HAMK
Hämeen ammatti-
korkeakoulu

UI-testiautomaation hyödyntäminen osana regressiotestausta

Ammattikorkeakoulututkinnon opinnäytetyö

Tietojenkäsittelyn koulutus

Syksy 2023

Jenna Nevalainen

HAMK
Hämeen ammatti-
korkeakoulu

Tietojenkäsittelyn koulutus Tiivistelmä
Tekijä Jenna Nevalainen Vuosi 2023
Työn nimi UI-testiautomaation hyödyntäminen osana regressiotestausta
Ohjaaja Esa Huiskonen

Tämän toiminnallisen opinnäytetyön tavoitteena oli kartoittaa toimeksiantajan tarpeisiin sopiva testiautomaatiotyökalu sekä kehittää testiautomaatio-ohjelma, jota voidaan jatkossa hyödyntää toimeksiantajan järjestelmien testaamiseen. Toimeksiantajan järjestelmissä esiintyy usein virhetilanteita päivitysten yhteydessä, johon toivottiin ratkaisua testiautomaatiosta. Opinnäytetyön toimeksiantajana toimi Sosiaali- ja terveysalan lupa- ja valvontavirasto Valvira.

Opinnäytetyön teoreettisessa osuudessa määritellään työn kannalta keskeiset käsitteet. Teoriapohja koostuu ohjelmistotestauksen, testiautomaation sekä ketterien menetelmien perusteista. Tietoa on kerätty useista eri lähteistä, kuten kirjoista, verkosta ja tutkimuksista. Teoriaosuudessa kerrotaan myös sopivan testiautomaatiotyökalun valinnasta sekä vertaillaan markkinoilla olevien testiautomaatiotyökalujen ominaisuuksia. Käytännönosuudessa käydään läpi testiautomaatio-ohjelman toteutus vaihe vaiheelta.

Opinnäytetyön lopputuloksena syntyi toimeksiantajan tarpeisiin vastaava testiautomaatio-ohjelma, jota voidaan jatkossa hyödyntää järjestelmien testaamiseen päivitysten yhteydessä. Toimeksiantajalle tehtiin kattava dokumentaatio ohjelman käyttöön, jonka avulla testejä voidaan myös tehdä lisää aina tarvittaessa. Opinnäytetyön myötä havaittiin, että regressiotestauksen automatisoinnilla voidaan vähentää testaukseen kuluvia resursseja sekä parantaa järjestelmien laatua pidemmällä aikavälillä.

Avainsanat ohjelmistotestaus, testiautomaatio, regressiotestaus

Sivut 34 sivua ja liitteitä 1 sivu

HAMK
Hämeen ammatti-
korkeakoulu

Degree Programme in Business Information Technology
Author Jenna Nevalainen
Subject Utilizing UI-test automation as part of regression testing
Supervisors Esa Huiskonen

Abstract
Year 2023

The purpose of this functional thesis was to identify a suitable test automation tool for the client's needs and develop a test automation program that can be utilized for regression testing of the client's systems in the future. The client often experiences error situations during updates, for which a solution was sought through test automation. The client for this thesis was the National Supervisory Authority for Welfare and Health (Valvira).

The theoretical part of the thesis defines key concepts relevant to the work. The theoretical foundation of the thesis consists of the fundamentals of software testing, test automation, and agile methods. Information was gathered from various sources such as books, the internet, and research papers. The theoretical section also discusses the selection of a suitable test automation tool and compares the features of test automation tools available in the market.

As a result of the thesis, a test automation program that meets the client's needs was developed. This program can be utilized in the future for testing systems during updates. Comprehensive documentation was provided to the client for using the program, enabling the creation of additional tests as needed. The thesis clarified what should be considered when transitioning to test automation, as well as the benefits and challenges that may arise from it. The thesis findings indicated that the automation of regression testing can reduce the resources dedicated to testing and improve the quality of systems in the long term.

Keywords software testing, test automation, regression testing
Pages 34 pages and appendices 1 page

Sanasto

VSCode	Ohjelmointiin tarkoitettu tekstieditori Visual Studio Code
Playwright	Microsoftin kehittämä testiautomaatio-ohjelmisto
Node.js	JavaScript-koodin suorittamiseen tarkoitettu ympäristö
UI (User Interface)	Järjestelmän käyttöliittymä
Testiautomaatio	Automatisoitua ohjelmiston testausta
Manuaalinen testaus	Käsin tehtävää testausta
Regressiotestaus	Järjestelmän uudelleen testaus muutosten yhteydessä
Open-source	Avoin lähdekoodi, jota kuka tahansa voi käyttää ja muokata

Sisällys

Kuvat, komennot, ohjelmakoodit, taulukot ja kaavat

Liitteet

1 Johdanto

Opinnäytetyön tavoitteena on löytää toimeksiantajan vaatimuksiin vastaava testiautomaatiotyökalu sekä kehittää testiautomaatio-ohjelma, jota voidaan jatkossa hyödyntää järjestelmien regressiotestaukseen. Toimeksiantajan järjestelmien laatu on heikentynyt niiden elinkaaren pidentyessä ja virheitä esiintyy erityisesti päivitysten yhteydessä. Työn tavoitteena on selvittää, voisiko testiautomaatiolla vähentää virhetilanteita sekä parantaa järjestelmien laatua pidemmällä aikavälillä. Testien automatisointi toteutetaan käyttöliittymätasolla, jota kutsutaan UI-testiautomaatioksi.

Opinnäytetyön toimeksiantajana toimii Sosiaali- ja terveysalan lupa- ja valvontavirasto Valvira. Valviran tehtävänä on valvoa sosiaali- ja terveydenhuollon, varhaiskasvatuksen, alkoholielinkeinon sekä ympäristöterveyshuollon asianmukaisuutta. Suoritin opintoihini kuuluvaa työharjoittelua Valviran tietohallinnolla, jonka aikana tuli ilmi tarve järjestelmien regressiotestien automatisoinnille.

Opinnäytetyön teoria pohjautuu pääasiassa ohjelmistotestauksen ja testiautomaation perusteisiin. Opinnäytetyössä käydään aluksi läpi ohjelmistotestauksen teoriaa pääpiirteittäin, kuten testauksen hyötyjä, testaustasoja ja -menetelmiä sekä testauksen merkitystä ohjelmistotuotannossa. Seuraavassa luvussa käsitellään testiautomaatiota yleisesti sekä vertaillaan markkinoilla olevia testiautomaatiotyökaluja. Tämän jälkeen kerrotaan lyhyesti ketteristä menetelmistä. Käytännön osuudessa käydään lopuksi läpi testiautomaatio-ohjelman toteuttaminen vaihe vaiheelta.

Opinnäytetyössä pyritään vastaamaan seuraaviin kysymyksiin:

- Mitä asioita tulee ottaa huomioon testiautomaatiotyökalun valinnassa?
- Mitä haasteita testiautomaation toteuttamisessa ja ylläpidossa voi ilmetä?
- Mitä hyötyjä toimeksiantajalle on testiautomaatiosta?

2 Ohjelmistotestaus

Tässä luvussa perehdytään ohjelmistotestaukseen yleisellä tasolla. Luvussa käydään läpi ohjelmistotestauksen hyötyjä, testauksen merkitystä ohjelmistotuotannossa sekä erilaisia testaustasoja ja -menetelmiä.

2.1 Ohjelmistotestaus yleisesti

Ohjelmistotestaus voidaan kuvata prosessina tai sarjana prosesseja, joiden tavoitteena on varmistaa ohjelmistokoodin toimivan suunnitellulla tavalla. Ohjelmiston toiminta tulisi olla mahdollisimman johdonmukaista ja helposti ennustettavaa, eikä sen tulisi tuottaa suuria yllätyksiä käyttäjilleen. (Myers ym., 2012, s. 2)

Yleisesti testauksen voi suorittaa lähes kuka tahansa, kuten ohjelmistokehittäjä tai liiketoiminnan ammattilainen. Ohjelmiston ollessa yksinkertainen, testaaminen voi olla helppoa. Ohjelmiston ollessa laajempi ja sisältäessä esimerkiksi erilaisia integraatioita, on testauksen asiantuntijan suositeltavaa suorittaa testaus. Tällöin voidaan varmistaa, että ohjelmiston jokainen osa-alue toimii alusta loppuun. (Rehn, 2023)

Yleinen väärinkäsitys testauksesta on, että sitä kannattaa tehdään vain silloin, jos projektin loppuvaiheessa jää ylimääräistä aikaa. Testaus on kuitenkin olennainen osa koko ohjelmistokehityksen elinkaarta ja se tulisi ottaa huomioon heti suunnitteluvaiheesta jatkuen ylläpitoon saakka. Testauksen tavoitteet ovat kehittyneet vuosien saatossa pelkästä ohjelmiston toimivuuden tarkastamisesta laajempaan rooliin. Testauksen merkitys ei ole enää pelkästään vikojen löytäminen, vaan myös tiedon tarjoaminen ohjelmistoa koskevia päätöksiä varten. Testauksella voidaan saada arvokasta tietoa esimerkiksi mahdollisista riskeistä liiketoiminnalle, sopivasta markkinoille tuloajasta sekä kustannuksista. Ohjelmistotestauksella voisi sanoa olevan kaksi päätavoitetta: vikojen havaitseminen ja korjaaminen mahdollisimman aikaisessa vaiheessa sekä tärkeisiin päätöksiin tarvittavan tiedon tarjoaminen. (Homès, 2012, s. 9)

Testausprosessi sisältää monia eri vaiheita kuten suunnittelu, vaatimusmäärittely, testitapausten luominen ja suoritus sekä ohjelmiston validointi. Suunnittelu aloitetaan jo kehityksen alkuvaiheessa tunnistamalla ohjelmiston vaatimukset sekä testaussuunnitelman laatimisella. Testitapaukset luodaan yleensä perustuen vaatimuksiin ja optimoidaan perustuen sovellusympäristöön, kehitysteknologiaan ja koodin kattavuuteen. Esimerkiksi eri ohjelmointikielillä esiintyy usein tyypillisiä vikoja, jotka olisi hyvä ottaa huomioon testitapauksia suunnitellessa. Testien luominen olisi hyvä aloittaa heti, kun tekniset

vaatimukset ja tiedot ovat selvillä. Testitapausten varhainen luominen helpottaa ajoitusongelmia, kun testejä voidaan suorittaa heti, kun testitapausta vastaava koodi on saatu valmiiksi. (Baresi & Pézze, 2006.)

Kun virheitä ei enää löydy testatessa ja ohjelman on todettu suorittavan siltä odotetut toiminnot, ohjelma voidaan ottaa käyttöön ja siirtyä ylläpitovaiheeseen. Ylläpitovaiheessa ohjelmaan tehdään tarvittavia muutoksia aina silloin, jos käyttöä haittaavia ongelmia ilmenee. Ohjelmisto ei kuitenkaan ole koskaan täysin virheetön, vaan hyvinkin testattuun ohjelmistoon jää aina virheitä. Tällöin virheet vain ovat niin pieniä, ettei ne vaikuta ohjelman käyttöön. (Kasurinen, 2013 s. 13)

2.2 Ohjelmistotestauksen merkitys ohjelmistotuotannossa

Nykyaikana erilaiset ohjelmistot vaikuttavat suuresti jokapäiväiseen elämäämme ja niiden vaikutus myös maailmantalouteen kasvaa jatkuvasti. Mili kertoo kirjassaan ilmiön alkaneen hitaasti 1900-luvun puolivälissä, jonka jälkeen sitä kiihdytti entisestään World Wide Webin syntyminen 2000-luvun alussa. Ilmiön myötä ohjelmistotuotteiden kysyntä on kasvanut suuresti ja synnyttänyt markkinoille kilpailua, johon ohjelmistoyritykset pyrkivät vastaamaan. (Mili, 2015, s. 19)

Ohjelmistot ovat nykyään monimutkaisempia kuin aikaisemmin, niiden käyttäessä lukuisia eri ohjelmointikieliä, käyttöjärjestelmiä ja alustoja. 1970-luvulla tietokoneiden käyttö oli vielä hyvin harvinaista, kun taas nykypäivänä monen työnteosta tulee mahdotonta ilman tietokonetta. Yhdellä ohjelmistolla voi olla suuri vaikutus miljoonien ihmisten elämään. Jos ohjelmisto ei toimi odotetusti, se voi aiheuttaa menetettyjä työtunteja, rahaa sekä turhautumista. Ohjelmiston sujuvalla toiminnalla taas mahdollistetaan tehokas työskentely. Ohjelmistojen monimutkaistumisen myötä myös niiden testaamisesta on tullut monimutkaisempaa. Toisaalta tänä päivänä on apuna kehittyneemmät ohjelmistot, joita voidaan hyödyntää testaukseen. (Myers et al., 2012, s. 1)

Mili mainitsee kirjassaan monien eri tieteen ja tekniikan alojen olevan työssään täysin riippuvaisia erilaisista ohjelmistoista. Ohjelmistoja käytetään esimerkiksi bioinformatiikkaan, sääennusteisiin sekä mallintamiseen ja simulointiin. Tietojenkäsittelykoulutusten on havaittu siirtyvän opetussuunnitelmissaan enemmän käytännön ohjelmistotekniikkaan perinteisen teoriasisällön sijaan. Ohjelmiston ottaessa vastuulleen entistä toimintakriittisiä sovelluksia tulee entistä tärkeämmäksi varmistaa ohjelmiston luotettavuus. Tämän toteutumiseen tarvitaan erilaisia käytäntöjä, kuten prosessinhallintaa, jolla varmistetaan ohjelmistokehityksen tapahtuvan hyväksi todettujen prosessien mukaisesti. Tarvitaan myös

