Opinnäytetyö (AMK)

Tieto- ja viestintätekniikan koulutus

2025

Tanja Honkanen

LASTAUSKONEEN SIMULAATIOTESTAUKSEN AUTOMATISOINTI

TURKU AMK

TURKU UNIVERSITY OF APPLIED SCIENCES

OPINNÄYTETYÖ (AMK) | TIIVISTELMÄ

TURUN AMMATTIKORKEAKOULU

Tieto- ja viestintätekniikan koulutus

2025 | 36

Tanja Honkanen

LASTAUSKONEEN SIMULAATIOTESTAUKSEN AUTOMATISOINTI

Opinnäytetyön tavoitteena oli automatisoida lastauskoneen ohjausjärjestelmän regressiotestit Robot Frameworkilla. Regressiotestit ovat testejä, jotka toistuvat usein, ja lastauskoneiden ohjausjärjestelmä oli siirtymässä uuteen sukupolveen, mikä mahdollisti testien automatisoinnin. Tarkoituksena oli myös selvittää testausautomaation hyötyjä ja kannattavuutta sekä mahdollisia ongelmia.

Työn toteutus aloitettiin tutustumalla lastauskoneen ohjausjärjestelmän simulaatiotestaukseen ja Robot Frameworkiin. Tämän jälkeen luotiin Robot Frameworkin vaatima testiympäristö. Työssä luotiin testikirjastoja Python-ohjelmointikielellä sekä erilaisia rajapintoja simulaatio-ohjelmiston ja ohjausjärjestelmän välille. Testitapaukset luotiin valmiiden regressiotestien dokumentointien pohjalta.

Työn tuloksena suuri osa regressiotestauksen testitapauksista saatiin automatisoitua. Robot Frameworkin kirjastoista ja testitapauksista saatiin selkeitä ja ymmärettäviä myös sellaisille henkilöille, jotka joutuvat työskentelemään testausautomaation parissa, mutta joilla ei välttämättä ole ohjelmointitaustaa.

Testauksen automatisointi osoittautui kannattavaksi, kun se toteutetaan oikeista lähtökohdista. Jos lähtökohtana on automatisoida koko testaus tai vain harvoin toistuvia ja haasteellisia testitapauksia, ei testausautomaatio ole kannattavin ratkaisu.

ASIASANAT:

Robot Framework, Python, ohjelmistokehitys, ohjelmistotestaus, testausautomaatio

BACHELOR´S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Information and Communications Technology

2025 | 36

Tanja Honkanen

SIMULATION TESTING AUTOMATION OF A LOADER CONTROL SYSTEM

The aim of the thesis was to automate the regression tests of a loader's control system with Robot Framework. Regression tests are frequently recurring and the control system of the loaders was shifting to a next generation, which made possible to automate the tests. Another aim was to find out the benefits and profitability of test automation as well as possible problems.

To start the implementation of the work, it wasimportant to become acquainted with the simulation testing of the control system and the Robot Framework. After that, the test environment required by the Robot Framework was created. In the thesis, test libraries and various interfaces between the simulation software, the loader's display software and the control system were created with the Python programming language. Test cases were created based on the documentation of the ready-made regression tests.

As a result of the work, a large part of the regression testing's test cases could be automated. Robot Framework libraries and test cases were also made clear and understandable to those who have to work with test automation but may not have any programming background.

Testautomation is profitable when the premises are suitable. If the purpose is to automate the entire testing or only infrequently repetitive and challenging test cases, test automation is not the best solution.

KEYWORDS:

Robot Framework, Python, software development, software testing, test automation

SISÄLTÖ

KÄYTETYT LYHENTEET TAI SANASTO 5

1 JOHDANTO 6

2 OHJELMISTOTESTAUS 7

2.1 Testaus osana ohjelmistokehitystä 7

2.2 Testauksen tasot 12

2.3 Testausmenetelmät 14

3 TESTAUSAUTOMAATIO 19

4 TYÖKALUT 21

4.1 Robot Framework 21

4.2 Python-ohjelmointikieli 22

4.3 Robot Framework IDE 22

5 TESTIEN TOTEUTTAMINEN 23

5.1 Testiympäristö 23

5.1.1 Robot-tiedosto 24

5.1.2 Resurssitiedosto 27

5.1.3 Python-tiedosto 30

5.2 Testien tulokset 31

6 LOPUKSI 34

LÄHTEET 35

KÄYTETYT LYHENTEET TAI SANASTO

Lyhenne	Lyhenteen selitys (Lähdeviite)
DLL-tiedosto	Dynaaminen linkkikirjasto. Binäärinen suoritettavan tiedoston tyyppi. (Openize Pty Ltd 2025.)
MCON	Rajapinta MCUn ja koneen näytön signaaleiden lukemiseen (Honkanen 2024).
MCU	Machine Control Unit. Koneen ohjausyksikkö. (Honkanen 2024.)
SUP	Lastauskoneen näyttö (Honkanen 2024).
UI	User Interface eli käyttöliittymä (Ahonen 2025).

1 JOHDANTO

Opinnäytetyön aiheena on maanalaisen lastauskoneen ohjausjärjestelmän regressiotestauksen automatisointi Robot Frameworkilla. Ohjelmistopaketin regressiotestaus toistuu usein ja samanlaisena, jolloin testauksen automatisointi on kannattavaa. Opinnäytetyön tarkoituksena on saada erityisesti regressiotestit automatisoitua, jotta testaajien resursseja vapautuisi enemmän esimerkiksi ohjelmiston uusien toiminnallisuuksien tarkempaan testaamiseen. Vaikka työkaluja ohjelmistotestauksen automatisointiin löytyy paljon, on testauksen automatisointi vasta tekemässä tuloaan osaksi ohjelmistotestausta.

Toimeksiantajan yrityksen maanalaisten kaivoskoneiden ohjausjärjestelmä on siirtymässä uuteen sukupolveen, mikä mahdollistaa testauksen automatisoinnin kehittämisen Robot Frameworkilla. Testausresurssit ovat rajalliset, ja lastauskoneen ohjausjärjestelmän ohjelmistopaketin regressiotestaus vaatii paljon aikaa manuaalisesti suoritettuna. (Keihäs, J. Haastattelu. 28.4.2020.)

Ohjelmistotuotannosta ja ohjelmistotestauksesta löytyy paljon materiaalia niin kirjallisuudesta kuin verkosta. Ohjelmistoala on kuitenkin nopeasti kehittyvä ala, jolloin tieto saattaa vanhentua melko nopeasti, joten on hyvä suhtautua kriittisesti tiedon ajankohtaisuuteen.

Opinnäytetyössä kerrotaan aluksi yleisesti ohjelmistotuotannosta ja miten testaus toimii osana sitä. Työ sisältää ohjelmistotestauksen päätasot ja yleisimpiä testausmenetelmiä. Luvussa kolme keskitytään testausautomaatioon yleisesti sekä testausautomaation hyötyihin ja mahdollisiin riskeihin ohjelmistotestauksessa. Lisäksi opinnäytetyössä esitellään työssä tarvittuja työkaluja.

2 OHJELMISTOTESTAUS

Ohjelmistotestauksella pyritään varmistamaan, että toteutettava ohjelmistotuote on toivotun kaltainen ja että kaikki sen sisältämät ominaisuudet toimivat oikein. Näin ollen testauksen päätavoitteena on tarkastaa, että ohjelmisto täyttää asiakkaan tarpeet, on toteutettu kuten alun perin suunniteltiin, ja että ohjelmisto on tehty oikein. (Kasurinen 2013, 10.) Viallinen ohjelmisto voi aiheuttaa mittavia menetyksiä taloudellisesti, mutta pahimmassa tapauksessa myös ihmishenget voivat olla vaarassa (Guru99 2023b). Testaustyö on myös kannattavuuden kannalta mitattuna tärkein työvaihe, sillä huolellisesti testatut tuotteet saavat paremman katteen kuin ne tuotteet, jotka ovat testattu huonosti. Ohjelmiston testaamiseen panostaminen jo ohjelmiston kehityksen aikana on tärkeää, sillä virheen korjaaminen suunnittelun ja kehityksen aikana maksaa 1–2 prosenttia siitä, mitä korjaaminen maksaisi, jos se tehtäisiin vasta tuotteen julkaisun jälkeen. Puutteellinen tai vajavainen testaus voi aiheuttaa ohjelmistoa kehittäville yrityksille suorien tappioiden lisäksi suuria rahallisia menetyksiä, esimerkiksi asiakkaille koituneiden vahinkojen tai yrityksen julkisuuskuvan korjaamisesta aiheutuvista kuluista. (Kasurinen 2013, 11–12.)

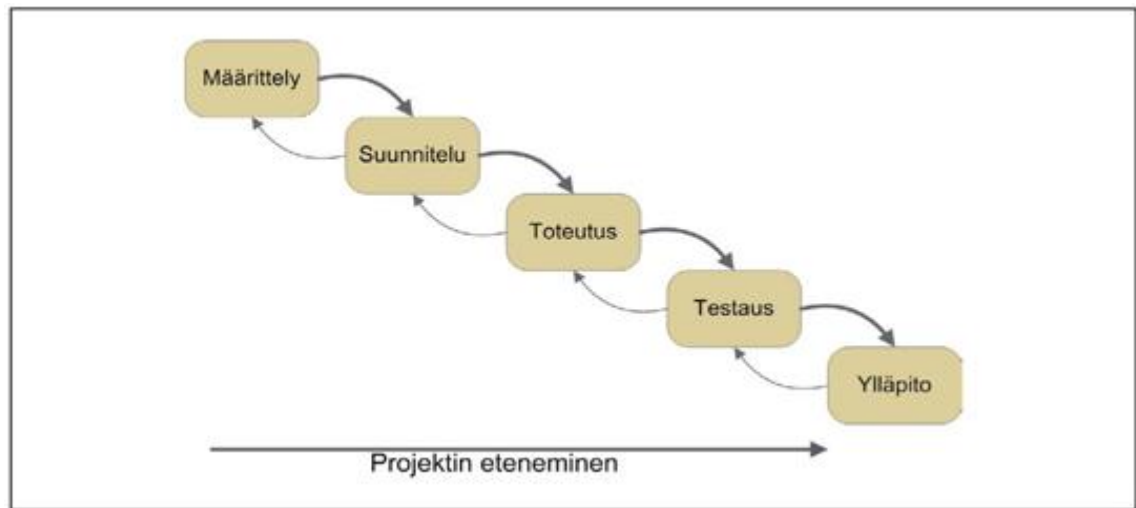
Kun ohjelmiston kokonaisuus on niin toimintakuntoinen, että se toteuttaa kaikki siltä odotetut toiminnot, eikä sisällä merkittäviä virheitä, testausvaihe voidaan päättää. Ohjelmista löytyy aina joitain virheitä, mutta hyvin testatuissa ohjelmissa virheet ovat ainutlaatuisia ja epätavallisia, jolloin ne eivät haittaa ohjelman käyttöä. (Kasurinen 2013, 13.)

2.1 Testaus osana ohjelmistokehitystä

Vaikka testaus on tärkeä ja iso osa ohjelmistokehitystä, testauksen käytännön toteus nähdään hyvin eri tavalla riippuen ohjelmistotuotannon mallista. Vesiputousmallissa testaus on erillinen vaihe ohjelmistotuotannon lopussa, kun taas V-malli, RUP ja Scrum pyrkivät toteuttamaan testausta jokaisessa ohjelmistotuotannon vaiheessa.

