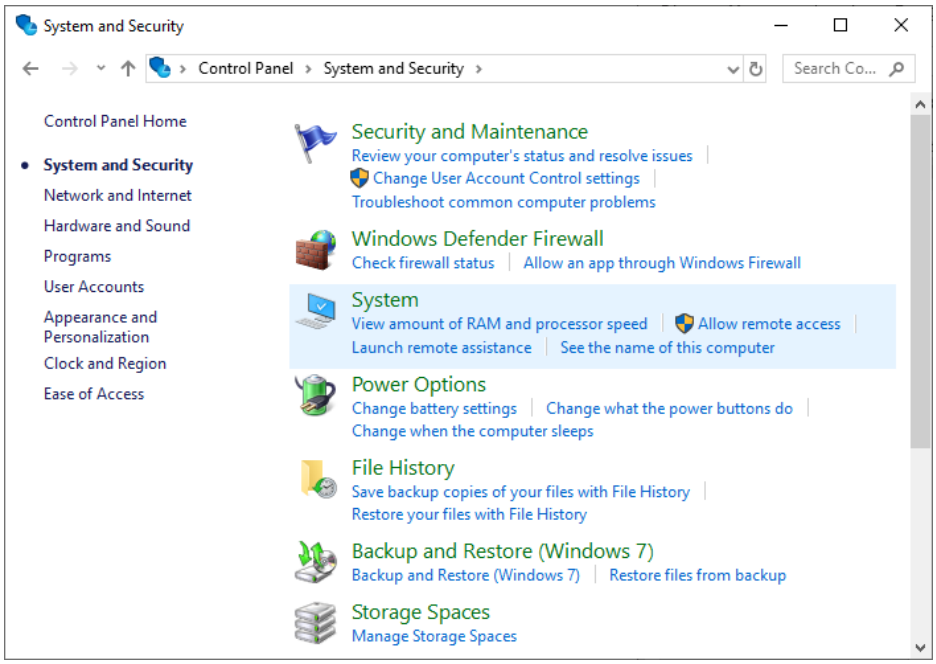


3 PATH-ympäristömuuttuja

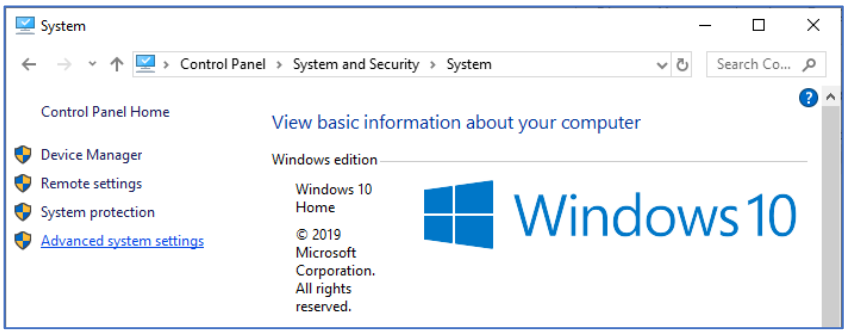
Ohjelmien kansiot pitää viedä PATH-ympäristömuuttujaksi, jotta ne esimerkiksi toimivat komentoriviltä. Lisäksi tämä auttaa Eclipseä löytämään Pythonin käyttöönsä. Ohjelman kansiopolku viedään tietokoneen Environment Variables-osioon Path-listaan. Tämän pystyy tekemään avaamalla Control Panel kautta System and Security -osion:



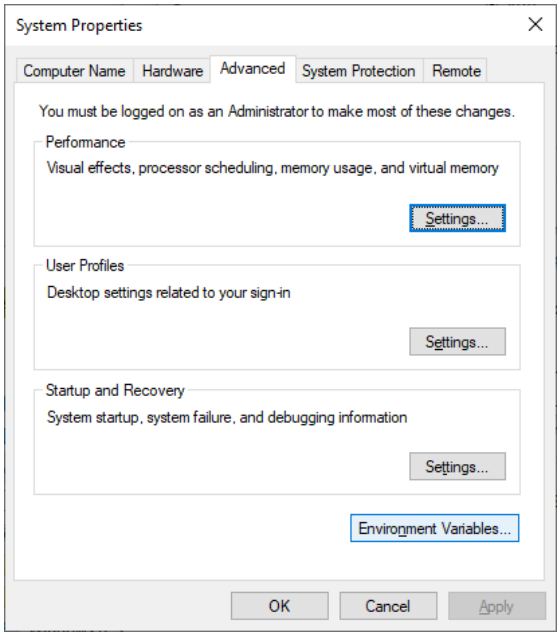
Valitaan System:



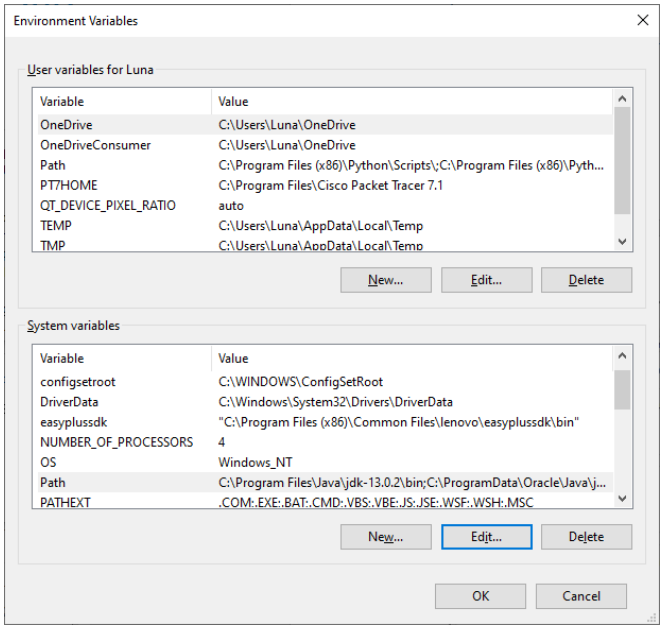
Valitaan Advance system settings:



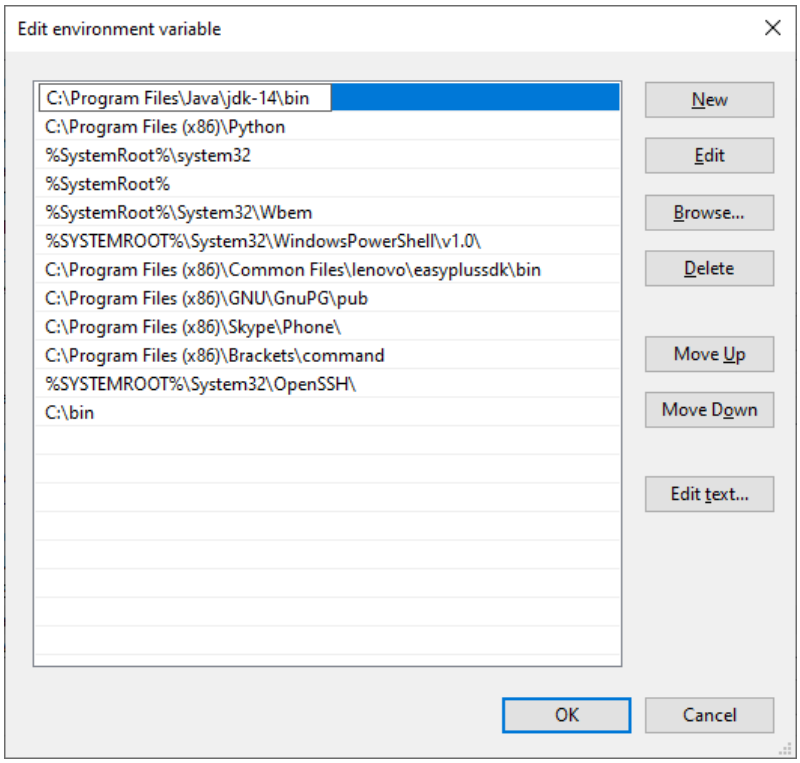
System properties -ikkunalta Advanced-välilehdeltä löytyy alhaalta Environment Variables:



Aukeavasta ikkunasta System variables -laatikosta löytyy Path. Valitsemalla sen ja Edit pääsee muokkaamaan listaa.

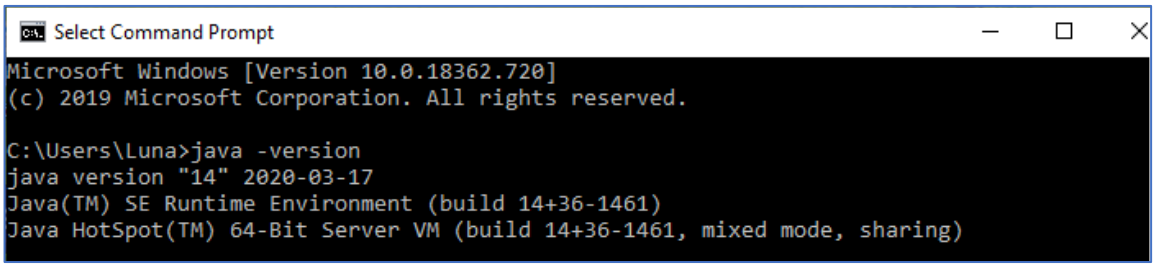


Valitsemalla ensin New ja sitten Browse pääsee valitsemaan halutun ohjelman kansion/polun tietokoneelta. Valitsemalla Move Up lisättyjä polkuja saa siirrettyä ylöspäin.



Poistutaan valitsemalla ok aukaistuihin ikkunoihin. Lisäyksen onnistuminen ja oikean polun valinta voidaan varmistaa komentorivin kautta, tässä tapauksessa esimerkiksi komennolla:

```
java -version
```



Jos polun lisäys ei ole onnistunut, komentorivin kautta tulee virheviesti.

4 Päivitys- ja poisto-ohjeet

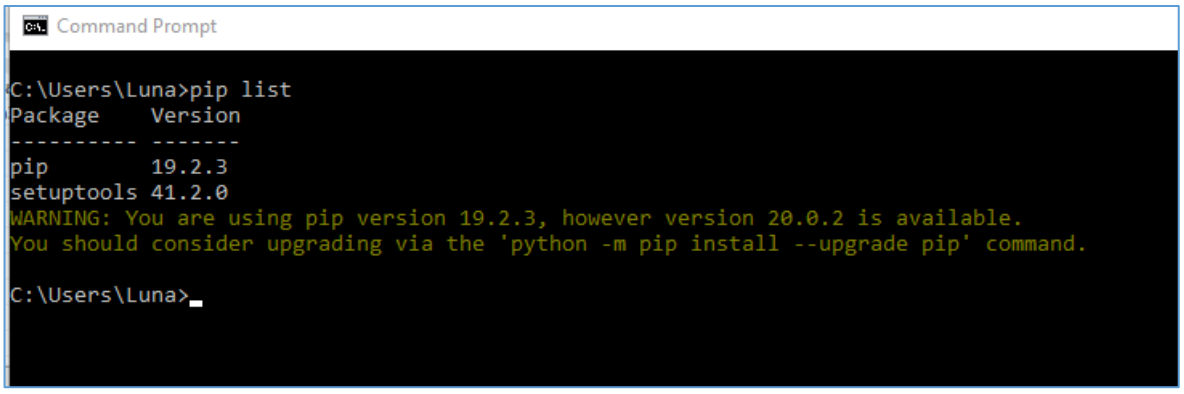
Käydään läpi ohjelmien päivitys- ja poisto-ohjeet.

4.1 Python

Ladataan uusin versio Python.org -sivustolta ja ajetaan asennustiedosto. Python Installer tunnistaa, jos on aiempi versio asennettuja sekä ehdottaa päivitystä. Pythonin voi poistaa Windowsilla ohjelmien / sovellusten poistamisen kautta.

4.2 Pip

Komentorivillä `pip list` ja `pip –version` -komennoilla saa esiin käytössä olevan version. Pip kertoo, kun päivitys saatavilla.



Päivityskomento löytyy https://pip.pypa.io/en/stable/installing/#upgrading-pip -sivustolta ja on:

```
python –m pip install –U pip
```

Poisto löytyy https://pip.pypa.io/en/stable/reference/pip_uninstall/

```
pip uninstall pip
```

4.3 Ajuri

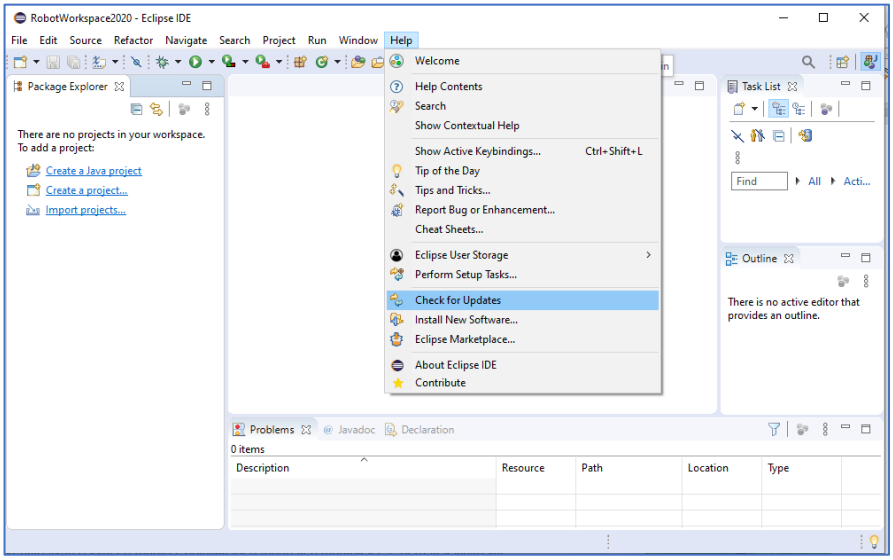
Päivitys tapahtuu lataamalla uusin versio ja viemällä se entisen tilalle samaan kansioon. Ajurin poisto onnistuu poistamalla se kansiosta.

4.4 Java

Ennen uuden version asennusta suositellaan poistamaan vanhat versiot, Windowsilla joko käyttöjärjestelmän ohjelmien / sovelluksien poistamisen kautta tai Javan poistotyökalulla.

4.5 Eclipse

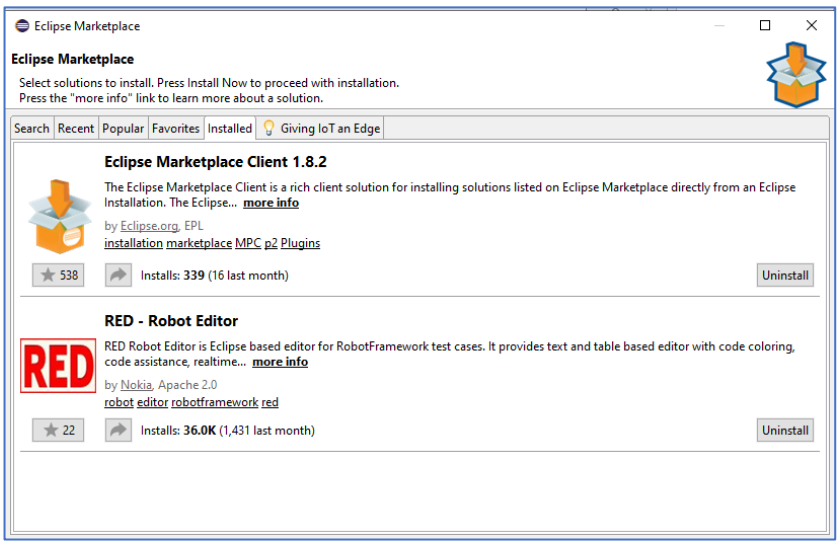
Päivittäminen onnistuu menemällä Eclipsen Help-alasvetovalikosta Check for updates.



