Tampereen yliopisto

Jukka Pajarinen

WEB-KÄYTTÖLIITTYMÄN HYVÄKSYMISTESTAUKSEN PRIORISOINTI PAINOTETUN VERKON AVULLA

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Tammikuu 2020

TIIVISTELMÄ

Jukka Pajarinen: Web-käyttöliittymän hyväksymistestauksen priorisointi painotetun verkon avulla
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-ohjelma
Tammikuu 2020

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

ABSTRACT

Jukka Pajarinen: Web User Interface Acceptance Testing Prioritization with a Weighted Graph
Master's Thesis
Tampere University
Degree Programme in Information Technology
January 2020

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

ALKUSANAT

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 ohjelmistokehittäjäksi 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 Rintalaa 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 puolisolleni Katille, joka on tukenut ja kuunnellut aina vaikeinakin hetkinä diplomityöprosessin aikana ja siitäkin huolimatta kulkenut aina vierelläni.

Tampereella, 5. tammikuuta 2020

Jukka Pajarinen

SISÄLLYSLUETTELO

KUVALUETTELO

TAULUKKOLUETTELO

LYHENTEET JA MERKINNÄT

API	Application Programming Interface, ohjelmointirajapinta
ATDD	Acceptance Test-driven Development, hyväksymistestausvetoinen kehitys
C#	Microsoftin kehittämä oliopohjainen ohjelmointikieli
CI	Continuous Integration, jatkuva integrointi
DOM	Document Object Model, dokumenttiobjektimalli
e2e	End-to-end, akronyymi päästä päähän -testaukselle
Front-end	Asiakaspuoli asiakas-palvelin arkkitehtuurissa
GoCD	Go Continuous Delivery, jatkuvan integroinnin ja julkaisemisen työkalu
HTML	HyperText Markup Language, hypertekstin merkintäkieli
IDE	Integrated Development Environment, integroitu ohjelmointiympäristö
ISO	International Organization for Standardization, kansainvälinen standardoimisjärjestö
JSON	JavaScript Object Notation, JavaScript-kieleen pohjautuva tiedostoformaatti
MoSCoW	Must, Should, Could ja Would priorisointimenetelmä
SQL	Structured Query Language, kyselykieli relaatiotietokannan hallitsemiseen
UNIX	Yleinen tietokoneiden käyttöjärjestelmäperhe
XSS	Cross Site Scripting, eräs verkkosivuihin kohdistuva haavoittuvuus
Xvfb	X Virtual Framebuffer, X-ikkunointijärjestelmän protokollan toteuttava virtualisointipalvelin
YAML	Yleinen konfiguraatiotiedostoissa käytetty merkintäkieli

1 JOHDANTO

Nykypäivänä web-sovellukset ovat kasvaneet laajuudessa, toiminnallisuudessa ja kompleksisuudessa todella suuriksi [1]. Web-sovelluksissa on useita eri näkymiä, niiden välisiä siirtymiä sekä niiden sisältämää loppukäyttäjille tarkoitettua toiminnallisuutta. Web-sovelluksien ja niiden käyttöliittymien ja toiminnallisuuden laajuuden kasvaessa on erittäin tärkeää, että niiden testaamiseen otetaan mukaan myös hyväksymistestausta, joka varmistaa loppukäyttäjille suunnatun toiminnallisuuden toimivuuden. Hyväksymistestaus voidaan nykypäivänä automatisoida testitapauksien muodossa. Hyväksymistestauksen testitapauksien kattavuus on yksi haaste, sillä läheskään kaikkia sovelluksen toimintoja ei usein ole järkevää tai edes mahdollista testata [2]. Tämän lisäksi testitapauksien priorisointi on äärimmäisen tärkeää muun muassa kustannussyistä ja resurssien optimoinnin kannalta.

Tämä diplomityö on yhtenäinen web-käyttöliittymien hyväksymistestauksen automatisoimiseen ja siihen liittyvien testitapauksien priorisoimiseen liittyvä kokonaisuus. Työstä voidaan erotella kaksi eri osuutta, jotka ovat testiautomaation mahdollistava hyväksymistestausjärjestelmä ja hyväksymistestauksen testitapauksien priorisointimenetelmä. Hyväksymistestausjärjestelmän tarkoituksena oli luoda järjestelmä web-käyttöliittymien hyväksymistestauksen automatisoimiseen testitapauksien avulla. Priorisointimenetelmän tarkoituksena puolestaan oli edellä mainittuun järjestelmään rakennettavien testitapauksien priorisoiminen, jotta epäolennaiset testitapaukset voitaisiin jättää toteuttamatta ja keskittyä vain prioriteetiltaan tärkeiden testitapauksien rakentamiseen.

Luvussa kaksi esitetään tutkimusasetelma, joka sisältää tutkimuksen taustan, tutkimuskysymykset, tutkimuksen rajauksen sekä sen tavoitteet. Luvussa kolme käydään läpi testiautomaation teoriaa, joka on tarpeellista esitietoa työn toteutuksen ymmärtämistä ajatellen. Testiautomaation teoriasta olennaisena osana käydään läpi ohjelmistotestauksen tasot, jotta diplomityössä hyväksymistestaukseen kohdistunut painopiste tulee hyvin esille. Lisäksi testiautomaatioon liittyen käydään läpi myös testitapauksien, testikokoelmien ja jatkuvan integroinnin käsitteet, jotka liittyvät olennaisesti diplomityössä toteutettuun hyväksymistestausjärjestelmään. Luvussa neljä käydään läpi hyväksymistestausta sekä esitetään suunniteltu ja työn tuloksena syntynyt hyväksymistestausjärjestelmä. Lisäksi käydään läpi testausjärjestelmään rakennettavia testitapauksia ja alustetaan niiden priorisointiongelmaa. Luvussa viisi esitetään painotettuun verkkoon perustuvan priorisointimenetelmän toteutus. Ensin käydään läpi priorisointimenetelmään olennaisesti liittyvää matemaattisten verkkojen teoriaa, jonka jälkeen esitetään priorisointiin vaikuttavat muut-

tujat, painofunktiot priorisointiin, verkon rakentaminen ja verkkoon tehtävä karsinta leikkauksien avulla. Lisäksi käydään läpi Dijkstran algoritmin hyödyntämistä pintapuolisesti painotetussa verkossa ja verkon sekä testitapauksien yhteys. Luvussa kuusi tarkastellaan ja arvioidaan tutkimuksen konkreettiset tulokset, joita ovat hyväksymistestausjärjestelmä sekä kehitetty priorisointimenetelmä. Hyväksymistestausjärjestelmä ja priorisointimenetelmä evaluoidaan erillisissä kappaleissa, minkä jälkeen esitetään myös työn toteutuksen jälkeen syntyneitä jatkokehitysehdotuksia. Luvussa seitsemän esitetään tutkimuksen yhteenveto ja pohditaan tutkimuksen tavoitteiden saavuttamista ja tutkimuskysymyksiin vastaamisen onnistumista.

2 TUTKIMUSASETELMA

Tässä luvussa esitetään diplomityön tausta, tutkimuskysymykset, käytetty tutkimusmenetelmä, tutkimuksen rajaus 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 hyväksymistestausjärjestelmän toteuttamista sekä matemaattisen, toistettavissa olevan menetelmän kehittämistä hyväksymistestauksen testitapauksien priorisointiongelman ratkaisemiseksi. 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, tamperelainen kieltenoppimiseen keskittyvä yritys [3]. WordDivellä on 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 prosesseista 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.

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 toteutettuna 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 sovelluskehykset ja työkalut testiautomaatiota varten, joten tässä työssä ei enää ollut tarvetta evaluoida eri työkaluja tarvittavan testiautomaation toteuttamiseksi. Näistä syistä diplomityön aihetta ja tutkimusongelmaa lähdettiin etsimään muualta.