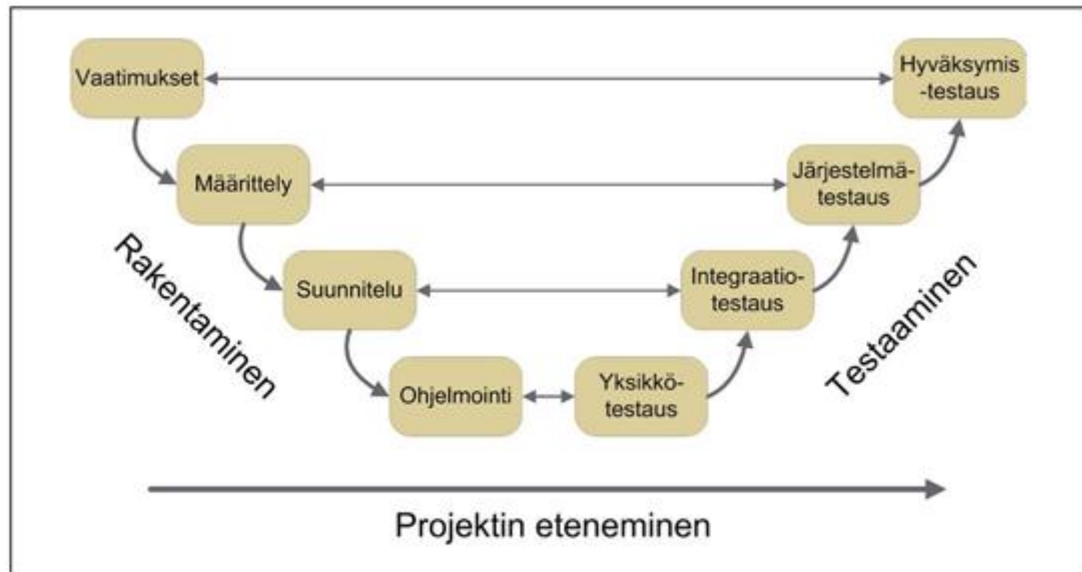
Perinteinen malli toiminnan etenemisestä ohjelmistokehityksessä on vesiputousmalli (kuva 1). Vesiputousmallissa jokainen työvaihe etenee vaiheittain seuraavaan ja lopulta, kun jokainen ohjelmiston yksittäinen komponentti on valmis, liitetään ne yhtenäiseksi kokonaisuudeksi. (Kasurinen 2013, 13.) Jokainen vaihe on siis suunniteltu toteuttamaan

tietty toiminta loppuun, ja vasta järjestelmän käyttöönoton jälkeen aloitetaan testausvaihe (Guru99 2023a).



Kuva 1. Vesiputousmallissa projekti etenee vaiheittain. Testaus on oma erillinen vaiheensa. (kuva Kasurinen 2013, 13.)

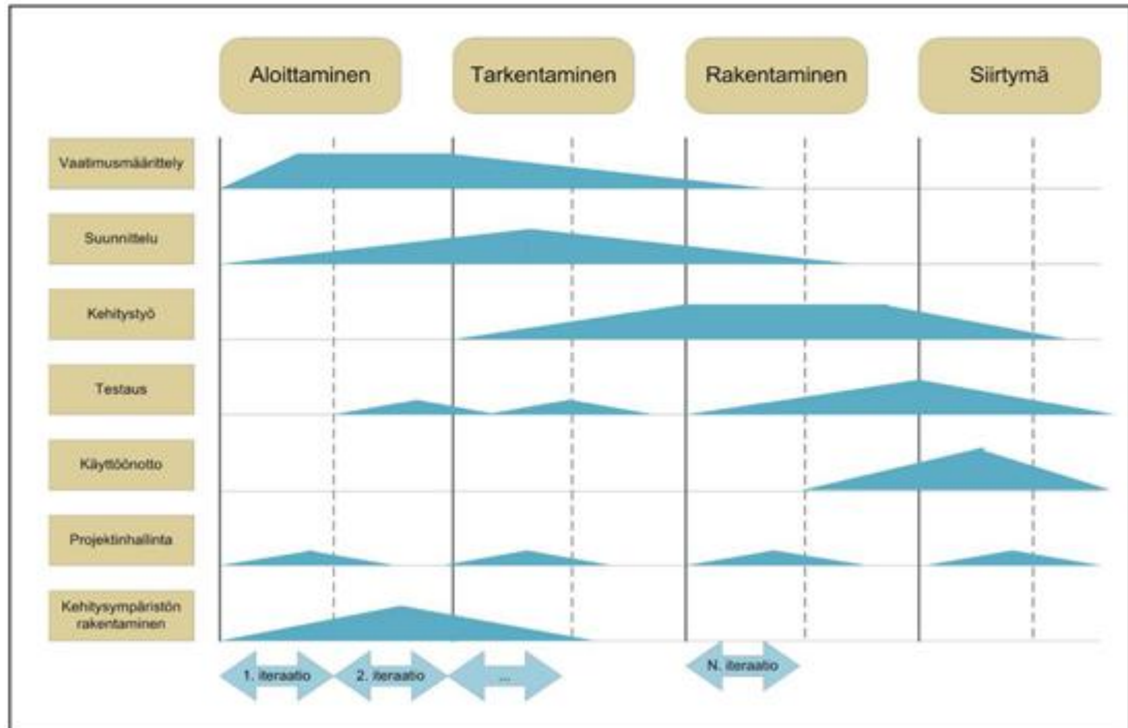
Vesiputousmallia pidetään kuitenkin vanhanaikaisena, koska testaus tapahtuu ainoastaan yhdessä vaiheessa. Testausta ja kehitystyötä kuvaavampana mallina pidetään esimerkiksi V-mallia (kuva 2). V-malli koostuu pääosin samoista vaiheista kuin vesiputousmalli, mutta V-mallissa on merkitty jokaiselle vaiheelle myös vastaava testauksen taso. (Kasurinen 2013, 14.) V-mallin vasen sivu kuvastaa ohjelmistokehityksen vaiheita ja mallin oikea sivu kuvastaa pelkästään ohjelmistotestauksen vaiheita. Näin ollen testaus ei ole vain itsenäinen toimintansa, vaan sitä suoritetaan jatkuvasti eri kehitysvaiheiden rinnalla. (Guru99 2023a.)



Kuva 2. V-mallissa projektin jokaiselle kehitysvaiheelle on oma testaustasonsa (kuva Kasurinen 2013, 14).

Kuten useat ohjelmistotuotannon mallit, myös vesiputousmalli ja V-malli törmäävät usein samaan ongelmaan, testaus alkaa liian myöhäisessä vaiheessa (Kasurinen 2013, 14). Tämän takia suunnittelulähtöinen RUP (Rational Unified Process) ja ketterä menetelmä Scrum ovat saaneet paljon suosiota ohjelmistotuotannon prosessimalleina (Kasurinen 2013, 24).

RUP-malli voidaan jakaa neljään päätyövaiheeseen (kuva 3), joista ensimmäinen on aloittaminen. Aloittaminen kattaa pääasiassa toimintakokonaisuuden mallintamiseen ja vaatimusmäärittelyyn liittyviä toimenpiteitä. Aloittamisvaiheen tarkoitusena on tutkia onko tuotetta kannattavaa tehdä ja minkälaisia resursseja projekti vaatii. (Kasurinen 2013, 24–25.) Vaatimusmäärittely selvittää, mitä järjestelmältä vaaditaan, miksi ja mitä tarpeita on hankittava sekä millaisia toimintoja ohjelmalta lopulta halutaan (Raitoharju 2019). Vaatimusmäärittelyn pohjalta voidaan luoda testitapauksia ja päättää miten ohjelmaa halutaan testata (Kasurinen 2013, 26).



Kuva 3. RUP-malli koostuu neljästä päävaiheesta (kuva Kasurinen 2013, 25).

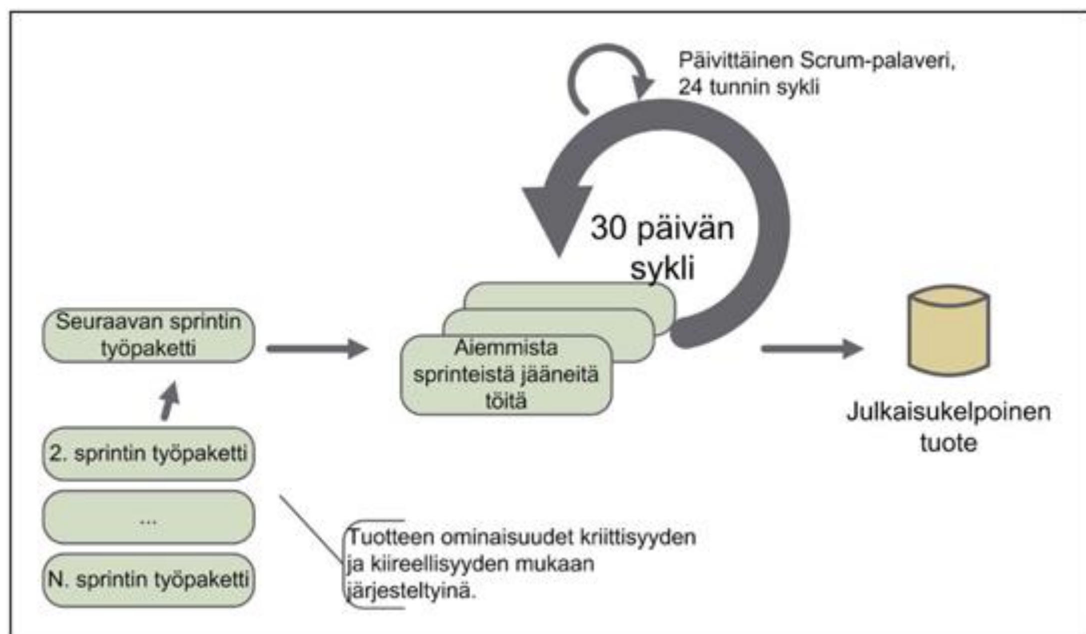
Toinen vaihe on tarkentaminen, johon siirrytään kun ohjelma on päätety toteuttaa. Tässä vaiheessa luodaan perusta ohjelman arkkitehtuurille ja tunnistetaan projektin riskialttiit tekijät. Tarkentamisvaihe on tärkein RUP-mallin neljästä vaiheesta, sillä tässä vaiheessa tekninen suunnittelu on aloitettu ja päätös siitä jatketaanko projektia rakennus- ja siirtymävaiheisiin tehdään. (Kruchten 2004, 69.) Testaustoiminta alkaa viimeistään tässä vaiheessa (Kasurinen 2013, 26).

Rakennusvaiheessa ohjelman rakentaminen toteutetaan vaihe kerrallaan aloittaen osista, joiden valmistumiseen liittyy suurimmat riskit. Rakentamisen edetessä voidaan vielä hienosäätää ohjelman arkkitehtuuria ja rakennetta. Testauksen osalta rakennusvaiheessa painotetaan erityisesti teknisiä testauksen tasoja, kuten yksikkötestausta, integrointitestausta ja lopulta järjestelmätestausta, mutta myös muut testausmenetelmät, kuten esimerkiksi musta- ja lasilaatikkotestaus, voidaan aloittaa tässä vaiheessa. (Kasurinen 2013, 26.)

