

Jukka Pajarinen

WEB-KÄYTTÖLIITTYMÄN HYVÄKSYMISTESTAUKSEN PRIORISOINTI PAINOTETUN VERKON AVULLA

Informaatioteknologian ja viestinnän tiedekunta

Diplomityö

Joulukuu 2019

TIIVISTELMÄ

Jukka Pajarinen: Web-käyttöliittymän hyväksymistestauksen priorisointi painotetun verkon avulla
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Joulukuu 2019

<TODO: miksi tärkeää, testiautomaatio, web-sovellukset, monta testauksen tasoa, hyväksymistestaus, priorisointiongelma, priorisoinnin tärkeys>
<TODO: mitä tässä työssä tutkittiin>
<TODO: miten työ tehtiin / miten työn tutkimusongelmaan löydettiin vastaus>
<TODO: työn tulokset lyhyesti / mitä tutkimusongelmaan vastaaminen tuotti ...>
<TODO: työn tulosten konkreettiset hyödyt lyhyesti>
<TODO: työn tuloksena syntynyt priorisointimenetelmä on työn tärkeintä antia, josta diplomitöön tekijä on iloinen jos siitä on hyötyä lukijalle>

Avainsanat: hyväksymistestaus, painotettu verkko, priorisointi, jatkuva integraatio, web-sovellukset, testiautomaatio

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

ABSTRACT

Jukka Pajarinen: Web User Interface Acceptance Testing Prioritization with a Weighted Graph
Master's Thesis
Tampere University
Degree Programme in Information Technology
December 2019

<TODO: why important, test automation, web applications, many testing levels, acceptance testing, prioritization problem, prioritization importance>
<TODO: what is researched in this research>
<TODO: how was the research done / how was the answer for research question found?>
<TODO: research results shortly / what answering to research question produced? ...>
<TODO: concrete benefits of the research results shortly>
<TODO: prioritization method is the most important research results, which researcher is very happy for bringing any value for the reader... >

Keywords: acceptance testing, weighted graph, prioritization, continuous integration, web applications, test automation

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

ALKUSANAT

Tämä diplomityö päättää korkeakouluopintoni Tampereen yliopistossa. Diplomi-insinöörin tutkinto ja sen kautta saavutettava asiantuntijuus tietotekniikan alalla on ollut pitkään yksi tavoitteistani elämässä. Opintoihin kului aikaa hieman tavoiteaikaa pidempään ja opintopisteitä kertyi rutkasti yli vähimmäisvaatimusten. Korkeakouluopintojeni aikana Tampereen teknillinen yliopisto ehti myös vaihtamaan nimensä Tampereen yliopistoksi. Nyt tämä pitkäaikainen elämäntavoite on saavutettu ja voin keskittyä kehittämään asiantuntijuuttani opintojen ulkopuolella. Diplomityöprosessin aloittamisen myötä päädyin myös sovelluskehittäjäksi tamperelaiseen yritykseen WordDiveen, johon tämä diplomityö on tehty.

Haluan kiittää opiskelukavereitani mahtavista hetkistä ja yhdessä viettämästämme ajasta Tampereen yliopistossa. Opiskelukavereistani kehittyi minulle erittäin hyviä ystäviä myös opintojen ulkopuolella. Kiitän myös vanhempiani Tuulikkia ja Olavia sekä veljeäni Mikkoa kannustamisesta ja siitä tuesta, minkä he ovat minulle kautta elämäni osoittaneet. Kiitän myös WordDiven toimitusjohtajaa Timo-Pekka Leinosta ja esimiestäni Juha Rintaa diplomityön tekemisen mahdollistamisesta ja joustavuudesta kirjoitusosuuden tekemiseen. Yliopiston puolelta haluan kiittää Kari Systää ja Hannu-Matti Järvistä työn ohjaamiseen ja tarkastamiseen liittyvissä asioissa. Lopuksi haluan osoittaa suurimmat kiitokset puolisololleni Katille, joka on tukenut ja kuunnellut aina vaikeinakin hetkinä diplomityöprosessin aikana ja siitäkin huolimatta kulkenut aina vierelläni.

Tampereella, 31. joulukuuta 2019

Jukka Pajarinen

SISÄLLYSLUETTELO

1	Johdanto	1
2	Tutkimusasetelma	2
2.1	Tausta	2
2.2	Tutkimuskysymykset	3
2.3	Tutkimusmenetelmä	4
2.4	Tutkimuksen rajaus	4
2.5	Tavoitteet	5
3	Testiautomaatio	7
3.1	Testiautomaation tarkoitus	7
3.2	Testauksen tasot	8
3.2.1	Yksikkötestaus	9
3.2.2	Integraatiotestaus	10
3.2.3	Järjestelmätestaus	10
3.2.4	Hyväksymistestaus	11
3.3	Testitapaus ja testikokoelmat	12
3.4	Jatkuva integrointi	13
3.5	Testausvetoinen kehitys	14
4	Hyväksymistestaus	16
4.1	Hyväksymistestauksen tarkoitus	16
4.2	Hyväksymistestausvetoinen kehitys	17
4.3	Web-sovelluksien erityispiirteet	19
4.4	Hyväksymistestauksen työkaluja	20
4.4.1	Robot Framework	20
4.4.2	Selenium	21
4.4.3	Xvfb	22
4.4.4	Docker	23
4.4.5	GoCD	24
4.5	Testitapauksien rakentaminen	24
4.6	Priorisointiongelma	25
5	Priorisointi painotetun verkon avulla	27
5.1	Matemaattisten verkkojen tarkoitus	27
5.2	Perusmerkinnät ja käsitteet	27
5.3	Priorisointiin vaikuttavat muuttujat	28
5.4	Painofunktiot priorisointiin	29
5.5	Verkon rakentaminen	30
5.6	Verkon karsiminen	33

5.7	Dijkstran algoritmin hyödyntäminen	34
5.8	Verkon ja testitapauksien yhteys	34
6	Tulosten tarkastelu ja arviointi	36
6.1	Tutkimuksen konkreettiset tulokset	36
6.2	Toteutuksen evaluointi	36
6.3	Menetelmän evaluointi	37
6.4	Jatkokehitysehdotukset	38
7	Yhteenveto	39
	Lähteet	40
	Liite A Esimerkki testitapauksesta Robot Framework:illä	41
	Liite B Dijkstran algoritmi pseudokoodina	42

KUVALUETTELO

3.1	Testauksen tasot pyramidin muodossa	8
3.2	Jatkuvan integroinnin perusperiaate on iteratiivinen	13
3.3	Testausvetoisen kehityksen vaiheet	15
4.1	Hyväksymistestausvetoisen kehityksen vaiheet	18
4.2	Robot Framework alustan arkkitehtuuri	21
4.3	Dockerin ja virtuaalikoneen eroavaisuus	23
5.1	Esimerkki painotetusta verkosta ennen leikkauksia	32
5.2	Esimerkki painotetusta verkosta leikkauksien jälkeen	34

TAULUKKOLUETTELO

5.1	Priorisointiin vaikuttavat muuttujat	29
5.2	Esimerkkiverkon näkymät, siirtymät ja priorisointimuuttujat	31

LYHENTEET JA MERKINNÄT

N	Luonnolisten lukujen joukko
A/B	A/B-testaus, kahta eri ohjelmarevisiota vertaileva testaus
API	Application Programming Interface, ohjelmointirajapinta
ATDD	Hyväksymistestausvetoinen kehitys, englanniksi: acceptance test driven development
CI	Continuous Integration, eli jatkuva integrointi
DOM	Document object model, eli html-sivujen dokumenttiobjektimalli
e2e	End-to-end, eli päästä päähän testaus
HTML	Hypertext markup language, eli internetin verkkosivuihin käytetty hypertekstin merkintäkieli
IDE	Integrated Development Environment, ohjelmointiympäristö
MoSCoW	Must, Should, Could, Would; priorisointimenetelmä
SQL	Structured query language, eli tietokantoihin kohdistettava haavoittuvuus
UAT	User Acceptance Testing, eli käyttäjän hyväksyttämistestaus
UNIX	Uniplexed Information and Computing Service, yksi suosittu ja vapaa tietokoneen käyttöjärjestelmäperhe
XSS	Cross site scripting, eli verkkosivuihin kohdistettava haavoittuvuus
Xvfb	X virtual framebuffer, X-ikkunointijärjestelmän protokollan toteuttava virtualisointipalvelin

1 JOHDANTO

<TODO: miksi tärkeää, testiautomaatio, web-sovellukset, monta testauksen tasoa, hyväksymistestaus, priorisointiongelma, priorisoinnin tärkeys>

<TODO: tässä työssä suunnitellaan ja rakennetaan testausjärjestelmä>

<TODO: tässä työssä suunnitellaan priorisointimenetelmä>

<TODO: luvussa X esitetään>

<TODO: luvussa X esitetään>

<TODO: luvussa X esitetään>

<TODO: lopuksi luvussa X esitetään>

2 TUTKIMUSASETELMA

Tässä luvussa esitetään diplomityön tausta, tutkimuskysymykset, käytetty tutkimusmenetelmä, tutkimuksen rajausta sekä tavoitteet. Tutkimuskysymykset liittyvät vahvasti yhteiseen hyväksymistestauksen testitapauksien priorisoinnin teemaan, johon tässä työssä erityisesti keskitytään. Tutkimus on soveltavaa ja sen tarkoituksena on muodostaa selvitys tutkimusongelman ratkaisemiseksi. Tässä työssä se tarkoittaa erityisesti matemaattisen, toistettavissa olevan menetelmän kehittämistä hyväksymistestauksen testitapauksien priorisointiongelman ratkaisemiseksi. Tutkimuskysymyksistä 2.2 itsessään voi päättellä tutkimuksen tarkoitusta ja tavoitteita, mutta tämä esitetään myös yksityiskohtaisemmin tavoitteet luvussa 2.5. Yhtenä diplomityön osana on myös toteutuksellinen osuus ??, joka on tehty diplomityön asiakasyrityksen tarpeita varten. Toteutuksellisessa osuudessa esitetään hyväksymistestausjärjestelmä, joka mahdollistaa tutkimusongelmaan liittyvien priorisoitavien testitapauksien toteuttamisen.

2.1 Tausta

Diplomityö tehtiin WordDive nimiselle yritykselle. WordDive on vuonna 2009 perustettu, Tampereella toimiva, suomalainen kieltenoppimiseen keskittyvä yritys. WordDivellä oli kirjoitushetkellä kieltenoppimissovellus mobiilialustalle sekä web-alustalle. Tämän diplomityön sisältö koskettaa vain web-alustalla toimivaa sovellusta. Hyväksymistestauksen osalta mobiilisovellukselle oli yrityksessä jo toteutettu testiautomaatio, mutta web-alustalle sitä ei vielä oltu tehty.

Allekirjoittanut aloitti työt kyseisessä yrityksessä 2018 vuoden loppupuolella, jolloin diplomityön aihetta ei vielä ollut. Tarkoituksena oli tuolloin ensin töitä tekemällä tutustua yrityksen web-alustalla toimivaan sovellukseen ja yrityksen ohjelmistotuotantoprosessiin. Diplomityön aihe alkoi muotoutua vasta vuoden 2019 alkupuolella, kun tarvittava tietämys ohjelmistotuotteesta ja prosessista oli saavutettu. Asiakasyrityksessä sai hyvinkin vapaasti löytää itseään kiinnostavan, varsinaisten töiden ohella tehtävän, mutta kaikille osapuolille hyödyllisen aiheen. Aiheen löytämisen taustalla olivat hyvinkin konkreettiset tarpeet, jotka ohjelmistotuotannon työssä tulivat esille.

Uusien ominaisuuksien ja koodimuutoksien tekemisen yhteydessä oli jatkuvasti tarve huolelliselle testaamiselle ja erityisesti asiakkaan näkökulmasta tärkeimpien sovelluksen ominaisuuksien toiminnan varmistamiselle. Tämä sai diplomityön aiheen suuntautumaan testiautomaatioon ja erityisesti hyväksymistestaukseen. Lisäksi yrityksessä oli jo toteu-

tettuna päivittäisessä käytössä oleva hyväksymistestaus mobiilialustalle, joka auttoi hahmottamaan web-sovelluksen testiautomaation integroimista osaksi yrityksen ohjelmistotuotantoprosessia. Mobiilialustalle tehtyä hyväksymistestausta varten oli yrityksessä jo valittu tietyt hyväksi todetut sovelluskehikset ja työkalut testiautomaatiota varten, joten tässä työssä ei enää ollut tarvetta evaluoida eri työkaluja tarvittavan testiautomaation toteuttamiseksi. Web-sovelluksen testiautomaatio toteutettiin käyttäen pääpiirteittäin samoja sovelluskehiksiä ja työkaluja kuin mobiilisovellukselle 4.4. Tästä syystä diplomityön aihetta ja tutkimusongelmaa lähdettiin etsimään muualta.

Tutkimusongelmaan mietittiin aihioita etenkin alkuvaiheessa polkutestauksen ja ohjelmistotestaukseen liittyvien heuristiikkojen osalta. Polkutestaus oli yhtenä vaihtoehtona, mutta siitä luovuttiin, koska sen todettiin olevan paremmin soveltuvampi hyväksymistestausta alemmille testauksen tasoille. Heuristiikkojen hyödyntämistä mietittiin kahteen eri ongelmaan; testitapauksien muodostamiseen sekä kriittisten testitapauksien määrittämiseen.

Lopullisen tutkimusongelman löytämiseen vaikuttivat erityisesti konkreettiset tarpeet, jotka tulivat esiin vasta web-sovelluksen testiautomaation suunnitteluvaiheessa. Hyväksymistestauksen testiautomaatiota varten oli ensin määritettävä mitä testauksen kohteena olevasta sovelluksesta tulisi testata. Testiautomaation rakentamiseen allokoitavia resursseja oli rajallinen määrä, jonka lisäksi testikattavuuden suppeus sekä ylikattavuus nähtiin selkeänä ongelmana. Tämä ongelma voidaan esittää yksinkertaisemmin testitapauksien priorisointiongelmana, joka myös lopulta muotoutui työn tutkimusongelmaksi. Priorisointiongelman valitsemiseen ratkaisevasti johtavia asioita olivat kaksi allekirjoittaneen oivallusta aiheesta. Ensimmäiseksi hyväksymistestattavaa web-sovellusta keksittiin ajatella käyttöliittymän näkymä- ja siirtymätasolla matemaattisena prioriteetein painotettu verkkona. Toiseksi oivallukseksi keksittiin käyttää lyhimmän polun ongelmaan kehitettyjä algoritmeja prioriteetein painotettuun verkkoon, jolloin kahden solmun välille voitiin löytää prioriteeteiltaan korkein polku. Nämä oivallukset vaikuttivat lopulta varsinaisen tutkimusongelman eli testitapauksien priorisointiongelman valitsemiseen, koska ne loivat järkevän ja mielenkiintoisen pohjan tutkimusongelmaan vastaamiselle. Kokonaisuutena diplomityön aihe saatiin muodostettua sellaiseksi, että se esittää yleishyödyllisen menetelmän tutkimusongelman ratkaisemiseen sekä siihen liittyvän toteutuksen suunnittelemisen ja rakentamisen asiakasyritykselle.

2.2 Tutkimuskysymykset

Tutkimuksen tarkoituksena on pohjimmiltaan tarkoitus löytää ja kehittää toistettavissa oleva menetelmä hyväksymistestauksen testitapauksien priorisoimiseen. Testitapauksien laatimisen yleisenä ongelmakohtana on erityisesti niiden priorisointi, joka usein johtaa liian suppean tai ylikattavan testiautomaation rakentamiseen. Tutkimuskysymykset on laadittu siten, että niihin vastaaminen antaa ratkaisun tähän edellä mainittuun testiautomaation ongelmaan.

