Tampereen yliopisto

Sami Mäkäräinen

FUNKTIONAALISEN TESTAUKSEN KEHITTÄMINEN WEB-SOVELLUKSELLE ROBOT FRAMEWORKILLA

Diplomityö
Tekniikan ja luonnontieteiden tiedekunta
Tarkastajat: yliopisto-opettaja Mikko Salmenperä
automaation tietoturva-asiantuntija Jari Seppälä
Marraskuu 2024

TIIVISTELMÄ

Sami Mäkäräinen: Funktionaalisen testauksen kehittäminen web-sovellukselle Robot Frameworkilla
Diplomityö
Tampereen yliopisto
Automaatiotekniikan DI-ohjelma
Marraskuu 2024

Testaus on tärkeä osa mitä tahansa ohjelmistoprojektia. Testauksen avulla saadaan tietoa ohjelman toimintavarmuudesta ja löydetään vikoja ennen kuin ne ehtivät aiheuttaa suurempia kustannuksia. Tavanomainen manuaalinen testaus on kuitenkin aikaa vievää ja siksi tähän on noussut ratkaisuksi testiautomaatio. Testiautomaatio mahdollistaa ohjelmiston laadunvarmistuksen modernissa ketterässä ohjelmistokehityksessä. Testiautomaatiota varten on kehitetty monia työkaluja, joista yksi on web-sovellustestaukseen tehty Robot Framework. Tässä diplomityössä tutkitaan keinoja kehittää testiautomaatiota käyttöliittymätasolla käyttäen Robot Frameworkia ja sen Browser Libraryä.

Työn tavoitteena on tutkimuksen avulla parantaa aiempaa SeleniumLibraryllä toteutettua web-sovelluksen käyttöliittymätestausta. Tähän tavoitteeseen pääsemiseksi teoriaosissa tutustutaan ohjelmistotestaukseen erityisesti ketterän kehityksen näkökulmasta. Lisäksi selvitetään, mikä tekee testauksesta laadukkaan. Selvinneistä asioista muodostuu työn tarkempi tavoite: luotettavuuden, ylläpidettävyyden ja nopeuden parantaminen. Työssä käytettäviä työkaluja, Robot Framework Browser -kirjastoa ja Page Object Model -suunnittelumallia, esitellään ja tutkitaan tarkemmin, jotta niiden hyödyntäminen olisi mahdollisimman tehokasta toteutusvaiheessa. Toteutusvaiheessa keskitytään testiautomaation kirjoittamiseen ja käyttöönottoon liittyviin vaiheisiin.

Työn lopussa käsitellään tuloksia ja arvioidaan ratkaisua. Tuloksissa todettiin, että parannuksia saatiin kaikissa testauksen kannalta kriittisissä osa-alueissa. Luotettavuutta parannettiin kattavuutta lisäämällä. Ylläpidettävyys koodin Page Object Model -mallin avulla ja tehokkailla avainsanoilla. Nopeuteen saatiin parannuksia vaihtamalla SeleniumLibrary Browser-kirjastoon ja poistamalla turhia testivaiheita.

Avainsanat: Robot Framework, testiautomaatio, web-sovellus

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

ABSTRACT

Sami Mäkäräinen: Improving web application functional testing with Robot Framework
Masters thesis
Tampere University
Masters Programme in Engineering Sciences, MSc (Tech)
November 2024

Testing is an essential part of any software project. Testing provides information about the reliability of the program and identifies defects before they can lead to larger costs. However, traditional manual testing is time-consuming, and thus test automation has emerged as a solution. Test automation enables software quality assurance in modern agile software development. Many tools have been developed for test automation, one of which is Robot Framework, designed for testing web applications. This thesis explores ways to develop test automation at the user interface level using Robot Framework and its Browser library.

The goal of this thesis is to improve the previous SeleniumLibrary implementation of user interface testing of the web application. To achieve this goal, the theory sections will examine software testing, particularly from the perspective of agile development. Additionally, it will investigate what makes testing high-quality. From the findings, the specific objectives of the work will be formed: improving reliability, maintainability, and speed. The tools used in the work, the Robot Framework Browser Library, and the Page Object Model design pattern are introduced and studied in more detail to enable their most effective utilization during the implementation phase. The implementation phase will focus on the steps related to writing and deployment of test automation.

The work concludes with an analysis of the results and an evaluation of the solution. The results indicated improvements in all critical aspects of testing. Reliability was enhanced by increasing coverage. Maintainability improved through the use of the Page Object Model and efficient keywords. Speed was improved by switching from SeleniumLibrary to Browser Library and by eliminating unnecessary test steps.

Keywords: Robot Framework, test automation, web application

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

Opinnäytteessäni käytetyt tekoälytyökalut ja niiden käyttötarkoitukset on kuvailtu alla:

Työkalun nimi (ja versio): ChatGPT 4o

Käyttötarkoitus ja osio, jossa työkalua käytettiin:

- Joidenkin epäformaalisti esitettyjen ajatusten tai kappaleiden selkeyttämiseen ja täydentämiseen järkeviksi, diplomityöhön sopiviksi lauseiksi ja kappaleiksi.
- Itse toteutusosioissa testeihin tarvittavan koodin kirjoitusapuna.
- Luvun 6.1 kuvien tuottamiseen antamani kaavan perusteella.
- Listauksien ja taulukoiden muuntamiseen LaTeX-muotoon.

Olen tietoinen siitä, että olen täysin vastuussa koko opinnäytteeni sisällöstä, mukaan lukien tekoälyllä tuotetut osat, ja hyväksyn vastuun mahdollisista eettisten ohjeiden rikkomuksista.

SISÄLLYSLUETTELO

1. Johdanto . . . 1
2. Ohjelmistotestaus . . . 3
   2.1 Testauksen merkitys ja testiautomaation kannattavuus . . . 3
   2.2 Funktionaalinen ja ei-funktionaalinen ohjelmistotestaus . . . 6
   2.3 Ohjelmistotestauksen eri tasot . . . 8
   2.4 Testausprosessi ketterissä ohjelmistoprojekteissa . . . 9
   2.5 Ohjelmistokehityksen jatkuva integraatio . . . 11
3. Laadukkaan testauksen ominaisuudet . . . 15
   3.1 Testauksen luotettavuus ja robustisuus . . . 15
   3.2 Testien ymmärrettävyys ja ylläpidettävyys . . . 17
   3.3 Testien kattavuus . . . 19
4. Työkalujen esittely . . . 20
   4.1 Robot Framework testiautomaatiotyökaluna . . . 20
   4.2 Robot Frameworkin SeleniumLibrary . . . 21
   4.3 Robot Frameworkin Browser-kirjasto . . . 22
   4.4 Robot frameworkin yleiset käytänteet ja syntaksi . . . 24
   4.5 Page Object Model -suunnittelumalli . . . 26
5. Testauksen toteutuksen suunnittelu ja toteutus . . . 29
   5.1 Testattavan ympäristön esittely . . . 29
   5.2 Ympäristön tuomat rajoitukset ja reunaehdot . . . 31
   5.3 Testauksen lopputuloksen tavoiteltava taso . . . 32
   5.4 Testausprojektin tiedostorakenne . . . 33
   5.5 Testien suunnittelu ja kirjoitus . . . 33
   5.6 Ympäristön ja putken konfigurointi . . . 37
   5.7 Haasteet toteutuksessa . . . 38
   5.8 Automaatiotestaus osana ohjelmistokehitystä . . . 39
6. Ratkaisun arviointi . . . 41
   6.1 Testauksen luotettavuuden mittarit . . . 41
   6.2 Testiautomaation ylläpidettävyys ja yksinkertaisuus . . . 44
   6.3 Testien suoritusnopeuden ja tehokkuuden arviointi . . . 47
   6.4 Muiden näkökulmien pohdintaa . . . 48
7. Jatkokehitysmahdollisuudet . . . 50
8. Yhteenveto . . . 52

Lähteet 53

LYHENTEET JA MERKINNÄT

Avainsana	Toiminnallisuus abstraktoitu yhteen määriteltyyn komento- tai käskykokonaisuuteen, jota voidaan käyttää uudelleen useissa testeissä.
Avainsanavetoinen	Testauksen tyyli, jossa toiminnallisuudet ja testit kuvataan avainsanoina (engl. keyword-driven), mikä yksinkertaistaa testitapauksia.
CD	Continuous Delivery (jatkuva uuden ohjelmiston toimitus asiakkaalle)
CI	Continuous Integration (jatkuva koodin integraatio osaksi ohjelmistoa)
Datavetoinen	Testauksen tyyli, jossa testitapaukset suoritetaan eri datasyötteillä (engl. data-driven), jolloin samaa logiikkaa testataan eri syöteyhdistelmillä.
HTML	Hypertext Markup Language (hypertekstin merkintäkieli)
Import-suhteet	Ohjelmakoodin tai testien rakenne, jossa kooditiedostoja tuodaan käyttöön toisista tiedostoista. Tämä auttaa jakamaan koodin modulaarisesti.
Ketterä kehitys	Ohjelmistokehitysmenetelmä, jossa projektia kehitetään iteratiivisesti pienissä osissa ja yhteistyössä eri sidosryhmien kanssa.
POM	Page Object Model (ohjelmistotestauksessa käytetty suunnittelumalli)
Regressiotestaus	Testaus, jonka tavoitteena on varmistaa, että uudet muutokset eivät riko aiemmin toimivia osia ohjelmistossa.
Scrum master	Ketterän kehitystiimin vetäjä, joka vastaa prosessin sujuvuudesta ja tiimin toiminnasta.
Staging-ympäristö	Kehitysympäristö, joka jäljittelee tuotantoympäristöä, ja jossa sovellusta testataan ennen tuotantoon viemistä.
TDD	Test Driven Development (testaukseen pohjautuva ohjelmistokehitys, jossa testit kirjoitetaan ennen muuta koodia)
UI	User Interface (käyttöliittymä)

1. JOHDANTO

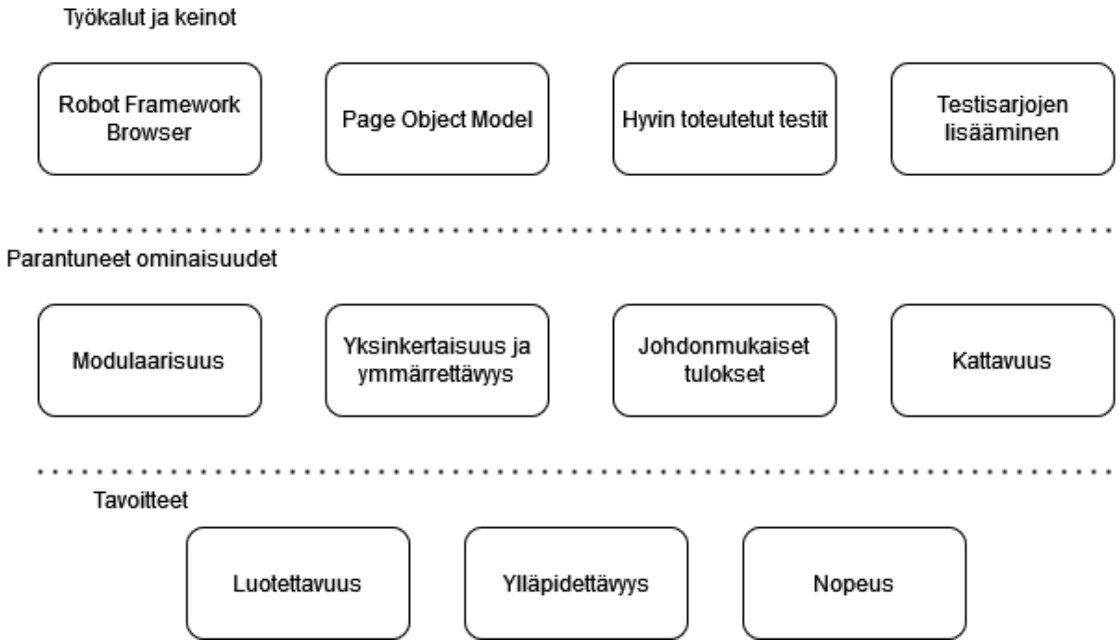
Ohjelmistokehityksen parissa kohtaamme jatkuvasti haasteita, kun ohjelmat kasvavat monimutkaisemmiksi ja niiden koodimäärä lisääntyy huimaa vauhtia [26]. Tämä tuo mukanaan kasvavan tarpeen varmistaa ohjelmiston laadukas toiminta, mikä vaatii kattavaa ja tehokasta testausta. Perinteinen manuaalinen testaus on kuitenkin usein aikaa vievää ja resursseja kuluttavaa, mikä luo haasteita erityisesti ketterälle ohjelmistokehitykselle, jossa nopeat muutokset ja tiheät julkaisut ovat arkipäivää. Tästä syystä keskeisenä tavoitteena on löytää keinoja vähentää testaukseen kuluvaa aikaa, samalla parantaen testien luotettavuutta ja kattavuutta.

Työn keskeinen tutkimuskysymys on selvittää tapoja kehittää web-sovellusten testiautomaation nopeutta, ylläpidettävyyttä sekä luotettavuutta. Samalla työssä tutkitaan Robot Frameworkin Selenium- ja Browser-kirjastojen välisiä eroja, joiden avulla voidaan arvioida, kuinka uudemmat testaustyökalut voivat parantaa testausprosessia. Erityisesti tutkitaan, kuinka testitapausten rakennuspalikoita eli avainsanoja voidaan kehittää parempien testien ominaisuuksien tavoittelussa. Näiden kehityskohteiden avulla pyritään ratkaisemaan monia ohjelmistotestaukseen liittyviä ongelmia, kuten pitkät testausajat ja testien haavoittuvuus ohjelmistomuutoksille.

Tässä työssä tavoitteena on toteuttaa tehokas testausautomaation käyttöönotto, jossa keskitytään testauksen nopeuden, ylläpidettävyyden ja luotettavuuden parantamiseen hyödyntämällä uusia teknologioita. Työssä hyödynnetään erityisesti Robot Frameworkin uudehkoa Browser-kirjastoa sekä Page Object Model -suunnittelumallia, joiden tarkoituksena on tuoda rakennetta ja skaalautuvuutta testien hallintaan. Tämän testausprosessin kohteena oleva web-sovellus on laskutusjärjestelmä, mutta työn tulokset ovat sovellettavissa myös muihin järjestelmiin, erityisesti päästä päähän -skenaarioiden web-sovellustestaukseen. Näiden työkalujen avulla pyritään rakentamaan testejä, jotka ovat robusteja ja helposti ylläpidettäviä, samalla kun ne antavat johdonmukaisia ja luotettavia tuloksia.

Luotettavuuden parantamiseksi työssä keskitytään kattavien ja robustien testien luomiseen, jotka antavat luotettavaa palautetta sovelluksen toiminnasta. Erityisesti tarkastellaan, miten nämä automaatiotestit voidaan integroida tehokkaasti osaksi ketterää ohjelmistokehitysprosessia, jossa iteratiiviset muutokset vaativat jatkuvaa laadunvarmistusta.

Työn tavoite ja siihen pääseminen on visualisoitu kuvassa 1.1.



Kuva 1.1. *Työssä käytettävät keinot, parantuneet ominaisuudet ja tavoitteet*

Diplomityö etenee seuraavasti: Luvussa 2 perehdytään testauksen teoriaan, erityisesti keskittyen testiautomaatioon. Luvussa 3 tutustutaan testauksen aihealueisiin, jotka ovat kriittisiä työn tavoitteiden kannalta. Luvussa 4 tarkastellaan työhön annettuja testausautomaation työkaluja ja luvussa 5 toteutetaan niillä sovellukselle korkean tason testaus. Sen jälkeen luvussa 6 arvioidaan ratkaisun tehokkuutta testien luotettavuuden, ylläpidettävyyden ja suoritusnopeuden näkökulmasta. Luvussa 7 pohditaan vielä, millä keinoilla toteutusta voisi edelleen parantaa, mutta joita ei ole toteutettu tässä työssä.

Työn lopputulokselta odotetaan parantunutta luotettavuutta, ylläpidettävyyttä, ja nopeutta. Robot Frameworkin Browser-kirjasto on uudempi versio korvattavalle SeleniumLibrarylle, missä on panostettu erityisesti luotettavuuteen ja nopeuteen uudenlaisilla ratkaisulla. Lisäksi Page Object Model -mallia käyttämällä pyritään tuomaan tiedostoille selkeää ja helposti ylläpidettävää rakennetta, mutta toisaalta sen käytännöllisyyttä päästä päähän -tyyppisissä testeissä on syytä epäillä.

2. OHJELMISTOTESTAUS

Ohjelmistotestaus on keskeinen osa ohjelmistokehitystä, ja sen avulla varmistetaan, että sovellus toimii oikein käyttäjien odotusten ja määritettyjen vaatimusten mukaisesti. Erityisesti funktionaalinen testaus on tärkeää, sillä sen avulla varmistetaan sovelluksen toiminnallisuuksien oikeellisuus. Funktionaalisessa testauksessa tarkastellaan, miten hyvin sovelluksen eri toiminnot, kuten kirjautuminen, tietojen syöttäminen ja haku, sekä käyttöliittymän vuorovaikutukset toimivat suunnitellusti. Web-sovelluksissa, jotka koostuvat monista käyttöliittymäkomponenteista ja taustajärjestelmistä, funktionaalinen testaus on välttämätöntä sen varmistamiseksi, että eri osat toimivat saumattomasti yhteen ja että käyttäjä voi suorittaa halutut tehtävät ilman virheitä. Koska web-sovellukset ovat dynaamisia ja toimivat monissa eri ympäristöissä, funktionaalisen testauksen merkitys korostuu ketterissä projekteissa, joissa sovelluksen tulee toimia saumattomasti jatkuvien muutosten ja päivitysten keskellä. Tämä luku tarkastelee ohjelmistotestauksen perusperiaatteita erityisesti web-sovellusten näkökulmasta.

Tässä luvussa tarkastellaan ohjelmistotestauksen perusteita ketterän kehityksen näkökulmasta, keskittyen erityisesti web-sovelluksiin. Alaluvuissa käsitellään ensin testauksen merkitystä ja miksi tai milloin testiautomaatio on kannattavaa. Sitten tarkastellaan funktionaalisen ja ei-funktionaalisen testauksen eroja, painottaen erityisesti testauksen roolia toiminnallisuuksien ja suorituskyvyn varmistamisessa. Lisäksi käsitellään testauksen eri tasoja, kuten yksikkö-, integraatio-, järjestelmä- ja hyväksyntätestausta, ja näiden tasojen merkitystä sovelluksen laadun varmistamisessa. Testausprosessin integrointi ketteriin projekteihin on ratkaisevaa, sillä se mahdollistaa jatkuvan kehityksen, säännölliset päivitykset ja nopean reagoinnin muutoksiin. Testiautomaation merkitys korostuu tässä yhteydessä, sillä se mahdollistaa toistuvien testisyklien ajamisen tehokkaasti ja vähentää inhimillisten virheiden riskiä. Näiden teemojen kautta luvussa tarkastellaan, kuinka testaus edistää laadunvarmistusta ja nopeuttaa web-sovellusten kehitystä ketterissä ympäristöissä.

2.1 Testauksen merkitys ja testiautomaation kannattavuus

Testaus on olennainen osa ohjelmistokehitystä, sillä se varmistaa ohjelmiston toimivuuden, laadun ja luotettavuuden ennen sen tuotantoon viemistä. Ilman riittävää testausta

ohjelmisto voi sisältää kriittisiä virheitä, jotka vaikuttavat sen käytettävyyteen, suorituskykyyn tai tietoturvaan. Testausprosessin avulla voidaan tunnistaa sekä pienet että suuret virheet, jotka voivat vaikuttaa käyttäjäkokemukseen tai liiketoimintatavoitteiden saavuttamiseen. Tämän vuoksi testauksen merkitys ulottuu paljon laajemmalle kuin pelkkään virheiden löytämiseen — se on keskeinen osa ohjelmiston laadunvarmistusta ja riskien hallintaa.

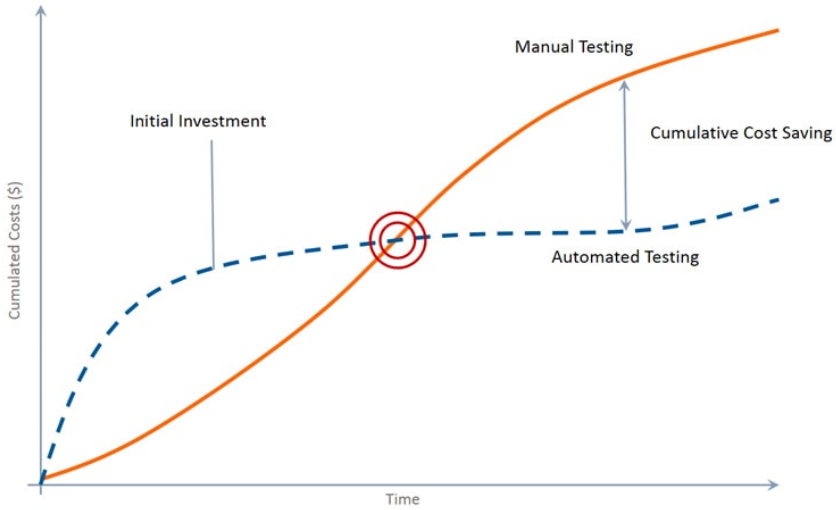
Manuaalinen testaus on perinteinen tapa varmistaa ohjelmiston toimivuus, ja se on erityisen hyödyllinen uusien ominaisuuksien, käyttöliittymien ja sovellusten varhaisessa kehitysvaiheessa. Manuaalisessa testauksessa testaaja suorittaa jokaisen testitapauksen käsin, käy läpi järjestelmän toiminnot ja vertaa tuloksia odotettuihin lopputuloksiin. Tämä menetelmä antaa mahdollisuuden joustavasti reagoida sovelluksen toimintaan, mutta on aikaa vievää ja altis inhimillisille virheille. Lisäksi, kun sovellus kehittyy ja kasvaa, manuaalinen testaus voi käydä työlääksi ja resursseja vaativaksi. Tämä korostuu erityisesti moderneissa ohjelmistoprojekteissa, joihin tulee jatkuvia muutoksia ja testitapauksia tulisi toistaa usein.

