Tampereen yliopisto

Jukka Pajarinen

WEB-KÄYTTÖLIITTYMÄN HYVÄKSYMISTESTAUKSEN PRIORISOINTI PAINOTETUN VERKON AVULLA

Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Joulukuu 2019

TIIVISTELMÄ

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

Avainsanat: hyväksymistestaus, painotettu verkko, priorisointi, jatkuva integraatio, testiautomaatio

Tämän julkaisun alkuperäisyys on tarkastettu Turnitin OriginalityCheck -ohjelmalla.

ABSTRACT

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

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

ALKUSANAT

Tampereella, 31. joulukuuta 2019

Jukka Pajarinen

SISÄLLYSLUETTELO

KUVALUETTELO

TAULUKKOLUETTELO

LYHENTEET JA MERKINNÄT

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

1 JOHDANTO

Tässä luvussa esitetään lyhyesti työn keskeisin sisältö ja rakenne. Lisäksi pohditaan miksi työ on tarpeellinen ja miksi testitapauksien priorisoiminen on ongelmallista.

2 TUTKIMUSASETELMA

Tässä luvussa esitetään diplomityön taustaa, tutkimuskysymykset, käytetty tutkimusmenetelmä, tutkimuksen rajaus sekä tavoitteet. Tutkimuskysymykset liittyvät vahvasti yhteiseen priorisoinnin teemaan, johon tässä työssä erityisesti paneudutaan. Tutkimus on soveltavaa ja sen tarkoituksena on muodostaa selvitys tutkimusongelman ratkaisemiseksi. Tässä työssä se tarkoittaa erityisesti matemaattisen, toistettavissa olevan menetelmän kehittämistä tutkimusongelman ratkaisemiseksi. Tutkimuskysymyksistä itsessään voi päätellä tutkimuksen tarkoitusta ja tavoitteita, mutta tämä esitetään myös yksityiskohtaisemmin tavoitteet luvussa 2.5. Lisäksi työn lopussa on myös toteutuksellinen osuus, joka on tehty diplomityön asiakasyrityksen tarpeita varten. Toteutuksellisessa osuudessa on paljon muutakin sisältöä, joka on varsinaisen priorisointiteeman ulkopuolella, mutta pysyy kuitenkin työn kokonaiskontekstissa.

2.1 Tausta

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

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

Uusien ominaisuuksien ja koodimuutoksien tekemisen yhteydessä oli jatkuvasti tarve huolelliselle testaamiselle ja erityisesti asiakkaan näkökulmasta tärkeimpien sovelluksen ominaisuuksien toiminnan varmistamiselle. Tämä sai diplomityön aiheen suuntautumaan testiautomaatioon ja erityisesti hyväksymistestaukseen. Lisäksi yrityksessä oli jo toteutettuna päivittäisessä käytössä oleva hyväksymistestaus mobiilialustalle, joka auttoi hah-

mottamaan 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. Web-sovelluksen testiautomaatio toteutettiin käyttäen pääpiirteittäin samoja sovelluskehyksiä ja työkaluja **??** kuin mobiilisovellukselle. Tästä syystä diplomityön aihetta ja tutkimusongelmaa lähdettiin etsimään muualta.

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

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

2.2 Tutkimuskysymykset

Tutkimuksen tarkoituksena on pohjimmiltaan tarkoitus löytää ja kehittää toistettavissa oleva menetelmä hyväksymistestauksen testitapauksien priorisoimiseen. Testitapauksien laatimisen yleisenä ongelmanakohtana 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 jatkuvan integraation ja testiautomaation kanssa?*

2.3 Tutkimusmenetelmä

Tutkimuskysymyksiin vastaamiseksi työn tutkimusmenetelmäksi valittiin tietotekniikan diplomitöissä yleisesti käytetty Design Science menetelmä. Menetelmän tarkoituksena on tuottaa teknologiaa hyödyntävä ratkaisu tutkimusongelmaan vastaamiseen. Design Science menetelmää käyttäessä pyritään tutkimaan uusia 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.

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

Seuraavaksi kartoitettiin ratkaisuvaihtoehto tutkimuskysymyksiin ja työn yleiseen priorisoinnin teemaan vastaamiseksi ja esitetään perustelut matemaattiseen toistettavissa olevaan ratkaisumenetelmään päätymiseen. Tutkimusta varten kerättiin teoreettista aineistoa, jonka tarkoituksena oli tukea menetelmän kehittämistä ja jonka avulla pyrittiin luomaan lukijalle mahdollisimman vahva teoreettinen pohja tutkimuskysymyksiin vastaavan ratkaisumenetelmän ymmärtämiseksi. Asiakasyrityksen ohjelmistotuotetta ja ohjelmistotuotantoprosessi huomioiden toteutettiin myös kokonaisratkaisu jossa toteutettiin testiautomaatio hyödyntäen kehitettyä priorisoinnin ratkaisumenetelmää.

Lopuksi vielä evaluoitiin menetelmän eli ratkaisun ja sitä hyödyntävän toteutuksen toimivuus käytännössä 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 Robot Framework alustalle. Tämä rajaus pohjautuu asiakasyrityksessä jo aiemmin valittuihin testauksen sovelluskehyksiin ja työkaluihin. Lisäksi Robot Framework on alustana yleisesti käytetty etenkin hyväksymistestauksen toteuttamiseen. Robot Framework on myös erityisen hyvin soveltuva tilanteissa, joissa halutaan koodia korkeampaa abstraktiotasoa.

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

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

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

Painotetun verkon lyhimmän polun etsimiseen on olemassa lukuisa määrä erilaisia algoritmeja, mutta tässä työssä hyödynnetään vain perinteistä Djikstran 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, joka tekee siitä ihanteellisen tässä työssä muodostettavaan painotettuun verkkoon tehtävien leikkauksien identifoimista ajatellen.

2.5 Tavoitteet

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

Primääritavoitteen lisäksi valmiin diplomityön tavoitteena on myös tarjota selkeä, eheä ja helposti ymmärrettävä kokonaisuus hyväksymistestauksen testitapauksien priorisoimiseen, työssä kehitettyä menetelmää käyttäen. Kehitetty priorisointimenetelmä pyritään esittämään siten, että valmiista diplomityöstä olisi mahdollisiman paljon hyötyä sen käyt-

tämistä harkitseville tai käyttäville tahoille.

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

3 TESTIAUTOMAATIO

Tässä luvussa esitetään perusteet ja tarvittavat tiedot ohjelmistojen testauksesta ja testiautomaatiosta, jotka liittyvät työn laajempaan teoreettiseen kehykseen. Ensin esitetään testiautomaation tarkoitus, jonka jälkeen käydään yksityiskohtaisesti läpi ohjelmistotestauksen tasot ja pohditaan niiden merkitystä testiautomaatiossa. Lopuksi vielä esitetään tarvittavia jatkuvan integroinnin ja testausvetoisen kehityksen perusteita sekä pyritään luomaan lukijalle ymmärrystä siitä miten ne liittyvät niitä laajempaan testiautomaation käsitteeseen. Testiautomaation perusteiden ymmärtämistä tarvitaan varsinkin työn myöhemmässä vaiheessa, jossa esitetään testiautomaatioon liittyvien testitapauksien varsinainen priorisointi painotetun verkon avulla.