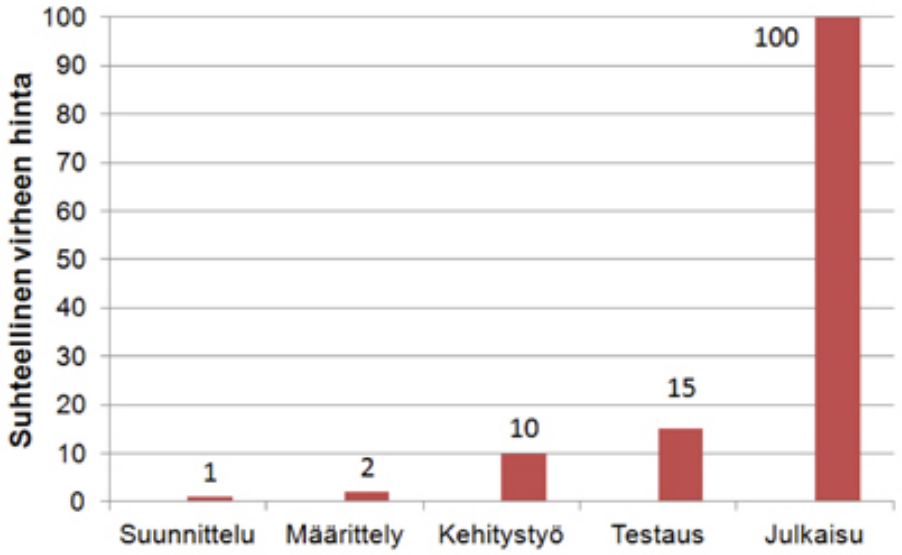
tarkkaa tuotevalvontaa, jonka avulla varmistetaan ohjelmistojen vastaavan laatuvaatimuksiin. (Mili, 2015, ss. 19–20)

Historian aikana on nähty useita vaaratilanteita, jotka ovat aiheutuneet puutteellisesta testauksesta. Vuonna 2009 lentokone Airbus A380:n autopilottitoiminto lakkasi toimimasta lennolla Pariisista New Yorkiin ja lentokone joutui palaamaan takaisin. Ohjelmistoviat ovat aiheuttaneet myös vakavia onnettomuuksia esimerkiksi terveydenhuollossa, kuten tapauksessa, jossa sädehoitojärjestelmän vika johti kolmen potilaan kuolemaan johtuen liian suuresta säteilyannoksesta. Tapahtuneen syyksi todettiin hyväksyntätestien puuttuminen, henkilöstön puutteellinen koodintarkastus ja tiedonpuute järjestelmän luotettavuudesta. Yhdysvalloissa ohjelmistoviat ovat aiheuttaneet ongelmia myös vaaleissa, kun äänestyskoneet eivät toimineet äänienlaskun aikana ja suuri määrä ääniä hylättiin. Myös Ranskassa on koettu ohjelmistovika vaalien aikana. Tällöin vaaliohjelmisto oli pyöristänyt ehdokkaan kokonaisäänien prosenttimäärän ylöspäin, jonka takia ehdokas pääsi virheellisesti jatkoon. (Homes, 2012, s. 2)

Ohjelmistotestaus on tärkeä osa ohjelmistotuotantoa ja jopa laajempi kokonaisuus, kuin esimerkiksi ohjelmointi. Testaajan vastuulle voi kuulua monta erilaista asiaa erilaisissa työvaiheissa, kuten haastattelut, dokumentointi ja koodin kirjoittaminen. Testaajan työnkuva vaihtelee myös paljon riippuen yrityksestä, esimerkiksi peliyrityksellä ja valtion virastolla voi olla täysin erilaiset testauskriteerit. Peliyritystä saattaa kiinnostaa erityisesti pelin hauskuus, kun taas valtion virastolle on todennäköisesti tärkeämpää palvelun helppokäyttöisyys. (Kasurinen, 2013 s. 10)

Testaustyö on tärkein ohjelmistoprojektin työvaihe, jos vertaillaan niiden kannattavuutta. Ohjelmistotestauksen ja tuotteen kannattavuudella on havaittu olevan selkeä yhteys. Yritykset, jotka panostavat testaukseen, saavat paremman kustannushyödyn tuotteilleen verrattuna yrityksiin, jotka eivät panosta. Huonosti toimiva ohjelmisto vähentää myös käyttäjämäärää ja vaikuttaa negatiivisesti yrityksen maineeseen. Kasurisen mukaan suunnitteluvaiheessa löydetty virhe kustantaa kymmenesosan kehitystyön aikana löydetystä virheestä ja sadasosan julkaisun jälkeen löydetystä (Kuva 1). Kasurinen mainitsee kirjassaan myös tutkimuksen, jonka mukaan yhdysvaltalaiset ohjelmistotalot menettivät jopa 21.2 miljardia dollaria vuonna 2002 puutteellisen testauksen vuoksi. (Kasurinen, 2013 ss. 11–12)

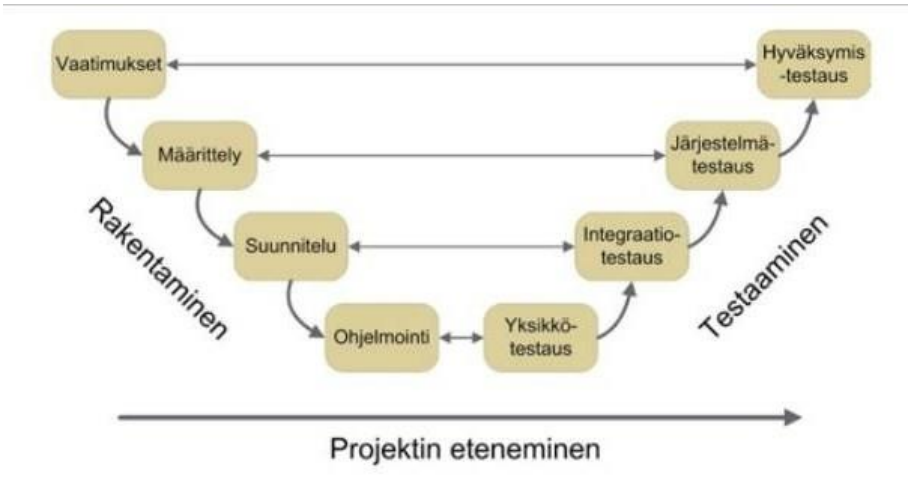
Kuva 1 Virheen kustannukset eri kehitysvaiheissa (Kasurinen, 2013, s. 18)



2.3 V-malli ja testaustasot

Testausta pidetään usein vain yksittäisenä vaiheena, joka suoritetaan projektin lopussa. Tämän sijaan se olisi kuitenkin parempi nähdä jatkuvana toimintana edeten ohjelmistokehityksen rinnalla kehityksen alusta loppuun saakka. Testauksen V-malli on prosessimalli, joka kuvaa testauksen jatkuvan etenemisen kehitystyön mukana. Se havainnoi myös sitä, kuinka testausta on mahdollista suunnitella vaihe kerrallaan kehityksen edetessä. Yleisesti V-mallin vasen haara kuvaa ohjelmakehitystä ja oikea ohjelmistotestausta (Kuva 2). (Mili, 2015, s. 48)

Kuva 2 V-malli (Kasurinen, 2013 s. 51)



Seuraavissa alaluvuissa perehdytään tarkemmin V-mallissa esitettyihin testausprosessin eri vaiheisiin eli testaustasoihin.

2.3.1 Yksikkötestaus

Yksikkötestaus on käytetyimpiä testausmenetelmiä ja yleisesti käytössä ohjelmistoyrityksissä. Yksikkötestauksessa tarkastellaan toteuttamisen yhteydessä yksittäisen moduulin, funktion tai olion toimintaa, useimmiten kehittäjän itsensä toimesta. Yksikkötestauksella pyritään varmistamaan, että uusi toiminto tai muutos järjestelmään toimii toivotulla tavalla. Testaus voidaan suorittaa esimerkiksi luomalla funktio- tai oliokutsuja, joihin testattavan komponentin kuuluu vastata tietyllä tavalla. Jos yksikkötesti epäonnistuu, kehittäjä näkee heti, että komponentti ei toimi toivotulla tavalla ja pystyy korjaamaan vian, ennen kuin se ehtii osaksi laajempaa ohjelmaa. (Kasurinen, 2013, ss. 51–52)

Kasurisen mukaan yksikkötestauksen ongelmana on usein se, että komponentti saattaa tarvita vuorovaikutusta toisen ominaisuuden kanssa toimiakseen. Esimerkiksi tapauksessa, jossa käyttöliittymän komponentin tulisi reagoida järjestelmän toimintaan, vaikka hakemalla tietoa tietokannasta. Tällöin kehittäjän täytyy luoda testikomponentteja, joilla voidaan matkia varsinaisen järjestelmän ja testikomponentin välistä toimivuutta. (Kasurinen, 2013, s. 52)

2.3.2 Integraatiotestaus

Integraatiotestaukseen siirrytään, kun yksikkötestauksessa ei enää löydetä virheitä. Integraatiotestauksessa järjestelmän eri moduuleja aletaan sovittamaan yhteen tavoitteena saada järjestelmä toimimaan yhtenä kokonaisuutena. Uusi komponentti yhdistetään osaksi aiemmin testattua toimivaa kokonaisuutta. Jos uusi komponentti sisältää yhteyksiä, joita ei ole vielä valmiina, joudutaan testausta varten rakentamaan tynkiä sijaiskomponenteiksi, jotta integraatiotestausta voidaan suorittaa. Integraatiotestauksen tärkein tavoite on selvittää, toimiiko järjestelmän eri osat yhdessä. Integraatiotestauksessa testitapaukset ovat laajempia, kuin yksikkötestauksessa, mutta ei kuitenkaan vielä koko järjestelmän käyttöä koskevia. (Kasurinen, 2013, s. 54)

Käytännössä integraatiotestauksessa luodaan laajempia testitapauksia, kuin yksikkötestauksessa. Integraatiotestit eivät kuitenkaan testaa vielä koko järjestelmän toimivuutta. Integraatiotestitapauksia voisi olla esimerkiksi moduulien välinen viestinnän tai samaa tietokantaa hyödyntävien komponenttien testaaminen. Käytännössä integraatiotestit toteutetaan lisäämällä toimivaan kokonaisuuteen yksi komponentti lisää, jonka jälkeen tarkastetaan kokonaisuuden toiminta. (Kasurinen, 2013, s. 54)

2.3.3 Järjestelmätestaus

Järjestelmätestauksessa testataan nimensä mukaisesti koko järjestelmää. Järjestelmä voi koostua monista eri osista, kuten esimerkiksi mobiilisovelluksesta sekä erillisellä palvelimella olevasta ohjelmistosta. Järjestelmätestauksessa testaaja kokeilee jotakin mobiilisovelluksen toimintoa ja tarkistaa, vastaako ohjelmisto toimintoon toivotulla tavalla. Järjestelmätestauksen perusteella ohjelmistoon voidaan tehdä vielä suuria muutoksia ja parannuksia. (Juvonen, 2018, s. 27)

Järjestelmätestauksen tavoitteena on varmistaa, että järjestelmä vastaa tavoitteita ja toimii kokonaisuutena. Käytännössä testaustyön menetelmät vaihtelevat riippuen projektista. Järjestelmätestaus voi olla käytännössä esimerkiksi käyttäjätestausta, kuormitustestausta, tutkivaa testausta tai jotain muuta työmenetelmää, jolla testataan toiminnallista kokonaisuutta. Järjestelmätestaus suoritetaan testiympäristössä, kun testattava järjestelmä siirtyy lopulliseen ympäristöön, siirrytään hyväksymistestaukseen. (Kasurinen, 2014, s. 56)

2.3.4 Hyväksymistestaus

Hyväksymistestaus on testauksen viimeistelyvaihe. Hyväksymistestauksen tavoitteena on varmistaa ennen julkaisua, että ohjelmisto vastaa liiketoiminnan sekä asiakkaiden vaatimuksia. Hyväksymistestaus aloitetaan, kun ohjelman koodi on täysin valmis eikä aikaisemmissa testivaiheissa enää löydy virheitä. Hyväksymistestauksessa myös asiakkaat otetaan usein mukaan testausprosessiin palautteen saamiseksi. Palautteen avulla tunnistetaan mahdolliset puutteet, jotka ovat jääneet huomaamatta kehitysvaiheessa. Hyväksymistestaus voi auttaa kehittäjiä myös ymmärtämään toimintojen tarvetta liiketoiminnan kannalta. (Gillis, 2021)

Käytännössä hyväksymistestaus toteutetaan usein sen kohdeympäristössä, toisin kuin esimerkiksi järjestelmätestauksessa, jossa testaus tapahtuu sitä varten rakennetussa ympäristössä. Hyväksymistestauksessa on tavoitteena saada asiakkaan hyväksyntä tuotteelle, jonka jälkeen ohjelmisto siirtyy laillisesti asiakkaan omaisuudeksi. Tällöin poistuu ohjelmiston toimittajalta myös ohjelmiston kehitysajan huolto- ja korjausvelvoite. (Kasurinen, 2014 s. 57)

2.4 Testausmenetelmät

Järjestelmää voidaan testata useilla eri menetelmillä, joista osa käydään läpi seuraavissa alaluvuissa.