Lopullisen tutkimusongelman löytämiseen vaikuttivat erityisesti vasta web-sovelluksen testiautomaation suunnitteluvaiheessa esiin tulleet tarpeet. 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 oleelliseksi tutkimusongelmaksi. Testitapauksien priorisointiongelman valitsemiseen ratkaisevasti johtavia asioita olivat kaksi allekirjoittaneen oivallusta aiheesta. Ensimmäiseksi hyväksymistestattavaa web-sovellusta keksittiin ajatella käyttöliittymän näkymä ja siirtymätasolla matemaattisena prioriteetein painotettuna verkkona. Toiseksi oivallukseksi keksittiin käyttää lyhimmän polun ongelmaan kehitettyjä algoritmeja prioriteetein painotettuun verkkoon, jolloin kahden solmun välille voitiin löytää prioriteeteiltaan korkein polku. 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. Priorisointimenetelmän lisäksi tutkimuksessa toteutettiin hyväksymistestausjärjestelmä, johon hyväksymistestauksen testitapaukset voidaan toteuttaa. 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 priorisoimiseen?*
- **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 testiautomaation kanssa?*

2.3 Tutkimusmenetelmä

Tutkimuskysymyksiin vastaamiseksi työn tutkimusmenetelmäksi valittiin tietotekniikan diplo mitöissä yleisesti käytetty Design Science -menetelmä. Valittua menetelmää käytettäessä pyritään tutkimaan uusia teknologiaan hyödyntäviä ratkaisumalleja ratkaisemattomiin ongelmiin tai kehittämään parempia ratkaisumalleja jo aiemmin ratkaistujen ongelmien tilalle. Tietotekniikan tutkimuksessa on aikojen saatossa kehitetty uusia tai parempia tietokonearkkitehtuureja, ohjelmointikieliä, algoritmeja, tietorakenteita ja tiedonhallintajärjestelmiä. Näiden osalta yhteistä on, että niissä on monesti käytetty usein jopa tiedostamatta Design Science -menetelmää [4].

Tutkimuksen tarkoituksena oli muodostaa uudenlainen ja toistettavissa oleva menetelmä tutkimuksen kohteena olevan hyväksymistestauksen testitapauksien priorisointiongelman ratkaisemiseksi. Tutkimusidean hahmottelemisen ja ratkaisua kaipaavan ongelman identifioinnin jälkeen, valittua tutkimusmenetelmää käyttäen määriteltiin ensin tutkimuskysymykset. Seuraavaksi kartoitettiin ratkaisuvaihtoehto tutkimuskysymyksiin ja työn yleiseen priorisoinnin teemaan liittyvään tutkimusongelmaan vastaamiseksi. Tutkimusta varten käytettiin oman ammattitaidon lisäksi kirjallisuutta, 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 hyväksymistestausjärjestelmä, 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 testitapauksia voidaan toteuttaa. Tässä työssä testitapauksien toteuttaminen rajataan tietylle ennalta määräytyneelle hyväksymistestaukseen tarkoitetulle Robot Framework -alustalle. Tämä rajaus pohjautuu asiakasyrityksessä jo aiemmin valittuihin testauksen sovelluskehyksiin ja työkaluihin. Lisäksi Robot Framework on alustana yleisesti käytetty ja erityisen hyvin soveltuva etenkin hyväksymistestauksen toteuttamiseen. Robot Framework on soveltuu 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 kui-

tenkin 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 tarkastellaan 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 tavoitteena oli kehittää hyväksymistestausjärjestelmä ja 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.

Edellä mainitun lisäksi valmiin diplomityön tavoitteena on myös tarjota selkeä, eheä ja helposti ymmärrettävä kokonaisuus hyväksymistestauksen automatisoimiseen ja sen testitapauksien priorisoimiseen, 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. Lisäksi tutkimusmielessä tavoitteena oli pystyä todentamaan kehitetyn menetelmän toimivuus käytännössä menetelmää itsessään sekä sen lisäksi tehtyä hyväksymistestausjärjestelmän toteutusta evaluoimalla. Evaluointi esitetään diplomityön lopussa ja se esitetään erikseen menetelmälle sekä toteutukselle.

3 OHJELMISTOJEN TESTAUS JA TESTIAUTOMAATIO

Tässä luvussa esitetään perusteet ja tarvittavat tiedot ohjelmistojen testauksesta ja erityisesti 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 laa-

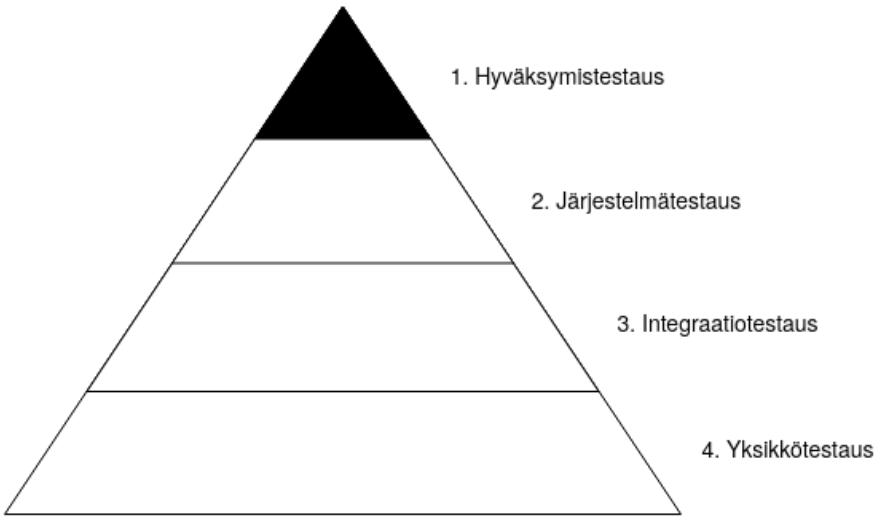
dullisia ominaisuuksia, joiden kattamisella voidaan validoida laatua. Laadullisia ominaisuuksia ovat ISO 9126-standardin mukaan [5]:

1. toiminnallisuus
2. luotettavuus
3. käytettävyys
4. tehokkuus
5. ylläpidettävyys
6. siirrettävyys

Näistä laadullisista ominaisuuksista testiautomaatiolla pystytään kattamaan erityisesti toiminnallisia, luetettavuudellisia 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, harmaalaatikkotestaus ja valkolaatikkotestaus, jotka eroavat toisistaan yleisesti ottaen siinä, otetaanko tieto ohjelmistotuotteen sisäisestä toteutuksesta mukaan itse testaamiseen. Testauksen tasot esitetään kirjallisuudessa usein hieman eri muodoissa [6] [7], 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 testiautomaatiota. 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 atomisin 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, jotta järjestelmätestauksen ja hyväksymistestauksen väliset eroavaisuudet tulevat lukijalle selkeästi esille.

Hyväksymistestaus on tämän diplomityön keskiössä ja siihen liittyvää teoriaa esitetään vielä laajemmin omassa luvussaan. 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. Yksikkötestausta hyödynnetään paljon myös testausvetoisen kehityksen aihepiirissä. Testausvetoisessa kehityksessä ohjelmistokehittäjät laativat ensin yksikkötestit, ennen yksiköiden toteuttamisen aloittamista. 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.

Ohjelmistotestauksen tasojen pyramidissa ja hyvin toteutetussa ohjelmistotestauksen monitasoisessa testauksessa tämä testauksen taso on usein testitapauksien määrässä kaikista laajin. Monitasoisessa testauksessa yksikkötestaus luo tärkeän pohjan testaamisel-

le 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 kokonaisena, 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 [5]. Aiemmin testiautomaation tarkoitus kappaleessa 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 tehok-

kuutta. Toiminnallisuutta voidaan testata käyttöliittymätestauksella, joka on mahdollista automatisoida käyttötapauksien muotoon. Luotettavuutta voidaan testata automaattisesti tietoturvaa testaavien käyttötapauksien muodossa. Tehokkuutta voidaan testata automaattisesti lisäämällä aikaleimoihin perustuvaa tarkastelua testitapauksiin, sekä tehdä kuormitusta testaavia testitapauksia. Edellä mainituista muista laadullisista ominaisuuksista voidaan kuitenkin ylläpidettävyyttä ja siirrettävyyttä testata toki manuaalisesti.

Testauksen tasona järjestelmätestaus voi olla testiautomaation teknisen toteutuksen kannalta jopa hyvin samanlainen kuin sitä kapeampi hyväksymistestaus. Usein kuitenkin hyväksymistestauksessa paneudutaan erityisesti vaatimusmäärittelyyn ja asiakaslähtöiseen testaamiseen, kun taas järjestelmätestauksessa voidaan testata esimerkiksi myös järjestelmän tehokkuutta tai tietoturvaa. 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.

Hyväksymistestauksen avulla voidaan testata erityisesti toiminnallisia laatuominaisuuksia, jotka usein toteutetaan käyttöliittymätasolla testitapauksien muodossa. Toiminnallisten ominaisuuksien lisäksi hyväksymistestauksessa voi olla mukana myös muitakin laadullisia ominaisuuksia jos ne ovat asiakastarpeiden mukaisia. 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 estää puutteellisesti toimivan ohjelmiston julkaiseminen. Hyväksymistestausta tarvitaan myös, jotta voidaan testata ja validoida vaatimusten ja loppukäyttäjän tarpeiden mukaisten ominaisuuksien toimivuus kokonaisessa järjestelmässä.