Tässä vaiheessa testiautomaatio tulee merkittäväksi osaksi testausprosessia. Testiautomaatio nopeuttaa testauksen toistuvia vaiheita ja varmistaa, että kaikki sovelluksen osa-alueet testataan jatkuvasti ja tarkasti ilman manuaalista työtä. Sen avulla voidaan suorittaa toistuvia testejä nopeammin kuin manuaalisesti, mikä vähentää testausjaksojen pituutta ja mahdollistaa tiheämmät julkaisusyklit [15]. Automaatio on erityisen hyödyllinen regressiotestauksessa, jossa pyritään varmistamaan, että uudet muutokset eivät riko olemassa olevia toimintoja. Testiautomaatio siis mahdollistaa nopeamman palautteen kehitystiimille ja varmistaa, että sovelluksen laatu pysyy korkeana myös jatkuvan muutoksen ja päivitysten aikana.

Testiautomaation käyttöönotto vaatii kuitenkin investointeja, aikaa ja resursseja. Automatisointi vaatii huolellista suunnittelua ja ylläpitoa, eikä kaikkea testausta voida automatisoida. Onkin tärkeää arvioida, missä vaiheessa ja millä osa-alueilla automaatio on kannattavaa. Projektin laajuus, monimutkaisuus ja kehitystiimin osaaminen määrittelevät pitkälti sen, missä määrin testiautomaatio tuottaa hyötyä verrattuna manuaaliseen testaukseen. Hyvin suunniteltu ja toteutettu testiautomaatio tuo kuitenkin pitkällä aikavälillä merkittäviä säästöjä ajassa, resursseissa ja laadunvarmistuksessa.

Testiautomaation käyttöä tulee harkita, kun sen hyödyt ylittävät sen käyttöönoton kustannukset. Tätä suhdetta esittää kuva 2.1. Erityisesti, kun projektissa odotetaan toistuvia ja laajoja regressiotestejä, testiautomaatio voi tarjota merkittäviä säästöjä ajan ja vaivan suhteen. Testiautomaatiosta on eniten hyötyä, kun sitä käytetään ohjelman kriittisten kohteiden testaamiseen, koska näiden rikkoontuminen voi aiheuttaa pahimpia seurauksia, jolloin niitä ei tarvitse aina manuaalisesti testata.

Testiautomaatio tuo mukanaan merkittäviä etuja ohjelmistokehityksen laadunvarmistuk-



Kuva 2.1. *Testiautomaation kannattavuutta kuvaava käyrä [14]*

sessa, erityisesti suurissa ja toistuvia testauksia vaativissa projekteissa. Taulukossa 2.1 on esitetty testiautomaation keskeiset hyödyt ja rajoitukset. Yksi merkittävimmistä eduista on parantunut tuotteen laatu. Kun testit automatisoidaan, voidaan varmistaa, että ohjelmistoa testataan laajemmin ja tehokkaammin kuin manuaalisilla testeillä [15]. Tämä mahdollistaa kattavan testauksen, joka voi kattaa suuren osan koodista (esimerkiksi haarautumat ja ehdot) ilman lisätyötä ihmisiltä.

Toinen merkittävä hyöty on luotettavuus. Koska automaattiset testit suoritetaan samalla tavalla joka kerta, ihmisten tekemät virheet eivät vaikuta tuloksiin. Tämä tuo vakautta ja luottamusta kehitystiimille, sillä testitulokset ovat yhdenmukaisia. Lisäksi testien uudelleenkäytettävyys ja mahdollisuus säästää kustannuksia pitkällä aikavälillä ovat merkittäviä etuja, kun automaatiota käytetään laajasti [15].

Toisaalta, kuten taulukosta 2.1 käy ilmi, testiautomaatiolla on myös rajoituksensa. Kaikkea testausta ei voida automatisoida, ja manuaaliset testit ovat edelleen tarpeen monimutkaisten ja intuitiivisten ongelmien löytämiseksi. Automaation käyttöönotto vaatii aluksi paljon resursseja: aikaa, osaamista ja rahaa. Erityisesti testausprosessin kehittäminen, testitapauksien valinta ja automaatiojärjestelmien ylläpito voivat vaatia huomattavia panostuksia, jotka eivät välttämättä ole helposti saavutettavissa pienille tiimeille [15].

Lisäksi väärät odotukset automaation suhteen voivat johtaa pettymyksiin. Usein kuvitellaan, että automaatio ratkaisee kaikki ongelmat, mutta se vaatii silti huolellista suunnittelua ja jatkuvaa ylläpitoa, jotta siitä saadaan täysi hyöty irti. On myös havaittu, että testiautomaatio voi olla vaikeasti hallittavissa, mikäli tiimiltä puuttuvat tarvittavat taidot ja ymmärrys työkaluista ja tekniikoista [15].

Yhteenvetona voidaan todeta, että testiautomaatio tarjoaa lukuisia etuja, kuten parantuneen tuotelaadun, testausajan vähenemisen ja testikattavuuden parantumisen. Samalla

Hyödyt	Rajoitukset
Parantunut tuotteen laatu: Vähemmän virheitä ohjelmistossa	Automaatiota ei voi täysin korvata manuaalisella testauksella
Testauksen kattavuus: Laaja koodikattavuus automaation avulla	Epäonnistuminen tavoitteiden saavuttamisessa: Organisaatioilla on vaikeuksia saavuttaa automaation tavoitteet
Vähentynyt testausaika: Nopeampi testisyklien suoritus	Testiautomaation ylläpidon vaikeudet: Muutokset teknologioissa lisäävät ylläpitotyötä
Luotettavuus: Automaatio on luotettavampi toistettaessa testejä	Testausprosessin kehittäminen vie aikaa
Itseluottamuksen kasvu: Kehittäjien luottamus järjestelmän laatuun kasvaa	Väärät odotukset: Organisaatiot odottavat säästöjä, mutta kohtaavat tuotantohäiriöitä
Testien uudelleenkäytettävyys: Hyvä toistuvuus tuo lisäarvoa	Sopimaton testiautomaatio-strategia: Huonot päätökset automaation tavoitteiden suhteen
Vähemmän ihmistyötä: Vähemmän ihmistyötä tarvitaan automaation ansiosta	Taidolliset puutteet: Testiautomaatio vaatii erityistä osaamista ja työkaluja
Kustannusten vähentäminen: Automaation avulla kustannuksia säästetään	
Parantunut virheiden havaitseminen: Parempi kyky havaita virheitä järjestelmässä	

Taulukko 2.1. *Testiautomaation hyödyt ja rajoitukset [15]*

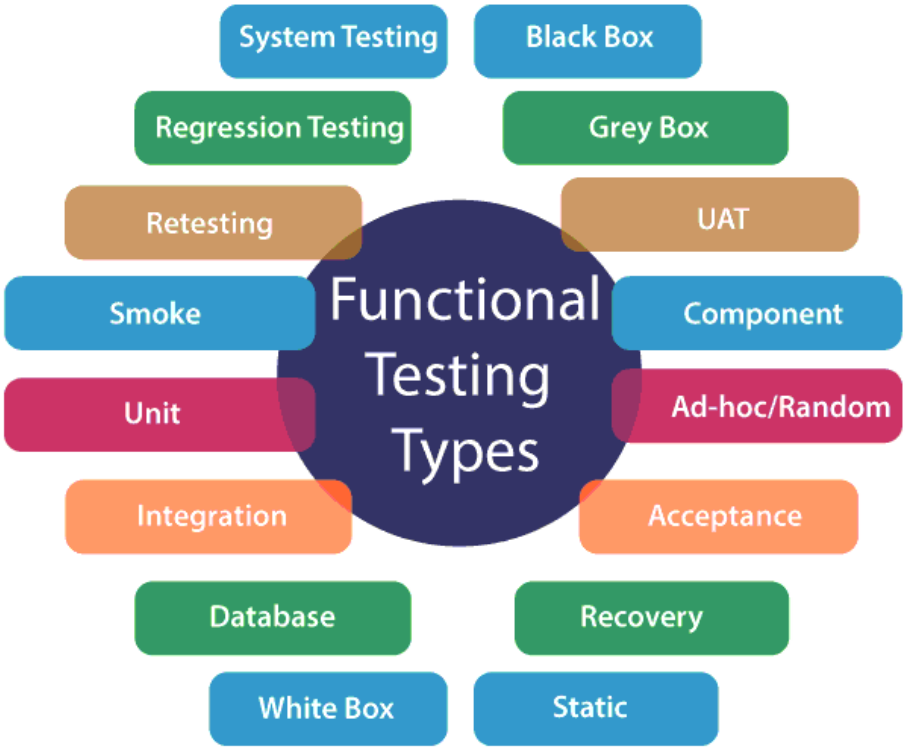
se vähentää manuaalisen työn tarvetta ja tarjoaa mahdollisuuden merkittäviin kustannussäästöihin pitkällä aikavälillä. Kuitenkin testiautomaation käyttöönottoon liittyy myös haasteita, kuten korkeat alkuinvestoinnit ja jatkuva ylläpito, jotka voivat rajoittaa sen tehokkuutta ja soveltuvuutta joissakin projekteissa. Testiautomaatio toimii parhaiten, kun se on suunniteltu huolellisesti ja sitä käytetään sopivissa tilanteissa, joissa hyödyt ylittävät selvästi käyttöönoton kustannukset ja rajoitukset.

2.2 Funktionaalinen ja ei-funktionaalinen ohjelmistotestaus

Funktionaalinen testaus on testausta, joka testaa ohjelman toimintoja. Vastaesimerkkinä suorituskyky- tai tietoturvatestaus eivät kuulu funktionaaliseen testaukseen.

Funktionaalinen testaus on prosessi, jossa ohjelmiston eri toiminnot tarkistetaan suunnitelmien ja vaatimusten mukaisiksi. Se keskittyy siihen, mitä järjestelmä tekee — sisältäen käyttöliittymän testauksen, liiketoimintalogiikan tarkastelun, tietokantayhteyksien validoinnin sekä tietoturvan ja käytettävyyden arvioinnin. Tämän tason testaus on olennainen var-

mistettaessa, että ohjelmisto vastaa liiketoiminnan tarpeisiin ja käyttäjien odotuksiin [3].



Kuva 2.2. *Funktionaalisen testauksen osa-alueet [11]*

Toisaalta ei-funktionaalinen testaus arvioi ohjelmiston attribuutteja, jotka eivät liity suoraan sen toiminnallisiin ominaisuuksiin. Tämä sisältää testauksen, joka keskittyy ohjelmiston suorituskykyyn, skaalautuvuuteen, turvallisuuteen ja käytettävyyteen. Tavoitteena on varmistaa, että ohjelmisto ei ainoastaan tee oikeita asioita. Sen tulee tehdä ne hyvin tarjoten käyttäjille miellyttävän ja turvallisen kokemuksen. Ei-funktionaalinen testaus on yhtä tärkeää kuin funktionaalinen testaus, sillä se auttaa varmistamaan ohjelmiston laadun ja luotettavuuden kaikilla tasoilla [3].

Esimerkkinä funktionaalisesta testistä voidaan mainita "Lisää ostoskoriin-toiminnon tarkistaminen verkkokaupassa, kun taas ei-funktionaalisena testinä voidaan suorittaa kuormitustesti. Tällöin nähdään, miten sama verkkokauppa suoriutuu suuren käyttäjämäärän aikana, kuten Black Friday -alennusmyynteinä. Molemmat testityypit ovat keskeisiä ohjelmistokehityksen eri vaiheissa. Täten varmistetaan, että ohjelmistotuote ei ainoastaan täytä kaikkia sen toiminnallisia vaatimuksia, vaan on myös suorituskykyinen, turvallinen ja miellyttävä käyttää [3].

Tämä diplomityö keskittyy pääosin saatujen hyötyjen arviointiin funktionaalisessa testauksessa järjestelmä- ja hyväksyntätasolla. Tässä työssä ei siis keskitytä esimerkiksi käyttäjäkokemuksen testaukseen, koska se ei kuulu automaatiotestaukseen lukuunottamatta suorituskykytestausta (eng. performance testing).

Kuva 2.3. *Ei-funktionaalisen testauksen tyypit. Lähde: [25]*

2.3 Ohjelmistotestauksen eri tasot

Ohjelmistotestaus on olennainen osa ohjelmistokehitystä, ja se jakautuu eri tasoihin, joista kukin keskittyy erilaisiin testauksen näkökohtiin. Tämä jaottelu mahdollistaa ohjelmiston perusteellisen tarkastelun sekä virheiden tunnistamisen ja korjaamisen mahdollisimman varhaisessa vaiheessa. Ohjelmistotestauksen tasot ovat yksikkötestaus, integraatiotestaus, järjestelmätestaus ja hyväksyntätestaus, jotka kattavat testauksen laajuuden alimmista yksiköistä aina kokonaisen ohjelmiston käyttäjän hyväksyntätesteihin. Nämä kaikki tasot luetaan funktionaalisen testauksen alle.

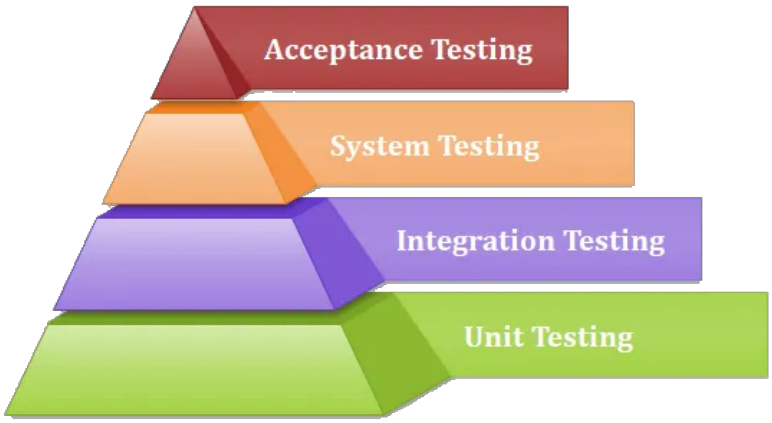
Yksikkötestauksessa keskitytään testaamaan ohjelmiston pienimpiä itsenäisiä osia, kuten funktioita tai metodeja. Tämän tason testaus on kriittistä, sillä se mahdollistaa virheiden nopean tunnistamisen kehityksen alkuvaiheessa, mikä on kustannustehokasta ja vähentää virheiden kulkeutumista myöhempään kehitysvaiheeseen.

Integraatiotestauksella pyritään varmistamaan, että ohjelmiston eri moduulit tai komponentit toimivat yhteen suunnitellusti. Tämä on erityisen tärkeää, kun uusia ominaisuuksia

integroidaan jatkuvasti kehitysprosessin aikana, ja se auttaa tunnistamaan rajapintojen välisiä yhteensopimattomuuksia.

Järjestelmätestaus tarkastelee ohjelmistoa kokonaisuutena ja varmistaa, että kaikki komponentit toimivat yhdessä suunnitellulla tavalla. Järjestelmätestaustasolla toteutetaan päästä päähän -testejä, mikä tarkoittaa kokonaisten käyttötapausten testejä. Tässä vaiheessa arvioidaan myös ohjelmiston suorituskykyä, turvallisuutta ja muita ei-funktionaalisia vaatimuksia.

Lopuksi hyväksyntätestauksessa arvioidaan, vastaako ohjelmisto asiakkaan tai loppukäyttäjän vaatimuksia ja odotuksia. Tämä testausvaihe on ratkaiseva ennen ohjelmiston julkaisua ja se toteutetaan usein yhteistyössä asiakkaan kanssa.



Kuva 2.4. *Testipyramidi, joka esittää testauksen neljä eri tasoa. Lähde: [22]*

Testipyramdi on yleisesti käytetty malli, jolla ohjelmistotestausta pyritään mallintamaan. Pyramidi kuvaa sitä, kuinka tyypillisessä projektissa alempana pyramidissa olevien testien määrä on suurempaa kuin ylempänä olevien. Tässä diplomityössä toteutetaan ja parannetaan järjestelmätason testejä, jotta testauksen taso kokonaisuudessaan parantuisi. Pyramidista on olemassa myös useita eri versioita, joissa tasojen määrä vaihtelee pääsääntöisesti kolmen ja viiden tason välillä. Jotkin pyramidit myös kuvaavat tasot käänteisesti siten, että ylhäällä ovat yksikkötestit ja alhaalla korkeamman tason testit. Nämä ovat kuitenkin harvinaisempia.

2.4 Testausprosessi ketterissä ohjelmistoprojekteissa

Ohjelmistokehityksen kentällä vallitsee kaksi perustavanlaatuista metodologiaa: perinteinen ja ketterä (eng. agile). Ketterä menetelmä tarjoaa iteratiivisen ja tiimipohjaisen lähestymistavan ohjelmistokehitykseen. Ketterä menetelmä painottaa toimivan ohjelmiston nopeaa toimitusta täysin toiminnallisina komponentteina [1]. Tämä lähestymistapa on erityisen tehokas, kun projektivaatimukset muuttuvat usein tai kun projektin lopullista pää-

määrää ei voida täysin määritellä etukäteen.

Luvun 2.3 testipyramidin vastuualueet on jaettu ketterässä tiimissä eri jäsenille. Yksikkötestaus on yleensä ohjelmistokehittäjien vastuulla, ja se suoritetaan koodin kehittämisen aikana. Integraatiotestauksessa kehittäjät ja testaajat työskentelevät yhdessä varmistaakseen, että eri komponentit toimivat saumattomasti yhdessä. Järjestelmätestaus suoritetaan useimmiten testaajien toimesta, mutta siihen voivat osallistua myös loppukäyttäjät — erityisesti silloin, kun testataan käytettävyyttä ja käyttäjäkokemusta. Lopulta hyväksyntätestaus suoritetaan asiakkaan tai loppukäyttäjän toimesta, usein yhteistyössä tuoteomistajan tai scrum masterin kanssa. Samalla varmistetaan, että ohjelmisto vastaa määriteltyjä vaatimuksia ja tarpeita [9]. Tämä kokonaisuus varmistaa, että laadukas ja toimiva ohjelmisto voidaan toimittaa tehokkaasti, nopeasti ja luotettavasti muuttuvassa ympäristössä.

Ketterän vastakohta, perinteinen lähestymistapa, tunnetaan myös vesiputousmenetelmänä. Se noudattaa lineaarista kehitysmallia, jossa jokainen kehitysvaihe on selkeästi erillinen ja seuraava vaihe voi alkaa vasta, kun edellinen on saatu päätökseen [1]. Tämä malli korostaa järjestystä ja suunnitelmallisuutta, mutta sen joustamattomuus muutosten edessä on haasteellista dynaamisemmissa projektikonteksteissa.

Ketterä menetelmä erottuu perinteisestä kehitystavasta useiden avaintekijöiden avulla, jotka tukevat sen joustavaa ja ihmiskeskeistä lähestymistapaa ohjelmistokehitykseen. Ketterä menetelmä tukee inkrementaalista ja iteratiivista kehitystä, mikä mahdollistaa projektin nopean edistymisen pienissä, hallittavissa osissa. Tämä lähestymistapa korostaa suurempaa yhteistyötä tiimin jäsenten kesken ja painottaa prosessin sijaan ihmiskeskeistä kehitystä, mikä tekee ketterästä ihanteellisen valinnan muuttuviin projektivaatimuksiin. Ketterän menetelmän avulla voidaan vastata muutoksiin nopeasti ja tehokkaasti, mikä edellyttää kehityssyklin aikana suurta kurinalaisuutta. Testauksen ja toteutuksen samanaikaisuus sekä monialaisten tiimien vaatimus ovat keskeisiä piirteitä, jotka tukevat ketterän menetelmän periaatteita.[1]

Käyttäjätarinat ovat ketterän menetelmän ytimessä, ja niiden avulla kuvataan järjestelmän uusia tulevia toiminnallisuuksia käyttäjän näkökulmasta. Jokaisen niistä tulee olla [1]:

- **Itsenäinen (Independent)**: Käyttäjätarinat ovat itsenäisiä, mikä mahdollistaa niiden toteutuksen missä tahansa järjestyksessä.
- **Neuvoteltavissa (Negotiable)**: Käyttäjätarinat ovat avoimia keskustelulle ja muutoksille, jotta ne voidaan mukauttaa parhaiten tiimin ja asiakkaan tarpeisiin.
- **Arvokas (Valuable)**: Jokaisen käyttäjätarinan tulee tuottaa konkreettista arvoa asiakkaalle tai käyttäjälle.
- **Arvioitavissa (Estimable)**: Käyttäjätarinat ovat arvioitavissa, jotta voidaan suunnitella sprintin tai iteraation kapasiteettia.

- **Pieni (Small)**: Käyttäjätarinat ovat tarpeeksi pieniä, jotta ne voidaan toteuttaa kohtuullisessa ajassa, yleensä yhden sprintin aikana.
- **Testattavissa (Testable)**: Käyttäjätarinoiden on oltava testattavissa, jotta niiden toteutuksen laatu ja toimivuus voidaan varmistaa.

Nämä **INVEST**-periaatteet varmistavat, että jokainen käyttäjätarina edistää projektin tavoitteita tehokkaasti ja että kehitystiimi voi jatkuvasti arvioida ja sopeuttaa työskentelyään projektin vaatimusten mukaisesti. Ketterän mallin mukainen testausprosessi edistää jatkuvaa palautekiertoa, parantaa lopputuotteen laatua ja vastaa paremmin asiakkaiden tarpeisiin, mikä on dokumentoitu lähteessä.