Viimeinen työvaihe on siirtyminen, jossa ohjelma on niin valmis, että käyttäjät voivat kokeilla sitä. Yleisesti järjestemä muistuttaa valmista tuotetta, vaikka joitain ominaisuuksia tai muutoksia saattaa vielä puuttua tuotteesta. Tärkeimpiä testimuotoja siirtymisvaiheessa on käytettävyystestit ja tutkiva testaaminen. (Kasurinen 2013, 26.)

RUP-mallin parhaimpia puolia on, että testaukseen ja vaatimustenhallintaan on varattu resursseja projektin alusta aina loppuun saakka. Koska tuotteen tekeminen aloitetaan riskialttiimista ominaisuuksista, on epätodennäköistä, että projekti jää jumiin loppuvaiheessa esiin tulleeseen merkittävään ongelmaan. Malli toteuttaa iteratiivisissa eli toistuvissa osissa valmistamista, missä jokaisen osan jälkeen tuotos tarkastetaan ja testataan ennen seuraavaan iteraatioon siirtymistä. RUP-mallissa yhden iteraation kesto on noin yhdestä kolmeen kuukautta. (Kasurinen 2013, 26-27.)

Iteraation keston ei tarvitse olla kuukausia, esimerkiksi Scrum-mallissa (kuva 4) yksi iteraatio eli sprintti on yleensä muutaman viikon tai kuukauden mittainen jakso. Tuotteen toteutus tapahtuu sprinteissä, joissa on tarkoitus kehittää ohjelmistoon joukko ominaisuuksia ja toimintoja sekä testata että ne toimivat oikein. Jokaisen sprintin alussa käydään Scrum-palaveri, jossa sovitaan tehtävät asiat ja kuka tekee mitäkin. Lisäksi järjestelmässä olevat ongelmat käydään palaverissa läpi. (Kasurinen 2013, 28-29.)



Kuva 4. Scrum-malli, jossa työ toteutetaan sprinteissä (kuva Kasurinen 2013, 28).

2.2 Testauksen tasot

Vaikka ohjelmistokehityksen V-malli ei ole paras sellaisenaan kuvaamaan ohjelmistotuotantoa, testaus jaetaan usein V-mallissa kuvattuihin testauksen tasoihin. V-mallissa testaus on jaettu neljään tasoon, jotka ovat yksikkö-, integraatio-, hyväksymis- ja järjestelmätestaus. (Kasurinen 2013, 14.)

Yksikkötestaus on tavallisin testauksen työvaihe, ja sitä käytetään yleisesti kaikissa ohjelmisto-organisaatioissa. Yksikkötestauksella pyritään varmistamaan, että uusi toteutettu muutos tai toiminto kehitteillä olevaan järjestelmään toimii ainakin perusajatukseltaan. Ohjelmoija eli muutoksen tai toiminnon kehittäjä suorittaa yksikkötestauksen useimmiten itse, tarkastamalla että muutos tai toiminto kääntyy ilman virheitä ja toteuttaa oikeat toiminnalliset ominaisuudet. (Kasurinen 2013, 51.) Kääntämisellä tarkoitetaan ohjelmointikielellä kirjoitetun koodin muuntamista tietokoneen ymmärtämään muotoon, eli biteiksi (Heikkinen & Saarenpää 2016). Yksikkötestauksen avulla ohjelmoija huomaa heti, jos jokin testeistä epäonnistuu ja pystyy korjaamaan vian ennen kuin komponentti eli muutos tai toiminto ehditään liittämään osaksi laajempaa ohjelmaa. Yksittäinen komponentti eli moduuli ei usein pysty kuitenkaan tekemään mitään itsenäistä, jolloin testausta helpottamaan luodaan testikomponentteja tai testitynkiä, joiden tarkoitus on simuloida tarvittavia järjestelmän osia. Esimerkiksi virhetilojen simulointi on helpompaa testikomponenteilla, jolloin vaikka muistikortin kirjoitusvirhettä testattaessa ei muistikorttia tarvitse rikkoa, vaan riittää että luodaan muistikortin tilalle komponentti, joka joko ilmoittaa kirjoituksen epäonnistuneen tai ei kirjoita dataa ollenkaan ylös. (Kasurinen 2013, 51-52.)

Seuraava testaustaso, jossa järjestelmän osia sovitetaan yhteen tavoitteena saada koko järjestelmä toimimaan yhtenä kokonaisuutena, on integrointitestaus (integration testing). Integrointitestauksessa uusi komponentti liitetään osaksi aiemmin todistetusti toimivaa järjestelmää, ja tarkastetaan että kokonaisuus toimii tämän jälkeen edelleen oikein. (Kasurinen 2013, 54.) Ohjelman testaamiseksi luodaan testitapauksia. Yksittäinen testitapaus sisältää esiehdot, ohjelmalle annettavat syötteet, ohjelman odotetun tuloksen annetuilla syötteillä sekä jälkiehdot. Esiehdot määrittelevät, missä tilassa ympäristön ja ohjelman pitää olla testitapauksen suorittamiseksi, vastaavasti jälkiehdot määrittelevät, missä tilassa testattavan ohjelman on oltava testauksen jälkeen. Testitapauksista luodaan testitapausten määrittelydokumentti, johon kirjataan lyhyesti testitapauksessa testattavat ohjelman osat ja toiminnot sekä mahdolliset yhteydet muihin testitapauksiin

ja testattaviin kohteisiin. (Turun ammattikorkeakoulu 2019.) Integrointitestaus toteutetaan testitapauksilla, jotka eivät sisällä vielä koko järjestelmän käyttöä koskevia testejä, mutta ovat kuitenkin laajempia kuin yksikkötestauksessa käytetyt kokonaisuudet. Osien integrointijärjestys voidaan toteuttaa pääsääntöisesti kolmella eri tavalla, mutta olemassa on myös niin kutsuttu ”kertarysäystesti”. (Kasurinen 2013, 54–55.)

Alhaalta ylöspäin (engl. bottom up testing) -menetelmässä järjestelmään tuodaan aluksi matalimman tason moduulit, jotka kommunikoivat suoraan käyttöjärjestelmän tai raudan kanssa. Tämän jälkeen aiemmin integrointitestattuja moduuleita käyttäviä uusia komponentteja tuodaan järjestelmään, kunnes kaikki osat ovat liitetty järjestelmään. Alhaalta ylöspäin menetelmällä saadaan helposti esiin matalan tason virheitä, jotka voisivat muuten olla hankalia havaita. (Kasurinen 2013, 55.)

Ylhäältä alaspäin (engl. top down testing) -menetelmä toimii samalla periaatteella kuin alhaalta ylöspäin, mutta testaus aloitetaan järjestelmähierarkian yläpäästä, josta edetään kohti matalimman tason moduuleja. Tällä menetelmällä saavutetaan tavallisesti parempi yleiskuva testauksen kohteena olevasta järjestelmästä, jolloin mahdolliset puutteet toiminnoissa paljastuu aikaisemmin. (Kasurinen 2013, 55.)

Kolmas menetelmä eli voileipätestaus (engl. sandwich testing) yhdistää yllä olevat menetelmät ja lähestyy integraatiota samanaikaisesti molemmista suunnista. Voileipätestauksessa etuna on, että testikomponentteja tarvitaan normaalia vähemmän, mutta loppua kohden mentäessä haasteeksi voi kuitenkin muodostua osien yhteensovittaminen. (Kasurinen 2013, 55.)

Niin sanotussa ”kertarysäystestissä” eli big bang -integrointitestauksessa kaikki yksikkötestatut moduulit liitetään yhteen kerralla kokonaiseksi järjestelmäksi. Haasteena menetelmässä on löydettyjen virheiden paikantaminen, sillä yksittäisten moduulien rajapintojen toimivuuteen ei pystytä syventymään tarpeeksi. Usein kertarysäystissä jää lisäksi huomaamatta kriittisiä virheitä, jotka saattavat tulla esiin vasta, kun ohjelmaa käytetään sen kohdeympäristössä. (Tutorialspoint 2023.)

Järjestelmätestaus (engl. system testing) on kolmas vaihe testauksen V-mallissa. Kun yksikkötestauksessa on testattu jokainen komponentti, ja integrointitestauksessa komponenteista on yhdistetty toimiva kokonaisuus, on vuorossa järjestelmätestaus. Järjestelmätestaus on yleisnimitys kaikelle testaukselle, jota tehdään kokonaiselle järjestelmälle, joka ei sisällä enää minkäänlaisia integrointitestauksessa tarvittuja ja luotuja testikomponentteja. Järjestelmätestaus suoritetaan testiympäristössä eikä

varsinaisessa kohdeympäristössä, sillä järjestelmään tulee tavallisesti vielä muutoksia ja virheitä etsitään vielä myös yksittäisistä moduuleista. Järjestelmätason testejä voidaan joissain tapauksissa toteuttaa hyvinkin aikaisessa vaiheessa, esimerkiksi jos tuotekehityksellä on käytössään prototyyppejä. (Kasurinen 2013, 56–57.) Järjestelmätestauksen suorittajan tulisi kyetä ajattelemaan mahdollisimman samalla tavalla kuin loppukäyttäjä ja olla joku muu kuin itse ohjelman kehittäjä. Ohjelman kehittäjillä on usein tunnesiteitä kehittämäänsä ohjelmaa kohtaan sekä toive saada testaus sujuvasti ja aikataulun mukaisesti suoritettua. Tälloin todellista motivaatiota ohjelmassa olevien mahdollisten vikojen löytämiseen ei välttämättä synny. (Myers ym. 2012, 104.)

Viimeinen V-mallissa esitetty testauksen taso on hyväksymistestaus (engl. acceptance testing). Tässä vaiheessa osoitetaan, että järjestelmä täyttää vaatimukset, jotka järjestelmälle oli määritelty, ja että ohjelma on riittävän korkealaatuinen. Toisin kuin järjestelmätestauksessa, hyväksymistestauksessa järjestelmää käytetään sen kohdeympäristössä. (Kasurinen 2013, 57.) Hyväksymistestauksen suorittaa usein asiakas tai loppukäyttäjä, joka pyrkii testitapauksilla osoittamaan kohdat, joissa ohjelma ei täytä alkuperäistä vaatimusta. Jos ohjelma täyttää vaatimukset, ohjelma hyväksytään. (Myers ym. 2012, 104.)