3.3 Testitapaukset 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 tuhotaan 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 suorituksestaan. 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.

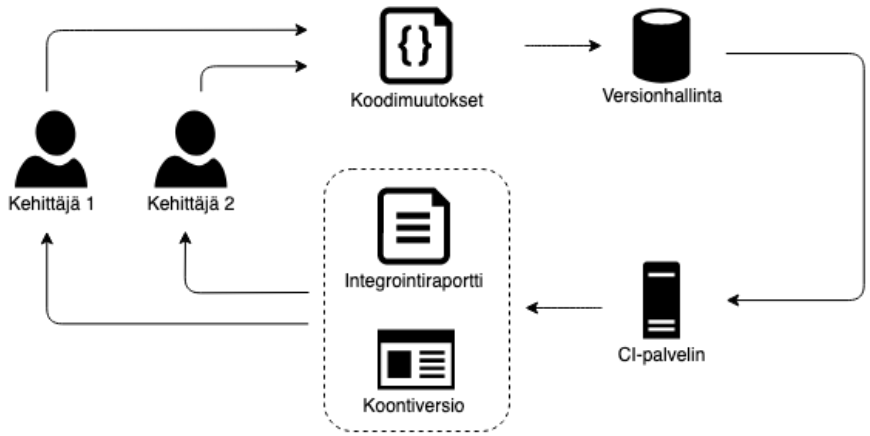
Testitapauksia voidaan ryhmitellä samaan kontekstiin liittyviksi testikokoelmiksi tilanteesta riippuen monilla eri tavoin. Ryhmittelyn perusteen valitsemiseen kannattaa käyttää harkintaa, sillä testikokoelmien laajuus on helpomman hallittavuuden takia tärkeää. Yksi tapa ryhmitellä testitapauksia on käyttää 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. Aiemmin testiautomaation tarkoitus kappaleessa 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 jatkuvaa. Jatkuvan integroinnin toteuttaminen tarvitsee teknisesti sen mahdollistavan versionhallintajärjestelmän ja varsinaisen jatkuvan integroinnin palvelimen.



Kuva 3.2. *Jatkuvan integroinnin perusperiaate*

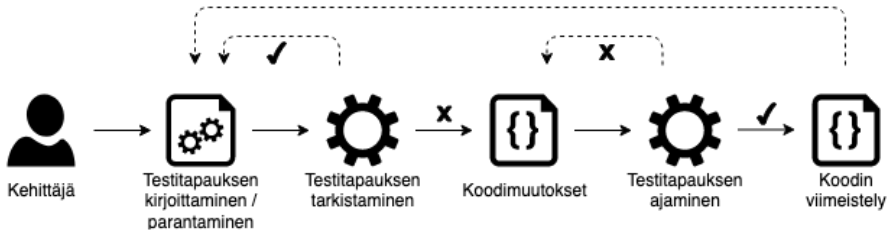
Esimerkkinä versionhallintajärjestelmänä voidaan käyttää nykyaikana suosittua git-ohjelmistoa ja jatkuvan integroinnin palvelimena esimerkiksi GoCD-ohjelmistoa. Perusideana jatkuvassa integroinnissa on konfiguroida jatkuvan integraation mahdollistava palvelinohjelmisto siten, että se kuuntelee versionhallintaan tulevia muutoksia ja suorittaa integrointiprosessin jatkuvasti aina muutoksia huomattuaan. Versionhallintaan 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 lopputuloksen on tarkoitus tarjota periaatteeltaan sama lopputulos kuin mitä se olisi manuaalisella integrointiprosessillakin. Jatkuvan integroinnin mahdollistava konfiguraatio sisältää aina jonkinlaisen koontiputken tai useita koontiput-

kia, joissa rakennetaan koontiversio kehitettävän ohjelmiston lähdekoodeista. 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 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 [8]. Näihin testiautomaatio on soveltunut huonosti, kun testattavaa ohjelmistoa tai lisättyä toiminnallisuutta ei ole vielä olemassa. Tähän ongelmaan on kehittynyt 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 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. Testausvetoista kehitystä voidaan hyödyntää ketterässä ohjelmistokehityksessä ja se on myös kasvattanut suosiotaan ketterien menetelmien mukana [9].

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 tämän diplomityön yhteydessä kehitetyn hyväksymistestausjärjestelmän rakentamiseen käytettyjä, mutta kuitenkin myös hyvin yleisiä hyväksymistestauksen työkaluja. Testausjärjestelmän rakenteen lisäksi käydään omassa kappaleessaan myös läpi 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 julkaistavaksi [10]. 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 sanotulla 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.

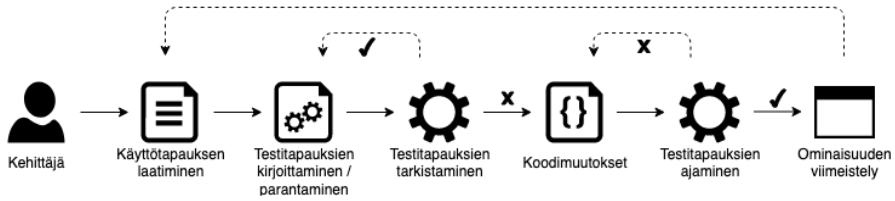
Hyväksymistestaus on aiemmin esitetyistä testauksen tasoista 3.2 viimeinen ja sen suorittamisen jälkeen saadaan tieto, onko järjestelmä toteutuksen osalta sellaisenaan valmis julkaistavaksi. Perinteisesti hyväksymistestauksen lähtökohtia ovat selvät hyväksymisvaatimukset sekä julkaisukelpoinen toteutus 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 toiminnalliset virheet ennen järjestelmän julkaisua loppukäyttäjille.

Kehittäjien käsitys järjestelmän toiminnallisuudesta ja sen vaatimuksista voi kuitenkin olla usein hyvinkin erilainen kuin loppukäyttäjien. Hyväksymistestauksen avulla voidaan 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 vaatimuksiin ja loppukäyttäjien toiminnallisiin tarpeisiin nähden.

4.2 Hyväksymistestausvetoinen kehitys

Hyväksymistestausvetoisen kehityksen tarkoituksena, kuten testausvetoisessakin kehityksessä 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 [11]. Tarvittavat ohjelmiston hyväksymistestit suoritetaan iteratiivisesti ohjelmistokehitysprosessin

aikana ja se tarkoittaa käytännössä jatkuvan integraation 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 on sen yläkäsitteen, testausvetoisen kehityksen, kanssa perusperiaatteeltaan samanlainen, mutta ennen ohjelmistokehityksen aloitusta asiakasvaatimukset kartoitetaan ja ohjelmiston hyväksyttävyys määritellään. Hyväksymistestitapaukset kirjoitetaan testausvetoisen kehityksen mukaisesti ennen toteutusta ja ohjelmistokehitys itsessään noudattaa iteratiivisesti testausvetoista kehitystä, vaikkakin hyväksymistestaus on perinteisesti vaatinut lähes valmista järjestelmää [12]. Asiakasvaatimukset määritetään usein käyttötapauksien muodossa ja hieman testialustasta riippuen ne voidaan kirjottaa suoraan testitapauksien muotoon. Hyväksymistestausvetoisessa kehityksessä ohjelmistokehitystä ohjaavat asiakasvaatimukset ja loppukäyttäjien tarpeiden toteutuminen, jotka ovat hyvin usein toiminallisia vaatimuksia. Hyväksymistestausvetoisessa kehityksessä mitataan jatkuvasti käyttötapauksien muodossa validoitavien haluttujen ominaisuuksien toteutumista. Perusperiaate on kirjoittaa asiakasvaatimus tai käyttötapaus testitapauksen muotoon, toteuttaa testitapaus, ajaa testitapaus läpäisemättömänä, toteuttaa ominaisuus, ajaa testitapaus läpäisevänä, refaktoroida toteutus ja siirtyä takaisin seuraavaan käyttötapaukseen. Käyttötapaus 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 testitapaus voi olla esimerkiksi käyttäjän tietojen muuttumisen 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 erittäin hyödyllistä, sillä jatkuva testaaminen antaa mahdollisuuden haluttujen ominaisuuk-

sien 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 erittäin 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 joko verkkoselaimet itse tai voidaan käyttää käyttöjärjestelmätasolla näyttöpalvelimen pro-

tokollan 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.