3.1 Testiautomaation tarkoitus

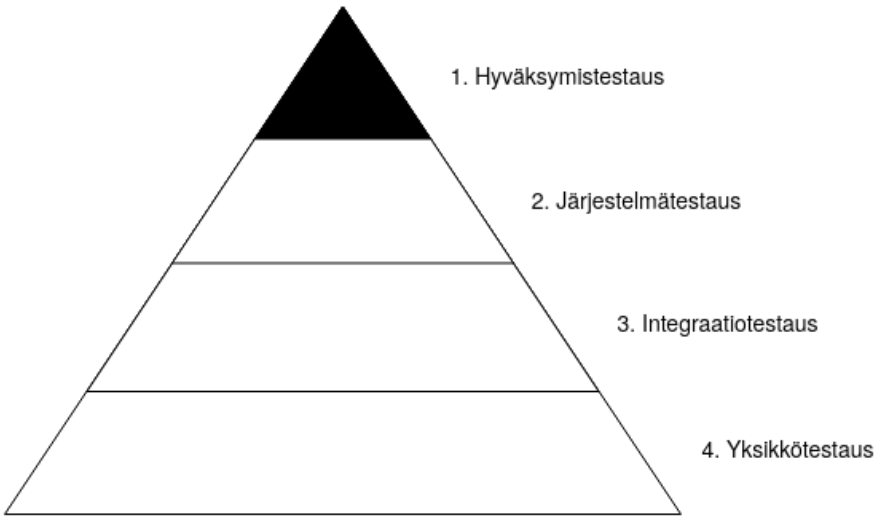
Testiautomaation tarkoitus on pohjimmiltaan mahdollistaa ohjelmistotuotteen jatkuva, tehokas ja vaivaton laadunvarmistus, nyt ja tulevaisuudessa. Testiautomaation vastakohtana voidaan ajatella manuaalista testausta, joka vaatii täydellistä ihmisen interaktiota testauksen suorittamiseen. Testiautomaatiossa käytetään erityisiä ohjelmistotyökaluja ennalta määritettyjen testitapauksien suorittamiseen, ihmisen tekemän manuaalisen testauksen sijaan. Ohjelmistojen testaamisella itsessään pyritään löytämään ohjelmistotuotteesta virheitä, anomalioita ja 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 (UAT).

Laadunvarmistuksen osalta ohjelmistokehityksessä on usein käytetty niin sanottuja laadullisia ominaisuuksia, joiden kattamisella voidaan validoida laatua. Laadullisia ominaisuuksia ovat ISO 9126-standardin mukaan: toiminnallisuus, luotettavuus, käytettävyys, tehokkuus, ylläpidettävyys ja siirrettävyys (ISO:9126-1 2001). 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 sub-

jektiivisia. Tässä diplomityössä testiautomaation yhteydessä keskitytään 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, 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 kasvaa. 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 käyttäjien hyväksyttämistestausta (UAT), jotta järjestelmätestauksen ja hyväksymistestauksen väliset eroavaisuudet tulevat lukijalle selkeästi esille.

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

3.2.1 Yksikkötestaus

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

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

Yksikkötestausta hyödynnetään usein myös ketterien menetelmien aihepiirissä, jossa ohjelmistotuotantoa voidaan toteuttaa muun muassa niin sanotulla testausvetoisella kehityksellä 3.4. Testausvetoisessa kehityksessä yksikkötestauksen osalta, ohjelmistokehittäjät laativat ensisijaisesti yksiköiden yksikkötestit ennen niiden toteuttamisen aloittamista. Ohjelmistotestauksen tasojen pyramidissa ja hyvin toteutetussa ohjelmistotestauksen monitasoisessa testauksessa tämä testauksen taso on usein kaikista laajin. Monitasoisessa testauksessa yksikkötestaus luo tärkeän pohjan testaamiselle kokonaisuutena ja antaa tietoa ohjelmiston pienimpien yksiköiden toimivuudesta. Yksikkötestaus on myös paljon käytetty ja tärkeä osa testiautomaatiossa, sillä se varmistaa sovelluksen yksiköiden suunniteltua toimintaa.

3.2.2 Integraatiotestaus

Integraatiotestauksen ajatuksena on testata ohjelmistotuotteen toteuttavien eri komponenttien yhteentoimivuutta niiden rajapintojen osalta. Integraatiotestaus toteutetaan ohjelmiston suunnitelmaa ja mallia vastaan. Integraatiotestauksen onnistuminen luo validoitavan perustan ohjelmiston toimimiseen ja sen koostamiseen kokonaisena, eri kom-

ponenteista koostuvana järjestelmänä. Integraatiotestausta tarvitaan, jotta voidaan varmistaa sovelluksen yksiköiden yhteentoimivuus, joka ei pelkällä yksikkötestauksella tulisi muuten katetuksi.

Integraatiotestauksen kohteita voivat olla esimerkiksi luokkien ja modulien 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 merkittävällä tavalla erilainen. Integraatiotestauksessa testitapauksien rakenteeseen tulee assert-funktioiden lisäksi myös tarvetta jäljitellä (englanniksi: mocking) rajapintojen tarjoamaa dataa. Rajapintojen 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 yhteentoimivuus. Integraatiotestaus on myös tärkeä osa testiautomaatiota, sillä sen avulla voidaan varmistaa sovelluksen yksiköiden, kuten esimerkiksi luokkien, komponettien tai modulien yhteentoimivuus.

3.2.3 Järjestelmätestaus

Järjestelmätestauksen ajatuksena on testata kokonaista ja toimivaa järjestelmää, yhtenä suurena yksikkönä. Järjestelmätestaus toteutetaan usein eräänlaisena tulikokeena, erityisesti ohjelmiston vaatimuksia vastaan. 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 (ISO:9126-1 2001).

<TODO: kirjoita tämä kappale paremmin...>

Aiemmin testiautomaation tarkoitus kappaleessa 3.1 esitettiin, edellä mainituista laadullisista ominaisuuksista kaikki eivät sovellu hyvin testiautomaation avulla testattaviksi. Esitetyistä syistä johtuen, automatisoidulla järjestelmätestauksella voidaan testata edellä mainituista ominaisuuksista lähinnä ohjelmiston toiminnallisuutta, luotettavuutta ja tehokkuutta. Sen myötä testauksen tasona se voi olla testiautomaation teknisen toteutuksen kannalta jopa hyvin samanlainen kuin sitä spesifimpi hyväksymistestaus. Usein kuitenkin hyväksymistestauksessa paneudutaan erityisesti vaatimusmäärittelyyn ja asiakaslähtöiseen testaamiseen, kun taas järjestelmätestauksessa voidaan testata esimerkiksi myös järjestelmän tehokkuutta tai tietoturvaa. Tämä on tosin täysin riippuvainen vaatimusmäärittelystä, ja 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 muistuttavat kovasti toisiaan esimerkiksi edellä maini-

tulla tavalla.

Järjestelmätestaus, osittain hyväksymistestauksen kanssa on erittäin merkittävä osa testiautomaatiosta, sillä sen avulla voidaan varmistaa kokonaisen järjestelmän vaadittu toiminnallisuus.