2.4.1 Regressiotestaus

Kasurisen mukaan regressiotestaus ei ole itsessään oma testausmenetelmänsä, vaan pikemminkin yleistermi, jolla viitataan uudelleentestaamiseen. Regressiotestauksesta voidaan puhua aina silloin, kun jotakin toimivan järjestelmän osaa muutetaan ja sen toimivuus täytyy varmistaa testaamalla. Usein järjestelmissä ilmenevät virheet kohdistuvat uusiin komponentteihin tai niitä hyödyntäviin toimintoihin. Regressiotestauksessa suhtaudutaan pieniinkin muutoksiin niin kuin koko järjestelmää olisi uudistettu. Regressiotestien avulla voidaan myös tarkastaa versionhallinnassa tehtyjen kehityshaarojen yhdistämisen jälkeen, onko aiemmista osista poistetut virheet hävinneet myös yhdistetystä versiosta. Regressiotestit ja sen testitapaukset ovat usein otollisia testiautomaation hyödyntämiselle niiden suuren määrän ja toistuvuuden vuoksi. (Kasurinen, 2014, ss.70–71)

2.4.2 Savutestaus

Savutestausta kutsutaan usein myös varmistustestaukseksi tai luottamustestaukseksi ja sen tavoitteena on määrittää, onko ohjelmistoversio valmis seuraavaan testausvaiheeseen. Savutestauksessa keskitytään ohjelmiston tärkeimpiin toiminnallisuuksiin, eikä siinä puututa pienempiin yksityiskohtiin. Savutestaus on ikään kuin alustava testausmenetelmä, jolla pyritään löytämään ohjelmiston kriittiset virheet, ennen siirtymistä perusteellisempaan testaukseen. Testaustiimi suorittaa savutestausta tekemällä minimaalisia testisarjoja jokaiseen ohjelmiston versioon, jotka keskittyvät ohjelmiston tärkeimpiin toiminnallisuuksiin. Jos virheitä ei löydetä savutestauksen aikana, testaustiimi jatkaa seuraavaan testausvaiheeseen. Virheen ilmetessä ohjelmisto palautetaan kehitystiimille korjattavaksi. Ilman savutestausta kriittiset ongelmat saattavat jäädä huomaamatta ja aiheuttaa suurempia ongelmia myöhemmin. (Bergs, 2023)

2.4.3 Kuormitustestaus

Kuormitustestaus on testausmenetelmä, jossa ohjelmistoa käytetään suunnitellulla käyttäjämäärällä esimerkiksi luomalla virtuaalikäyttäjiä, joille luodaan tehtäväksi testejä, jotka simuloivat normaalia käyttäjän toimintaa. Kuormitustestauksella on kolme tärkeää tehtävää;

tunnistaa järjestelmän mahdolliset ongelmakohdat, selvittää järjestelmän maksimikapasiteetti sekä miten järjestelmä selviytyy normaaleista käyttöolosuhteista. Esimerkiksi verkkokaupalle, jonka tulisi pystyä palvelemaan 250 käyttäjää kerrallaan, voidaan kuormitustestaus suorittaa luomalla 250 virtuaalikäyttäjää, jotka ohjelmoidaan tekemään järjestelmässä eri toimintoja samanaikaisesti. Jos järjestelmä ei suoriudu normaalista käytöstä, voidaan sanoa kuormitustestauksessa löytyneen virheitä, jotka tulee korjata. (Kasurinen, 2014 s. 71)

2.4.4 Käytettävyystestaus

Kasurisen mukaan käytettävyystestauksessa keskitytään pääasiassa järjestelmän käyttöliittymän toimivuuteen. Käytettävyystestaus voidaan aloittaa jo järjestelmän suunnitteluvaiheessa, jolloin testejä voidaan suorittaa prototyypeille, joissa ei ole muita toiminnallisuuksia käyttöliittymän lisäksi. Suurimmaksi osaksi käytettävyystestaus on kuitenkin valmiin järjestelmän käyttöliittymän testaamista, jolla varmistetaan, että käyttöliittymä on suunniteltu ja toteutettu oikein. Käytettävyyden testaamiseen voidaan käyttää useita eri menetelmiä. Menetelminä voidaan käyttää esimerkiksi käyttökokeiluja, joiden jälkeen suoritetaan haastattelututkimukset. Käytettävyystestaukseen on myös omia työkaluja, kuten monitori, jolla pystytään selvittämään, kuinka kauan käyttäjä katsoo mihinkin kohtaan järjestelmässä. Testaukseen voidaan käyttää myös toiminnannauhoitussovelluksia, joiden avulla nähdään kaikki käyttäjän tekemät toiminnot sekä syöttämät tiedot. Käyttäjätestauksessa voidaan jopa nauhoittaa itse järjestelmän käyttäjää, jolloin seurataan kasvojen eleitä, jotka voivat paljastaa paljon käyttöliittymän toimivuudesta. Käytettävyystestauksen merkitys vaihtelee eri alojen yrityksissä. Esimerkiksi peliteollisuudessa käytettävyystestauksella on suuri merkitys. Kasurinen mainitsee kirjassaan esimerkin, jossa tuotteen pääominaisuudet olivat muuttuneet merkittävästi käytettävyystestauksen myötä, kun pelin käyttäjät tykästyivät järjestelmän virheestä aiheutuneesta poikkeuksellisesta liikkumistavasta. (Kasurinen, 2014 s. 70–71)

3 Testiautomaatio

Tässä luvussa perehdytään testiautomaatioon yleisellä tasolla. Luvussa käydään läpi myös testiautomaation hyötyjä sekä yleisiä haasteita, joita testiautomaatioon siirtyessä saattaa ilmetä. Lopuksi vertaillaan markkinoilla olevia suosittuja testiautomaatiotyökaluja.

3.1 Testiautomaatio yleisesti

Testiautomaatiolla tarkoitetaan prosessia, jolla automatisoidaan manuaalinen testausprosessi jonkin teknologian avulla. Testiautomaatiolla pyritään vähentämään ihmisten tekemiä virheitä sekä säästää työntekijöiden aikaa tärkeämpiin ja mielekkäämpiin tehtäviin. Automatisoiduilla testeillä vähennetään myös testien suoritukseen kuluvaa aikaa. Kun testitapaus on kirjoitettu kerran, se voidaan suorittaa automaattisesti, kuinka monta kertaa tahansa. (Rana, 2019)

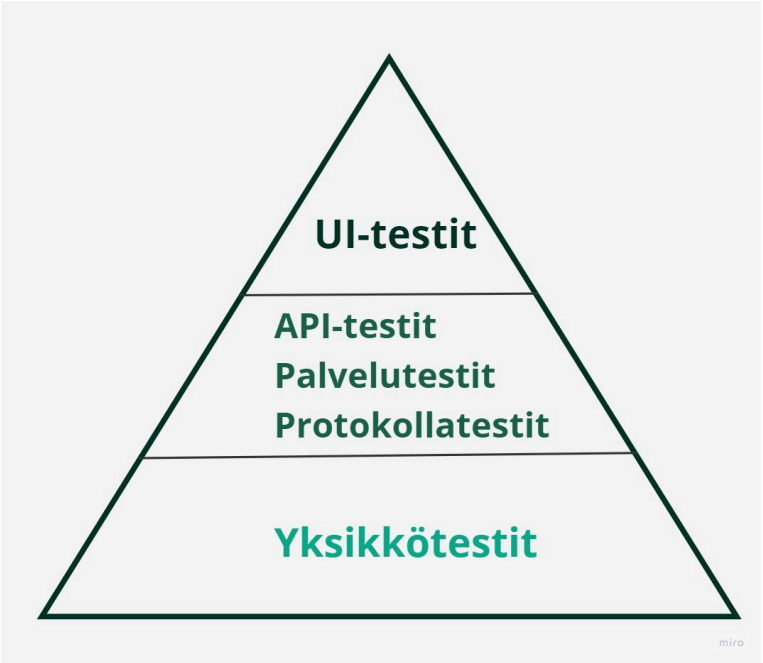
Testiautomaatio vähentää merkittävästi aikaa ja resursseja, joita tarvitaan järjestelmän perusteelliseen testaamiseen. Testiautomaatiolla testeistä saadaan kattavammat ja johdonmukaisemmat, jonka myötä asiakkaiden luottamus järjestelmään kasvaa. Vaikka automatisoidut testit vähentävät manuaalisen testauksen tarvetta pitkällä aikavälillä, ei sen tarve häviä kokonaan. Tällöin manuaalista testausta voidaan kohdentaa enemmän tutkivaan testaukseen, yksittäisiin testeihin ja käytettävyystesteihin. (Boby, 2021 s. 10)

Testiautomaatio toteutetaan luomalla erillinen ohjelma testien suorittamiseen. Testiautomaation kohteeksi sopivat testitapaukset, jotka toistuvat usein. Jos esimerkiksi järjestelmälle tehdään joka yö uusi päivitys, ei testaajien ole järkevää suorittaa manuaalisesti joka kerta samoja perustestejä. Testiautomaation tavoitteena on pääasiassa säästää testaajien aikaa muihin tehtäviin. Testiautomaatio-ohjelma voidaan asettaa esimerkiksi yön ajaksi suorittamaan testejä ja testaajien tehtäväksi jää vain seuraavana aamuna tarkistaa testien tulokset. (Kasurinen, 2013, s. 76)

Automatisoidut testit yhdistetään usein yhdeksi kokoelmaksi, jonka kaikki testit voidaan suorittaa kerralla. Suorituksen jälkeen testiautomaatio-ohjelma luo testiraportin, josta voidaan nähdä jokaisen testin tulokset, kuten menivätkö testit onnistuneesti läpi vai eivät. Testiraportista voidaan nähdä myös yksityiskohtaisempaa tietoa testien suoriutumisesta, kuten kuinka paljon mihinkin vaiheeseen on kulunut aikaa. Jos jokin testi epäonnistuu, raportista voidaan nähdä missä vaiheessa testi on epäonnistunut ja mahdollisesti myös epäonnistumisen syy. Raportista nähdään myös, jos testi on jäänyt kokonaan suorittamatta, joka johtuu yleensä virheellisestä testiskriptistä. (Bierig ym., 2022, ss. 230–231)

Automatisoidut testit voidaan toteuttaa järjestelmän arkkitehtuurin eri tasoilla. Testien automatisointi voi kohdistua järjestelmän käyttöliittymään tai järjestelmän mukaan sen komentorivikäyttöliittymään. Syvemmällä järjestelmässä automatisoidut testit voivat kohdistua rajapinta- (API), palvelu- tai protokollatestaukseen. Käyttöliittymään perustuvat testit ovat usein haastavimpia ylläpitää, koska käyttöliittymän rakenteeseen kohdistuu usein muutoksia. Mike Cohn esitteli kirjassaan suureen suosioon nousseen testiautomaatiopyramidin, joka kuvaa miten eri tasoilla tapahtuvat automatisoidut testit tulisi kohdentaa, jotta testiautomaatiosta saadaan ajan myötä mahdollisimman kustannustehokasta (Kuva 3). (Baumgartner ym., 2022, luku 1.2)

Kuva 3 Testiautomaatiopyramidi (Mukaillen Baumgartner ym., 2022, luku 1.2)

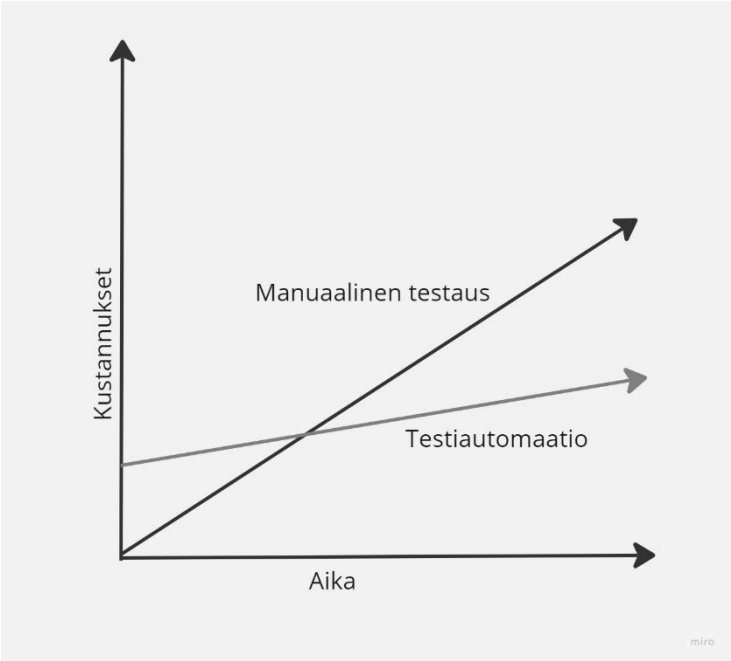


3.2 Hyödyt ja haasteet

Testauksen automatisoinnilla voidaan saavuttaa useita hyötyjä. Automatisoinnin avulla saadaan suoritettua suurempi määrä testitapauksia ohjelmiston julkaisua kohti. Manuaalisesti tehtävällä regressiotestauksella kustannustehokkuuden ja toteuttamismahdollisuuksien rajat tulevat usein nopeasti vastaan. Manuaalinen regressiotestaus sitoo työntekijöiden resursseja eikä kaikkia mahdollisia testitapauksia pystytä toteuttamaan. Automatisoinnin avulla saadaan luotua monimutkaisiakin testitapauksia, joita voidaan suorittaa niin usein kuin on tarpeen. Automatisoituja testejä voidaan ajaa myös useita samanaikaisesti ja monilla eri selaimilla, joka vähentää huomattavasti testien suorittamiseen kuluvaa aikaa. (Baumgartner ym., 2022, luku 1.3)