Ketterissä projekteissa testauksen ja erityisesti testiautomaation merkitys korostuu. Testiautomaatio mahdollistaa jatkuvan laadunvarmistuksen ja nopean palautteen kehitystiimille. Testauksen ja toteutuksen samanaikaisuus, jonka ketterä menetelmä mahdollistaa, tekee automaatiosta erityisen arvokkaan työkalun. Automaatio vähentää inhimillisten virheiden riskiä ja nopeuttaa toistuvia testiprosesseja, kuten regressiotestausta, mikä on olennaista dynaamisessa ja nopeasti muuttuvassa ympäristössä. Tällöin tiimit voivat keskittyä kehittämään uusia ominaisuuksia samalla, kun he luottavat siihen, että vanhat toiminnot säilyvät ehjinä.

2.5 Ohjelmistokehityksen jatkuva integraatio

Jatkuva integraatio (eng. Continuous Integration, CI) on ohjelmistokehityksen käytäntö, jossa kehittäjät yhdistävät työstämänsä koodimuutokset keskusvarastoon (eng. repository) useita kertoja päivässä. Tämän prosessin tavoitteena on tunnistaa ja ratkaista virheitä nopeasti, parantaa ohjelmiston laatua ja vähentää integrointiongelmia. Jatkuvan integraation perusajatus on, että jokainen koodimuutos laukaisee automaattisen testaus- ja rakentamisprosessin (eng. build process), mikä mahdollistaa nopean palautteen antamisen kehittäjille [7].

Jatkuva integraatio tuo mukanaan useita merkittäviä etuja ohjelmistokehitykseen:

- **Nopeampi virheiden tunnistaminen ja korjaaminen**: Kun jokainen koodimuutos testataan välittömästi, virheet havaitaan nopeasti ja ne voidaan korjata ennen kuin ne ehtivät levitä laajemmalle kehityksessä [4]. Virheiden löytäminen mahdollisimman aikaisessa vaiheessa tuo huomattavia taloudellisia etuja.
- **Korkeampi ohjelmiston laatu**: Automatisoidut testit ja rakentamisprosessit varmistavat, että uusi koodi täyttää laadulliset standardit ja ei riko olemassa olevaa toiminnallisuutta [10].
- **Parantunut yhteistyö ja koodin yhdistely**: Jatkuva integraatio rohkaisee tiimin jäseniä integroimaan työnsä usein, mikä vähentää koodiyhdistämisestä johtuvia

konflikteja ja parantaa yhteistyötä [13].

- **Nopeampi julkaisu**: Kun koodi on jatkuvasti testattu ja integroitu, ohjelmiston julkaisu voidaan tehdä nopeammin ja luotettavammin.

Jatkuvan integraation onnistunut toteutus edellyttää useiden käytäntöjen ja työkalujen hyödyntämistä:

- **Versiohallinta**: Kaikkien kehittäjien tulee käyttää versiohallintajärjestelmää (kuten Git), jotta koodimuutoksia voidaan seurata ja yhdistää keskitetysti.
- **Automaattiset testit**: Kaikki testit tulee automatisoida ja ne tulee ajaa jokaisen koodimuutoksen yhteydessä. Tämä sisältää yksikkötestit, integraatiotestit ja mahdollisesti myös järjestelmätestit [7].
- **Rakennuspalvelin**: Rakennuspalvelin (eng. Build server) automatisoi rakentamisprosessin ja varmistaa, että koodi voidaan kääntää ja testata kaikissa ympäristöissä [10].
- **Palautekanavat**: Kehittäjien tulee saada välitöntä palautetta testien tuloksista, jotta he voivat reagoida nopeasti virheisiin ja ongelmiin [4].

Kuvassa 2.5 esitetty vuokaavio havainnollistaa testiautomaatioprosessin vaiheet alkaen uuden muutoksen puskemisesta versionhallintaan aina testilokien tuottamiseen ja raportointiin saakka.

Testiautomaatio käynnistyy aina, kun kehittäjä puskee uuden muutoksen versionhallintajärjestelmään, kuten Gitiin. Tämä muutos voi olla uusi ominaisuus, bugikorjaus tai mikä tahansa koodin muutos. Uusi muutos aktivoi automaatioputken, joka on ennalta määritelty joukko tehtäviä, jotka suoritetaan automaattisesti. Tämä putki käynnistää prosessin, joka sisältää muun muassa koodin rakentamisen (eng. build), testaamisen ja mahdollisen käyttöönoton. Ennen testien aloittamista, automaatioputki alustaa tarvittavat ympäristöt ja varmistaa, että kaikki tarvittavat resurssit, kuten testidata, ovat käytettävissä. Kun kaikki alustukset on suoritettu, testien suorittaminen alkaa. Tämä sisältää useita testisarjoja ja yksittäisiä testejä. Testisarjalla tarkoitetaan testien sarjaa, jotka jollain tapaa liittyvät toisiinsa.

Jokaisen testisarjan alussa suoritetaan testisarjan alustus, jossa määritellään ympäristö ja tarvittavat alkutoimet ennen yksittäisten testien suorittamista. Jokainen yksittäinen testi alkaa testin alustuksella, jossa määritellään testikohtaiset asetukset ja valmistelut. Yhden testisarjan sisällä kaikilla testeillä on sama alustus, ellei toisin ole määritelty. Testi suorittaa määritellyt toiminnot käyttöliittymässä, kuten esimerkiksi käyttäjän kirjautumisen tai lomakkeen täyttämisen. Jokaisen toiminnon jälkeen tarkistetaan, onnistuiko se odotetusti tai vaihtoehtoisesti ei tarkasteta, mutta epäonnistuminen selviää, kun myöhempi toiminto toimii odottamattomasti ja aiheuttaa virheen. Jos toiminto epäonnistuu, raportoidaan virhe ja prosessi keskeytetään kyseisen testin osalta. Jos toiminto onnistui, siirrytään seu-

raavaan vaiheeseen ja tarkistetaan, onko kaikki tarvittavat toiminnot kyseisessä testissä suoritettu.

Jos testissä on jäljellä toimintoja, prosessi palaa toimintojen suorittamiseen käyttöliittymässä, kunnes kaikki toiminnot on suoritettu. Lopuksi varmistetaan, että lopputulos vastaa odotuksia. Tämä voi sisältää esimerkiksi tietojen tarkistamista tai käyttöliittymän tilan varmistamista. Kun yksittäinen testi on suoritettu, suoritetaan test teardown -vaihe, jossa siivotaan kaikki testin aikana tehdyt muutokset ja palautetaan ympäristö alkuperäiseen tilaansa. Seuraava testi aloitetaan uudella testin alustuksella, ja prosessi toistuu kaikille testisarjan testeille.

Kun kaikki testisarjan testit on suoritettu, suoritetaan testisarjan purku, jossa palautetaan koko ympäristö alkuperäiseen tilaansa ja valmistellaan seuraavaa testisarjaa varten. Sarjan alustukseen ja purkuun voidaan esimerkiksi sisällyttää kirjautuminen ja uloskirjautuminen, jotta niitä ei tarvitse toistaa jokaisessa testin alustuksessa. Aiemmin luvussa kuvattu prosessi toistetaan kaikille testisarjoille. Kun kaikki testisarjat on suoritettu, tarkistetaan testien onnistuminen kokonaisuudessaan. Onnistuneet ja epäonnistuneet testit raportoidaan. Lopuksi generoidaan raportti, joka sisältää yhteenvedon testien tuloksista. Tämä raportti antaa kehittäjille ja sidosryhmille tärkeää tietoa ohjelmiston laadusta ja mahdollisista virheistä ja helpottaa virheiden korjausta niiden ilmaantuessa.

Jatkuva integraatio on keskeinen osa DevOps-kulttuuria, joka yhdistää ohjelmistokehityksen ja IT-toimintojen hallinnan. DevOps pyrkii nopeuttamaan ohjelmistojen toimitusprosessia ja parantamaan ohjelmistojen laatua hyödyntämällä jatkuvan integraation kaltaisia käytäntöjä. DevOps-malli korostaa jatkuvaa palautteenantoa, automaatiota ja yhteistyötä eri tiimien välillä [13].

Vaikka jatkuva integraatio tarjoaa merkittäviä etuja, sen käyttöönottoon liittyy haasteita:

- **Infrastruktuurin kustannukset**: Jatkuva integraatio vaatii investointeja infrastruktuuriin, kuten rakennuspalvelimiin ja testaustyökaluihin. Tämä voi olla haastavaa pienille tiimeille tai projekteille, joilla on rajalliset resurssit [7].
- **Testien laatu ja kattavuus**: Automaatiotestien laatu ja kattavuus on kriittistä prosessin kannalta. Puutteelliset testit voivat johtaa siihen, että virheitä jää huomaamatta, mikä vaarantaa jatkuvan integraation prosessin tehokkuuden. Ratkaisuna on investoida kattaviin ja laadukkaisiin testipaketteihin [4]. Lisäksi on oleellista huolehtia siitä, että puutteellisen kattavuuden tapauksessa ainakin kriittisimmät toiminnot testataan.
- **Organisatoriset muutokset**: Jatkuva integraatio vaatii muutoksia kehitystiimien työskentelytapoihin ja kulttuuriin. On tärkeää, että kaikki tiimin jäsenet sitoutuvat jatkuvan integraation periaatteisiin ja käytäntöihin [10].

Yhteenvetona voidaan todeta, että jatkuva integraatio on keskeinen osa modernia oh-

Kuva 2.5. *Testiautomaatioputken toimintaa kuvaava kaavio*

jelmistokehitystä, joka parantaa ohjelmistojen laatua ja nopeuttaa kehitysprosessia. Sen menestyksekäs käyttöönotto vaatii huolellista suunnittelua, investointeja infrastruktuuriin ja tiimin sitoutumista. Kun nämä elementit ovat kohdallaan, jatkuva integraatio voi merkittävästi tehostaa ohjelmistokehitystä ja parantaa lopputuotteen laatua.

3. LAADUKKAAN TESTAUKSEN OMINAISUUDET

Ketterässä ohjelmistokehityksessä testauksen merkitys korostuu, sillä sen avulla varmistetaan, että sovellus pystyy vastaamaan jatkuviin muutoksiin ja kehittyviin vaatimuksiin. Laadukas testaus ei tarkoita pelkästään virheiden löytämistä, vaan sen avulla tuetaan ohjelmiston kehityksen jokaista vaihetta ja varmistetaan, että kaikki kriittiset toiminnallisuudet toimivat suunnitellusti. Testauksen suunnittelussa keskeistä on varmistaa, että testit ovat ymmärrettäviä ja helposti ylläpidettäviä, jotta niitä voidaan käyttää projektin koko elinkaaren ajan. Lisäksi testien on oltava kattavia ja robusteja eli niiden tulee pystyä käsittelemään monimutkaisiakin tilanteita luotettavasti ja joustavasti. Tässä luvussa tarkastellaan näitä laadukkaan testauksen ominaisuuksia, joiden avulla voidaan parantaa ohjelmistojen laatua ja kehityksen sujuvuutta ketterässä ympäristössä.

Seuraavissa alaluvuissa syvennytään erityisesti neljään laadukkaan testauksen keskeiseen piirteeseen. Ensimmäiseksi tarkastellaan testauksen luotettavuutta ja robustisuutta, joiden avulla varmistetaan, että testit toimivat luotettavasti erilaisissa ympäristöissä ja olosuhteissa. Toisessa alaluvussa käsitellään, miksi testien ymmärrettävyys on merkityksellistä ja kuinka selkeästi rakennetut testit voivat helpottaa niin tiimin sisäistä yhteistyötä kuin testien ylläpitoa. Lisäksi samassa luvussa keskitytään testien ylläpidettävyyteen ja siihen, miten testit voidaan rakentaa kestämään muutoksia ilman, että ne vaativat jatkuvaa huoltamista. Viimeisenä käsitellään testauksen kattavuutta, jonka tarkoituksena on varmistaa, että kaikki tärkeät toiminnallisuudet ja mahdolliset virhetilanteet on testattu kattavasti, jotta tuotteen laatu ja luotettavuus voidaan taata koko kehitysprosessin ajan.

3.1 Testauksen luotettavuus ja robustisuus

Automaattisen testauksen luotettavuus ja robustisuus ovat olennaisia tekijöitä laadukkaan ohjelmistonkehityksen varmistamiseksi. Luotettava ja robusti testaus takaa sen, että sekä ohjelmisto että testaus toimivat odotetulla tavalla erilaisissa olosuhteissa ja että virhetilanteet tunnistetaan ja käsitellään asianmukaisesti. Tämä osio käsittelee näiden ominaisuuksien merkitystä ja tarjoaa käytännön menetelmiä niiden saavuttamiseksi.

Testien luotettavuus tarkoittaa testien kykyä antaa johdonmukaisia ja toistettavia tuloksia. Luotettavat testit vähentävät väärien positiivisten ja negatiivisten tulosten määrää, mikä parantaa ohjelmiston laadunvalvontaa ja lisää kehittäjien luottamusta testien tuloksiin. Yk-

si tapa parantaa testien luotettavuutta on varmistaa, että testit ovat idempotentteja, eli niiden tulos pysyy samana riippumatta siitä, kuinka monta kertaa testi suoritetaan samassa ympäristössä. Testien luotettavuuden parantamiseksi on suositeltavaa käyttää seuraavia menetelmiä:

- Selkeä ja tarkka testitapausten määrittely: Testitapaukset tulee määritellä yksiselitteisesti, ja niiden odotetut tulokset on dokumentoitava tarkasti. Tämä vähentää tulkinnanvaraisuuden riskiä ja varmistaa, että kaikki testaajat ymmärtävät testit samalla tavalla.
- Monimuuttujatestit: Testien tulisi kattaa laaja valikoima erilaisia käyttöolosuhteita ja syöteyhdistelmiä. Tämä auttaa varmistamaan, että järjestelmä toimii odotetusti erilaisissa tilanteissa ja että kaikki mahdolliset virheskenaariot tulevat testatuiksi.

Robustisuus puolestaan viittaa testien kykyyn käsitellä poikkeustilanteita ja virheitä hallitusti. Robustit testit eivät kaada testiympäristöä tai anna virheellisiä tuloksia odottamattomissa tilanteissa, vaan tunnistavat ja raportoivat virheet asianmukaisesti. Tämä ominaisuus on erityisen tärkeä, kun testataan järjestelmiä, jotka toimivat kriittisissä tai monimutkaisissa ympäristöissä. Robustisuuden selvittämiseksi ja parantamiseksi voidaan käyttää seuraavia menetelmiä:

- Poikkeuskäsittelyn testaus: Testit tulee suunnitella siten, että ne kattavat myös poikkeustilanteet, kuten virheelliset syötteet, verkon katkokset ja muut odottamattomat tapahtumat. Poikkeuskäsittelyn testaaminen auttaa varmistamaan, että järjestelmä toimii luotettavasti myös häiriötilanteissa. Usein testiautomaation työkalut, kuten Robot Framework, tarjoavat sisäänrakennetun poikkeuksenkäsittelyn, mikä helpottaa robustien testien kehittämistä.
- Erillisten testausympäristöjen käyttö: Testit tulee suorittaa erillisissä, hallituissa ympäristöissä, jotta ulkoiset tekijät eivät vaikuta tuloksiin. Tämä auttaa varmistamaan, että testitulos on seurausta testattavan ohjelmiston toiminnasta, ei ympäristön satunnaisista muutoksista.

Luotettavuuden ja robustisuuden parantaminen vaatii systemaattista lähestymistapaa ja huolellista testien suunnittelua. Käyttämällä edellä mainittuja menetelmiä voidaan varmistaa, että testit tarjoavat johdonmukaisia tuloksia ja että ne kykenevät käsittelemään poikkeustilanteet tehokkaasti. Näin parannetaan ohjelmiston laatua ja vähennetään virheiden aiheuttamia riskejä tuotantoympäristössä. Luotettavat ja robustit testit tuovat turvallisuuden tunnetta ja luottamusta ohjelmiston oikeasta toiminnasta ohjelmiston kehittäjille ja sidosryhmille. Näin myös ohjelmistosta saadaan luotettava.

3.2 Testien ymmärrettävyys ja ylläpidettävyys

Testien ymmärrettävyys on kriittistä ketterässä kehityksessä, sillä se mahdollistaa nopean ja tehokkaan kommunikaation tiimin sisällä sekä helpottaa uusien jäsenten perehtymistä projektiin. Ketterissä ympäristöissä, joissa muutokset ja päivitykset ovat jatkuvia, selkeästi kirjoitetut testit helpottavat kehitystiimin jäsenten ymmärrystä siitä, mitä testi tarkoittaa ja mitä se tarkastaa. Tämä vähentää väärinkäsityksiä ja nopeuttaa virheiden korjausta sekä testien muutoksiin kuluvaa aikaa ohjelman muuttuessa.

Testien ymmärrettävyyden varmistamiseksi on tärkeää, että testitapaukset kirjoitetaan selkeästi ja yksinkertaisesti. Yksinkertainen kieli ja selkeät tavoitteet tekevät testeistä helpommin lähestyttäviä ja ymmärrettäviä kaikille tiimin jäsenille. Lisäksi testien dokumentointi tulisi pitää ajan tasalla, jotta testien tarkoitus ja toiminta olisi selvää kaikille projektin osapuolille. Dokumentoinnin tarpeellinen määrä riippuu testin ymmärrettävyydestä, johon voidaan vaikuttaa selkeillä muuttujanimillä, lyhyemmillä testitapauksilla ja selkeällä ohjelmistoarkkitehtuurilla.

Kun testit on kirjoitettu ymmärrettävästi, virheenkorjausprosessi nopeutuu merkittävästi. Selkeät testit helpottavat virheiden jäljittämistä ja niiden syiden ymmärtämistä, mikä on erityisen tärkeää ketterässä kehityksessä, jossa nopeat syklit ja jatkuva palautteenanto ovat käytäntöjä. Kehitystiimin kyky reagoida nopeasti ja tehokkaasti virhetilanteisiin parantaa ohjelmiston laatua ja vähentää kehityskustannuksia.

Parantaakseen testien ymmärrettävyyttä, tiimi voi myös hyödyntää hyviä käytäntöjä kuten koodin katselmointeja, joissa testitapauksia tarkastellaan ja arvioidaan yhdessä. Tämä ei ainoastaan lisää ymmärrystä testien tarkoituksesta ja toiminnasta, vaan myös edistää yhteisöllistä oppimista ja tiimin sisäistä kommunikaatiota. Koodin katselmoinnit tarjoavat mahdollisuuden keskustelulle ja parannusehdotuksille, mikä lisää ymmärrettävyyttä ja testien laatua. Siksi katselmoinnit ovat tavallinen käytäntö ohjelmistokehityksessä.

Automaatiotestauksen toimivuuden edellytyksenä tulee säännöllisesti tarkistaa ja päivittää testitapauksia, jotta ne pysyvät yhteneväisinä ohjelmiston kehityksen kanssa. Testien jatkuva ylläpito ja päivitys varmistavat, että ne pysyvät relevantteina ja toimivina koko projektin ajan.

Näiden käytäntöjen avulla testien ymmärrettävyys pysyy korkealla tasolla. Tämä tukee ketterän kehityksen nopeaa sykliä ja edistää tehokasta ohjelmiston toimittamista.

Ymmärrettävyys liittyy vahvasti myös ohjelmiston ylläpidettävyyteen. Jatkuvasti muuttuvassa ja iteratiivisessa ketterässä ohjelmistokehityksessä testiautomaation ylläpidettävyys on avainasemassa projektin pitkäaikaisen onnistumisen kannalta. Kun testejä kirjoitetaan, tulee aina pitää mielessä mahdolliset tulevat muutokset vaatimuksissa ja järjestelmän toteutuksessa. Jos testit eivät ole helposti muokattavissa, jopa pienet muutokset

voivat johtaa siihen, että suuri määrä testejä rikkoutuu ja niiden korjaamiseen kuluu huomattava määrä aikaa.

Automaatisoidut testit suoritetaan usein, joten niitä tulee voida helposti päivittää. Tämä mahdollistaa nopean palautteen saamisen ja testikattavuuden ylläpidon jatkuvasti muuttuvassa kehitysympäristössä. Testiautomaation yksi suurimmista eduista on sen tuoma apu ohjelman toiminnallisuuden kasvaessa, sillä ohjelman kehittyessä testattavien toiminnallisuuksien määrä luonnollisesti kasvaa. Automaattiset testit tekevät regressiotestauksesta hallittavaa. Jos testejä ei ole tehty ylläpidettäviksi, tulee testien ylläpidosta erittäin vaikeaa.[6, s. 494]

Yksi tärkeimmistä strategioista, on testien yksinkertaistaminen ja keskittyminen olennaisiin järjestelmän vastuualueisiin [5]. Kun testit keskittyvät ydintoimintoihin ja varmistavat, että järjestelmä täyttää näitä vastuualueita riippumatta siitä, kuinka järjestelmä on toteutettu, testit ovat joustavampia muutoksille. Tämä vähentää testien haurautta, eli tilannetta, jossa pienetkin muutokset järjestelmän koodissa saavat useita testejä epäonnistumaan.

Testiautomaation ylläpidettävyyden kannalta keskeisiä periaatteita ovat siis seuraavat:

- **Turhat yksityiskohdat** - turhat yksityiskohdat, jotka eivät ole olennaisia testin tarkoituksen kannalta, pitäisi piilottaa. Tällaiset yksityiskohdat voidaan piilottaa käyttämällä abstrakteja avainsanoja, jotka tekevät testistä selkeämmän ja helpommin muokattavan. Avainsanoista lisää luvussa 4.1.
- **Duplikointi** - koodin tai testivaiheiden toistaminen eri testeissä nostaa merkittävästi ylläpitokustannuksia. Kun testejä joudutaan päivittämään, jokainen toistettu osa täytyy käydä läpi ja muokata. Ratkaisuna tähän voidaan käyttää uudelleenkäytettäviä avainsanoja tai toiminnallisuuksia, jotka on eristetty yksittäisiksi moduuleiksi. Näin muutokset täytyy tehdä vain kerran, mikä parantaa testien ylläpidettävyyttä huomattavasti.
- **Muutoskestävyyden parantaminen** - Hyvin suunnitellut avainsanat ja eriytetyt moduulit auttavat varmistamaan, että testit kestävät suuria muutoksia järjestelmän toteutuksessa. Esimerkiksi siirtyminen eri teknologioiden välillä saattaa vaatia muutoksia vain niihin avainsanoihin, jotka käsittelevät käyttöliittymän vuorovaikutusta, mutta testien ydinlogiikka pysyy ennallaan.