3.2.4 Hyväksymistestaus

Hyväksymistestauksen ajatuksena on varmistaa toteutettavan ohjelmiston vaatimusten toimivuus erityisesti käytännön tilanteissa siten, että voidaan varmistaa vastaako ohjelmisto loppukäyttäjän tarpeita. Hyväksymistestaus toteutaan ohjelmiston toimintoja kuvaavaa vaatimusmäärittelyä ja loppukäyttäjistä sekä heidän tarpeista laadittuja käyttötapauksia vastaan. Samassa asiayhteydessä puhutaan usein myös niin sanotusta päästä päähän testauksesta (englanniksi: e2e, end-to-end). Päästä päähän testauksessa on tarkoituksena toteuttaa testaaminen siten, että polkuina ajateltuna, se sisältää kaiken siltä väliltä mitä loppukäyttäjä voi tarpeidensa saavuttamiseksi tehdä ja nähdä aloittaessaan ohjelmiston käytön ja lopettaessaan sen käytön. Testiautomaatio on erittäin hyödyllinen myös hyväksymistestauksen osalla, koska sillä voidaan automatisoida ohjelmiston validointi ja hyväksyminen, sekä estää puutteellisesti toimivan ohjelmiston julkaiseminen. Hyväksymistestausta tarvitaan myös, jotta voidaan testata ja validoida vaatimusten mukaisten ominaisuuksien toimivuus.

<TODO: kirjoita tämä kappale paremmin...>

Edellisessä kappaleessa käytiin jo hieman läpi järjestelmätestauksen ja hyväksymistestauksen samankaltaisuutta. Tässä on asiaa toisinpäin tarkasteltuna, osittainen lista ohjelmistotuotannossa havaittavista järjestelmätestauksen ja hyväksymistestauksen eroista, järjestelmätestauksen näkökulmasta:

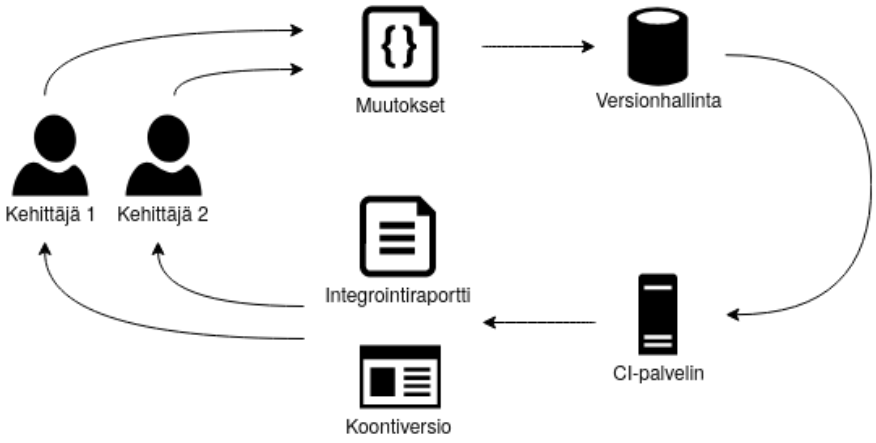
- Painoarvo vaatimusmäärittelyssä
- Ohjelmistokehittäjien ja testaajien lisäksi myös sidosryhmät ja asiakkaat
- Testaus on vain lähinnä toiminnallista
- Testauksen syötteet tulevat suoraan käyttäjältä
- Testaus keskittyy arvioimaan täyttääkö ohjelmisto käyttäjän tarpeet
- A/B testaamisen mahdollisuus
- Hyväksymistestaus tapahtuu järjestelmätestauksen jälkeen
- Virheet käsitellään epäonnistumisina

Hyväksymistestaus, osittain järjestelmätestauksen kanssa on äärimmäisen hyödyllinen osa testiautomaatiosta, sillä sen avulla voidaan varmistaa kokonaisen järjestelmän toiminnallisuus ja verifioida, että se vastaa vaatimusmäärittelyä. Hyväksymistestauksen rooli testiautomaatiossa ja erityisesti jatkuvan integraation yhteydessä on indikoida voidaanko järjestelmä sellaisenaan julkaista loppukäyttäjille.

3.3 Jatkuva integrointi

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

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



Kuva 3.2. *Jatkuvan integroinnin perusperiaate on iteratiivinen*

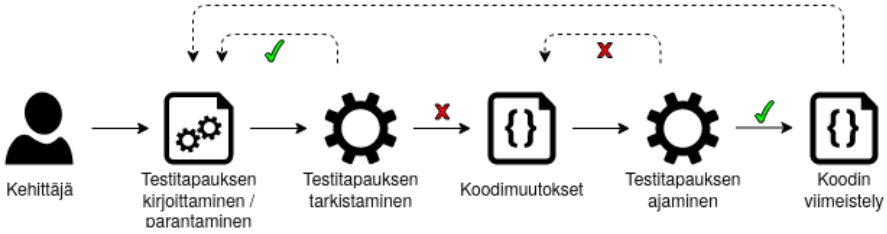
Esimerkkinä versionhallintajärjestelmänä voidaan käyttää nykyaikana suosittua git versionhallintaohjelmistoa ja jatkuvan integroinnin palvelimena esimerkiksi GoCD ohjelmistoa. Perusideana jatkuvassa integraatiossa on konfiguroida jatkuvan integraation mahdollistava ohjelmisto siten, että se kuuntelee versionhallintaan tulevia muutoksia ja suorittaa integrointiprosessin jatkuvasti sellaisia huomattuaan. Versionhallintaan tulevat muutokset voidaan jatkuvan integraation osalta kuunnella ajastetusti tietyin väliajoin tai oikeasti

jatkuvasti käyttämällä esimerkiksi web-koukkuja, jotka tiedottavat jatkuvan integraation palvelimelle versionhallintaan saapuneista muutoksista. Jatkuvan integroinnin yhden iteraatiokerran integrointiprosessin tuloksena on tarkoituksena tarjota periaatteeltaan sama lopputulema kuin mitä se olisi manuaalisella integrointiprosessillakin. Jatkuva integroinnin mahdollistava konfiguraatio sisältää jonkinlaisen koontiputken tai koontiputkia, joissa rakennetaan koontiversio kehitettävän ohjelmiston lähdekoodeista. Koontiputki voi sisältää esimerkiksi ohjelman lähdekoodien kääntämisen asiaan sopivalla kääntäjällä. Kääntämisen lisäksi koontiputkeen on tässä vaiheessa erittäin kannattavaa yhdistää testiautomaatiota, kuten esimerkiksi automaattiset yksikkötestit ennen kääntämistä ja hyväksymistestit kääntämisen jälkeen.

Jatkuvan integroinnin yhteydessä suoritettavat testikokoelmat ja niiden sisältävät testitapaukset ovat erittäin järkevää 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 ohjelmistokehitetyksessä tehty esimerkiksi viikoittain, niin jatkuva integroiminen korjaa sen tuomat haasteet turhan laajasta integrointiprosessista ja mahdollisesta ohjelmistokoodin hajoamisesta. Tällaisissa tapauksissa ohjelmistokoodi voi sisältää epäyhteensopivia moduleita tai muita rajapintoja sekä mahdollisuuden käännettävien lähdekoodien kääntämisen onnistumisesta.

3.4 Testausvetoinen kehitys

Perinteisesti testiautomaatio on soveltunut hyvin vain vakaille ohjelmistoille ja niiden regressiotestaamiseen. Nykypäivänä ohjelmistokehitys on siirtynyt suunnitelmapohjaisista prosesseista iteroiviin ketteriin ohjelmistotuotannon prosesseihin. Näihin testiautomaatio on soveltunut huonosti, kun testattavaa ohjelmistoa tai lisättyä toiminnallisuutta ei ole vielä olemassa. Tähän ongelmaan on 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 3.3 alkavat testitapauksien luomisesta ja niiden tarkastamisesta. Tarkastaminen tapahtuu siten, että testitapaukset ajetaan oletuksella, että niiden täytyy tässä vaiheessa epäonnistua. Alkuvaiheen testitapauksien luomisen jälkeen ohjelmistkehittäjät 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 ohjelmisttuotannon 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 ohjelmistkehitys on siis iteratiivista ja näin ollen testitapauksien suorittamisesta saadaan palautetta ja raportointia koko ohjelmistotuotantoprosessin aikana. Testausvetoinen kehitys kuuluu ohjelmistotuotannossa vahvasti ketterien menetelmien alaisuuteen ja on kasvattanut suosiotaan ketterien menetelmien mukana.