Testiautomaation avulla saavutetaan pitkällä aikavälillä kustannussäästöjä, jos se toteutetaan oikeaoppisesti (Kuva 4). Ohjelmistokehitys itsessään myös helpottuu, kun kehittäjät uskaltavat tehdä ohjelmistoon muutoksia rohkeammin, eikä tarvitse huolehtia tärkeiden osa-alueiden hajoamisesta. Testiautomaation myötä kehittäjät saavat myös nopeammin palautetta työstään ja sen laadusta, joka auttaa kehittämistyössä. Toistuvan regressiotestauksen automatisoinnilla testaajien osaaminen saadaan paremmin hyödynnettyä, joka lisää myös testaajien tyytyväisyyttä ja motivaatiota työtään kohtaan. (Testiautomaatio | Ite wikin digitalisoinnin opas, 2019)

Kuva 4 Testiautomaation kustannukset verrattuna manuaaliseen testaukseen (Mukaillen Boby, 2021, s. 6)



Testien automatisointiin siirtyminen on päätös, joka vaatii huolellista harkintaa ja tiedon keräämistä. Monesti ongelmaksi koituu se, että organisaatiot lähestyvät testiautomaatiota pelkästään kustannussäästöjä tavoitellen ja epäonnistuvat lopulta tavoitteissaan. Testiautomaatio ei ole oikotie kustannusten vähentämiseen, ajan säästämiseen tai laadun parantamiseen. Ennen testiautomaatioon siirtymistä olisi tärkeää ottaa huomioon useat eri tekijät, kuten testauksen kohde, teknologia ja elinkaari. Oikein toteutettuna testiautomaatio on kuitenkin luotettavimpia tapoja suorittaa ohjelmistotestaukset onnistuneesti. (Boby, 2021 s. 6)

Testiautomaatiota toteuttaessa kohdataan usein haasteita, jotka olisi hyvä ottaa huomioon jo etukäteen, jotta projektin edetessä vältytään ikäviltä yllätyksiltä. Testiautomaatioon siirryttäessä seuraa lähes aina aluksi lisäkustannuksia, kun investoidaan tarvittaviin uusiin

teknologioihin ja työkaluihin. Kustannuksia voi syntyä myös tarvittavien laitteiden hankinnasta testaajille sekä uusien prosessien ja työvaiheiden kehittämisestä. Uusien laitteistojen ja prosessien lisäksi tulisi myös panostaa testitiimin osaamisen kehittämiseen ja ylläpitoon. Haasteita seuraa usein myös siitä, että testiautomaatioratkaisujen ja erityisesti testiskriptien ylläpitoon kuluva aika ja vaiva aliarvioidaan yrityksissä. Testiautomaatio-ohjelmistot vaativat jatkuvaa ylläpitoa, jota voi hankaloittaa esimerkiksi sopimaton arkkitehtuuri, riittämätön dokumentaatio ja sovittujen käytäntöjen laiminlyönti. (Baumgartner ym., 2022)

Käyttöliittymätestauksessa kohdataan usein haasteita käyttöliittymän vaihtuvuuden vuoksi. Käyttöliittymään tehtyjen muutosten myötä tulee aina varmistaa, että nykyiset testit mukautuvat muutoksiin. Kun testejä tehdään useilla eri selaimilla ja eri asetuksilla tulee testien toimivuus tarkistaa kaikilla niillä. Käyttöliittymätestausta on myös perinteisesti pidetty haastavana verkkosovellusten monimutkaisuuden vuoksi. Selainten välillä voi olla merkittäviä yhteensopivuusongelmia, joka vaatii erittäin huolellisesti toteutettua testausprosessia. Jokainen selain tulkitsee koodit omalla tavallaan, joten testien kirjoittaminen useammalle selaimelle voi olla haastavaa. (Fahad, 2022)

3.3 Testiautomaatiotyökalujen vertailu

Testiautomaatiotyökalu on pohjimmiltaan ohjelmistoviitekehys, joka tarjoaa valmiin pohjan automatisoitujen testien tekemiseen. Se on räätälöity työkalu, joka on suunniteltu erityisesti ylläpitämään ja optimoimaan testausprosessia parempien tulosten saavuttamiseksi. Vaikka useimmat testiautomaatiotyökalut sisältävät valmiin oletuskehyksen, niitä voidaan myös mukauttaa vastaamaan omiin tarpeisiin. Työkalu voidaan esimerkiksi asettaa suorittamaan testejä ajoitetusti tavalla, joka parhaiten palvelee omaa tarvetta sekä luomaan kustomoituja testiraportteja. (Boby, 2021 s. 5)

Testiautomaatioon siirtyessä organisaatiot lähtevät usein hankkimaan uutta ohjelmistoa tekemättä perusteellista tutkimusta ja työkalujen vertailua. Tämä saattaa johtaa väärinkäsityksiin omiin tarpeisiin parhaiten sopivasta lisenssimuodosta, saatavilla olevista tukipalveluista sekä osaamisen saatavuudesta. Tämän myötä testiautomaatiolla ei päästä tavoiteltuihin tuloksiin ja organisaatio ei ole tyytyväinen ohjelmistoon. Ohjelmistojen toimittajat käyttävät usein eri termejä lisensseistä, joilla jokaisella on omanlaiset kustannuksena. Tämä voi olla aluksi hämmentävää ja tehdä valinnasta haastavaa. Lisäksi monet avoimen lähdekoodin työkalut näyttävät aluksi kustannustehokkailta, mutta voivatkin lopulta sisältää lisäkuluja, jos tehokkaaseen käyttöön tarvitaankin lisäosia tai muita kolmannen osapuolen ratkaisuja. (Boby, 2021 s. 131)

Seuraavissa alaluvuissa perehdytään paremmin viiteen suosittuun testiautomaatiotyökaluun, joiden ominaisuudet esitellään lyhyesti ensin alla olevassa taulukossa (Taulukko 1).

Taulukko 1 Testiautomaatiotyökalujen vertailu

TYÖKALU	OHJELMOINTIKIELET	NAUHOITUSTYÖKALU	SELAIMET	LISENSSI
Selenium	Java, Javascript, C#, Python, Ruby	Kyllä	Google Chrome, Mozilla Firefox, Microsoft Edge, Safari, Opera, IOS & Android	Open-source
Cypress	JavaScript/TypeScript	Kyllä	Google Chrome, Chromium, Microsoft Edge, Mozilla Firefox, Webkit	Open-source
Playwright	Javascript/Typescript, Python, Java, C# (.Net)	Kyllä	Google Chrome, Chromium, Microsoft Edge, Mozilla Firefox, Apple Safari, Webkit	Open-source
Puppeteer	JavaScript/TypeScript	Kyllä	Chromium	Open-source
Test Complete	JavaScript, Python, VBScript, JScript, Delphi, C++, C#	Kyllä	Internet Explorer,Microsoft Edge,Google Chrome Mozilla Firefox, Safari	Kaupallinen

3.3.1 Selenium

Selenium on suuren suosion kasvattanut testiautomaatiotyökalu. Selenium perustuu avoimeen lähdekoodiin. Se tukee useita eri ohjelmointikieliä, kuten Javaa, C# ja Pythonia. Selenium koostuu neljästä eri komponentista, jotka ovat suunniteltu eri käyttötarkoituksiin. Komponentit ovat nimeltään IDE, Remote Control, WebDriver ja Grid. (Manasa, 2019)

Selenium IDE on Chrome ja Firefox-selaimille tarkoitettu lisäosa, jonka avulla voidaan nauhoittaa testejä perustuen graafisessa käyttöliittymässä tehtyihin toimintoihin. IDE on yksinkertaisin Selenium-työkalu ja helpoin käyttää. IDEn avulla testejä voidaan kirjoittaa myös ilman ohjelmointitaitoja nauhoitustyökalulla. Myös sen käyttöönotto on helppoa, koska siihen vaaditaan ainoastaan lisäosan asentaminen. IDEssä on kuitenkin rajoituksia sen yksinkertaisuuden takia, eikä sitä suositella esimerkiksi laajojen kokonaisuuksien tai tietokantojen testaamiseen. IDEllä ei ole toimintoa testiraporttien luomiseen. Selenium IDE kannattaa valita silloin, kun testit ovat yksinkertaisia eikä testejä tarvitse suorittaa useammalla eri selaimella. (Manasa, 2019)

Selenium RC oli pitkään koko Selenium-kehyksen päätyökalu. RC oli ensimmäinen automaattinen verkkotestaustyökalu, jota käyttäjät pystyivät käyttämään haluamallaan ohjelmointikielellä. RC:n avulla testejä voidaan suorittaa nopeammin, kuin IDE:llä, mutta

asentaminen on monimutkaisempaa ja testien luominen vaatii ohjelmointitaitoja. RC tukee testien suorittamista useilla eri selaimilla sekä alustoilla. RC:n käyttö vaatii RC palvelimen kommunikoidakseen selaimen kanssa, jonka vuoksi se on hitaampi kuin WebDriver. (Manasa, 2019)

WebDriver kommunikoi suoraan selaimen kanssa, joka tekee siitä nopeamman kuin muut Seleniumin työkalut. Se on myös helpompi asentaa, kuin RC, eikä se tarvitse muita komponentteja toimiakseen. WebDriverin käyttö vaatii kuitenkin ohjelmointiosaamista eikä se sisällä sisäänrakennettua ominaisuutta testiraporttien luomiseen. (Manasa, 2019)

Selenium Grid on tarkoitettu käytettäväksi yhdessä Selenium RC:n kanssa, jolloin sillä voidaan suorittaa useita testejä samanaikaisesti eri koneilla ja selaimilla. Sen avulla testien suorittamisessa voidaan säästää huomattavasti aikaa. Grid on hyvä valinta silloin, kun testejä on paljon ja ne pitää saada suoritettua mahdollisimman nopeasti. (Manasa, 2019)

3.3.2 Cypress

Cypress on monipuolinen ja helppokäyttöinen avoimen lähdekoodin työkalu, joka on kehitetty verkkosovellusten päästä päähän testaukseen. Cypress on verrattain uusi testiautomaatioteknologia, jonka käyttö on ollut viime aikoina suuressa kasvussa. Toisin kuin Selenium, Cypress tukee ainoastaan JavaScriptiä ohjelmointikielenä. Tämän vuoksi Cypress on ollut suosiossa erityisesti front-end kehittäjien sekä JavaScriptiä osaavien testaajien keskuudessa. Cypressiä voidaan käyttää myös API-testaukseen ja se tukee eri selainversioita monista suosituista selaimista, kuten Google Chromesta, Mozilla Firefoxista ja Microsoft Edgestä. (Cypress Tutorial, ei pvm.)

Cypressin parhaita puolia ovat sen nopeus ja luotettavuus testien suorittamisessa. Se on myös nopea ja helppo asentaa, eikä sen käyttöön tarvita lisäksi muita kirjastoja tai ohjelmia. Cypress ottaa kuvakaappauksia testien suorituksen aikana, joka on hyödyllinen ominaisuus. Cypressin heikkouksia on suppea ohjelmointikielivalikoima, sen tukiessa ainoastaan JavaScriptiä. Sillä ei pystytä suorittamaan testejä useammalla selaimella tai välilehdellä samanaikaisesti. Cypress ei tue XPath paikantimia, vaan niiden käyttöä varten tulee asentaa erillinen lisäosa. (Pathak, 2023)

3.3.3 Playwright

Playwright on Microsoftin kehittämä avoimen lähdekoodin testiautomaatiotyökalu verkkosovellusten testaamiseen. Playwright tukee useita eri ohjelmointikieliä, kuten

JavaScriptiä, Pythonia, C# ja Javaa. Playwright toimii hyödyntämällä yhtä ohjelmointirajapintaa, jossa voidaan simuloida käyttäjän toimintoja useilla eri selaimilla. (Fast and Reliable End-to-End Testing for Modern Web Apps | Playwright, ei pvm.)