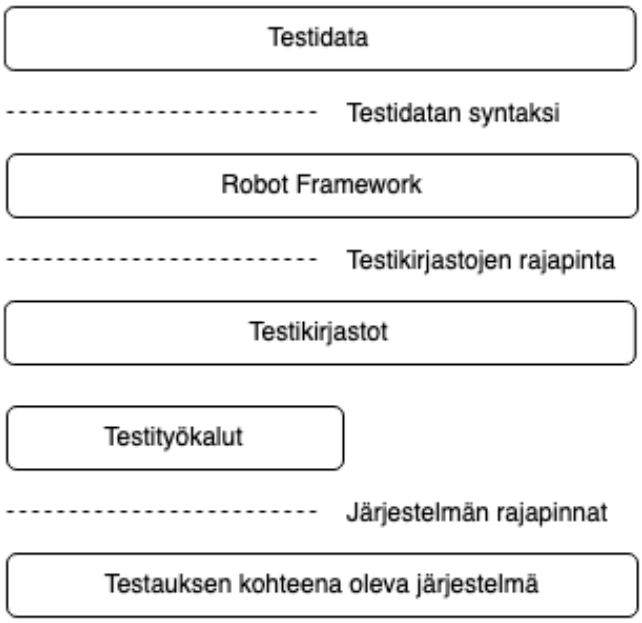
4.4 Hyväksymistestausjärjestelmä

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 virtualisoimisesta 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 [13]. Robot Framework:in avainsanaperustainen syntaksi on helposti ymmärrettävä, luettava ja selkeä. Testialustan etuna on helppo lähestyttävyys, eikä sen päälle rakennettujen testitapauksien ymmärtäminen vaadi ohjelmointikielien ymmärtämistä. Robot Framework on Python-ohjelmointikieleen perustuva 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 tägejä testitapauksien sisällä.



Kuva 4.2. *Robot Framework alustan arkkitehtuuri [14]*

Robot Framework:illä rakennettuja testitapauksia voidaan ajaa komentoriviltä sen tarjoamalla 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 hyödyntävillä testialustoilla löytyville kontrollirakenteille, joita esiintyy esimerkiksi yksikkötestaukseen tarkoitetuissa testialustoilla. Robot Framework on selkeästi vain hyväksymistestauksen testitapauksien rakentamista varten tarkoitettu testialusta ja siinä se on erinomainen vaihtoehto testitapauksien rakentamiseen.

4.4.2 Selenium

Selenium on suosittu avoimen lähdekoodin työkalu ja kirjastokokoelma 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 käyttöjärjestelmille [15]. Sama yhtiö on toteuttanut myös tässä diplomityössä myöhemmin esitettävän GoCD-ohjelmiston, jota voidaan käyttää jatkuvan integroimisen ja julkaiseminen rakentamiseen.

Selenium tuoteperheeseen kuuluvat Selenium WebDriver, Selenium IDE ja Selenium

Grid komponentit [16]. 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.

Selenium on todella tärkeä osa web-sovelluksien testiautomaation rakentamista, sillä se pohjimmiltaan mahdollistaa web-sovelluksien käyttöliittymien käsittelemisen automatisoidusti. 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 kommunikoivat 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.

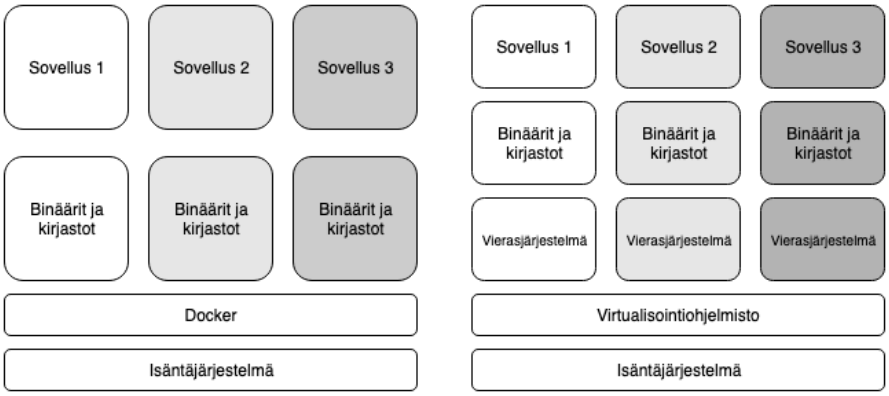
Xvfb soveltuu web-sovelluksien hyväksymistestauksen automatisointiin mahdollistaen päätteettömän testaamisen testitapauksille. Päätteetöntä testausta voidaan toteuttaa myös verkkoselaimiin rakennettujen ominaisuuksien avulla, mutta Xvfb:n suurena etuna niihin on se, että sitä voidaan käyttää mihin tahansa 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-pohjaisiinX-näyttöpalvelimen sisältäviin käyttöjärjestelmiin, kuten Linux ja MacOS. Näin ollen esimerkiksi Windows-alustalla toimivaa Internet Explorer verkkoselainta 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 kommunikoimiseen.

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 resurssien käytössä optimaalisempi, sillä se jakaa käyttöjärjestelmän ytimen eri säiliöiden kesken ja virtualisoi vain sovellusympäristön jonka säiliön määrittävä konfiguraatio sisältää. Säiliöiden sisään voidaan paketoida 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 arkkitehtuurivertailu [17]*

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 tietosäilöjä, 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 Ro-

bot Framework:iä, 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 jatkuvan integroinnin ja jatkuvan julkaisemisen mahdollistava palvelinohjelmisto. Ohjelmisto mahdollistaa koko koonti-testaus-julkaisu 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 [18].

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 suositun 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 vaiheita, 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 jouk-

koa. 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. Lisäksi hyväksymistestauksen priorisoimiseen painotetun verkon avulla on välttämätöntä suunnitella ja rakentaa testitapaukset näkymä- ja siirtymäperusteisesti, koska menetelmä hyödyntää matemaattisia näkymä- ja siirtymäperusteisesti laadittuja painotettuja verkkoja. Terminä näkymä- ja siirtymäperusteisuus tarkoittaa yksinkertaisesti web-käyttöliittymien hyväksymistestauksessa yksittäisiä käyttöliittymän näkymiä ja niiden välisiä siirtymiä. Yleisiä web-käyttöliittymien näkymiä ovat esimerkiksi kirjautumisnäkymä, päänäkymä ja asetusnäkymä joiden välillä käyttäjät voivat siirtymä näkymästä toiseen.

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 [19].

Robot Framework:in perustaja on kirjoittanut laajan ohjeistuksen siitä, miten Robot Framework:iä käyttäen luodaan hyviä testitapauksia [20]. 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. Tämän diplomityön liitteenä A on yksinkertaistettu esimerkki toteutettua hyväksymistestausjärjestelmää varten rakennetusta ja Robot Framework:iä käyttävästä testitapauksesta.

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 priorisoimisesta. 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 tuotteena 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 painotettuihin verkkoihin liittyviä käsitteitä. 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 Dijkstran algoritmia käyttäen.

5.1 Matemaattisten verkkojen tarkoitus

Matemaattisten verkkojen tarkoituksena on mallintaa parittaisia riippuvuuksia verkkomaisessa 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 [21].

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

Verkkoteoriaa käsittelevässä kirjallisuudessa [22][23][24] käytetään muun muassa seuraavia perusmerkintöjä ja käsitteitä:

- **Solmujoukko** $V = \{v_a, v_b, v_c\}$ on joukko joka sisältää solmut v_a, v_b ja v_c.
- **Kaarijoukko** $E = \{e_{ab}, e_{bc}, e_{ac}\}$ on joukko joka sisältää kaaret e_{ab}, e_{bc} ja e_{ac}.

- **Verkko** $G = V(G) \cup E(G)$ on joukko joka sisältää solmujoukon $V(G)$ ja kaarijoukon $E(G)$.
- **Aliverkko** $G_s \subset G$ on verkko G_s joka koostuu osasta verkon G solmuja ja kaaria.
- **Polku** $P = \{v_a, v_b, ..., v_n | v_a \rightarrow v_n\}$ on solmujono jota pitkin voidaan kulkea solmusta v_a solmuun v_n.
- **Sykli** $C = \{v_a, ..., v_n, ..., v_a \mid v_a \rightarrow v_a\}$ on sellainen polku, jonka aloitussolmu v_a ja lopetussolmu v_a ovat sama solmu siten, että polun jokaista kaarta kuljetaan vain kerran.
- **Verkon yhtenäisyys** $\forall\, v_a \neq v_b\, \exists\, P_{ab}$ tarkoittaa sitä, että $v_a \rightarrow v_b$, jokaiselle solmuparille on olemassa niitä yhdistävä polku.
- **Solmun asteluku** $d_G(v_a)$ on solmuun v_a liittyvien kaarten lukumäärä.
- **Eristetty solmu** on solmu v_a, jonka asteluku on nolla, eli $d_G(v_a) = 0$.
- **Silta** on solmujen v_a ja v_b välinen kaari e_{ab} siten, että $d_G(v_a) = 1$ ja $d_G(v_b) = 1$.
- **Silmukka** on kaari, jonka aloitus- ja lopetussolmu ovat sama solmu, eli $v_a \rightarrow v_a$.