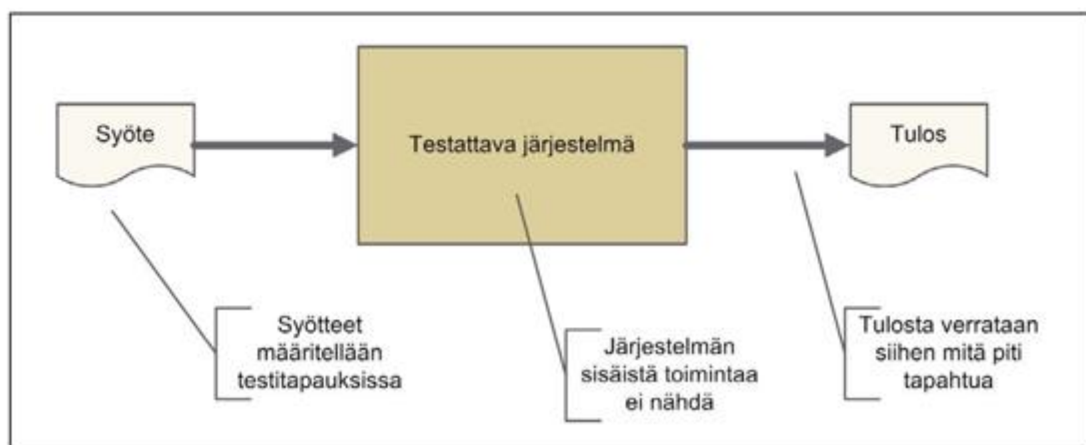
2.3 Testausmenetelmät

Kehitysvaiheessa testaus noudattaa V-mallissa esiteltyä lähestymistapaa, jossa testaus aloitetaan komponenttien yksikkötestauksella, minkä jälkeen siirrytään komponentit yhdistävään integrointitestaukseen, jota seuraa järjestelmän kokonaisuuden testaaminen järjestelmätestauksessa. Viimeisessä vaiheessa järjestelmä hyväksytään hyväksymistestauksessa. Näiden testaustasojen sisällä testausta tehdään monella eri tavalla, joita kutsutaan testausmenetelmiksi. (Kasurinen 2013, 64.)

Testausmenetelmät voidaan jakaa kahteen termiin, staattinen ja dynaaminen testaus. Staattinen testaus kattaa revisioinnit, läpikäynnit ja muut tarkastelut. (Turun ammattikorkeakoulu 2019.) Staattisessa testauksessa järjestelmää ei varsinaisesti käytetä, vaan testaus tapahtuu tutkimalla järjestelmää esimerkiksi koodiarviointien tai arkkitehtuurisuunnittelun näkökulmasta. Staattinen testaus pyrkii löytämään perustason ongelmat ja poistamaan ohjelmakoodin kielioppivirheet, eli syntaksivirheet sekä tarkastamaan, että järjestelmän sisäinen logiikka on kunnossa. (Kasurinen 2013, 65.)

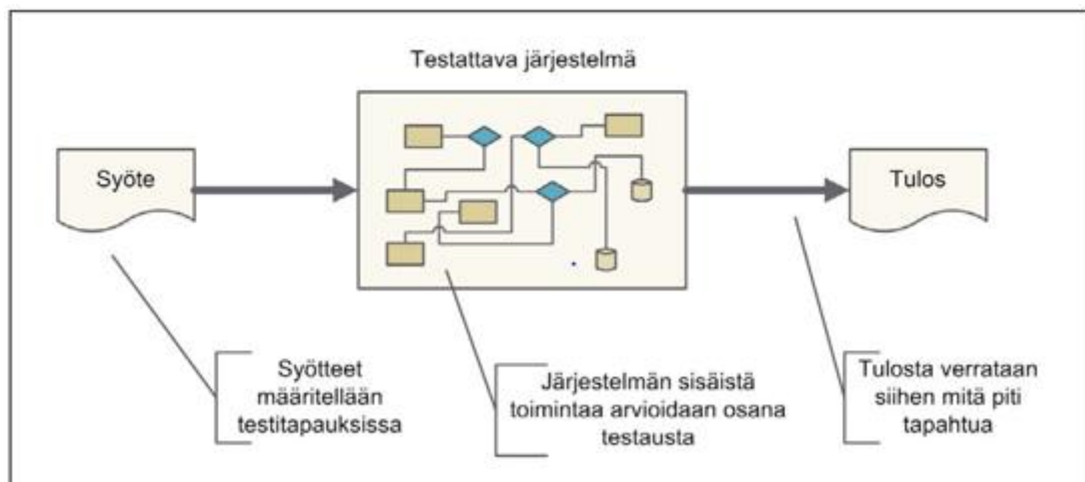
Dynaamisella testauksella tarkoitetaan käytännön testausta, kuten esimerkiksi koodin testaamista testitapauksilla kehitysvaiheen aikana, jo ennen kuin ohjelma on täysin valmis (Turun ammattikorkeakoulu 2019). Toisin kuin staattisessa testauksessa, dynaamisessa testauksessa järjestelmää varsinaisesti käytetään sen testaamiseksi. Useimmat testauksen tasot kuuluvat dynaamisen testauksen piiriin, sillä niissä tarkastellaan järjestelmän reaktioita järjestelmälle annettuihin syötteisiin käytännössä. Ilmeisin ero näiden kahden menetelmän välillä on, että staattisen testauksen voi aloittaa heti esimerkiksi arkkitehtuurisuunnitelman perusteella. (Kasurinen 2013, 65.)

Perinteisin testausmuoto testausmenetelmistä on Musta laatikko -testaus (engl. black box testing) (kuva 5), jossa ohjelmalle annetaan syötteitä ja tarkastellaan ohjelman lopputulosta, eikä sitä mitä ohjelman sisällä tapahtuu. Ohjelmalle annettavat syötteet ja tehtäväsarjat määritellään normaalisti testitapauksissa, joissa kerrotaan miten laitetta tulisi käyttää ja mihin lopputulokseen annetuilla syötteillä tulisi päästä. Musta laatikko -testeillä voidaan testata esimerkiksi tiedostojen avaamista ja eri painikkeista tapahtuvia toimintoja sekä tarkastaa helposti, miten ohjelma reagoi haitallisiin tai virheellisiin syötteisiin, kuten esimerkiksi arvoasteikon ulkopuolella oleviin lukuarvoihin. Testaajan tehtävänä on varmistaa, että annetut syötteet ovat oikein, ja että lopputulos on halutun kaltainen. Mikäli näin ei ole, testaaja kirjaa virheen tai ongelman ylös, minkä jälkeen ohjelmoija tutkii mitä ohjelman tai laitteen sisällä tapahtuu, ja miksi ohjelma ei toimi toivotulla tavalla. Koska musta laatikko -testaus on yksinkertaisesti syötteen antamista ohjelmalle ja tuloksen vertaamista siihen, mitä piti tapahtua, soveltuvat musta laatikko -testit hyvin automatisoitaviin testitapauksiin. Musta laatikko-testausta voidaan käyttää jokaisissa testauksen tasojen vaiheissa. (Kasurinen 2013, 65–66.)



Kuva 5. Ohjelmalle annetaan syöte ja syntyvää tulosta verrataan toivottuun lopputulokseen (kuva Kasurinen 2013, 66).

Lasilaatikkotestauksessa (kuva 6) testaaja näkee, mitä annetulle syötteelle tapahtuu ohjelman sisällä. Testausmenetelmänä lasilaatikkotestaus on musta laatikko -testausta syvällisempi ja tarkempi, sillä menetelmässä voidaan tarkistaa ohjelman toimivuus aina lähdekooditasolle asti ja varmistua siitä, että ohjelma ei sisällä virheitä eikä tulos ollut sattuma. Koska lasilaatikkotestaus on täydellisempi versio musta laatikko -menetelmästä, vaatii menetelmä myös testaajalta järjestelmän hyvää tuntemusta sekä ohjelmointityön ymmärrystä. Näin ollen lasilaatikkotestauksen etuna on se, että testaajat pystyvät havaitsemaan ohjelmakoodista sellaisia ongelmia ja virheitä, joita voisi olla erittäin vaikea saada esiin musta laatikko -testauksella. (Kasurinen 2013, 67-68.)



Kuva 6. Lasilaatikkotestauksessa ohjelmalle annetaan syöte ja tulosta verrataan toivottuun lopputulokseen, minkä lisäksi myös järjestelmän sisäistä toimintaa tarkastellaan (kuva Kasurinen 2013, 67).

Harmaa laatikko -testaus (engl. grey box testing) on lasilaatikko- ja musta laatikko -testauksen yhdistelmä. Perusajatuksena on yhdistää molempien testausmenetelmien parhaat puolet. Käytännössä harmaa laatikko -testauksella pyritään testaamaan mallikattavuus eli vaatimusmäärittelyn vaatimuksien täyttyminen sekä koodikattavuus eli että kaikki lähdekoodi on varmasti tarkastettu, yhtäaikasesti. Testausmenetelmä sopii tapauksiin, jossa kaikkiin komponentteihin ei päästä käsiksi lasilaatikkotestauksen vaatimalla syvyydellä. Esimerkiksi verkkokaupan tekemiseen käytetty järjestelmä pystytään testaamaan ohjelmakoodin tasolle asti, mutta verkkokaupassa suoritettavien maksujen varmentamiseen käytetyn rajapinnan toiminnan voi testata vain musta laatikko -menetelmällä, sillä rajapinta ei ole käytettävissä kokonaisuutena. (Kasurinen 2013, 68.)

Regressiotestauksella tarkoitetaan uudelleentestaamista, eikä se ole varsinaiseti erillinen testausmuoto, kuten aiemmat testausmenetelmät. Kun toimivaan järjestelmään

tehdään jokin muutos ja halutaan testata, että järjestelmä toimii muutoksesta huolimatta edelleen oikein, puhutaan regressiotestauksesta. Myös silloin, kun kehitettävä järjestelmä on saavuttanut osatavoitteen, ja sen kehitysversion kaikkien toimintojen oikeellisuus halutaan varmentaa, voidaan puhua regressiotestauksesta. Tärkein tavoite regressiotestauksessa on todentaa, ettei komponentteihin tehtyjen muutosten jälkeen kertaalleen korjatut ongelmat esiinny uudestaan, ja lisäksi ettei mitään uutta ole mennyt rikki. Koska regressiotestit suoritetaan kehityksen jokaiselle pääversiolle tai osatavoitteiden täyttyessä, toistetaan testitapauksia useaan kertaan projektin aikana. Testien toistuvuus on testausautomaation keskeisempiä vaatimuksia, minkä takia regressiotestauksessa suoritettavat testitapaukset soveltuvat hyvin testausautomaation käyttämiselle. (Kasurinen 2013, 68–70.)

Kaikkea testaamista ei kuitenkaan voi toteuttaa kehitysvaiheessa. Testaukseen kuuluu myös työvaiheita, joissa testattava ohjelmisto on jo lähes valmis tai enimmillä ominaisuuksilla varustettuna, mutta ohjelmistoa ei ole vielä julkaistu eikä luovutettu asiakkaalle. Käytettävyystestauksella tarkoitetaan testaamista, jossa tarkastetaan, että käyttöliittymä on suunniteltu oikein. Käytettävyyttä voidaan testata esimerkiksi käyttökokeiluilla, joita seuraavat haastattelututkimukset sekä kohdekäyttäjäryhmän tekemällä tutkivalla testauksella. (Kasurinen 2013, 70.)

Kuormitustestaus (engl. load testing) testaustoiminnalla ohjelmaa kuormitetaan luomalla esimerkiksi virtuaalikäyttäjiä, jotka simuloivat normaalia toimintaa testitapauksien avulla. Esimerkiksi verkkokaupan kuormitustestauksessa virtuaalikäyttäjät luodaan kuormittamaan verkkokaupan erilaisia toimintoja: osa selailee tuotteita, osa ostaa ja tilaa ja tuotteita ja osa simuloi verkkokaupan satunnaista kävijävirtaa avaamalla ja sulkemalla verkkosivu. Kuormitustestauksella pyritään tunnistamaan ohjelman ongelmia aiheuttavat kohdat, kuten esimerkiksi tapahtumien vasteaikoja muistin ja suorittimen ollessa kuormitettuna tarkastelemaan, miten testattava tuote selviytyy normaaleista käyttöolosuhteista ja selvittämään ohjelman maksimikapasiteetti. Kuormitustestauksen seuraavana tasona voidaan pitää rasitustestausta, jossa ohjelma laitetaan suuremman rasituksen alle, kuin mihin se on normaaleissa käyttöolosuhteissa suunniteltu. Koska kuormitustestaus ja rasitustestaus on vaativat paljon työtä ja testisyötteitä, tällainen testaus on hyvin usein automatisoitu. (Kasurinen 2013, 71-72.)