4 HYVÄKSYMISTESTAUS

Tässä luvussa esitetään perusteet ja tarvittavat tiedot hyväksymistestauksesta, johon testauksen tasoista tässä diplomityössä keskitytään. Ensin esitetään hyväksymistestauksen tarkoitus, jonka jälkeen keskitytään hyväksymistestausvetoiseen kehitykseen ja sen esittelemiseen ohjelmistotuotannollisena menetelmänä. Hyväksymistestausvetoisen kehityksen jälkeen käydään läpi tässä diplomityössä käytettyä ja lähes de facto testialustaa, Robot frameworkia, hyväksymistestauksen testitapauksien rakentamiseen. Robot frameworkin perusteiden jälkeen esitetään hyväksymistestauksen testitapauksien laatiminen käyttäen Robot frameworkia sekä esitetään web-sovelluksien erityispiirteitä jotka on huomioitava hyväksymistestauksessa. Hyväksymistestaksen perusteiden ymmärtämistä tarvitaan työn toteutusvaiheessa, jossa esitetään korkealla tasolla asiakasyritykselle toteutettua hyväksymistestausta ja sen automaatiota. Lopuksi esitetään yleisestikin ottaen testitapauksiin tärkeästi liittyvä priorisointiongelma ja käydään läpi sen eri ratkaisumalleja, keskittyen erityisesti tässä diplomityössä myöhemmin esitettävään priorisointiin painotetun verkon avulla.

4.1 Hyväksymistestauksen tarkoitus

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 käyttäjän, asikkaiden tai muun autorisoidun tahon päättää hyväksytäänkö järjestelmä (*ISTQB Glossary* 2019). Ohjelmistotestauksen tekniikoiden näkökulmasta hyväksymistestaus on mustalaatikkotestausta, eli sitä testataan tietämättä sen teknisestä toteutuksesta. Hyväksymistestauksen painoarvo on asiakaperusteisessa 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 järjestlemää testataan siten kuin loppukäyttäjä sitä käyttää. Hyväksymistestauksessa ei anneta suurta painoarvoa kosmeettisille tai kirjoitusvirheille, vaan pyritään selvittämään loppukäyttäjille

oleellisten ja tarpeellisten toimintojen toteutuminen.

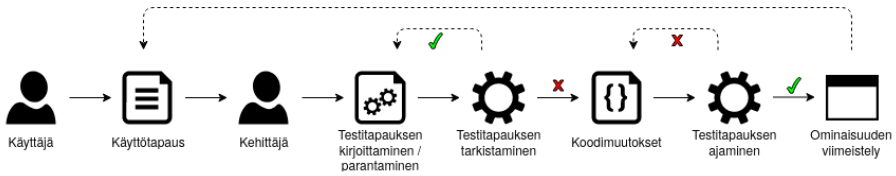
Hyväksymistestaus on aiemmin esitetyistä testauksen tasoista 3.2 viimeinen ja sen suorittamisen jälkeen saadaan tieto siitä onko järjestlemä toteutuksen osalta sellaisenaan valmis julkaistavaksi. Perinteisesti hyväksymistestauksen lähtökohtia ovat selvät hyväksymisvaatimukset sekä julkaisukelpoinen toteutus joka voi sisältää vain kosmeettisia virheitä. Hyväksymisvaatimukset voivat olla esimerkiksi liiketoiminnallisia käyttötapauksia, prosessivirtauskaavioita sekä ohjelmiston vaatimusmäärittely. Testiautomaatiota varten käytettävästä testialustasta riippuen hyväksymistestauksen käyttötapaukset voidaan muodostaa joko osittain tai suoraan testitapauksiksi. Hyväksymistestaukseen usein osallistuu ohjelmistokehittäjien lisäksi myös muut sidosryhmät ja loppukäyttäjät. 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 antaa mahdollisuuden korjata usein liiketoiminnalisestakin näkökulmasta merkittävät toiminalliset virheet ennen järjestelmän julkaisua loppukäyttäjille.

Kehittäjien käsitys järjestelmän toiminnallisuudesta ja sen vaatimuksista voi olla usein hyvinkin erilainen kuin loppukäyttäjien. Hyväksymistestauksen avulla voidaan tätä lievittää tätä ongelmaa, ja saattaa ohjelmistokehittäjät loppukäyttäjien kanssa vaatimusmäärittelyn suhteen samalle sivulle. Testiautomaation avulla toteutettavalla toistuvalla hyväksymistestauksella varmistetaan, että järjestelmä toteuttaa loppukäyttäjän tarpeet vielä järjestlemään tehtyjen muutoksien jälkeenkin. Hyväksysmistestauksen testitapaukset tarkoituksenmukaisesti heijastavat suoraan loppykäyttäjien tarpeita, joka on iso etu sillä sen avulla ohjelmistokehittäjät ja muut sidosryhmät voivat tehokkaasti varmistaa järjestlemän valmiuden ja tilan. Hyväksymistestauksella siis saadaan katsaus ohjelmiston valmiudesta sen vaatimuksiin ja loppukäyttäjien toiminnallisiin tarpeisiin nähden.

4.2 Hyväksymistestausvetoinen kehitys

Hyväksymistestausvetoisen kehityksen (englanniksi: ATDD, acceptance test driven development) tarkoituksena, kuten testausvetoisessakin kehityksessä 3.4 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 (*ISTQB Glossary* 2019). Tarvittavat ohjelmiston hyväksymistestit suoritetaan iteratiivisesti ohjelmistokehitysprosessin aikana ja se tarkoittaa käytännössä jatkuvan integraation 3.3 ottamista käyttöön ohjelmistokehityksessä. Hyväksysmistestausvetoinen kehitys on erittäin hyödyllinen ohjelmistokehityksessä käy-

tetty menetelmä, sillä kehitysvaiheessa on aina tarkasti tiedossa vastaako ohjelmiston tila asiakasvaatimuksia ja kuinka hyvin se niiden täyttämisessä onnistuu.