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, joka voi johtaa nyt toteutettavien testitapauksien vanhentumiseen tulevaisuudessa. 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. Prioriteetin määrittäminen on tässä menetelmässä lineaarista, eli prioriteetti määräytyy sen osiensa summana laskukaavalla joka myöhemmin esitetään. Toisin sanoen epälineaarisuutta, eli sellaista tilannetta jossa prioriteettia ei syystä tai toisesta voitaisi ilmoittaa yksinkertaisesti osiensa summana, ei tässä menetelmässä oteta huomioon.

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 nollaksi, jolloin se sivuutetaan myöhemmin esitettävässä painofunktiossa.

Poikkeukselliset muuttujat ovat käyttötapauksien 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. *Näkymä- ja siirtymäperustaiseen priorisointiin vaikuttavat muuttujat*

m	**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	$\pm$	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 verkkoon sovellettavat lyhimmän polun algoritmit toimisivat halutulla tavalla, eli etsien prioriteetiltaan tärkeitä polkuja.

Ennen prioriteetin suunnanvaihtoa, voidaan kokonaisprioriteetti yksittäiselle solmulle eli näkymälle määrittää kaavalla

$$p(v) = \sum_{i=1}^{5} m_i - \sum_{j=6}^{9} m_j \pm m_{10}, \tag{5.4.1}$$

jossa kokonaisprioriteettia solmulle v kuvataan funktiona $p(v)$. Siinä kokonaisprioriteetin arvo määräytyy lineaarisesti osiensa m_k summana siten, että $1 \leq k \geq 10$. Toisin sanoen prioriteettiin vaikuttavia erilaisia muuttujia on kaavassa yhteensä kymmenen. Jokainen kaavassa esiintyvä muuttuja sisältää etumerkin aiemmin esitetyn taulukon 5.1 mukaisesti. Kaavassa esitetään ensin kokonaisprioriteettiin positiivisesti vaikuttavien muuttujien summa, joka sisältää etumerkiltään positiiviset muuttujat väliltä $1 \leq i \geq 5$. Samalla periaatteella kaavassa esitetään seuraavaksi negatiivisesti vaikuttavat muuttujat väliltä $6 \leq j \geq 9$, jotka lasketaan ensin yhteen ja vähennetään sitten yhteisesti etumerkillään negatiivisena edellisestä summasta. Lopuksi kaavassa on esitetty vielä viimeinen muuttuja m_{10}, joka tarkoittaa taulukon 5.1 mukaisesti omaa lisämuuttujaa tai lisämuuttujia, joiden etumerkki voi olla joko positiivinen tai negatiivinen. Lisämuuttujien sisällyttämisellä

kaavaan on tarkoitus mahdollistaa ja selventää kokonaisprioriteetin laskeminen erilaisissa kontekstiriippuvaisissa tilanteissa sekä osittain myös rohkaista kaikkien prioriteettiin vaikuttavien muuttujien evaluointiin ja muokkaamiseen kontekstista riippuen.

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 nollalle yrittäisi etsiä käänteislukua, tulisi eteen nollalla jakaminen jota ei voi tehdä. Nämä molemmat ongelmatapaukset voidaan kuitenkin painofunktioissa ratkaista siten, että käänteisfunktiota ei etsitä, vaan korvataan painofunktion tulos yhdellä.

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

$$\alpha(v) = \begin{cases} p^{-1}(v) & p(v) > 0 \\ 1 & p(v) \leq 0 \end{cases}, \tag{5.4.2}$$

jossa solmun v käännettyä kokonaisprioriteettia kuvataan funktiona $\alpha(v)$. Solmun kokonaisprioriteettia vastaava käänteisluku etsitään vain siinä tapauksessa jos kokonaisprioriteetti on suurempi kuin nolla. Jos vastaan tulee tilanne, jossa kokonaisprioriteetti olisi negatiivinen tai yhtä suuri kuin nolla ei käänteislukua yritetä ottaa vaan tulos korvataan suoraan käännettyjen prioriteettien alhaisimmalla arvolla, eli luvulla yksi. Toisin sanoen painofunktiosta saatava numeerinen arvo tarkoittaa käytännössä sitä, että mitä pienempi se on sitä korkeampaa prioriteettia se edustaa.

Painofunktio yksittäiselle kaarelle e_{ab} eli solmuja v_a ja v_b yhdistävälle käyttöliittymän näkymien väliselle siirtymälle saadaan kaavalla

$$\beta(e_{ab}) = \begin{cases} [p(v_a) + p(v_b)]^{-1} & p(v_a) + p(v_b) > 0 \\ 1 & p(v_a) + p(v_b) \leq 0 \end{cases}, \tag{5.4.3}$$

jossa kaaren e_{ab} käännettyä kokonaisprioriteettia kuvataan funktiona $\beta(e_{ab})$. Kaaren painofunktiota varten pitää kuitenkin huomioida, että sen kokonaisprioriteetti on kaaren päätepisteiden, eli aloitus- ja lopetussolmujen kokonaisprioriteetin summa $p(v_a) + p(v_b)$. Kaaren kokonaisprioriteetti pitää siis laskea ennen käänteisluvuksi muuttamista. Kaaren painofunktioon $\beta(e)$ pätee samat reunaehdot kuin yksittäisen solmun painofunktioonkin $\alpha(v)$.

5.5 Verkon rakentaminen

Tässä diplomityössä on aiemmin moneen otteeseen kerrottu näkymä ja siirtymäperusteisesta 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. Jos käyttöliittymän näkymä sisältää tilan, eli kyseinen näkymä muuttuu käyttäjän tekemien toimenpiteiden perusteella, käsitellään kyseinen tilallinen näkymä painotetussa verkossa kuitenkin yhtenä solmuna. Tällaisessa tapauksessa menetelmän käyttö priorisoi kyseisen näkymän joka periaatteessa vastaa sellaista testikokoelmaa, johon rakennettaisiin testitapaukset eri tilanteita varten. Siirtymät ovat usein sivujen välisiä linkkejä tai vaihtoehtoisesti jotakin sellaista toiminnallisuutta, joka muuttaa nykyisen näkymän tai osan siitä toiseksi näkymäksi.

Seuraavassa taulukossa 5.2 on esitetty kuvitteellisen web-sovelluksen mukainen näkymien ja siirtymien mukaan laadittu esimerkki. 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 $\beta(e)$ avulla. Painoarvo lasketaan kyseisen funktion avulla jokaiselle kahta näkymää yhdistävälle siirtymälle, eli painotetun verkon solmujen väliselle kaarelle. Painofunktio $\beta(e)$ käyttää kaaren molempien päätepisteiden yhteenlaskettua prioriteettia, josta käänteisluku otetaan. Näin saadaan laskettua kaarelle sellainen painoarvo, joka tar-

koittaa painotetussa verkossa siirtymän näkymiin sidottua prioriteettia. Esimerkkinä voidaan laskea taulukon 5.2 mukaisten näkymien v_a ja v_b välisen siirtymän, eli kaaren e_{ab} painoarvo $\beta(e_{ab})$ seuraavalla laskutoimituksella

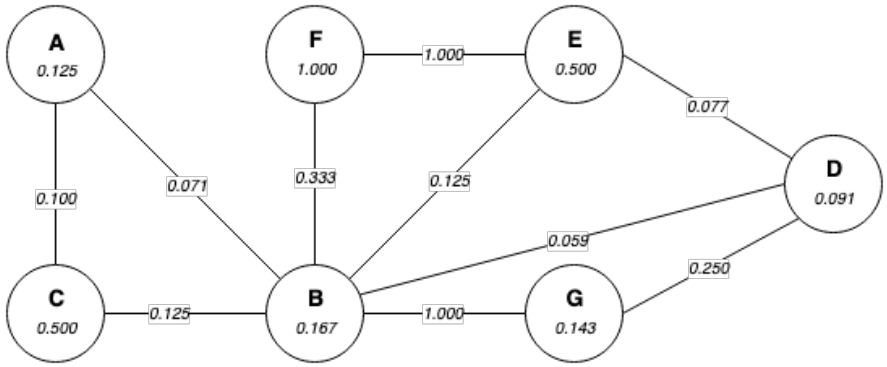
$$\beta(e_{ab}) = [p(v_a) + p(v_b)]^{-1} = (8+6)^{-1} = \frac{1}{14} \approx 0.071. \tag{5.5.1}$$