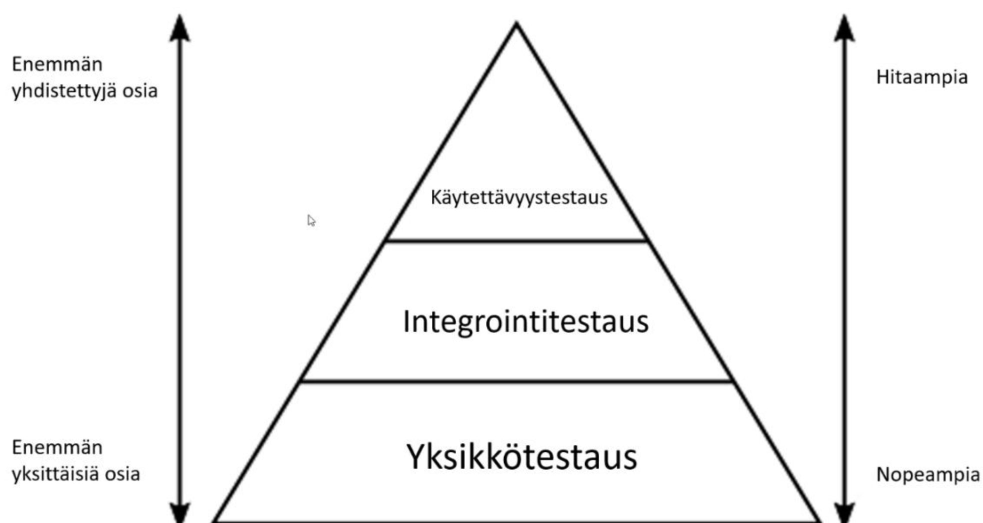
Ohjelman perusasioita testataan niin sanotulla savutestauksella (smoke test). Savutestauksessa käydään läpi tarkastuskohtia, kuten esimerkiksi lähteekö ohjelma käyntiin tai toimivatko kaikki päävalikon näppäimet. Näillä varmistetaan minimitaso

ohjelman toimivuudelle, jotta testaajien resursseja ei tarvitse tuhlata projekteihin, joissa ei toimi edes alkeelliset perustason toiminnot. (Kasurinen 2013, 72.)

3 TESTAUSAUTOMAATIO

Testausautomaatio on testausta, jossa automaattityökalujen avulla rakennetaan testejä ohjelman testaamista varten. Testausautomaation tavoitteena on luoda usein toistuven testitapausten tehokasta tarkastamista varten testausympäristö ja vapauttaa testaajien resursseja muihin tehtäviin. Testaajien ajankäytön kannalta ei ole esimerkiksi kannattavaa testata jokaöisen uuden ohjelmakäännöksen (engl. nightly build) jälkeen päivittäin samoja perustestejä. Automaattitestit voivat myös olla tarkastuksia, jotka suoritetaan yön aikana, ja joiden tulokset tarkastetaan esimerkiksi joka aamu testaajien tai kehittäjien toimesta. (Kasurinen 2013, 76.)

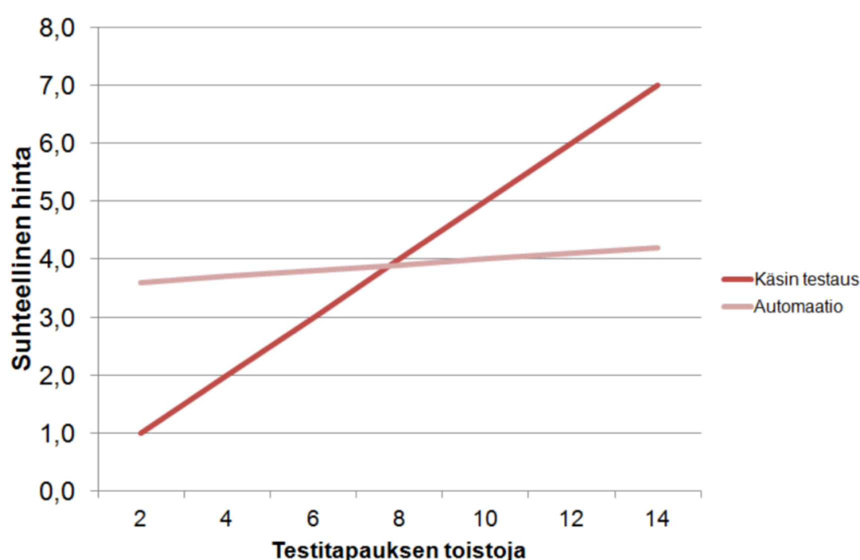
Kaikkien testien automatisointi ei ole kannattavaa. Testauspyramidin (kuva 7) avulla voidaan tarkastella testauksen automatisoinnin kannattavuutta. Vaikka testauspyramidin on kritisoitu olevan liian yksinkertainen ja näin ollen harhaanjohtava, on pyramidin perusajatus kuitenkin toimiva ohjenuora testausautomaation kehittämisessä. Pyramidin alimmaisella tasolla on yksikkötestaus, seuraavalla tasolla integrointitestaus ja ylimmällä tasolla käyttöjärjestelmän testaus. Alimmaisen tason yksikkötestaus ja keskimmäisen tason integrointitestaus ovat testausautomaatiolle suotuisimmat kohteet. Yksikkötestaus on nopeaa pienempien osien testausta, mikä mahdollistaa myös testitapausten vähemmän resursseja vaativan ylläpidon (Vocke 2018.)



Kuva 7. Testauspyramidin alemman tason testaus on kannattavampaa ja kestävämpää automatisoida kuin ylemmän tason testaus (kuva Vocke 2018).

Testausautomaation ajatellaan usein virheellisesti korvaavan kokonaan käsintestauksen, vaikka parhaimmillaan testausautomaatio on vain täydentävänä osana sitä. Testausautomaation avulla säästäminen on myös yksi tavallisimmista väärinkäsityksistä, sillä testausautomaation hyödyntäminen tehokkaasti vaatii paljon työtä ja ylläpitoa. Tällainen väärinkäsitys saattaa johtaa konflikteihin ja lopulta testausautomaatiohankkeen kaatumiseen. (Kasurinen 2013, 76.)

Testausautomaation ylläpito ja käyttö on helpompaa ja halvempaa, kun käsin toistettavien testien määrä ylittää tietyn pisteen (kuva 8). Yhden testitapauksen automatisointi on kallista ja hidasta, mutta testitapaus, joka voidaan toistaa muuttumattomana useaan kertaan on helppo toimenpide testausautomaatiolle, eikä se maksa käytännössä mitään. Riippuen lähteestä, testien määrän toistuessa 4-20 kertaa projektina aikana, on hyvä harkita testien automatisointia. Normaalisti noin neljäsosa testeistä toistetaan yli 20 kertaa, mutta usein myös harvemmin suoritettavat testit toistetaan ainakin viidesti. Esimerkiksi integraatio-, yksikkötestaus ja savutestit suoritetaan aina uudelleen, kun on tarve testata järjestelmä kattavalla regressiotestauksella läpi. Testausautomaatio on parhaimmillaan varmistettaessa, että aiemmin toimineet testit toimivat edelleen, ja ettei jo ennestään toimineet moduulit ole rikkoutuneet kehitystyön edetessä. Uusien osien toimivuuden testaamisessa testausautomaatio olisi resurssien tuhlausta, sillä tämä vaatisi testien jatkuvaa ylläpito- ja muutostyötä. (Kasurinen 2013, 77-79.)



Kuva 8. Kustannuskäyrä automatisoiduille testitapauksille (kuva Kasurinen 2013, 78).

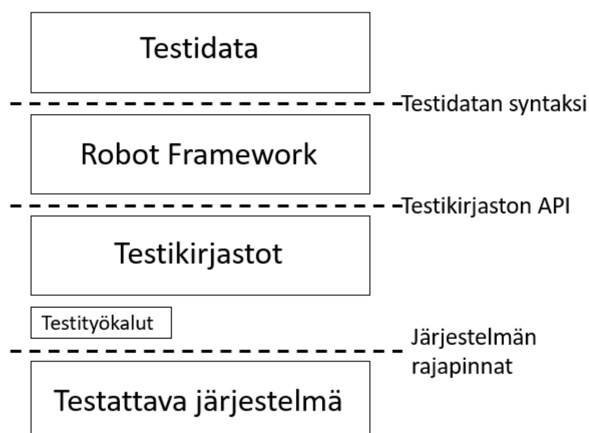
4 TYÖKALUT

Ohjelmiston testaukseen soveltuu hyvin usein samat työvälineet kuin itse ohjelmiston kehittämiseenkin. Kehitysympäristö sekä kommunikointivälineet, kuten esimerksi järjestelmä, jonka kautta voidaan ilmoittaa havaitut ongelmat eteenpäin, ovat ohjelmistotestaajien tärkeimmät työvälineet. Testausautomaation työkalut ovat usein organisaation itsensä kehittämiä testaustyökaluja. Testausautomaation kehittämisen avuksi löytyy kaupallisia työkaluja ohjelmistoyrityksiltä, mutta myös avoimen lähdekoodin rajapintoja on saatavilla. (Kasurinen 2013, 84-85.)

Testausautomaatioon soveltuvia työkaluja on olemmassa laaja kirjo, joten jokaisen on valittava sopivin työkalu projektilleen. Työkalujen kustannuksissa on suuria eroja, mikä on hyvä huomata etsittäessä soveltuvinta vaihtoehtoa testausautomaation kehittämiseen. Osa työkaluista on ilmaisia, kun taas osasta työkaluista voi joutua maksamaan huomattavia summia. (Guru99 2025.)

4.1 Robot Framework

Robot framework on yleinen avoimen lähteen kehys testausautomaation kehittämiseen. Sen käyttö pohjautuu Python-ohjelmointikieleen, ja sitä voidaan hyödyntää ympäristöissä, joissa testausautomaatio edellyttää erilaisten teknologioiden ja rajapintojen käyttöä. Robot framework käyttää helposti luettavissa muodoissa olevia avainsanoja ja muodostaa testien tuloksista selkeitä html-raportteja, mikä tekee Robot frameworkin käytöstä yksinkertaista ja tehokasta. (Robot Framework Foundation 2016.)



Kuva 9. Robot frameworkin modulaarinen arkkitehtuuri (Robot Framework Foundation 2016).

Testikirjastot (kuva 9) luovat pohjan Robot frameworkin testaukselle. Robot frameworkin avulla on mahdollista sekä hyödyntää jo olemassa olevia testikirjastoja että luoda kokonaan uusia testikirjastoja Pythonin avulla, jotta testit soveltuvat paremmin testattavan järjestelmän tarpeisiin. (Robot Framework Foundation 2016.)