Eclipselle ei ole poisto-ohjelmaa eikä Windows'n ohjelmien/sovelluksien poisto tunnista eclipseä. Suosituksena on poistaa Eclipsen kansio ja mahdolliset pikakuvakkeet tietokoneelta.

4.6 RED

Päivitys ja poisto onnistuu Eclipsen Marketplacen kautta Installed-välilehdellä:



Robot Framework käyttöohje

Versio 1

25.4.2020

Sisällys

Versionhallinta

Päivä	Versio	Kuvaus	Tekijä
15.10.2019	0.1	Luonnos	Tiina Anttila
5.4.2020	0.2	Testaus ja täydennys	Tiina Anttila
10.4.2020	0.3	Täydennys	Tiina Anttila
25.4.2020	1.0	Viimeistelty versio	Tiina Anttila

1 Johdanto

Tämän on ohje Robot Framework selaintestien tekoon Eclipse-pohjaisella RED editorilla. Testejä voi tehdä tekstieditoreilla, mutta on helpompi käyttää Robot Frameworkin syntaksin tuntevaa editoria. Tässä linkkejä oppaisiin:

Robot Framework käyttöopas:
http://robotframework.org/robotframework/#user-guide

Selenium-kirjaston käytettävissä olevat avainsanat ja niiden toiminnot:
https://robotframework.org/SeleniumLibrary/SeleniumLibrary.html

RESTinstance-kirjaston dokumentointi:
https://asyrjasalo.github.io/RESTinstance/

RED-editorin ohjeet:
http://nokia.github.io/RED/help/index.html

RED-editorin pikanäppäinlista:
http://nokia.github.io/RED/help/keys.html

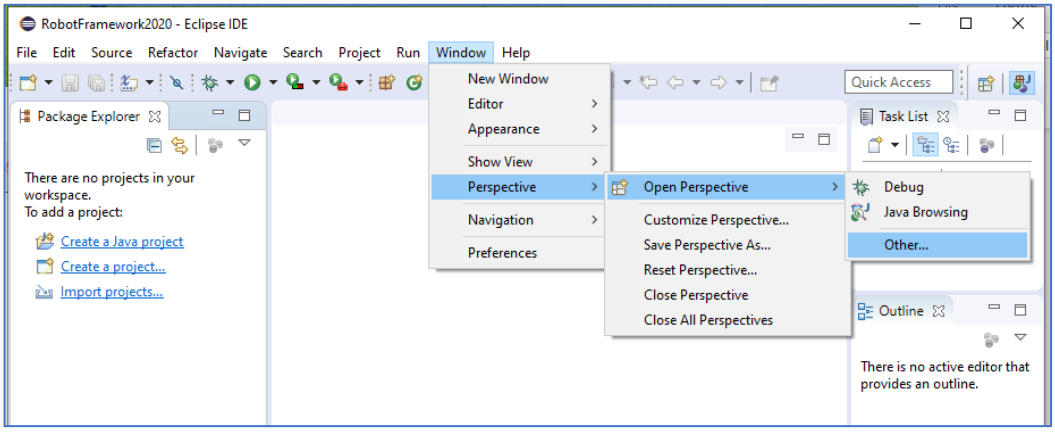
Kappaleessa kaksi käydään läpi projektin luonti, kappaleessa kolme testien luonti, kappaleessa 4 testien ajo ja kappaleessa 5 Robot Frameworkin tuottamat raportit.

2 Robot Framework -projektin luonti

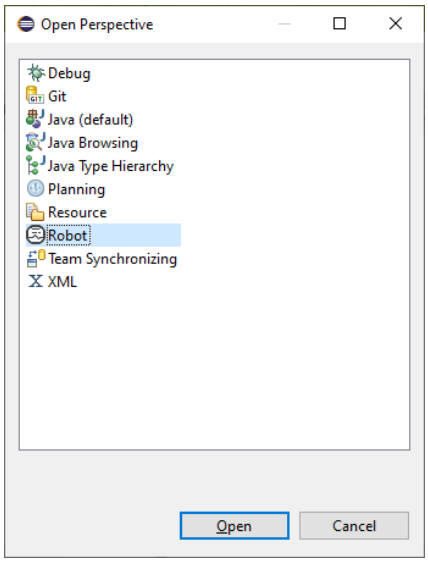
Asennuksen yhteydessä on luotu Eclipsen työtila, jonne luodaan testauksen projekti.

2.1 Projektin luonti

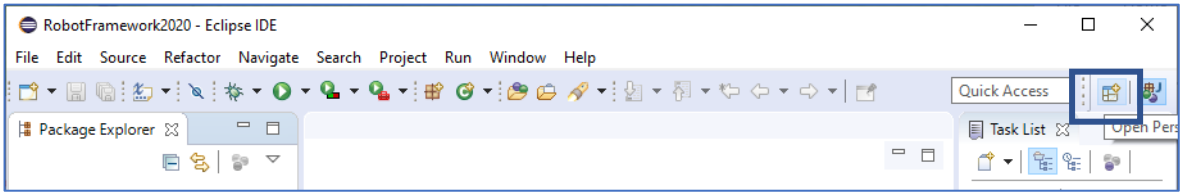
Avataan Robot Framework -perspektiivi valitsemalla Window-valikosta Perspective, josta valitaan Open Perspective ja Other.



Tästä aukeaa Open Perspective ikkuna, josta valitaan Robot.

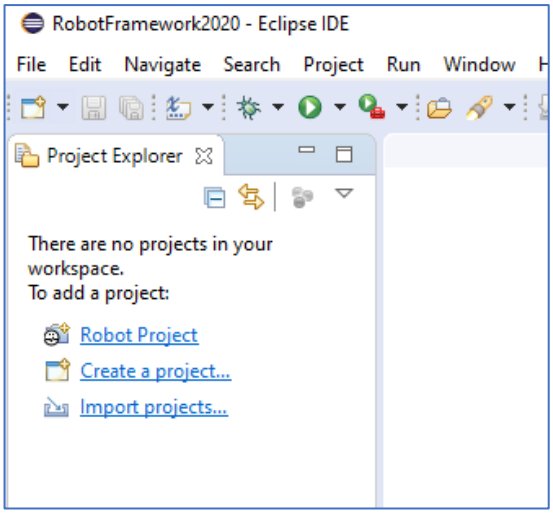


Saman Open Perspective -ikkunan saa auki oikeasta ylänurkasta  -kuvakkeesta.

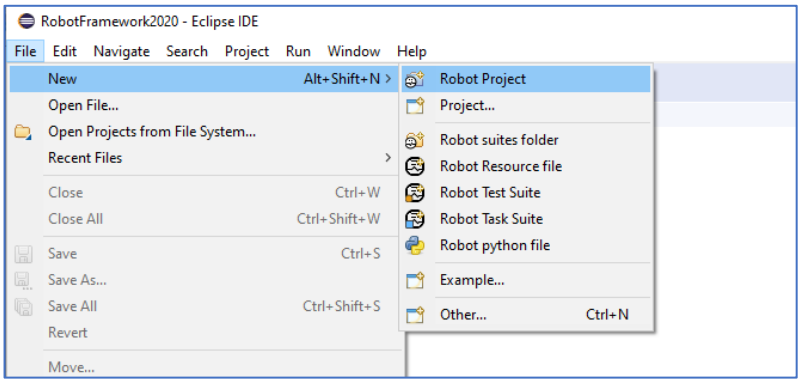


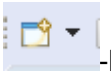
Tämän jälkeen Eclipsen valikoista löytyy Robot project -vaihtoehdot.

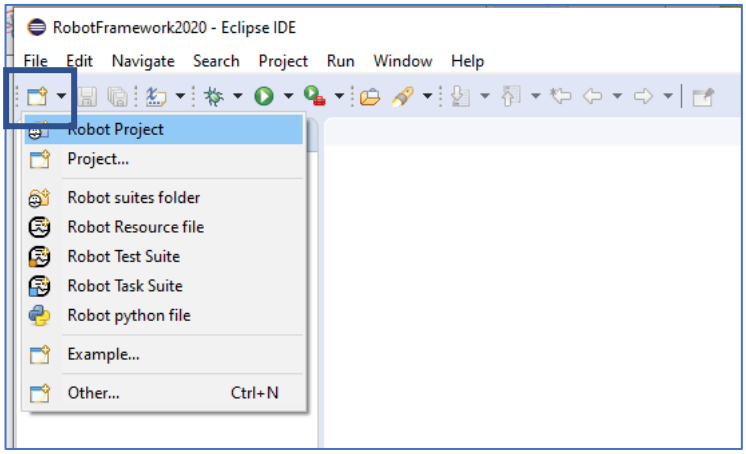
Uuden projektin voi aloittaa Project Explorer -osion suoraan ehdottamasta linkistä:



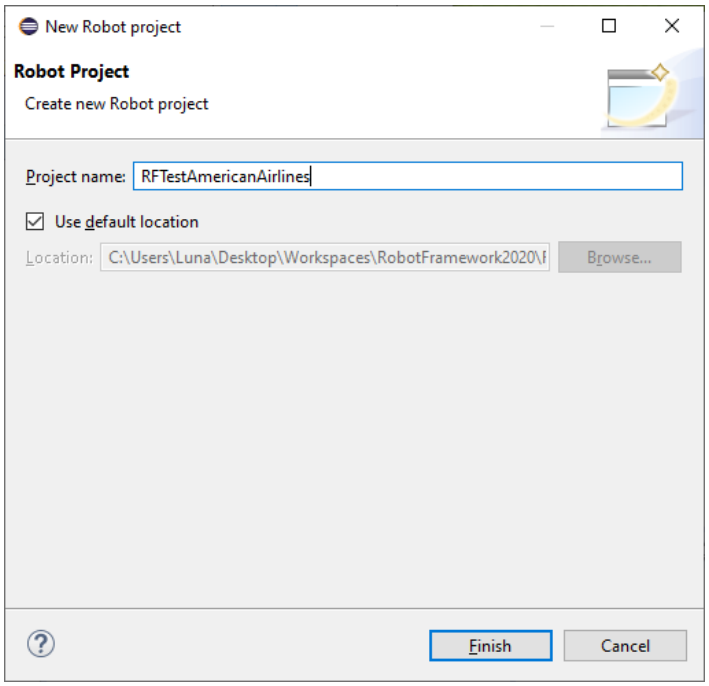
Toinen vaihtoehto on valita File-valikosta New ja Robot Project:



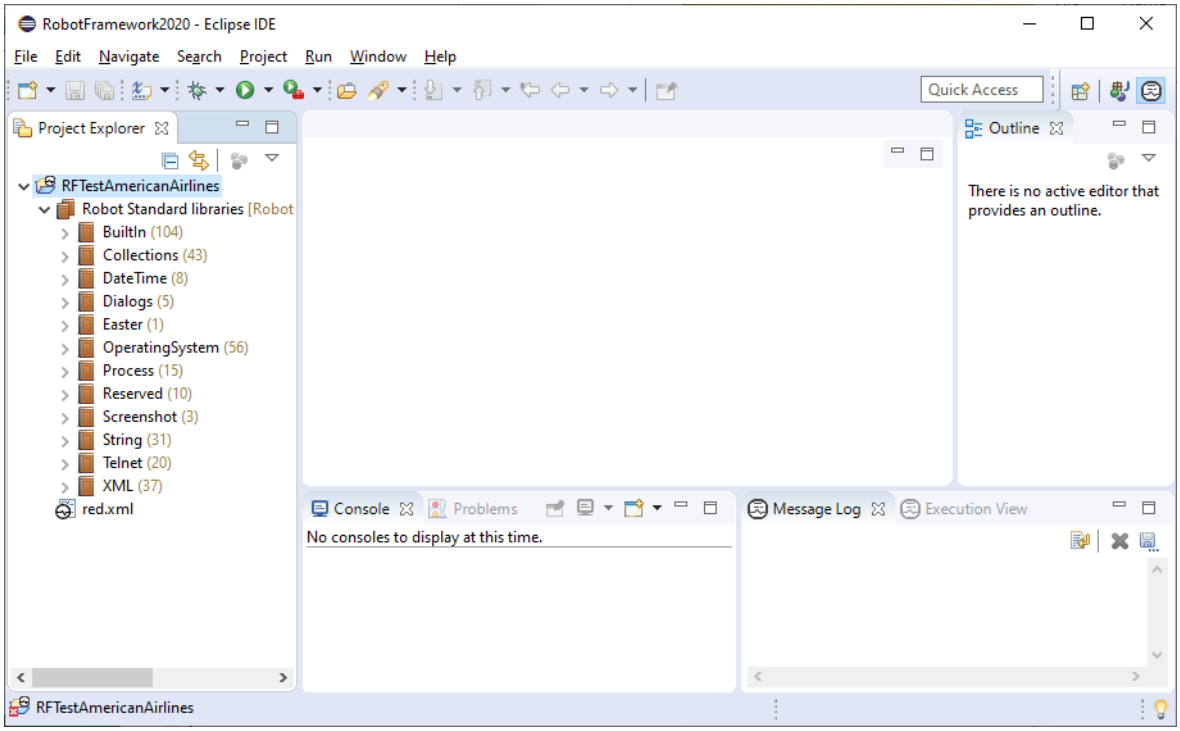
Kolmas vaihtoehto on  -kuvakkeen kautta kolmiosta aukeavasta valikosta:



Projekti nimetään kuvaavasti:



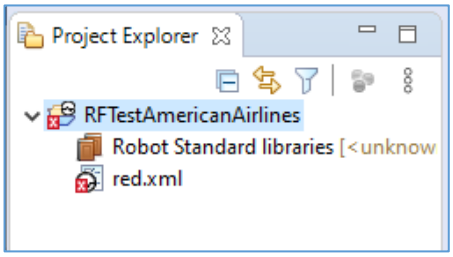
Onnistuneesti luotu projekti näyttää tältä:



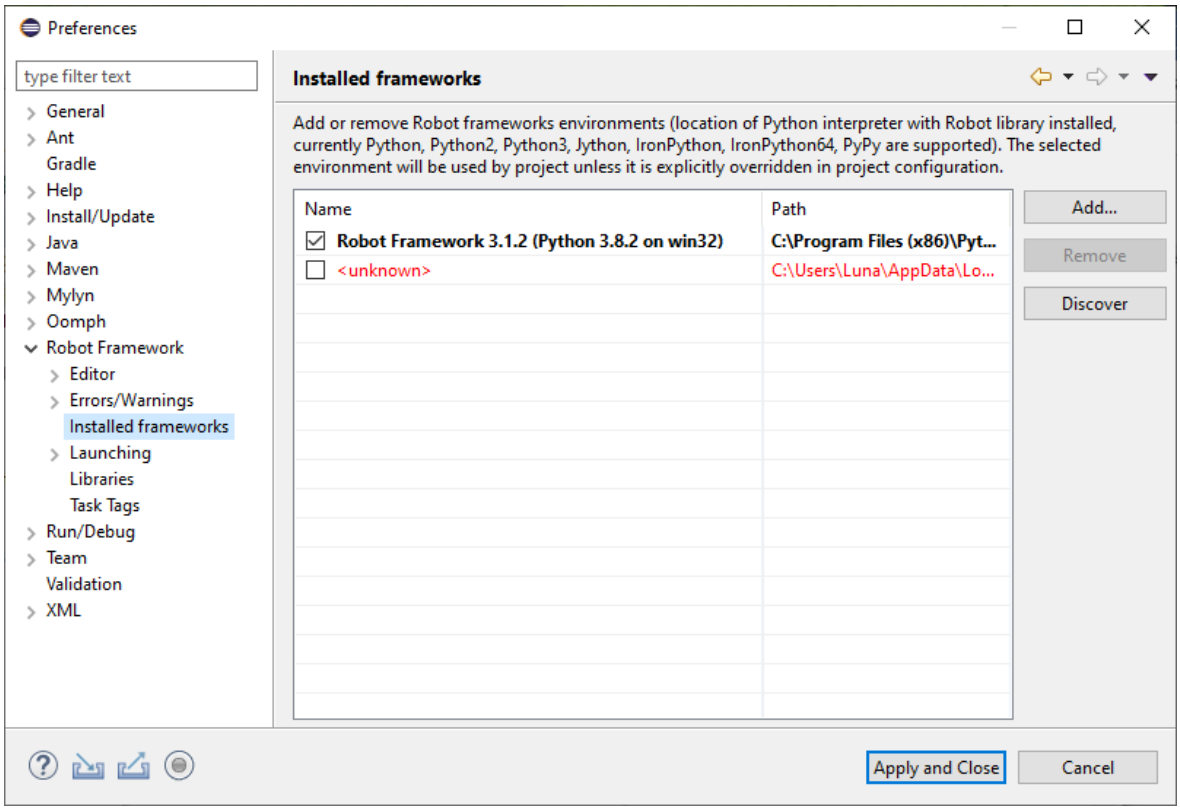
Kun projekti luotu onnistuneesti, näkyvät Robot Frameworkin kiinteät kirjastot projektin alla.

2.2 Projektin luonnin jälkeen virhe

Jos projektin kirjastojen kohdalle virheviesti <unknown>, eivätkä Robot Frameworkin kirjastot aukea, niin luultavasti Python ei löydy suoraan käyttöön.

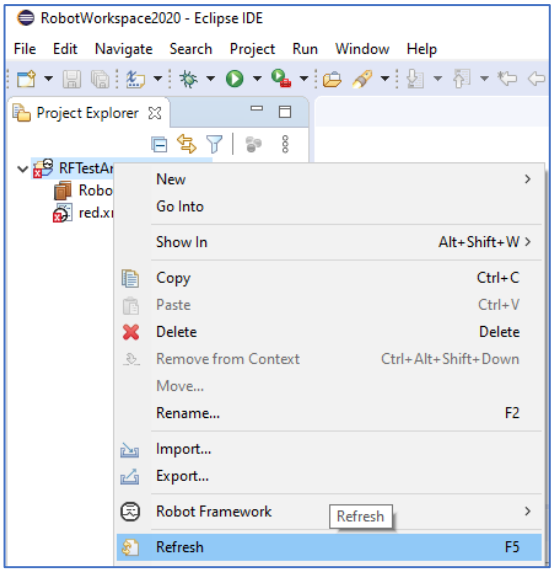


Tällöin Python ei ole oikein Environmental variable Path-listassa tai Eclipsen käytössä on väärä ja/tai vanha tieto. Eclipsen puolella tämän voi tarkistaa polusta Window -> Preferences -> Robot Framework -> Installed frameworks. Installed frameworks -näkymässä näytetään mitä on käytössä. Jos sijainti on vanhentunut tai poistettu, niin tässä näkyy listalla nimi ja vanha kansiopolku virheilmoituksen kera. Sen voi poistaa ja lisätä uuden.

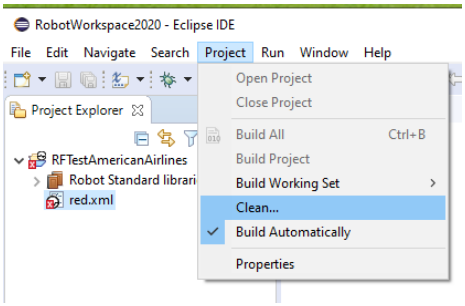


Jos lista on tyhjä eikä siinä näy valmiiksi yllä lihavoituna olevan kaltaista kohtaa, niin Add -painikkeen kautta pystyy lisäämään tietokoneelta python.exe kansiopolun. Tätä kautta pystyy hakemaan tietokoneelta Python kansion. Sen jälkeen valitaan Apply and Close sekä vahvistetaan valinta.

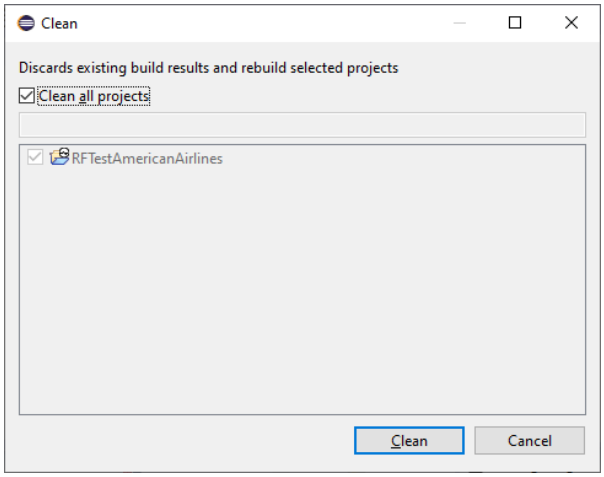
Muutosten jälkeen voi näkyä hetkellisesti virhekolmio tai -ruksi, jolloin voi kokeilla päivittää tai siivota projektin. Päivittäminen onnistuu helposti klikkaamalla projektin kohdalla hiiren oikeaa painiketta ja valitsemalla Refresh tai painamalla F5-painiketta.



Vastaavasti voi projektin voi siivota Project-osiosta Clean-toiminnolla.



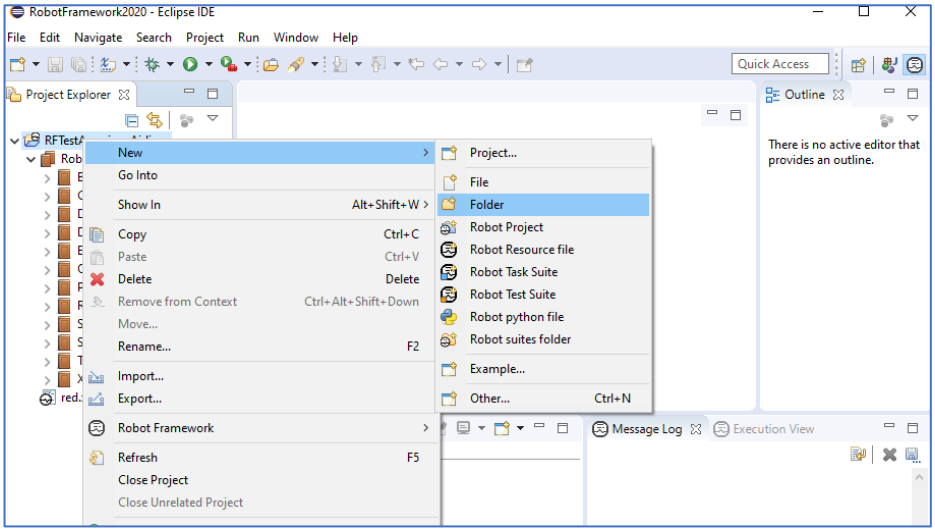
Varmistetaan, että oikea projekti on valittuna, jos työtilassa useampia projekteja.



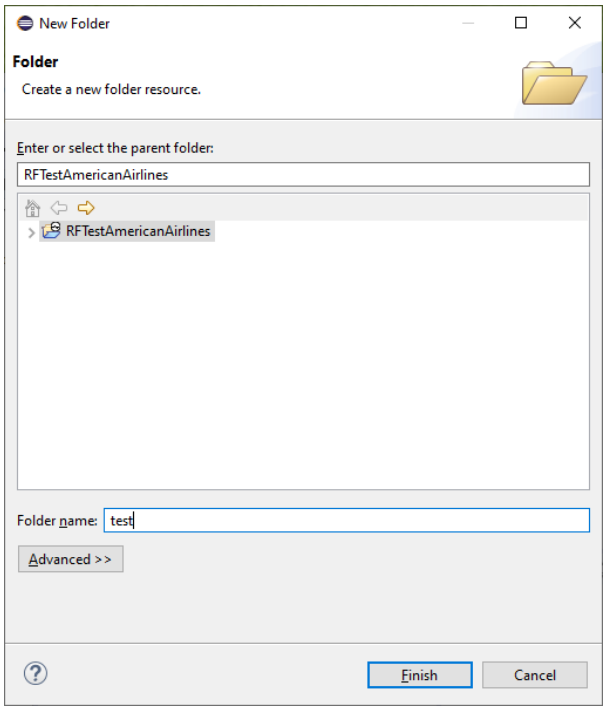
2.3 Testisetin luonti

Ensin luodaan kansio, jonne tehdään testisetit. Kansion tekeminen ei ole välttämätöntä, mutta se voi helpottaa, jos on erilaisia projekteja samassa Eclipse Workspacessa.

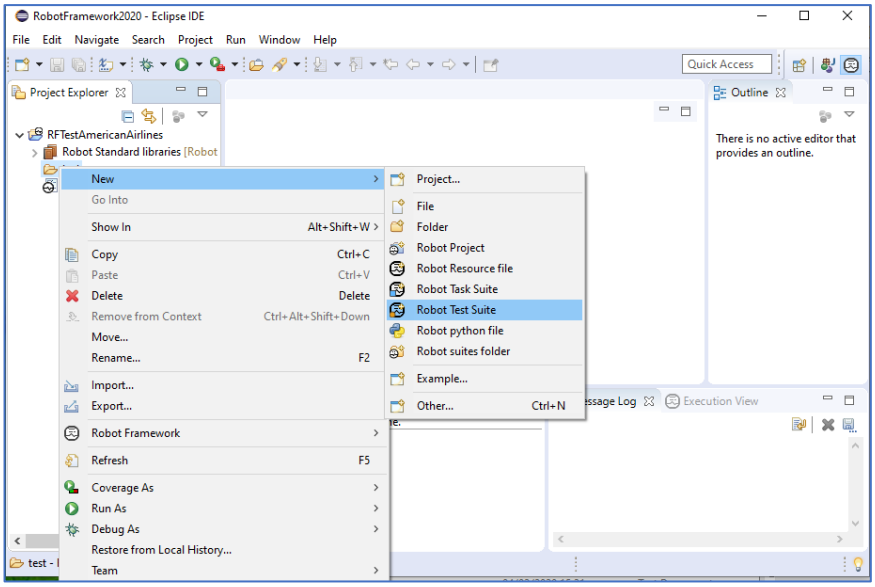
Kansion luonti onnistuu klikkaamalla projektia hiiren oikealla painikkeella, valitsemalla New ja Folder.



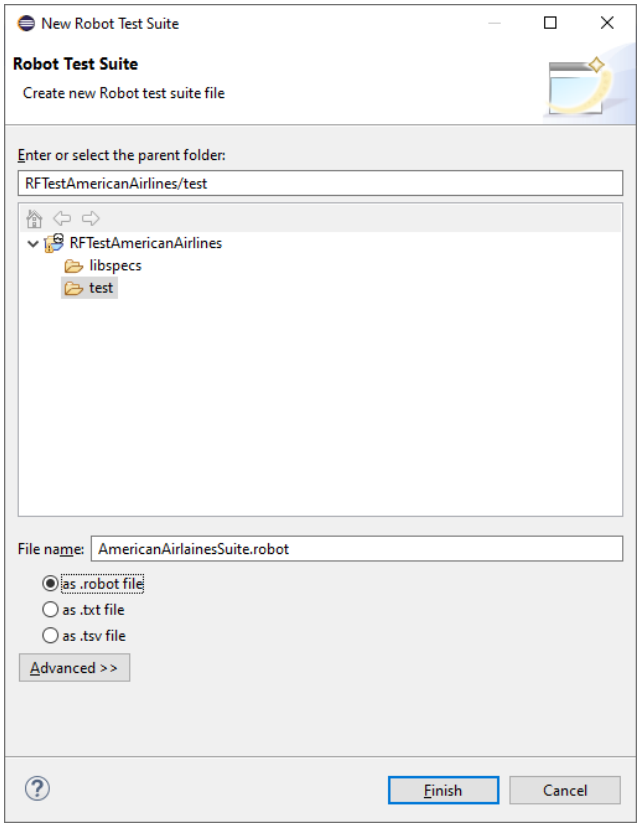
Valitaan minkä kansion alle uusi kansio tehdään ja nimetään se:

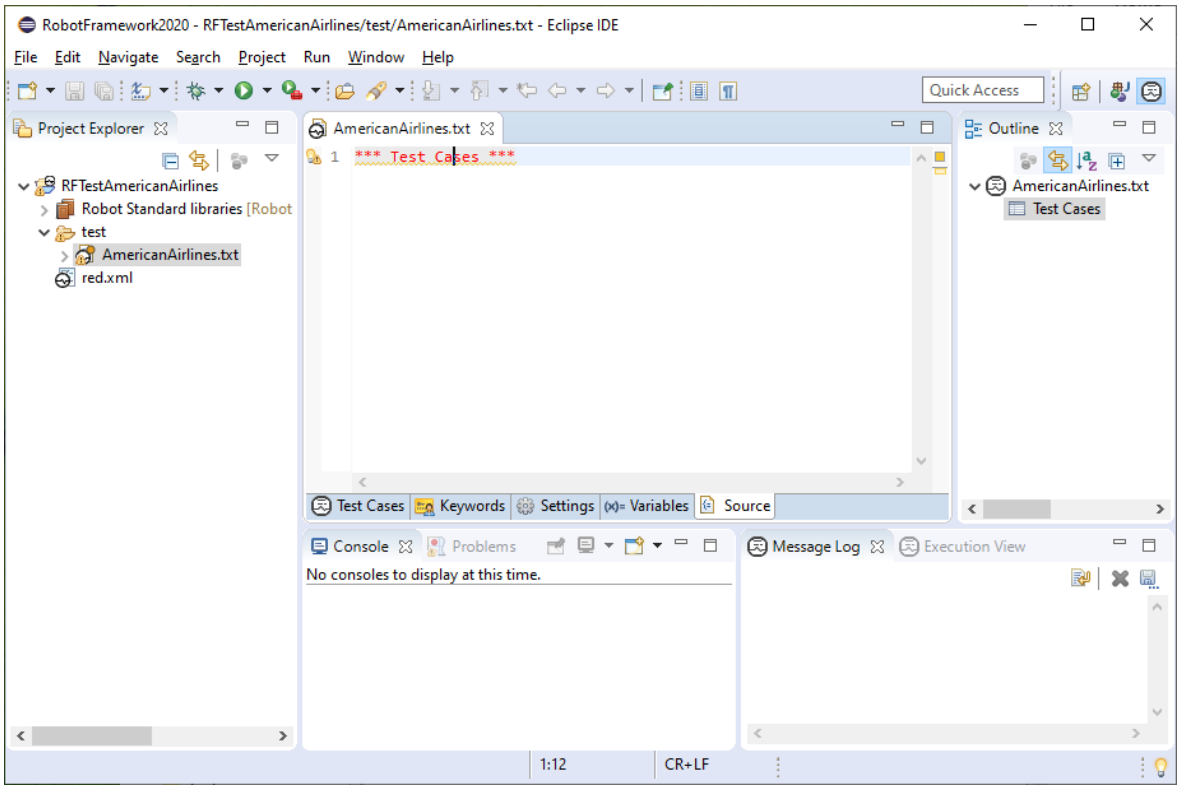


Kansioon luodaan testisetti klikkaamalla kansiota hiiren oikealla painikkeella, valitaan new ja Robot Test Suite.