Työlle asetettiin seuraavat tutkimuskysymykset:

- **T1:** *Miten painotettua verkkoa voidaan käyttää testitapauksien priorisointiin?*
- **T2:** *Mitkä muuttujat vaikuttavat web-käyttöliittymän hyväksymistestauksen testitapauksien priorisointiin?*
- **T3:** *Kuinka prioriteetein painotetusta verkosta valitaan toteutettavat testitapaukset?*
- **T4:** *Miten painotetun verkon avulla tehty priorisointi liitetään yhteen jatkuvan integraation ja testiautomaation kanssa?*

2.3 Tutkimusmenetelmä

Tutkimuskysymyksiin vastaamiseksi työn tutkimusmenetelmäksi valittiin tietotekniikan diplomaatissa yleisesti käytetty Design Science -menetelmä. Menetelmän tarkoituksena on tuottaa teknologiaa hyödyntävä ratkaisu tutkimusongelmaan vastaamiseen. Valittua menetelmää käytettäessä pyritään tutkimaan uusia ratkaisumalleja ratkaisemattomiin ongelmiin tai kehittämään parempia ratkaisumalleja jo aiemmin ratkaistujen ongelmien tilalle. Tietotekniikan tutkimuksessa on aikojen saatossa kehitetty uusia tai parempia tietokonearkkitehtuuria, ohjelmointikieliä, algoritmeja, tietorakenteita ja tiedonhallintajärjestelmiä. Näiden osalta yhteistä on, että niissä on monesti käytetty usein jopa tiedostamatta Design Science -menetelmää.

Tutkimuksen tarkoituksena oli muodostaa uudenlainen ja toistettavissa oleva menetelmä tutkimuksen kohteena olevan ongelman ratkaisemiseksi. Tutkimusidean hahmottelemisen ja ratkaisua kaipaavan ongelman identifioinnin jälkeen, valittua tutkimusmenetelmää käyttäen ensin määriteltiin tutkimuskysymykset 2.2.

Seuraavaksi kartoitettiin ratkaisuvaihtoehto tutkimuskysymyksiin ja työn yleiseen priorisoinnin teemaan vastaamiseksi ja esitetään perustelut painotettua verkkoa hyödyntävään ratkaisumenetelmään päätymiseen 5.1. Tutkimusta varten kerättiin teoreettista aineistoa, jonka tarkoituksena oli tukea menetelmän kehittämistä ja jonka avulla pyrittiin luomaan lukijalle mahdollisimman vahva teoreettinen pohja tutkimuskysymyksiin vastaavan ratkaisumenetelmän ymmärtämiseksi. Asiakasyrityksen ohjelmistotuote ja ohjelmistotuotantoprosessi huomioiden toteutettiin myös testausjärjestelmä, joka mahdollistaa testiautomaation toteuttamisen ja työssä kehitetyn priorisoinnin ratkaisumenetelmän hyödyntämisen.

Lopuksi vielä evaluoitiin menetelmän eli ratkaisun ja sitä hyödyntävän toteutuksen toimivuus ja esitetään yhteenveto tutkimuksesta.

2.4 Tutkimuksen rajaus

Ohjelmistotestauksen tasojen osalta tutkimus rajoittuu hyväksymistestaukseen. Tämä rajaus pohjautuu ohjelmistotuotannon työssä konkreettisesti havaittuun tarpeeseen sekä yhdenmukaisen testiautomaation toteuttamiseen mobiili- ja web-sovelluksille asiakasyrityksessä.

Testitapauksien osalta on olemassa lukuisia eri testausalustoja, joita hyödyntäen testi-

tapauksia voidaan toteuttaa. Tässä työssä testitapauksien toteuttaminen rajataan tietylle ennalta määräytyneelle hyväksymistestaukseen tarkoitettulle Robot Framework -alustalle. Tämä rajaus pohjautuu asiakasyrityksessä jo aiemmin valittuihin testauksen sovelluskehysiin ja työkaluihin. Lisäksi Robot Framework on alustana yleisesti käytetty ja erityisen hyvin soveltuva etenkin hyväksymistestauksen toteuttamiseen. Robot Framework on soveltuva myös hyvin sellaisiin tilanteisiin, joissa halutaan ohjelmointikielillä määritettyjä testitapauksia korkeampaa abstraktiotasoa.

Jatkuvan integroinnin osalta tutkimus rajoittuu perusteisiin ja tutkimuksen painoarvo pidetään testitapauksien toteuttamisessa ja niiden priorisoinnissa. Jatkuva integrointi on kuitenkin asiakasyrityksessä tärkeä osa testiautomaation ja jatkuvan käyttöönoton toteutuksessa. Jatkuvan integroinnin osalta ei tässä työssä esitetä muuta kuin testiautomaation toteutusosaan kokonaisuutena erityisesti liittyvät käsitteet ja ratkaisu. Tämä rajaus pohjautuu tutkimusongelman tarkempaan spesifioimiseen ja tutkimuksen kokonaislaajuuden hallitsemiseen.

Verkkoteorian osalta tutkimus rajoittuu perusteisiin ja painotettua verkkoa sekä kehitettyä menetelmää tukeviin käsitteisiin. Tämä rajaus pohjautuu työn kohdentamiseen ohjelmistotuotantoon ja diplomityön kirjoitusvaiheessa saatuun ohjauspalautteeseen, jossa matematiikan osuus oli kasvanut liiallisen suureksi.

Priorisointiin vaikuttavien muuttujien osalta tutkimus rajoittuu muuttujien kartoittamiseen, mutta niiden määrittäminen jätetään työn ulkopuolelle. Tämä tarkoittaa käytännössä sitä, että jokainen menetelmää hyödyntävä taho hankkii itse varsinaiset numeeriset arvot muuttujille. Esimerkiksi liiketoiminnallisen vision numeerinen arvo on yksinomaan menetelmää käyttävän tahon harkittavissa.

Lyhimmän polun etsimiseen painotetusta verkosta on olemassa lukuisa määrä erilaisia algoritmeja, mutta tässä työssä hyödynnetään vain perinteistä Dijkstran algoritmia. Tämä rajaus pohjautuu työssä kehitetyn priorisointimenetelmän käyttämisen perimmäiseen tarkoitukseen, jossa ei algoritmin tehokkuudella tai lisäominaisuuksilla ole suurta merkitystä. Lisäksi Dijkstran algoritmi on selkeä, paljon tutkittu ja käytetty ratkaisu lyhimmän polun etsimiseen sekä sitä käytetään tässä työssä vain jo priorisoidussa verkossa tapahtuvaan analysointiin.

2.5 Tavoitteet

Tutkimuksen primäärisenä tavoitteena oli kehittää toistettavissa oleva matemaattinen menetelmä web-käyttöliittymien hyväksymistestauksessa tarvittavien testitapauksien priorisointiongelman ratkaisemiseksi. Kehitetyn menetelmän tavoitteena on tarjota ratkaisua, joka helpottaa ja tehostaa kyseisen hyväksymistestaukseen liittyvän testiautomaation sekä siihen liittyvien testitapauksien suunnittelua ja rakentamista.

Primääritavoitteen lisäksi valmiin diplomityön tavoitteena on myös tarjota selkeä, eheä ja helposti ymmärrettävä kokonaisuus hyväksymistestauksen testitapauksien priorisoi-

seen, työssä kehitettyä menetelmää käyttäen. Kehitetty priorisointimenetelmä pyritään esittämään siten, että valmiista diplomityöstä olisi mahdollisimman paljon hyötyä sen käyttämistä harkitseville tai käyttäville tahoille.

Tutkimusta ja tutkimusmenetelmää itsessään ajatellen tavoitteena oli tarjota ratkaisumalli ja ratkaisut aiemmin esitettyihin tutkimuskysymyksiin 2.2. Lisäksi tutkimusmielessä tavoitteena oli pystyä todentamaan kehitetyn menetelmän toimivuus käytännössä menetelmää itsessään sekä asiakasyritykselle sen lisäksi tehtyä testausjärjestelmän toteutusta evaluamalla. Evaluointi esitetään diplomityön lopussa ja se esitetään erikseen menetelmälle sekä toteutukselle.

3 TESTIAUTOMAATIO

Tässä luvussa esitetään perusteet ja tarvittavat tiedot ohjelmistojen testauksesta ja testiautomaatiosta, jotka liittyvät työn laajempaan teoreettiseen kehykseen. Ensin esitetään testiautomaation tarkoitus, jonka jälkeen käydään yksityiskohtaisesti läpi ohjelmistotestauksen tasot ja niiden merkitystä testiautomaatiossa. Ohjelmistojen testaukseen ja erityisesti testiautomaatioon sekä tämän diplomityön aiheeseen liittyvät käsitteet testitapaus ja testikokoelma esitetään omassa kappaleessaan. Lopuksi vielä esitetään tarvittavia jatkuvan integroinnin ja testausvetoisen kehityksen perusteita sekä pyritään luomaan lukijalle ymmärrystä siitä, miten ne liittyvät niitä laajempaan testiautomaation käsitteeseen ja diplomityön tuloksien käyttöönottoon.

Testiautomaation perusteiden ymmärtämistä tarvitaan varsinkin työn myöhemmissä vaiheissa, joissa esitetään testiautomaatioon liittyvien hyväksymistestauksen testitapauksien testausjärjestelmä ja tutkimusongelmaan vastaava varsinainen priorisointi painotetun verkon avulla.

3.1 Testiautomaation tarkoitus

Testiautomaation tarkoitus on pohjimmiltaan mahdollistaa ohjelmistotuotteen jatkuva, tehokas ja vaivaton laadunvarmistus nyt ja tulevaisuudessa. Testiautomaation vastakohtana voidaan ajatella manuaalista testausta, joka vaatii täydellistä ihmisen vuorovaikutusta testauksen suorittamiseen. Testiautomaatiossa käytetään erityisiä ohjelmistotyökaluja ennalta määritettyjen testitapauksien suorittamiseen, ihmisen tekemän manuaalisen testauksen sijaan. Ohjelmistojen testaamisella itsessään pyritään löytämään ohjelmistotuotteesta virheitä ja anomalioita sekä varmistamaan, että se toimii asetettujen vaatimusten sekä suunnitelmien mukaisesti. Testauksen automatisoiminen vapauttaa aikaa, kustannuksia ja henkilöresursseja manuaalisesta testaamisesta muihin tuotantotehtäviin sekä parantaa toistettavien testien luotettavuutta poistamalla manuaalisessa testauksessa tapahtuvat inhimillisen virheet. Testiautomaatiolla, joka kytketään osaksi ohjelmistotuotantoprosessia, voidaan myös löytää ohjelmistokehityksen aikana ohjelmistokoodiin lipsuvia virheitä ja näin ollen saavuttaa mahdollisuus korjata niitä ennen kuin ohjelmisto julkaistaan loppukäyttäjille.

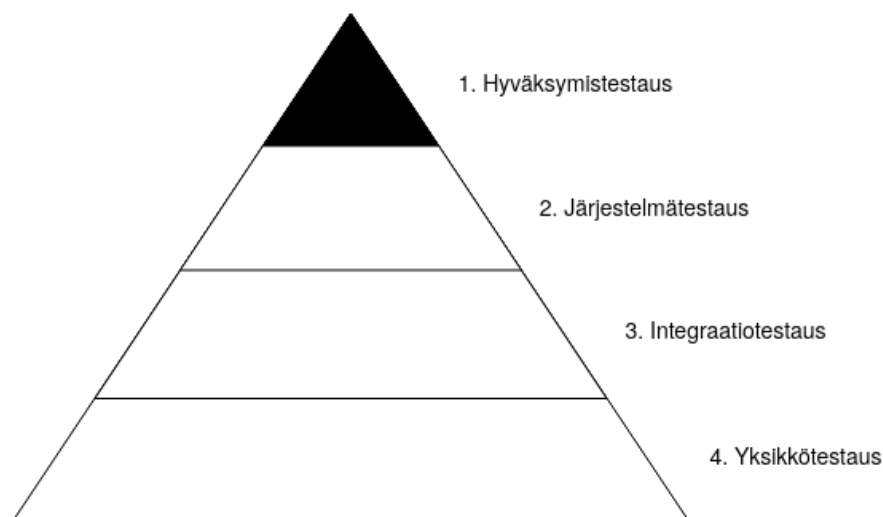
Laadunvarmistuksen osalta ohjelmistokehityksessä on usein käytetty niin sanottuja laadullisia ominaisuuksia, joiden kattamisella voidaan validoida laatua. Laadullisia ominaisuuksia ovat ISO 9126-standardin mukaan [1]:

- toiminnallisuus
- luotettavuus
- käytettävyys
- tehokkuus
- ylläpidettävyys
- siirrettävyys

Näistä laadullisista ominaisuuksista testiautomaatiolla pystytään kattamaan erityisesti toiminnallisia, luotettavuudellisia ja tehokkuudellisia ominaisuuksia. Käytettävyyden, ylläpidettävyyden ja siirrettävyyden validointi puolestaan on vaikeampaa testiautomaation avulla, sillä ne ovat varsin subjektiivisia. Tässä diplomityössä testiautomaation yhteydessä keskitytään hyväksymistestauksen kannalta erityisesti toiminnallisiin laatuominaisuuksiin ja niiden testaamiseen.

3.2 Testauksen tasot

Testauksen tasoja on useita ja usein ohjelmistojen kattavaan testaamiseen on suositeltavaa käyttää ohjelmistotuotantoprosessissa eri tasojen yhdistelmää. Ohjelmistojen testaus usein jaotellaan kolmeen erilaiseen menetelmään, jotka myös vaikuttavat eri testauksen tasojen käyttökelpoisuuteen. Erilaisia menetelmiä ovat mustalaatikkotestaus, harmaalaa-tikkotestaus ja valkoolatikkotestaus, jotka eroavat toisistaan yleisesti ottaen siinä, otetaanko tieto ohjelmistotuotteen sisäisestä toteutuksesta mukaan itse testaamiseen. Testauksen tasot esitetään kirjallisuudessa usein hieman eri muodoissa, mutta yleisesti ne jaetaan neljään eri tasoon, jotka voidaan kuvata pyramidin tasoavaruuteen projisoituna muotona.



Kuva 3.1. Testauksen tasot pyramidin muodossa

Pyramidimuodossa esitetyistä testauksen tasoista kaikkiin on mahdollista soveltaa testi-

automaatiota. Testauksen menetelmien osalta hieman yksinkertaistaen valkolaatikkotestauksen alaisuuteen kuuluvat yksikkötestaus ja integraatiotestaus sekä mustalaatikkotestauksen alaisuuteen kuuluvat järjestelmätestaus ja hyväksyntätestaus. Pyramidimuodossa alimpana kuvataan aina yksikkötestaus, joka on tasoista atomisiin ja myös luo vahvan pohjan kokonaisvaltaiselle testaamiselle. Noustessa pyramidissa ylöspäin, atomisuus häviää ja testattavana olevan kohteen laajuus sekä kompleksisuus kasvavat. Ylimpänä pyramidissa on hyväksymistestaus, joka on tarkoituksellista toteuttaa vaatimusmäärittelyn täyttävää valmista järjestelmää vastaan siten, että sen varmistetaan vastaavan loppukäyttäjän tarpeita. Monissa tapauksissa järjestelmätestauksen ja hyväksymistestauksen rajat saattavat olla epäselvät ja häilyvät. Tässä työssä hyväksymistestauksella tarkoitetaan käyttäjän hyväksyttämistestausta (UAT), jotta järjestelmätestauksen ja hyväksymistestauksen väliset eroavaisuudet tulevat lukijalle selkeästi esille.

Hyväksymistestaus on tämän diplomityön keskiössä ja siihen liittyvää teoriaa esitetään vielä laajemmin omassa hyväksymistestaus luvussaan 4. Seuraavissa kappaleissa esitetään vielä yksityiskohtaisemmin jokainen pyramidissa 3.1 esitetty testauksen taso, jotta lukijalle muodostuisi käsitys erityisesti hyväksymistestauksen suhteesta muihin testauksen tasoihin.

3.2.1 Yksikkötestaus

Yksikkötestauksen ajatuksena on testata ohjelmistotuotteen lähdekoodista löytyviä yksiköitä, kuten luokkia, funktioita tai moduuleita. Yksikkötestaus toteutetaan ohjelmiston toteuttavia pienimpiä yksikköjä vastaan ja sen avulla pyritään validoimaan, että jokainen yksikkö toimii siten kuin ne on ohjelmistokehityksen aloitusvaiheessa suunniteltu toimimaan. Yksikkötestaus eroaa muista testauksen tasoista siinä, että sen voi suorittaa ainoastaan ohjelmistokehittäjät tai muut ohjelmiston lähdekoodiin perehtyneet henkilöt. Yksikkötestaus on näin ollen teknisesti valkolaatikkotestausta. Yksikkötestausta tarvitaan, jotta voidaan pyrkiä varmistamaan, että ohjelmiston koostavat pienimmät yksiköt toimivat tarkoituksenmukaisella tavalla.