4.2 Python-ohjelmointikieli

Guido van Rossum loi Python-ohjelmointikielen, ja se julkaistiin käyttöön ensimmäisen kerran vuonna 1991. Vaikka useat eri ohjelmoijat ja käyttäjät ovat olleet mukana kehittämässä Pythonia, on sen idea alunperin vain yhden ihmisen luoma. Van Rossum asetti Pythonille tavoitteet olla selkeästi ymmärrettävää englannin kielellä ja sopiva jokapäiväiseen lyhyiden kehitystehtävien tekoon, ja onnistui siinä. Python on helppo oppia, ja sen avulla on nopeaa luoda uutta koodia. Python on maksuton ja sopii yhteen useiden eri alustojen kanssa, näin ollen Pythonista on tullut suosittu ohjelmointikieli. (Open Education and Development Group 2022.)

4.3 Robot Framework IDE

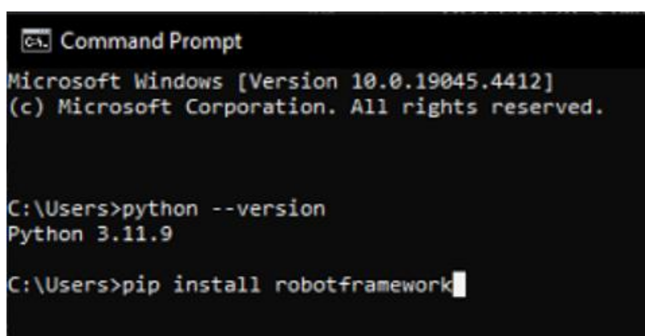
Robot Framework IDE (Integrated Development Environment) tarjoaa selkeän käyttöliittymän testien luomiseen sekä luotujen testien mahdollisten virheiden etsimiseen. Lisäksi itse testien ajaminen on mahdollista RIDEn käyttöliittymän kautta. (SoftwareTestingHelp 2023.) RIDE päädyttiin kuitenkin jättämään testausautomaation kehitysvaiheessa pois ja sen käyttöä harkitaan, kun testien kehitys on saatu pidemmälle.

5 TESTIEN TOTEUTTAMINEN

5.1 Testiympäristö

Ensimmäisenä tietokoneelle asennettiin Visual Studio Code ja Python-ohjelmointikieli. Visual Studio Code valittiin tiedostojen tekstieditoriksi, sillä ohjelma korjaa yleisimmät syntaksi- eli kielioppivirheet koodissa, tarjoaa Gitin käyttöön selkeän visuaalisen käyttöliittymän sekä näyttää funktioiden (aliohjelmien) määrittelyt (Microsoft 2023b; Microsoft 2023a). Tämän jälkeen täytyi asentaa Python 3.7.x. ja tarkistaa, että asennus vaiheessa ”Määritä ympäristömuuttujat” (Modify environment variables) on valittuna, jotta ympäristömuuttujien polku tulee tallennettua oikein. Robot frameworkin asennukseen oli helpoin käyttää pip-paketinhallintatyökalua, joka tulee 3.7.4 ja uudempien Python versioiden mukana automaattisesti.

Robot frameworkin sai asennettua kirjoittamalla komentoriville (enlg. Command Prompt) käsky *pip install robotframework* (kuva 10).



Kuva 10. Robot frameworkin asennus pip installin avulla.

Vaikka yrityksellä oli jo olemassa tarvittavat DLL-tiedostot, jotka toimivat rajapintana kommunikoinnin mahdollistamiseksi lastauskoneen ohjausyksikön (engl. MCU, Machine Control Unit) ja näytön (SUP) sekä simulaatio-ohjelmiston välillä, täytyi Robot-tiedostojen ja DLL-tiedostojen väliin luoda vielä Python-tiedostot. Kommunikointi koneen ohjausyksikön välillä kulkee MCON-yhteyden kautta, joka mahdollistaa koneen näytössä ja ohjausyksikössä kulkevien signaaleiden arvojen seuraamisen ja muuttamisen. MCON-yhteydelle ja simulaatioyhteydelle luotiin erilliset Python-tiedostot, jotka sisältävät tarvittavat funktiot erilaisten komentojen välittämiseen niin koneen ohjausyksikölle kuin simulaatio-ohjelmistollekin. Edellä mainittujen tiedostojen lisäksi täytyi luoda kaksi

konfiguraatiotiedostoa. Toinen tiedosto sisältää koneen näytön ja ohjausyksikön signaalitiedot, ja toinen sisältää simulaatio-ohjelmistossa käytössä olevat signaalit. Näiden tiedostojen avulla on mahdollista pyytää toimintoja simulaatiolta ja koneen ohjausyksiköltä ja varmistaa, että koneen ohjausjärjestelmä reagoi annettuihin komentoihin vaaditulla tavalla.

Testitapaukset luotiin Robot-tiedostoihin, joihin on luotu testien toteuttamiseen tarvittavat avainsanat ja asetukset. Avainsanojen toiminta on määritelty resurssitiedostoissa, jotta Robot-tiedostot pysyvät selkeinä ja helppolukuisina. Robot frameworkilla on saatavilla valmiita resurssitiedostoja, mutta niiden soveltuessa huonosti sellaisenaan lastauskoneiden ohjausjärjestelmän testaamiseen, luotiin yrityksen käyttöön täysin omat resurssitiedostot. Yksinkertaisimmat avainsanat toteutettiin resurssitiedostoissa, mutta monivaiheisten avainsanojen toiminnallisuudet määriteltiin erillisissä Python-tiedostoissa.

Yrityksellä oli manuaalitestauksen regressiotestien testitapaukset dokumentoituna yrityksen käyttämässä tarpeiden hallintatyökalussa. Näiden testitapausten pohjalta lähdettiin luomaan testiautomaatioon testitapauksia. Testit ajettiin virtuaalisella koneella, jossa lastauskoneen toiminnot ovat simuloitu, mutta koneen ohjausyksikkö ja näyttö ovat mukana fyysisenä laitteistona (engl. hardware).

5.1.1 Robot-tiedosto

Ensin luotiin ylätason Robot-tiedosto vastaamaan tarpeiden hallintatyökalussa olevaa testitapausta. Tarkoituksena oli luoda testiautomaatiotesteistä ja avainsanoista ymmärrettäviä ja helposti luettavia, joten Robot-tiedoston ohjelmakoodit käyvät testitapauksen dokumentaationa sellaisenaan. Robot-tiedostoon luotiin ensin asetukset (Settings), joihin sisällytettiin tarpeiden hallintatyökalusta löytyvä testitapauksen dokumentaatio (Documentation), tarvittavat resurssitiedostot (Resource), mahdolliset testitunnisteet (Test Tags), koko testitapauksen esiehdot ja jälkiehdot (Suite Setup ja Suite Teardown) sekä testitapauksen yksittäisten testien esiehdot ja jälkiehdot (Test Setup ja Test Teardown) (kuva 11).

```
ParkingBrake.robot
1   *** Settings ***
2   Documentation      Test cases for Engine Start: Parking brake button released
3
4   Suite Setup        Test Suite Setup
5   Test Setup         Test Setup
6   Test Teardown      Test Teardown
7   Suite Teardown     Test Suite Teardown
8
9   Resource           ../Resources/Machine.resource
10  Test Tags          Brakes    ParkingBrake
11
```

Kuva 11. Seisontajarrun vapautus -testitapauksen asetukset.

Asetusten jälkeen määriteltiin testitapauksen esiehto- ja jälkiehtovaiheet avainsanoihin (Keywords). Testitapauksen esiehdot ajetaan vain kerran, joko aloitettaessa koko testitapauksen tai yksittäisen testin ajo, kun taas testin esiehdot ajetaan aina ennen yksittäisen testin alkua. Testin jälkiehdot ajetaan jokaisen yksittäisen testin jälkeen ja testitapauksen jälkiehdot koko ajon lopuksi. Esiehto- ja jälkiehtovaiheet ovat tärkeitä, jotta ohjausjärjestelmä on oikeassa tilassa testiajon alkaessa ja loppuessa. Testitapauksen esiehdot ja jälkiehdot sisältää simulaattoriympäristön asettamisen sopivaan tilaan (Initialize Simulator environment) niin alkavaa testiä, kuin lopuksi sitä mahdollisesti seuraavia testitapauksia varten. Testin esiehtovaihe sisältää avainsanat simuloidun lastauskoneen virtojen kytkemisestä päälle (Turn Ignition key to Run) ja tarkistuksen, ettei koneen moottori ole käynnissä (Check that Engine is not running). Testin jälkiehtovaiheessa avainsanat sammuttaa moottorin (Press start button to Stop the engine), kääntää koneen virrat pois päältä (Turn Ignition key to Park) ja tarkistaa, ettei moottori ole käynnissä. (Kuva 12).

```
*** Keywords ***
Test Suite Setup
    Initialize Simulator environment

Test Setup
    Turn Ignition key to Run
    Check that Engine is not running

Test Teardown
    Press start button to Stop the engine
    Turn Ignition key to Park
    Check that Engine is not running

Test Suite Teardown
    Initialize Simulator environment
```

Kuva 12. Seisontajarrun vapautus -testitapauksen esiehdot ja jälkiehdot.

Seisontajarrun vapautus -testitapaus sisältää kaksi yksittäistä testiä, joissa on kummassakin kolme avainsanaa testin toteuttamiseen. Ensimmäisen testin tarkoitus on varmistaa, ettei moottori käynnisty seisontajarrun ollessa vapautettuna (TEST: Verify that engine is not started while parking brake released). Testin esiehtojen jälkeen simulaatio on tilassa, jossa lastauskoneen virrat ovat päällä ja moottori ei ole käynnissä. Testi alkaa avainsanalla, joka vapauttaa seisontajarrun (Release Parking brake). Tämän jälkeen koitetaan käynnistää moottori (Press start button to Start engine). Lopuksi tarkistetaan, ettei moottori ole käynnissä (Check that Engine is not running), sillä jos seisontajarru on vapautettu, moottorin ei kuulu käynnistyä. Toinen testi tarkistaa, että moottori käynnistyy, jos seisontajarru on kytketty (TEST: Verify that engine is started while parking brake engaged). Testi alkaa taas testin esiehtojen jälkeen tilasta, jossa lastauskoneen virrat ovat päällä moottorin ollessa sammutettuna. Seisontajarru kytketään päälle (Engage Parking Brake) ja moottori käynnistetään. Näiden avainsanojen jälkeen tarkistetaan, että moottori on käynnissä (Check that Engine is running). (Kuva 13).

```
*** Test Cases ***
TEST: Verify that engine is not started while parking brake released
    [Documentation]    Engine should not be started when parking brake is released
    Release Parking brake
    Press start button to Start engine
    Check that Engine is not running

TEST: Verify that engine is started while parking brake engaged
    [Documentation]    Engine should start when parking brake is engaged
    Engage Parking brake
    Press start button to Start engine
    Check that Engine is running
```

Kuva 13. Seisontajarrun vapautus -testitapauksen testit.