Annetaan testisetille kuvaava nimi.





Testisetin luonnin onnistuttua näkyviin tulee testisettitiedosto, jossa otsikko on luotu automattisesti. Tähän pystyy tekemään testejä.

3 Testien luonti

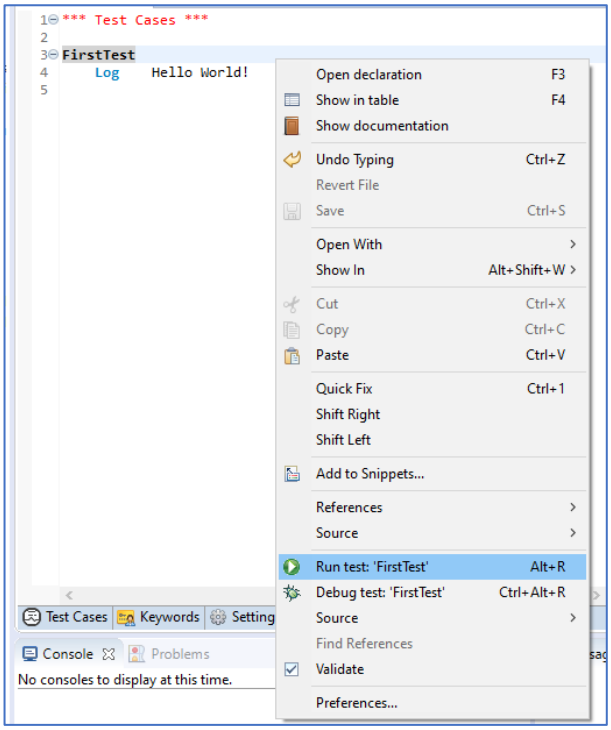
Ensin käydään läpi suppean testin teko ja sen ajaminen. Sitten käydään läpi mitä avainsana tarkoittaa, Robot Framework testin rakenne ja viimeisenä tehdään kattavampi selaintesti.

3.1 Yksinkertainen testi ja sen ajo

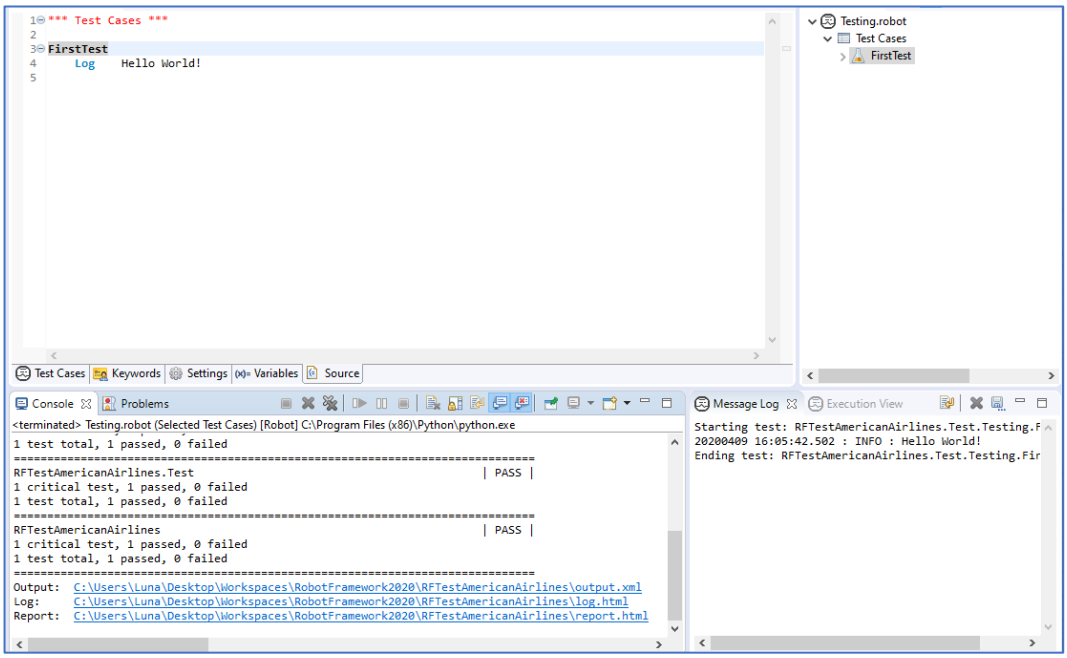
Ensimmäinen testi voi olla yksinkertainen, jolla kokeillaan RED-editorin ja Robot Frameworkin toimintaa. Esimerkiksi:

```
*** Test Cases ***

FirstTest
    Log    Hello World!

```

Testin nimeksi on annettu FirstTest ja annettu Robot Frameworkin avainsana Log, jolle arvo ”Hello World!”. Testisetin ja RF:n asetukset voi testata ajamalla testin. Testin pystyy ajamaan helposti menemällä halutun testin päälle ja painamalla hiiren oikeaa painiketta ja valitaan tässä tapauksessa Run test: FirstTest.



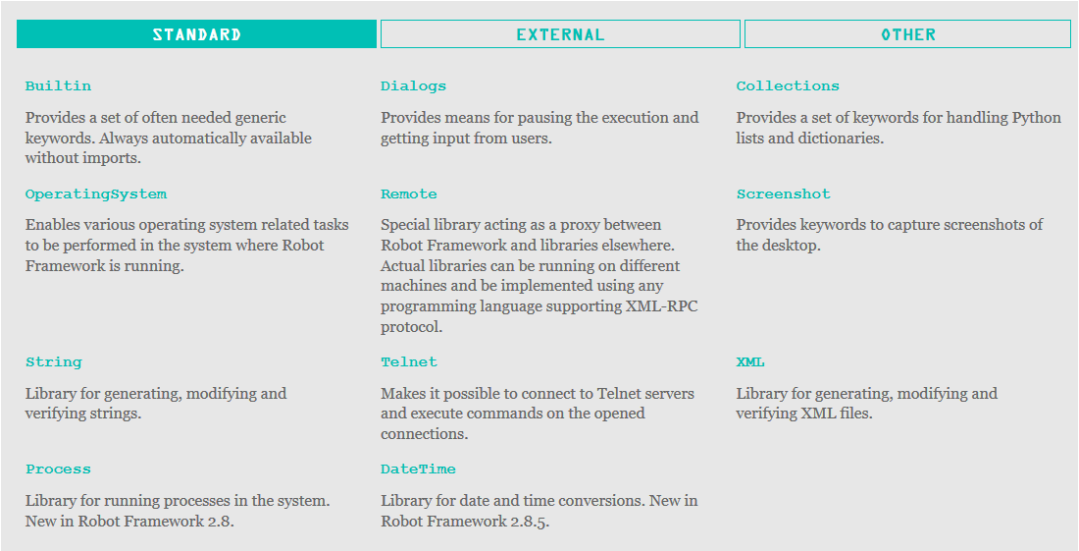
Testiajon onnistuessa näkymä on seuraavanlainen:



Konsolissa näkyy testiajon tapahtumat ja kooste. Testiajosta muodostetaan myös Output-, Log ja Report -tiedostot, joihin konsolissa linkit. Tämän testin ajossa voi tulla virhe esimerkiksi silloin, jos Eclipsestä on uudempi versio käytössä, jota ei ole testattu RED editorin kanssa. Silloin esimerkiksi testin ajo ei välttämättä edes käynnisty. Virheilmoituksesta löytyy yleensä Details-osion kautta lisätietoa ja voi tarkistaa missä vika. Jos ajo lähtee liikkeelle ja epäonnistuu, niin log/report-linkkien kautta pystyy tarkastelemaan missä virhe tapahtuu.

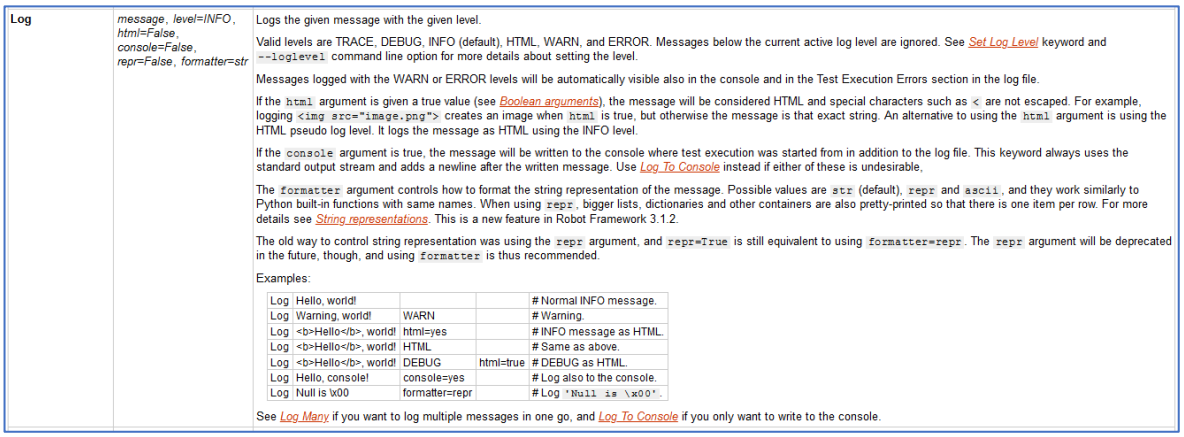
3.1 Avainsana

Robot Framework on avainsanapohjainen testauskehys ja testit muodostetaan avainsanoilla. Avainsanojen toiminnallisuus on tehty kirjastoissa. Robot Framework sisältää itsessään käytettävissä olevia kirjastoja ja siihen voi tuoda ulkoisia kirjastoja helposti eri käyttötarkoituksia varten. Kirjastot ja niiden kuvaukset löytyvät Robot Frameworkin sivujen kautta.

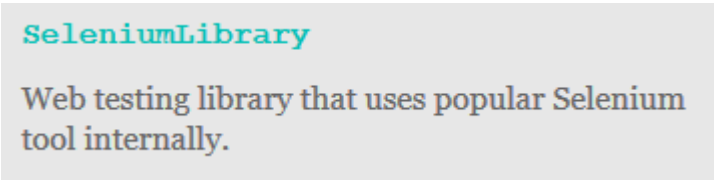


Tätä kautta voi mennä tutkimaan kirjaston käyttötarkoituksia sekä avainsanoja ja niiden käyttötarkoituksia. Standard-osion kirjastot ja työkalut ovat valmiina käytettäviksi ja External- sekä Other-osioiden kirjastot ovat otettavissa käyttöön.

Esimerkiksi avainsana Log on Robot Frameworkin sisäisen BuiltIn-kirjaston avainsana ja se on kuvattu seuraavasti:

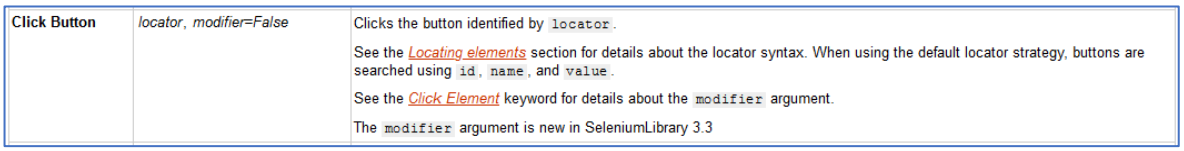


Esimerkiksi SeleniumLibrarysta on Robot Frameworkin sivuilla seuraava kohta:



Robot Frameworkin sivuilta ohjataan SeleniumLibraryn GitHub-sivuille, josta löytyy esimerkiksi Selenium-kirjaston esittely, avainsanadokumentaatio sekä asennuksen ja ajurien dokumentit.

Avainsanadokumentaatiosta löytyy esimerkiksi painikkeen painamisen toiminto:



Click Button	*locator*, *modifier=False*	Clicks the button identified by `locator`. See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, buttons are searched using `id`, `name`, and `value`. See the *Click Element* keyword for details about the `modifier` argument. The `modifier` argument is new in SeleniumLibrary 3.3

Click Button -avainsanan lisäksi tarvitaan painikkeen tunniste, joka voi olla id, nimi tai arvo. Verkkosivujen toiminnallisuuksiin tarvittavat tiedot löytyvät suhteellisen helposti verkkosivuja tarkastellessa developer tools -työkaluilla. Eli jos painikkeen ID on ”SearchButton”, saa sitä painettua seuraavalla komennolla:

Click Button SearchButton

Robot Framework ymmärtää tekstin välilyönnin kanssa yhdeksi kokonaisuudeksi, joten ”Click Button” on yksi avainsana. Avainsanan jälkeen tulee väli tabulaattorilla, jolloin Robot Framework käsittelee seuraavan tekstin omana kokonaisuutenaan.