Playwrightin vahvuuksia ovat sen tuki monille eri selaimille, ohjelmointikielille ja käyttöalustoille. Playwright on myös helppo asentaa ja ottaa käyttöön. Playwright tukee automaattista odotustoimintoa, jolloin se odottaa automaattisesti esimerkiksi sivun avautuvan tai elementin tulevan näkyviin. Playwright hyödyntää niin sanottuja selainkonteksteja, joita voidaan luoda eri evästeillä, käyttäjäagenteilla tai välimuisteilla. Selainkontekstien avulla samoja testejä voidaan suorittaa eri ominaisuuksin. Playwrightilla on oma työkalu testien nauhoitukseen, jolloin testejä ei tarvitse kirjoittaa itse. Playwright pystyy ottamaan kuvakaappauksia sekä nauhoittaa videoita testien suorituksesta, joiden avulla voidaan paikantaa virheet tehokkaasti. Playwrightilla on myös elementtien paikantamiseen oma työkalu, jonka myötä elementtejä ei tarvitse paikantaa itse sivustolta testejä kirjoittaessa. (Pathak, 2023)

Playwrightin ollessa uusi työkalu markkinoilla, sen kehittäjäyhteisö on vielä suppeampi, kuin monilla kilpailijoilla. Playwrightin heikkouksia on myös se, että mobiilitestaus tapahtuu simuloimalla oikeaa mobiililaitetta oikean laitteen hyödyntämisen sijaan. (Pathak, 2023)

3.3.4 Puppeteer

Puppeteer on Node-pohjainen kirjasto, joka hyödyntää yhtä ohjelmointirajapintaa Chrome-selainten automatisointiin DevTools-protokollan kautta. Puppeteerillä voidaan suorittaa automatisoituja testejä verkkosovelluksiin. Pohjimmiltaan Puppeteer on kuitenkin tarkoitettu enemmän automatisointiin, kuin testaukseen. Puppeteeria käytetään usein esimerkiksi PDF-tiedostojen luomisen automatisointiin sekä tietojen poimimiseen verkosta. (Neha, 2023)

Puppeteerin hyviä puolia ovat sen kyky suorittaa testejä erittäin nopeasti, koska se hyödyntää ainoastaan JavaScriptiä kielenään. Se on myös helppo asentaa ja käyttää. Puppeteerillä on myös oma nauhoitustyökalu, jonka avulla testiskriptejä voidaan nauhoittaa, eikä ohjelmointiosaamista tarvita. (Nechytailo, 2023)

Puppeteerin heikkouksia ovat sen suppea selaintuki, sen toimiessa ainoastaan Chrome-selaimilla. Vaikka pelkästä JavaScriptin tukemisesta on hyötyä testien suoritusnopeudessa, voi suppeaa ohjelmointikielivalikoimaa pitää myös heikkoutena. (Nechytailo, 2023)

3.3.5 TestComplete

TestComplete on kaupallinen UI-testiautomaatiotyökalu, jonka on kehittänyt SmartBear Software. Hinta vaihtelee tilauspaketin mukaan. TestComplete tarjoaa myös 30 päivän ilmaisia kokeilujaksoja uusille asiakkailleen. Sen avulla voidaan luoda automatisoituja testejä verkko-, työpöytä- ja mobiilisovelluksille. TestCompletella testien luominen, ylläpitäminen sekä suorittaminen on helppoa ja nopeaa. Sillä on oma hybridi objektintunnistusmoottori, joka hyödyntää tekoälyä. Ominaisuuden avulla ohjelma tunnistaa ja löytää käyttöliittymän elementit nopeasti ja helposti. Testejä sillä voidaan kirjoittaa seitsemällä eri ohjelmointikielellä. (Mahajan, 2023)

Vaikka TestComplete on erittäin tehokas työkalu testiautomaatioon, se ei ole kovin suosittu sen korkeiden kustannusten vuoksi. TestCompleten heikkouksia on myös käyttöjärjestelmätuki ainoastaan Windowsille. Pienen käyttäjämäärän vuoksi sillä on pieni kehittäjäyhteisö, joka vaikeuttaa tuen saamista. (Mahajan, 2023)

4 Ketterät menetelmät

Lisääntynyt kilpailu on painostanut yrityksiä nopeuttamaan tuotteiden markkinoille vientiä sekä parantamaan tuotteidensa laatua. Ilmiö näkyy erityisesti ohjelmistokehitysmarkkinoilla, jossa internet mahdollistaa tuotteiden lähes välittömän toimittamisen. Tänä päivänä asiakkaat vaativat korkealaatuisia sovelluksia mahdollisimman nopeasti toimitettuina, eikä perinteiset ohjelmistokehitysprosessit pysty vastaamaan vaatimuksiin. (Myers ym., 2012, ss. 175–176)

Vuonna 2001 ryhmä ohjelmistokehittäjiä kokoontui yhteen keskustelemaan siitä, mikä yhdisti heidän hyvin onnistuneita projektejaan. Keskustelun myötä syntyi ketterän ohjelmistokehityksen julistus, joka yleisesti tunnetaan Ketterä-manifestina. (Agile manifesto) Se toimii arvojen julistuksena onnistuneelle ohjelmistokehitykselle, jossa yksilöitä ja vuorovaikutusta arvostetaan prosessien ja työkalujen sijaan, toimivaa ohjelmistoa dokumentaation sijaan, asiakasyhteistyötä neuvottelujen sijaan sekä muutokseen reagoimista suunnitelman tarkan noudattamisen sijaan. (Layton & Ostermiller, 2020, s. 13)

Ketterä on käytännössä joukko erilaisia menetelmiä, jotka auttavat tiimiä ajattelemaan, työskentelemään sekä tekemään päätöksiä tehokkaammin. Menetelmät koskevat kaikkia ohjelmistokehityksen alueita, kuten suunnittelua, projektinhallintaa ja arkkitehtuuria. Jokainen menetelmä koostuu optimoiduista käytännöistä, jotka on helppo ottaa käyttöön. (Greene & Stellman, 2015, s. 2)

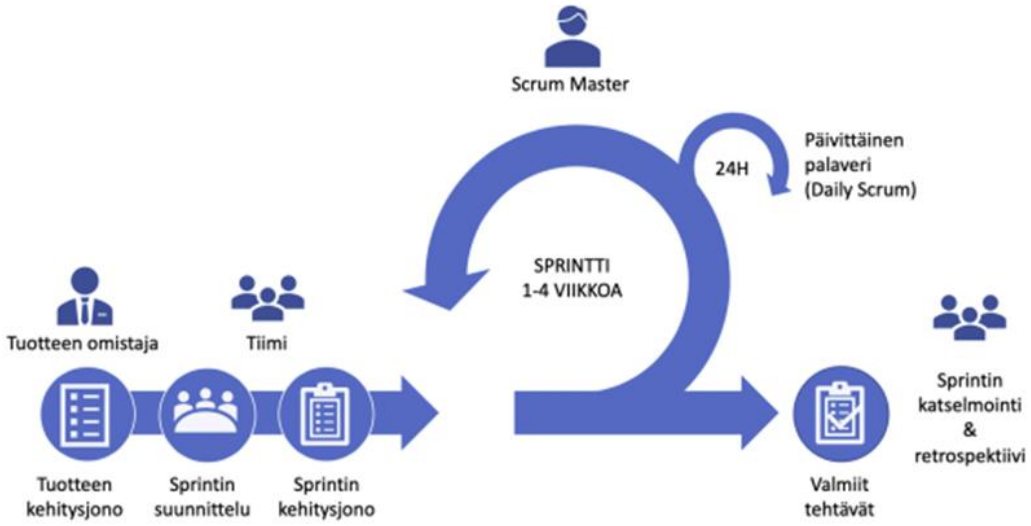
Ketterään kuuluu olennaisesti asiakaslähtöisyys ja valmius muutoksille kesken projektin. Ketterä kehitys perustuu prosessin jakaminen pieniin, toistuviin vaiheisiin. Perinteisten ohjelmistokehitysmenetelmien ongelmana on ollut erityisesti asiakkaan merkityksen pitäminen vähäisenä. Vaikka ketterät menetelmät tekevätkin prosesseista myös joustavampia, on pääpaino asiakastyytyväisyydessä. Asiakas on keskeinen osa ketterää projektia, eikä ketteriä menetelmiä voida toteuttaa onnistuneesti ilman asiakkaan osallistumista. Tilanteessa, jossa asiakas ei kykene sitoutumaan täysin projektiin, on perinteiset kehitysmenetelmät todennäköisesti parempi valinta. (Myers ym., 2012, ss.175–176)

Ketterällä kehityksellä ei kuitenkaan tarkoiteta tiettyä menetelmää eikä se vaadi tiettyjen teknisten käytäntöjen noudattamista, vaikka ne antavatkin hyvän perustan. Näitä tärkeämpää ketterissä menetelmissä on arvot sekä periaatteet, joiden myötä siirrytään kohti ihmiskeskeisempää työskentelytapaa. Tämä auttaa tiimejä kehittämään ohjelmistoja vastaamaan paremmin asiakkaiden tarpeisiin. (Flewelling, 2018, s. 30).

4.1 Scrum

Ketterien menetelmien toteuttamiseen on kehitetty lukemattomia eri menetelmiä, joista Scrum on epäilemättä käytetyin. Scrumille ominaista on työn toteuttaminen lyhyissä sykleissä, joita kutsutaan sprinteiksi (Kuva 5). Scrumissa kehitykseen käytettävä aika pyritään maksimoimaan tuotetavoitteen saavuttamiseksi. Scrumia käytetään yleisimmin ohjelmistokehityksessä, mutta sitä voidaan hyödyntää menestyksellisesti myös liiketoiminnan projekteissa. Scrumia noudattaessa jokainen päivä aloitetaan 15 minuutin palaverilla tiimin kesken, jossa käydään läpi tehtävien edistyminen ja mahdollisesti ilmenneet ongelmat. Palaverin perusteella suunnitellaan myös päivän työtehtävät. (Santo, 2022)

Kuva 5 Scrum-prosessi (Walden, 2020, s.14, Mukailtu Gonçalves 2018)



Ketterässä manifestissa linjataan parhaiden tulosten syntyvän itseorganisoituvista tiimeistä. Manifestin periaatteiden mukaisesti Scrum-tiimi saa itse päättää sopivimman tavan projektin vaatimusten toteuttamiseen. Tiimin koko ei tulisi kuitenkaan olla liian suuri, jotta tiimi pystyy ylläpitämään riittävää yhteistyö- ja kommunikointitasoa, joiden avulla saadaan tuotettua laadukkaita tuotteita. Scrumissa kehitystiimiin kuuluu yleisesti noin 5–9 jäsentä. Tätä pienemmät tiimit taas saattavat kohdata haasteita taitojen riittävyydessä. (Measy & Radtac., 2015, s. 134)

Scrum-tiimin jäsenillä on kolme eri roolia: Scrum master, tuoteomistaja sekä kehitystiimi. Scrum masterin vastuulle kuuluu kehitystiimin ohjaus, jotta tiimi pystyy toimimaan mahdollisimman tehokkaasti saavuttaakseen tavoitteet. Tuoteomistaja määrittelee, mitä tehdään ja missä järjestyksessä, kun kehitystiimi taas määrittää, miten tehdään ja missä

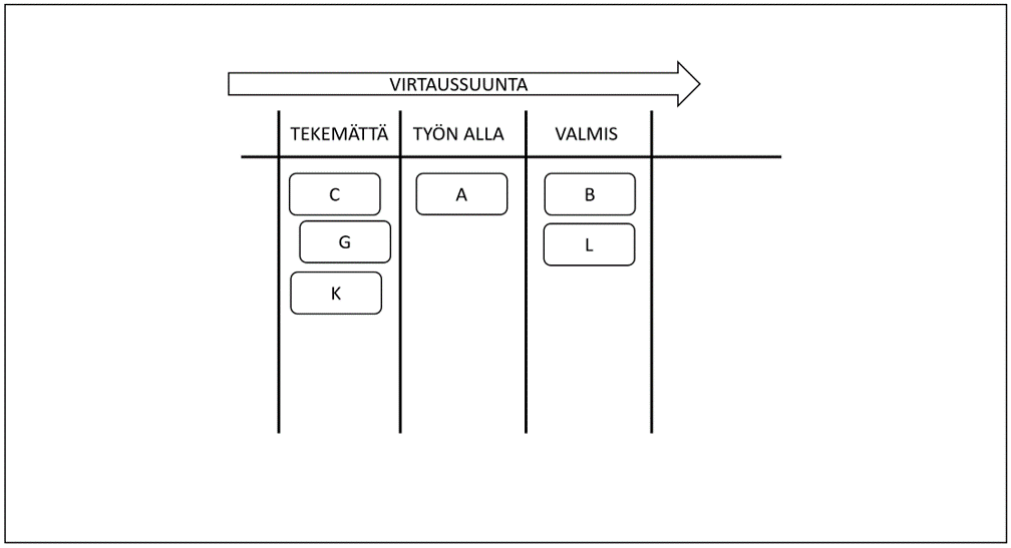
ajassa. Tiimin päätavoitteena on saavuttaa tuoteomistajan kanssa sovitut tavoitteet. (Measy & Radtac., 2015, s. 132)

Scrum prosessi koostuu sprinteistä, jotka kestävät yleensä yhdestä neljään viikkoon. Sprintit ovat työkokonaisuuksia, joista jokaisessa on tavoitteena saada toteutettua asiakkaalle jotakin toimitettavaa. Projektin alussa tulevien sprinttien määrä ei tarvitse olla tiedossa, vaan sprinttejä jatketaan niin kauan, kuin projektin kannalta on tarpeen. Jokaisen sprintin alussa tavoitteet voidaan määrittää uudelleen. Scrum eroaa tässä perinteisistä ohjelmistokehitysmalleista, joissa ensin suunnitellaan koko projektin tavoitteet ja sitten toteutetaan. (Juvonen, 2018, s. 20)

4.2 Kanban

Kanban on visuaalinen taulu, jonka avulla työ tehdään läpinäkyvämmäksi. Taulussa tehtävät kirjoitetaan korteille ja jaetaan omiin sarakkeisiin niiden vaiheiden mukaisesti. Yksinkertaisimmillaan Kanban-taulu sisältää kolme saraketta: tekemättä, työn alla ja tehty (Kuva 6). Tehtävän edetessä se siirretään sarakkeesta seuraavaan, kunnes tehtävä on suoritettu. Sarakkeita voidaan kuitenkin lisätä tarpeen mukaan. Kanban voidaan toteuttaa digitaalisena tai fyysisenä esimerkiksi toimiston seinällä. (Herranen, 2020, s. 51)

Kuva 6 Esimerkki yksinkertaisesta Kanban-taulusta (Hietaniemi, 2020)



Sana ”Kanban” tarkoittaa japaniksi kylttiä. Kanban-menetelmä sai alkunsa Toyotan tehtaalta, jossa alettiin hyödyntää kortteja tuotannon edistymisen seuraamiseksi. Syy Kanbanin kehitykseen oli Toyotan tehtaan riittämätön tuottavuus ja tehokkuus verrattuna sen kilpailijoihin. Kanbanin avulla Toyota saavutti joustavan ja tehokkaan

tuotannonohjausjärjestelmän, jonka avulla pienennettiin kustannuksia. Tänä päivänä se on laajalti käytössä myös autoteollisuuden ulkopuolella, monilla eri aloilla. (Bhaskar, 2023)