Kuva 4.1. *Hyväksymistestausvetoisen kehityksen vaiheet*

Hyväksymistestausvetoinen kehitys voidaan luokitella ketteräksi ohjelmistokehitysmenetelmäksi, kuten sen yläkäsitteenä oleva testausvetoinen kehityskin 3.4. Hyväksymistestausvetoinen kehitys on testausvetoisen kehityksen kanssa perusperiaatteeltaan samanlainen, mutta ennen ohjelmistokehityksen aloitusta asiakasvaatimukset kartoitetaan ja ohjelmiston hyväksyttävyys määritetään. Hyväksymistestitapaukset kirjoitetaan testausvetoisen kehityksen mukaisesti ensin ja ohjelmistokehitys itsessään noudattaa iteratiivisesti testausvetoista kehitystä, vaikkakin hyväksymistestaus itsessään on perinteisesti vaatinut lähes valmista järjestelmää. Asiakasvaatimukset määritetään usein käyttötapauksien muotoon, ja riippuen testialustasta ne voidaan kirjottaa testitapauksien muotoon niitä vahvasti hyödyntäen. Hyväksymistestausvetoisessa kehityksessä ohjelmistokehitystä siis ohjaavat asiakasvaatimukset ja loppukäyttäjien tarpeiden toteutuminen, jotka ovat hyvin usein toiminallisia vaatimuksia. Hyväksymistestausvetoisessa kehityksessä mitataan jatkuvasti iteroiden 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 on 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 muuttuminen varmistaminen, kuten tason läpäiseminen pelisovelluksessa, joka muuttaa käyttäjän edistystä. Hyväksymistestausvetoisen kehityksen tarkoituksena menetelmänä on onnistua vastaamaan loppykäyttäjän 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 ominaisuuksien toteutumisen validoimiselle menetelmän jokaisen iteraation koontiversiossa.

4.3 Web-sovelluksien erityispiirteet

Web-sovelluksilla on omia erityispiirteitä, jotka vaikuttavat testitapauksien laatimiseen. Nykypäivänä web-sovellukset ovat kasvaneet kompleksisuudessa ja front-end puolen toteutuksesta tarkasteltuna web-sovellukset usein muistuttavat jo perinteisiä työpöytäsovelluksia. Web-sovelluksia päivitetään nykyään tiheään tahtiin ja 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 dokumenttiobjektimalliin, DOM. Dokumenttiobjektimallin avulla verkkoselaimet renderöivät käyttöliittymän ja siinä näkyvän sisällön. Tämän lisäksi dokumenttiobjektimalli mahdollistaa käyttöliittymässä olevien elementtien valitsemisen, jota hyödynnetään myös testitapauksien kirjoittamisessa.

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

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

Erilaisia web-sovelluksen loppukäyttäjien asiakasympäristöjä on huikean paljon, joka kannustaa moniselaimellisen testauksen rakentamiseen. Näissä ympäristöissä on omat verkkoselaimensa, näyttöresoluutio ja selainasetukset, jotka saavat saman 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 yleisestikkin ottaen käyttöliittymien testaaminen on perinteisesti tapahtunut manuaalisesti. Nykyään web-sovelluksia voidaan testata niin sanotun päätteettömän testauksen keinoin. Web-sovelluksien päätteettömässä testauksessa verkkoselaimen, näyttöresoluution ja selainasetuksien muodostamaa asiakasympäristö rakennetaan virtualisoinnin avulla. Virtualisoinnista vastaa joko verkkoselain itse tai voidaan käyttää UNIX järjestelmissä x-näyttöpalvelimen protokollan toteuttavaa virtualisointiratkaisua, esimerkiksi Xvfb. Virtualisoitu asiakasympäristö rakennetaan siten, että se päätteettömänä vastaa täysin päätteellistä vaihtoehtoa, ja siitä voi-

daan ottaa esimerkiksi kuvankaappaus vaikka mitään ihmisen astittavaa ei ole näkyvillä.

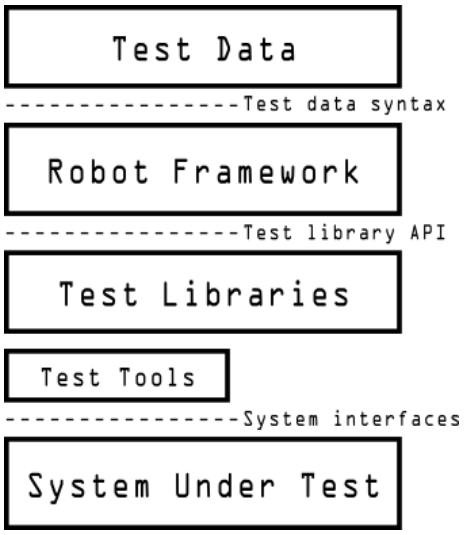
4.4 Hyväksymistestauksen työkaluja

Tässä kappaleessa esitetään diplomityötä tehdessä käytettyjä ja osin myös varsin yleisiä testiautomaation mahdollistavia työkaluja, kukin omassa alikappaleessaan. Ensin esitetään hyväksymistestauksen automatisoimisen kannalta kaikista tärkeimmät työkalut, eli testikehyksenä käytettävä Robot framework ja web-sovelluksien interaktioiden automatisoimisen mahdollistava Selenium kirjasto. Lisäksi esitetään kolme muuta tärkeää työkalua joiden avulla voidaan rakentaa kokonainen ohjelmistotuotannon prosessiin intergroitavissa oleva hyväksymistestausjärjestelmä. Toteutuksessa GoCD vastaa jatkuvan integroinnin tarjoamisesta, Xvfb vastaa päätteettömän testauksen tarjoamisesta ja Docker vastaa työkalujen virtualisoimisesta ja säiliöinnistä, jolloin työkaluista saadaan luotua yhtenäinen kokonaisuus.

4.4.1 Robot Framework

Robot framework on geneerinen avoimen lähdekoodin testausalusta hyväksymistestaukseen, hyväksymistestausvetoiseen kehitykseen ja robotisten prosessien automaatioon (*Robot Framework User Guide* ei julkaisupäivää). Testialustana Robot frameworkilla on helposti ymmärrettävä, luettava ja selkeä avainsanaperustainen syntaksi. Testialustan etuna on helppo lähestyttävyys ja sen pohjalle rakennettujen testitapauksien ymmärtäminen ei vaadi ohjelmakoodin ymmärtämistä. Robot framework on Python-perustainen testialusta joka on helppo asentaa, sitä on helppoa ymmärtää, sillä on kattava dokumentaatio ja se on helppoa ottaa käyttöön.

Robot frameworkissa on sisäänrakennettu tuki ulkoisille kirjastoille ja dokumentaatiosta löytyy paljon tietoa omien avainsanojen ja omien kirjastojen tekemiseen. Lisäksi Robot framework on todella suosittu ja sisäänrakennettujen ominaisuuksien lisäksi ulkoisia kolmansien osapuolien kirjastoja löytyy alustalle paljon. Robot framework tukee muuttujien käyttöä testitapauksien rakentamisessa, joilla voi lisätä hieman kompleksisuutta ja logiikkaa omiin testitapauksiin. Robot frameworkista 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*

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

Yhtenä heikkoutena Robot frameworkissa on tuen puuttuminen ohjelmistokieliperustaisissa testikehyksissä löytyville kontrollirakenteille, joita esiintyy esimerkiksi yksikkötestaukseen tarkoitetuissa testikehyksissä. 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 Apache 2.0 lisenssoitu työkalu ja kirjastokokelma verkkoselainten automatisoimiseen. Ensisijaisesti se on tarkoitettu web-sovelluksien automatisoimiseen testaustarpeita varten, mutta se ei ole rajattu vain siihen (**noauthor_selenium_no**
Erityisen hyvin Selenium soveltuu hyväksymistestauksen testiautomaation rakentamiseen, sillä sen avulla automatisoidaan web-sovelluksien käyttöliittymissä tehtäviä toimenpiteitä. Selenium on ThoughtWorks yhtiön kehittämä verkkoselainten automatisoimiseen tarkoitettu työkalujen ja kirjastojen kokoelma ja se on saatavilla Windows, Linux ja MacOS alustoille. Sama yhtiö on toteuttanut myös tässä diplomityössä myöhemmin esitettävän GoCD ohjelmiston, jota voidaan käyttää jatkuvan integroimisen ja julkaiseminen rakentamiseen 4.4.5.