Yksikkötestauksen toteuttamiseen käytetään pääsääntöisesti jotakin tarkoitusta varten räätälöityä testikirjastoa, joissa on keskenään yleensä hyvin samankaltainen perusperiaate. Yksikkötestaukseen tarkoitetuissa testikirjastoissa löytyy usein yksittäisen testitapauksen kuvaava tietorakenne, esimerkiksi luokka, sekä siihen usein kuuluvat alustus- ja lopetusfunktiot. Näiden lisäksi varsinainen testauskoodi toteutetaan pääsääntöisesti käyttäen niin sanottuja testikirjaston tarjoamia assert-funktioita, joiden avulla voidaan esimerkiksi varmistaa, onko jokin muuttuja tietyssä arvossa.

Yksikkötestausta hyödynnetään usein myös ketterien menetelmien aihepiirissä, jossa ohjelmistotuotantoa voidaan toteuttaa muun muassa niin sanotulla testausvetoisella kehityksellä 3.5. Testausvetoisessa kehityksessä yksikkötestauksen osalta, ohjelmistokehittäjät laativat ensisijaisesti yksiköiden yksikkötestit ennen niiden toteuttamisen aloittamista.

Ohjelmistotestauksen tasojen pyramidissa ja hyvin toteutetussa ohjelmistotestauksen mo-

nitasisessa testauksessa tämä testauksen taso on usein testitapauksien määrässä kaikista laajin. Monitasisessa testauksessa yksikkötestaus luo tärkeän pohjan testaamiselle kokonaisuutena ja antaa tietoa ohjelmiston pienimpien yksiköiden toimivuudesta. Yksikkötestaus on myös paljon käytetty ja tärkeä osa testiautomaatiossa, sillä se varmistaa sovelluksen yksiköiden suunniteltua toimintaa.

3.2.2 Integraatiotestaus

Integraatiotestauksen ajatuksena on testata ohjelmistotuotteen toteuttavien eri komponenttien yhteensopivuutta niiden rajapintojen osalta. Integraatiotestaus toteutetaan ohjelmiston suunnitelmaa ja suunniteltua mallia vastaan. Integraatiotestauksen onnistunut toteuttaminen luo validoitavan perustan ohjelmiston toimimiseen ja sen koostamiseen kokonaisuutena, eri komponenteista koostuvana järjestelmänä. Integraatiotestausta tarvitaan, jotta voidaan varmistaa sovelluksen yksiköiden yhteensopivuus, joka ei pelkällä yksikkötestauksella tulisi muuten katetuksi.

Integraatiotestauksen kohteita voivat olla esimerkiksi luokkien ja moduulien väliset rajapinnat sekä web-sovelluksien api-ohjelmointirajapinnat. Integraatiotestauksen toteutuksen kannalta voidaan usein käyttää myös yksikkötestaukseen tarkoitettuja testikirjastoja ja työkaluja, mutta itse testitapauksien rakenne on silloin yksikkötestauksen testitapauksista merkittävällä tavalla erilainen. Integraatiotestauksessa testitapauksien rakenteeseen tulee assert-funktioiden lisäksi myös usein tarvetta jäljitellä rajapintojen tarjoamaa dataa. Rajapintojen tarjoaman datan jäljittelemiseen on olemassa useita valmiita työkaluja ja kirjastoja, joita integraatiotestauksen tapauksessa voi käyttää testitapauksien rakentamisen apuna.

Integraatiotestauksen yhteydessä puhutaan usein myös niin sanotusta savutestauksesta, jonka tarkoituksena integraatiotestauksen yhteydessä on koostaa toistuva, esimerkiksi päivittäinen, koontiversio ohjelmistosta ja testata sen kriittisten komponenttien yhteensopivuus. Integraatiotestaus on myös tärkeä osa testiautomaatiota, sillä sen avulla voidaan varmistaa sovelluksen yksiköiden, kuten esimerkiksi luokkien, komponenttien tai moduulien yhteensopivuus.

3.2.3 Järjestelmätestaus

Järjestelmätestauksen ajatuksena on testata kokonaista ja toimivaa järjestelmää, yhtenä suurena yksikkönä. Järjestelmätestausta tarvitaan, jotta voidaan varmistaa kokonaisen ohjelmiston toimivuus, jota ei muuten pelkällä yksikkötestauksella ja integraatiotestauksella saataisi täydellisellä varmuudella selville.

Järjestelmätestaukseen liittyy laajasti erilaisia testattavia laadullisia ominaisuuksia kuten toiminnallisuus, luotettavuus, käytettävyys, tehokkuus, ylläpidettävyys ja siirrettävyys [1]. Aiemmin testiautomaation tarkoitus kappaleessa 3.1 esitettiin että, edellä mainituista laadullisista ominaisuuksista kaikki eivät sovellu hyvin testiautomaation avulla testattaviksi.

Esitetyistä syistä johtuen, automatisoidulla järjestelmätestauksella voidaan testata edellä mainituista ominaisuuksista lähinnä ohjelmiston toiminnallisuutta, luotettavuutta ja tehokkuutta. Toiminnallisuutta voidaan testata käyttöliittymätestauksella, joka on mahdollista automatisoida käyttötapauksien muotoon. Luotettavuutta voidaan testata automaattisesti tietoturva testaaivien käyttötapauksien muodossa. Tehokkuutta voidaan testata automaattisesti lisäämällä aikaleimoihin perustuvaa tarkastelua testitapauksiin, sekä tehdä kuormitusta testaavia testitapauksia. Edellä mainituista muista laadullisista ominaisuuksista voidaan toki kuitenkin ylläpidettävyyttä ja siirrettävyyttä testata manuaalisesti.

Testauksen tasona järjestelmätestaus voi olla testiautomaation teknisen toteutuksen kannalta jopa hyvin samanlainen kuin sitä spesifimpi hyväksymistestaus. Usein kuitenkin hyväksymistestauksessa paneudutaan erityisesti vaatimusmäärittelyyn ja asiakaslähtöiseen testaamiseen, kun taas järjestelmätestauksessa voidaan testata esimerkiksi myös järjestelmän tehokkuutta tai tietoturva. Tämä on tosin täysin riippuvainen vaatimusmäärittelystä, joten jos tehokkuus ja tietoturva ovat ohjelmiston asiakasvaatimuksia niin niiltä osin järjestelmätestaus ja hyväksymistestaus lomittuvat. Joissakin yhteyksissä järjestelmätestaus ja hyväksymistestaus esitetään jopa yhteisenä testauksen tasona, etenkin silloin kun testiautomaation kannalta ne esimerkiksi edellä mainitulla tavalla muistuttavat kovasti toisiaan. Järjestelmätestaus osittain hyväksymistestauksen kanssa on erittäin merkittävä osa testiautomaatiosta, sillä sen avulla voidaan testata toteutettavaa järjestelmää kokonaisuutena.

3.2.4 Hyväksymistestaus

Hyväksymistestauksen ajatuksena on varmistaa toteutettavan ohjelmiston vaatimusten toimivuus erityisesti käytännön tilanteissa siten, että voidaan varmistaa vastaako ohjelmisto loppukäyttäjän tarpeita. Hyväksymistestaus toteutetaan ohjelmiston toimintoja kuvaavaa vaatimusmäärittelyä tai loppukäyttäjistä sekä heidän tarpeista laadittuja käyttötapauksia vastaan. Hyväksymistestauksen rooli testiautomaatiossa ja erityisesti jatkuvan integraation yhteydessä on osoittaa, voidaanko järjestelmä sellaisenaan julkaista loppukäyttäjille.

Hyväksymistestaus testaa lähinnä toiminnallisia laatuominaisuuksia ja usein se toteutetaan käyttöliittymätasolla testitapauksien muodossa. Poikkeuksena toiminnallisten ominaisuuksien lisäksi hyväksymistestauksessa voi olla mukana muita laadullisia ominaisuuksia vain jos ne ovat asiakastarpeiden mukaisia, joka on kuitenkin tavallisesta varsin poikkeava tilanne. Samassa asiayhteydessä puhutaan usein myös niin sanotusta e2e-testauksesta, eli päästä päähän -testauksesta. Päästä päähän -testauksessa on tarkoituksena toteuttaa testaaminen siten, että testaus pitää sisällään kaiken siltä väliltä mitä loppukäyttäjä voi tarpeidensa saavuttamiseksi tehdä ja nähdä aloittaessaan ohjelmiston käytön ja lopettaessaan sen käytön.

Testiautomaatio on äärimmäisen hyödyllinen hyväksymistestauksen osalla, koska sillä voidaan automatisoida ohjelmiston validointi ja hyväksyminen, sekä parhaimmillaan es-

tää puutteellisesti toimivan ohjelmiston julkaiseminen. Hyväksymistestausta tarvitaan myös, jotta voidaan testata ja validoida vaatimusten ja loppukäyttäjän tarpeiden mukaisten ominaisuuksien toimivuus kokonaisessa järjestelmässä.

3.3 Testitapaus ja testikokoelmat

Testitapaus on ohjelmistotestauksen automatisoimisen kannalta erittäin tärkeä käsite. Testitapaus kuvaa yhden testattavana olevan asian testaamiseksi suoritettavaa tai suoritettavia toimenpiteitä. Testitapauksen sisältämien toimenpiteiden suorittamisen tarkoituksena on saada selville täyttääkö se toimenpiteiden mukaiset ehdot ja toimiiko testattava asia oikein. Testitapauksella on usein alustusvaihe, varsinainen testausvaihe ja lopetusvaihe. Alustusvaiheessa testitapauksen vaativa ympäristö ja muuttujat alustetaan. Varsinaisessa testitapauksen testausvaiheessa suoritetaan testattavan asian testaukseen liittyvät toimenpiteet. Lopetusvaiheessa testitapauksen ajaksi muodostettu ympäristö usein tuhoetaan ja käytetyt resurssit nollataan, jotta ne eivät enää vaikuta muihin testitapauksiin.

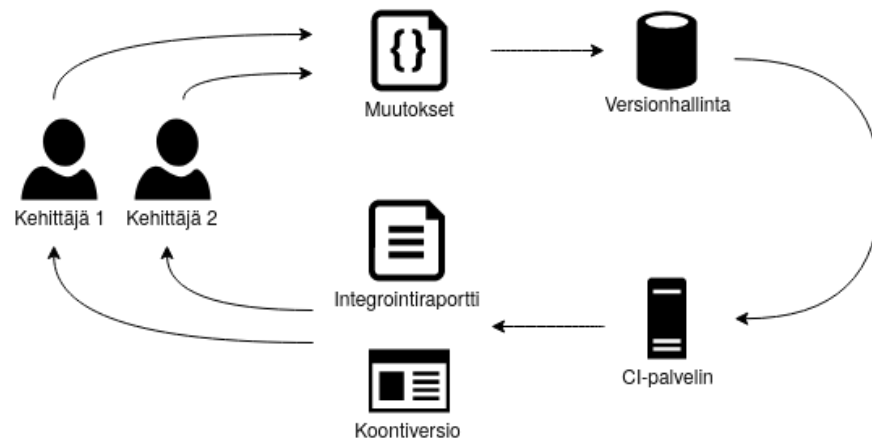
Testikokoelma on yksittäisistä testitapauksista koostuva ryhmitelty testitapauksien joukko. Testikokoelman saattaa kuulua myös sellaisia testitapauksia, joiden suoritusjärjestys on etukäteen määritetty. Suoritusjärjestyksellisissä testikokoelmissa voi esiintyä testitapauksia, jotka toimivat samassa testausympäristössä, muokaten ja jättäen jälkiä omasta suorituksesta. Myöhemmin suoritusjärjestyksessä tulevat testitapaukset voivat siinä tapauksessa hyötyä tai vaatia testausympäristön ominaisuuksia, jotka aiemmat testitapaukset ovat asettaneet. Testikokoelman sisältämät testitapaukset voidaan kuitenkin luonnollisesti laatia myös sellaisella tavalla, että jokainen testitapaus hoitaa yksityiskohtaiset alustustoimenpiteensä itsenäisesti.

Testitapauksia voidaan ryhmitellä samaan kontekstiin liittyviksi testikokoelmiksi tilanteesta riippuen monilla eri tavoin. Ryhmittelyn perusteen valitsemiseen kannattaa käyttää harkintaa, sillä testikokoelmien laajuus on helpomman hallittavuuden takia tärkeää. Yksi tapa ryhmitellä testitapauksia on käyttää ohjelmistojen laadullisia ominaisuuksia ryhmittelyn perustana. Tällaisessa ryhmittelyssä yksi kokoelma voi olla toiminnallisille testitapauksille ja toinen tehokkuutta mittaaville testitapauksille. Laadullisien ominaisuuksien mukaan tehty testitapauksien ryhmittely saattaa kuitenkin johtaa määrällisesti liian suuriin testitapauksien eroihin testikokoelmien kesken. Hyväksymistestauksen näkökulmasta tarkasteltuna testitapauksia on mahdollista ryhmitellä käyttöliittymän näkymiin perustuviin testikokoelmiin. Näkymäperusteinen ryhmittely on osaltaan looginen tapa jakaa testitapaukset eri testikokoelmiin, sillä jokainen käyttöliittymän näkymä voidaan tarvittaessa testata erikseen suorittamalla kyseisen testikokoelman testitapaukset. Tässä diplomityössä hyödynnetään perustavanlaatuisesti näkymäperusteista testitapauksien ryhmittelyä, koska sen avulla on mahdollista suorittaa testikokoelmien näkymäperusteinen priorisointi työssä myöhemmin esitettävää painotettua verkkoa hyödyntäen.

3.4 Jatkuva integrointi

Testiautomaation rakentaminen manuaalisen testaamisen sijaan mahdollistaa sen liittämisen osaksi jatkuvaa integrointia. Lisäksi useissa ohjelmistotuotannon prosesseissa pelkkä manuaalinen testaus kävisi selkeästi automatisoitujen koonti- tai julkaisuputkien periaatteita vastaan. Testiautomaation tarkoitus kappaleessa 3.1 aiemmin esitettiin testiautomaation ja manuaalisen testauksen eroa hyötyjen ja haittojen näkökulmasta. Testiautomaation toteuttaminen testitapauksien muodossa on jo itsessään testiautomaatiota, mutta käsitettä voidaan kuitenkin laajentaa, että myös jatkuva integrointi liittyy oleellisesti testiautomaation toteuttamiseen varsinkin nykyaikana ja erityisesti ketteriin menetelmiin painottuvassa ohjelmistokehityksessä.

Jatkuvalla integroinnilla tarkoitetaan versiohallintaisessa ohjelmistokehityksessä väistämättömän integrointiprosessin muuntamista luonnostaan jatkuvaksi. Ohjelmistokehityksessä integrointiprosessi tulee vastaan, kun eri ohjelmistokehittäjät tai tiimit toteuttavat muutoksia tai uusia ominaisuuksia kehitettävänä olevaan ohjelmistotuotteeseen. Tällaisessa tilanteessa yksittäiset ohjelmistokehittäjät tai tiimit toteuttavat uutta ohjelmakoodia toisistaan irrallaan siihen asti, kunnes muutokset tai ominaisuudet tulee yhdistää yhdeksi kokonaiseksi kehityksen kohteena olevaksi ohjelmistotuotteeksi, jota prosessina kutsutaan integrointiprosessiksi. Jatkuvan integroinnin tarkoituksena on nopeuttaa integrointiprosessia ja muuttaa ohjelmistokehityksessä käytössä olevia periaatteita siten, että siitä tulee luonnostaan jatkuvaa. Jatkuvan integroinnin toteuttaminen tarvitsee teknisesti sen mahdollistavan versiohallintajärjestelmän ja varsinaisen jatkuvan integroinnin palvelimen.