Kanban-taulusta tiimin jäsenet näkevät välittömästi, miten tehtävät etenevät prosessin aikana, joka kasvattaa projektin läpinäkyvyyttä. Kanbanin käyttäminen lisää tiimin tehokkuutta, joka luonnollisesti johtaa myös toiseen hyötyyn, tuottavuuden kasvuun. Yleinen harhaluulo on, että monen asian tekeminen saamaan aikaan lisää tehokkuutta, mutta tosiasiassa se voi lisätä hukkaan menevää aikaa, kun tehtävää joudutaan vaihtamaan usein. Kanbanin avulla tiimin jäsenet saavat keskittyä yhteen tehtävään kerrallaan sen sijaan, että huomion veisi useampi asia samanaikaisesti. (Siderova, 2018)

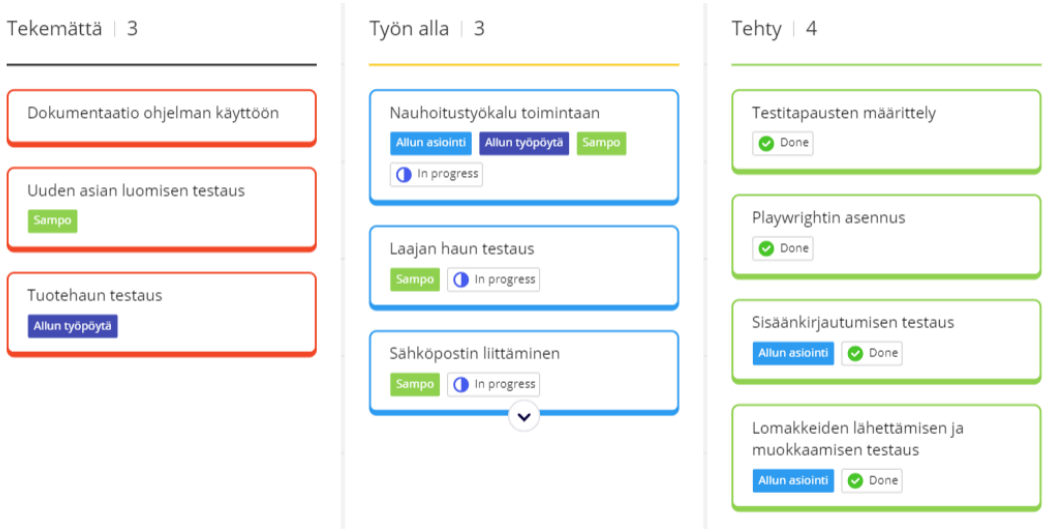
5 Työn tavoite ja menetelmät

Työn tavoitteena oli löytää toimeksiantajan tarpeisiin sopiva testiautomaatiotyökalu ja toteuttaa testiautomaatio-ohjelma, jota voidaan tulevaisuudessa hyödyntää toimeksiantajan järjestelmien regressiotestaukseen. Ohjelman käyttöön tuli laatia myös kattava dokumentaatio, koska käytetty testiautomaatiotyökalu oli toimeksiantajalle uusi ja laaditut testit vaativat ylläpitoa. Testiautomaation käyttöön siirtymisen päätavoitteena oli toimeksiantajan järjestelmien laadun parantaminen.

Työn alkaessa toimeksiantajan järjestelmät olivat itselleni vieraita, joten opinnäytetyön tutkimusmenetelmänä hyödynnettiin teemahaastatteluja. Teemahaastattelujen tavoitteena oli kerätä tietoa järjestelmän toiminnallisuuksista, jonka pohjalta myös testitapaukset laadittiin. Teemahaastatteluissa haastateltiin kahta Valviran järjestelmäasiantuntijaa, jotka ovat vastuussa testattavien järjestelmien ylläpidosta ja niiden testaamisesta. Haastattelut suoritettiin etänä Teamsin välityksellä.

Projektinhallintamenetelmänä hyödynnettiin sovelletusti ketteriä menetelmiä. Pidimme esimiehen kanssa viikoittaisen tapaamisen, jossa seurattiin työn edistymistä. Tapaamisessa käytiin läpi mahdollisia ongelmia sekä päätettiin seuraavat työtehtävät. Projektin aikana hyödynnettiin myös visuaalista Kanban-taulua, jonka avulla työn vaiheet ja niiden tila oli helpompi hahmottaa. Alla olevassa kuvassa nähdään projektissa käytetty Kanban-taulu kesken projektin (Kuva 7).

Kuva 7 Projektin Kanban-taulu



6 Testiautomaatio-ohjelman toteutus

Tässä luvussa käydään läpi opinnäytetyön käytännönosuus, jossa kerrotaan testiautomaatiotyökalun valintaprosessista, asennetaan valittu testiautomaatio-ohjelmisto, jolla kirjoitetaan ja suoritetaan testejä. Lopuksi käydään läpi projektin aikana ilmenneitä haasteita.

6.1 Testiautomaatiotyökalun valinta

Testiautomaatiotyökalun valinta lähti liikkeelle huolellisesta kartoituksesta, jossa käytiin läpi toimeksiantajan tarpeita ja vaatimuksia ohjelmistolle. Testien helppo ylläpidettävyys nousi esiin yhtenä tärkeimpänä vaatimuksena. Testien ylläpidon olisi hyvä onnistua ilman ohjelmointitaitoja ja syvempää teknistä osaamista. Lisäksi toimeksiantajalla oli oma ohjeistus avoimen lähdekoodin ohjelmistojen valintaan, joka oli otettava huomioon päätöstä tehtäessä.

Toimeksiantajan ohjeistuksessa korostettiin, että avoimen lähdekoodin työkalun valinnassa tulisi arvioida, onko ohjelmistolla ajan tasalla oleva ja kattava dokumentaatio. Ohjelmistolla toivottiin olevan taustalla tunnettuja ja luotettavia toimijoita, kuten Microsoft tai Google. Lisäksi painotettiin, että ohjelmalla olisi hyvä olla laaja käyttäjäyhteisö, joka parantaa tukipalvelujen saatavuutta. Tärkeää oli myös selvittää, onko ohjelman lisenssi aidosti avoin vai liittyykö siihen mahdollisia lisäehtoja, jotka voivat tuoda kustannuksia. Normaalisti olisi myös huomioitu saatavilla oleva osaaminen kyseiselle ohjelmistolle, mutta tässä tapauksessa päätettiin, ettei ulkopuolista osaamista tarvita, sillä kehitys- ja ylläpitotyö tehdään itse organisaation sisällä.

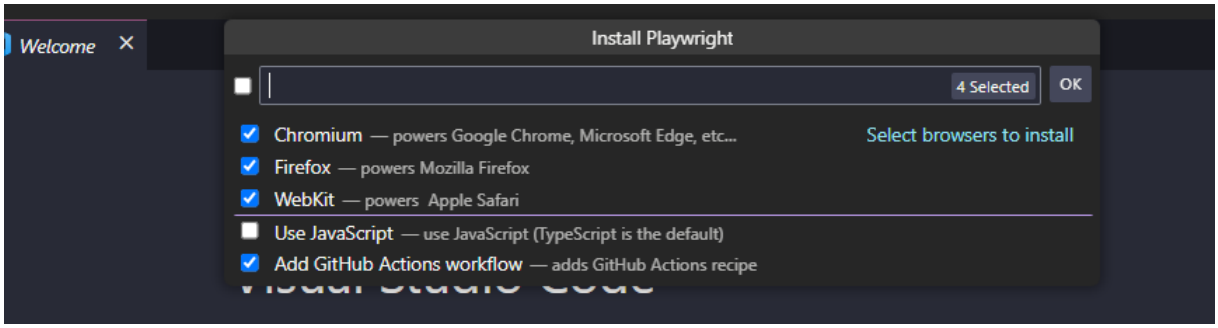
Toimeksiantajan vaatimusten pohjalta vertailtiin eri testiautomaatiotyökaluja, jonka perusteella valittiin Microsoftin kehittämä avoimen lähdekoodin Playwright-ohjelmisto. Playwright erottui kilpailijoistaan erinomaisella testien nauhoitustyökalullaan, joka vastasi toimeksiantajan tarpeisiin helppokäyttöisyyden ja testien ylläpidettävyyden suhteen. Uusien testien laatiminen olisi vaivatonta ilman ohjelmointiosaamista. Lisäksi Playwrightin asennus ja käyttöönotto osoittautuivat helpoiksi verrattuna moniin muihin vaihtoehtoihin. Playwrightilla pystytään suorittamaan testejä usealla eri selaimella samanaikaisesti, mikä lyhentää testaukseen kuluvaa aikaa. Lisäksi sillä on aktiivinen ja kasvava kehittäjäyhteisö, mikä takaa tukipalvelujen hyvän saatavuuden.

6.2 Playwrightin käyttöönotto

Playwright vaatii toimiakseen Node.js-ympäristön, joten se asennettiin ensin tietokoneelle Node.js-ympäristön virallisilta verkkosivuilta. Seuraavaksi siirryttiin Playwrightin asennukseen. Playwrightin voi asentaa helposti joko komentokehotteen kautta tai hyödyntämällä tekstieditori Visual Studio Coden (lyh. VSCode) Playwright-lisäosaa. Tässä projektissa asennus tehtiin VSCoden kautta.

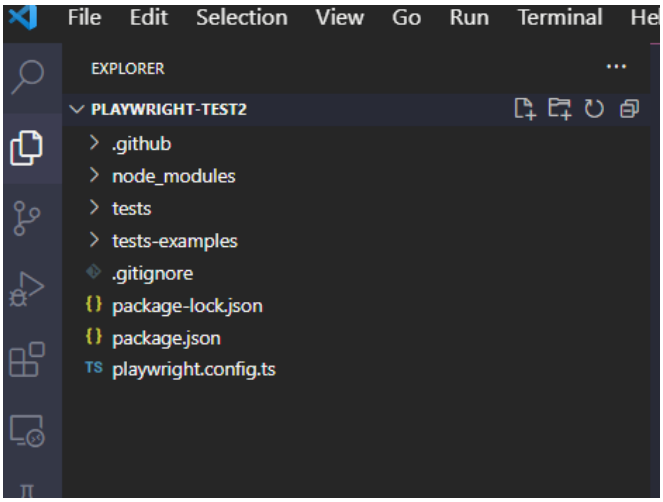
Playwrightin asennus aloitettiin luomalla uusi kansio C-asemalle, jonka jälkeen kansio avattiin tekstieditorissa. Tämän jälkeen avattiin tekstieditorin Extensions -välilehti, josta etsittiin ”Playwright Test for VScode” -lisäosa ja asennettiin se. Seuraavaksi klikattiin yläreunasta View > Command Palette > Test: Install Playwright, josta valittiin tarvittavat selaimet testaamiseen sekä ohjelmointikieli Typescriptin ja Javascriptin välillä (Kuva 8). Tässä projektissa valittiin selaimiksi chromium, firefox sekä webkit ja ohjelmointikieleksi TypeScript.

Kuva 8 Playwrightin asennus



Valintojen jälkeen klikattiin OK, jonka jälkeen lisäosa loi asennuskomennon VScoden terminaaliin. Kun asennus oli mennyt onnistuneesti läpi, näkyi VScoden vasemmassa reunassa kansiossa lisäosan lataamat oletuskansiot ja -tiedostot (Kuva 9).

Kuva 9 Playwrightin oletuskansiot



Playwrightin asentamassa ”playwright.config”-tiedostossa on oletusasetukset testien ajamiseen ja tiedostoa voidaan muokata omien tarpeiden mukaan. Tiedostossa voidaan esimerkiksi määrittää mitä selaimia testaamiseen käytetään. Oletuksena Playwright suorittaa testejä sen kaikilla kolmella selaimella; firefoxilla, chromiumilla ja webkitillä.

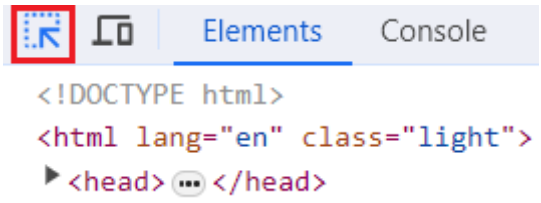
Tests-kansio on testien kirjoittamista varten. Kansio sisältää yksinkertaisen esimerkkitestin, joka antaa hyvän kuvan testien rakenteesta. Yksityiskohtaisempi esimerkki löytyy tests-examples-kansiosta, joka sisältää valmiita testejä Playwrightin verkkosovellukseen.

6.3 Elementtien paikantaminen

Testien kirjoittamista varten tulee ensin paikantaa elementit testattavalta verkkosivulta, jotta ohjelmalla voidaan simuloida käyttäjän tekemiä toimintoja. Esimerkiksi etsitään painike, jota klikataan. Elementtejä voidaan paikantaa helposti Playwrightin nauhoitustyökalulla, jota käydään läpi myöhemmin tässä luvussa. Playwrightilla on myös sisäänrakennettuja paikantimia, joiden avulla elementit voidaan tunnistaa testejä varten. Jotta testeistä saadaan mahdollisimman pitkään toimivia, suositellaan ensisijaisesti attribuutteja hyödyntäviä paikantimia, kuten esimerkiksi GetByRole-paikanninta. Tämä paikannin tunnistaa elementin sen nimen ja roolin perusteella, jotka on määritetty sivun rakenteeseen.