Painofunktioiden lisäksi painotetun verkon esittämistä varten tarvitaan painomatriiseja. Painomatriisin avulla voidaan rakentaa matemaattinen painotettu verkko, joka kuvaa näkymiä ja siirtymiä sekä niiden prioriteetteja. Painomatriiseissa tulee lähes väistämättä esiin tilanne, jossa pitäisi määrittää painoarvo olemattomalle silmukalle, eli kaarelle jonka aloitussolmu ja lopetussolmu ovat sama solmu itsessään, mutta kaarta ei ole olemassa. Tällaisissa tapauksissa, tilanteesta riippuen painomatriiseihin usein merkitään 0, ∞ tai $-$ [22]. Tässä diplomityössä esitettävän menetelmän painomatriiseissa solmuun itseensä johtuvan kaaren, eli silmukan painoksi merkitään aina $-$, koska käyttöliittymän näkymästä siirtymät itseensä ei tässä menetelmässä käsitellä varsinaisina 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. Painomatriisi taulukon 5.2 mukaiselle esimerkille ja siitä lasketuille painoarvoille on pyöristettynä kolmen desimaalin tarkkuudelle seuraavanlainen

$$M_G \approx \begin{array}{c} \\ v_a \\ v_b \\ v_c \\ v_d \\ v_e \\ v_f \\ v_g \end{array} \begin{array}{c} \begin{array}{ccccccc} G & v_a & v_b & v_c & v_d & v_e & v_f & v_g \end{array} \\ \begin{pmatrix} - & 0.071 & 0.100 & - & - & - & - \\ 0.071 & - & 0.125 & 0.059 & 0.125 & 0.333 & 1.000 \\ 0.100 & 0.125 & - & - & - & - & - \\ - & 0.059 & - & - & 0.077 & - & 0.250 \\ - & 0.125 & - & 0.077 & - & 1.000 & - \\ - & 0.333 & - & - & 1.000 & - & - \\ - & 1.000 & - & 0.250 & - & - & - \end{pmatrix} \end{array}. \tag{5.5.2}$$

Painomatriisin määrittämisen jälkeen voidaan edetä painotetun verkon kuvaamiseen, jossa piirretään jokainen erilaista käyttöliittymän näkymää vastaava, esimerkkidatan mukainen solmu ja niiden välisiä siirtymiä kuvaavat yhteydet eli kaaret. Kaarien yhteyteen lisätään painomatriisista kaaren prioriteettia kuvaava painoarvo. Seuraavassa on esitetty painomatriisia vastaava painotetun verkon kuvaaja sellaisena kuin se on ennen siihen tehtäviä prioriteettileikkauksia. Toisin sanoen kyseinen painotetun verkon kuvaaja on lähtötilanne, josta priorisointi aloitetaan. Priorisoimista varten tehtävien leikkauksien te-

keminen niitä varten kehitetyllä toistettavalla karsimisalgoritmilla esitetään myöhemmin omassa kappaleessaan.



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 on kuitenkin lisätty painomatriisin sisältämän informaation lisäksi painofunktion $\alpha(v)$ avulla lasketut yksittäisten solmujen eli näkymien painoarvot. Esimerkkinä voidaan laskea taulukon 5.2 mukaisen näkymän eli solmun v_a painoarvo $\alpha(v_a)$ seuraavalla laskutoimituksella

$$\alpha(v_a) = p^{-1}(v_a) = 8^{-1} = 0.125. \tag{5.5.3}$$

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.

5.6 Verkon karsiminen

Painotetun verkon karsiminen eli sen kaarien leikkaaminen on yksi prioriteeillä painotetun verkon erittäin tärkeä ominaisuus. Verkkoteorian soveltaminen prioriteettien avulla painotettuun verkkoon on erityisen hyödyllistä silloin, kun verkon kaarissa alhainen paino tarkoittaa suurta prioriteettia. Tässä diplomityössä verkon karsiminen tapahtuu varta vasten kehitetyllä kolmivaiheisella iteratiivisesti toistettavalla algoritmilla, jonka käyttämistä varten valitaan ensin kattavuus joka vastaa sitä minimirajaa jonka jälkeen karsiminen lopetetaan. Kattavuus tarkoittaa samalla myös testikattavuutta testikokoelmien näkökulmasta, sillä painotetussa verkossa jokainen solmu eli näkymä voidaan ajatella vastaavan sen mukaan kategorisoitua testikokoelmaa.

Verkkoon tehtäviä leikkauksia varten määritettävä kattavuus c on prosentuaalinen raja

sille kuinka suuri osa verkon solmuista eli näkymistä täytyy verkkoon jäädä karsimisen jälkeen. Leikkauksien tekemistä suoritetaan toistuvasti niin kauan, kuin karsittavan verkon solmujen lukumäärä on suurempi, mitä kattavuuden määräämä alaraja sallii, tai jos kyseisellä iteraatiokerralla ei yksinkertaisesti enää löydy algoritmiin kuuluvilla toimenpiteillä poistettavia solmuja. Matemaattiseen muotoon kirjoitettuna kattavuuteen perustuva toistamisen lopettava ehto on

$$|V(G_s)| > \frac{c}{100} \cdot |V(G)|, \tag{5.6.1}$$

jossa $|V(G_s)|$ tarkoittaa karsitun verkon solmujoukon mahtavuutta, ja vastaavasti $|V(G)|$ tarkoittaa alkuperäisen verkon solmujoukon mahtavuutta eli solmujen lukumäärää. Ehto voidaan myös ajatella yksinkertaisesti aliverkon solmujen lukumääränä, jonka täytyy olla suurempi kuin muuttujan c mukainen prosentuaalinen osuus alkuperäisen verkon solmujen lukumäärästä. Tässä verkon karsimisen esimerkissä kattavuutena käytetään $c = 80$, joka tarkoittaa esimerkkiverkon alkuperäisten solmujen määrän 7 karsimista määrään $80 \cdot \frac{7}{100} = 5.6$, eli lukumäärään 5 asti.

Algoritmissa on kolme erilaista toimenpidettä verkon karsimiseen. Toimenpiteillä on suoritusjärjestys, jonka mukaan kyseisellä iteraatiokerralla tehtävä leikkaus määräytyy. Jokaisella toimenpiteellä on myös oma suoritusehto, jonka täytyy täyttyä ennen kyseisen toimenpiteen suorittamista. Suoritusjärjestyksen ja ehtojen perusteella siirrytään toimenpiteestä toiseen yhden iteraatiokerran aikana siten, että jos esimerkiksi ensimmäistä toimenpidettä ei sen suoritusehdon mukaan voida tehdä yritetään suorittaa toimenpiteistä seuraavaa. Jokaisella iteraatiokerralla suoritetaan vain yksi toimenpide, jonka jälkeen aloitetaan uusi iteraatiokierros. Algoritmiin sisältyy seuraavat toimenpiteet niiden oikeaan suoritusjärjestykseen järjestettynä.

1. **Poistetaan verkosta löytyvä eristetty solmu** eli solmu jonka asteluku on nolla. Eristettyjä solmuja ei alkuperäisestä verkosta pitäisi löytyä lainkaan, vaan ne ovat seurausta eri iteraatiokerroilla tapahtuneista leikkauksista. Toimenpiteen suoritusehto on muotoa

   $$d_G(v) = 0, \tag{5.6.2}$$

   jossa $d_G(v)$ tarkoittaa astelukua kuvaavaa funktiota solmulle v.

2. **Poistetaan verkosta löytyvä sillattu solmu** leikkaamalla siihen liittyvä kaari. Solmun asteluvun täytyy olla yksi ja sen painon pienempi kuin alkuperäisen verkon solmujen painojen keskiarvo. Toimenpiteen suoritusehto on kaksiosainen ja muotoa

   $$d_G(v) = 1 \quad \wedge \quad \alpha(v) < \frac{1}{|V(G)|} \cdot \sum_{v \in V(G)} \alpha(v), \tag{5.6.3}$$

   jossa $d_G(v)$ tarkoittaa astelukua solmulle v ja $|V(G)|$ tarkoittaa alkuperäisen verkon solmujen lukumäärää sekä $\sum_{v \in V(G)} \alpha(v)$ tarkoittaa alkuperäisen verkon solmujen pai-

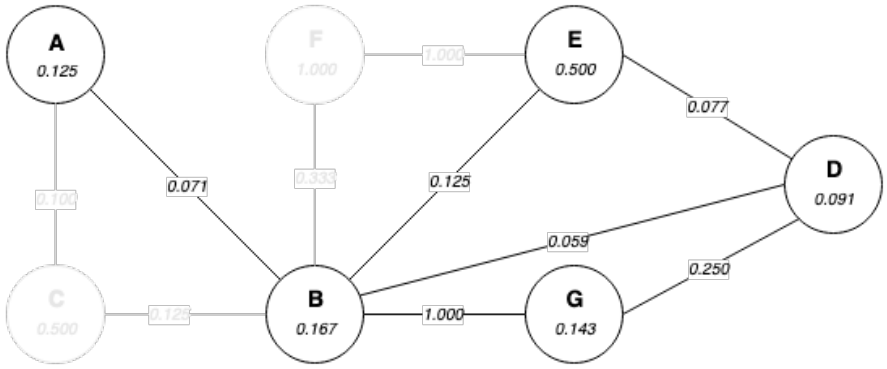
noarvojen yhteenlaskettua summaa.