Kuva 3.2. Jatkuvan integroinnin perusperiaate on iteratiivinen

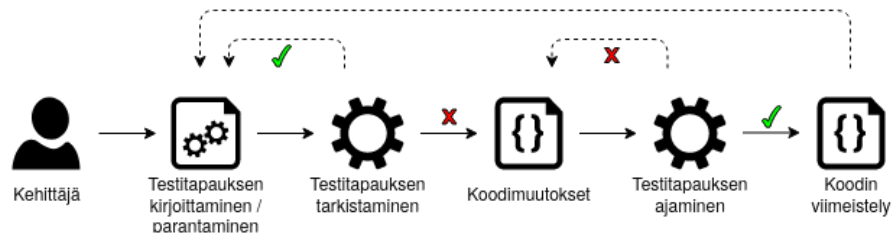
Esimerkkinä versiohallintajärjestelmänä voidaan käyttää nykyaikana suosittua git-ohjelmistoa ja jatkuvan integroinnin palvelimena esimerkiksi GoCD-ohjelmistoa. Perusideana jatkuvassa integroinnissa on konfiguroida jatkuvan integraation mahdollistava palvelinohjelmisto siten, että se kuuntelee versiohallintaan tulevia muutoksia ja suorittaa integrointiprosessin jatkuvasti aina muutoksia huomattuaan. Versiohallintaan tulevat muutokset

voidaan jatkuvan integraation osalta kuunnella ajastetusti tietyin väliajoin tai aidosti jatkuvalla tavalla käyttämällä esimerkiksi web-koukkuja, jotka tiedottavat jatkuvan integraation palvelimelle versionhallintaan saapuneista muutoksista. Jatkuvan integroinnin yhden iteraatiokerran integrointiprosessin tuloksena on tarkoituksena tarjota periaatteeltaan sama lopputulema kuin mitä se olisi manuaalisella integrointiprosessillakin. Jatkuvan integroinnin mahdollistava konfiguraatio sisältää aina jonkinlaisen koontiputken tai useita koontiputkia, joissa rakennetaan koontiversio kehitettävän ohjelmiston lähdekoodista. Koontiputki voi sisältää esimerkiksi ohjelman lähdekoodien kääntämisen asiaan sopivalla kääntäjällä. Kääntämisen lisäksi koontiputkeen on tässä vaiheessa mahdollista ja erittäin kannattavaa yhdistää testiautomaatiota, kuten esimerkiksi automaattisten yksikkötestien suorittaminen ennen kääntämistä ja hyväksymistestien suorittaminen valmiille koontiversiolle kääntämisen jälkeen.

Jatkuvan integroinnin yhteydessä suoritettavat testikokoelmat 3.3 ja niiden sisältävät testitapaukset ovat erittäin järkeviä toteuttaa, sillä ne esimerkiksi parantavat ohjelmistokehityksen ja lopputuotteen luotettavuutta ja laatua. Jatkuvan integroinnin sisältämästä koontiputkesta saadaan hyödyllistä palautetta ja raportteja integrointiprosessin onnistumisesta, joka voidaan ohjata pääasiassa ohjelmistokehittäjille sekä myös muillekin sidosryhmille. Jatkuvalla integroinnilla itsessään on myöskin paljon sen käyttöönoton antamia hyötyjä, kuten esimerkiksi toteutettujen muutosten tai toimintojen integrointitiheyden kasvattamisen tuomat edut. Jos muutosten tai toimintojen integroiminen on perinteisessä ohjelmistokehityksessä tehty harvoin, kuten esimerkiksi viikoittain, niin jatkuva integroiminen korjaa sen tuomat haasteet turhan laajasta integrointiprosessista ja mahdollisesta ohjelmistokoodin hajoamisesta. Tällaisissa tapauksissa ohjelmakoodi voi sisältää epäyhteensopivia moduuleita tai muita rajapintoja sekä mahdollisuuden käännettävien lähdekoodien kääntämisen onnistumisesta.

3.5 Testausvetoinen kehitys

Perinteisesti testiautomaatio on soveltunut hyvin vain vakaille ohjelmistoille ja niiden regressiotestaamiseen. Nykypäivänä ohjelmistokehitys on siirtynyt suunnitelmapohjaisista prosesseista iteroiviin ketteriin ohjelmistotuotannon prosesseihin. Näihin testiautomaatio on soveltunut huonosti, kun testattavaa ohjelmistoa tai lisättyä toiminnallisuutta ei ole vielä olemassa. Tähän ongelmaan on kehitetty niin sanottu testausvetoinen kehitys, jossa testitapaukset suunnitellaan ja toteutetaan ennen varsinaisen ohjelmiston tai toiminnon toteutuksen toteuttamista.



Kuva 3.3. Testausvetoisen kehityksen vaiheet

Testausvetoinen kehityksen sisältämät vaiheet 3.3 alkavat testitapauksien luomisesta ja niiden tarkastamisesta. Tarkastaminen tapahtuu siten, että testitapaukset suoritetaan sillä oletuksella, että niiden täytyy tässä vaiheessa epäonnistua. Alkuvaiheen testitapauksien luomisen jälkeen ohjelmistokehittäjät kehittävät ohjelmistoa tekemällä siihen muutoksia, ihanteellisesti testitapauksien kokoisia paloja kerrallaan. Kun koodimuutoksia on syntynyt, riippuen ohjelmistotuotannossa käytössä olevasta integrointiprosessista, ajetaan testitapaukset manuaalisesti tai jatkuvan integroinnin avulla. Integrointiprosessista saadaan palautetta, jonka mukaan ohjelmakoodia korjataan tai viimeistellään. Testausvetoisella kehityksellä pyritään nopeuttamaan ohjelmistokehitysprosessia verrattuna perinteisiin ohjelmistotuotannon menetelmiin. Tämän jälkeen testausvetoista kehitystä käyttävässä ohjelmistotuotantoprosessissa siirrytään takaisin testitapauksien luomiseen ja parantamiseen sekä aloitetaan toinen iteraatiokierros mikäli ohjelmisto ei vielä ole valmis.

Testausvetoisessa kehityksessä testitapaukset siis laaditaan jo varhaisessa vaiheessa jolloin niiden tekeminen saattaa usein olla liiketoiminnan näkökulmasta helpommin perusteltavaa liiketoiminnan johdolle. Tämän lisäksi testitapauksien kirjoittaminen etukäteen luo kattavat testikokoelmat jo alusta alkaen, joita voidaan hyödyntää iteratiivisesti ohjelmistotuotteesta riippuen usein hyvinkin pitkään, etenkin jos niihin tehdään tarvittavaa hienosäätöä ohjelmistokehityksen aikana. Ohjelmistokehittäjät voivat kehittää helposti hallittavissa olevia testitapauksien rajaavia kokonaisuuksia, jolloin ohjelmistotuote valmistuu ikään kuin pala kerrallaan. Itse ohjelmistokehitys on testausvetoisessa kehityksessä siis iteratiivista ja näin ollen testitapauksien suorittamisesta saadaan palautetta ja raportointia koko ohjelmistotuotantoprosessin aikana. Testausvetoinen kehitys kuuluu ohjelmistotuotannossa vahvasti ketterien menetelmien alaisuuteen ja on kasvattanut suosiotaan ketterien menetelmien mukana.

4 HYVÄKSYMISTESTAUS

Tässä luvussa esitetään perusteet ja tarvittavat tiedot hyväksymistestauksesta, johon testauksen tasoista tässä diplomityössä keskitytään. Ensin esitetään hyväksymistestauksen tarkoitus, jonka jälkeen keskitytään hyväksymistestausvetoiseen kehitykseen ja sen esittelemiseen ohjelmistotuotannollisena menetelmänä. Hyväksymistestausvetoisen kehityksen jälkeen käydään läpi web-sovelluksien yhteydessä huomioitavia erityispiirteitä hyväksymistestauksen toteuttamisen näkökulmasta. Web-sovelluksien erityispiirteiden jälkeen esitetään erityisesti tässä diplomityössä käytettyjä, mutta kuitenkin yleisiä hyväksymistestauksen työkaluja. Hyväksymistestauksen työkaluista siirrytään esittämään niistä koostuvan testausjärjestelmän rakenne ja sen käyttöönottoaminen osaksi ohjelmistotuotantoprosessia. Testausjärjestelmän rakenteen ja käyttöönottoamisen lisäksi käydään omassa kappaleessaan läpi myös testitapauksien rakentaminen painottuen testausjärjestelmässä käytettyihin työkaluihin. Lopuksi esitetään yleisestikin ottaen testitapauksiin tärkeästi liittyvä priorisointiongelma, pyritään esittämään miksi sen ratkaiseminen on tärkeää ja esitetään erilaisia menetelmiä sen ratkaisemiseen.

4.1 Hyväksymistestauksen tarkoitus

Hyväksymistestaus on testauksen tasoista tärkeimpiä, sillä sen ollessa kattava, voidaan verifioida ohjelman toiminta korkealla tasolla saaden samalla varmuus siitä, että hyväksymistestausta alemmilla tasoilla testattavat asiat toimivat riittävän oikein. Hyväksymistestauksen tarkoituksena on varmistaa toteutettavan ohjelmiston vaatimusten toimivuus erityisesti käytännön tilanteissa siten, että voidaan varmistaa vastaako ohjelmisto loppukäyttäjän tarpeita. Hyväksymistestaus antaa vastauksen siihen, toimiiko toteutettu järjestelmä loppukäyttäjän tarpeiden mukaisesti ja loppukäyttäjän näkökulmasta oikein. Hyväksymistestauksen sanotaan olevan muodollista testaamista, jossa käyttäjän tarpeet, vaatimukset ja liiketoimintaprosessit otetaan huomioon selvittäessä täyttääkö järjestelmä hyväksymisen kriteerit ja sallii auktorisoidun tahon päättää hyväksytäänkö järjestelmä [2] julkaistavaksi. Ohjelmistotestauksen tekniikoiden näkökulmasta hyväksymistestaus on mustalaatikkotestausta, eli testauskohdetta testataan tietämättä sen teknisestä toteutuksesta. Hyväksymistestauksen painoarvo on asiakasperusteisessa vaatimusmäärittelyssä ja loppukäyttäjän tarpeiden kartoittamisessa. Testiautomaation osalta hyväksymistestausta varten voidaan rakentaa testitapaukset, joiden avulla voidaan keskittyä varmistamaan loppukäyttäjille tarpeellisten toimintojen toteutuminen testitapauksien suorittamisen jälkeen. Hyväksymistestauksen osalta testitapauksia voidaan toteuttaa niin sa-

notulla päästä päähän -periaatteella, jossa testattavaa järjestelmää testataan siten kuin loppukäyttäjä sitä käyttäisi. Hyväksymistestauksessa ei anneta painoarvoa esimerkiksi kosmeettisille tai kirjoitusvirheille, vaan pyritään selvittämään loppukäyttäjille oleellisten ja tarpeellisten toimintojen toteutuminen.

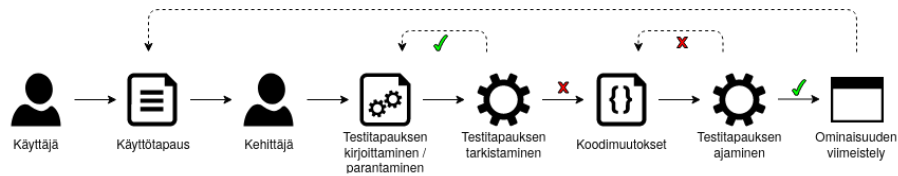
Hyväksymistestaus on aiemmin esitetyistä testauksen tasoista 3.2 viimeinen ja sen suorittamisen jälkeen saadaan tieto, onko järjestelmä toteutuksen osalta sellaisenaan valmis julkaistavaksi. Perinteisesti hyväksymistestauksen lähtökohtia ovat selvät hyväksymisvaatimukset sekä julkaisukelpoinen toteutus joka voi sisältää vain kosmeettisia tai kirjoitusvirheitä. Hyväksymisvaatimukset voivat olla esimerkiksi liiketoiminnallisia käyttötapauksia, prosessivirtauskaavioita sekä vaikkapa ohjelmiston vaatimusmäärittely. Testiautomaatiota varten käytettävästä testialustasta riippuen hyväksymistestauksen käyttötapaukset voidaan muodostaa joko osittain tai suoraan testitapauksiksi. Hyväksymistestaukseen usein pyydetään osallisiksi ohjelmistokehittäjien lisäksi myös muita sidosryhmiä ja toisinaan jopa loppukäyttäjiä. Keskeistä on, että loppukäyttäjiltä hankitaan tieto tarvittavista ja toteutettavista ominaisuuksista, kun taas muut sidosryhmät kuten esimerkiksi johtoryhmä voivat tehdä liiketoiminnallisia päätöksiä hyväksymistestauksen onnistumisen osalta ja esimerkiksi peruuttaa julkaisun. Hyväksymistestaus siis antaa mahdollisuuden korjata usein liiketoiminnallisestakin näkökulmasta merkittävät toiminalliset virheet ennen järjestelmän julkaisua loppukäyttäjille.

Kehittäjien käsitys järjestelmän toiminnallisuudesta ja sen vaatimuksista voi kuitenkin olla usein hyvinkin erilainen kuin loppukäyttäjien. Hyväksymistestauksen avulla voidaan tätä lievittää tätä ongelmaa, ja saada ohjelmistokehittäjät loppukäyttäjien kanssa vaatimusmäärittelyn suhteen samalle aaltopituudelle. Testiautomaation avulla toteutettavalla toistuvalla hyväksymistestauksella varmistetaan, että järjestelmä toteuttaa loppukäyttäjän tarpeet vielä järjestelmään tehtyjen muutoksien jälkeenkin. Hyväksymistestauksen testitapaukset tarkoituksenmukaisesti heijastavat suoraan loppukäyttäjien tarpeita, jonka avulla ohjelmistokehittäjät ja muut sidosryhmät voivat tehokkaasti varmistaa järjestelmän valmiuden ja sen hetkisen tilan. Hyväksymistestauksella siis saadaan katsaus ohjelmiston valmiudesta sen vaatimukseen ja loppukäyttäjien toiminnallisiin tarpeisiin nähden.

4.2 Hyväksymistestausvetoinen kehitys

Hyväksymistestausvetoisen kehityksen tarkoituksena, kuten testausvetoisessakin kehityksessä 3.5 on toteuttaa ohjelmistotuotannollinen prosessi laatien toistettavasti suoritettavat testitapaukset ennen ohjelmiston varsinaista toteutusta. Hyväksymistestausvetoisessa kehityksessä tämä tarkoittaa käytännössä sitä, että ennen toteutusta luodaan tarvittavat ohjelmiston asiakasvaatimuksia palvelevat hyväksymistestit, jotka ohjelmiston on tarkoitus läpäistä sen julkaisemisen hyväksymiseksi. Hyväksymistestausvetoisen kehityksen sanotaan olevan yhteistyöhön perustuva lähestymistapa kehitykseen, jossa tiimi ja asiakkaat käyttävät asiakkaiden oman ympäristön kieltä ymmärtääkseen heidän vaatimukset, jotka muodostavat pohjan komponentin tai järjestelmän testaamiseen [2]. Tar-

vittavat ohjelmiston hyväksymistestit suoritetaan iteratiivisesti ohjelmistokehitysprosessin aikana ja se tarkoittaa käytännössä jatkuvan integraation 3.4 ottamista käyttöön ohjelmistokehityksessä. Hyväksymistestausvetoinen kehitys on erittäin hyödyllinen ohjelmistokehityksessä käytetty menetelmä, sillä kehitysvaiheessa on aina tarkasti tiedossa, vastaako ohjelmiston sen hetkinen tila asiakasvaatimuksia ja kuinka hyvin se niiden täyttämisessä onnistuu.



Kuva 4.1. Hyväksymistestausvetoisen kehityksen vaiheet

Hyväksymistestausvetoinen kehitys voidaan luokitella ketteräksi ohjelmistokehitysmenetelmäksi, kuten sen yläkäsitteenä oleva testausvetoinen kehityskin 3.5. Hyväksymistestausvetoinen kehitys on testausvetoisen kehityksen kanssa peruseriaatteeltaan samanlainen, mutta ennen ohjelmistokehityksen aloitusta asiakasvaatimukset kartoitetaan ja ohjelmiston hyväksyttävyyttä määritellään. Hyväksymistestitapaukset kirjoitetaan testausvetoisen kehityksen mukaisesti ennen toteutusta ja ohjelmistokehitys itsessään noudattaa iteratiivisesti testausvetoista kehitystä, vaikkakin hyväksymistestaus on perinteisesti vaatinut lähes valmista järjestelmää. Asiakasvaatimukset määritetään usein käyttötapausten muodossa ja hieman testialustasta riippuen ne voidaan kirjoittaa suoraan testitapauksien muotoon. Hyväksymistestausvetoisessa kehityksessä ohjelmistokehitystä ohjaavat asiakasvaatimukset ja loppukäyttäjien tarpeiden toteutuminen, jotka ovat hyvin usein toiminnallisia vaatimuksia. Hyväksymistestausvetoisessa kehityksessä mitataan jatkuvasti käyttötapausten muodossa validoitavien haluttujen ominaisuuksien toteutumisesta. Peruseriaate on kirjoittaa asiakasvaatimus tai käyttötapaus testitapauksen muotoon, toteuttaa testitapausta, ajaa testitapausta läpäisemättömänä, toteuttaa ominaisuus, ajaa testitapausta läpäisevänä, refaktoroida toteutus ja siirtyä takaisin seuraavaan käyttötapaukseen. Käyttötapausta koostuu rakenteellisesti usein tilanteesta, motivaatiosta ja halutusta lopputuloksesta. Esimerkki käyttötapauksesta voi olla: *käyttäjänä, haluan sisäänkirjautumisen jälkeen voida avata premium ominaisuudet tekemällä sovelluksensisäisen oston.*

Hyväksymistestausvetoisessa kehityksessä hyväksymistestit ovat hyödyllistä pilkkoa pieniin hallittaviin kokonaisuuksiin, jolloin voidaan iteratiivisesti toteuttaa valmiiksi tietyn testitapauksen mukainen ominaisuus, joka vastaa jotakin käyttötapausta tai loppukäyttäjän tarvetta. Hyväksymistestauksessa testitapausta voi olla esimerkiksi käyttäjän tietojen muuttamisen varmistaminen, kuten tason läpäiseminen pelisovelluksessa, joka muuttaa käyttäjän edistystä. Menetelmänä hyväksymistestausvetoisen kehityksen tarkoituksena on onnistua vastaamaan loppukäyttäjän tarpeisiin tehokkaasti ja hyvin ottamalla tarpeet huomioon jo ennen toteutuksen aloittamista. Menetelmän avulla myös luodaan ymmärrystä ohjelmistotuotteen valmiuden määritelmästä, kun eri sidosryhmän voidaan saada sen suhteen samalle aaltopituudelle. Hyväksymistestausvetoinen kehitys on lisäksi erit-

täin hyödyllistä, sillä jatkuva testaaminen antaa mahdollisuuden haluttujen ominaisuuksien toteutumisen validoimiselle menetelmän jokaisen iteraation koontiversiossa.