Lisäksi testien ylläpidettävyys ei perustu vain teknisiin ratkaisuihin, vaan myös hyvään dokumentaatioon. Jokaisen testin tarkoitus, logiikka ja vaikutusalue tulisi olla selkeästi kuvattu, jotta kehittäjät ja testaajat voivat nopeasti ymmärtää, miten testit liittyvät järjestelmän vaatimuksiin. Tämä nopeuttaa vianetsintää ja uusien tiimiläisten perehdyttämistä, mikä on erityisen tärkeää ketterissä tiimeissä, joissa tiimijäsenet ja vastuualueet voivat vaihtua nopeasti.

Ylläpidettävyyden parantaminen testiautomaatiossa ei ole vain tekninen kysymys, vaan

se liittyy myös testien kirjoittamisen tapaan ja siihen, miten testit suunnitellaan alusta lähtien. Ketterässä kehityksessä tämä tarkoittaa jatkuvaa refaktorointia ja parannusten tekemistä sekä järjestelmään että testeihin, jotta ne säilyttävät joustavuutensa ja kykynsä vastata nopeasti muuttuviin vaatimuksiin.

3.3 Testien kattavuus

Kun testien luotettavuus ja ymmärrettävyys on varmistettu, on tärkeää varmistaa myös testien kattavuus. Testikattavuus on keskeinen mittari, joka kertoo, kuinka hyvin testit käyvät läpi ohjelmiston koodin ja toiminnot. Kattavissa testeissä huomioidaan kaikki kriittiset polut ja toiminnot ohjelmistossa, mikä varmistaa, että ohjelmisto toimii suunnitellusti kaikissa tilanteissa ja että kaikki mahdolliset virhetilanteet on testattu.

Testikattavuuden optimoimiseksi on tärkeää hyödyntää eri testaustasoja, kuten yksikkötestaus, integraatiotestaus, ja järjestelmätestaus. Yksikkötestauksella varmistetaan, että ohjelmiston pienimmät yksiköt toimivat oikein. Integraatiotestauksessa tarkastellaan, miten eri ohjelmiston osat toimivat yhdessä, ja järjestelmätestaus kattaa koko ohjelmiston toiminnallisuuden testaamisen.

Testikattavuuden seurantaan voidaan käyttää automatisoituja työkaluja, jotka analysoivat, mitä osia koodista on testattu ja mitkä osat vaativat lisää huomiota. Työkaluja testikattavuudelle on monenlaisia, joista osa tarjoaa visuaalisen raportin testikattavuudesta, mikä auttaa kehitystiimiä tunnistamaan alueet, joille tarvitaan lisää testejä.

Laajan testikattavuuden ylläpitäminen vaatii jatkuvaa arviointia ja päivitystä, sillä ohjelmiston kehittyessä myös testien tulee kehittyä. Kattavuuden tavoitteena on paitsi tunnistaa ohjelmiston virheet myös varmistaa, että ohjelmisto täyttää kaikki käyttäjien ja liiketoiminnan vaatimukset. Suurta testikattavuutta tavoitellaan yleensä alemmilla testaustasoilla. Toisin sanottuna yksikkötestejä tehdään enemmän, koska niiden toteuttaminen ja ylläpitäminen on nopeampaa. Jos korkean tason testejä toteutetaan liikaa, voi ylläpidon määrä paisua kannattamattomaksi.

Näiden toimien avulla voidaan varmistaa, että ketterässä ohjelmistokehityksessä testien suunnittelu, ylläpidettävyys ja kattavuus tukevat tehokasta ja laadukasta ohjelmistotuotantoa.

4. TYÖKALUJEN ESITTELY

Luvussa 4 esitellään tässä diplomityössä käytetyt testiautomaatiotyökalut. Työkalut ja teknologiat ovat annettuja, joten näiden valintaa ei perustella, vaan esitellään niiden ominaisuuksia. Tässä luvussa pohditaan, kuinka näiden avulla kehitetään web-sovelluksen testauksen laadukkuutta. Pääasiallinen työkalu on Robot Framework, joka tarjoaa avainsanapohjaisen testikielen ja laajennettavuuden erilaisten kirjastojen kautta. Luvussa käsitellään SeleniumLibrarya ja Browser-kirjastoa, joista jälkimmäinen tuo modernimman ratkaisun selainpohjaisten sovellusten testaamiseen, erityisesti asynkronisten toimintojen osalta.

Lisäksi luku kattaa Page Object Model (POM) -suunnittelumallin, joka selkeyttää ja parantaa testikoodin ylläpidettävyyttä eriyttämällä testien logiikan ja sivuelementit. Lopuksi esitetään Robot Frameworkin yleiset käytänteet ja syntaksi, jotka varmistavat testien luotettavuuden ja skaalautuvuuden projektin laajentuessa.

4.1 Robot Framework testiautomaatiotyökaluna

Robot Framework on avoimen lähdekoodin automaatiotyökalu, joka on suunniteltu helpottamaan testaus- ja automaatiotehtäviä ohjelmistokehityksen alueella. Se on kehitetty käyttäen Pythonia, ja sen suosio perustuu sen kykyyn tukea monipuolisia testausstrategioita, kuten hyväksymistestivetoinen kehitys (ATDD) [18]. Robot Framework tarjoaa käyttäjäystävällisen syntaksin, joka mahdollistaa testitapausten kirjoittamisen selkeällä ja ymmärrettävällä tavalla.

Robot Frameworkin käyttö on suosittua monesta syystä. Sen tarjoamien avainsanavetoisten (eng. keyword-driven) ja vaihtoehtoisesti datavetoisten (eng. data-driven) syntaksien ansiosta testitapauksista voidaan tehdä selkeitä ja helposti ymmärrettäviä. Tämä vähentää uuden oppimiskynnyksen korkeutta ja tekee työkalusta houkuttelevan vaihtoehdon monille tiimeille. Lisäksi Robot Framework tukee laajasti kirjastoja ja laajennuksia, mikä tekee siitä erittäin mukautettavan eri tarpeisiin. Sen avoimen lähdekoodin luonne tarkoittaa myös, että käyttäjäkunta on aktiivinen ja resurssit päivittyvät jatkuvasti.

Robot Framework erottuu monista muista testausvälineistä sen joustavuuden ja laajennettavuuden ansiosta. Toisin kuin monet muut työkalut, se ei rajoitu vain tiettyihin ohjel-

mointikieliin tai ympäristöihin, vaan sitä voidaan käyttää laajalti eri teknologioiden kanssa. Tämä tekee siitä ihanteellisen valinnan moniteknologiaisille projekteille. Lisäksi sen kyky käsitellä sekä testiautomaatiota että kehityksen tehtäviä tekee siitä arvokkaan resurssin ketterissä- ja DevOps-ympäristöissä.

Yksi Robot Frameworkin tehokkaimmista ominaisuuksista on kyky määritellä käyttäjäkohtaisia avainsanoja, mikä mahdollistaa testiskriptien räätälöinnin tietyille vaatimuksille. Käyttäjä voi luoda uusia avainsanoja, jotka koostuvat olemassa olevista avainsanoista, luoden näin abstraktiotasoja, jotka helpottavat monimutkaisten testien hallintaa. Tämä ominaisuus ei ainoastaan paranna testien luettavuutta vaan myös tehostaa testausprosessia, sillä yleisesti käytettyjä toimintoja voidaan uudelleenkäyttää eri testeissä. Esimerkkinä tästä on itse tehdyn avainsanan nimi "Login To Website", joka käyttää toimiakseen muita valmiita avainsanoja, joita ovat "Open Browser", "Fill Text", "Click". Tällaisen avainsanan toimintaperiaate on seuraavanlainen: "Avaa selain, syötä käyttäjätunnus ja salasana, ja paina sisäänkirjautumisnappia."Tämän avainsanan avulla siis voidaan lyhentää kolme avainsanaa yhdeksi kuvaavaksi toiminnoksi. Räätälöityjen avainsanojen käyttö tekee testien ylläpidosta siis helpompaa ja vähentää virheiden mahdollisuutta, kun samoja toistuvia askeleita ei tarvitse kirjoittaa uudelleen ja uudelleen.

Nämä ominaisuudet tekevät Robot Frameworkista erittäin suositun ja tehokkaan työkalun monenlaisissa ohjelmistoprojekteissa. Erityisesti tämä koskee jatkuvaa ja dynaamista testausta vaativia projekteja.

4.2 Robot Frameworkin SeleniumLibrary

SeleniumLibrary on suosittu Web-sovellusten testauskirjasto Robot Frameworkille, ja se hyödyntää Selenium-työkalua selainpohjaisten testien automatisointiin [16]. SeleniumLibrary tarjoaa koodausrajapinnan selainpohjaiselle automaatiolle [24]. Selenium on pitkään ollut laajalti käytetty työkalu, joka tukee useita selaimia ja käyttöjärjestelmiä. Se tarjoaa vankan perustan toiminnalliselle testaukselle ja on erityisen tehokas, kun sitä käytetään yhdessä Robot Frameworkin avainsana-pohjaisen testausmallin kanssa. SeleniumLibraryä on käytetty perinteisesti automatisoimaan monimutkaisia selaintoimintoja, kuten lomakkeiden täyttämistä, elementtien käsittelyä, navigointia ja JavaScript-käskyjen suorittamista.

Vaikka SeleniumLibrary on pitkään ollut suosittu web-sovellusten testauksen väline, sen rajoitukset, kuten vaikeudet asynkronisten toimintojen käsittelyssä, ovat johtaneet tarpeeseen päivittää testausratkaisuja modernimpaan Browser-kirjastoon. SeleniumLibraryssä on omat heikkoutensa, joihin luvun 4.3 Browser-kirjasto on löytänyt uusia ratkaisuja. Toisaalta Browser-kirjasto ei tarjoa kaikkia SeleniumLibraryn ominaisuuksia ja moni sopiva korvaava avainsana ei ole aina helposti löydettävissä, mikä luo haasteita siirtymään.

SeleniumLibrary tarjoaa seuraavia etuja:

Laaja selain- ja järjestelmätuki: SeleniumLibrary tukee useita selaimia, kuten Chrome, Firefox ja Safari, sekä useita käyttöjärjestelmiä, mukaan lukien Windows, Linux ja macOS [20]. Helppo käyttöönotto: Asennus tapahtuu yksinkertaisella pip-komennolla ilman lisäriippuvuuksia [8]. Yhteensopivuus vanhempien projektien kanssa: Koska Selenium on ollut markkinoilla pidempään, monet projektit käyttävät sitä jo valmiiksi, mikä tekee sen integroimisesta helppoa ja mahdollistaa olemassa olevan automaation hyödyntämisen.

Vaikka SeleniumLibrary on ollut vakiintunut ratkaisu, sillä on myös rajoituksia. Esimerkiksi Shadow DOM ja iframes-elementtien käsittely on haastavaa, koska SeleniumLibrary toimii suoraan selaininstanssin päällä ilman mahdollisuutta hallita verkkoliikenteen tasoa [8]. Lisäksi selainten päivittämiseen liittyvät haasteet voivat aiheuttaa odottamattomia katkoksia testauksessa [16].

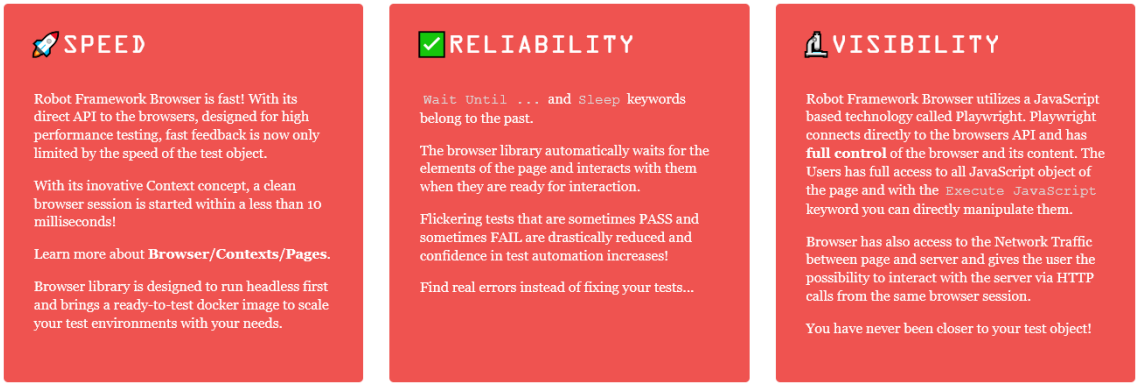
4.3 Robot Frameworkin Browser-kirjasto

Browser-kirjasto, toiselta nimeltään Robot Framework Browser, on Robot Frameworkin laajennus, joka mahdollistaa web-sovellusten testaamisen selainympäristössä. Tämä laajennus hyödyntää Playwright-tekniikkaa, modernia verkkoautomaatiokirjastoa, joka tarjoaa tavan hallita selaimia ja niiden toimintaa kattavasti. Playwrightin avulla voidaan suorittaa testejä eri selaimilla, kuten Chromella, Firefoxilla ja Safarilla, mikä lisää testien kattavuutta ja varmistaa sovelluksen yhteensopivuuden eri selainympäristöissä.

Robot Framework Browser laajentaa Robot Frameworkin kyvykkyyksiä selainpohjaiseen automaatioon tarjoamalla monipuolisia työkaluja esimerkiksi elementtien löytämiseen, sivujen hallintaan ja vuorovaikutuksen automatisointiin. Tämä mahdollistaa esimerkiksi käyttäjätarinoiden testaamisen end-to-end-skenaarioissa, joissa koko toiminnallisuus testataan esimerkiksi kirjautumisesta ostosten tekemiseen ja maksamiseen saakka.

Yksi merkittävistä hyödyistä Robot Framework Browserin käytössä on sen kyky käsitellä asynkronisia toimintoja, jotka ovat yleisiä moderneissa web-sovelluksissa. Asynkronisten toimintojen hallinta on usein haastavaa, mutta Playwrightin avulla voidaan helposti odottaa tiettyjen elementtien ilmestymistä tai tapahtumien suorittamista ennen testin jatkumista. Tätä voidaankin pitää yhtenä Browser-kirjaston parhaimmista ominaisuuksista, koska se mahdollistaa testien etenemisen heti, kun elementti näkyy, eikä ylimääräiseen odotteluun tarvitse tuhlata aikaa. Tämä lisää myös testien luotettavuutta ja vähentää virheitä, jotka johtuvat ajoitusongelmista. [2]

Robot Framework Browserin avulla voidaan myös helposti integroida testit osaksi jatkuvaa integraatiota ja toimitusputkia (CI/CD). Tämä mahdollistaa testien automaattisen ajamisen aina, kun koodiin tehdään muutoksia, mikä nopeuttaa palautteen saamista ja vähentää virheiden leviämistä tuotantoon. Lisäksi se tukee erilaisten mittareiden ja ra-



Kuva 4.1. *Browser-kirjaston tuomat edut [2]*

```
1   *** Settings ***
2   Library     Browser
3
    Load in Interactive Console
4   *** Variables ***
5   ${URL}  https://localhost:7000/
6
7   *** Test Cases ***
    Run | Debug | Run in Interactive Console
8   Login Test
9       New Browser
10      New Page     ${URL}
11      Click     text=Login
12      Fill Text     id=username     myusername
13      Fill Text     id=password     mypassword
14      Click     text=Submit
15      Get Text     id=welcome-message     Welcome, myusername!
16      Close Browser
```

Kuva 4.2. *Robot Framework koodi -esimerkki*

porttien generointia, jotka auttavat seuraamaan testien kattavuutta ja suorituskykyä ajan mittaan.

Kirjaston käyttöönotto on suoraviivaista, sillä Robot Framework Browser on helposti asennettavissa ja konfiguroitavissa. Asennuksen jälkeen voidaan määritellä testiskenaariot hyödyntäen avainsanoja, jotka tekevät testien kirjoittamisesta ja ylläpidosta yksinkertaista. Esimerkiksi kirjautumistoiminnon testaaminen voi näyttää kuvan 4.2 mukaiselta:

Tässä esimerkissä avataan selain, navigoidaan kirjautumissivulle, syötetään käyttäjätunnus ja salasana, painetaan kirjautumispainiketta ja haetaan tervetuloviesti tarkistusta varten. Lopuksi selain suljetaan. Tämä yksinkertainen esimerkki havainnollistaa, kuinka Robot Framework Browserin valmiita avainsanoja voidaan käyttää tehokkaasti ja ymmärret-

```
1  *** Settings ***
2  Library     Browser
3  Resource     ../Resources/PageObject/KeywordDefinitionFiles/LoginPage.robot
4  Resource     ../Resources/PageObject/KeywordDefinitionFiles/Common.robot
5
6  Test Setup     Set Up Browser
7  Test Teardown     Close Browser
8
   Load in Interactive Console
9  *** Variables ***
10 ${URL}  https://localhost:7000/
11
12 *** Test Cases ***
   Run | Debug | Run in Interactive Console
13 Login Test
14     Login With Credentials     myusername     mypassword
15     Is In Correct Login State     SHOULD_LOGIN=true
16     Close Browser
```

Kuva 4.3. *Parannettu kirjautumistesti -esimerkki*

tävästi web-sovellusten testaamiseen [19].

Testeistä saadaan vielä lyhyempiä ja selkeämpiä luomalla omia avainsanoja ja käyttämällä test setup ja test teardown toiminnallisuuksia. Lisäksi uusien testien luominen on nopeampaa, kun ei tarvita turhaa toistoa. Tästä esimerkki kuvassa 4.3, jossa näitä on hyödynnetty. Testi on sama kuin kuvan 4.2 esimerkissä.

Näillä ominaisuuksilla ja eduilla Robot Framework Browser tarjoaa tehokkaan ratkaisun web-sovellusten automaatiotestaukseen parantaen testauksen luotettavuutta, nopeutta ja ylläpidettävyyttä.

4.4 Robot frameworkin yleiset käytänteet ja syntaksi

Robot Frameworkin yleiset käytänteet ja syntaksi ovat keskeisiä tekijöitä, jotka vaikuttavat testien ymmärrettävyyteen, ylläpidettävyyteen ja suoritusnopeuteen. Hyvin suunnitellut ja toteutetut testit parantavat ohjelmiston laadunvalvontaa ja helpottavat virheiden havaitsemista ja korjaamista.

Testitiedostojen rakenne Robot Frameworkissa on selkeä ja looginen. Testitiedostot jaetaan yleensä osioihin, kuten asetukset, muuttujat, testit ja avainsanat. Asetuksissa määritellään testin asetukset, kuten käytettävät kirjastot ja resurssit. Muuttujissa määritellään testissä käytettävät muuttujat. Testeissä varsinaiset testitapaukset määritellään ja avainsanoissa määritellään uudelleenkäytettävät avainsanat, jotka koostuvat useista testiaskeleista. Ohjelmistokehityksen periaatteisiin kuuluen on sopivaa pilkkoa nämä osiot eri tiedostoihin, jotta tiedostojen koko ei kasva liian suureksi. Avainsanoja saadaan siirrettyä tiedostojen välillä käyttämällä komentoa ’Resource’ ja muuttujia komennolla ’Variables’.

```
Variables    ../Locators/Common.py
```

Sopiva abstraktiotaso on olennainen testien selkeyden ja ylläpidettävyyden kannalta. Tämä saavutetaan käyttämällä avainsanoja, jotka kokoavat yhteen useita testiaskeleita. Esimerkiksi avainsana "Login To Website" voi koostua askelista, kuten selaimen avaaminen, kirjautumissivulle navigointi, käyttäjätunnuksen ja salasanan syöttäminen sekä kirjautumispainikkeen painaminen. Tällaiset avainsanat parantavat testien luettavuutta ja vähentävät toistuvien tehtävien kirjoittamista.

Robot Frameworkin kehittäjä Pekka Klärck on laatinut testien kirjoittamiseen ohjeita, joissa korostetaan seuraavia periaatteita [21]:

- Käytä testisarjoille setup ja teardown -komentoja testien nopeuttamiseen ja siivoamiseen.
- Vältä kovakoodattuja arvoja ja käytä muuttujia, jotta testit ovat helpommin muokattavissa.
- Siirrä monimutkainen logiikka avainsanoista kirjastoihin, mikä parantaa testien ylläpidettävyyttä.
- Käytä pollausmekanismeja synkronointiin odottamisen sijaan; Browser Libraryssä tämä on sisäänrakennettu ominaisuus.