Selenium tuoteperheeseen kuuluvat Selenium WebDriver, Selenium IDE ja Selenium Grid komponentit. Selenium WebDriver on varsinainen web-sovelluksien automatioimiseen käytettävä ohjelmisto, jota myös tässä diplomityössä Selenium tuotteista käytetään.

Selenium IDE on kehitysympäristö ohjelmistokehittäjille ja testaajille, jota voidaan halutessaan käyttää testitapauksien laatimiseen. Selenium Grid on järjestemä, jonka avulla voidaan Selenium pohjaisten testitapauksien suorittaminen skaalautuvasti hajauttaa useille etäkoneille. Tässä diplomityössä ei ole käytetty Selenium Grid järjestelmää vaan testitapauksien suorittamiseen tarvittavat ohjelmistot on säiliöity Docker työkalua käyttäen, joka mahdollistaa tarvittaessa skaalautuvuuden 4.4.4.

Selenium on todella tärkeä osa web-sovelluksien testitautomaation 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 avulla sitä voidaan käyttää muistakin ohjelmointikielistä, kuten C#, JavaScript tai Python.

Tässä diplomityössä Selenium työkalua käytetään Robot Frameworkin yhteyteen integroituna ulkoisena kirjastona. Robot Frameworkille on saatavilla SeleniumLibrary niminen kirjasto, josta löytyy Robot Frameworkin 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 avulla näytön ja syöttölaitteiden käsittelyn. Xvfb ei tulosta mitään näytölle, vaan kaikki näytölle normaalisti tulostuva graafisia käyttöliittymiä sisältävä sisältö on ajonaikaisessa muistissa. Xvfb toimii kuten tavallinenkin X-näyttöpalvelin, eli vastaa X-ohjelmien pyyntöihin ja hoitaa niihin liittyvän tapahtumien ja virheiden käsittelyn.

Xvfb soveltuu web-sovelluksien hyväksymistestauksen automatiointiin 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.

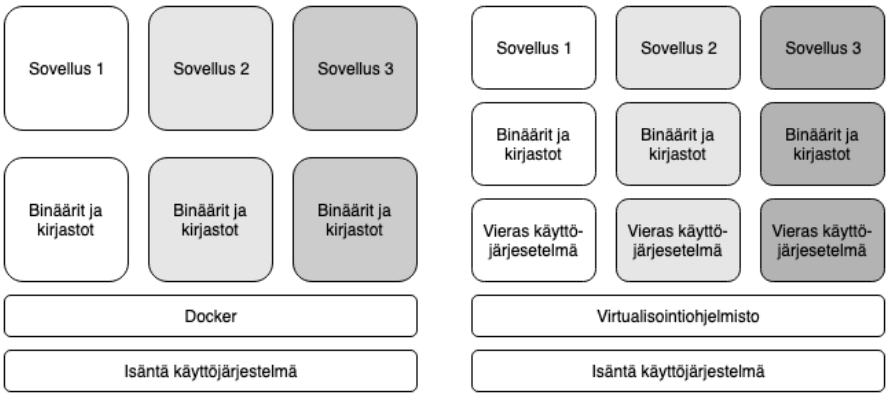
Yhtenä Xvfb heikkoutena on, että se on saatavilla vain UNIX pohjaisiin käyttöjärjestelmiin, kuten Linux ja MacOS. Näin ollen esimerkiksi Windows alustalla toimivaa Internet Explorer selainta ei voida natiivisti testata.

Robot Frameworkille on saatavilla XvfbRobot niminen kirjasto, jota tämän diplomityön toteutuksessa käytettiin. XvfbRobot on kirjasto, josta löytyy Robot Frameworkin syntaksin

mukaisesti määritellyt avainsanat Xvfb palvelimen kanssa kommunikoimiseen.

4.4.4 Docker

Docker on säiliöinti (englanniksi: containerization) työkalu, jonka avulla on mahdollista luoda, rakentaa ja ajaa säiliöiden muotoon konfiguroituja sovelluksia. Docker muistuttaa virtuaalikoneita, mutta se on kevyempi ja optimaalisempi, sillä se jakaa käyttöjärjestelmän saman ytimen eri säiliöiden kesken ja virtualisoi vain itse sovellusympäristön jonka säiliön konfiguraatio sisältää. Säiliöiden sisään voidaan paketoida kaikki kokonaisen sovelluksen tarvitsemat ohjelmistot, kirjastot, ympäristöt, riippuvuudet ja itse sovelluksen ohjelmakoodi. Säiliön rakentamalla ja käynnistämällä voidaan sitä käyttää konfiguraatioltaan samanalaisena eri ympäristöissä, joissa Docker ohjelmisto on saatavilla.



Kuva 4.3. *Dockerin ja virtuaalikoneen eroavaisuus*

Dockerfilen avulla voidaan luoda räätälöity säiliö, jota voidaan rakentaa yksi tai useampia instansseja. Räätälöidyn säiliön etuna on etenkin se, että sen avulla saadaan 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 konfiguratiion käyttämisen sovelluksen pystyttämiseen ja julkaisemiseen nopeasti sekä helposti eri lokaatioihin. Docker compose on tapa rakentaa docker-verkko, joka koostuu palveluista jotka ovat joko valmiiksi tehtyjä docker imageja tai itse Dockerfilen avulla tehtyjä imageja. Docker verkkoon voidaan myös lisätä yhteisiä tietosäilöjä, joita verkkoon kuuluvat palvelut voivat yhteishyödyntää. Yksittäisen säiliön konfiguraation sisältämä Dockerfile ja kokonaisen docker-verkon konfiguraation sisältämä docker-compose tiedosto kirjoitetaan yaml-kielellä.

Tässä diplomityössä Dockeria käytettiin hyväksymistestauksen testitapauksien automatisoimiseen tarvittavien työkalujen säiliöinnissä. Olemassa olevaan docker-verkkoon lisättiin hyväksymistestauksen testitapauksia varten tarkoitettu säiliö, joka hyödyntää Robot frameworkiä, Seleniumia, Xvfbää ja sisältää muun muassa testauksessa tarvittavat verkkoselaimet. Dockeria käyttämällä siis pystyttiin luomaan monistettava ja uniikki hy-

väksymistestauksen automatisointiympäristö, jota voidaan käyttää jatkuvan integraation yhteydessä testitapauksien suorittamiseen.