4.3 Web-sovelluksien erityispiirteet

Web-sovelluksilla on omia erityispiirteitä, jotka vaikuttavat testitapauksien laatimiseen. Nykypäivänä web-sovellukset ovat kasvaneet kompleksisuudessa ja front-end puolen toteutuksia tarkasteltaessa web-sovellukset usein muistuttavat jo perinteisiä dynaamisia työpöytäsovelluksia. Web-sovelluksia päivitetään usein tiheään tahtiin, jolloin niille on suuri tarve luoda testiautomaatiota, jota hyödyntäen voidaan varmistaa, että ne toimivat oikein muutoksien jälkeenkin.

Hyväksymistestauksen priorisoimisen osalta tärkeä web-sovelluksien erityispiirre liittyy käyttöliittymiin ja DOM-dokumenttiobjektimalliin. Dokumenttiobjektimallin avulla verkkoselaimet esittävät käyttöliittymän ja siinä näkyvän sisällön. Tämän lisäksi dokumenttiobjektimalli mahdollistaa käyttöliittymässä olevien elementtien valitsemisen, jota hyödynnetään vahvasti testitapauksien kirjoittamisessa.

Navigointi ja navigointiketjut ovat myös yksi web-sovelluksien erityispiirre. Historiallisesti verkkosivuilla navigointi tapahtuu niin sanottujen hyperlinkkien avulla, verkkosivujen itse ollessa hypertekstiä. Tämä historiallinen lähestymistapa on edelleen käytössä ja web-sovelluksissa on lähes poikkeuksetta useita hyperlinkkejä joiden avulla navigoiminen luo navigointiketjuja, joissa edelliseen sivuun tiedetään palata. Hyperlinkkien avulla tapahtuva navigointi ja navigointiketjut ovat sellainen erityispiirre, joka on hyvä tiedostaa myös hyväksymistestauksen testitapauksia rakentaessa.

Web-sovelluksien syötteet ja niiden yhteyteen liittyvä tietoturva ovat sellainen erityispiirre joka vaatii suurta huomiota. Web-sovelluksien syötteisiin on perinteisesti liittynyt paljon haavoittuvuuksia, kuten esimerkiksi XSS-hyökkäykset ja SQL-injektiot. Web-sovelluksien hyväksymistestauksen testitapauksiin on hyvä sisällyttää syötteisiin liittyvää testaamista, joissa tietoturva pidetään mielessä.

Erilaisia web-sovelluksen loppukäyttäjien asiakasympäristöjä on huikean paljon, joka kannustaa moniselaimellisen testauksen rakentamiseen. Näissä ympäristöissä on omat verkkoselaimensa, näyttöresoluutiot ja selainasetukset, jotka saavat saman web-sovelluksen toimimaan eri tavoilla eri ympäristöissä ja luovat siten usein jopa päänvaivaa ohjelmistokehittäjille. Etenkin web-käyttöliittymiin keskittyessä testitapauksiin on hyvä sisällyttää erilaisia näyttöresoluutioita, kuvankaappauksien ottamista ja selainasetuksista esimerkiksi JavaScript-ominaisuuksien estäminen.

Web-sovelluksien käyttöliittymien testaaminen ja yleisesti ottaen kaikenlaisien käyttöliittymien testaaminen on perinteisesti tapahtunut manuaalisesti. Nykyään web-sovelluksia voidaan testata niin sanotun päätteettömän testauksen keinoin. Web-sovelluksien päätteettömässä testauksessa verkkoselaimen, näyttöresoluution ja selainasetuksien muodostama asiakasympäristö rakennetaan virtualisoinnin avulla. Virtualisoinnista vastaa jo-

ko verkkoselaimet itse tai voidaan käyttää käyttöjärjestelmätasolla näyttöpalvelimen protokollan toteuttavaa virtualisointiratkaisua. Virtualisoitu asiakasympäristö rakennetaan siten, että se päätteettömänä vastaa täysin päätteellistä vaihtoehtoa ja siitä voidaan ottaa esimerkiksi kuvankaappauksia, vaikka mitään ihmisen aistittavaa ei olisikaan näkyvillä.

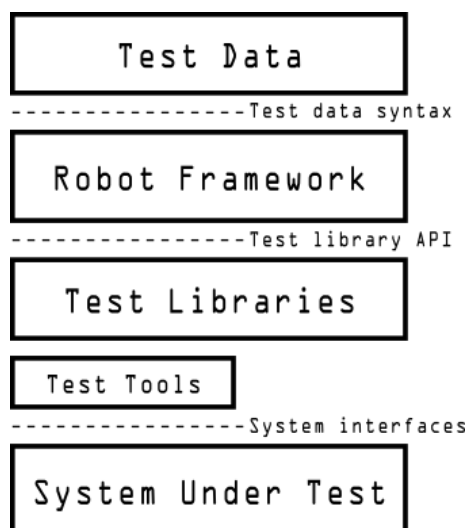
4.4 Hyväksymistestauksen työkaluja

Tässä kappaleessa esitetään diplomityötä tehdessä käytettyjä ja osin myös varsin yleisiä testiautomaation mahdollistavia työkaluja, kukin omassa alakappaleessaan. Ensin esitetään hyväksymistestauksen automatisoimisen kannalta kaikista tärkeimmät työkalut, eli testialustana käytettävä Robot Framework ja web-sovelluksien kanssa vuorovaikuttamisen automatisoimisen mahdollistava Selenium-kirjasto. Lisäksi esitetään kolme muuta tärkeää työkalua, joiden avulla voidaan rakentaa kokonainen ohjelmistotuotannon prosessiin integroitavissa oleva hyväksymistestausjärjestelmä. Toteutuksessa GoCD vastaa jatkuvan integroinnin tarjoamisesta, Xvfb vastaa päätteettömän testauksen tarjoamisesta ja Docker vastaa työkalujen virtualisoinnista ja säiliöinnistä, jolloin työkaluista saadaan rakennettua yhtenäinen kokonaisuus. Tämän diplomityön tuloksena syntynyt hyväksymistestausjärjestelmä koostuu samoista työkaluista kuin seuraavissa kappaleissa on esitetty.

4.4.1 Robot Framework

Robot Framework on geneerinen avoimen lähdekoodin testialusta hyväksymistestaukseen, hyväksymistestausvetoiseen kehitykseen ja robottisten prosessien automaatioon [3]. Robot Framework:in avainsanaperustainen syntaksi on helposti ymmärrettävä, luettava ja selkeä. Testialustan etuna on helppo lähestyttävyyys, eikä sen päälle rakennettujen testitapauksien ymmärtäminen vaadi ohjelmointikielten ymmärtämistä. Robot Framework on Python-perustainen testialusta ja se on helppo asentaa, sitä on helppo ymmärtää, sillä on kattava dokumentaatio ja se on helppoa ottaa käyttöön.

Robot Framework:issa on sisäänrakennettu tuki ulkoisille kirjastoille ja sen kattavasta dokumentaatiosta löytyy tietoa omien avainsanojen ja omien kirjastojen tekemiseen. Lisäksi Robot Framework on todella suosittu, joka näkyy muun muassa siitä, että sisäänrakennettujen ominaisuuksien lisäksi ulkoisia kolmansien osapuolien kirjastoja löytyy alustalle paljon. Robot Framework tukee muuttujien käyttöä testitapauksien rakentamisessa, joilla voi hieman lisätä kompleksisuutta ja logiikkaa omiin testitapauksiin. Robot Framework:ista löytyy myös tuki dataperustaisien testitapauksien rakentamiseen, joille annetaan eri syötteitä sisältävää testidataa. Testitapauksia voi myös ryhmitellä testikokoelmiin käyttämällä tagejä testitapauksien sisällä.



Kuva 4.2. Robot Framework alustan arkkitehtuuri

Robot Framework:illä rakennettuja testitapauksia voidaan ajaa komentoriviltä robot-komennolla. Testitapauksien ajaminen tulostaa komentoriville yksinkertaisen raportin testitapauksen onnistumisesta ja lisäksi tallettaa varsin yksityiskohtaisen ja selkeän testitaportin ajetuille testitapauksille. Testiraportit ovat erittäin hyvin tehtyjä ja html-pohjaisia, joka tarkoittaa, että ne voidaan helposti integroida osaksi jatkuvan integraation koontiputkia.

Yhtenä heikkoutena Robot Framework:issa on tuen puuttuminen ohjelmistokieliä hyödyn-tävillä testialustoilla löytyville kontrollirakenteille, joita esiintyy esimerkiksi yksikkötestaukseen tarkoitetuissa testialustoilla. Robot Framework on selkeästi vain hyväksymistestauk-sen testitapauksien rakentamista varten tarkoitettu testialusta ja siinä se on erinomainen vaihtoehto testitapauksien rakentamiseen.

4.4.2 Selenium

Selenium on suosittu avoimen lähdekoodin Apache 2.0 lisensoitu työkalu ja kirjastokoelma verkkoselainten automatisoimiseen. Ensisijaisesti se on tarkoitettu web-sovelluksien automatisoimiseen testaustarpeita varten. Erityisen hyvin Selenium soveltuu hyväksymistestauksen testiautomaation rakentamiseen, sillä sen avulla automatisoidaan web-sovelluksien käyttöliittymissä tehtäviä toimenpiteitä. Selenium on ThoughtWorks yhtiön kehittämä verkkoselainten automatisoimiseen tarkoitettu työkalujen ja kirjastojen kokoelma ja se on saatavilla Windows, Linux ja MacOS alustoille. Sama yhtiö on toteuttanut myös tässä diplomityössä myöhemmin esitettävän GoCD-ohjelmiston, jota voidaan käyttää jatkuvan integroimisen ja julkaisemisen rakentamiseen 4.4.5.

Selenium tuoteperheeseen kuuluvat Selenium WebDriver, Selenium IDE ja Selenium Grid komponentit. Selenium WebDriver on varsinainen web-sovelluksien automatisoimiseen käytettävä ohjelmisto, jota myös tässä diplomityössä Selenium tuotteista käytetään. Selenium IDE on kehitysympäristö ohjelmistokehittäjille ja testaajille, jota voidaan halutessaan käyttää testitapauksien rakentamiseen. Selenium Grid on järjestelmä, jonka

avulla voidaan Selenium pohjaisten testitapauksien suorittaminen skaalautuvasti hajauttaa useille eri etäkoneille. Tässä diplomityössä ei ole käytetty Selenium Grid -järjestelmää vaan testitapauksien suorittamiseen tarvittavat ohjelmistot on säiliöity Docker-työkalua käyttäen, joka mahdollistaa tarvittaessa skaalautuvuuden 4.4.4.

Selenium on todella tärkeä osa web-sovelluksien testiautomaation rakentamista, sillä se pohjimmiltaan mahdollistaa web-sovelluksien käyttöliittymien käsittelyn automatisoimisen. Selenium-työkalua voidaan käyttää erityisesti hyväksymistestauksen testitapauksien automatisoimiseen suoraan Selenium IDE:n avulla nauhoittaen testitapauksia tai kirjoittaen ne Selenium-skriptauskielellä. Selenium on joustava työkalu ja se tarjoaa Selenium Client API -rajapinnan, jonka avulla sitä voidaan käyttää muistakin ohjelmointikielistä, kuten C#, JavaScript tai Python.

Tässä diplomityössä Selenium työkalua käytetään Robot Framework:in yhteyteen integroituna ulkoisena kirjastona. Robot Framework:ille on saatavilla SeleniumLibrary niminen kirjasto, josta löytyy Robot Framework:in syntaksin mukaisesti määritellyt avainsanat verkkoselainten ohjaamiseen Selenium-pohjaisesti.

4.4.3 Xvfb

Xvfb, eli X virtual framebuffer, on X-näyttöpalvelimen protokollan toteuttava virtuaalinen X-näyttöpalvelin. X-näyttöpalvelimen tehtävä on mahdollistaa graafisten ohjelmien toiminta käyttöjärjestelmän ytimen päällä, jossa X-palvelin ja X-asiakasohjelmat kommunikovat keskenään sekä X-palvelin hoitaa ytimen kautta näytön ja syöttölaitteiden käsittelyn. Xvfb ei tulosta mitään näytölle, vaan kaikki näytölle normaalisti tulostuva graafisia käyttöliittymiä sisältävä sisältö on ajonaikaisessa tietokoneen muistissa. Xvfb toimii aivan kuten tavallinenkin X-näyttöpalvelin, eli vastaa X-ohjelmien pyyntöihin ja hoitaa niihin liittyvän tapahtumien ja virheiden käsittelyn.

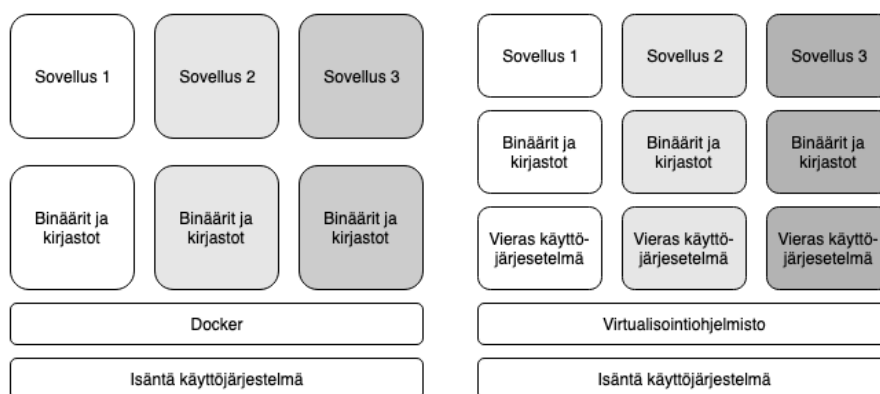
Xvfb soveltuu web-sovelluksien hyväksymistestauksen automatisointiin mahdollistaen päätteettömän testaamisen testitapauksille. Päätteetöntä testausta voidaan toteuttaa myös verkkoselaimiin rakennettujen ominaisuuksien avulla, mutta Xvfb:n suurena etuna niihin on se, että sitä voidaan käyttää mihin tahansa graafiseen ohjelmaan. Päätteettömän testauksen mahdollistaminen on erittäin tärkeää, sillä se mahdollistaa myös käyttöliittymätestauksen suorittamisen jatkuvan integroinnin palvelimilla, jossa ei graafista ympäristöä ajon aikana muuten olisi.

Yhtenä Xvfb:n heikkoutena on, että se on saatavilla vain UNIX-pohjaisiin käyttöjärjestelmiin, kuten Linux ja MacOS. Näin ollen esimerkiksi Windows-alustalla toimivaa Internet Explorer -selainta ei voida natiivisti testata.

Robot Framework:ille on saatavilla XvfbRobot niminen kirjasto, jota tämän diplomityön toteutuksessa käytettiin. XvfbRobot on kirjasto, josta löytyy Robot Framework:in syntaksin mukaisesti määritellyt avainsanat Xvfb-palvelimen kanssa kommunikointiin.