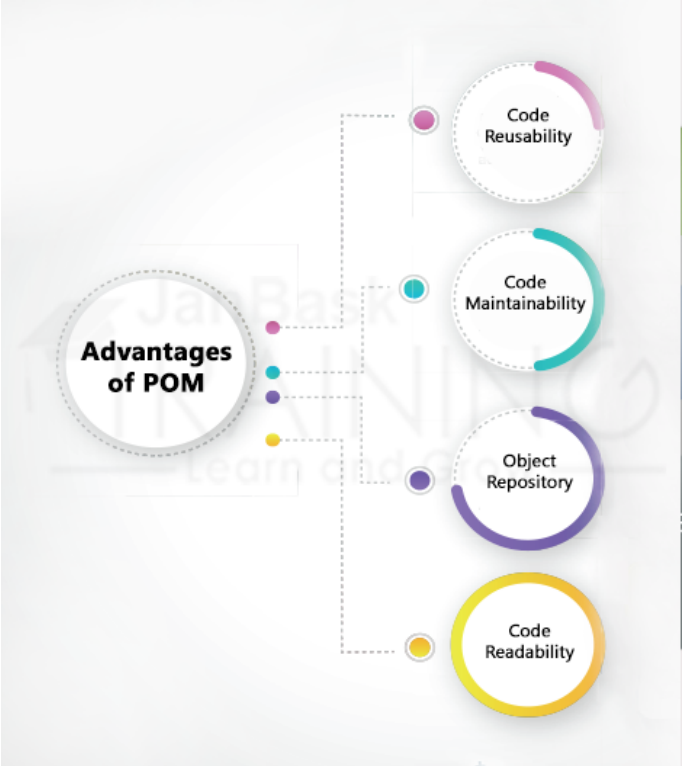
Lisäksi on myös käytäntöjä, joita tulisi välttää:

- Vältä liian matalaa abstraktiotasoa, jossa yksittäiset testiaskeleet ovat liian yksityiskohtaisia ja monimutkaisia. Esimerkiksi yksittäisten HTML-elementtien tarkka sijainti testikoodissa voi johtaa testien haurastumiseen, jos käyttöliittymä muuttuu.
- Kommenttien ja dokumentaation sijaan tulisi käyttää abstraktiotasoja. Robot Frameworkin avainsanat ovat juuri tätä varten. Liian tekniset kommentit voivat hämärtää testien tarkoitusta ja vaikeuttaa niiden ymmärtämistä.
- Testeissä tulee aina olla tarkistuksia. Testejä ilman tarkistuksia ei voida pitää luotettavina, sillä ne eivät validoi testattavan sovelluksen toimivuutta.
- Älä käytä liikaa muuttujia testien tasolla, sillä se voi tehdä testeistä monimutkaisia ja vaikeasti ylläpidettäviä.

Gherkin-tyylinen ATDD/BDD (Given, When, Then) on suositeltava käytäntö, joka parantaa testien luettavuutta ja ymmärrettävyyttä. Tämä tyyli tekee testien tarkoituksen ja logiikan selkeämmäksi kaikille sidosryhmille, mukaan lukien ei-tekniset jäsenet.

Viimeisenä asioita, joita tulisi välttää:

- Testeissä, joissa on turhia tarkistuksia, voi olla tarpeetonta monimutkaisuutta, joka hidastaa testien suorittamista ja vaikeuttaa niiden ylläpitoa.



Kuva 4.4. *Page object model mallinnuksen edut kuvattuna [23]*

- Toisistaan riippuvat testit voivat johtaa tilanteisiin, joissa yhden testin epäonnistuminen vaikuttaa muihin testeihin, mikä vaikeuttaa virheiden diagnosointia.
- Komplekseja logiikoita testidatassa tulisi välttää, koska ne voivat tehdä testeistä vaikeasti ymmärrettäviä ja ylläpidettäviä. Monimutkaisempi logiikka on suotavaa pitää erillisissä kirjastoissa.

Näiden käytäntöjen noudattaminen auttaa varmistamaan, että Robot Frameworkin testit ovat selkeitä, tehokkaita ja helposti ylläpidettäviä, mikä parantaa koko ohjelmistokehitysprosessin laatua ja luotettavuutta.

4.5 Page Object Model -suunnittelumalli

Page Object Model (POM) on suunnittelumalli, joka parantaa testauskoodin ylläpidettävyyttä, uudelleenkäytettävyyttä ja luettavuutta. Tämä malli eriyttää testikoodin ja sivukohtaisen koodin, kuten käyttöliittymäelementtien paikantimet (eng. locators), mikä helpottaa muutosten hallintaa ja vähentää koodin toistoa [23].

Page Object Modelin perusperiaatteena on luoda jokaiselle verkkosivulle oma Page Object -luokka, joka toimii rajapintana sivun ja testikoodin välillä. Page Object -luokat sisältävät kaikki sivun elementit ja niihin liittyvät toiminnot, jolloin testikoodissa voidaan viitata näihin luokkiin ilman suoraa viittausta sivun elementteihin. Tämä lähestymistapa tekee testikoodista modulaarista ja helpottaa sen ylläpitoa.

Page Object Modelin käytöstä on useita hyötyjä, tässä joitakin niistä [23]:

- **Koodin uudelleenkäytettävyys**: POM mahdollistaa koodin uudelleenkäytön kirjoittamalla koodin kerran ja käyttämällä sitä eri testeissä myöhemmin. Tämä vähentää koodin määrää ja parantaa sen hallittavuutta.
- **Koodin ylläpidettävyys**: POM erottaa testikoodin ja sivukohtaisen koodin, mikä tekee koodin ylläpidosta helpompaa laadunvarmistajille (eng. QA-engineer). Testaajien tarvitsee tehdä muutoksia vain sivuobjektien luokkiin, mikä parantaa ylläpidettävyyttä ja vähentää koodin päällekkäisyyttä.
- **Objektivarasto**: Jokainen sivu tulisi määritellä Java-luokkana, ja sivun kentät määritellään rajapinnaksi tai jäseniksi. Tämä lähestymistapa parantaa koodin organisointia ja tekee siitä selkeämpää ja helpommin hallittavaa.
- **Koodin luettavuus**: POM parantaa koodin kokonaisluettavuutta erottamalla sivukohtaisen koodin ja testikoodin toisistaan. Tämä auttaa testaajia ja kehittäjiä ymmärtämään koodin rakenteen ja toiminnan paremmin.

Page Object Modelin käyttöönotto testausautomaatioprojektissa vaatii huolellista suunnittelua ja selkeitä toimintamalleja. On tärkeää varmistaa, että kaikilla testikoodiin vaikuttavilla tiimin jäsenillä on yhteinen ymmärrys POMin periaatteista ja että mallin mukaisia käytäntöjä noudatetaan johdonmukaisesti koko projektin ajan, jolloin lopputuloksen koodiarkkitehtuuri on yhtenevä.

Page Object Modelin käyttöönoton vaiheet:

- **Sivujen tunnistaminen**: Identifioi kaikki sivut, jotka tarvitsevat testauksen. Määritä jokaiselle sivulle oma Java-luokka, joka sisältää sivun toiminnot ja elementit.
- **Paikantimien määrittely**: Määritä sivun elementit paikantimina, kuten ID, nimi, CSS-valitsin tai XPath. Paikantimet tulee pitää keskitetysti omissa tiedostoissaan, jotta niiden hallinta on helpompaa.
- **Sivujen toiminnot**: Määritä sivun toiminnot metodina Java-luokassa. Näitä metodeja voidaan kutsua testikoodista suorittamaan toimenpiteitä sivuilla.
- **Testikoodin erottaminen**: Pidä testitapaukset erillään sivukohtaisesta koodista. Testitapausten tulisi keskittyä testaamaan sovelluksen toiminnallisuutta kutsumalla sivuluokkien metodeja.

Tämän diplomityön Browser-kirjastoa käyttävä toteutus soveltaa tyypillistä POM- toteutusta siten, että ei käytetä Java-luokkia. Tiedostot jaotellaan silti sivujen mukaan, mutta toiminnallisuus tapahtuu Robot Frameworkin avainsanoja käyttäen. Paikantimet puolestaan ovat erillisissä Python-tiedostoissa, jotka samaan tapaan on eritelty sivujen perusteella. Monimutkaisempi logiikka on vielä erillisessä Python-kirjastossa, johon on kirjoitettu sovellukselle tarpeellista lisälogiikkaa. Monimutkaisen logiikan eristys pois avainsa-

noista kuuluu luvussa 4.4 mainittuihin hyviin käytäntöihin.

Page Object Model on tehokas koodiarkkitehtuuri, joka parantaa testauskoodin uudelleenkäytettävyyttä, ylläpidettävyyttä ja luettavuutta. Tämä malli helpottaa muutosten hallintaa ja vähentää koodin päällekkäisyyttä, mikä tekee testausprosessista tehokkaamman ja luotettavamman. Page Object Modelin käyttö on suositeltavaa erityisesti suurissa ja monimutkaisissa projekteissa, joissa koodin hallittavuus ja ylläpidettävyys ovat kriittisiä tekijöitä.

Käytettäessä Page Object Model-suunnittelumallia yhdessä jatkuvan integraation (CI) ja jatkuvan toimituksen (CD) putkien kanssa voidaan saavuttaa erittäin tehokas ja luotettava testausprosessi, joka tukee nopeita kehityssyklejä ja jatkuvaa laadunvarmistusta. Tämä tekee Page Object Model -suunnittelumallista erityisen arvokkaan työkalun ketterässä ohjelmistokehityksessä, jossa muutokset ja parannukset ovat jatkuvia.

5. TESTAUKSEN TOTEUTUKSEN SUUNNITTELU JA TOTEUTUS

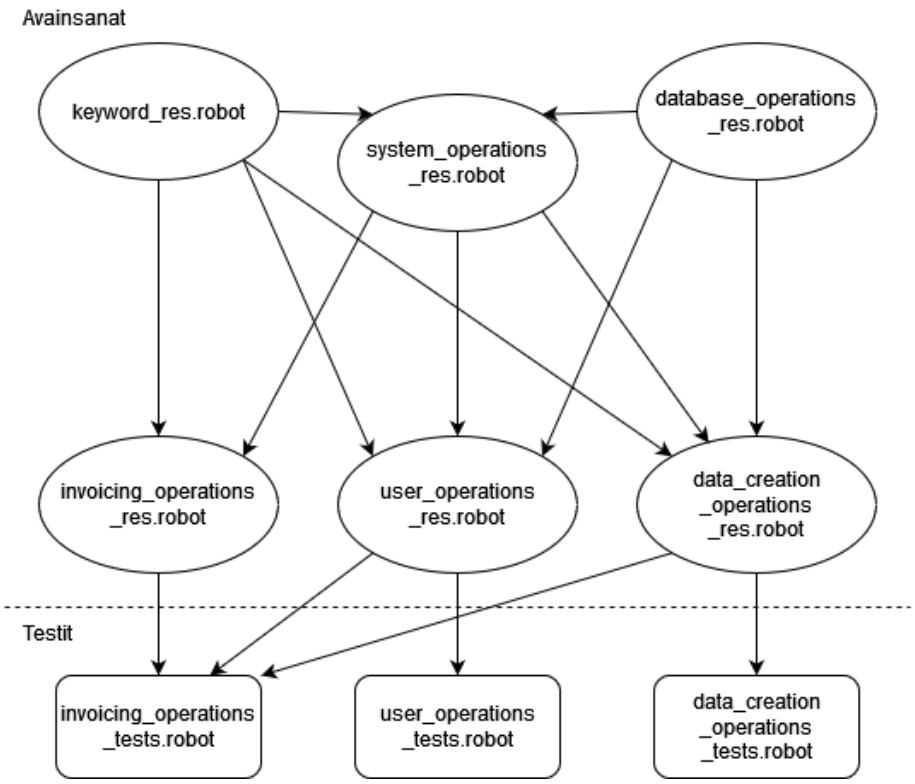
Tässä luvussa käsitellään testauksen toteutuksen suunnittelua ja sen vaiheita projektin testiautomaatiossa. Aluksi esitellään testattava ympäristö, joka on selainpohjainen laskutusjärjestelmä. Testauksen suunnittelussa keskityttiin järjestelmän kriittisiin toimintoihin, ja niille tehtäviin testeihin kiinnitettiin erityisen paljon huomiota. Testiprojektin tiedostorakenne toteutettiin Page Object Model (POM) -arkkitehtuurin mukaisesti, mikä mahdollistaa testikoodin selkeän ja ylläpidettävän rakenteen. Lisäksi testien kirjoituksessa hyödynnettiin avainsanoja ja paikantimia, jotka eriyttävät testilogiikan ja käyttöliittymän elementit.

Luvun toisessa osassa käsitellään testien suunnittelua ja toteutusta. Testien tarkoituksena oli korvata aiemmin SeleniumLibraryllä tehdyt testit uudemmilla, Browser-kirjastoon perustuvilla testeillä. Suunnittelun aikana pyrittiin säilyttämään testikattavuus ja optimoimaan suorituskyky. Testien toteutuksessa hyödynnettiin avainsanojen käyttöä, mikä lisäsi abstraktiotasoa ja selkeytti testikoodia.

5.1 Testattavan ympäristön esittely

Testauskohteena toimiva järjestelmä on laskutusjärjestelmä, joka on verkossa toimiva selainpohjainen sovellus (eng. web-application). Sovelluksia on kaksi, yksi on laskutuksen tekemiseen tarkoitettu ja toinen on portaali. Loppuasiakkaita varten on tehty portaali, josta he voivat tarkastella erilaista informaatiota, johon heidän laskunsa perustuvat. Pääsovellus puolestaan on yrityksen varsinaisille asiakkaille laskutuksen tekemiseen. Sekä portaalista ja pääsovelluksesta on olemassa kaksi ympäristöä eli versiota, joilla on paljon yhteistä koodia, mutta joiltain osin ne eriävät.

Nämä asiat on huomioitava testauksen toteutuksessa, sillä niiden kriittiset toiminnot ovat pääsääntöisesti eri kooditiedostoja. Tämä vaikuttaa omalta osaltaan jo testauksen laajuuteen, kun tavoitellaan kriittisten ominaisuuksien kattavuutta. Tämän seurauksena muusta ohjelman testauksesta joudutaan leikkaamaan. Koska ohjelman laajeneminen on mahdollista esimerkiksi ympäristöjen määrän kasvamisena, tulee myös huomioida sen seuraukset testiautomaation kannalta. Tässäkin työssä on tilanne, jossa ympäristöjen määrä tulee laajenemaan, mikä tarkoittaa runsaasti uusien testien kirjoittamista uudelle ympäris-



Kuva 5.1. *Vanhan testauksen import-suhteet*

tölle. Lisäksi ylläpidettävien testien määrä luonnollisesti kasvaa, mikä on erityisen tärkeää huomioida ketterissä ohjelmistoprojekteissa. Resurssien ollessa rajallisia tulee toteutuksen vastata yrityksen toivomuksia resurssien saatavuuden ja käytön kannalta. Tämän työn kannalta resursseja tulee olemaan hyvin vähän, koska yritys on kokoluokaltaan pieni.

Toteutettavaa automaatiotestausta havainnollistaa kuva 2.5, joka on tavallinen toteutus testiautomaatioputkesta (eng. test automation pipeline). Aikaisempaan verrattuna testiautomaatioputkeen ei tule muutoksia, lukuunottamatta pieniä konfiguraatiomuutoksia putken alustuksessa. Esimerkiksi siinä tulee muutoksen jälkeen asentaa Browser-kirjasto SeleniumLibraryn sijaan. Kuitenkin itse testit tullaan korvaamaan täysin uusilla, Browser-kirjaston kanssa yhteensopivilla testeillä.

Aiempaa SeleniumLibraryllä tehtyä toteutusta havainnollistaa kuva 5.1. Siinä ei ole käytetty paikantimille ollenkaan muuttujia, mikä vaikeuttaa projektin ylläpitoa. Esimerkiksi jonkun elementin muuttuessa tulee kaikki paikantimiin liittyvät viittaukset käydä muuttamassa, sen sijaan, että korjattaisiin muuttuja paikantimia sisältävässä tiedostossa. Lisäksi tiedostoja on importattu sekavasti järjestelmällisen lähestymisen sijaan. Viimeisenä haluttiin puuttua resurssi- ja testitiedostojen kokoon. Käyttäjäoperaatiotestit voidaan esimerkiksi pilkkoa pienempiin osiin, jolloin testitapauksia lisättäessä tiedostokoot eivät paisu vaikeasti hallittaviksi.

5.2 Ympäristön tuomat rajoitukset ja reunaehdot

Tässä työssä laskutusjärjestelmän testauksen ja kehityksen yhteydessä ympäristö asettaa tiettyjä rajoituksia ja reunaehtoja, jotka on otettava huomioon. Yksi keskeisimmistä prioriteeteista on tiedon eheys, mikä tarkoittaa sitä, että laskutustiedot säilyvät muuttumattomina järjestelmän eri vaiheissa. Tämä on kriittistä laskutoimitusten oikeellisuuden ja tietojen luotettavuuden varmistamiseksi. Laskutuksen yhteydessä pienetkin virheet voivat johtaa vakaviin seurauksiin, kuten taloudellisiin tappioihin tai asiakassuhteiden heikentymiseen, joten tietojen eheyden varmistaminen on ensisijaisen tärkeää.

Toinen huomionarvoinen prioriteetti on järjestelmän yleinen luotettavuus, mikä tarkoittaa, että järjestelmän on tuotettava oikeat tulokset ja sen on toimittava vakaasti eri käyttötilanteissa. Saatavuus on toki tärkeää, mutta tässä kontekstissa sen merkitys jää toissijaiseksi tiedon oikeellisuuden rinnalla. Saatavuuden merkitys on myös pienempää siitä syystä, että käyttäjämäärät ovat alhaisia. On usein tilanteita, joissa kukaan ei käytä kyseisen domainin sovellusta. Laskutusjärjestelmän tapauksessa luotettavuus ja eheys ovat tärkeämpiä, koska pääpaino on varmistaa, että laskutusprosessit suoritetaan tarkasti ja ilman virheitä.

Kehitystiimin yhteistyö testauksen luonnissa ja ylläpidossa on kriittistä näiden reunaehtojen varmistamisessa. Kun testiautomaatioon tulee uusia ominaisuuksia, on harkittava, tuleeko elementteihin lisätä uusia tunnistekenttiä (eng. identifier), jotka helpottavat testausta. Tämä voi tarkoittaa sitä, että elementtejä varten määritellään tunnisteet jo kehitysvaiheessa, jolloin ne ovat helposti käytettävissä testiautomaation yhteydessä. Vaihtoehtoisesti tunnisteet voidaan lisätä testejä kirjoitettaessa, mutta tämä voi hidastaa kehitysprosessia, jos elementtejä ei ole suunniteltu testattavuus mielessä.

Testausta suunniteltaessa on tärkeää huomioida myös testien ylläpidon helppous nopeasti muuttuvassa ympäristössä. Koska vaatimukset ja järjestelmän ominaisuudet voivat muuttua nopeastikin, testien ja avainsanojen täytyy olla riittävän geneerisiä, jotta niitä voidaan uudelleenkäyttää mahdollisimman laajasti ilman suurta määrää manuaalisia muutoksia. Avainsanojen abstrahointi ja modulaarisuus ovat avainasemassa tässä prosessissa. Geneeriset avainsanat ja uudelleenkäytettävät testirakenteet vähentävät koodin määrää ja toistojen tarvetta, mikä tekee testien ylläpidosta ja laajentamisesta joustavampaa. Tämä lähestymistapa ei ole suoraa testivetoista ohjelmistokehitystä (TDD), mutta testattavuus on silti otettava huomioon jo ominaisuuksia suunniteltaessa, jotta testien luominen ja ylläpito on helpompaa. Erityisesti, kun testausresursseja on käytössä rajallinen määrä, tulee testauksen kokonaisuuden olla vain sen kokoinen, jota voidaan ylläpitää.

Tärkeänä jatkokehityksen tavoitteena on varmistaa, että ympäristön asettamat reunaehdot otetaan huomioon kehitysprosessin kaikissa vaiheissa, erityisesti tiedon eheyden ja luotettavuuden osalta.

5.3 Testauksen lopputuloksen tavoiteltava taso

Tämän diplomityön tavoitteelle olennaista oli tietysti myös kuunnella yritystä. Palaveri pidettiin siksi, että yrityksen puolelta saatiin selvitettyä, minkälaisia odotuksia diplomityölle oli ja koska joihinkin teknisiin kysymyksiin tarvittiin vastauksia. Palaveriin osallistuivat toimitusjohtaja, diplomityön tekijä sekä yksi ohjelmistokehittäjä, jolla on kokemusta Robot Frameworkista ja automaatioputkista. Keskustelujen avulla pyrittiin määrittämään testauksen lopputuloksen tavoiteltava taso ja varmistamaan, että diplomityö vastaa sekä yrityksen että projektin tarpeisiin. Seuraavassa on esitetty keskeiset palaverissa käsitellyt kysymykset ja niiden pohjalta tehdyt päätökset.

Millainen on testauksen laajuus ja miten se määritellään? Testauksen laajuudeksi määriteltiin aluksi toteuttaa samat testit kuin aikaisemmin, jotta voidaan vertailla suoritusajassa ja koodirivien määrässä tapahtuneita muutoksia diplomityötä varten. Tämän jälkeen tavoitteena on lisätä joitakin tällä hetkellä puuttuvia, mutta tärkeinä pidettyjä testejä. Näin saavutetaan laajempi testikattavuus, mikä on yksi työn päätavoitteista.

Miten avainsanat, paikantimet ja testien tiedostorakenteet organisoidaan? Palaverissa päätettiin, että paikantimet ja avainsanat järjestetään jokaiselle sivulle omaksi tiedostoksi. Testien osalta sovittiin, että ne sijoitetaan tarpeen mukaan eri tiedostoihin sen perusteella, mitä ominaisuutta testi pohjimmiltaan testaa. Testien sijoittamisessa käytetään tulkintaa, jotta ne päätyvät tarkoituksenmukaisiin tiedostoihin ja säilyvät selkeinä.

Miten testidata käsitellään ja kuinka se parannetaan? Testidatan osalta sovittiin, että datan parannukset tehdään pääasiassa diplomityön jälkeen. Joitakin parannuksia voidaan toteuttaa kuitenkin jo nyt. Esimerkiksi voidaan hyödyntää olemassa olevia testejä, luoda uusia tapauksia tietokantaan tai käyttää snapshotteja estämään testauksen hidastumista.

Millaisia muutoksia putkeen tarvitaan ja miten siirtymä uusiin työkaluihin onnistuu? Putkeen tehdään tarvittavat muutokset asentamalla Browser Library SeleniumLibraryn sijaan. Tämä mahdollistaa uusien testikirjastojen käytön ja parantaa testauksen kattavuutta ja suorituskykyä.