5.1.2 Resurssitiedosto

Robot-tiedostossa käytössä olevat avainsanat ovat määritelty resurssitiedostoon (Machine.resource), jotta Robot-tiedosto pysyy mahdollisimman selkeänä ja helppolukuisena. Resurssitiedoston alussa on asetukset (Settings), jotka sisältävät tarvittavat Python-kirjastot (Library) ja muuttujat (Variables). Machine.resource -tiedoston avainsanojen teknisemmät toiminnallisuudet ovat määriteltynä Python-kirjastoon (MachineFunctions.py). Muuttujina on annettu avainsanojen tarvitsemat arvot, kuten esimerkiksi moottori käynnissä (ENGINE_RUNNING) arvolla ${1} ja moottori sammutettu (ENGINE_NOT_RUNNING) arvolla ${0}. (Kuva 14).

```
Machine.resource
1   *** Settings ***
2   Library          MachineFunctions.py
3
4   *** Variables ***
5   ${RELEASED}               ${1}
6   ${ENGAGED}                ${0}
7
8   ${RUN}                    ${1}
9   ${PARK}                   ${0}
10
11  ${ENGINE_RUNNING}         ${1}
12  ${ENGINE_NOT_RUNNING}     ${0}
```

Kuva 14. Resurssitiedoston asetukset ja muuttujat.

Seuraavana resurssitiedostoon on listattu avainsanat, joissa kutsutaan Python-kirjastossa sijaitsevia funktioita (kuva 15).

```
*** Keywords ***
Turn Ignition key to Run
    forceIgnitionKey    ${RUN}

Turn Ignition key to Park
    forceIgnitionKey    ${PARK}

Press start button to Start engine
    pressEngineStartbutton

Press start button to Stop engine
    pressEngineStartbutton

Check that Engine is running
    ${engine_state}    getEngineState    ${MCON_ENGINE_STATE_SIGNAL}
    Should Be Equal    ${engine_state}    ${ENGINE_RUNNING}

Check that Engine is not running
    ${engine_state}    getEngineState    ${MCON_ENGINE_STATE_SIGNAL}
    Should Be Equal    ${engine_state}    ${ENGINE_NOT_RUNNING}

Engage Parking brake
    forceParkingBrake    ${ENGAGED}

Release Parking brake
    forceParkingBrake    ${RELEASED}

    ${cur_current}         checkServiceBrakeControlCurrent         ${req_current}
    Should Be True      ${cur_current}

Check brake circuit more than neutral limit
    [Arguments]         ${Circuit}     ${REQ_STATE}
    ${NEUTRAL_LIMIT}        Common.Read from MCON signal        ${SIGNAL_BRAKES_NEUTRAL_ENGAGE_PRESSURE}
    ${CURRENT_STATE}        checkBrakeCircuitNeutralLimit     ${Circuit}    ${NEUTRAL_LIMIT}
    Should Be Equal     ${CURRENT_STATE}            ${REQ_STATE}
```

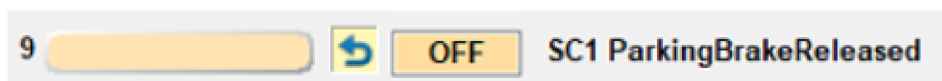
Kuva 15. Seisontajarrun vapautus -testitapauksen käyttämät avainsanat resurssitiedostossa.

Resurssitiedostossa on annettu muuttujat, joiden perusteella funktio tietää, mitä tilaa pyydetään. Esimerkiksi kytkettäessä koneen seisontajarru päälle (Engage Parking brake), funktiolla pakota seisontajarru (forceParkingBrake) ”päälle” asetetaan simulaatiossa seisontajarrun tilaksi ”0” muuttujan (ENGAGED) kautta, jolloin seisontajarru on päällä (kuva 16, kuva 17). Koneen näytöllä näkyy myös seisontajarru kytketty -tila seisontajarrun kuvakkeena (kuva 18).

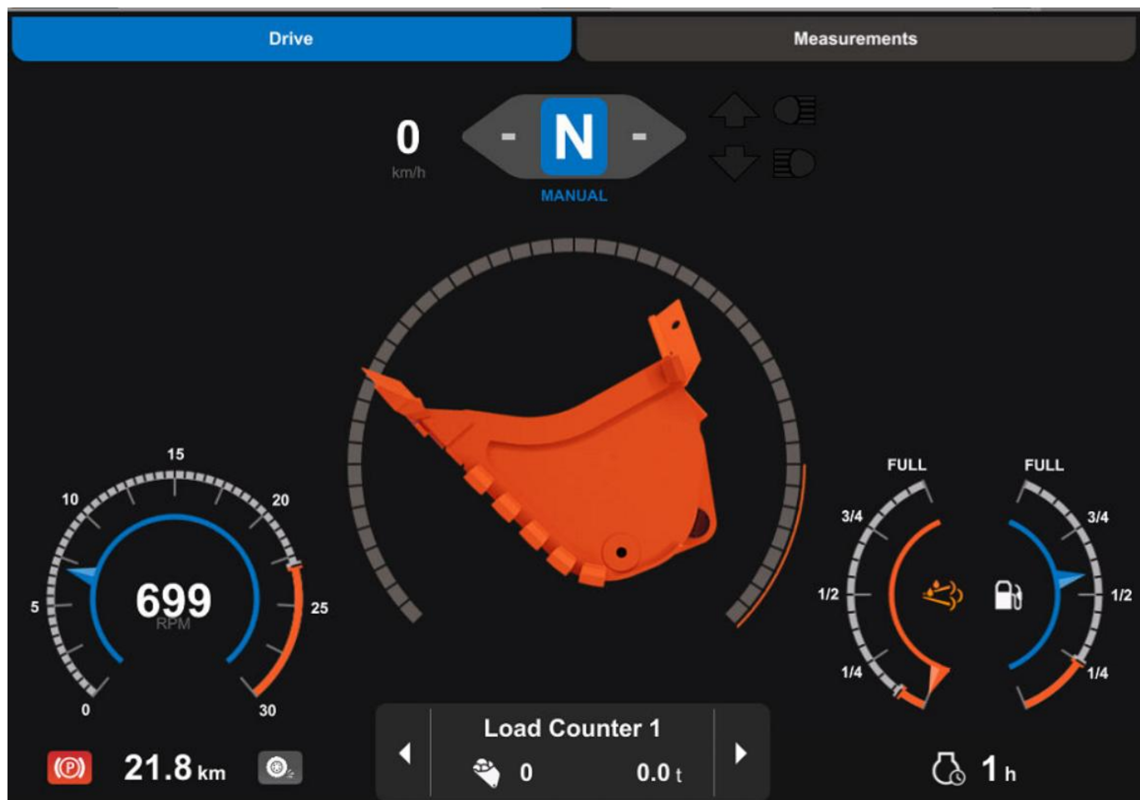
```
*** Variables ***
${RELEASED}              ${1}
${ENGAGED}               ${0}

*** Keywords ***
Engage Parking brake
    forceParkingBrake    ${ENGAGED}
```

Kuva 16. Kytke seisontajarru päälle -avainsana resurssitiedostossa.



Kuva 17. Seisontajarru on kytketty (Parking Brake Released = 0) simulaatiossa.



Kuva 18. Seisontajarru kytketty -kuvake virtuaalisen koneen näytöllä vasemmassa alareunassa.

5.1.3 Python-tiedosto

Python-tiedostoihin määriteltiin avainsanojen teknisemmät toiminnot. Alussa koneen toimintoja ohjaavaan MachineFunctions.py-tiedostoon tuotiin ohjausjärjestelmän ja simulaation rajapintafunktioita sisältävät erilliset Python-tiedostot. Näiden jälkeen tuotiin Robot frameworkin oma, lokitekstiä konsolille tulostava ”logger”-ohjelma. Lopuksi lisättiin vielä tarvittavat konfiguraatiotiedostot, joista löytyy ohjausjärjestelmän ja simulaatio-ohjelmiston välillä kulkevat signaalit. (Kuva 20.)

```
MachineFunctions.py > ...
1    from SimulatorFunctions import simulatorFunctions
2    from MconFunctions import mconFunctions
3    from robot.api import logger
4    import MconConfig as mconConf
5    import SimulatorConfig as ioConf
6
```

Kuva 20. Asetukset MachineFunctions.py-tiedostossa.

Tämän jälkeen tiedostoon kirjoitettiin avainsanojen tarvitsemat toiminnot funktioina. Funktioiden nimet haluttiin pitää mahdollisimman selkeinä ja toimintaa kuvaavina. Seisontajarrun vapautukseen ja kytkemiseen käytetään samaa funktiota pakota seisontajarru (forceParkingBrake). Funktiolle määriteltiin muuttuja ”sStatus”, joka saa arvon resurssitiedostossa. Kirjoitus simulaattorille tapahtuu rivillä 11 funktiolla kirjoita simulaattorille (writeToSimulator). Kirjoita simulaattorille -funktio kirjoittaa simulaattorille muuttujan sStatus arvon, joka on resurssitiedostossa annetusta tilasta riippuen joko kytketty eli 0 (ENGAGED) tai vapautettu eli 1 (RELEASED). Funktion lopuksi logger.console kirjoittaa vielä komentoriville lokitekstinä, mikä tila on haluttu antaa seisontajarru vapautettu -signaalille. (Kuva 21.)

```
8
9   def forceParkingBrake(self, sStatus="0"):
10      """ Force parking brake status to 'released' (=1) or 'engaged' (=0). """
11      simulatorFunctions.writeToSimulator(ioConf.IO_SC1ParkingBrakeReleased, sStatus)
12      logger.console("Parking brake released signal is now " + str(sStatus))
13
```

Kuva 21. Funktio seisontajarrun tilan muuttamiseen simulaattorilla.

Testit ajettiin komentoriviltä suoraan, jolloin testien etenemistä on helpompaa seurata ja tulokset ovat näkyvillä heti testien päätyttyä. Funktioihin lisätyt lokitekstit ovat tärkeitä, jotta komentorivillä pystyy seuraamaan testin etenemistä. (Kuva 22.)

```
C:\git\lh-common\rnd\test_automation\robot_scripts\LHCS_scripts\DieselLoader>robot ParkingBrake.robot
Creating new MCon instance
Parsing MCON_DEBUG=MCON_DEBUG= LEVEL_ERROR+DEBUG_FILE=C:\temp\mcon_traces\mcon.log
Level 0 stored in syslib address 6ACA90DC
==============================================================================
ParkingBrake :: Test Machine RFW tests
==============================================================================
TEST: Verify that engine is started while parking brake engaged ::... Turning ignition on...
Checking that engine is not running...
Engine is not running
.Parking brake released signal is now 0
..Engine is running
.Turning ignition off...
Checking ignition status...
Ignition key in Park position
TEST: Verify that engine is started while parking brake engaged ::... | PASS |
------------------------------------------------------------------------------
ParkingBrake :: Test Machine RFW tests                                | PASS |
1 test, 1 passed, 0 failed
==============================================================================
Output:  C:\git\lh-common\rnd\test_automation\robot_scripts\LHCS_scripts\DieselLoader\output.xml
Log:     C:\git\lh-common\rnd\test_automation\robot_scripts\LHCS_scripts\DieselLoader\log.html
Report:  C:\git\lh-common\rnd\test_automation\robot_scripts\LHCS_scripts\DieselLoader\report.html
MCon:  Clean Up
```

Kuva 22. Testi, jossa tarkistetaan, että moottori käynnistyy seisontajarrun ollessa kytkettynä komentorivillä.

5.2 Testien tulokset

Robot Framework luo ja tallentaa tarkemmat testiraportit määrättyyn kansioon, mikä helpottaa testitulosten tarkastamista ajettaessa useampi testejä kerralla esimerkiksi öisin. Testiraportin yleiskatsauksessa näkee testien ajossa mahdollisesti käytetyt testitunnisteet sekä kaikkien testien tulokset, jos on ajettu useampia testejä. (Kuva 23.)