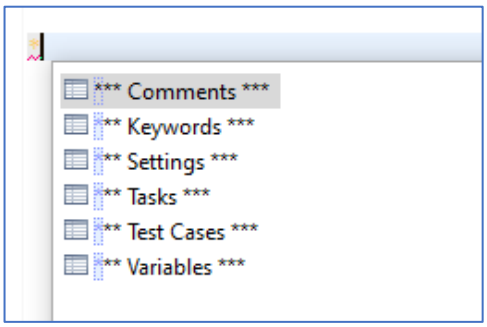
3.2 Testisetin rakenne

Testisetin rakenne on neljäosainen ja voidaan kirjoittaa tai luoda RED-editorin välilehtien kautta. Testisettitiedostossa osioiden otsikot merkitään kolmella tähdellä ennen ja jälkeen tekstin:

Settings
Variables
Test Cases tai ***Tasks***
Keywords
Comments

Kaikkia osioita ei tarvitse käyttää, jolloin osio otsikoineen jätetään tekemättä. Osiot ovat RED editorilla omia osioita/välilehtiä ja näiden lisääminen sekä muokkaus onnistuu sekä Source-editorin kautta, että näillä välilehdillä.

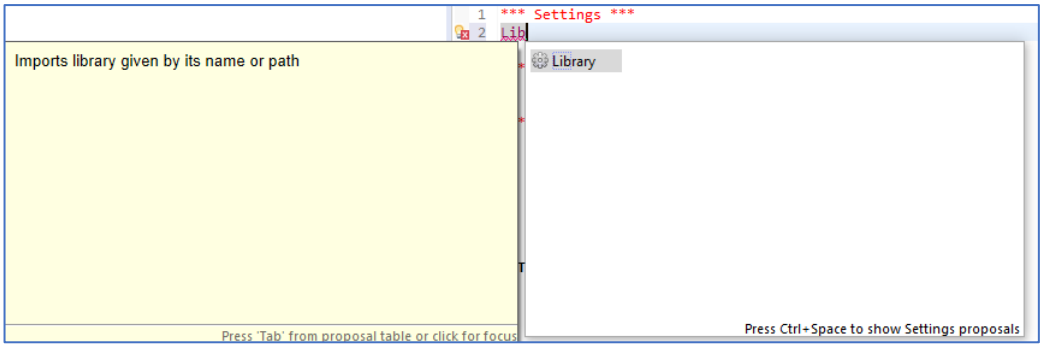
Editorissa kannattaa käyttää kautta linjan pikanäppäinyhdistelmää Ctrl+space, jolla saa kirjoitettavaan kohtaan ehdotuksia kirjoittamansa alun mukaan. Tässä kirjoittamalla tähden ja käyttämällä yhdistelmää saa ehdotuksena kaikki osioiden otsikot ja voi valita lisättävän:



Settings

Settings-osiossa tuodaan testikirjastot, resurssitiedostot ja muuttujien tiedostot.

```
*** Settings ***
Documentation     Chrome test example using SeleniumLibrary.
Library           SeleniumLibrary

```



Lisäksi määritetään metatietoa kuten vapaata tekstiä sekä Tag-tietoja. Tag-tieto on yhdelle tai useammalle testisetille annettava tieto, jota voidaan käyttää esimerkiksi merkitsemään kriittiset testisetit ja ajaa ne erikseen.

Settings-osiossa on myös mahdollista määritellä testisetin aloitus- ja lopetustoimia, esimerkiksi avaa ja sulje selain ennen/jälkeen testin/testisetin:

Suite Setup - testisetin aloitustoimi, joka suoritetaan ennen jokaista testisettiä
Test Setup – testin aloitustoimi, joka suoritetaan ennen jokaista testiä
Suite Teardown – testisetin lopetustoimi, joka suoritetaan jokaisen testisetin jälkeen
Test Teardown – testin lopetustoimi, joka suoritetaan jokaisen testin jälkeen

Variables

Variables-osiossa määritellään muuttujat, joita voidaan käyttää muualla testidatassa. Yleisin tapa käyttää muuttujia, on käyttämällä ${} (scalar syntax). Nämä ovat yleensä String-muotoisia, mutta voivat käytännössä sisältää minkä tahansa objektin esimerkiksi numeroita, listoja tai sanakirjoja.

```
*** Variables ***
${START URL}    https://www.americanairlines.fi/intl/fi/index.jsp
${BROWSER}    Chrome
${DELAY}    1
${FROM}    NYC
${TO}    ORD
${DAY FROM}    15.06.2020
${DAY RETURN}    15.06.2020
```

Test Cases -osiossa kirjoitetaan testit, jotka luodaan käytettävissä olevilla avainsanoilla. Nämä voivat olla Robot Frameworkin sisäisten kirjastojen, käyttöön tuotujen ulkoisten kirjastojen tai itse tehtyjä avainsanoja.

```
*** Test Cases ***
FirstTest
    Log    Hello World...

TravelSearch
    Open Browser To Start Page
    Accept Cookies
    Input From    ${FROM}
    Input To    ${TO}
    Input Day From    ${DAY FROM}
    Input Day Return    ${DAY RETURN}
    Search
    Results Page Should Be Open
    [Teardown]    Close Browser
```

Testille voidaan antaa myös lisätietoa samaan tapaan kuin itse testisetille:

[Documentation] This is additional information for a test case.

[Tags] This is a tag to mark a single test / group of tests

Lisäksi voidaan antaa testin lopuksi aina suoritettava osio, esimerkiksi

[Teardown] Close Browser

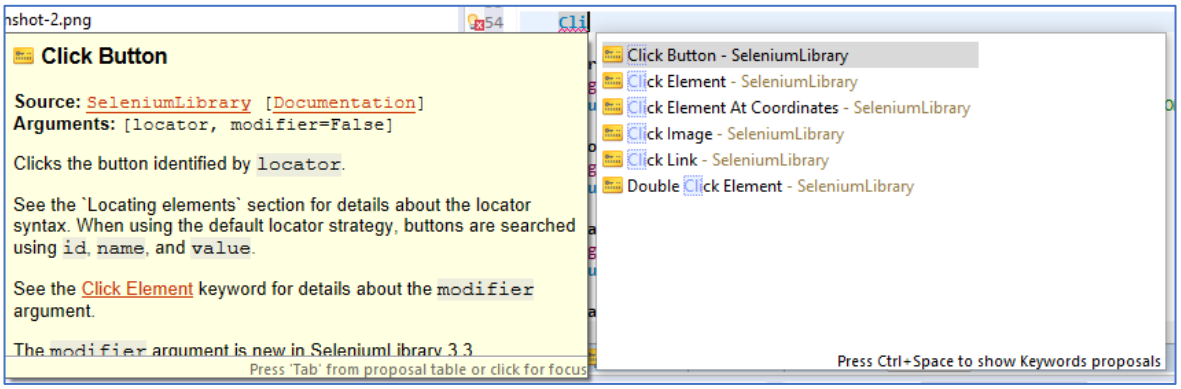
Tämä sulkee selaimen aina testin lopuksi, vaikka jokin aiempi testisteppi epäonnistuisi.

```
TravelSearchForTwo
    [Documentation]    Testing two adults
    [Tags]    Tag
    Open Browser To Start Page
    Accept Cookies
    Input From    ${FROM}
    Input To    ${TO}
    Select From List By Value    flightSearchForm.adultPassengerCount    2
    Input Day From    ${DAY FROM}
    Input Day Return    ${DAY RETURN}
    Set Selenium Speed    10
    Search
    Results Page Should Be Open
    [Teardown]    Close Browser
```

Keywords

Keywords-osiossa luodaan käyttäjän omia avainsanoja käyttäen alemman tason käytettävissä olevia avainsanoja. Jokin vaihe voi toistua useammassa testissä, joten sen voi kirjoittaa tähän avainsanaksi antamalla kuvaavan nimen ja kirjoittamalla sen alle käytössä olevilla avainsanoilla toiminnallisuuden. Näin itse testiin kirjoitetaan vain itse tehdyn avainsanan nimi.

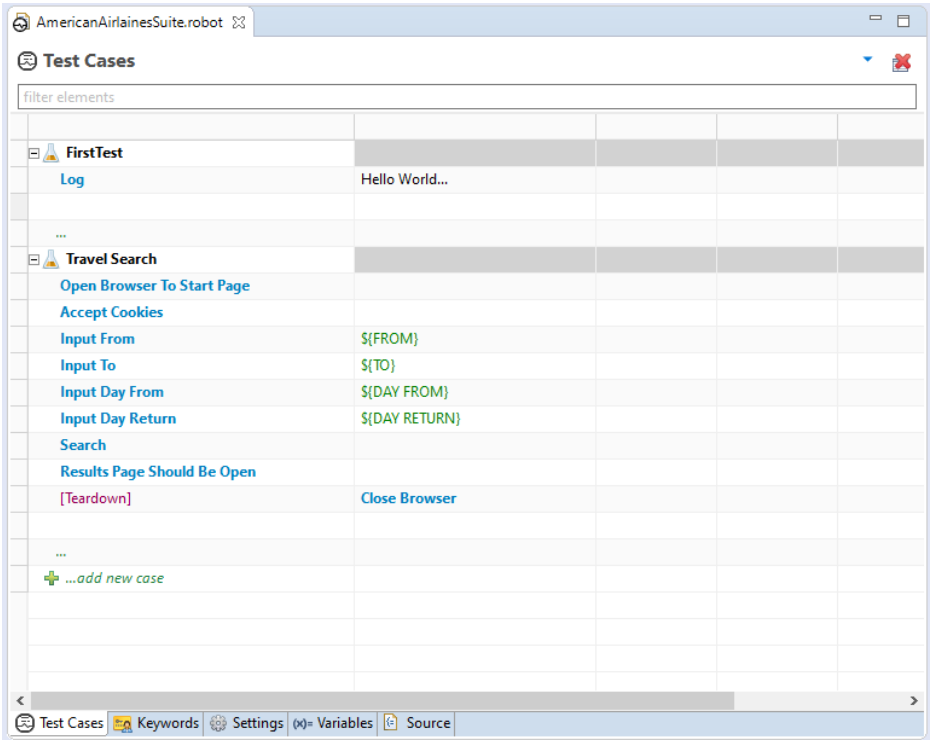
```
*** Keywords ***
Open Browser To Start Page
    Open Browser    ${START URL}    ${BROWSER}
    Set Selenium Speed    ${DELAY}
    Title Should Be    Airline Tickets and Airline Reservations from American Airlines | aa.com

Accept Cookies
    Click Button    optoutmulti_button
    Set Selenium Speed    ${DELAY}
```



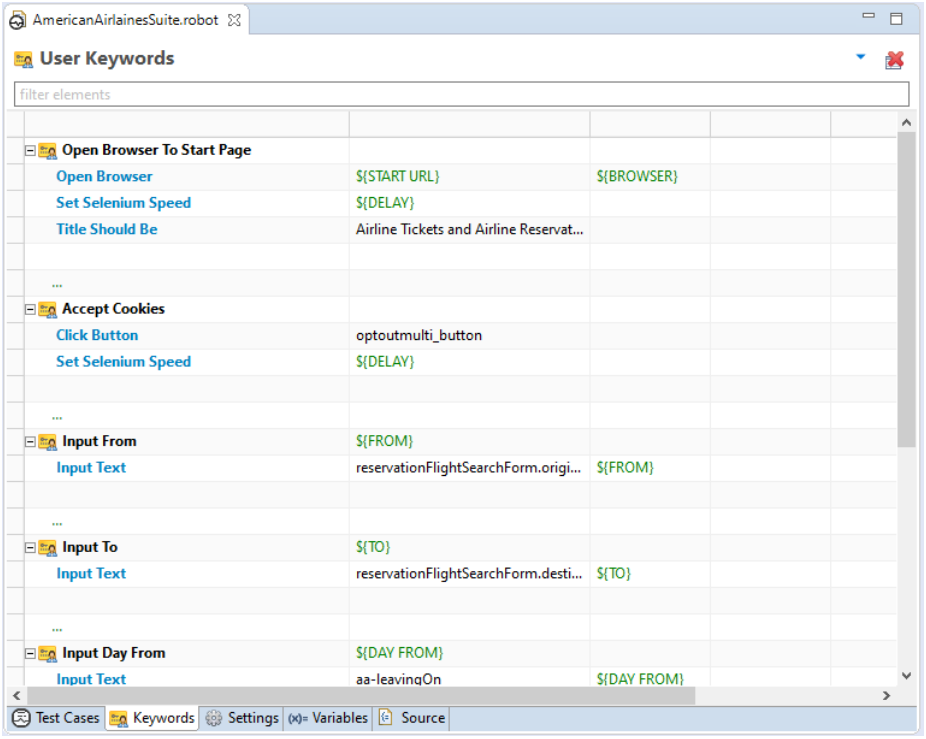
Lisäksi on mahdollista käyttää Tasks- ja Comments-osioita. Tasks-osiossa on mahdollista määritellä tehtäviä käytettävissä olevilla avainsanoilla, mutta tiedostossa voi käyttää vain joko testejä tai tehtäviä. Comments-osioon voi kirjoittaa kommentteja tai dataa, mitä Robot Framework ei käytä.

Eclipsen kautta osioita voi tarkastella ja muokata erikseen omilla välilehdillään.

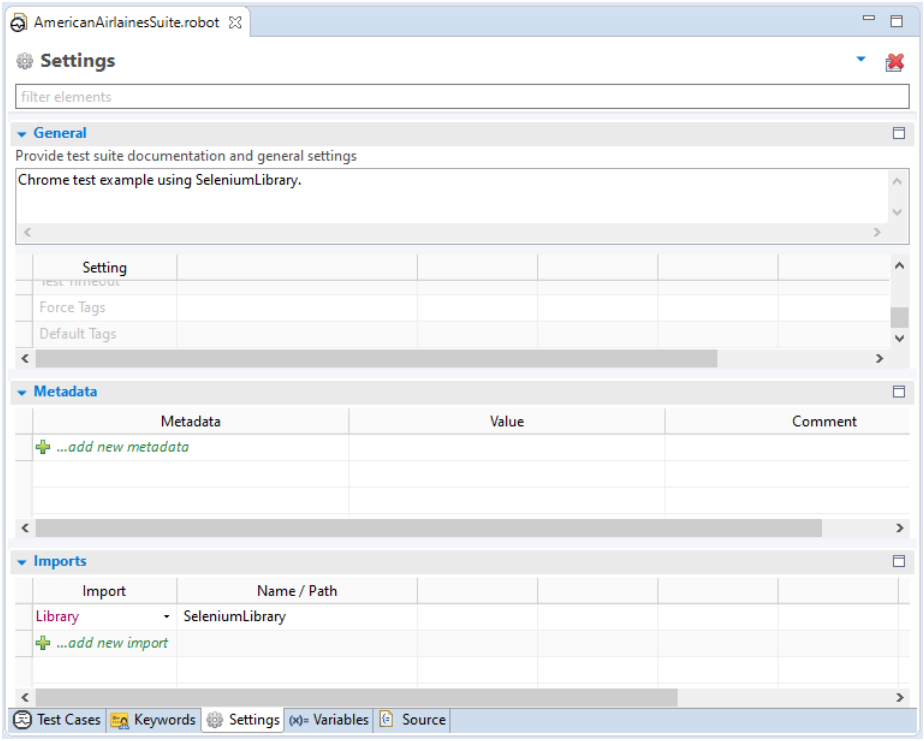
Test Cases eli testit:



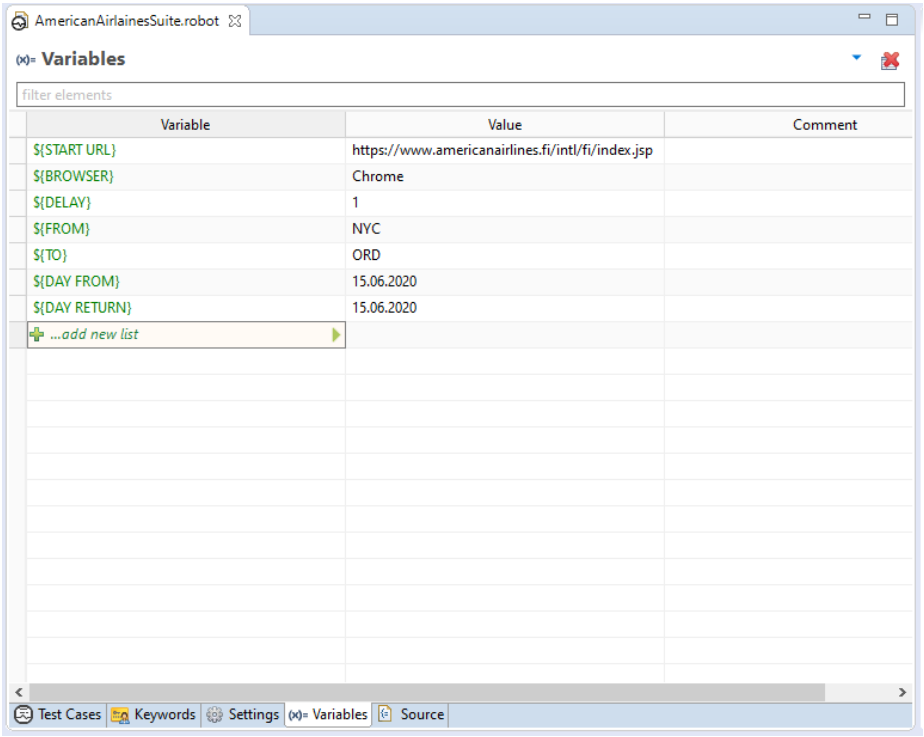
Keywords - avainsanat:



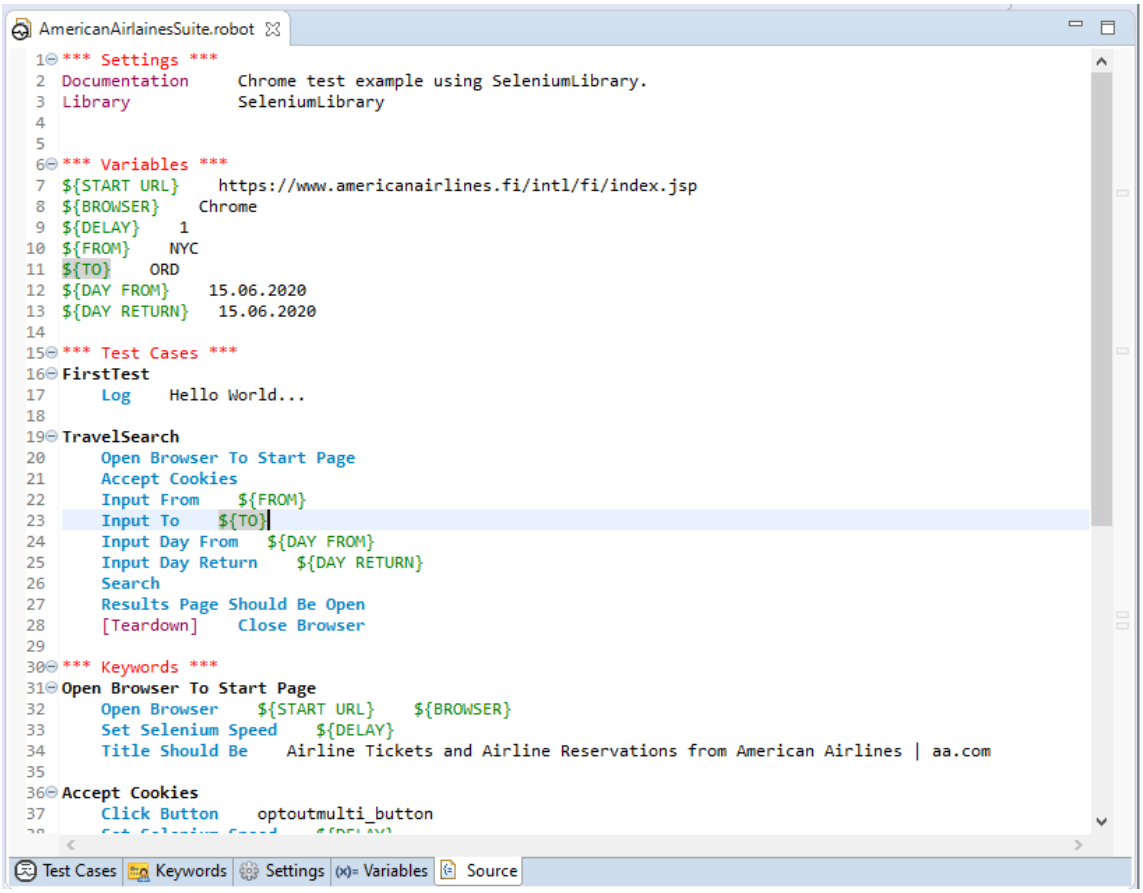
Settings - asetukset:



Variables:



Source -osiossa nämä osiot näytetään yhtenä kokonaisuutena ja tämä näkymä on sama kuin testisetin tiedosto. Tiedoston voi avata tavallisella tekstieditorilla kuten Notepad-ohjelmalla ja testiä pystyy muokkaamaan siinä, tosin ilman RED-editorin apukeinoja.



Kirjoitettaessa testiä millä tahansa editorilla erotetaan tietueet tab-välillä. Jos sanat on erotettu pelkällä välilyönnillä, nämä käsitetään yhdeksi tietueeksi.

3.3 Selaintestin kirjoitus

Selaimen testaamiseksi tuodaan käyttöön SeleniumLibrary-kirjasto. Lisätään samalla kuvaus testistä. Kirjoitetaan ***Settings***, seuraavalle riville Documentation, lisätään väli tab-painikkeella ja kirjoitetaan testin kuvaus. Seuraavalle riville kirjoitetaan Library ja lisätään väli tab-painikkeella ja kirjoitetaan tuotavan kirjaston nimi SeleniumLibrary.

```
*** Settings ***
Documentation     Chrome test example using SeleniumLibrary.
Library           SeleniumLibrary
```

Muuttujiin tuodaan kaikki tiedot, joita halutaan päivittää helposti yhdestä paikasta ja jotka ovat tiedoston testeille yhteiskäytössä. Esimerkiksi testissä halutaan avata selain tiettyyn osoitteeseen ja tehdä matkahaku. Muuttujat merkitään dollarin merkillä ja {}-sulkeilla. Sulkeiden sisään laitetaan muuttujan kuvaava nimi. Nimen jälkeen lisätään tab-painikkeella väli ja kirjoitetaan muuttujan sisältö eli se tieto, jota käytetään testeissä.

```
*** Variables ***
${START URL}    https://www.americanairlines.fi/intl/fi/index.jsp
${BROWSER}    Chrome
${DELAY}    1
${FROM}    NYC
${TO}    ORD
${DAY FROM}    15.06.2020
${DAY RETURN}    15.06.2020
```

Testien ja omien avainsanojen tekeminen menee käsi kädessä. Raportoinnin ja virheenjäljityksen kannalta testissä kannattaa olla selkeät stepit eli nähdään mikä käytännössä onnistuu tai epäonnistuu. Omiin avainsanoihin voi tehdä alistepit, joita ei halua kirjoittaa jokaiseen testiin erikseen. Esimerkiksi testiin riittää yhdellä rivillä tieto selaimen avaamisesta ja tämän saa aikaiseksi tekemällä oman avainsanan alle konkreettiset stepit selaimen avaamisesta haluttuun osoitteeseen ja latauksen odottaminen ja varmistus otsikon oikeellisuudesta. Itse testi:

```
Travel Search
    Open Browser To Start Page
    Accept Cookies
    Input From    ${FROM}
    Input To    ${TO}
    Input Day From    ${DAY FROM}
    Input Day Return    ${DAY RETURN}
    Search
    Results Page Should Be Open
    [Teardown]    Close Browser
```

Ja omat avainsanat:

```
*** Keywords ***
Open Browser To Start Page
    Open Browser    ${START URL}    ${BROWSER}
    Set Selenium Speed    ${DELAY}
    Title Should Be    Airline Tickets and Airline Reservations from American Airlines | aa.com

Accept Cookies
    Click Button    optoutmulti_button
    Set Selenium Speed    ${DELAY}

Input From
    [Arguments]    ${FROM}
    Input Text    reservationFlightSearchForm.originAirport    ${FROM}

Input To
    [Arguments]    ${TO}
    Input Text    reservationFlightSearchForm.destinationAirport    ${TO}

Input Day From
    [Arguments]    ${DAY FROM}
    Input Text    aa-leavingOn    ${DAY FROM}

Input Day Return
    [Arguments]    ${DAY RETURN}
    Input Text    aa-returningFrom    ${DAY RETURN}

Search
    Click Button    bookingModule-submit

Results Page Should Be Open
    Title Should Be    Valitse lennot - Lentohaun tulokset - American Airlines
```

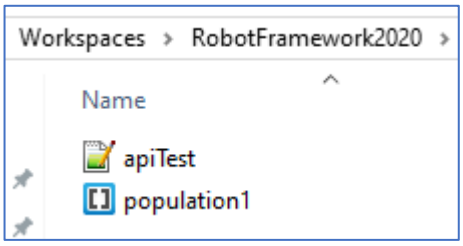
Omia avainsanoja ei ole pakko käyttää, mutta kannattaa esimerkiksi silloin, jos näitä toiminnallisuuksia haluaa käyttää useampaan testiin. Jos esimerkiksi hakuosoite tai verkkosivun otsikko tulee muuttumaan, niin helpompi päivittää vain kerran muuttujaan kuin jokaiseen testiin erikseen. Lisäksi muuttujat listataan testin yläosassa ja näitä on selkeämpi tarkastaa kuin etsiä missä kohdin itse testejä tai avainsanoja on päivitettävä tieto.

3.4 API-testin kirjoitus

API-testin kirjoitus seuraa samaa logiikkaa, mutta käytetään rajapintojen toiminnallisuuksia ja RESTinstance-kirjaston toiminnallisuuksia. Yksinkertainen testi on esimerkiksi hakea dataa ja tallentaa se tiedostona. Testisettiin tuodaan rajapintakirjasto käyttöön ja tehdään testiin tässä StatFin-tietokannasta GET-avainsanahaulla population-dataa ja viedään se tietokoneelle testikansioon json-tiedostona.

```
1  *** Settings ***
2  Library        REST
3
4  *** Test cases ***
5  GetPopulation
6    GET      http://pxnet2.stat.fi/PXWeb/api/v1/en/StatFin?query=population
7    Output    response body   file_path=${CURDIR}/population1.json
8
```

Tiedosto tallennetaan näin samaan kansioon testin kanssa.

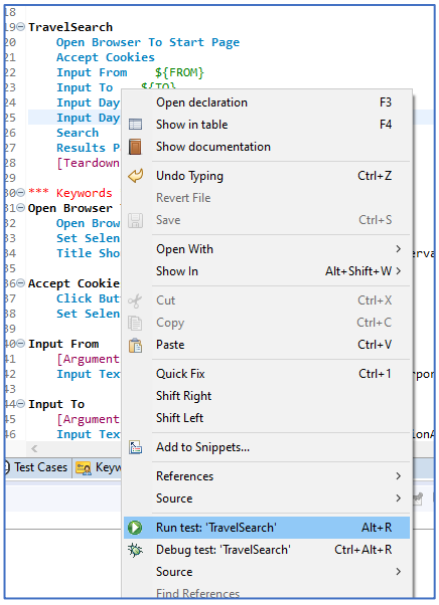


4 Testien ajo

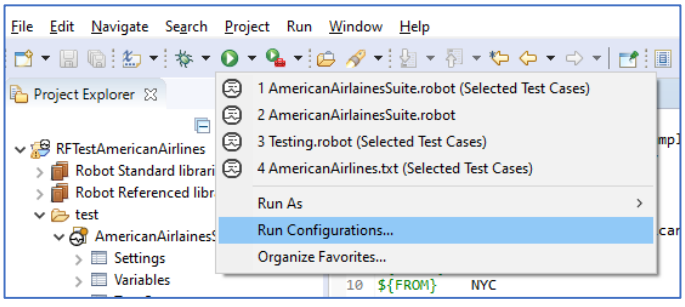
Eclipsellä voidaan suorittaa testejä tai tehdä virheenjäljitystä. Testejä voidaan ajaa vaihtoehtoisesti komentorivin kautta.

4.1 Eclipse testien ajaminen

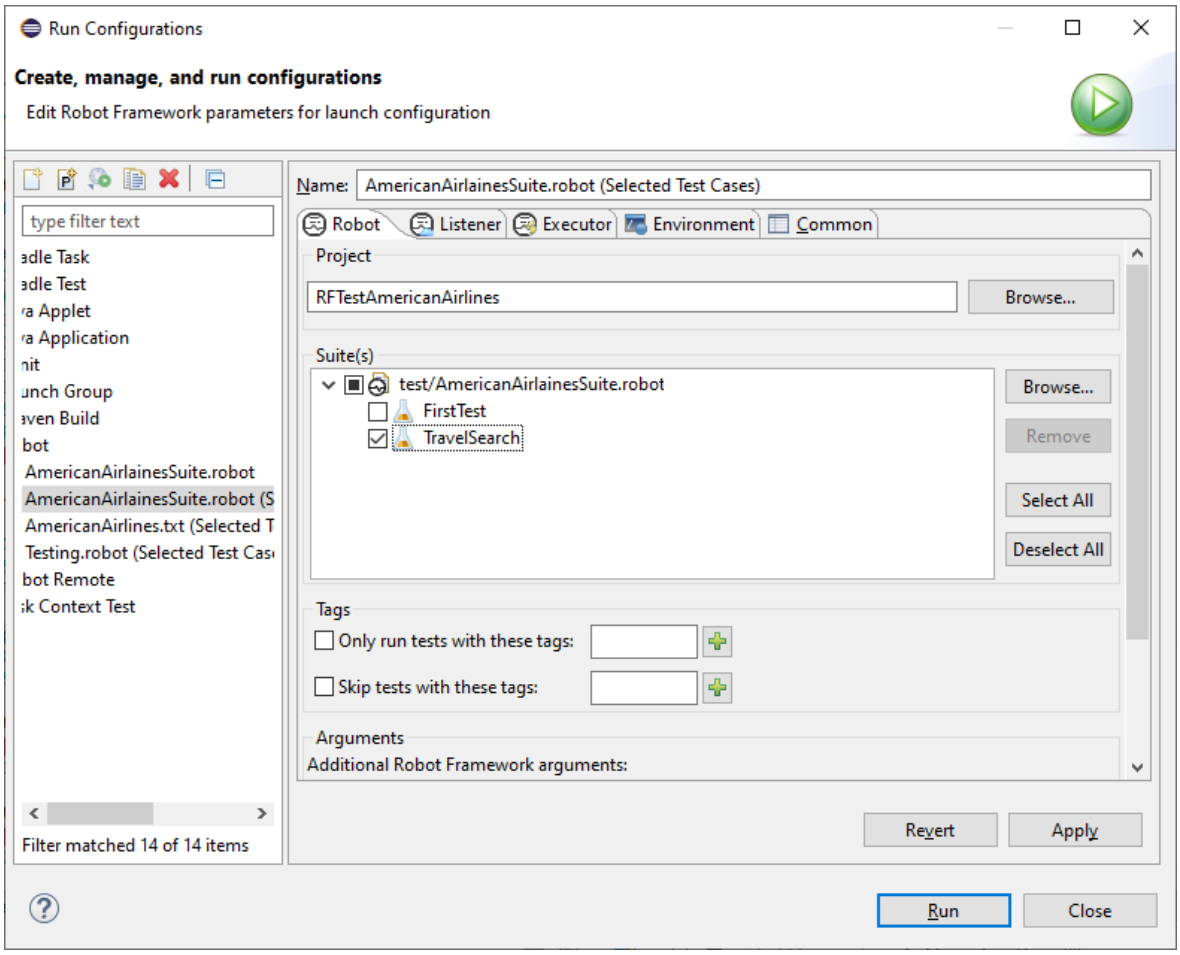
Testin ajaminen onnistuu helposti valitsemalla ajettavan testin ja painamalla hiiren oikeaa painiketta testin kohdalla. Aukeavassa valikossa näkyy Run test: ”testin nimi”.