4.4.4 Docker

Docker on säiliöintityökalu, jonka avulla on mahdollista määrittää, rakentaa ja ajaa säiliöiden muotoon konfiguroituja sovelluksia. Docker muistuttaa virtuaalikoneita, mutta se on kevyempi ja optimaalisempi, sillä se jakaa käyttöjärjestelmän ytimen eri säiliöiden kesken ja virtualisoi vain sovellusympäristön jonka säiliön määrittävä konfiguraatio sisältää. Säiliöiden sisään voidaan paketoita kaikki kokonaisen sovelluksen tarvitsemat ohjelmistot, kirjastot, ympäristöt, riippuvuudet ja itse sovelluksen ohjelmakoodi. Rakentamalla säiliön ja käynnistämällä sen, voidaan sitä käyttää konfiguraatioltaan samanlaisena eri ympäristöissä joissa Docker-ohjelmisto on saatavilla.



Kuva 4.3. Dockerin ja virtuaalikoneen eroavaisuus

Dockerfile:n avulla voidaan luoda räätälöity säiliö, josta voidaan rakentaa yksi tai useampia instansseja. Räätälöidyn säiliön etuna on etenkin se, että sen avulla saadaan aikaan sovellus joka on periaatteessa alustariippumaton. Sovelluskehittäjät voivat käyttää samaa Docker-konfiguraatiota rakentaakseen identtisiä säiliöitä sovelluskehityksen ajaksi, tarviten vain Docker-ohjelmiston. Tämän lisäksi Docker mahdollistaa saman Docker-konfiguraation käyttämisen sovelluksen pystyttämiseen ja julkaisemiseen nopeasti, helposti sekä jopa kustannustehokkaasti eri paikkoihin. Docker-compose on tapa rakentaa Docker-verkko, joka koostuu palveluista jotka ovat joko valmiiksi tehtyjä Docker-kuvia tai itse Dockerfile:n avulla tehtyjä Docker-kuvia. Docker-verkkoon voidaan myös lisätä yhteisiä tietosäiliöjä, joita verkkoon kuuluvat palvelut voivat yhteisesti hyödyntää. Yksittäisen säiliön konfiguraation sisältämä Dockerfile ja kokonaisen Docker-verkon konfiguraation sisältämä docker-compose -tiedosto kirjoitetaan yaml-kielellä.

Tässä diplomityössä Docker:ia käytettiin hyväksymistestauksen testitapauksien automatisoimiseen tarvittavien työkalujen säiliöinnissä. Olemassa olevaan Docker-verkkoon lisättiin hyväksymistestauksen testitapauksia varten tarkoitettu säiliö, joka hyödyntää Robot Framework:ia, Selenium:ia, Xvfb:ää ja sisältää muun muassa testauksessa tarvittavat verkkoselaimet. Docker:ia käyttämällä siis pystyttiin luomaan monistettava ja uniikki hyväksymistestauksen automatisointiympäristö, jota voidaan käyttää jatkuvan integraation yhteydessä testitapauksien suorittamiseen.

4.4.5 GoCD

GoCD on avoimen lähdekoodin Apache 2.0 lisensoitu jatkuvan integroinnin ja jatkuvan julkaisemisen mahdollistava palvelinohjelmisto. Ohjelmisto mahdollistaa koko koonti-testausjulkaisu putkiryhmän tai vain sen osien automatisoimisen. GoCD-palvelinta mainostetaan soveltuvan hyvin erityisesti jatkuvan julkaisemisen rakentamiseen. GoCD on saman ThoughtWorks yhtiön kehittämä ohjelmisto, kuten aiemmin esitetty Selenium työkalukin 4.4.2.

Koonti-testaus-julkaisu putkiryhmän voi rakentaa GoCD-palvelimen graafisen käyttöliittymän kautta tai koodina käyttäen yaml tai json-syntaksia. Teknisesti GoCD-ohjelmisto koostuu itse palvelimesta ja agenteista, jotka voivat suorittaa palvelimen pyytämänä ennalta määritettyjä koonti-testaus-julkaisu putkiryhmän tehtäviä. Agentit ovat tarkoituksenmukaista sijoittaa eri järjestelmään kuin missä itse palvelin sijaitsee ja agenteille voi määrittää resurssiominaisuuksia, jotka kertovat palvelimelle mitä tehtäviä agenteilla voi teettää. GoCD-ohjelmiston terminologia on hieman tavallisesta poikkeavaa ja erilainen esimerkiksi todella suosituksen Jenkins-ohjelmiston vastaavista. GoCD-terminologiassa ylin käsite on putkiryhmä, jonka avulla yhteen kuuluvat putket voidaan järjestää samaan kokonaisuuteen. GoCD-terminologiassa yksittäinen putki vastaa esimerkiksi koontivaihetta tai testausvaihetta. Yksittäisen putken alaisuudessa on vaihteita, jotka antavat GoCD-palvelimen käyttöliittymässä tiedon vaiheen onnistumisesta. Vaiheet itsessään sisältävät vielä tehtäviä, jotka ovat yksittäisiä komentoja tai sellaisia suoritettavia tehtäviä, jotka agentit pystyvät käsittelemään. GoCD-ohjelmiston terminologiaan kuuluvat vielä vahvasti artefaktit, jotka ovat sellaisia tiedostoja mitä tehtävien suorittamisen yhteydessä syntyy ja jotka on merkitty säästettäväksi. Esimerkkejä artefakteista ovat ohjelman koontiversiot tai testiraportit.

Jatkuvan integraation yhteydessä tapahtuvan testiautomaation puolesta ei välttämättä ole suurta merkitystä mikä jatkuvan integraation mahdollistava palvelinohjelmisto on käytössä. Tämä havainto tuli esiin, kun tätä diplomityötä varten testiautomaatioon tarvittavat ohjelmistot säiliöitiin aiemmin esitetyllä Docker-työkalulla, jota voidaan yhden testausvaiheen tehtävän aikana kutsua komentorivipohjaisesti.

4.5 Testitapauksien rakentaminen

Testitapaus on testiautomaation näkökulmasta määritelty toimenpiteiden, ehtojen ja muuttujien joukko, joka suorittamalla voidaan verifioida jokin osa, ominaisuus tai toiminnallisuus ohjelmistosta. Testitapauksien rakentaminen on järkevää järjestää testikokoelmiksi, jotka tarkoittavan samaan kontekstiin kuuluvista testitapauksista muodostettua joukkoa. Tässä diplomityössä keskityttyyn hyväksymistestaukseen liittyen testitapaukset kirjoitetaan usein käyttötapauksien muodossa. Hyväksymistestauksen tapauksessa testitapauksien määrittäminen testiautomaatiota varten voidaan toteuttaa Robot Framework:illä ja apuna käyttää muita aiemmin mainittuja työkaluja 4.4. Lisäksi hyväksymistestauksen

priorisointiin painotetun verkon avulla on välttämätöntä suunnitella ja rakentaa testitapaukset näkymä ja siirtymäperusteisesti, koska menetelmä hyödyntää matemaattisia näkymä ja siirtymäperusteisesti laadittuja painotettuja verkkoja.

Testitapauksen perusformaatti koostuu lähtötilanteesta, laukaisijasta ja verifikaatiosta. Lähtötilanteessa oletetaan jotakin ja seuraavassa vaiheessa seurataan, kun jokin ehto tapahtuu, jonka jälkeen voidaan tarkistaa seuraus ja verifioida onko se oletuksen mukainen. Testitapauksien yleisiä tavoitteita ovat: yksinkertaisuus, läpinäkyvyys, käyttäjätietoisuus, epätoistuvuus, olettamattomuus, kattavuus, tunnistettavuus, jälkensä puhdistava, toistettava, syvyyttömyys ja atomisuus.

Robot Framework:in perustaja on kirjoittanut laajan ohjeistuksen siitä, miten Robot Framework:iä käyttäen luodaan hyviä testitapauksia [4]. Klärckin ohjeistuksen pohjalta on huomioitavaa erityisesti testikokoelmien, testitapauksien ja avainsanojen nimeäminen jonka kuuluisi olla selkeää, kuvaavaa ja ytimekästä. Dokumentaation määrää testitapauksissa tulisi rajoittaa, sillä hyvin kirjoitetut testitapaukset ovat Robot Framework:iä käyttäen selkeitä jo sellaisenaan. Dokumentaatiota kuuluisi lisätä lähinnä vain testikokoelmiin yleisellä tasolla. Testikokoelmat kuuluisi sisältää vain toisiinsa liittyviä testejä ja testitapauksien sekä avainsanojen tulisi olla sellaisinaan selkeästi ymmärrettäviä. Muuttujien käytöllä suositellaan kapseloimaan pitkiä ja kompleksisia arvoja, mutta arvojen syöttäminen ja palauttaminen muuttujia hyödyntäen tulisi pitää pois testitapauksien tasolta.

4.6 Priorisointiongelma

Testitapauksien priorisointi on kustannussyistä tai resurssien optimoinnin kannalta erittäin tärkeää. Ohjelmistotestauksessa on myös hyvä tiedostaa, että ohjelmistotuotetta ei usein voida testata täydellisesti, joka nostaa esiin tarpeen tärkeimpien testitapauksien priorisoinnista. Priorisoinnin toteuttamisen tärkeys korostuu erityisesti silloin kun kohdejärjestelmä on kompleksinen ja toiminnallisia ominaisuuksia on paljon. Priorisointi vaatii kuitenkin priorisointimenetelmästä riippumatta ylimääräistä työtä ohjelmistokehittäjiltä ja testaajilta.

Priorisointiongelmaa voidaan ajatella sen laiminlyömisestä seuraavien haittojen näkökulmasta. Ilman testitapauksien priorisointia voi esiintyä muun muassa seuraavia haittoja. Prioriteettien puuttumisen seurauksena tärkeät ongelmat voidaan havaita vasta liian myöhään. Testitapauksia ei voida järjestää prioriteettien mukaan suoritettaviksi. Prioriteettijärjestyksen puuttumisesta johtuen epäoleellisten testitapauksien mukaan katkeava testaus voi piilottaa oleellisia testitapauksia. Tämän lisäksi myös prioriteettien puolesta epäoleellisetkin testitapaukset toteutetaan. Epäoleellisten testitapauksien toteuttaminen puolestaan kuluttaa resursseja ja lisää kustannuksia. Ajan myötä ohjelmistot ja niiden testaukseen toteutetut testitapaukset muuttuvat ja vanhenevat. Prioriteettien puuttuminen poistaa mahdollisuuden varautua oleellisten testitapauksien huolellisempaan ja aikaa kestävään suunnitteluun. Lisäksi testikattavuutta ei voida optimoida vähentämällä täysin epäoleellisia testitapauksia, jos niitä varten ei ole tehty priorisointia ennen toteutusta.

Priorisointiongelman ratkaisemiseen on olemassa useita erilaisia lähestymistapoja ja menetelmiä, kuten esimerkiksi heuristinen priorisointi tai MoSCoW-menetelmä. Tässä diplomityössä esitetään ja käytetään priorisointiin kuitenkin vain matemaattista painotettuihin verkkoihin perustuvaa lähestymistapaa, joka on uudenlainen tämän diplomityön tuotteenä kehittynyt matemaattinen menetelmä priorisointiongelman ratkaisemiseen.

5 PRIORISOINTI PAINOTETUN VERKON AVULLA

Tässä luvussa käsitellään ensin työhön keskeisesti kuuluvan verkkoteorian perusteita ja käydään huolellisesti läpi niistä tässä työssä käytettävät osat. Työssä sovelletaan erityisesti verkkoteorian painotettua verkkoa sekä verkkoteoriassa esiintyvän lyhimmän polun ongelmaan kehitettyä Dijkstran algoritmia. Verkkoteoria itsessään on osa diskeettiä matematiikkaa.

Verkkoteorian jälkeen tässä luvussa esitetään vaiheittain työn tuloksena kehitetty priorisointimenetelmä. Priorisointia varten esitetään harkintaa käyttäen valitut priorisointiin vaikuttavat muuttujat, niitä käyttävät painofunktiot, verkon rakentaminen ja karsiminen sekä verkon ja testitapauksien yhteys. Lisäksi käydään läpi miten menetelmää käyttäen tuotetun painotetun verkon sisältämää informaatiota voidaan hyödyntää prioriteeteiltaan tärkeimmän polun löytämiseen 5.7.

5.1 Matemaattisten verkkojen tarkoitus

Matemaattisten verkkojen tarkoituksena on mallintaa parittaisia riippuvuuksia verkko-
maisessa objektijoukossa. Verkkoteoriassa peruskäsitteitä ovat itse *verkko* eli *graafi*, joka muodostuu *solmuista* ja niiden välisiä riippuvuuksia esittävistä *kaarista* tai *nuolista*. Verkkoteorialla on lukuisia käytännön sovellutuksia. Verkkoteoriaa sovelletaan muun muassa tietokonetieteissä, kielitieteissä, fysiikan ja kemian sovellutuksissa, sosiaalisissa tieteissä ja biologiassa. Alun perin verkkoteoria katsotaan syntyneen 1700-luvulla esiintyneestä niin sanotusta Königsbergin siltaongelmasta, johon Leonhard Euler esitti todistuksensa.

Matemaattisten verkkojen käyttöön päädyttiin tässä työssä siksi, että niiden avulla on hyväksymistestauksen kohteena oleva käyttöliittymä mahdollistaa mallintaa verkoksi. Käyttöliittymän verkkomuotoiseen esitykseen voidaan vielä lisätä painot, jotka tässä tapauksessa kuvaavat prioriteetteja, mahdollistaen testikokoelmien priorisoinnin.

5.2 Perusmerkinnät ja käsitteet

Verkkoteoriassa käytetään muun muassa seuraavia perusmerkintöjä ja käsitteitä:

- $V := \{v_1, v_2, v_3\}$ Solmujoukko joka sisältää *solmut* v_1 , v_2 ja v_3 .
- $E := \{e_1, e_2, e_3\}$ Kaarijoukko joka sisältää *kaaret* e_1 , e_2 ja e_3 .
- $\phi(e_1) := \langle v_1, v_2 \rangle$ Kaariparin v_1 ja v_2 yhdistävän *kaaren* e_1 kuvaaja.

- $d_G(x)$ Solmun asteluku, eli solmuun liittyvien *kaarten* määrä.
- $G_2 \subset G_1$ Aliverkko, eli *verkko* G_2 joka koostuu osasta *verkon* G_1 *solmuja* ja *kaaria*.
- $v_1 \neq v_2, v_1 \rightarrow v_2$ Verkon yhtenäisyys, eli jokaiselle solmuparille $v_1 \neq v_2$ on olemassa niitä yhdistävä *kaari*.
- $P = \{v_0, v_1, \dots, v_n\}, v_0 \rightarrow v_n$ Polku, eli *suunnattu solmujono* jota pitkin voidaan kulkea *solmusta* v_0 *solmuun* v_n .
- $P = \{v_0, v_1, \dots, v_n | e \in E_P, e \notin \{E_P \setminus \{e\}\}\}$ Sykli, eli *polku*, jonka aloitus v_0 ja lopetussolmu v_n on sama, mutta polun jokaista kaarta e kuljetaan vain kerran.
- $d_G(v_1) = 0$ Eristetty solmu, eli *solmu* jonka *asteluku* on nolla.
- $v_1 \rightarrow v_2, d_G(v_1) = 1 \vee d_G(v_2) = 1$ Silta, eli *kaari* johon yhdistyvän *solmun asteluku* on yksi ja jonka poistaminen epäyhteinäistää *verkon*.
- $v_x \rightarrow v_x$ Silmukka, eli *kaari* jonka *aloitussolmut* ja *lopetussolmu* ovat sama *solmu*.
- $\alpha := V(G), E(G) \rightarrow \mathbb{N}$ Painofunktion yleinen kuvaus verkossa G , solmuille V ja kaarille E .

5.3 Priorisointiin vaikuttavat muuttujat

Näkymä- ja siirtymäperustaiseen priorisointiin vaikuttavat monet eri asiat, joista osa kasvattaa prioriteettia ja osa laskee sitä. Prioriteettia kasvattava muuttuja on esimerkiksi liiketoiminnallinen arvo ja laskeva muuttuja on esimerkiksi projektin muutosherkkyys. Muuttujat ovat kuitenkin hyvin kontekstiriippuvaisia, joten yleispätevää ja kaikkiin tilanteisiin soveltuvaa listaa muuttujista on hankala antaa. Kontekstiriippuvaisuuden takia muuttujiin ja myöhemmin esitettäviin painofunktioihin on varattu paikka omille lisämuuttujille.