4.4.5 GoCD

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

Koonti-testaus-julkaisu prosessin voi rakentaa GoCD-palvelimen graafisen käyttöliittymän kautta tai koodina käyttäen yaml tai json syntaksia. Teknisesti GoCD palvelinohjelmisto koostuu itse palvelimesta ja agenteista, jotka voivat suorittaa palvelimen pyytämänä ennalta määritettyjä koonti-testaus-julkaisu prosessin tehtäviä. Agentit on tarkoituksenmukaista sijoittaa eri järjestelmään kuin missä itse palvelin sijaitsee ja agenteille voi määrittää resurssiominaisuuksia, jotka kertovat palvelimelle mitä tehtäviä agenteilla voi teettää. GoCD palvelinohjelmiston terminologia on hieman tavallisesta poikkeavaa ja erilainen esimerkiksi todella suositun Jenkins ohjelmiston vastaavista. Koonti-testaus-julkaisu prosessin ylin käsite GoCD terminologiassa on putkiryhmä (englanniksi: pipeline group). Putkiryhmän avulla yhteen kuuluvat putket (englanniksi: pipeline) voidaan järjestää samaan kokonaisuuteen. GoCD terminologiassa yksittäinen putki vastaa esimerkiksi koontivaihetta tai testausvaihetta. Yksittäisen putken alaisuudessa on vaiheita (englanniksi: stage), jotka antavat GoCD palvelimen käyttöliittymässä tiedon vaiheen onnistumisesta. Vaiheet itsessään sisältävät vielä tehtäviä (englanniksi: job), jotka ovat yksittäisiä komentoja tai suoritettavia tehtäviä, jotka agentit pystyvät käsittelemään. GoCD palvelimen terminologiaan kuuluvat vielä vahvasti artifaktit, jotka ovat tiedostoja mitä tehtävien suorittamisen yhteydessä syntyy ja jotka on merkitty säästettäväksi. Esimerkkejä artifakteista ovat 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 määrittäminen

Testitapaus on testiautomaation näkökulmasta, määritelty toimenpiteiden, ehtojen ja muuttujien joukko, joka suorittamalla voidaan verifioida ominaisuus tai toiminnallisuus ohjelmistosta. Testitapauksiin liittyy oleellisesti testikokoelman käsite, joka tarkoittaa samaan kontekstiin kuuluvista testitapauksista muodostettua joukkoa. Tässä diplomityössä keskitettyyn hyväksymistestaukseen liittyen testitapaukset kirjoitetaan usein käyttötapauksien

muodossa. Lisäksi hyväksymistestauksen priorisoimiseen painotetun verkon avulla on suositeltavaa suunnitella ja rakentaa testitapaukset näkymä ja siirtymäperusteisesti, jotta matemaattisia verkkoja voidaan kunnolla hyödyntää.

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

Robot Frameworkin perustaja on kirjoittanut laajan ohjeistuksen siitä, miten Robot Frameworkiä käyttäen luodaan hyviä testitapauksia (Klärck 2019). 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 frameworkiä 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 käyttöä suositellaan kapsuloimaan pitkiä ja kompleksisia arvoja, mutta arvojen syöttäminen ja palauttaminen muuttujia hyödyntäen tulisi pitää pois testitapauksien tasolta.

4.6 Priorisointiongelma

Testitapauksien priorisointi on kustannussyistä tai resurssien optimoinnin kannalta erittäin tärkeää. Ohjelmistotestauksessa on hyvä tiedostaa, että ohjelmistotuotetta ei usein voida testata täydellisesti, joka nostaa esiin tarpeen tärkeimpien testitapauksien löytämisestä. Priorisoinnin toutettamisen tärkeys korostuu erityisesti silloin kun kohdejärjestelmä on kompleksinen ja toimminallisia ominaisuuksia on paljon.

Priorisoinnista saatavia hyötyjä:

- Tärkeät ongelmat löydetään aikaisin
- Testitapauksien suorittamisen järjestäminen
- Epäoleelliset testitapaukset voidaan jättää toteuttamatta
- Kustannuksia ja resursseja säästyy
- Käytännönläheisyys
- Korkean prioriteetin testitapauksiin voidaan käyttää huolellista suunnittelua

<TODO: kirjoita tähän lisää tekstiä...>

Priorisointiongelmaan on olemassa useita erilaisia lähestymistapoja ja menetelmiä, kuten esimerkiksi heuristinen priorisointi tai MoSCoW menetelmä. Tässä diplomityössä priorisointiin käytetään kuitenkin matemaattista painotettuihin verkkoihin perustuvaa lähestys-

mistapaa, joka on uudenlainen tämän diplomityön tuotteena kehittynyt menetelmä priorisointiongelman ratkaisemiseen.

5 VERKKOTEORIA

Tässä luvussa käsitellään työhön keskeisesti kuuluvan verkkoteorian perusteet käydää huolellisesti läpi erityisesti työssä käytettävät osat. Työssä sovelletaan erityisesti verkkoteorian painotettua verkkoa sekä verkkoteoriassa esiintyvän lyhimmän polun ongelmaan kehitettyä Djikstran algoritmia. Verkkoteoria itsessään on osa diskeettiä matematiikkaa.

5.1 Matemaattisten verkkojen tarkoitus

Matemaattisten verkkojen tarkoituksena on mallintaan parittaisia riippuvuuksia verkkomaisessa objektijoukossa. Verkkoteoriassa peruskäsitteitä ovat itse *verkko* eli *graafi*, joka muodostuu *solmuista* ja niiden välisiä riippuvuuksia esittävistä *kaarista* tai *nuolista*. Verkkoterialla 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.

5.2 Perusmerkinnät ja käsitteet

Verkkoteoriassa käytetään seuraavia perusmerkintöjä:

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

Verkkojen solmujen välisiä yhteyksiä, eli kaaria esitetään usein myös yhteys- tai painomatriisina.

$$M_G = (a_{ij})_{3\times 3} = \begin{array}{c} G \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{pmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \end{pmatrix} \end{array}$$

Verkkoteoriassa käytetään myös muun muassa seuraavia käsitteitä:

- Solmun asteluku, $d_G(x)$, eli solmuun liittyvien *kaarten* määrä.
- Surkastunut verkko, $E = \varnothing$, eli verkko jossa ei ole *kaaria*.
- Täydellinen verkko, eli jokaista solmuparia $v_1 \neq v_2$ yhdistää ainakin yksi *kaari*.

- Aliverkko, $G_2 \subset G_1$, eli *verkko* G_2 joka koostuu osasta *verkon* G_1 *solmuja* ja *kaaria*.
- Verkon komplementti, G', eli sellainen *verkko*, jossa on kaikki ne *kaaret* joita *verkossa* G ei esiinny.
- Verkon yhtenäisyys, $v_1 \neq v_2, v_1 \rightarrow v_2$, eli jokaiselle solmuparille $v_1 \neq v_2$ on olemassa niitä yhdistävä *kaari*.
- Polku, $\{v_0, v_1, ..., v_n\}, v_0 \rightarrow v_n$, eli *suunnattu solmujono* jota pitkin voidaan kulkea *solmusta* v_0 *solmuun* v_n.
- Eristetty solmu, $d_G(v_1) = 0$, eli *solmu* jonka *asteluku* on nolla.
- Silta, $v_1 \rightarrow v_2, d_G(v_1) = 1 \vee d_G(v_2) = 1$, eli *kaari* johon yhdistyvän *solmun asteluku* on yksi ja jonka poistaminen epäyhteinäistää *verkon*.

5.3 Painotettu verkko ja leikkaaminen

- $\alpha := V(G), E(G) \rightarrow \mathbb{N}$ Painofunktion yleinen kuvaus.

5.4 Lyhimmän polun ongelma

- $d_G^\alpha(v_1, v_2) = min\{\alpha(P) | P : v_1 \rightarrow v_2 | v_1, v_2 \in V(G)\}$ Lyhimmän polun ongelma.
- Ongelman ratkaiseminen Djikstran algoritmilla

6 PRIORISOINTI PAINOTETUN VERKON AVULLA

Tässä luvussa esitetään tutkimuksen tärkein sisältö ja kokonaisuutena vastaus tutkimuskysymykseen *T1*, eli toistettavissa oleva menetelmä testitapauksien priorisoimiseen. Priorisointiin vaikuttavat muuttujat luvussa 6.1 esitetään myös suora vastaus tutkimuskysymykseen *T2*. Lisäksi painofunktiot 6.2 ja verkon karsiminen 6.4 esittää vastaukset tutkimuskysymykseen *T3*. Testitapauksien muodostaminen verkosta **??** antaa osittaisen vastauksen myös tutkimuskysymykseen *T4*.