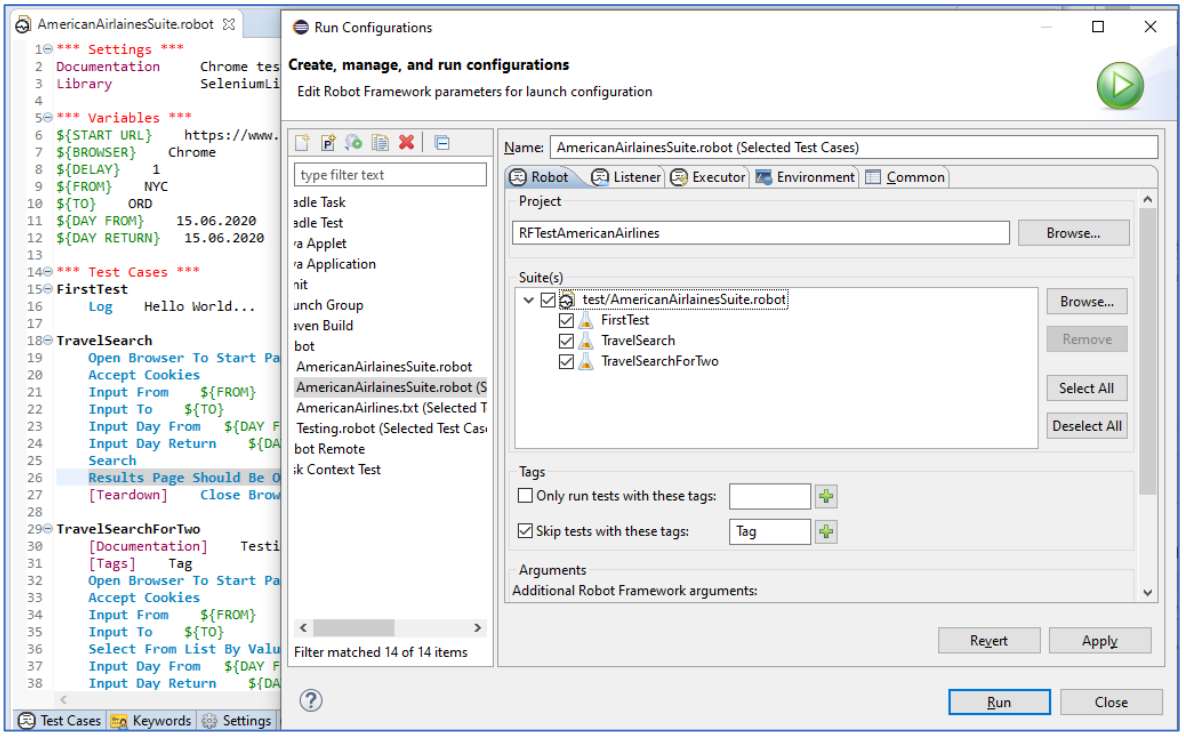
Vaihtoehtoisesti testin ajaminen onnistuu Tools-nauhalta vihreästä Play-painikkeesta ja valitaan Run Configurations.



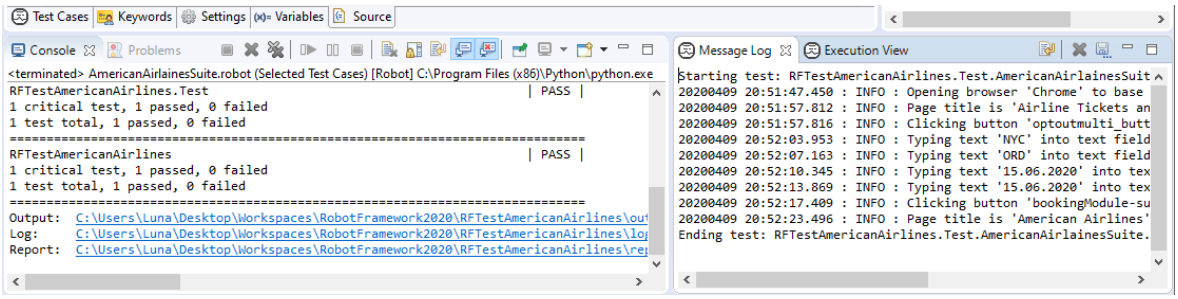
Aukeavassa ikkunassa voidaan valita ajetaanko koko testisetti vai pelkästään yksittäisiä testejä.



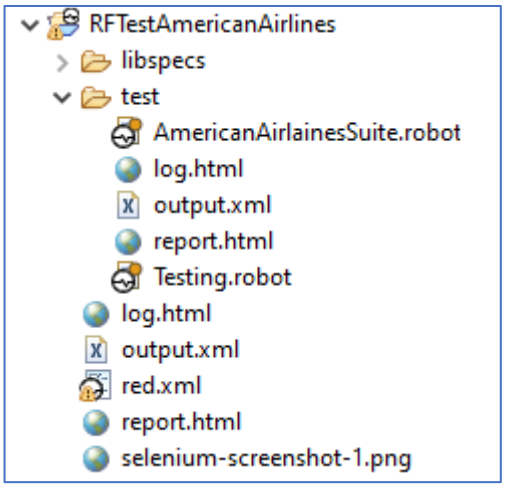
Run conficurations -ikkunasta voidaan myös ajaa testejä tai testisettejä tagien perusteella:



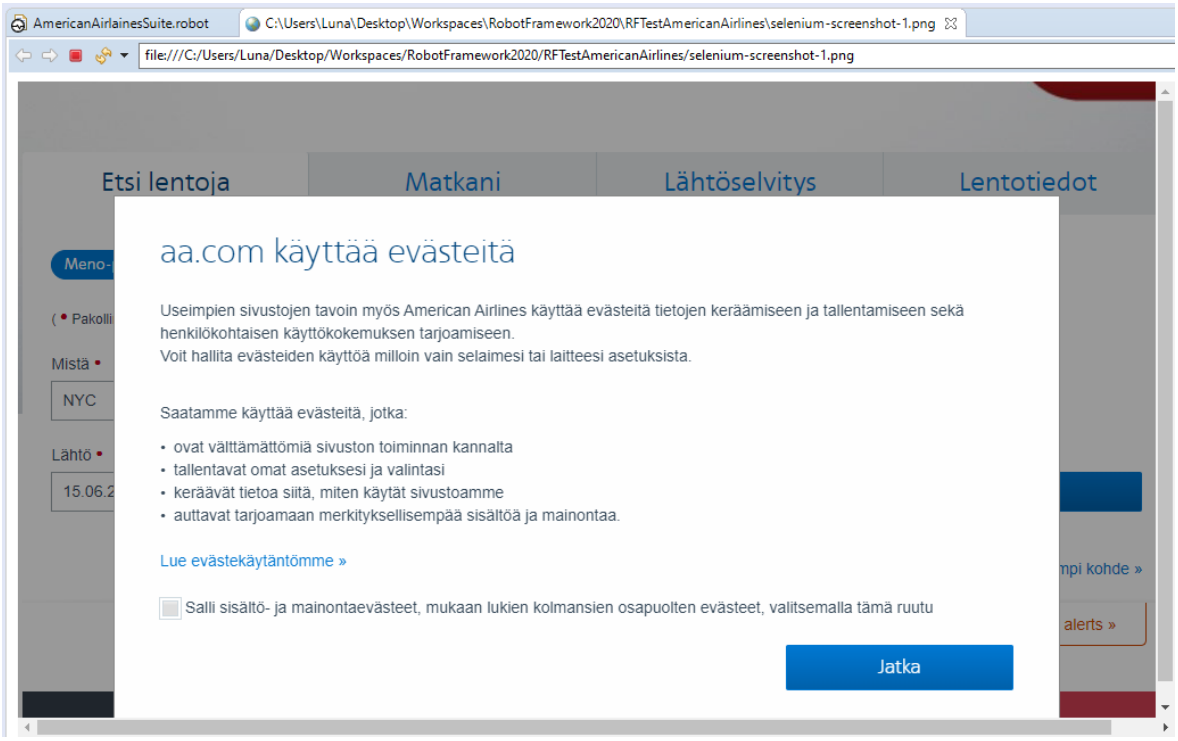
Ajon etenemistä voi seurata konsolista ja logitusikkunasta.



Ajosta muodostuu kolme raporttia: Output (.xml-tiedosto), Log (html-tiedosto) ja Report (html-tiedosto). Testin epäonnistuessa tulee myös kuvakaappaus (.png-tiedosto) epäonnistuneesta kohdasta.

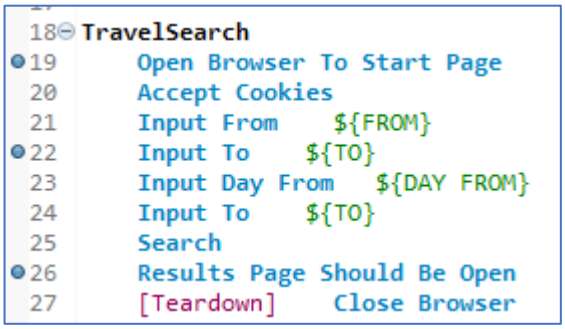


Kuvakaappaus avattu Eclipsessä:

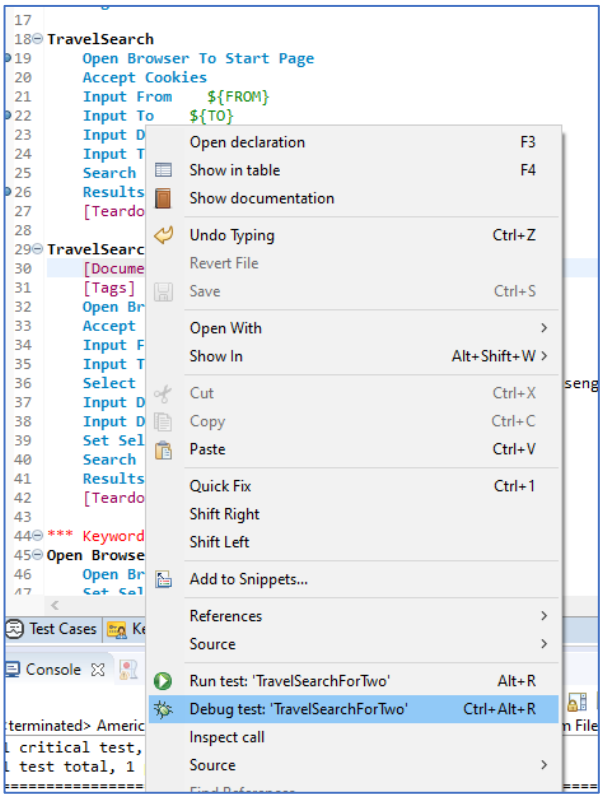


4.2 Eclipse virheenjäljitys (debugging)

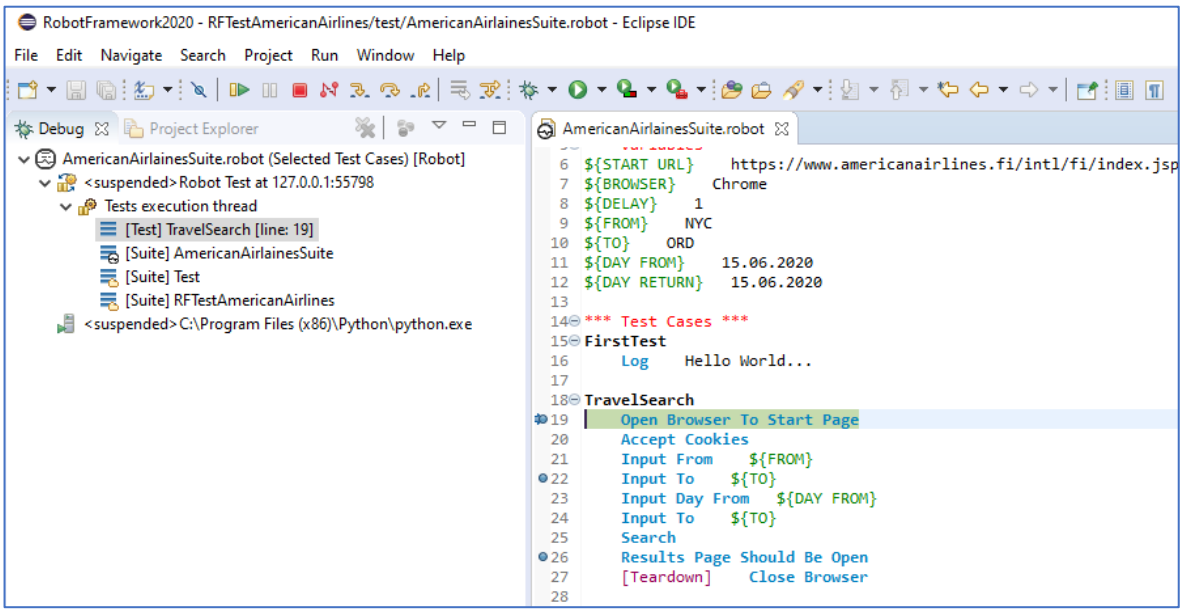
Eclipsessä on mahdollista ajaa testi(t) virheenjäljitystilassa. Tätä varten testiin pitää tehdä breakpoint-kohdat.