3. **Poistetaan verkosta solmu, jolla on maksimia pienempi asteluku ja keskimääräistä pienempi paino.** Toisin sanoen poistetaan verkosta sellainen alhaisimman prioriteetin solmu, jonka asteluku on pienempi kuin solmujen astelukujen keskiarvo ja sen paino on pienempi kuin alkuperäisen verkon solmujen painojen keskiarvo. Toimenpiteen suoritusehto on kaksiosainen ja muotoa

$$d_G(v) < \max\{d_G(x)|x \in V(G)\} \quad \wedge \quad \alpha(v) < \frac{1}{|V(G)|} \cdot \sum_{v \in V(G)} \alpha(v), \tag{5.6.4}$$

jossa $d_G(v)$ tarkoittaa astelukua solmulle v ja $\max\{d_G(x)|x \in V(G)\}$ tarkoittaa maksimia alkuperäisen verkon solmujen asteluvuista. Keskiarvon laskemiseen liittyvät merkinnät $|V(G)|$ ja $\sum_{v \in V(G)} \alpha(v)$ ovat samat kuin edeltävässäkin toimenpiteessä.

Tässä diplomityössä läpikäytävässä taulukon 5.2 mukaisessa verkon karsimisen esimerkissä algoritmia suoritetaan kaksi iteraatiokierrosta. Kahden iteraatiokierroksen jälkeen verkon solmujen määrä karsiutuu prioriteettien ja valitun kattavuuden $c = 80$ mukaisesti seitsemästä viiteen.



Kuva 5.2. *Esimerkki painotetusta verkosta leikkauksien jälkeen*

Verkon karsimisen jälkeen voidaan huomata, että verkosta on karsiutunut pois kaksi prioriteeteiltaan alhaisinta solmua C ja F. Lisäksi alkuperäisestä verkosta 5.1 voidaan huomata, että solmuilla C ja E olisi keskenään yhtä suuri prioriteetti, mutta asteluvuiltaan ne ovat erilaiset. Algoritmi toimii kyseisessä tapauksessa oikein ja karsii edellä mainituista solmuista solmun C. Esimerkin mukaisen solmujen karsimisen myötä voidaan todeta, että algoritmi toimii siten kuin sen kuuluukin.

5.7 Dijkstran algoritmin hyödyntäminen

Priorisointimenetelmän mukaan karsittuun painotettuun verkkoon on mahdollista soveltaa lyhimmän polun ongelman ratkaisemiseen kehitettyjä algoritmeja. Ne toimivat normaaliin tapaan etsien numeerisesti alhaisimman painoarvon polkuja, joka kuitenkin tässä

menetelmässä tarkoittaa käänteisesti korkeimman prioriteetin polkuja. Lyhimmän polun etsimiseen on tarkoitus valita sellaiset aloitus- ja lopetussolmut, joiden välille lyhin polku verkossa halutaan etsiä. Lyhimmän polun löytymisen yhtenä perusedellytyksenä on verkon yhtenäisyys, joka tarkoittaa sitä, että mistä tahansa yksittäisestä verkon solmusta on löydettävissä yhteys mihin tahansa toiseen samassa verkossa olevaan solmuun.

Prioriteeteiltaan tärkeimmän polun löytämiseksi valitaan ensin aloitussolmuksi pienimmän painoarvon solmu ja lopetussolmuksi seuraavaksi pienimmän painoarvon solmu sekä etsitään sitten Dijkstran algoritmia hyödyntäen 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 pienimmille, eli prioriteeteiltaan korkeimmille solmuille antaa tuloksena kyseisiä solmuja yhdistävän polun, jonka sisältävät solmut eli näkymät ovat prioriteetiltaan tärkeimmät. Jos aloitus- ja lopetussolmut ovat vierekkäiset eli niitä yhdistää kaari, ei polussa luonnollisesti ole kuin kaksi solmua. Muussa tapauksessa lyhin mahdollinen polku kertoo kaikki sellaiset käyttöliittymän näkymät, jotka ovat yhteydessä toisiinsa ja yhteisesti prioriteeteiltaan tärkeimmät. Koska painotetun verkon painofunktiot on laadittu käänteislukuja hyödyntäen Dijkstran algoritmi löytää siis painoarvoltaan matalimman, mutta prioriteetiltaan tärkeimmän polun aloitus- ja lopetussolmujen välille. Näin ollen saadaan helposti ja vaivattomasti tietää sellaiset solmut eli käyttöliittymän näkymät jotka kuuluvat prioriteetiltaan tärkeimpiin ja joista testiautomaation rakentaminen kannattaa aloittaa.

5.8 Verkon ja testitapauksien yhteys

Priorisointi painotetun verkon avulla havainnollistaa käyttöliittymän näkymiä ja niiden välisiä siirtymiä ennen varsinaisten testitapauksien rakentamista. Tällaisesta painotetusta verkosta saadaan priorisoitua käyttöliittymän näkymät ja siirtymät. Tämä tarkoittaa käytännössä sitä, että prioriteettijärjestys saadaan määritettyä vain testikokoelmien laajuudella yksittäisien johonkin testikokoelmaan liittyvien testitapauksien sijaan. 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.

Aiemmin testitapaukset ja testikokoelmat kappaleessa on esitetty niiden välistä eroa ja sitä kuinka testikokoelmat koostuvat yhteen liittyvistä testitapauksista. Painotetun verkon avulla tehtävää priorisointia käyttäessä on tarkoitus ajatella testiautomaation testitapauksien kategorisoimista testikokoelmiksi käyttöliittymän näkymiä vastaavalla tavalla. Kun käyttöliittymän näkymillä on niitä vastaavat testikokoelmat, toimii tässä diplomityössä kehitetty painotetun verkon avulla toteutattava 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 merkitsevyydestä. 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 lähestysmistavaksi 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 Xfvb-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 verkkoselaimen testaaminen. Robot Framework takaa helpon testitapauksien luettavuuden kenelle 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 mahdollisien rakenteiden tasolle. Tämä ei kuitenkaan ole mahdollista Robot Framework:issä, jossa testitapauksien laatimiseen käytetään Robot Framework:in omaa, rajattua syntaksia.

6.3 Menetelmän evaluointi

Tässä diplomityössä kehitetyn testitapauksien priorisointimenetelmän kehittäminen onnistui myös erittäin hyvin ja sen käyttämisellä saavutetaan lisäarvoa etenkin keskisuurien ja suurien web-sovelluksien käyttöliittymien hyväksymistestaukseen.

Priorisointimenetelmän positiivisia puolia ovat muun muassa sen ominaisuuksiin liittyviä asioista kuten priorisointimenetelmän toistettavuus ja mahdollisuus priorisoida käyttöliittymien näkymiä ja siirtymiä. Näkymä ja siirtymäperustaisen priorisoinnin tarkoituksena on mahdollistaa näkymiin perustuvien testikokoelmien priorisoiminen, jolloin niiden tärkeysjärjestys saadaan selville ja testitapauksien kirjoittaminen voidaan aloittaa prioriteetiltaan tärkeimmästä näkymästä. Menetelmän käyttäminen on tehokkainta kun testitapaukset

kategorisoidaan näkymittäin laadittuihin testikokoelmiin, sillä menetelmän kehittämisen taustalla on ollut ajatus jossa näkymät vastaavat testikokoelmia. Priorisointimenetelmä perustuu matemaattisiin painotettuihin verkkoihin, jotka tuovat hyötynä lyhimmän polun ongelman ratkaisemiseen kehitettyjen algoritmien käyttämisen mahdollistamisen prioriteetiltaan korkeimpien polkujen löytämiseen kahden solmun, eli näkymän välille. Lisäksi painotetun verkon ja matemaattisen lähestymistavan käyttäminen tuo hyötynä sen, että menetelmä on kohtalaisen pienellä vaivalla muunnettavissa tietokoneohjelmaksi. Painotettujen verkkojen käyttäminen priorisointiin pakottaa myös menetelmän käyttäjät piirtämään näkymä ja siirtymäperustaisen painotetun verkon, jolloin se kasvattaa käyttäjien ymmärrystä testauskohteena olevasta järjestelmästä. Priorisointimenetelmä on tässä diplomityössä esitetyn esimerkin 5.2 mukaan todettavissa toimivaksi ja sen avulla on suoritettu priorisointi varsinaisesta testauskohteesta yksinkertaistetulle käyttöliittymälle.