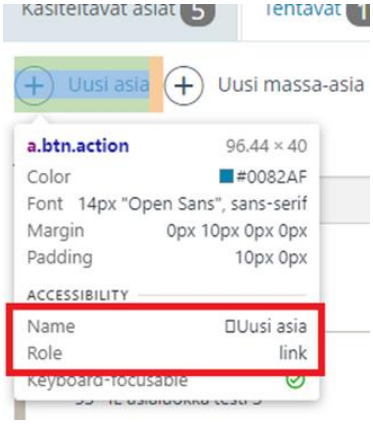
Käydään seuraavaksi läpi esimerkin kautta, miten elementti paikannetaan getByRole-paikanninta käyttämällä. Elementin paikantaminen aloitetaan menemällä testattavalle sivulle, jonka jälkeen klikataan tarvittavaa elementtiä hiiren oikealla. Seuraavaksi klikataan ”Tarkista”, jonka jälkeen oikealle avautuu ikkuna, jossa valittu elementti näkyy korostettuna. Valitaan ikkunan vasemmasta yläreunasta valitsin (Kuva 10).

Kuva 10 Valitsin



Seuraavaksi valitsin viedään tarvittavan elementin päälle, jolloin nähdään elementin nimi ja rooli, jos sellaiset on määritetty sivun rakenteeseen (Kuva 11).

Kuva 11 Elementin rooli ja nimi



Kun elementti on paikannettu, voidaan sille tehdä tarvittava toiminto, kuten klikkaus. Seuraavaksi voidaan muodostaa koodi, jossa paikannetaan elementti sen roolin sekä nimen avulla ja lopuksi klikataan elementtiä (Kuva 12).

Kuva 12 Elementin paikantaminen GetByRole-paikantimella

```
await page.getByRole('link', { name: 'Uusi asia' }).click();
```

Playwrightin sisäänrakennettujen paikantimien lisäksi, elementtejä voidaan paikantaa CCS tai XPath paikantimilla. Näitä paikantimia käyttäessä tulee huomioida, että ne saattavat lakata toimimasta, jos sivun rakenteeseen tehdään muutoksia. Tämän takia XPath ja CSS paikantimia kannattaa käyttää vain silloin, jos muita tapoja ei ole käytettävissä elementin tunnistamiseen. Elementin XPath saadaan haettua klikkaamalla tarvittavaa elementtiä, jonka jälkeen klikataan Tarkista. Tämän jälkeen klikataan hiiren oikealla konsolissa korostettuna näkyvää elementtiä ja valitaan Kopioi ja Kopioi XPath. Seuraavaksi voidaan muodostaa koodi, joka etsii elementin XPathin perusteella (Kuva 13).

Kuva 13 Elementin paikantaminen XPath-paikantimella

```
await page.locator('//*[@id="app"]/div[1]/div/div[2]/div/div/div/div/div/div/div[1]/div/input')
```

6.4 Testien kirjoittaminen ja suorittaminen

Asennuksen jälkeen aloitettiin ensimmäisien testien kirjoittaminen. Järjestelmää käyttääkseen käyttäjän tulee ensin kirjautua sisään Suomi.fi-tunnuksilla, joten ensimmäisessä testissä testattiin sisäänkirjautumista. Koska sisäänkirjautuminen tulisi suorittaa myös myöhemmin ennen jokaista testiä, päätettiin tallentaa tunnistautunut tila JSON-tiedostoon, jotta sitä voitaisiin hyödyntää seuraavissa testeissä ja testien suorittamiseen kuluva aika lyhenisi.

Testin kirjoittaminen aloitettiin luomalla uusi ”auth.setup.ts” niminen tiedosto, jonka avulla tunnistautunut tila tallennettiin JSON-tiedostoon. Tiedostoon määritettiin skripti, jossa Playwright navigoi ensin testattavan järjestelmän osoitteeseen, valitsee sitten oikean tunnistautumistavan ja syöttää henkilötunnuksen. Tämän jälkeen Playwright klikkaa ”Tunnistaudu”-painiketta, jonka jälkeen se tarkastaa, onko asiointipalvelun etusivun otsikko näkyvillä, jonka avulla voidaan nähdä, menikö tunnistautuminen läpi. Lopuksi Playwright tallentaa sivuston evästeiden avulla tunnistautumisen tiedot JSON-tiedostoon (Kuva 14).

Kuva 14 Tunnistautuneen tilan tallentaminen JSON-tiedostoon

tests > TS auth.setup.ts > ...

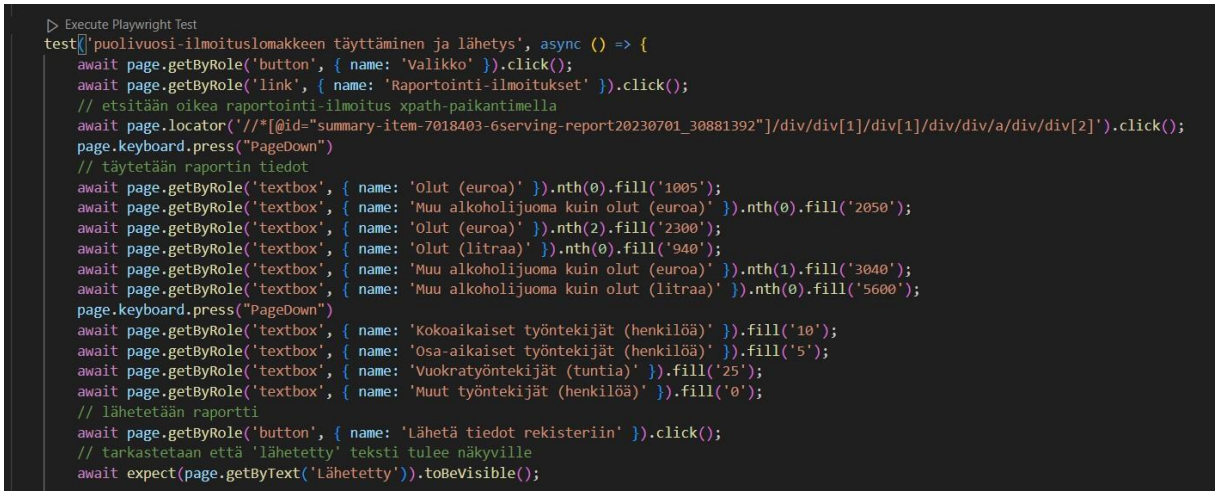
```
import { test as setup, expect } from '@playwright/test';

// tallennetaan testattavan järjestelmän osoite muuttujaan
const URL = [redacted]

// määritetään json-tiedostopolku muuttujaan
const authFile = './loggedinState.json';

setup('authenticate', async ({browser})  => {
 const context = await browser.newContext({

 })
const page = await context.newPage();

// navigoidaan testattavalle sivulle
await page.goto(URL);
// täytetään tunnistautumistiedot
await page.getByRole('link', {name: 'Testitunnistaja'}).click();
await page.getByRole('textbox', {name: 'Henkilötunnus'}).click();
await page.getByRole('textbox', {name: 'Henkilötunnus'}).fill('[redacted]');
await page.getByRole('button', {name: 'Tunnistaudu'}).click();
await page.getByRole('button', {name: 'Jatka palveluun'}).click();
// tarkastetaan että asiointipalvelun etusivun otsikko on näkyvillä
await expect(page).toHaveTitle('Valvira - Sähköinen asiointipalvelu')
// tallennetaan tunnistautuneen tilan evästeet json tiedostoon
await page.context().storageState({ path: authFile });
await context.close();

})
```

Seuraavaksi lisättiin vielä konfiguraatiotiedostoon asetus, jonka avulla Playwright osaa hyödyntää JSON-tiedostoa sisäänkirjautumiseen seuraavissa testeissä. Tämä tehtiin lisäämällä selaimien alle ”storageState”-asetus, johon määritettiin JSON-tiedoston polku (Kuva 15).

Kuva 15 JSON-tiedoston käyttö testien suorituksessa

Seuraavaksi siirryttiin asiointipalvelun lomakkeiden testaamiseen, jossa Playwright hyödyntää tallennettua JSON-tiedostoa, eikä tunnistautumisvaihetta tarvitse enää tehdä. Testissä Playwright syöttää lomakkeen kenttiin tiedot, jonka jälkeen klikataan ”Lähetä lomake”-painiketta. Lomaketta ei pysty lähettämään, jos arvot on syötetty väärin tai pakollisia arvoja puuttuu. Lomakkeen lähettämisen jälkeen tarkistetaan vielä, onko ”Lähetetty”-teksti näkyvillä, josta tiedetään, onko lomakkeen lähettäminen onnistunut (Kuva 16 Lomakkeen testaus.

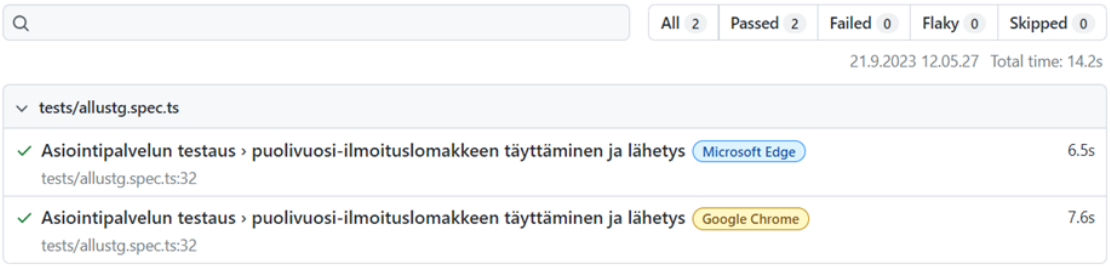
Kuva 16 Lomakkeen testaus



Testin mennessä onnistuneesti läpi, terminaaliin tulostuu teksti ”passed”, kun taas epäonnistuessa tulostuu teksti ”failed”. Jos testi epäonnistuu, Playwright avaa automaattisesti tarkemman testiraportin. Tässä projektissa testiraportti oli määritetty html-

muotoon, mutta Playwright tukee myös muita muotoja ja se voidaan määrittää konfiguraatiotiedostossa. Raportti voidaan avata myös aina tarvittaessa syöttämällä terminaaliin komento ”npx playwright show-report”. Seuraavassa kuvassa näkyvästä testiraportista nähdään testien menneen onnistuneesti läpi molemmilla selaimilla. Painamalla raportin testitapausta, nähdään myös tarkemmat tiedot vaihe vaiheelta, kuten kuinka paljon aikaa mihinkin vaiheeseen on kulunut (Kuva 17).

Kuva 17 Testiraportti HTML-muodossa



Seuraavaksi testattiin toisen järjestelmän laajaa hakutoimintoa. Laajan haun testaamista varten luotiin ensin järjestelmään testiasia, jota voitiin hyödyntää hakua testatessa. Laajassa haussa on käytössä useita eri hakukriteerejä. Projektissa laajaa hakua testattiin muutamalla eri yhdistelmällä. Seuraavassa esimerkissä laajaa hakua testattiin täyttämällä hakukenttiin asian diaarinumero sekä käsittelijän nimi. Lopuksi tarkastettiin, että "Hakutuloksia: 1"-teksti on näkyvillä sivulla, josta voidaan päätellä haun toimineen onnistuneesti (Kuva 18).

Kuva 18 Laajan haun testaus

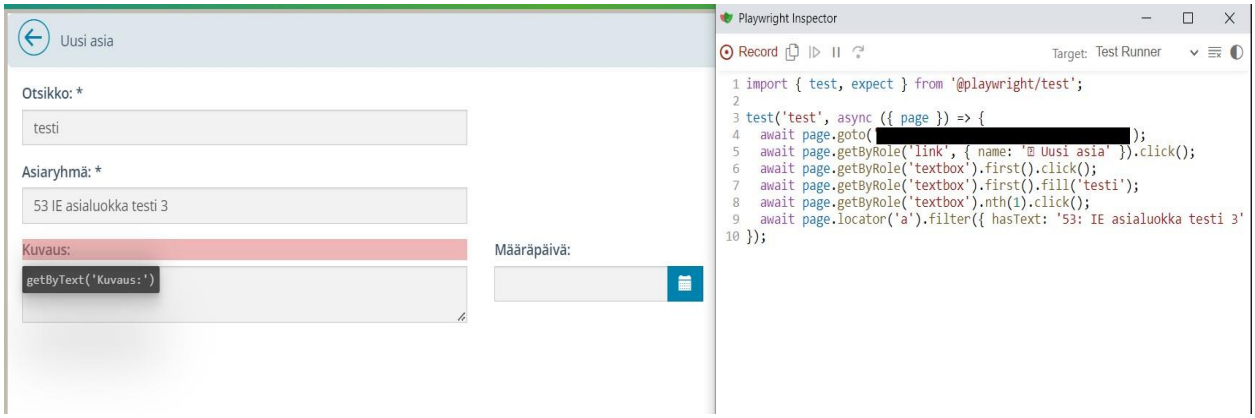
```
test('Laaja haku diaarinumerolla ja käsittelijällä', async ({browser}) => {
  const context = await browser.newContext({

  })
  // paina uusi asia
  const page = await context.newPage();
  await page.goto(URL);
  await page.getByRole('link', { name: 'Laaja haku' }).click();
  await page.locator('div').filter({ hasText: /^Diaarinumero:$/ }).getByRole('textbox').click();
  await page.locator('div').filter({ hasText: /^Diaarinumero:$/ }).getByRole('textbox').fill('V/173/2023');
  await page.getByLabel('Käsittelijä').fill('Nevalainen Jenna');
  await page.getByRole('button', { name: 'Hae'}).nth(1).click();
  await expect(page.getByText('Hakutuloksia: 1')).toBeVisible();
})
```

Osa projektin testeistä luotiin Playwrightin nauhoitustyökalua hyödyntämällä. Nauhoitustyökalun avulla testejä ei tarvitse kirjoittaa itse, vaan Playwright luo koodin perustuen sivulla suoritettaviin toimintoihin. Nauhoitustyökalu avattiin VSCoden terminaalin kautta komennolla. Komentoon määritettiin selain, jolla testi nauhoitetaan sekä testattavan sivuston URL-osoite (Kuva 19).