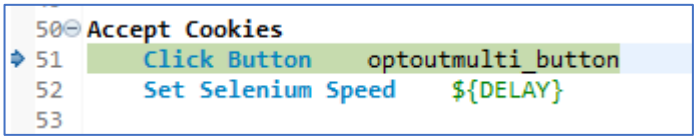


Testin virheenjäljitys voidaan aloittaa klikkaalla testin kohdalla hiiren oikeaa painiketta ja valitaan aukeavasta valikosta Debug test: ”testin nimi”.



Virheenjäljitys toimii kuten ohjelmoinnissakin. Hyvä puoli on se, että pystyy seuraamaan kohta kerrallaan mitä tapahtuu ja palaamaan takaisin. Eclipsen virheenajon painikkeet ovat selkeitä ja testissä näytetään missä kohdassa mennään.

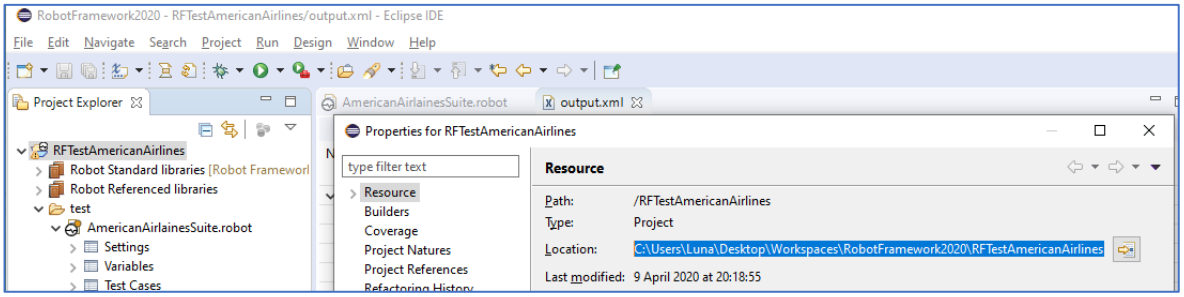




4.3 Komentorivin kautta ajaminen

Komentorivin kautta testien ajaminen voi olla nopeampaa eikä silloin koneella tarvitse olla Eclipse asennettuna. Periaatteessa tarvitaan testitiedosto ja komentorivi. Lisäksi koneella pitää olla asennettuna python, Robot Framework sekä tarvittavat kirjastot ja selainajurit.

Komentorivin kautta testiä ajamalla vaihdetaan ensin polku projektin sijaintiin, jotta komennoissa ei tarvitse joka kerta kuvata koko polkua. Kansiopolun voi tarkistaa esimerkiksi Eclipsellä valitsemalla projektin hiiren oikealla painikkeella ja valitsemalla Properties ja täältä löytyy Location:



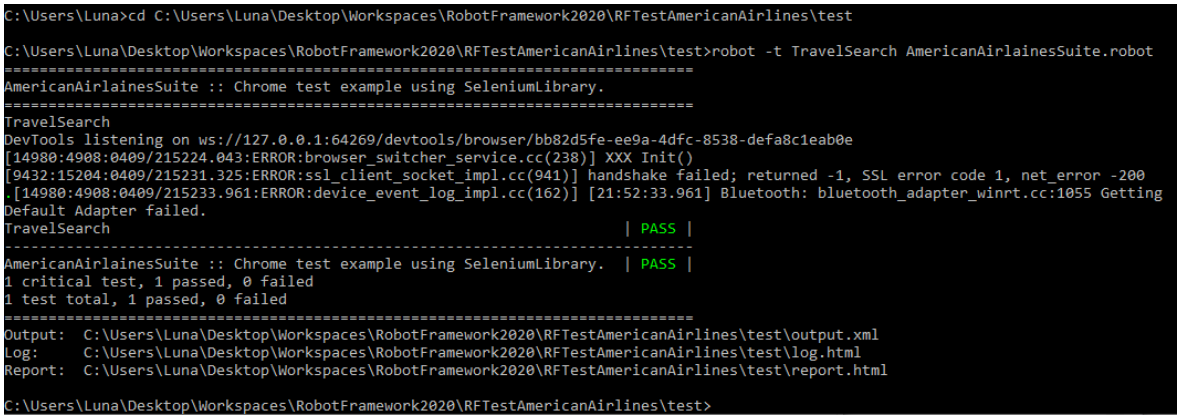
Komentorivillä vaihdetaan sijainti testikansioon, tässä tapauksessa komentolla:

```
cd C:\Users\Luna\Desktop\Workspaces\RobotFramework2020\RFTestAmerica-
nAirlines\test
```

Testi ajetaan tämän jälkeen komennolla

```
robot -t testname testsuitename
```

-> `robot -t TravelSearch AmericanAirlainesSuite.robot`



Voi myös ajaa useita testejä ketjuttamalla robot sanan jälkeen komentoa:

```
robot -t testname testsuitename -t testname testsuitename
```

Testisetin pystyy ajamaan komennolla

```
robot testsuite name
```

Testejä voi ajaa tagien mukaan:

```
robot --include exampletag
robot --exclude not_ready
```

Komentoriviin tulee samat Output-, Log- ja Report-tiedostojen linkit ja nämä saa auki selaimessa.

Komentorivillä saa Robot Framework ohjeita esiin komennolla:

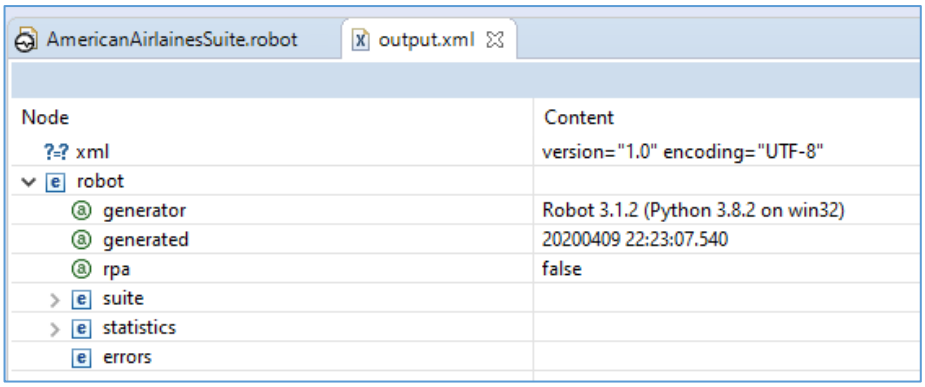
```
robot –help
```

Robot Framework oppaassa on myös listattu kaikki komentorivin käskyt: http://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html#all-command-line-options

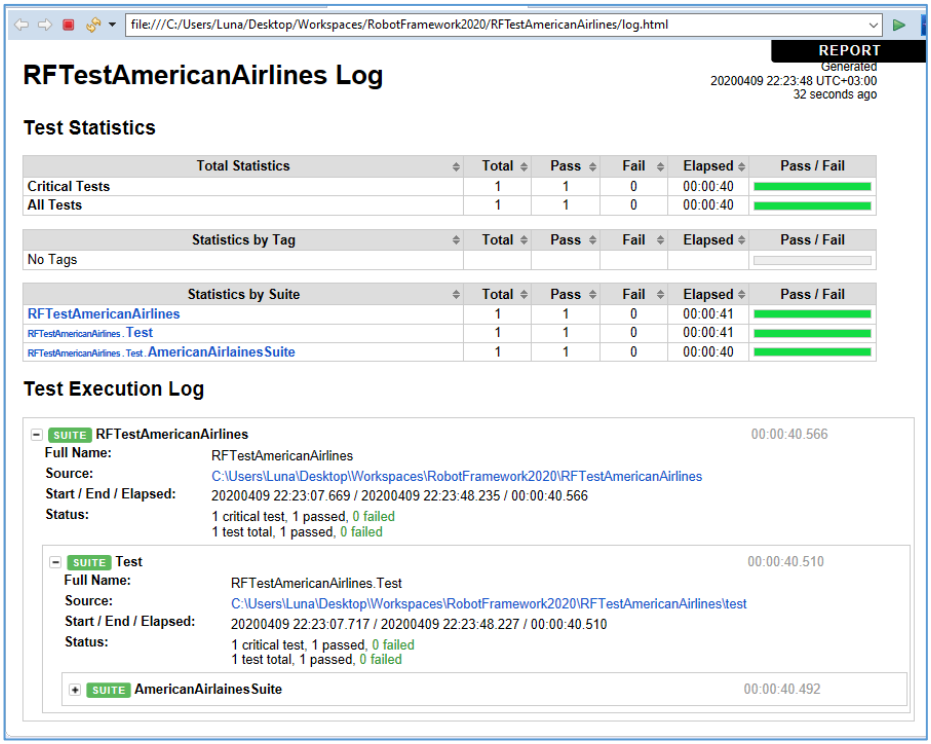
5 Testien tulokset

Testien ajotavasta huolimatta muodostuu testeistä oletusarvoisesti kolme tiedostoa: Output-tiedosto xml-muodossa, Log-tiedosto html-muodossa ja Report-tiedosto html-muodossa. Nämä tulevat oletuksena projektikansioon ja siten nämä ylikirjoitetaan projektin testien ajoissa. Näille voi tehdä myös oman kansionsa tai siirtää tiedostot ajojen jälkeen esimerkiksi ajopäivien mukaan nimettyihin kansioihin.

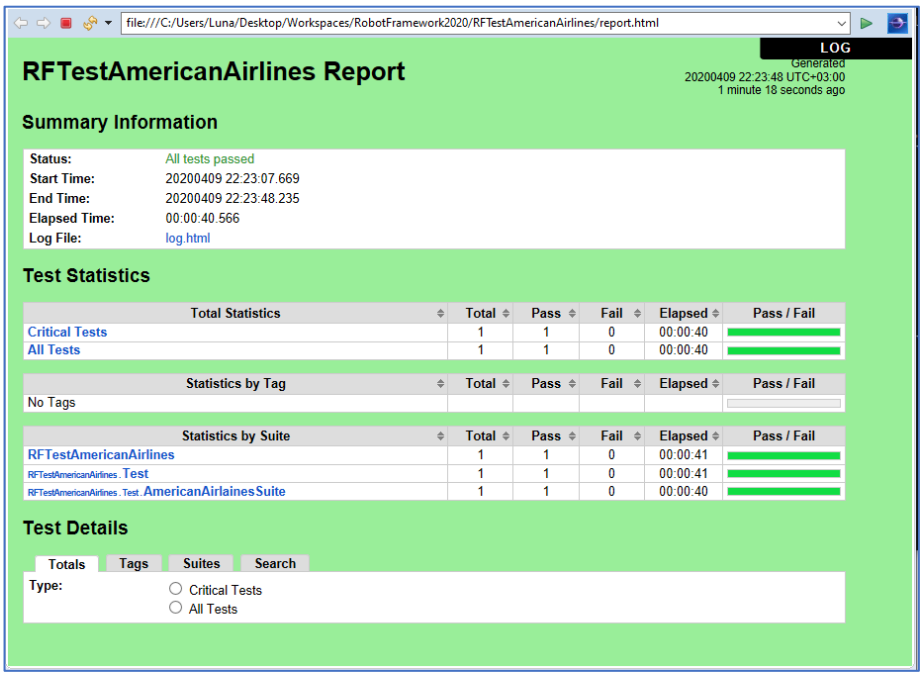
Output-tiedosto sisältää testin ajon tulokset koneluettavassa xml-formaatissa.



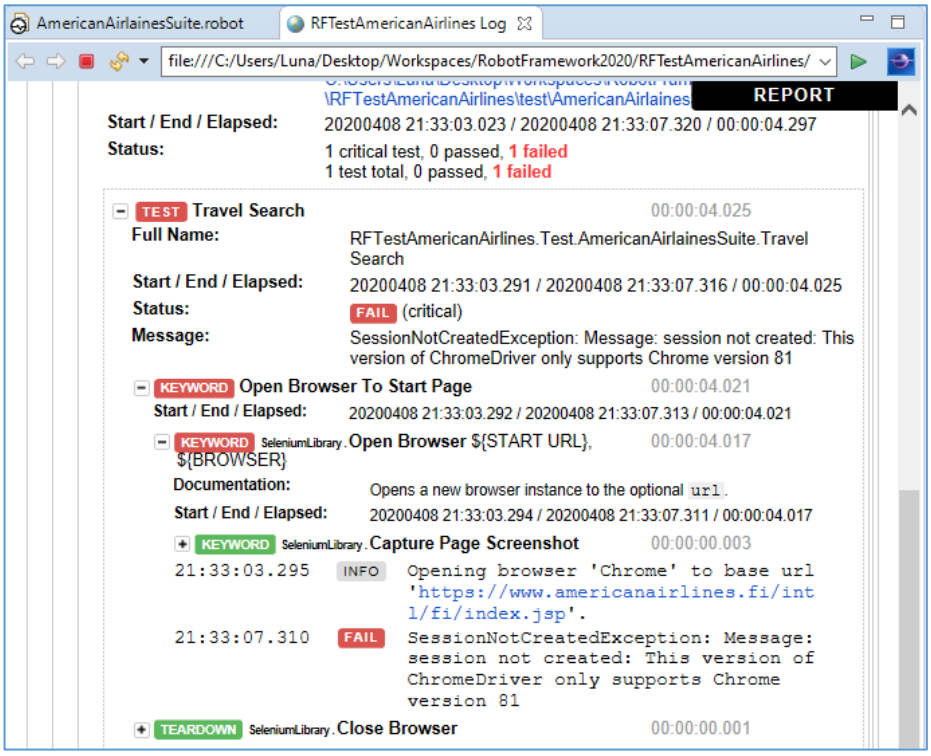
Log-tiedosto sisältää tietoa ajetuista testeistä HTML-formaatissa. Näissä rakenne seura testisetin, testitapauksen ja avainsanojen polkua.



Report-tiedosto sisältää yleisnäkymän testien ajon tuloksista HTML-formaatissa. Tässä on statistiikka perustuen tageihin ja suoritettuihin testisetteihin sekä lista kaikista ajetuista testeistä. Jos kaikki kriittiset testit menevä läpi onnistuneesti, on tausta vihreä, muutoin se on punainen.



Jos testin ajo epäonnistuu, niin kannattaa katsoa Log-tiedostosta missä vaiheessa virhe tapahtuu, mikä on testitapaus ja testikohta sekä mahdolliset virheilmoitukset. Esimerkiksi tässä selaimen haltuunotto epäonnistui:



Tässä virheessä syynä oli Chromen ja ajurin yhteensopimattomuus, mikä ratkesi päivittämällä Chromen 81-versioon.

Seuraavassa esimerkkivirheessä testissä on päästy eteenpäin, mutta Search-testivaihe epäonnistuu. Virheilmoitusta ja kuvakaappausta tarkastelemalla käy ilmi, että Hae-painike ei ole valittavissa, sillä verkkosivulla on evästeiden hyväksyntä -ikkuna. Lisäämällä stepin, jossa hyväksytään evästeet, menee testi onnistuneesti läpi.