Tässä diplomityössä esiteltävää priorisointimenetelmää varten jokainen priorisointiin vaikuttava muuttuja arvioidaan asteikolla 1-10, paria poikkeusta lukuun ottamatta. Numeerisella asteikolla on tarkoitus antaa korkea numero, jos muuttuja on prioriteetiltaan tärkeä kyseisen näkymän, eli verkon solmun kohdalla. Jos jokin muuttuja ei ole kelpoinen siinä kontekstissa, jossa menetelmää yritetään hyödyntää, tulee muuttujan arvo asettaa nolaksi, jolloin se sivuutetaan painofunktiossa 5.4.

Poikkeukselliset muuttujat ovat käytötapauksien määrä ja siirtymien määrä, joissa numeerisen asteikon sijaan käytetään kyseisten muuttujien määrää suhteessa koko verkkoon. Esimerkiksi siirtymien määrää ilmaiseva suhde määritetään laskemalla solmun asteluku $d_G(v)$, eli solmuun liittyneiden kaarien määrä, jaettuna kaikilla verkossa olevien kaarien määrällä. Lisäksi siirtymien määrän suhde vielä kerrotaan luvulla 10, jotta se saadaan skaalautumaan muiden muuttujien kanssa samalle tasolle.

Taulukko 5.1. Priorisointiin vaikuttavat muuttujat

<i>m</i>	Muuttuja	Etumerkki	Asteikko
1	Liiketoiminnallinen arvo	+	1 - 10
2	Liiketoiminnallinen visio	+	1 - 10
3	Negatiivinen käyttäjäpalaute	+	1 - 5
4	Käyttötapauksien määrä	+	10 · suhde
5	Siirtymien määrä	+	10 · suhde
6	Positiivinen käyttäjäpalaute	–	1 - 5
7	Muutosherkkyys	–	1 - 10
8	Toteuttamisen kompleksisuus	–	1 - 5
9	Toteutuksen virheherkkyys	–	1 - 5
10	Omat lisämuuttujat	±	1 - 10

5.4 Painofunktiot priorisointiin

Painofunktioiden määrittäminen on tärkeä osa painotetun verkon avulla priorisointia, sillä niiden avulla määritetään verkon solmujen ja kaarien prioriteetit. Tavanomaisesti numeerinen prioriteetti usein mielletään olevan korkea, jos priorisoitu muuttuja on tärkeä. Painotettujen verkkojen tapauksessa on kuitenkin järkevää vaihtaa numeerisen prioriteetin suuntaa, jotta painotettuun verkoon sovellettavat lyhimmän polun algoritmit toimisivat etsien prioriteetiltaan tärkeitä polkuja. Ennen prioriteetin suunnanvaihtoa, voidaan kokonaisprioriteetti p yksittäiselle solmulle v , eli näkymälle määrittää seuraavasti.

$$p(v) = \sum_{i=1}^5 m_i - \sum_{j=6}^9 m_j \pm m_{10}$$

Prioriteetin suunnan vaihtamiseksi suuresta pieneen, säilyttäen kuitenkin prioriteetin sisältämän informaation, voi hoitaa käänteislukujen avulla. Ennen käänteisluvuksi muuttamista, prioriteettiin vaikuttavien muuttujien yhteenlaskettu summa voi olla ongelmallisesti negatiivinen tai nolla. Negatiiviset arvot eivät ole painotetun verkon kannalta erityisen järkeviä, sillä tässä diplomityössä hyödynnettävää Dijkstran algoritmia ei voida käyttää negatiivisien painojen kanssa. Dijkstran algoritmin toiminta nollan tapauksessa voi myös kuulostaa epäilyttävältä, kuten esimerkiksi tilanne, jossa painotetun verkon kaikki painot olisivat nollia. Dijkstran algoritmin tapauksessa tällainen verkko on kuitenkin sallittu, koska silloin lyhimmän polun ratkaisu on verkon kaikki solmut. Lyhimmän polun ongelman erityisvaatimusten lisäksi käänteislukua varten nolla on huono arvo siinä mielessä, että sille ei ole olemassa lainkaan käänteislukua. Tämä johtuu siitä, että jos nolalle yrittäisi etsiä käänteislukua, tulisi eteen nolalla jakaminen jota ei voi tehdä. Nämä molemmat ongelmatapaukset voidaan kuitenkin painofunktiossa ratkaista siten, että käänteisfunktiota

ei etsitä, vaan korvataan painofunktion tulos yhdellä.

Painofunktio yksittäiselle solmulle v , eli näkymälle saadaan solmun kokonaisprioriteetin $p(v)$ käänteislukuna.

$$\alpha(v) = \begin{cases} p^{-1}(v) & p(v) > 0 \\ 1 & p(v) \leq 0 \end{cases}$$

Painofunktio yksittäiselle solmut v_x ja v_y yhdistävälle kaarelle e_{xy} , eli siirtymälle saadan myös käänteislukuna. Kaaren painofunktiota varten pitää kuitenkin huomioida, että sen kokonaisprioriteetti on kaaren solmujen kokonaisprioriteetin summa $p(v_x) + p(v_y)$. Kaaren kokonaisprioriteetti $p(v_1) + p(v_2)$ pitää laskea ennen käänteisluvuksi muuttamista.

$$\beta(e_{xy}) = \begin{cases} (p(v_x) + p(v_y))^{-1} & p(v_x) + p(v_y) > 0 \\ 1 & p(v_x) + p(v_y) \leq 0 \end{cases}$$

5.5 Verkon rakentaminen

Tässä diplomityössä on aiemmin moneen otteeseen kerrottu näkymä ja siirtymäperusteista testiautomaation toteuttamisesta ja priorisoinnista. Painotetun verkon rakentamista varten tulee tarvittavat näkymät ja niiden väliset siirtymät muodostavat testauskohteen käyttöliittymästä. Web-sovelluksen käyttöliittymän näkymiä ovat muun muassa sivut, sivujen sisältämät säiliö-elementit ja dialogit. Siirtymät ovat usein sivujen välisiä linkkejä tai vaihtoehtoisesti jotakin sellaista toiminnallisuutta, joka muuttaa nykyisen näkymän tai osan siitä toiseksi näkymäksi.

Seuraavassa taulukossa on esitetty kuvitteellisen web-sovelluksen mukainen näkymien ja siirtymien mukaan laadittu esimerkki 5.2. Taulukossa esitetään näkymät kirjautumisnäkymästä ohjenäkymään ja jokaisen näkymän siirtymät eli yhteydet toisiin näkymiin. Näkymät ja siirtymät luovat matemaattisen verkon laatimisen perusedellytykset, eli datan jonka avulla myöhemmin esitettävä painomatriisi voidaan laatia. Taulukossa on lisäksi esitetty jokainen näkymään liittyvä priorisointiin vaikuttava muuttuja. Priorisointiin vaikuttavien muuttujien arvot on laadittu subjektiivisesti kuvitteellisen esimerkin muodossa. Priorisointiin vaikuttavien muuttujien yhteenlaskettu prioriteetti yksittäiselle näkymälle on laskettu taulukkoon valmiiksi käyttäen aiemmin esitettyä prioriteettifunktiota $p(n)$, jossa n tarkoittaa sitä näkymää jolle prioriteetti lasketaan.

Taulukko 5.2. Esimerkkiverkon näkymät, siirtymät ja priorisointimuuttujat

n	Näkymä	Siirtymät	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	$p(n)$
A	Kirjautuminen	B	10	10	0	2	1	0	5	5	5	8
B	Pelivalikko	A, C, D, G	8	10	1	2	4	4	5	5	5	6
C	Asetukset	A, B	4	6	5	2	2	2	5	5	5	2
D	Peli	B, E, G	10	10	4	2	3	4	4	5	5	11
E	Tulokset	B, D, F	6	8	0	2	3	5	5	4	5	2
F	Onnittelu	B, E	1	8	0	0	2	2	5	2	5	-3
G	Ohje	B, D	1	10	2	0	2	0	8	0	0	7

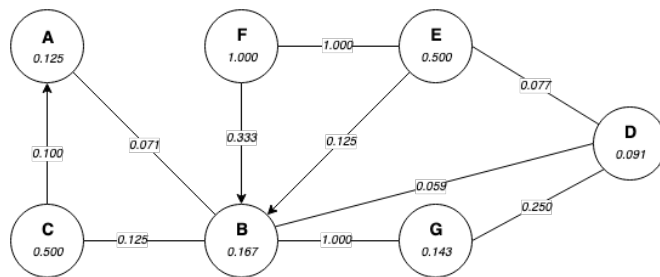
Painotetun verkon rakentamisen syötteeksi täytyy käyttöliittymän näkymät ja siirtymät sekä niiden painoarvot esittää painomatriisin muodossa. Painoarvot saadaan aiemmin esitetyn painofunktion β avulla 5.4. Painoarvo lasketaan β funktion avulla jokaiselle kahta näkymää yhdistävälle siirtymälle, eli painotetun verkon solmujen väliselle kaarelle. Painofunktio β käyttää kaaren molempien päätepisteiden v_A ja v_B yhteenlaskettua prioriteettia, josta käänteisluku otetaan. Näin saadaan laskettua kaarelle sellainen painoarvo, joka tarkoittaa painotetussa verkossa siirtymän näkymiin sidottua prioriteettia.

$$\beta(e_{AB}) = (p(v_A) + p(v_B))^{-1} = (8 + 6)^{-1} = \frac{1}{14} \approx 0.071$$

Painomatriisi, esimerkin mukaiselle datalle 5.2 ja siitä lasketuille painoarvoille on esitetty seuraavassa matriisissa M_G . Painomatriisissa tulee väistämättä esiin tilanne, jossa pitää määrittää painoarvo kaarelle jonka aloitussolmu ja lopetussolmu ovat sama solmu itsessään. Tällaisissa tapauksissa, tilanteesta riippuen painomatriiseihin usein merkitään 0, ∞ tai $-$. Tässä diplomityössä esitettävän menetelmän painomatriiseissa solmuun itseensä johtuvan kaaren, eli silmukan painoksi merkitään aina $-$, koska käyttöliittymän näkymästä siirtymät itseensä ei menetelmässä käsitellä aitoina siirtyminä. Tämän lisäksi luonnollisesti jokainen sellainen solmupari, jolla ei ole niitä yhdistävää kaarta merkitään painomatriisiin käyttäen $-$ merkintää. Sellaisien siirtymien toiminnallisuuden testaaminen on tarkoitus kattaa näkymän mukaisen testikokelman testitapauksissa ja ne tulee priorisoiduiksi näkymä ja siirtymäperusteisesti. Painotetun verkon kaaret voivat verkkoteorian mukaan olla suunnattuja tai suuntaamattomia. Tässä esimerkkitapauksessa jokainen siirtymä näkymien välillä on suuntaamaton, eli toisin sanoen käyttöliittymässä kaksisuuntainen ja se priorisoidaan sen mukaisesti. Painomatriisissa suuntaamattomien kaarien johdosta voidaan huomata, että painomatriisin diagonaalin erottamat puoliskot ovat toistensa peilikuvia.

$$M_G \approx \begin{matrix} & \begin{matrix} v_A & v_B & v_C & v_D & v_E & v_F & v_G \end{matrix} \\ \begin{matrix} v_A \\ v_B \\ v_C \\ v_D \\ v_E \\ v_F \\ v_G \end{matrix} & \begin{pmatrix} - & 0.071 & 0.100 & - & - & - & - \\ 0.071 & - & 0.125 & 0.059 & 0.125 & 0.333 & 1.000 \\ 0.100 & 0.125 & - & - & - & - & - \\ - & 0.059 & - & - & 0.077 & - & 0.250 \\ - & 0.125 & - & 0.077 & - & 1.000 & - \\ - & 0.333 & - & - & 1.000 & - & - \\ - & 1.000 & - & 0.250 & - & - & - \end{pmatrix} \end{matrix}$$

Painomatriisin avulla voidaan siis rakentaa matemaattinen painotettu verkko, joka kuvaa näkymiä ja siirtymiä sekä niiden prioriteetteja. Painotetun verkon kuvaamiseen piirretään jokainen erilaista käyttöliittymän näkymää vastaava ja esimerkkitiedon 5.2 mukainen solmu ja niiden välisiä siirtymiä kuvaavat yhteydet eli kaaret. Kaarien yhteyteen lisätään kuvaajassa kaaren prioriteettia kuvaava painoarvo. Seuraavassa on esitetty painomatriisin dataa vastaava painotetun verkon kuvaaja 5.1 sellaisena, kuin se on ennen siihen tehtäviä prioriteettileikkauksia. Priorisoimista varten tehtävien leikkauksien tekeminen esitetään myöhemmin omassa kappaleessaan 5.6.



Kuva 5.1. Esimerkki painotetusta verkosta ennen leikkauksia

Perinteisesti painotetuissa verkoissa ei esitetä yksittäisiä solmupainoja vaan painotetun verkon painoilla tarkoitetaan solmujen välisien kaarien painoarvoja. Tässä diplomityössä kehitettyä menetelmää käytettäessä edellä esitettyyn painotettuun verkkoon 5.1 on kuitenkin lisätty painomatriisin sisältämän informaation lisäksi painofunktion $\alpha(v)$ avulla lasketut yksittäisten solmujen eli näkymien painoarvot.

$$\alpha(v_A) = p^{-1}(v_A) = 8^{-1} = \frac{1}{8} = 0.125$$

Yksittäisten solmujen prioriteettia kuvaavat painoarvot ovat erittäin merkittäviä ja hyödyllisiä, sillä niiden avulla voidaan järjestää itse solmut, eli näkymät prioriteettien mukaiseen järjestykseen. Tämän lisäksi solmujen prioriteettien avulla voidaan verkkoon muun muassa soveltaa lyhimmän polun ratkaisemiseen kehitettyjä algoritmeja, kuten myöhemmin Dijkstran algoritmin osalta esitetään omassa kappaleessaan 5.7.

5.6 Verkon karsiminen

Painotetun verkon karsiminen eli leikkaaminen on prioriteeilla painotetun verkon tärkeä ominaisuus. Verkkoteorian soveltaminen prioriteettien avulla painotettuun verkkoon on erityisen hyödyllistä, kun verkon kaarissa alhainen paino tarkoittaa suurta prioriteettia. Verkon karsimista varten valitaan kattavuus, joka vastaa minimirajaa ja jonka jälkeen karsiminen lopetetaan. Kattavuus tarkoittaa myös testikattavuutta testikokoelmien näkökulmasta, sillä painotetussa verkossa jokainen solmu, eli näkymä vastaa näkymän mukaan kategorisoitua testikokoelmaa.

Verkkoon tehtäviä leikkauksia varten tarvitsee määrittää haluttu kattavuus $0 \leq c \leq 100$, joka on prosentuaalinen luku siitä kuinka suuri osa verkon solmuista eli näkymistä tai testikokoelmista täytyy verkkoon jäädä karsimisen jälkeenkin. Leikkauksien tekeminen ja toistaminen suoritetaan käyttäen seuraavia toimenpiteitä n -kertaa, niin kauan kunnes karsittu aliverkko on suurempi kuin kattavuuden mukaan laskettu osuus alkuperäisestä verkosta tai jos iteraatiokerralla ei enää löydy toimenpiteillä poistettavia solmuja.

$$|V(G_s)| > c \cdot \frac{|V(G)|}{100}, G_s \subset G$$

Tässä verkon karsimisen esimerkissä kattavuutena käytetään $c = 80$, joka tarkoittaa esimerkin solmujen määrän 7 karsimista $80 \cdot \frac{7}{100} = 5.6$, eli lukumäärään 5 asti.