Kuva 19 Komento nauhoitustyökalun avaamiseen

Kun komento suoritettiin, Playwright avasi selaimen, joka navigoi suoraan testattavan järjestelmän sivulle. Komento voidaan ajaa myös ilman sivun määrittelemistä, jolloin sivu voidaan itse lisätä selaimen kenttään. Selaimen lisäksi oikealle aukesi ikkuna, johon Playwright luo testiskriptin perustuen sivulla tehtyihin toimintoihin (Kuva 20). Seuraavaksi tehtiin sivulla toiminnot, joita haluttiin testata, jonka jälkeen kopioitiin Playwrightin luoma testiskripti. Tämän jälkeen testiskripti liitettiin testitiedostoon, jonka jälkeen se oli valmis suoritettavaksi.

Kuva 20 Testien nauhoittaminen



Testien nauhoitustyökalu toimi lähes poikkeuksetta moitteetta ja kopioidut testiskriptit menivät onnistuneesti läpi. Välillä nauhoitustyökalun luomaan skriptiin jouduttiin tekemään pieniä muutoksia. Esimerkiksi näin kävi yhdessä testissä, jossa nauhoitustyökalu määritti skriptin täyttämään kentän ”fill”-toiminnolla, mutta testiä suorittaessa se ei toiminutkaan, vaan kenttä jäi tyhjäksi. ”Fill”-toiminto tuli vaihtaa itse ”type”-toiminnoksi, jonka jälkeen ohjelma osasi täyttää tekstikentän.

6.5 Projektin haasteet

Kehitysprojektin aikana kohdattiin haasteita johtuen organisaation kovennetusta työympäristöstä, jossa peruskäyttäjällä ei ole kaikkiin toimintoihin oikeuksia. Työasema esimerkiksi esti Playwrightin lataamien selainten avaamisen, eikä niitä voitu käyttää

tekstieditorin kautta. Jos Playwrightin selaimia halusi käyttää, tuli testit suorittaa terminaaliin ylläpitäjäkäyttäjänä. Tässä kohtaa ongelmia seurasi myös testattavien järjestelmien käyttämistä erilaisista tunnistautumistavoista. Yksi järjestelmistä hyödyntää Kerberos-tunnistautumista, joka tunnistaa käyttäjän automaattisesti. Ylläpitäjäkäyttäjällä ei ollut oikeuksia järjestelmään, jonka vuoksi sillä ei päässyt kirjautumaan järjestelmään. Lopulta selvisi, että työasemaympäristö esti selainten käyttämisen niiden sijaintikansion vuoksi. Ongelma saatiin ratkaistua siirtämällä selaimet toiseen kansioon ylläpitäjäoikeuksilla, jonka jälkeen kaikki selaimet ja nauhoitustyökalu toimivat myös tekstieditorin kautta.

Projektin aikana tuli myös ilmi, ettei Playwright tue organisaatiossa käytettävää Firefoxin Enterprise-versiota, joten testejä ei saatu suoritettua Firefoxilla. Ongelman olisi saanut tarvittaessa ratkaistua lataamalla itse Playwrightin tukeman Firefox-version ja lisäämällä sen tiedostopolun Playwrightin konfiguraatiotiedostoon. Tässä tapauksessa kuitenkin päätettiin, että testien suorittaminen Chromella sekä Edgellä riittää.

7 Tulokset

Huolimatta projektin aikana ilmenneistä haasteista opinnäytetyöprosessi eteni aikataulun mukaisesti ja kehitysprojektin tuloksena syntyi toimeksiantajan vaatimuksia vastaava testiautomaatio-ohjelma. Testejä kirjoitettiin kolmea eri järjestelmää varten ja testitiedostot jäivät toimeksiantajan käyttöön. Valmiita testejä voidaan hyödyntää järjestelmien testaamiseen päivitysten yhteydessä. Jatkossa toimeksiantaja voi myös kirjoittaa testejä lisää aina tarvittaessa.

Toimeksiantajalle tehtiin myös kattava dokumentaatio testiautomaatio-ohjelman käytöstä heidän työasemaympäristössään, jotta ohjelmaa voidaan jatkossa ylläpitää. Dokumentaatiossa käydään läpi Playwrightin asennus, testien kirjoittaminen ja suorittaminen sekä Playwrightin päivittäminen. Toimeksiantaja oli tyytyväinen testiautomaatio-ohjelman toteuttamiseen sekä dokumentaatioon. Ohjelmaa aiotaan jatkossa hyödyntää Valviran järjestelmien testaukseen. Tulevaisuudessa testitapauksia myös laajennetaan.

Projektin aikana kirjoitetut testitapaukset laadittiin pitkälti järjestelmäasiantuntijoiden haastattelujen pohjalta. Projektissa automatisoitiin testit niihin ominaisuuksiin, joissa useimmin esiintyy virheitä. Tulevaisuudessa testejä tulee ylläpitää sekä testitapauksia laajentaa vastaamaan mahdollisiin uusiin virhetilanteisiin. Monissa testeissä ominaisuuden toimivuutta testattiin vain yhdellä tai kahdella erilaisella syötteellä. Tulevaisuudessa myös syötteiden määrää voisi laajentaa, jotta testeistä saadaan entistä kattavammat.

Projektiin sopivan testiautomaatiotyökalun valinta onnistui mielestäni hyvin ja Playwright vastasi toimeksiantajan vaatimuksiin. Playwright on helppo asentaa ja testien ylläpitäminen tulisi olla helppoa nauhoitustyökalun avulla. Playwrightista päivitetään säännöllisesti uusia versioita ja sen ominaisuudet sekä yhteisö tulevat hyvin todennäköisesti laajenemaan tulevaisuudessa.

9 Yhteenveto

Tutkimuskysymyksiin vastattiin onnistuneesti opinnäytetyössä. Sopivan testiautomaatiotyökalun valintaa käytiin kattavasti läpi jo teoriaosuudessa. Teoriassa kerrottiin, miksi sopivan testiautomaatiotyökalun valinta on tärkeää sekä mitä valinnassa tulee ottaa erityisesti huomioon. Työkalun valintaprosessia käytiin läpi myös käytännön osuudessa. Testiautomaation toteuttamisen ja ylläpidon mahdollisista haasteista kerrottiin niin teoriassa, kuin käytännössäkin. Testiautomaatioon siirtymisen hyödyt käytiin läpi kattavasti teoriaosuudessa.

Testiautomaation siirtymisen päätavoitteena oli vähentää toimeksiantajan järjestelmissä esiintyviä virhetilanteita ja parantaa niiden laatua kokonaisuudessaan. Testiautomaatio-ohjelma tulee olla kuitenkin pidempään käytössä, jotta voidaan tehdä johtopäätöksiä ohjelmiston laadun paranemisesta. Opinnäytetyön aikanakin ilmi tulleiden tietojen perusteella uskon kuitenkin järjestelmien laadun parantuvan pidemmällä aikavälillä, jos automatisoituja testejä suoritetaan ja ylläpidetään säännöllisesti. Työntekijöiden aikaa tulee myös selvästi säästymään, kun manuaalisen testauksen tarve vähenee.

Ennen opinnäytetyötä olin tutustunut opinnoissani testiautomaation ja ohjelmistotestauksen perusteisiin. Opinnäytetyön myötä opin paljon uutta testauksesta, niin teoriassa kuin käytännössäkin. Työn myötä itselleni konkretisoitui erityisesti se, mitä asioita testiautomaatioon siirtymisessä tulee ottaa huomioon sekä omiin tarpeisiin sopivan testiautomaatiotyökalun valinnan tärkeys. Opin myös itselleni täysin uuden testiautomaatiotyökalun sekä TypeScript-ohjelmointikieltä, jota en ollut aikaisemmin käyttänyt. Projektin myötä kiinnostukseni aiheisiin on kasvanut ja toivon pääseväni tulevaisuudessa oppimaan lisää ohjelmistotestauksesta käytännössä.

Lähteet

Baresi, L., & Pézze, M. (2006). An Introduction to Software Testing. https://www.researchgate.net/publication/222672291_An_Introduction_to_Software_Testing

Baumgartner, Manfred, et al. Test Automation Fundamentals: A Study Guide for the Certified Test Automation Engineer Exam – Advanced Level Specialist – ISTQB® Compliant. 1. edition, dpunkt.verlag, 2022. (E-kirja)

Bhaskar, S. (2023). What is kanban? An overview of the kanban method. https://www.nimblework.com/kanban/what-is-kanban/

Bergs, E. (2023). What is smoke testing and why is it important? TestDevLab Blog. https://testdevlab.com/blog/what-is-smoke-testing-and-why-is-it-important

Bierig, R., Brown, S., Edgar, G., & Timoney, J. (2022). Essentials of software testing. Cambridge University Press.

Boby, J. (2021). Test Automation: A manager's guide. BCS Learning & Development Limited.

Flewelling, P. (2018). The the agile developer's handbook: Get more value from your software development: Get the best out of the agile methodology. Packt Publishing, Limited.

Gillis, A. (2021). What is acceptance testing? Definition from SearchSoftwareQuality. Software Quality. https://www.techtarget.com/searchsoftwarequality/definition/acceptance-test

Herranen, K. (2020). Ketterä kasvu. Alma Talent.

Homès, B. (2012). Fundamentals of software testing. ISTE/Wiley.

Juvonen, R. (2018). Ohjelmistoprojektin sudenkuopat ja miten ne vältetään.

Kasurinen, J. (2013). Ohjelmistotestauksen käsikirja. Docendo.

Layton, M. C., & Ostermiller, S. J. (2020). Agile project management for dummies 3e: (3rd ed.). John Wiley and sons.

Mahajan, A. (2023). Selenium vs Testcomplete: What are the differences? https://www.knowledgehut.com/blog/software-testing/selenium-vs-testcomplete

Measy, P. & Radtac. (2015). Agile foundations: Principles, practices and framework. BCS Learning & Development.

Mili, A. (2015). Software testing: Concepts and operations. John Wiley & Sons Inc.

Myers, G. J., Sandler, C., & Badgett, T. (2012). The art of software testing (3.painos). John Wiley & Sons.

Nehytailo, Y. (2023). Puppeteer vs selenium: Which to choose. https://oxylabs.io/blog/puppeteer-vs-selenium

Neha, Vaidya. (2023). Puppeteer vs selenium: Core differences. BrowserStack. https://browserstack.wpengine.com/guide/puppeteer-vs-selenium/

Pathak, K. (2023). Cypress vs playwright with pros and cons. The Talent500 Blog. https://talent500.co/blog/cypress-vs-playwright-with-pros-and-cons/

Rana, K. (2019). What is automation testing? Types, tools, process & benefits. ArtOfTesting. https://artoftesting.com/automation-testing

Rehn, E. (2023). Mitä on testaus? Ohjelmistotestaus laadunvarmistajana. Gofore. https://gofore.com/blog/2023/05/30/mita-on-testaus-ohjelmistotestaus-laadunvarmistajana/

Santo, D. E. (2022). Top 5 main Agile methodologies: Advantages and disadvantages. Xpand IT. https://www.xpand-it.com/blog/top-5-agile-methodologies/

Siderova, S. (2018). The top 10 benefits of kanban | nave. Nave Blog - Expert Tips and Guidelines for Kanban Teams. https://getnave.com/blog/kanban-benefits/

Testiautomaatio | Ite wikin digitalisoinnin opas. (2019). https://www.itewiki.fi/opas/testiautomaatio/

Waldén, J. (2020). Innovaation edistäminen ketterässä ohjelmistokehityksessä. [Kandityö, LUT yliopisto)

What is scrum? | the agile journey with pm-partners. PM Partners. https://www.pm-partners.com.au/the-agile-journey-a-scrum-overview/

Liite 1: Aineistonhallintasuunnitelma

Opinnäytetyön materiaali on hankittu pääasiassa teoriaosuudessa käytetyistä lähteistä sekä käytännönosuuden myötä syntyneestä tiedosta. Tietoja on tallennettu prosessin aikana päiväkirjamaisesti koulun henkilökohtaiseen OneDrive-pilvipalveluun sekä tietokoneeni C-asemalle. Opinnäytetyössä on myös hyödynnetty toimeksiantajayrityksen sisäistä materiaalia, joka säilytetään toimeksiantajan sisäisessä työtilapalvelussa. Tietokoneellani on vahva salasana ja ajantasainen virussuoja, joiden avulla huolehditaan materiaalin tietoturvasta. Opinnäytetyön aineistoa ei jatkokäytetä. Säilytän materiaaleja tietokoneellani ja pilvipalvelussa vuoden ajan opinnäytetyön valmistumisen jälkeen. Tämän jälkeen materiaalit hävitetään asianmukaisesti. Projektin aikana kehitetyt testitiedostot jäävät toimeksiantajan sisäiseen käyttöön. Opinnäytetyön valmistuttua se julkaistaan Theseus-palvelussa.