Ajetaanko eri määrä testejä PR-vaiheessa (eng. Pull request), eli koodipäivityksen yhteydessä, verrattuna muihin vaiheisiin? Päätettiin, että alkuvaiheessa kaikki testit ajetaan PR:n yhteydessä. Tavoitteena on pitää testien suoritusaika alle 10 minuutissa. Tällä hetkellä testien suoritus kestää noin 3 minuuttia 30 sekuntia, mutta testikattavuuden lisääntyessä suoritusaika voi pidentyä, jolloin tarvitaan säätöä.

Miten kriittiset testit tunnistetaan ja ajetaan? Jos testien suoritusaika ylittää 10 minuuttia, käyttöön otetaan erilliset kriittisyystagit (eng. critical tags) tärkeimmille testeille. Tällöin vain kriittiset testit ajetaan PR:n yhteydessä, kun taas vähemmän kriittiset testit voidaan ajaa esimerkiksi yöaikaan, jotta varsinainen kehitysputki ei hidastu.

Näiden keskustelujen ja päätösten perusteella sovittiin, että testauksen lopputuloksen tavoiteltava taso on laaja kattavuus ja nopea suoritusaika, jotka varmistetaan käyttämällä järkevää testien organisointia ja kriittisyystasojen huomioimista.

5.4 Testausprojektin tiedostorakenne

Ennen kuin testejä alettiin kirjoittamaan, tuli suunnitella tiedostorakenne, joka vastaa haluttua Page Object Model -arkkitehtuuria. Lopputuloksena saatiin kuvan 5.2 mukainen rakenne, jossa testeihin tarvittavat resurssit, avainsanat, paikantimet ja räätälöidyt Python-skriptit on sijoitettu Resources-kansion alle.

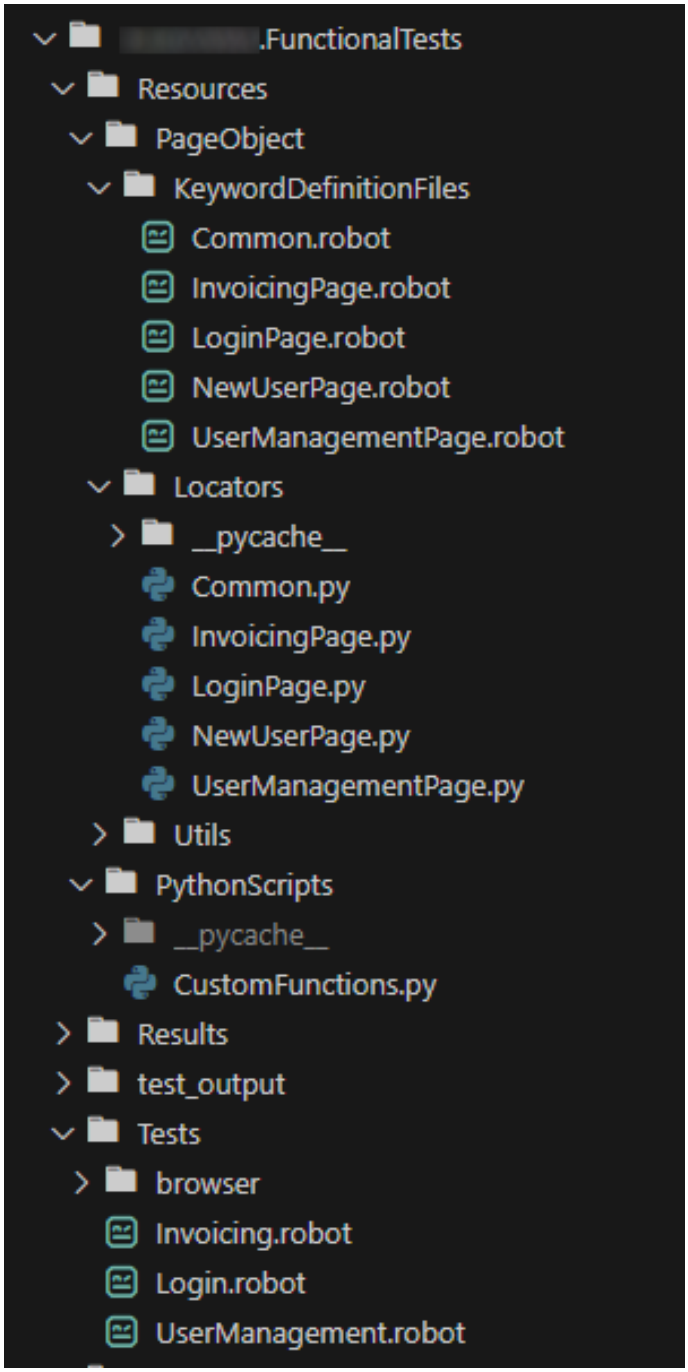
Kuten POM-mallissa on tyypillistä, jokaiselle testitiedostolle on oma vastaava paikannin ja avainsanatiedosto. Koska mallin mukaan testit tuli kohdistaa sivukohtaisesti, halutuille testitapauksille jouduttiin tekemään tulkintaa sen suhteen, mille sivulle pidemmät päästä päähän -testit kuuluvat. Esimerkkinä tästä on UserManagement.robot-testitiedosto. Siinä testataan käyttäjänhallintaan liittyviä operaatioita, joissa tulee myös navigoida uuden käyttäjän luontisivun kautta. Lisäksi testissä halutaan varmistaa käyttäjän toiminnallisuus kirjautumalla sisään luodulla tai muokatulla käyttäjällä. Tämän seurauksena tuli käyttää harkintaa, onko testi Login-, NewUser-, vai UserManagement-testi.

Kaikille sivuille ei ole testejä, mutta niiden kautta navigoidaan, jolloin POM-mallin mukaisesti niille tulee tehdä paikannin- ja avainsanatiedostot. Näissä tiedostoissa on elementtien paikantimet sekä toiminnallisuus, joita voidaan helposti hyödyntää testeissä. Tämä selittää, miksi Keyword- ja Locators-kansioissa on kansiota kohden enemmän tiedostoja kuin testitiedostoja. Lisäksi yleisiä testeissä tarvittavia toimintoja, kuten kirjautumista ja navigointia, varten lisättiin Common.robot-avainsanatiedosto sekä Common.py-paikannintiedosto. Näiden avulla saadaan vähennettyä toistoa.

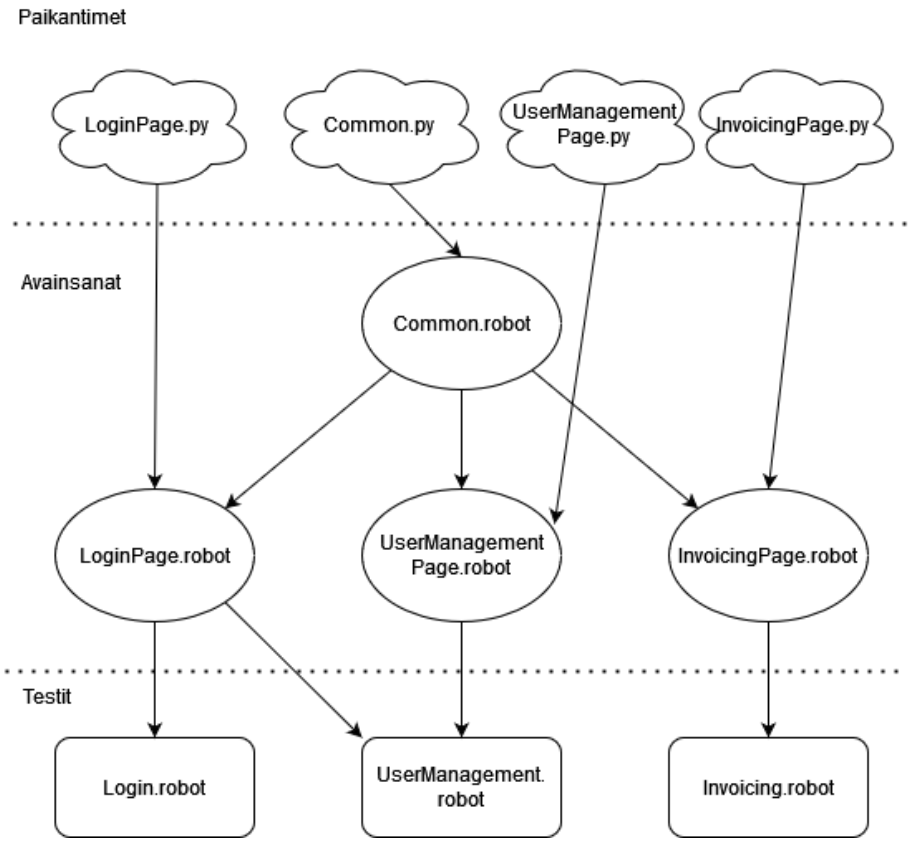
Paikantimien, avainsanojen ja testien suhde esitetään kuvassa 5.3. Siitä nähdään, kuinka avainsanatiedostot saavat tarvittavat elementtien tunnisteet sivukohtaisista Python-tiedostoista. Testit käyttävät sitten näitä avainsanatiedostoja, jotta testit olisivat helppolukuisia, lyhyitä ja ylläpidettäviä.

5.5 Testien suunnittelu ja kirjoitus

Tässä luvussa kerrotaan missä järjestyksessä testit kirjoitettiin, kuinka paljon niitä kirjoitettiin ja minkälaisia testejä kirjoitettiin. Suunnitelma oli aloittaa korvaamalla SeleniumLibraryllä toteutetut aiemmat testit. Nämä korvaavat testit ovat Browser-kirjastolla toteutettuja, mutta samaa käyttötapausta testaavia testejä. Testeissä pyrittiin testaamaan mahdollisimman pitkälle samoja asioita siten, että käytettiin samoja polkuja lopputulokseen pääsemiseksi. Näin toimittiin siksi, että esimerkiksi testien nopeuden tai koodimäärän vertailu olisi reilua. Kuitenkin joitain redundantteja tarkistuksia poistettiin, jos niitä ei koettu erityisen



Kuva 5.2. *Työssä käytetty page object model -tiedostorakenne*



Kuva 5.3. *Tiedostorakenteen import-suhteet*

tärkeäksi testien kannalta.

Lisäksi näihin testeihin tehtiin erilaisia parannuksia luvun 4.4 suosituksia myötäillen. Erityisesti keskityttiin siihen, onko vanhassa testissä sopiva abstraktiotaso, pituus, tarkistusten määrä ja sijainti sekä luotettava ja robusti totetutus. Nämä kaikki vaikuttavat työn lopputulokselta tavoiteltaviin ominaisuuksiin.

Testin toteuttajan vastuulle jäi arviointi siitä, mikä on testille sopiva abstraktiotaso. Tällä tarkoitetaan sitä, kuinka yksityiskohtaisesti testi on kirjoitettu. Lisäämällä avainsanojen käyttöä saadaan lisättyä abstraktiotasoa ja sen seurauksena lyhyempiä testejä. Toisaalta liian korkea abstraktiotaso hankaloittaa testien lukemista. Testiautomaatioputken tuottamassa raportissa saa toki avattua myös avainsanojen sisällä tapahtuvat vaiheet, joten se ei tuota suurempia ongelmia. Sopivaksi abstraktiotasoksi havaittiin yksi yhteen suhde. Tällä tarkoitetaan sitä, että yhden avainsanan abstraktiotasoksi pyrittiin jättämään yksi toiminnoksi luokiteltava vaihe. Näin päästä päähän testeissä käytettäisiin useaa avainsanaa ja tarkistukset näkyisivät erikseen. Abstraktiotason tarkoituksena on pohjimmiltaan tehdä testistä helppolukuisia. Kuitenkin yksittäiselle testille relevantit kohdat on syytä kuvata yksityiskohtaisemmin ilman liian korkeaa abstraktiotasoa, koska testi keskittyy tähän alueeseen.

Testien kirjoittamista tehtiin iteratiivisesti. Tällä tarkoitetaan sitä, että myöhemmissä vai-

```
Input Username
    [Arguments]    ${USERNAME}
    Fill Text    ${INPUT_EMAIL}    ${USERNAME}

Load in Interactive Console
Input Password
    [Arguments]    ${PASSWORD}
    Fill Text    ${INPUT_PASSWORD}    ${PASSWORD}

Load in Interactive Console
Login With Credentials
    [Arguments]    ${USERNAME}=${DEFAULT_USER_EMAIL}    ${PASSWORD}=${DEFAULT_USER_PASSWORD}    ${SHOULD_LOGIN}=True
    Input Username    ${USERNAME}
    Input Password    ${PASSWORD}
    Click    ${LOGIN_BUTTON}
    User Is Logged In    ${SHOULD_LOGIN}
```

Kuva 5.4. *Huonoksi havaittu abstraktiotaso*

heissa palattiin taaksepäin arvioimaan aikaisempaa toteutusta. Jos aiempi toteutus poikkesi hyviksi havaituista käytänteistä tehtiin kyseiseen kohtaan parannuksia. Näin työ säilyi rakenteeltaan ja tavoitteeltaan yhtenäisenä. Esimerkkinä huonosta abstraktiotasosta on kuva 5.4. Siinä on turhaan tehty omat avainsanat käyttäjänimen ja salasanakenttien täyttämiselle. Yhtä selkeää on korvata avainsanan Login With Credentials sisälle:

```
Fill Text    ${INPUT_EMAIL}    ${USERNAME}
```

Sen sijaan, että käytettäisiin avainsanaa ja sisällä lukisi:

```
Input Username    ${USERNAME}
```

Muuttujat ovat merkittynä seuraavalla merkinnällä:

```
${MUUTTUJA_NIMI}
```

Merkintöjen hyvällä nimeämisellä voidaan vähentää avainsanojen tarvetta. Näin on toimittu juuri tässäkin esimerkissä.

Luvussa 4.1 on esitetty Robot Frameworkin mahdollistama datavetoinen syntaksi. Sen avulla voidaan tehdä tehokkaasti eri muuttuja-arvoilla tapahtuvaa testausta. Näin voidaan saada parempaa kattavuutta ohjelman toiminnoille, joissa muuttujien arvot voivat hyvinkin vaikuttaa ohjelman oikeaan toiminnallisuuteen. Datavetoisesta testistä esimerkkinä toimii kuva 5.5. Kuvassa ylempänä on avainsana, joka toimii testissä käytettävänä templaattina. Tämä toimii niin, että templaatti ajetaan eri muuttuja-arvoilla siten, että yhden testin argumentit ovat yhdellä rivillä.

Datavetoiset testit ovat siis tehokkaita koodin määrän hallinnassa, mikä helpottaa ymmärrettävyyttä ja selkeyttä. Tällaisia testejä pyrittiin käyttämään aina, kun se vain olisi mahdollista ja järkevää. Datavetoisia testejä on helppo muuttaa jälkikäteen, jos jotkin muuttujayhdistelmät ovat turhia tai joitakin puuttuu. Toisaalta datavetoisissa testeissä on se ongelma, että verkkoa käyttävässä selainympäristöissä ne vievät paljon aikaa saman toiminnallisuuden testaamiseen. Yleensä tällaiset tapaukset kannattaa kattaa yksikkötes-

```
*** Keywords ***
Load in Interactive Console
Login Test Template
    [Documentation]    Test logging in with different credentials and validate the login status.
    [Arguments]    ${USERNAME}    ${PASSWORD}    ${SHOULD_LOGIN}
    Set Up Browser
    Login With Credentials    ${USERNAME}    ${PASSWORD}    ${SHOULD_LOGIN}
    Close Browser

You, 14 seconds ago | 1 author (You)
*** Test Cases ***
Run | Debug | Run in Interactive Console
Test Valid And Invalid Logins
    [Documentation]    Implemented by using Data Driven Approach
    [Template]    Login Test Template
    ${VALID_USER}    ${VALID_PASSWORD}    True
    ${VALID_USER}    ${INVALID_PASSWORD}  False
    ${INVALID_USER}  ${INVALID_PASSWORD}  False
    ${INVALID_USER}  ${VALID_PASSWORD}    False
```

Kuva 5.5. *Datavetoinen kirjautumistesti*

teillä, jotka ovat huomattavasti nopeampia suorittaa verrattuna käyttöliittymää navigoiviin testeihin.

5.6 Ympäristön ja putken konfigurointi

Ympäristön ja putken konfigurointi oli ratkaiseva vaihe automaatiotestauksen toiminnan varmistamisessa. Lokaalissa ympäristössä funktionaalisen testauksen suorittamiseksi oli tarpeen lisätä uusi VSCode debug & run -vaihtoehto, joka mahdollisti testien ajamisen suoraan kehitysympäristöstä. Tämä toteutettiin muokkaamalla `launch.json`-tiedostoa lisäämällä siihen seuraavat parametrit:

- **Nimi**: `"name": "Robot Tests: (local) Run All (Browser library)"`
- **Tyyppi**: `"type": "PowerShell"`
- **Pyyntö**: `"request": "launch"`
- **Skripti**:

  ```
  "script": "python -m robot --pythonpath . -V ..\Variables\
      local_variables.py -x outputxunit.xml -d test_output *.
      robot"
  ```

- **Työhakemisto**:

  ```
  "cwd": "${workspaceFolder}/tests/ (sensuroitu).
      FunctionalTests/Tests"
  ```

Näiden parametrien lisääminen mahdollisti testien helpon suorittamisen painamalla ajo-vaihtoehtoa kuvan 5.6 mukaisesti, mikä yksinkertaisti testaajien ja kehittäjien työskentelyä ja vähensi manuaalisesti ajettavien komentojen tarvetta.

Kuva 5.6. *VSCoden uusi käynnistysvaihtoehto*

Putken konfigurointiin tarvittiin muutoksia, jotta Browser-kirjastolla tehdyt testit toimivat oikein putkessa. SeleniumLibrarystä siirryttäessä tehtiin seuraavat muutokset:

- Robot Frameworkin `Browser`-kirjasto asennettiin korvaamaan vanha `Selenium`-kirjasto.
- Node-riippuvuudet lisättiin suorittamalla komento `script: rfbrowser init`, jotta Browser-kirjasto saatiin toimimaan oikein.
- `SeleniumLibrary` ja `chromedriver` poistettiin tarpeettomina.
- Testien ja testilokien tallennuspaikat päivitettiin uusille hakemistoille, mikä mahdollisti selkeämmän rakenteen ja testitulosten hallinnan.
- `Run Critical Tests` -vaihe päivitettiin `Run Functional Tests` -vaiheeksi, koska testit eivät enää olleet jaoteltuja kriittisiin ja muihin testeihin, vaan kaikki funktionaaliset testit ajettiin yhtenä kokonaisuutena.

Näiden parametrien korjaaminen mahdollisti sen, että putki toimii nyt täysin uudella `Browser`-kirjastolla. Toimivan putken avulla varmistettiin, että testit voidaan ajaa luotettavasti ja tehokkaasti osana jatkuvaa integraatiota.

5.7 Haasteet toteutuksessa

Testeissä kohdattiin useita haasteita. Yksi merkittävimmistä oli paikantimien määrittäminen, erityisesti silloin, kun verkkosivun elementeiltä puuttui `id`-tunniste, mikä vaikeutti niiden paikantamista. Toisinaan joillakin sivuilla oli käytetty samaa id:tä kahteen kohtaan, jonka seurauksena toista piti muuttaa, jotta elementtiä voitaisiin käyttää testissä. Tämän lisäksi jatkuvat muutokset koodissa aiheuttivat sen, että myös ennen testien valmistumista jouduttiin tekemään muutoksia olemassa oleviin testeihin. Jatkuva ylläpito on kuitenkin osa ketterää ohjelmistokehitystä, joten siksi on tärkeää tehdä testeistä helposti ylläpidettäviä.

Yksi toinen merkittävä haaste oli tiedostojen välisten import-suhteiden hallinta. Tavoitteena oli hyödyntää olemassa olevia avainsanoja muissa tiedostoissa, mutta tämä johti helposti kiertäviin import-suhteisiin (eng. circular imports). Ratkaisuna oli viitata useaan avainsanatiedostoon ainoastaan testeistä. Vaikeaa oli myös löytää sopivat testi- ja testisarja-alustus sekä -purkaminen, kun pyrittiin yhdistämään *datavetoisia*- ja *avainsanavetoisia*-lähestymistapoja samoihin testeihin. Tähän onneksi oli olemassa käypä ratkaisu: alustuksen ja purkamisen ylikirjoitus datavetoisissa testeissä.

CI/CD-putken debuggaus osoittautui myös haastavaksi, koska putkiajojen suorittaminen

vei huomattavasti aikaa. Muutosten jälkeen oli aina odotettava, että muut vaiheet suoritettiin ennen funktionaalisten testien ajoa. Putkessa käytetyt Pythonin ja Robot Frameworkin versiot erosivat paikallisesta ympäristöstä, mikä saattoi johtaa erilaisiin testituloksiin. Tämän havaitsemisen ja korvaamisen jälkeen testien tulokset olivat lähempänä toisiaan. Verkkoyhteyksien käyttö putkiajossa toi mukanaan lisää vaihtelua, sillä esimerkiksi JavaScript-tiedostojen lataus ei aina tapahtunut samalla nopeudella.

5.8 Automaatiotestaus osana ohjelmistokehitystä

Yrityksessä käytetään ketterää ohjelmistokehitysmallia, jonka tavoitteena ovat nopeat päivityssyklit. Uusia päivityksiä pyritään julkaisemaan 2-4 viikon välein. Kuitenkin puutteellisen automaattisen regressiotestauksen vuoksi huomattava osa ohjelmistokehitysresursseista kuluu regressiotestaukseen ennen uusien päivitysten julkaisemista. Tämä johtaa siihen, että ohjelmistokehitysajasta yhä suurempi osuus menee manuaaliseen testaukseen päivitysten tihentyessä.

Diplomityön päätavoitteena on ollut parantaa järjestelmätason regressiotestausta, jotta manuaaliseen testaukseen kuluva aika vähenisi. Tämä parannus mahdollistaisi tiheämmät ja luotettavammat ohjelmistopäivitykset, sillä automaattisesti suoritetut regressiotestit varmistaisivat ohjelmiston laadun jokaisen koodimuutoksen jälkeen. Tällöin kehitystiimi voisi olla varmempi siitä, että päivitykset eivät tuo mukanaan uusia virheitä tai ongelmia tuotantoympäristössä.