1. <TODO: Näkymien ja siirtymien määrä esimerkissä ja arvio järkevästä näkymien määrästä priorisointia varten>
2. <TODO: Testien määrät näkymää kohden, tähän joku lähde yleisistä määristä>
3. <TODO: Resurssien säästö näkymissä (c = 80%) ja näin ollen säästyneistä testitapauksista>

Myös priorisointimenetelmässä on negatiivisiakin puolia, mutta ne eivät tässäkään tapauksessa ylitä menetelmän käytöstä saatavaa hyötyä. Menetelmässä esittävien toistuvien leikkauksien määräää rajaavan testikattavuuden päättäminen näkymä ja siirtymäperusteisesti voi olla haastavaa. Toinen menetelmään kohdistuva kritiikki koskee menetelmän geneerisyyttä, eli käyttöönottamisen mahdollisuutta ilman muutoksia, jota on vaikea arvioida. Priorisointimenetelmässä käytettävät priorisointiin vaikuttavat muuttujat ovat varsin subjektiivisia ja voivat olla testausta toteuttavan tahon mukaan muuttuvia, jonka takia muuttujiin joudutaan mahdollisesti tekemään muutoksia. Lisäksi menetelmässä esitetty priorisointiin vaikuttavia muuttujia hyödyntävä funktio $p(v)$ kokonaisprioriteetin laskemiseen ei ota muuttujien määrittämisessä mahdollisesti esiintyvää epälineaarisuutta lainkaan huomioon. Tämä tarkoittaa käytännössä sitä, että kokonaisprioriteetin määrittäminen on voitava olla ilmaistavissa siihen vaikuttavien osiensa summana, joka rajoittaa menetelmän käyttöä epälineaarisissa tapauksissa. Negatiivista on myös se että menetelmän käyttö on soveltuva painotettujen verkkojen luonteen mukaisesti käyttöliittymien tapauksessa soveltuvat vain kokonaisia näkymiä peilaavien testikokoelmien priorisointiin yksittäisten testitapauksien sijaan. Priorisointimenetelmän käyttö voi olla turhan aikaa vievää jos käyttöliittymä on yksinkertainen.

6.4 Jatkokehitysehdotukset

Tämän diplomityön konkreettisina tuloksina syntyneet testausjärjestelmä ja priorisointimenetelmä ovat sellaisenaan käyttövalmiita ja toimivaksi todettuja, mutta jatkokehittelylle on luonnollisesti niissäkin sijaa. Testausjärjestelmän avulla toteuttava web-sovelluksien päätön hyväksymistestaus mahdollisestaan Xvfb-näyttöpalvelimen tarjoaman virtualisoin-

nin avulla. Xvfb-näyttöpalvelin on kuitenkin saatavilla vain UNIX-ympäristöihin, joka rajaa testausjärjestelmään lisättävien verkkoselaimien saatavuutta. Xvfb-näyttöpalvelimelle voitaisiin jatkokehityksenä etsiä monialustaisempi vaihtoehto tai ainakin vastine Window-ympäristöön, jonka avulla myös vain Window-alustalle saatavat verkkoselaimet olisi mahdollista lisätä järjestelmään.

Yksi priorisointimenetelmään liittyvä rajoite on käyttöliittymän näkymä ja siirtymäperustainen priorisointi, joka asettaa näkymät vastaamaan testikokoelmia. Jatkokehityksenä voitaisiin tutkia näkymäperusteisuuden mukaan tehtävän priorisoinnin muuntamisen mahdollisuutta käyttötapausperustaiseksi. Käyttötapausperustaisesti luotava verkko parhaimmillaan vastaisi oikeita käyttäjien tarpeisiin tarkoitettuja toiminallisuuksia ja voisi parantaa priorisointia. Priorisointimenetelmä on vahvasti matemaattinen, joka mahdollistaa sen muuntamisen kohtalaisella vaivalla tietokoneohjelmaksi. Priorisointimenetelmän rakentaminen automaattisen tietokoneohjelman muotoon olisi erittäin järkevää ja laskisi menetelmän käyttöönottamiseen tarvittavaa vaivannäköä huomattavasti. Lisäksi priorisointimenetelmän näkymäpohjaisten graafien, eli painotettujen verkkojen visualisointi voitaisiin hoitaa tietokoneohjelman yhteydessä.

7 YHTEENVETO

Tämän diplomityön tavoitteena oli jo aluksi laaditun tutkimusasetelmankin mukaisesti kehittää hyväksymistestausjärjestelmä ja toistettavissa oleva menetelmä web-käyttöliittymien hyväksymistestauksessa tarvittavien testitapauksien priorisointiin. Lisäksi tavoitteena oli tarjota selkeä, eheä ja helposti ymmärrettävä kokonaisuus hyväksymistestauksen toteuttamiseen ja priorisoimiseen testausjärjestelmää ja kehitettyä priorisointimenetelmää käyttäen. Tutkimusta varten laadittiin neljä tutkimuskysymystä, joihin vastaaminen asetettiin myös yhdeksi työn tavoitteeksi.

Tutkimuskysymykseen *T1* vastattiin kokonaisuutena priorisointi painotetun verkon avulla luvussa, eli esitetään ratkaisuna toistettavissa oleva menetelmä testitapauksien priorisoimiseen. Priorisointiin vaikuttavat muuttujat luvussa esitetään myös suora vastaus tutkimuskysymykseen *T2*. Lisäksi painofunktiot priorisointiin ja verkon karsiminen esittävät vastaukset tutkimuskysymykseen *T3*. Lopuksi verkon ja testitapauksien yhteys luku antaa suoraan vastauksen tutkimuskysymykseen *T4*.

Tässä diplomityössä kehitettiin hyväksymistestausjärjestelmä sekä siihen rakennettavien testitapauksien priorisointimenetelmä. Hyväksymistestausjärjestelmä laadittiin käyttäen Robot Framework:iä, Selenium:ia, Xvfb-näyttöpalvelinta ja Docker:ia. Lisäksi jatkuvan integroinnin tarve otettiin huomioon käyttäen GoCD-palvelinta hyväksymistestausjärjestelmän kanssa. Testitapauksien priorisointimenetelmä kehitettiin itsenäisesti käyttäen matemaattista painotettua verkkoa. Painotetun verkon avulla tehtävään priorisointiin liittyivät tärkeimpinä priorisointiin vaikuttavat muuttujat, painofunktiot ja verkon karsimiseksi tehtävät leikkaukset. Painotetun verkon avulla tehtävä priorisointi todettiin toimivaksi esimerkin ja aidosta sovelluksesta tehdyn yksinkertaistetun mallin avulla.

Edellä mainitut diplomityön tavoitteet ovat nyt saavutettu ja sen tekeminen myös loi tarvittavan perustan WordDivellä tarvittavan web-sovelluksen hyväksymistestauksen testiautomaation rakentamiseen priorisointimenetelmää hyödyntäen. Lopuksi vielä toivon, että tässä diplomityössä esitetystä hyväksymistestauksen testiautomaation lähestymistavasta ja erityisesti sen priorisointia varten kehitetystä priorisointimenetelmästä olisi mahdollisimman paljon hyötyä sen käyttämistä harkitseville tai käyttäville tahoille.

LÄHTEET

[1] *web app complexity*.
[2] *testing possibility*.
[3] *worddive details*.
[4] *design science history*.
[5] *iso quality attributes*.
[6] *testing levels 1*.
[7] *testing levels 2*.
[8] *waterfall to agile*.
[9] *tdd popularity*.
[10] *istqb acceptance testing 1*.
[11] *istqb acceptance testing 2*.
[12] *traditional acceptance testing*.
[13] *robot framework info*.
[14] *robot framework architecture*.
[15] *selenium info 1*.
[16] *selenium info 2*.
[17] *docker vs virtual machine*.
[18] *gocd info*.
[19] *test case goals*.
[20] *robot framework good test cases*.
[21] *graph theory history*.
[22] *graph theory concepts 1*.
[23] *graph theory concepts 2*.
[24] *graph theory concepts 3*.

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
```