1. Poistetaan verkosta löytyvä eristetty solmu, eli solmu jonka asteluku on nolla.

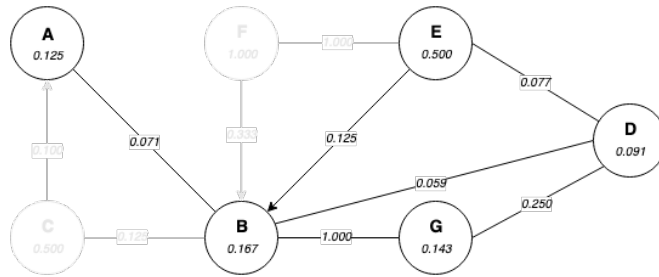
$$d_G(v) = 0$$

2. Poistetaan verkosta löytyvä sillattu solmu, eli solmu jonka asteluku on yksi ja paino on pienempi kuin solmujen painojen keskiarvo.

$$d_G(v) = 1 \wedge \alpha(v) > \frac{1}{|V(G)|} \cdot \sum_{v \in V(G)} \alpha(v)$$

3. Poistetaan verkosta sellainen alhaisimman prioriteetin solmu, jonka asteluku on pienempi kuin solmujen astelukujen keskiarvo ja paino on pienempi kuin solmujen painojen keskiarvo.

$$d_G(v) < \max\{d_G(x) | x \in V(G)\} \wedge \alpha(v) > \frac{1}{|V(G)|} \cdot \sum_{v \in V(G)} \alpha(v)$$



Kuva 5.2. Esimerkki painotetusta verkosta leikkauksien jälkeen

5.7 Dijkstran algoritmin hyödyntäminen

Priorisointimenetelmän mukaan karsittuun painotettuun verkkoon on mahdollista soveltaa lyhimmän polun ongelman ratkaisemiseen kehitettyjä algoritmeja, jolloin ne toimivat etsien alhaisimman, eli korkeimman prioriteetin polkuja. Lyhimmän polun etsimiseen on tarkoitus valita aina sellaiset aloitus ja lopetus pisteet, joiden välille lyhin polku verkossa halutaan etsiä. Lyhimmän polun löytymisen yhtenä perusedellytyksenä on verkon yhtenäisyys, joka tarkoittaa sitä, että verkon kaikkia solmuja tulee yhdistää vähintään yksi kaari ja että verkon jokaisesta solmusta on löydettävissä yhteys mihin tahansa verkon solmuun. Lyhimmän polun ongelma johon muun muassa Dijkstran algoritmi antaa ratkaisun on matemaattisessa muodossaan seuraavanlainen.

$$d_G^\alpha(v_1, v_2) = \min\{\alpha(P) \mid P : v_1 \rightarrow v_2 \mid v_1, v_2 \in V(G)\}$$

Prioriteeteiltaan tärkeimmän polun löytämiseksi, valitaan ensin $\min\{\alpha(v_1) \mid v_1 \in V(G)\}$ ja $\min\{\alpha(v_2) \mid v_2 \in V(G), v_1 \neq v_2\}$ sekä etsitään sitten Dijkstran algoritmin avulla niiden välinen lyhin polku. Dijkstran algoritmin sisäinen toimintaperiaate ei ole tämän diplomityön näkökulmasta oleellista, mutta se on kuitenkin esitetty tarkemmin pseudokoodina liitteessä B. Dijkstran algoritmi kahdelle painoltaan pienimmälle, eli prioriteetiltaan korkeimmalle antaa tuloksena v_1 ja v_2 solmuja yhdistävän polun, jonka sisältävät solmut eli näkyvät ovat prioriteetiltaan tärkeimmät. Koska painotetun verkon painofunktiot on laadittu käänteislukuja hyödyntäen, Dijkstran algoritmi löytää painoarvoltaan matalimman, mutta prioriteetiltaan tärkeimmän polun solmujen v_1 ja v_2 välille. Näin ollen saadaan helposti ja vaivattomasti tietää sellaiset solmut eli käyttöliittymän näkymät jotka kuuluvat tärkeimpiin ja joista testiautomaation rakentaminen kannattaa aloittaa.

5.8 Verkon ja testitapauksien yhteys

Ennen testitapauksien suunnittelua tehtävä painotetun verkon avulla tehty priorisointi havainnollistaa käyttöliittymän näkymiä ja niiden välisiä siirtymiä. Tällaisesta painotetusta verkosta saadaan priorisoitua näkymät ja siirtymät, mutta lopulliset testitapauksien prioriteetit ovat kuitenkin testitapaukseen kuuluvien näkymien tai siirtymien prioriteetteja. Tä-

mä tarkoittaa käytännössä sitä, että kun näkymät ja siirtymät on priorisoitu, on esimerkiksi yhden yksittäisen tarkasteltavana olevan näkymän toiminnoilla sama keskenään prioriteetti. Painotetun verkon näkymät ovatkin suoraan yhteydessä testiautomaatiota varten rakennettaviin testikokoelmiin, jotka sisältävät kokoelman testitapauksia kyseiselle näkymälle. Toisin sanoen, painotetun verkon näkymiä vastaavat testikokoelmat ovat varsinaisen priorisoinnin kohteena.

Testitapaukset ja testikokoelmat kappaleessa 3.3 on esitetty niiden välistä eroa ja siitä kuinka testikokoelmat koostuvat yhteen liittyvistä testitapauksista. Painotetun verkon avulla tehtävää priorisointia käyttäessä on tarkoitus ajatella testiautomaation testitapauksien kategorisoinnista testikokoelmiksi käyttöliittymän näkymiä vastaavalla tavalla. Kun käyttöliittymän näkymillä on niitä vastaavat testikokoelmat, toimii tässä diplomityössä kehitetty painotetun verkon avulla toteutettava priorisointi oikein ja siten kuin se on tarkoitettu. Jos testiautomaation halutaan lisätä testitapauksia tai testikokoelmia, jotka eivät ole luettavissa painotetusta verkosta, niille ei luonnollisesti ole olemassa prioriteettia ja sellaiset täytyy käsitellä ylimääräisinä, täydentävinä testitapauksina.

Painotetun verkon kuvaamisen seurauksena, voidaan verkosta nähdä myös paljon hyödyllistä informaatiota, kuten muun muassa siinä esiintyviä sillattuja solmuja sekä syklejä. Sillatut solmut ovat sellaisia käyttöliittymän näkymiä, joihin käyttäjä ei kovinkaan usein päädy ja näin ollen jos niitä lopullisessa karsitussa verkossa esiintyy, ne ovat testiautomaatin rakentamisen kannalta usein vain vähän merkitseviä. Eristetyt solmut ovat samaan tapaan vain vähän merkitseviä kuin sillatut solmut. Syklit puolestaan ovat erittäin merkittävä osa painotetussa verkossa ja testiautomaation rakentamisessa, sillä ne ovat sellaisia käyttöliittymän näkymiä ja niiden välisiä siirtymiä, jotka toistuvat käyttäjälle usein käyttöliittymää käyttäessään. Solmujen asteluvut kertovat myös paljon solmujen merkityksestä. Sellainen solmu jonka asteluku, eli siihen liittyvien kaarien lukumäärä on korkea, on testiautomaation rakentamisen kannalta yhtälailla erittäin merkittävä osa testiautomaatiota.

6 TULOSTEN TARKASTELU JA ARVIOINTI

Tässä kappaleessa esitetään yhteenveto tutkimuksen tuloksista ja evaluoidaan testausjärjestelmän ja priorisointimenetelmän toteutuksien onnistumista. Ensin evaluoidaan diplomityössä suunnitellun ja asiakasyritykselle toteutetun testausjärjestelmän positiivisia sekä negatiivisia puolia. Seuraavaksi evaluoidaan diplomityössä kehitetyn priorisointimenetelmän positiivisia ja negatiivisia puolia ja muun muassa pohditaan kuinka hyvin soveltuva ja toistettavissa oleva kyseinen kehitetty priorisointimenetelmä on. Lisäksi lopussa vielä esitetään toteutuksen jälkeen esiin tulleita jatkokehitysehdotuksia testausjärjestelmälle sekä priorisointimenetelmälle.

6.1 Tutkimuksen konkreettiset tulokset

Työn tuloksena kehitetty web-käyttöliittymien hyväksymistestauksen automatisoimisen mahdollistava testausjärjestelmä on integroitu onnistuneesti osaksi GoCD-palvelimen avulla suoritettavaa jatkuvaa integrointia. Lyhyesti sanottuna testausjärjestelmän osalta konkreettinen tulos koostuu järjestelmästä, joka mahdollistaa testitapauksien luomisen Robot Framework alustalle käyttäen Selenium kirjastoa, Xvfb virtualisointipalvelinta ja Docker säiliöintiohjelmistoa. Testausjärjestelmän toimivuus käytännössä todettiin esimerkkitestitapauksien muodossa oikeassa ympäristössään ja testausjärjestelmän mahdollistamat ominaisuudet ovat jo itsessään oikeassa ja lopullisessa käyttöympäristössä tarvittavia.

Web-sovelluksien näkymä ja siirtymäperustainen painotettua verkkoa hyödyntävä priorisointimenetelmä on myös todettu toimivaksi lähestymistavaksi priorisointiin. Lyhyesti sanottuna priorisointimenetelmän tuloksena on painotettuja verkkoja hyödyntävä menetelmä, jossa määritetään priorisointiin vaikuttavat muuttujat, painofunktiot, painomatriisi, prioriteetteihin perustuvien leikkauksien tekeminen ja prioriteettien löytäminen sekä lukeminen verkosta. Priorisointimenetelmän toimivuus käytännössä todettiin aidosta ympäristöstä yksinkertaistaen poimitusta web-sovelluksesta. Priorisointimenetelmän avulla todettiin, että priorisoinnin aikana toteutetut painotetun verkon leikkaukset olivat juuri niitä näkymiä, jotka vaistonvaraisesti ilman menetelmänkin käyttöä karsittaisiin.

6.2 Toteutuksen evaluointi

Kokonaisuutena hyväksymistestausjärjestelmän toteutus onnistui erittäin hyvin ja sen avulla on mahdollista jopa geneerisesti rakentaa web-sovelluksesta riippumattomasti hy-

väksymistestaus testauskohteena olevalle web-sovellukselle.

Testausjärjestelmän positiivisia puolia ovat muun muassa Docker säiliöinnin avulla saatava tuki myös manuaaliselle testitapauksien ajamiselle. Docker säiliö, joka mahdollistaa testitapauksien ajamisen voidaan pystyttää periaatteessa mihin tahansa ympäristöön, jossa Docker on saatavilla. Docker säiliöinnin avulla myös ohjelmistokehittäjät saavat valmiin hyväksymistestausjärjestelmän helposti käyttöönsä. Docker säiliöinnin ja Docker-compose:n avulla rakennettu järjestelmä ei myöskään ole sidottu mihinkään ennalta määritettyyn jatkuvan integroinnin palvelimeen, joka huomattavasti helpottaa testausjärjestelmän käyttöönottoa osaksi uusia tai muuttuvaa ohjelmistotuotannon prosessia. Testausjärjestelmä mahdollistaa päätteettömän testauksen virtuaalisen Xvfb-näyttöpalvelimen avulla, joka on itsessään erittäin tarvittu ominaisuus jatkuvan integroinnin ja testausjärjestelmän yhdistämiseen. Xvfb-näyttöpalvelimen tarjoaman virtualisoinnin avulla voidaan myös uusia WebDriver rajapinnan toteuttavia verkkoselaimia lisätä päätteettömän testauksen alaisuuteen erittäin helposti ja käytännössä rajoituksitta. Ainoa vaatimus on, että verkkoselain on saatavilla siihen ympäristöön jossa Xvfb-näyttöpalvelinta ajetaan.

Testausjärjestelmässä on kuitenkin myös negatiivisia puolia, joiden osalta järjestelmän käyttö on rajattua. Xvfb-näyttöpalvelimen avulla voidaan päätteettömästi testata periaatteessa mitä tahansa GUI-ohjelmia, mutta rajoitteena on kuitenkin, että niiden täytyy olla saatavilla siihen ympäristöön, jossa Xvfb-näyttöpalvelinta ajetaan. Tämä tarkoittaa käytännössä sitä, että web-sovelluksien hyväksymistestauksen automatisoimisesta on jätettävä pois vain Window-ympäristöön saatavien verkkoselainten, kuten Internet Explorer selaimen testaaminen. Robot Framework takaa helpon testitapauksien luettavuuden kelle tahansa, mutta ohjelmistokehittäjille se voi olla turhan rajatun tuntuinen. Ohjelmistokehittäjänä testitapauksien laatimisen yhteyteen olisi hyvä saada mahdollisuus yksikkötestauskehyksissä käytettävistä ohjelmointikielistä tuttuihin kontrollirakenteisiin, joilla testitapauksien monipuolisuutta voisi kasvattaa perinteisesti yksikkötestauksessa mahdollisten rakenteiden tasolle. Tämä ei kuitenkaan ole mahdollista Robot Framework:issä, jossa testitapauksien laatimiseen käytetään Robot Framework:in omaa, rajattua syntaksia.

6.3 Menetelmän evaluointi

Priorisointimenetelmän toistettavuus

Menetelmän avulla on mahdollista priorisoida näkymiä ja siirtymiä

Dijkstran algoritmin avulla voidaan selvittää verkosta prioriteetiltaan korkein polku, eli näkymät joiden testikokoelmat tulisi suorittaa ensimmäisenä.

Painotettu verkko voidaan piirtää, joka kasvattaa ymmärrystä järjestelmästä

Menetelmän käyttö on tehokkainta kun testitapaukset kategorisoidaan näkymittäin testikokoelmiin

Testikattavuuden päättäminen näkymä- ja siirtymäperusteisesti voi olla haastavaa

Ei välttämättä ainakaan muokkauksia tekemättä geneerinen menetelmä

Menetelmän käyttö soveltuu testikokoelmien, ei testitapauksien, priorisointiin

Priorisointimenetelmän käyttö voi olla turhan aikaa vievää jos käyttöliittymä on yksinkertainen

6.4 Jatkokehitysehdotukset

Xvfb:lle vastineen löytäminen Window ympäristöön

Parempi priorisointi muuttamalla näkymäperusteisuus käyttötapauserustaiseksi

Näkymägraafien visualisointi ja muu priorisointiohjelman jatkokehittely

7 YHTEENVETO

<TODO: diplomityön tavoite saavutettu....>

<TODO: tavoitteena oli xxxxxx>

Tutkimuskysymykseen *T1* vastattiin kokonaisuutena priorisointi painotetun verkon avulla luvussa 5, eli esitetään ratkaisuna toistettavissa oleva menetelmä testitapauksien priorisointiin.

Priorisointiin vaikuttavat muuttujat luvussa 5.3 esitetään myös suora vastaus tutkimuskysymykseen *T2*.

Lisäksi painofunktiot 5.4 ja verkon karsiminen 5.6 esittää vastaukset tutkimuskysymykseen *T3*.

Verkon ja testitapauksien yhteys 5.8 antavat vastaukset myös tutkimuskysymyksiin *T3* ja *T4*.

<TODO: konkreettisista tuloksista hyötyä.....>

<TODO: diplomityö loi perustan asiakasyrityksessä tarvittavan testiautomaation rakentamiseen....>

LÄHTEET

- [1] ISO:9126-1. *ISO/IEC 9126-1:2001*. en. 2001. URL: <http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/02/27/22749.html>.
- [2] *ISTQB Glossary*. 2019. URL: <https://glossary.istqb.org/en>.
- [3] *Robot Framework User Guide*. URL: <https://robotframework.org/robotframework/3.1.2/RobotFrameworkUserGuide.html>.
- [4] Klärck, P. *How to write good test cases using Robot Framework*. 2019. URL: <https://github.com/robotframework/HowToWriteGoodTestCases/blob/master/HowToWriteGoodTestrst>.

A ESIMERKKI TESTITAPAUKSESTA ROBOT FRAMEWORK:ILLÄ

```
1 *** Settings ***
2 Library      SeleniumLibrary
3 Library      XvfbRobot
4
5 *** Test Cases ***
6 Search TUNI from Google
7     Start Virtual Display      1920      1080
8     Open Browser      https ://www.google.com/      firefox
9     Set Window Size      1920      1080
10    Input Text xpath :// input [ @title = 'search ' ]      TUNI
11    Click Button xpath :// input [ @value = 'Google Search ' ]
12    Capture Page Screenshot      firefox_1920_1080.png
13    [Teardown]      Close BROWSER
```

B DIJKSTRAN ALGORITMI PSEUDOKOODINA

```

1 function Dijkstra(Graph, source):
2   // Distance from source to source
3   dist[source] := 0
4   // Initializations
5   for each vertex v in Graph:
6     if v != source
7       // Unknown distance function from source to v
8       dist[v] := infinity
9       // Previous node in optimal path from source
10      previous[v] := undefined
11    end if
12    // All nodes initially in Q
13    add v to Q
14  end for
15
16  // The main loop
17  while Q is not empty:
18    // Source node in first case
19    u := vertex in Q with min dist[u]
20    remove u from Q
21
22    // where v has not yet been removed from Q.
23    for each neighbor v of u:
24      alt := dist[u] + length(u, v)
25      // A shorter path to v has been found
26      if alt < dist[v]:
27        dist[v] := alt
28        previous[v] := u
29      end if
30    end for
31  end while
32  return dist[], previous[]
33 end function

```