Yksi keskeinen haaste automaatiotestauksen käyttöönotossa on se, että testien toimintavarmuus vaihtelee ympäristöstä riippuen. Tämän vuoksi olisi suotavaa testata ohjelmiston toimintaa mahdollisimman todellisessa ympäristössä tai sen tarkassa kopiossa. Kysymys kuuluukin, tuleeko yrityksen hankkia erillinen, anonymisoitu kopio tuotantoympäristöstä testiä varten, vai voidaanko hyödyntää yrityksen olemassa olevia resursseja, kuten esimerkiksi staging-ympäristöä ja staattista testidataa? Erityisesti testauksen ympäristöjen hallinta on tärkeää, jotta voidaan varmistaa, että testitulokset ovat luotettavia ja vertailukelpoisia todellisen käytön kanssa.

Testidatan ja testikoodin säilyttäminen yhtä huolellisesti kuin muun koodin on myös olennainen osa tehokasta automaatiotestausta. Tämä tarkoittaa, että testiskriptit ja niiden ylläpitoa tulisi versioida ja hallita samanlaisin käytännöin kuin tuotantokoodia. Tällä tavoin voidaan taata testien jäljitettävyys ja mahdollistaa niiden helppo päivitettävyys, kun ohjelmistoon tehdään muutoksia.

Automaattinen testien generointi voisi myös tarjota uusia mahdollisuuksia vähentää manuaalisen työn määrää. Automaattisten testiskenaarioiden luonti, esimerkiksi tekoälyn tai muiden työkalujen avulla, mahdollistaisi entistä nopeamman ja tehokkaamman testien kirjoittamisen ja ylläpidon. Näiden testien luomisessa tekoälystä havaittiin olevan jo mer-

kittävästi apua eri vaiheissa. Tämä voi olla erityisen hyödyllistä järjestelmätason testauksessa, jossa yksittäiset testit voivat olla laajoja ja kattavat monimutkaisia käyttötilanteita.

Näin ollen automaatiotestaus ei pelkästään vähennä manuaalisen testauksen tarvetta, vaan myös parantaa ohjelmiston laatua ja nopeuttaa julkaisusyklejä, jolloin yritys pystyy reagoimaan ketterämmin markkinoiden ja asiakkaiden muuttuviin tarpeisiin.

6. RATKAISUN ARVIOINTI

Tässä luvussa arvioidaan ratkaisun toimivuutta suhteessa työn johdannossa esitettyihin tavoitteisiin 1.1: testauksen luotettavuuteen, ylläpidettävyyteen ja tehokkuuteen. Arvioinnin pohjalta selvitetään, kuinka hyvin ratkaisu on vastannut näihin vaatimuksiin ja miten se on parantanut testausprosessia. Ensimmäisessä osassa käsitellään testauksen luotettavuutta ja sen mittaamista eri tavoilla. Tässä yhteydessä tarkastellaan sitä, miten testien riittävä kattavuus varmistetaan ja kuinka CI/CD-putken toistuvilla testiajoilla voidaan havaita mahdollisia vääriä negatiivisia tuloksia.

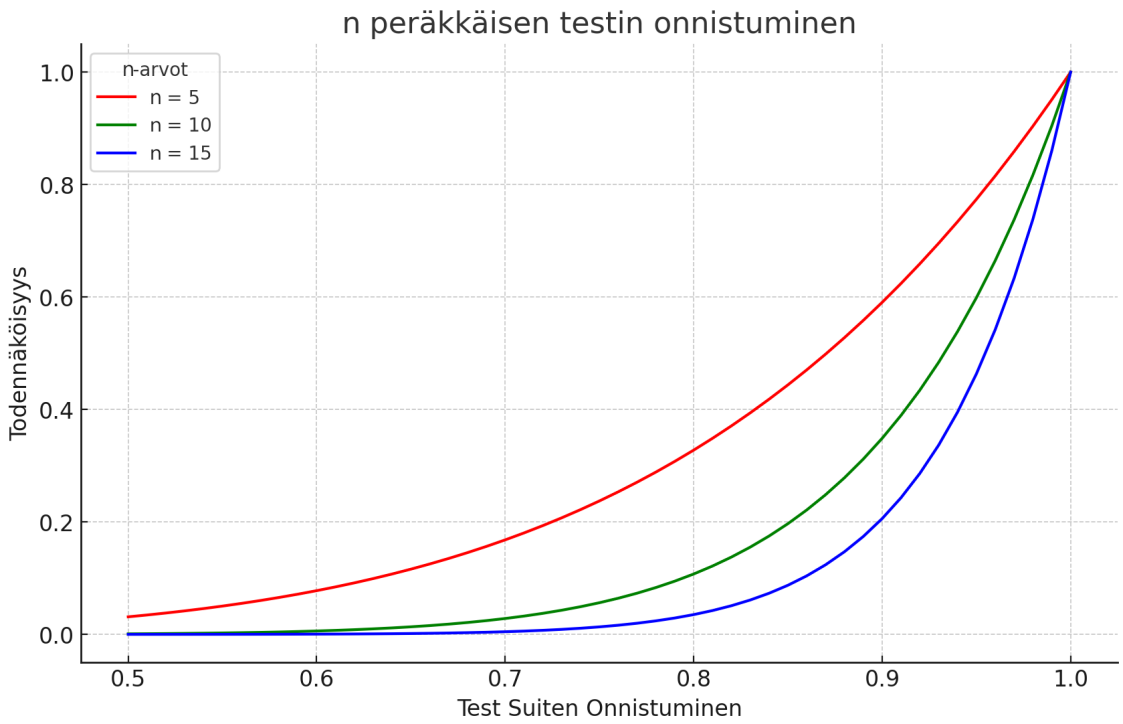
Lisäksi keskitytään testiautomaation ylläpidettävyyteen ja yksinkertaisuuteen. Ratkaisun tarkoituksena oli vähentää testien monimutkaisuutta sekä parantaa niiden ylläpidettävyyttä käyttämällä Page Object Model -arkkitehtuuria. Tämä mahdollistaa sen, että testikoodi pysyy selkeänä ja modulaarisena, mikä helpottaa tulevien muutosten ja lisäysten toteutusta. Lisäksi arvioidaan, kuinka uudemman Browser-kirjaston käyttö vaikutti suoritusnopeuteen verrattuna aiempaan SeleniumLibraryyn, ja miten parannettujen avainsanojen hyödyntäminen yksinkertaistaa testikoodia. Ratkaisun tehokkuus tulee esille erityisesti siinä, kuinka testit pystytään suorittamaan nopeammin ja vähemmällä koodimäärällä ilman, että menetetään testauksen luotettavuutta.

6.1 Testauksen luotettavuuden mittarit

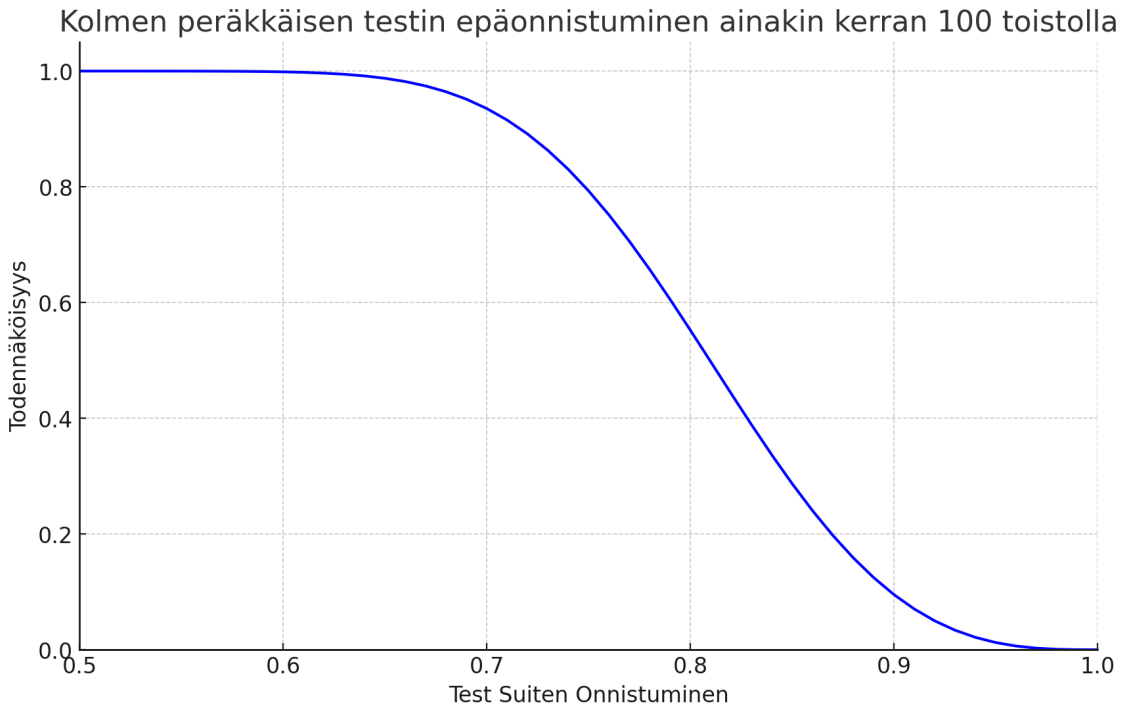
Testauksen luotettavuuden varmistaminen on ratkaisevaa, sillä testauksen tehtävänä on havaita virheitä mahdollisimman aikaisessa vaiheessa. Tässä osiossa arvioidaan ratkaisun luotettavuutta erityisesti testikattavuuden ja vääriin negatiivisiin tuloksiin liittyvien mittareiden avulla.

Testien toimintavarmuudesta ei koskaan voi olla täysin varma. Siksi putkessa ajetaan funktionaaliset testit uudelleen, jos aiemmalla kerralla jokin testi epäonnistui. Näin tehdään kolmanteen kertaan asti. Jos kaikki kolme kertaa epäonnistuvat, niin koodimuutosta ei hyväksytä. Ennen vanhojen testien korvaamista haluttiin selvittää uusien testien toimintavarmuus. Tätä varten uusia testejä ajettiin toistuvasti samalla datalla.

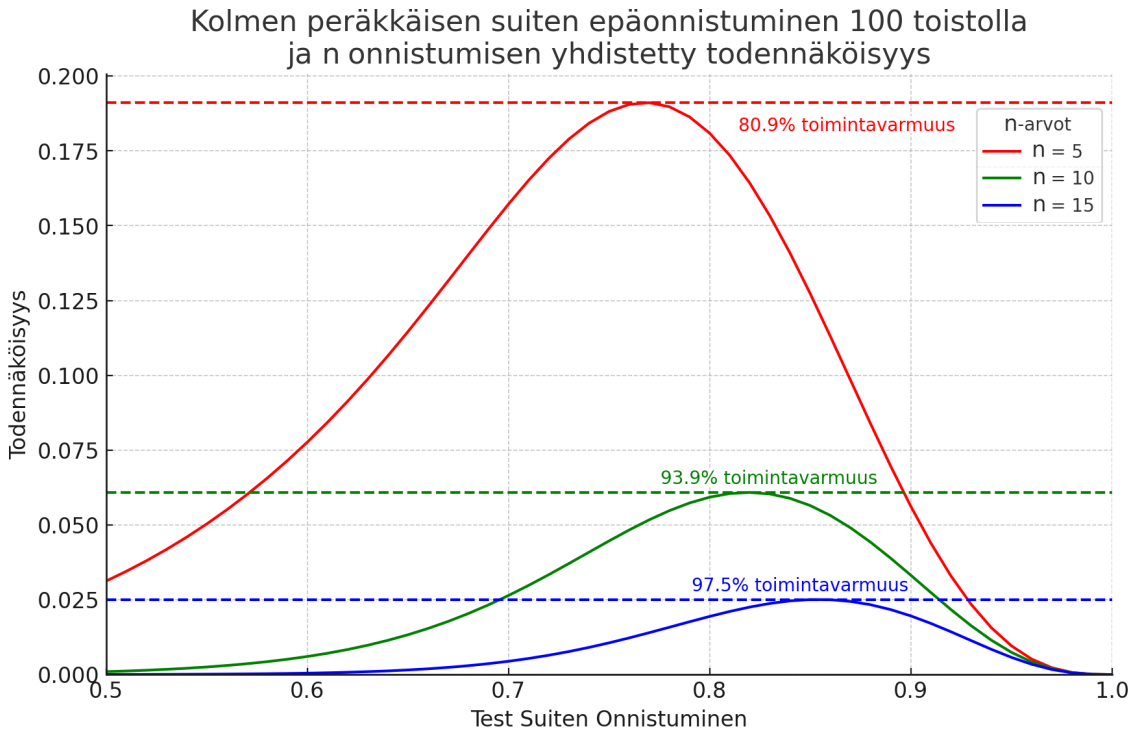
Testin epäonnistumisen todennäköisyyttä ei kuitenkaan voitu tietää, ja sen mallintamiselle käytettiin muuttujaa. Eri muuttujan arvoilla saatiin laskettua eri todennäköisyyksiä.



Kuva 6.1. *Todennäköisyys, että n testiajoa onnistuu peräkkäin eri onnistumisen todennäköisyyksillä*



Kuva 6.2. *Kolmen peräkkäisen epäonnistumisen todennäköisyys eri testin onnistumisen todennäköisyyksillä*



Kuva 6.3. *Yhdistetty todennäköisyys eli testauksen luotettavuuden taso*

Tavoitteena oli laskea pahin tapaus -todennäköisyys, minkä perusteella voidaan varmaksi sanoa testauksen luotettavuuden taso. Vastaus, joka haluttiin selvittää, oli: "Millä todennäköisyydellä testisarja ei läpäise putken kolmea peräkkäistä testiajoa, mutta läpäisee testisarjan testauksen onnistuneesti?". Tälle voitiin piirtää kuva 6.3, josta nähdään minkälainen varmuus voidaan luvata, jos esimerkiksi 10 testiajoa menee läpi peräkkäin testien testauksessa. Kuvan perusteella voidaan sanoa vähintäänkin 94% todennäköisyydellä, että seuraavassa 100:ssa putken ajossa sama testisarja ei tule epäonnistumaan kolmea kertaa peräkkäin. Se tarkoittaa, että väärän negatiivisen tuloksen todennäköisyys yhdelle testisarjalle 100 ajossa on korkeintaan 6%. Hyvänä tasona voidaan pitää yli 90%, mutta sopiva taso on kuitenkin tapauskohtainen, riippuen testattavan kohteen kriittisyydestä.

Tässä on tärkeää kuitenkin huomata ero paikallisesti lokaalin eli paikallisen testauksen ja verkkoyhteyksiä käyttävän testauksen välillä. Verkkoyhteydet tuovat mukanaan lisää muuttujia testiajoihin. Esimerkiksi jotkin javascript-tiedostot eivät ehkä lataudu ajoissa, mikä voi aiheuttaa napin painalluksen toimimattomuuden. Tämän takia eri ympäristössä tapahtuvan testauksen toimintavarmuutta on vaikea arvioida etukäteen. Todellisuudessa kuvaaja esittääkin samassa ympäristössä tapahtuvan testauksen toimintavarmuutta.

Robot Framework Browserissa on onneksi avainsanoja, joilla tällaisia ajoitusongelmia voidaan vähentää. Avainsana Wait For Load State, argumentilla domcontentloaded, odottaa, kunnes HTML-sivu saavuttaa DOMContentLoaded-tilan. Tämä tarkoittaa, että selain on ladannut kaiken HTML-sisällön ja rakentanut DOM-puun (Document Object Model), mut-

ta sivun ulkoiset resurssit, kuten kuvat <img> ja tyylitiedostot, eivät välttämättä ole vielä latautuneet. [12]

Koska putken avulla testaus vie paljon aikaa, päätettiin todellista testausta simuloida käyttämällä putken käyttämää tietokantaa ja verkkosivua. Näin saatiin realistisempia testaustuloksia ja sen avulla voitiin arvioida putken todellista toimintavarmuutta. Tuloksissa havaittiin, että toimintavarmuus putkessa on huonompi, mutta parannuksia tekemällä verkkoyhteydestä aiheutuvaa vaihtelua saatiin vähennettyä.

6.2 Testiautomaation ylläpidettävyys ja yksinkertaisuus

Ylläpidettävyyden arviointi on keskeinen osa ratkaisun toimivuutta, sillä projektin pitkäikäisyyden ja skaalautuvuuden kannalta koodin yksinkertaisuus ja modulaarisuus ovat ratkaisevia. Tässä osiossa tarkastellaan, kuinka Page Object Model ja avainsanojen parantaminen ovat vaikuttaneet testikoodin ylläpidettävyyteen.

Taulukko 6.1 kertoo koodirivien määrän muutoksista vanhan ja uuden toteutuksen välillä, keskittyen vanhoihin ja vastaaviin uusiin testikoodiriveihin. Kuten taulukosta nähdään, koodirivejä on saatu merkittävästi vähennettyä. Tämä parantaa koodin ylläpidettävyyttä, sillä turha toisto ja tarpeettomat kommentit on poistettu tehokkaiden avainsanojen avulla. Paikannintiedostojen lisääminen tosin kasvatti kokonaiskoodimäärää, mutta niiden avulla refaktorointi helpottui, mikä on erityisen tärkeää ketterässä ohjelmistokehityksessä.

Kategoria	**Vanha**	**Uusi**	**Muutos (%)**
Avainsanat	1172	469	-59.98
Testit	274	149	-45.62
Avainsanat + Testit	1446	618	-57.26
Paikantimet	0	94	NaN
Avainsanat + Testit + Paikantimet	1446	712	-50.76

Taulukko 6.1. *Vanha toteutus ja uusi vertailukelpoinen toteutus: Avainsanojen, testien ja paikantimien rivimäärien vertailu sekä niiden muutokset prosentteina.*

Taulukko 6.1 on rajoitettu vertailu, joka kattaa vain vanhojen ja vastaavien uusien testien koodirivimäärät. Tämä osoittaa, että testien toteuttaminen voidaan optimoida hyödyntämällä Browser-kirjastoa ja muita parannuksia. Esimerkiksi SeleniumLibraryyn verrattuna Browser-kirjaston Click-avainsana hoitaa automaattisesti elementin odottamisen ja vierittämisen näkyville, mikä säästää koodirivejä ja yksinkertaistaa koodia [17]. Kuvassa 6.4 on esimerkki siitä, miten SeleniumLibraryssa käytetään useampia rivejä toimintaan, joka Browser-kirjastossa tehdään yhdellä rivillä.

Kun uusia testejä toteutettiin, tavoitteena oli tehdä niistä mahdollisimman tehokkaita ja helposti ylläpidettäviä. Tämä näkyy seuraavassa taulukossa 6.2, jossa on esitetty alku-

```
IF     '${TAB}' ==
    Scroll Element Into View    xpath://a[
    Wait Until Element Is Visible    xpath
    Click Element    xpath://a[@href=
ELSE IF     '${TAB}' ==
    Scroll Element Into View    xpath://a[
    Wait Until Element Is Visible    xpath
    Click Element    xpath://a[@href='
ELSE IF     '${TAB}' ==
    Scroll Element Into View    xpath://a[
    Wait Until Element Is Visible    xpath
    Click Element    xpath://a[@href=
ELSE IF     '${TAB}' ==
    Scroll Element Into View    xpath://a[
    Wait Until Element Is Visible   xpath:
    Click Element    xpath://a[@href=
ELSE IF     '${TAB}' ==
    Scroll Element Into View    xpath://a[
    Wait Until Element Is Visible   xpath:
    Click Element    xpath://a[@href=
ELSE IF     '${TAB}' ==
    Scroll Element Into View    xpath://a[
    Wait Until Element Is Visible   xpath:
    Click Element    xpath://a[@href=
END
```

Kuva 6.4. *SeleniumLibrary-esimerkki avainsanoista, jotka on Browser-kirjastossa yhdistetty*

peräisen SeleniumLibrary-toteutuksen koodirivit verrattuna uuteen lopulliseen Browser-kirjastolla tehtyyn toteutukseen. Tässä taulukossa on eritelty koodimäärä sekä ilman paikantimia että paikantimien kanssa. Paikantimien erottelu omiin tiedostoihinsa kasvattaa koodin määrää, mutta helpottaa huomattavasti koodin ylläpitoa.

Kategoria	**Vanha**	**Uusi**	**Muutos (%)**
Avainsanat	1172	1221	4.18
Testit	274	716	161.31
Avainsanat + Testit	1446	1937	33.95
Paikantimet	0	377	NaN
Avainsanat + Testit + Paikantimet	1446	2314	60.04

Taulukko 6.2. *Vanha toteutus ja uusi lopullinen toteutus: Avainsanojen, testien ja paikantimien rivimäärien vertailu sekä niiden muutokset prosentteina.*

Uusien testien lopullinen toteutus kolminkertaisti testien määrän verrattuna alkuperäiseen, mikä näkyy myös koodirivien määrässä. Vaikka paikantimien lisääminen kasvatti rivimäärää, uusi toteutus yksinkertaisti koodia merkittävästi ja teki siitä paremmin ylläpidettävän. Avainsanojen tarkastuksia ja tarpeettomia elementtejä karsittiin, jolloin koodi keskittyy olennaiseen toiminnallisuuteen.

Uusia testitapauksia tuli lopullisessa toteutuksessa kolmikertainen määrä alkuperäiseen verrattuna. Se näkyy myös testitiedostojen koodirivien kokonaismäärässä, kuitenkin itse testeissä tarvittavat avainsanat saatiin tehtyä paljon tehokkaammin. Lopullisessa toteutuksessa oli avainsanatiedostoissa lähes sama määrä koodirivejä kuin alkuperäisessä, vaikka testien määrä oli vain kolmannesosa.