Priorisointia varten esitetään harkintaa käyttäen lähdeaineistosta suodatetut priorisointiin vaikuttavat muuttujat, painofunktio, testitapauksien näkymäperusteinen koostaminen ja painotetun verkon laatiminen. Lisäksi menetelmää käyttäen tuotetun painotetun verkon sisältämää informaatiota käytetään prioriteeteiltaan tärkeiden polkujen löytämiseen ja testikattavuuden arviointiin.

6.1 Priorisointiin vaikuttavat muuttujat

- Liiketoiminnallinen arvo
- Liiketoiminnallinen visio
- Käyttäjäpalaute
- Käyttäjän saama arvo
- Riski
- Projektin muuttumisen volatiliteetti
- Kehittämisen kompleksisuus
- Vaatimusten taipumus virheellisyyteen

6.2 Painofunktiot priorisointiin

Painofunktion yleinen kuvaus verkossa G, solmuille V ja kaarille E.

$$\alpha := V(G), E(G) \rightarrow \mathbb{N}$$

Painofunktio yksittäiselle solmulle v, eli näkymälle.

$$\alpha(v) = value + vision \pm feedback - volatility - complexity - errorness$$

Painofunktio yksittäiselle kaarelle e, eli siirtymälle.

$$\beta(e) = value - volatility - complexity - errorness$$

Painofunktion polulle P solmusta v_1 solmuun v_2.

$$\gamma(P) = \sum_{v \in P} \alpha(v) + \sum_{e \in P} \beta(e)$$

6.3 Verkon rakentaminen

$$M_G = (a_{ij})_{3\times 3} = \begin{array}{c} G \\ v_1 \\ v_2 \\ v_3 \end{array} \begin{array}{c} \begin{array}{ccc} v_1 & v_2 & v_3 \end{array} \\ \begin{pmatrix} 1 & 1 & 2 \\ 1 & 0 & 0 \\ 2 & 0 & 0 \end{pmatrix} \end{array}$$

Kuva 6.1. *Esimerkki painotetusta verkosta ennen leikkauksia*

6.4 Verkon karsiminen

Painotetun verkon karsiminen eli leikkaaminen on prioriteeillä painotetun verkon tärkeä ominaisuus. Verkkoteorian soveltaminen prioriteettien avulla painotettuun verkkoon on erityisen hyödyllistä, kun verkon kaarissa korkea paino tarkoittaa suurta prioriteettia. Tällaisessa tapauksessa on mahdollista soveltaa lyhimmän polun ongelman ratkaisemiseen kehitettyjä algoritmeja, jolloin ne toimivat etsien alhaisimman prioriteetin polkuja. Lyhimmän polun etsimiseen on tarkoituksenmukaista valita aina aloitus ja lopetuspisteet, joiden välille lyhin polku verkossa voidaan etsiä. Prioriteetein painotetun verkon karsimistarkoitukseen olisi järkevää valita sellaiset aloitus- ja lopetuspisteet, joiden välillä ei näyttäisi olevan korkean prioriteetin solmuja. Voidaan kuitenkin menetellä myös siten, että valitaan aloitus- ja lopetuspisteeksi sellaiset solmut, jotka ovat painoltaan verkon alhaisimmat $v_1 = min(V)$ ja $v_2 = min(V \setminus \{v_1\})$ ja esimerkiksi verrata niiden lyhimmän polun kokonaisprioriteettia muuhun verkkoon.

- Pienimmän prioriteetin solmuparin etsiminen, eli $v_1 = min\{\alpha(V)\}$ ja $v_2 = min\{\alpha(V \setminus \{v_1\}\}$.
- Dijkstran algoritmin käyttö lyhimmän (prioriteetiltaan pienimmän) polun löytämiseen, eli $s = min(\gamma(P)), P \in G$.
- Leikkauksien tekeminen ja toistaminen n-kertaa.
- Poistetaan yhden yksittäiseen solmuun johtavat sillat, jossa $\alpha(v) < aivanliianalhainen$.
- Poistetaan kaikki yksittäiset eristetyt solmut, eli asteluku on nolla $d_G(X) = 0$.
- Poistetaan Dijkstran lyhimmän polun kaaret, jossa $\beta(e) < liianalhainen$.

Kuva 6.2. *Esimerkki painotetusta verkosta leikkauksien jälkeen*

6.5 Verkon ja testitapauksien yhteys

Ennen testitapauksien suunnittelua tehtävä priorisointi kuvainnollistaa käyttöliittymän näkymiä, niiden osanäkymiä ja niiden välisiä siirtymiä. Tällaisesta painotetusta verkosta saadaan priorisoitua näkymät ja siirtymät, mutta lopulliset testitapauksien prioriteetit ovat testitapaukseen kuuluvien näkymien tai siirtymien prioriteetteja. Tämä tarkoittaa käytännössä sitä, että kun näkymät ja siirtymät on priorisoitu, on esimerkiksi yhden yksittäisen tarkasteltavana olevan näkymän toiminnoilla sama keskenään prioriteetti.

7 TULOSTEN TARKASTELU JA ARVIOINTI

Tässä kappaleessa esitetään yhteenveto tutkimuksen tuloksista ja muun muassa pohditaan kuinka hyvin toistettavissa oleva kyseinen kehitetty menetelmä on.

7.1 Tutkimuksen konkreettiset tulokset

7.2 Menetelmän evaluointi

- Priorisointimenetelmän toistettavuus

7.3 Toteutuksen evaluointi

- Docker säiliönnin avulla tuki myös manuaaliselle testitapauksien ajamiselle
- Docker säiliönnin avulla sovelluskehittäjät saavat valmiin hyväksymistestausjärjestelmän helposti käyttöönsä
- Docker säiliöinnin takia toteutus ei ole sidottu CI palvelimeen
- Xvfb virtualisoinnin avulla voidaan uusia verkkoselaimia lisätä helposti
- Xvfb virtuallisoinnin avulla ei voida testata vain Windows ympäristöön saatavia GUI-ohjelmia
- Robot framework takaa helpon luettavuuden kenelle tahansa, mutta ohjelmistokehittäjälle rajatun tuntuinen
- Hyvä skaalautuvuus, docker säilöitä on helppo rakentaa ja GoCD agentteja lisätä

7.4 Jatkokehitysehdotukset

- Xvfb:lle vastineen löytäminen Window ympäristöön
- Parempi priorisointi muuttamalla näkymäperusteisuus käyttötapausperustaiseksi

8 YHTEENVETO

Tässä kappaleessa esitetään yhteenveto tehdystä työstä.

LÄHTEET

ISO:9126-1 (2001). *ISO/IEC 9126-1:2001*. en. URL: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/02/27/22749.html.

ISTQB Glossary (2019). URL: https://glossary.istqb.org/en.

Klärck, P. (2019). *How to write good test cases using Robot Framework*. URL: https://github.com/robotframework/HowToWriteGoodTestCases/blob/master/HowToWriteGoodTestCase
rst.

Robot Framework User Guide (ei julkaisupäivää). URL: https://robotframework.org/robotframework/3.1.2/RobotFrameworkUserGuide.html.

A ESIMERKKI TESTITAPAUKSESTA ROBOT FRAMEWORKILLÄ

```
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 DJIKSTRAN 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
```