ParkingBrake Report

Generated
20250301 16:05:33 UTC+02:00
4 minutes 23 seconds ago

Summary Information

Status:	All tests passed
Documentation:	Test Machine RFW tests
Start Time:	20250301 16:05:03.644
End Time:	20250301 16:05:33.309
Elapsed Time:	00:00:29.665
Log File:	log.html

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	2	2	0	0	00:00:13	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Test	2	2	0	0	00:00:13	

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
ParkingBrake	2	2	0	0	00:00:30	

Kuva 23. Yleiskatsaus testiraportista.

Valitsemalla haluamansa testin aukeaa tarkempi lokiraportti. Lokiraportissa on nähtävillä testin jokainen vaihe ja läpimenotilasto. Lokiraportissa pystyy myös avaamaan testitapauksen esiehdot ja jälkiehdot vaihe vaiheelta, jos esimerkiksi testi on jostain syystä epäonnistunut jo alustuksessa. (Kuva 24.)

ParkingBrake Log

Generated
20250301 16:05:33 UTC+02:00
1 minute 24 seconds ago

Test Statistics

Total Statistics	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
All Tests	2	2	0	0	00:00:13	

Statistics by Tag	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
Test	2	2	0	0	00:00:13	

Statistics by Suite	Total	Pass	Fail	Skip	Elapsed	Pass / Fail / Skip
ParkingBrake	2	2	0	0	00:00:30	

Test Execution Log

SUITE ParkingBrake
Full Name: ParkingBrake
Documentation: Test Machine RFW tests
Source: C:\git\lh-common\md\test_automation\robot_scripts\LHCS_scripts\DieselLoader\ParkingBrake.robot
Start / End / Elapsed: 20250301 16:05:03.644 / 20250301 16:05:33.309 / 00:00:29.665
Status: 2 tests total, 2 passed, 0 failed, 0 skipped
SETUP Test Suite Setup
TEARDOWN Test Suite Teardown
TEST TEST: Verify that engine is not started while parking brake released
Full Name: ParkingBrake.TEST: Verify that engine is not started while parking brake released
Documentation: Engine should not be started when parking brake is released
Tags: Test
Start / End / Elapsed: 20250301 16:05:20.172 / 20250301 16:05:27.745 / 00:00:07.573
Status: PASS
SETUP Test Setup
KEYWORD ParkingBrake . Release Parking Brake
KEYWORD Engine . Press start button for ${5}
KEYWORD Engine . Check that engine is not running
TEARDOWN Test Teardown
TEST TEST: Verify that engine is started while parking brake engaged

Kuva 24. Onnistunut seisontajarru-testitapaus.

Testitapauksen epäonnistuessa lokiraportista löytyy tarkalleen vaihe, missä testi on epäonnistunut. Esimerkissä (kuva 25) on testattu, ettei moottori käynnisty, kun seisontajarru on vapautettu. Moottorin tilan tarkastus (Check engine state) on epäonnistunut, koska moottorin tilan haluttiin olevan 1 eli sammutettu, mutta simulaattorilta luettu tila oli 2 eli moottori oli käynnistymässä.

TEST TEST: Verify that engine is not started while parking brake released
Full Name: ParkingBrake.TEST: Verify that engine is not started while parking brake released
Documentation: Engine should not be started when parking brake is released
Tags: Test
Start / End / Elapsed: 20250301 16:17:02.839 / 20250301 16:17:13.624 / 00:00:10.785
Status: FAIL
Message: Keyword 'Check engine state' failed after retrying 4 times. The last error was: 4 != 1
SETUP Test Setup
KEYWORD ParkingBrake.Release Parking Brake
KEYWORD Engine.Press start button for ${5}
KEYWORD Engine.Check that engine is not running
Documentation: Checking that engine state is Off
Start / End / Elapsed: 20250301 16:17:10.195 / 20250301 16:17:13.384 / 00:00:03.189
KEYWORD BuiltIn.Log Checking that engine is not running... console=yes
KEYWORD BuiltIn.Wait Until Keyword Succeeds 4x 1s Check engine state ${1}
Documentation: Runs the specified keyword and retries if it fails.
Start / End / Elapsed: 20250301 16:17:10.197 / 20250301 16:17:13.384 / 00:00:03.187
KEYWORD Engine.Check engine state ${1}
Documentation: Get engine state value. Value interpretation: 0 = Unknown 1 = Off 2 = Starting 3 = Heating 4 = Running 5
Start / End / Elapsed: 20250301 16:17:10.197 / 20250301 16:17:10.235 / 00:00:00.038
KEYWORD ${ENGINE_STATE} = EngineFunctions.Get Engine State
KEYWORD BuiltIn.Should Be Equal ${ENGINE_STATE} ${expected_state}
Documentation: Fails if the given objects are unequal.
Start / End / Elapsed: 20250301 16:17:10.234 / 20250301 16:17:10.234 / 00:00:00.000
16:17:10.234 FAIL 2 != 1

Kuva 25. Epäonnistunut seistontajarru-testitapaus avattuna vaihe vaiheelta lokiraporttiin.

6 LOPUKSI

Tavoitteena oli automatisoida lastauskoneen simulaatiotestaus ja luoda käytännöllisiä sekä mahdollisimman tehokkaasti ylläpidettäviä testitapauksia. Tehokkaan ylläpidon edellytyksenä oli luoda myös tarvittava dokumentaatio testitapauksista, kirjastoista ja testiympäristön pystyttämisestä.

Työssä tutustuttiin maanalaisen lastauskoneen ohjausjärjestelmään, ja lastauskoneen käyttöön simulaatioympäristössä. Lisäksi ohjelmistotestaus, sen tasot ja menetelmät tulivat tutuiksi työn edetessä. Robot Framework osoittautui hyväksi testausautomaatiokehykseksi sen yksinkertaisen ja helposti luettavan syntaksin eli kieliopin ansiosta. Koulutuksia sekä ohjeita ja vinkkejä Robot Frameworkin käyttöön oli runsaasti saatavilla, minkä vuoksi Robot Framework oli helposti lähestyttävä testausautomaatiokehys.

Testauksen automatisointi saatiin työn tuloksena hyvin alkuun ja suuri osa regressiotesteistä automatisoitiin. Lisäksi luotiin regressiotestien sarja, joka ajetaan joka yö viimeisimmälle, ilman virheitä kääntyneelle ohjelmistopaketille. Testitapausten ja kirjastojen ohjelmakoodi onnistuttiin tekemään suurimmalta osin ”itseään kommentoivana”, eli ohjelmakoodin toiminnot ja avainsanat kuvaavat hyvin, mitä mikäkin ohjelmakoodin toiminto tekee.

Automatisoituja testitapauksia voidaan laajentaa tulevaisuudessa lastauskoneiden lisäksi kattamaan myös maanalaisen kuorma-auton ohjausjärjestelmän regressiotestauksen. Regressiotestauksen lisäksi yksinkertaisten ja usein toistuvien yksikkötestien automatisointi voisi olla hyödyllistä pitkällä aika välillä. Testitapauksien luomiseen olisi myös hyvä saada selkeät ohjelmakoodin kirjoitussäännöt, jotta useampien henkilöiden olisi helpompaa luoda ja ylläpitää yhteisiä testitapauksia sekä kirjastoja.

Testausautomaatio on nopeasti kasvava ja kehittyvä testausmuoto. Vaikka testausautomaatiosta on puhuttu jo vuosikymmeniä, se on silti melko tuore testauskeino osana yrityksien ohjelmistotestausta. Näin ollen tutustuminen testausautomaatioon ja Robot Frameworkin käyttöön on ollut hyödyllistä.

LÄHTEET

Ahonen, V. 2025. UI ja UX – mitä eroa niillä on? Verkkosivu 2025. Viitattu 22.3.2025 https://identio.fi/blogi/ui-ja-ux-mita-eroa-niilla-on/.

Guru99 2025. 13 BEST Automation Testing Tools. Verkkosivu 2025. Viitattu 5.1.2025 https://www.guru99.com/automated-testing-tools.html.

Guru99 2023a. V-Model in Software Testing. Verkkosivu 2023. Viitattu 23.4.2023 https://www.guru99.com/v-model-software-testing.html.

Guru99 2023b. What is software testing? Inroduction, Definition, Basics & Types. Verkkosivu 2023. Viitattu 23.4.2023 https://www.guru99.com/software-testing-introduction-importance.html.

Heikkinen, S. & Saarenpää, A. 2016. Koodi on kaikkialla – lyhyt johdatus ohjelmoinnin maailmaan. Yle 17.09.2016. Viitattu 22.4.2023 https://yle.fi/aihe/artikkeli/2016/09/17/koodi-kaikkialla-lyhyt-johdatus-ohjelmoinnin-maailmaan.

Honkanen, T. 2024. Introduction to software testing. Koulutusmateriaali 2024. Turku: Sandvik Mining and Construction Oy.

Kasurinen, J. 2013. Ohjelmistotestauksen käsikirja. Jyväskylä: Docendo. ISBN 978-952-5912-99-9 (painettu).

Kruchten, P. 2004. The rational unified process: an introduction. 3. Painos. Boston, Massachusetts: Pearson Education, Inc. ISBN 0-321-19770-4.

Microsoft 2023a. Code editing. Redifined. Verkkosivu 2023. Viitattu 7.11.2023 https://code.visualstudio.com/.

Microsoft 2023b. Learn to code with Visual Studio Code. Verkkosivu 2023. Viitattu 7.11.2023 https://code.visualstudio.com/learn.

Myers, G.; Badgett, T. & Sandler, C. 2012. The art of software testing. 2. Painos. Hoboken, New Jersey: John Wiley & Sons, Inc. ISBN 0-471-46912-2.

Open Education and Development Group 2022. Python – the language of today and tomorrow. Verkkosivu 2023. Viitattu 4.5.2023 https://pythoninstitute.org/about-python.

Openize Pty Ltd 2025. What is a DLL file? Verkkosivu 2025. Viitattu 22.3.2025 https://docs.fileformat.com/system/dll/.

Raitoharju, R. 2019. Tietojärjestelmien suunnittelu ja toteutus. Luentomateriaali 24.9.2019. Turku: Turun ammattikorkeakoulu.

Robot Framework Foundation 2016. Robot Framework User Guide. Verkkosivu 2016. Viitattu 4.6.2023 https://robotframework.org/robotframework/latest/RobotFrameworkUserGuide.html.

SmartBear Software 2021. What is automated testing? Verkkosivu 2021. Viitattu 2.3.2023 https://smartbear.com/learn/automated-testing/what-is-automated-testing/.

SoftwareTestingHelp 2023. Getting started with RIDE – Robot Framework IDE. Verkkosivu 2023. Viitattu 17.10.2023 https://www.softwaretestinghelp.com/getting-started-with-robot-framework-ride/.

Turun ammattikorkeakoulu 2019. Testaus, käyttöönotto, ylläpito. Luentomateriaali 2019. Turku: Turun ammattikorkeakoulu.

Tutorialspoint 2023. Big-Bang Testing. Verkkosivu 2023. Viitattu 30.4.2023 https://www.tutorialspoint.com/software_testing_dictionary/big_bang_testing.

Vocke, H. 2018. The Practical Test Pyramid. Verkkosivu 2018. Viitattu 16.3.2022 https://martinfowler.com/articles/practical-test-pyramid.html.