Uudessa toteutuksessa on yhteensä yhdeksän testisarjaa ja 33 yksittäistä testiä. Avainsanat ja paikantimet on jaettu selkeästi omiin tiedostoihinsa, jolloin avainsanojen tiedostoja on 16 ja paikantimien tiedostoja myös 16. Tämä rakenne tukee paremmin koodin ylläpidettävyyttä ja joustavuutta, sillä se mahdollistaa paikantimien erillisen hallinnan ja muokkaamisen ilman, että testien tai avainsanojen logiikkaan tarvitsee puuttua. Tällainen jaottelu helpottaa myös projektin skaalautuvuutta, sillä koodia voidaan laajentaa ja päivittää vaikuttamatta muihin komponentteihin.

Merkittävä parannus saatiin aikaan myös poistamalla turhat tarkastukset avainsanoista. Avainsanojen tulisi keskittyä ainoastaan toiminnallisuuteen, ja tarkistukset, kuten elementtien saatavuuden tai oikeellisuuden varmistaminen, kuuluvat testeihin. Tämä ero selkeyttää koodia ja jakaa vastuut paremmin: avainsanat toteuttavat toiminnallisuuden, kun taas testit vastaavat siitä, että nämä toiminnot toimivat odotetulla tavalla. Tällainen selkeä roolijako helpottaa myös virheiden jäljittämistä ja koodin ylläpitoa pitkällä aikavälillä.

6.3 Testien suoritusnopeuden ja tehokkuuden arviointi

Suoritusnopeus on yksi tehokkuuden tärkeimmistä mittareista, sillä nopeammat testit mahdollistavat tiheämmät testiajot ja nopeamman palautteen kehitystiimille. Tässä osiossa verrataan vanhan SeleniumLibraryn ja uuden Browser-kirjaston suoritusnopeutta samoilla testitapauksilla. Molemmilla kirjastoissa testit suoritettiin viisi kertaa, ja tuloksina käytettiin ajon keskiarvoja. Paikallisessa ympäristössä suoritusajoissa on hyvin vähän eroavaisuuksia, mutta verkkoympäristössä vaihtelua esiintyy enemmän.

Testi	**Selenium (s)**	**Browser (s)**	**Parannus (%)**
Laskutus	1:49	1:04	41
Muut toiminnot	0:43	0:32	25
Yhteensä	2:32	1:36	37

Taulukko 6.3. *Testien suoritusnopeuden vertailu SeleniumLibraryn ja Browser-kirjaston välillä paikallisessa ympäristössä.*

Kuten tuloksista käy ilmi, Browser-kirjaston avulla saavutettiin merkittäviä parannuksia suoritusajoissa. Laskutuksen testit nopeutuivat 41 % ja käyttäjänhallintaan liittyvät testit 25 %. Tämä johtuu pitkälti Browser-kirjaston kyvystä optimoida selainpohjaisten toimintojen ajoa sekä paremmasta asynkronisten toimintojen hallinnasta.

On kuitenkin huomioitava, että verkkoyhteyksiä käyttävissä ympäristöissä suoritusajat voivat pidentyä merkittävästi verrattuna lokaalisti ajettuihin testeihin. Tällaisia käytetään usein ketterän kehityksen jatkuvan integraation putkissa (eng. CI/CD-pipeline). Verkkotesteissä sivujen lataaminen ja sovelluksen prosessointi vievät enemmän aikaa, mikä kaventaa kirjastojen välistä prosentuaalista eroa. Tästä johtuen, kun testit ajetaan jatkuva integraatio -ympäristössä, molemmat kirjastot näyttävät suoriutuvan melko tasaisesti, vaikka Browser-kirjasto on edelleen hieman nopeampi.

Testi	**Browser keskiarvo (s)**
Laskutus	3:09
Muut toiminnot	0:33
Yhteensä	3:42

Taulukko 6.4. *Testien suoritusnopeuden keskiarvot Browser-kirjastolla jatkuva integraatio -ympäristössä.*

Vanhoissa testeissä havaittiin myös puutteita testien tarkistusten osalta. Tämä tarkoittaa, että testi saattoi ajoittain virheellisesti ilmoittaa läpäisyn. Se osaltaan vääristää SeleniumLibraryn tuloksia nopeampaan suuntaan, sillä se saattoi siirtyä liian aikaisin seuraavaan vaiheeseen. Tämän takia näitä tuloksia kannattaa pitää ainoastaan suuntaa antavina. Voidaan siis sanoa, että Browser-kirjasto on toki nopeampi, mutta ei voida sanoa kuinka

paljon nopeampi se tarkalleen on. Tarkkojen tuloksien ilmoittaminen vaatisi vanhojen testien päivittämistä identtisiksi, mikä ei ole työn tavoitteen kannalta olennaista. Tämän takia vanhojen testien suoritusnopeudesta ei ole tarkkaa dataa.

Kun vanhat testit oli korvattu, toteutettiin vielä uudet testit kattavuuden parantamiseksi. Suoritusaika luonnollisesti kasvoi, mutta lopullisessa optimoinnissa suoritusaikaa saatiin pudotettua vielä 40% siitä, mitä se oli ennen optimointia. Näin päästiin tavoitteeseen suoritusajan osalta, koska 10 minuutin rajaa ei haluttu ylittää. Tämän seurauksena kaikki testit voitiin ajaa koodimuutosten yhteydessä, eikä niitä tarvinnut eritellä kriittisten ja vähemmän kriittisten testien osalta.

6.4 Muiden näkökulmien pohdintaa

Testausratkaisun teknisen toimivuuden lisäksi on tärkeää pohtia sen vaikutuksia laajemmassa kontekstissa, erityisesti ohjelmiston laadun ja käyttäjäkokemuksen näkökulmasta. Tässä osiossa tarkastellaan, kuinka parannettu testauksen kattavuus ja luotettavuus ovat vaikuttaneet sovelluksen vakauteen ja käyttäjäkokemukseen.

Erityisesti testien kattavuuden ja luotettavuuden parantamisella on ollut keskeinen rooli sovelluksen vakauden ja käyttäjäkokemuksen varmistamisessa. Tämän seurauksena kriittisiin ominaisuuksiin on onnistuttu rakentamaan uusia testejä, jotka toimivat aiempaa tarkemmin ja vähentävät vääriä hälytyksiä.

Lisätty kattavuus tarkoittaa, että yhä useammat virheet havaitaan aikaisemmassa kehitysvaiheessa, mikä paitsi nopeuttaa vikojen korjausta, myös vähentää virheiden leviämistä järjestelmän myöhempiin vaiheisiin. Tämä parantaa merkittävästi ohjelmiston tuotantoon päätyvän version laatua. Kun viat havaitaan varhaisessa vaiheessa, niiden korjaaminen ei ainoastaan säästä aikaa ja rahaa, vaan estää myös tilanteita, joissa vakavia ongelmia päätyy loppukäyttäjien ulottuville.

Kun putken kautta havaitaan virhe, kehittäjät voivat puuttua siihen välittömästi. Tämä parantaa työn sujuvuutta ja estää ongelmien eskaloitumisen suuremmiksi. Vian juurisyyn korjaaminen heti on kriittistä, koska se vähentää tarvetta tehdä laajamittaisia analyysejä myöhemmin, mikä usein vie enemmän aikaa ja vaatii useamman tiimin osallistumista.

On kuitenkin tärkeää huomioida, että testausprosessi ei ole täydellinen. Automaatio ja manuaalinen testaus voivat epäonnistua, jolloin virheitä voi silti päätyä tuotantoon. Näissä tilanteissa vaikutukset voivat olla merkittäviä. Jos kriittinen virhe jää havaitsematta, se voi aiheuttaa suuria ongelmia, jotka johtavat taloudellisiin menetyksiin tai sovelluksen maineen heikkenemiseen. Huonosti toimiva sovellus voi johtaa asiakastyytyväisyyden heikkenemiseen, mikä puolestaan vaikuttaa negatiivisesti liiketoiminnan tulokseen.

Sovelluksen viallisen toiminnan vakavin seuraus on se, että se voi aiheuttaa taloudellisia

menetyksiä asiakkaille, jolloin yritys joutuu vastaamaan vahingoista. Tämä voi olla kriittistä erityisesti liiketoiminta-alustojen kohdalla, joissa sovellusten virheet voivat vaikuttaa esimerkiksi maksutapahtumiin tai tilausten käsittelyyn. Näiden riskien minimointi testauksen avulla on siksi keskeinen tavoite, ja yrityksen tulisi jatkuvasti kehittää testiautomaatiota mahdollisten ongelmien havaitsemiseksi mahdollisimman aikaisessa vaiheessa.

Näitä riskejä voidaan vähentää jatkuvalla testiprosessien optimoinnilla ja ottamalla käyttöön uusia automaatiotyökaluja, jotka lisäävät testauksen tehokkuutta ja kattavuutta. Esimerkiksi nykyisessä testausautomaation implementaatiossa on jo havaittu, että uudet testit ovat parantaneet virheiden havaitsemista ja vähentäneet vääriä hälytyksiä, mikä osoittaa suoraan investoinnin kannattavuuden.

7. JATKOKEHITYSMAHDOLLISUUDET

Testien ylläpito ja jatkokehitys on keskeinen osa testausprosessia, ja tulevaisuudessa vastuut voidaan jakaa tiimissä selkeästi. Kehittäjät voivat olla vastuussa omien muutostensa aiheuttamista rikkoutuneista testeistä ja niiden korjaamisesta, erityisesti yksikkö- ja integraatiotestien osalta, kuten ketterässä tiimissä on tapana. Lisäksi funktionaalisten testien ylläpito kuuluu kehittäjille niissä tapauksissa, joissa koodimuutos rikkoo aiemman testin. Tämä on perusteltua, koska Robot Frameworkin syntaksi on yksinkertainen ja avainsanapohjainen lähestymistapa tekee siitä helppolukuista. Tämä takaa sen, että testit pysyvät ajantasaisina koodimuutosten myötä. Uusien funktionaalisten testien laajentaminen voidaan kuitenkin antaa pääosin testitiimin vastuulle, koska heillä on paras käsitys testien kirjoittamisen käytänteistä ja kattavuuden varmistamisesta.

Testidatan osalta on tärkeää pohtia, miten sitä voidaan laajentaa kattamaan laajasti erilaisia käyttötilanteita. Yksi keskeinen parannus voisi olla osittain dynaamisen testidatan hyödyntäminen. Esimerkiksi päivämäärät ja muut muuttuvat tiedot voidaan generoida automaattisesti, kun taas keskeiset testattavat tiedot pysyvät vakaina. Tämä mahdollistaisi monipuolisemman testauksen ja ehkäisisi tilanteita, joissa dynaaminen data aiheuttaisi epäluotettavia tuloksia. Dynaamisuuden liiallinen käyttö voi johtaa vaikeasti toistettaviin virheisiin, joten tasapaino dynaamisuuden ja vakaan datan välillä on tarpeen. Lisäksi olisi hyödyllistä, jos aiempien kausien laskutustiedot voitaisiin kopioida testidataan. Näin koodimuutosten jälkeisellä laskutuksella voitaisiin vertailla vanhoja ja uusia tietoja, ja varmistaa, että jos laskutus on ollut aiemmin oikein, se pysyy oikeana myös uuden koodin kanssa. Tämä parannus toisi arvokasta tietoa sovelluksen laskutusprosessien eheydestä.

Testien laajentaminen eri laitteille ja selaimille on seuraava tärkeä askel. Sovelluksen tulee toimia moitteettomasti eri päätelaitteilla ja resoluutioilla, kuten puhelimilla, tableteilla ja eri näytön resoluutioilla (esim. 1080p ja 1440p). Browser-kirjaston moniselain- ja monilaitetuki, mukaan lukien Chrome, Firefox ja WebKit, mahdollistaa testien ajamisen useissa eri ympäristöissä. Tämä voidaan integroida suoraan CI/CD-putken osaksi, jolloin testit ajetaan automaattisesti eri laite- ja selainympäristöissä, mikä varmistaa sovelluksen laajan toimivuuden.

Testausta voidaan tehostaa myös lisäämällä Azure-agentteja, jolloin testejä voidaan ajaa rinnakkain. Tämä vähentää testien suorittamiseen kuluvaa kokonaisaikaa ja nopeuttaa

palautteen saamista tiimille. Rinnakkainen testien ajo vähentää myös ruuhkautumisen riskiä CI/CD-putkessa. Tämä on erityisen hyödyllistä laajoissa testikokonaisuuksissa, joissa yksittäiset testit voivat viedä paljon aikaa.

Jatkossa testisarjoja voidaan myös laajentaa. Aikaisempiin testisarjoihin voidaan kehittää lisää testejä kattamaan uusia skenaarioita ja varmistamaan entistä parempi kattavuus. Tämä lisää sovelluksen toimintavarmuutta ja vähentää riskiä, että jokin osa sovelluksesta jää testaamatta.

Tulevaisuuden teknologiat ja työkalut voivat parantaa testiautomaatiota entisestään. Esimerkiksi Playwrightin kehitys ja sen mahdollisuudet Browser-kirjastossa tarjoavat uusia tapoja nopeuttaa testejä ja käsitellä moderneja web-teknologioita tehokkaammin. Myös muiden testauskirjastojen, kuten Cypressin ja Selenium 4:n, kehityksen seuraaminen voi tuoda uusia ideoita ja työkaluja testausprosessin optimointiin. Koneoppimisen ja tekoälyn käyttö testidatan luomisessa ja testien optimoinnissa voi tarjota lisämahdollisuuksia, mikä tekee testauksesta entistä älykkäämpää ja tehokkaampaa.

Näiden työkalujen integrointi osaksi olemassa olevia testausprosesseja on tärkeää, jotta testauksen kehitys voi jatkua saumattomasti ja vastata sovelluksen tuleviin tarpeisiin.

8. YHTEENVETO

Työn tavoitteena oli kehittää aikaisempaa, puutteellista käyttöliittymään kohdistuvaa funktionaalista testausta. Parannuksia haettiin yksinkertaisuuden, luotettavuuden, suoritusnopeuden, ylläpidettävyyden ja kattavuuden osalta. Työ toteutettiin käyttämällä vuonna 2020 julkaistua Robot Framework Browser -kirjastoa selainpohjaisen testiautomaation toteuttamiseen. Koodi- ja tiedostotasolla käytettiin Page Object Model -suunnittelumallia, jolla koodimassa olisi helpommin hallittavaa.

Tässä diplomityössä päästiin johdannossa esitettyihin tavoitteisiin eli, luotettavuuden, yksinkertaisuuden ja nopeuden parantamiseen. Luotettavuutta parannettiin lisäämällä kattavuutta. Kattavuutta saatiin lisättyä tekemällä lisää testisarjoja sovelluksen eri toiminnallisuuksille. Kaikki kriittiset sovelluksen osa-alueet, joista osasta ennen puuttui testisarja, saatiin katettua. Muihin parannuksiin päästiin tekemällä tehokkaita, uudelleenkäytettäviä ja yksinkertaisempia avainsanoja. Vanhat testit korvaamalla havaittiin koodimäärän pudonneen jopa puoleen alkuperäisestä. Page Object Model -tyyli tarjosi myös yksinkertaisuutta ja ylläpidettävyyttä keskisuurelle ohjelmistoprojektille. Robot Framework Browser tarjosi myös nopeampia testiaikoja verrattuna vanhempaan SeleniumLibraryyn, vaikkakin verkkoympäristöissä ero jää prosentuaalisesti pienemmäksi verkkoyhteyksistä johtuen.

Kattavuudessa parannettavaa jäi vielä vähemmän kriittisten osa-alueiden osalta. Myös toteutettuihin testisarjoihin jäi laajentamiselle varaa. Avainsanat ovat kuitenkin suunniteltu niin, että uusia testejä on helppo lisätä jälkikäteen. Uusia testejä voidaan lisätä esimerkiksi uusien testitapausten kannalta tai eri muuttujan arvoilla tekemällä data-vetoisia testejä. Testejä lisättiin tässä vaiheessa maltillisesti, sillä yrityksen resurssit tulee huomioida testauksen ylläpidettävyyttä pohtiessa. Nopeuden kannalta Robot Framework Browser oli selvästi SeleniumLibraryä parempi, mutta toisaalta se tuotti myös ongelmia joissakin testitapauksissa, koska sivustoja ei oltu optimoitu käyttöliittymätestaukselle. Joskus toiminnot olivat liian nopeita, eikä sivusto ollut vielä ladannut kaikkia tarvittavia komponentteja ajoissa.

Yleisesti voidaan sanoa, että Robot Framework ja sen Browser-kirjasto sekä Page Object Model -suunnittelumalli sopivat hyvin keskikokoisiin ja suuriin selainpohjaisiin ohjelmistoprojekteihin. Pienemmissä projekteissa Robot Framework on toki myös käypä, mutta Page Object Model voi olla enemmän haitaksi kuin hyödyksi.

LÄHTEET

[1] Anil Agarwal, N. K. Garg ja Avirag Jain. ”Quality Assurance for Product Development Using Agile”. *ICROIT 2014 - Proceedings of the 2014 International Conference on Reliability, Optimization and Information Technology* (2014). Web, s. 44–47. (Viitattu 28. 03. 2024).

[2] *Browser Library*. 2024. URL: https://robotframework-browser.org/ (viitattu 28. 06. 2024).

[3] BrowserStack. *Differences Between Functional and Non-functional Testing*. 2024. URL: https://www.browserstack.com/guide/functional-vs-non-functional-testing (viitattu 25. 03. 2024).

[4] Paul M. Duvall, Steve Matyas ja Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, 2007. (Viitattu 03. 06. 2024).

[5] Dale H. Emery. *Writing Maintainable Automated Acceptance Tests*. 2009. URL: http://www.cs.uni.edu/~wallingf/teaching/172/resources/emery-on-acceptance-tests.pdf (viitattu 12. 10. 2024).

[6] Mark Fewster ja Dorothy Graham. *Software Test Automation*. Addison-Wesley Longman Publishing Co., Inc., 1999. (Viitattu 13. 06. 2024).

[7] Martin Fowler. *Continuous Integration*. 2018. URL: https://martinfowler.com/articles/continuousIntegration.html (viitattu 31. 05. 2024).

[8] Sergio Freire. *Browser Library vs Selenium Library*. 2024. URL: https://www.sergiofreire.com/post/robotframework-browser-architecture/ (viitattu 15. 09. 2024).

[9] Matthew Heusser ja Michael Larsen. *Software Testing Strategies: A Testing Guide for The 2020s*. Packt Publishing, 2023. ISBN: 9781837638024. (Viitattu 22. 03. 2024).

[10] Jez Humble ja David Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 2010. (Viitattu 31. 05. 2024).

[11] JavaTpoint. *Functional Testing*. 2024. URL: https://www.javatpoint.com/functional-testing (viitattu 02. 08. 2024).

[12] Ilya Kantor. *Page: DOMContentLoaded, load, beforeunload, unload*. 2024. URL: https://javascript.info/onload-ondomcontentloaded (viitattu 21. 09. 2024).

[13] Gene Kim ym. *The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations*. IT Revolution Press, 2016. (Viitattu 10. 06. 2024).

[14] I. Kyryk. ”QATestLab: What projects need test automation”. *QATestLab Blog* (2018). URL: https://blog.qatestlab.com/2023/04/12/when-automate-testing/ (viitattu 11. 04. 2024).

[15] Dudekula Mohammad Rafi ym. ”Benefits and Limitations of Automated Software Testing: Systematic Literature Review and Practitioner Survey”. Teoksessa: *2012 7th International Workshop on Automation of Software Test (AST)*. IEEE, 2012, s. 36–42. DOI: 10.1109/IWAST.2012.6228988. URL: https://doi.org/10.1109/IWAST.2012.6228988 (viitattu 15. 09. 2024).

[16] *Robot Framework - SeleniumLibrary*. 2024. URL: https://robotframework.org/SeleniumLibrary/ (viitattu 05. 07. 2024).

[17] *Robot Framework Browser - Click*. 2024. URL: https://marketsquare.github.io/robotframework-browser/Browser.html#Click (viitattu 28. 09. 2024).

[18] *Robot Framework User Guide*. 2024. URL: https://robotframework.org/robotframework/latest/robotFrameworkUserGuide.html (viitattu 24. 06. 2024).

[19] *robotframework-browser · PyPI*. 2024. URL: https://pypi.org/project/robotframework-browser/ (viitattu 17. 06. 2024).

[20] Scaler. *Which Browser Supports Selenium?* 2024. URL: https://www.scaler.com/topics/selenium-tutorial/which-browser-supports-selenium/ (viitattu 15. 09. 2024).

[21] Slideshare. *Robot Framework Dos and Don’ts*. 2024. URL: https://www.slideshare.net/slideshow/robot-framework-dos-and-donts/38564910 (viitattu 20. 06. 2024).

[22] Jonathan Sychan. *Some Code Testing Concepts*. 2024. URL: https://medium.com/@jonathansychan/some-code-testing-concepts-85fbc8fc27e2 (viitattu 02. 08. 2024).

[23] JanBask Training. *Page Object Model with Page Factory in Selenium WebDriver*. 2024. URL: https://www.janbasktraining.com/blog/pom-with-page-factory-in-selenium-webdriver/ (viitattu 21. 06. 2024).

[24] *Turning the Page on Front-end Automation - Robot Framework Browser Library*. 2024. URL: https://www.valagroup.com/blog/turning-the-page-on-front-end-automation-robot-framework-browser-library/ (viitattu 05. 07. 2024).

[25] UTOR. *Functional vs Non-Functional Testing: Expert Guide*. 2024. URL: https://u-tor.com/topic/functional-vs-non-functional (viitattu 02. 08. 2024).

[26] Timothy D. West ja Mark Blackburn. ”Is Digital Thread/Digital Twin Affordable? A Systemic Assessment of the Cost of DoD’s Latest Manhattan Project”. *Procedia Computer Science* 114 (2017), s. 47–56. URL: https://doi.org/10.1016/j.procs.2017.09.003 (viitattu 08. 08. 2024).