TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

VILLE KÄHKÖNEN

# WEB-SOVELLUKSEN TEHOKAS PÄÄSTÄ PÄÄHÄN -TESTIAUTOMAATIO

Diplomityö

Tarkastaja: professori Hannu-Matti Järvinen
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunta-neuvoston kokouksessa 8. helmi-kuuta 2017

# TIIVISTELMÄ

**VILLE KÄHKÖNEN**: Web-sovelluksen tehokas päästä päähän -testiautomaatio
Tampereen teknillinen yliopisto
Diplomityö, 44 sivua, 1 liitesivu
Tammikuu 2017
Tietotekniikan diplomi-insinöörin tutkinto-ohjelma
Pääaine: Pervasive Systems
Tarkastaja: professori Hannu-Matti Järvinen

Avainsanat: päästä päähän -testaus, end to end -testaus, e2e-testaus, web-sovelluksen testaus, testaus, testiautomaatio

Ohjelmiston testaaminen ei ole triviaali tehtävä. Testauksen suunnittelun ja toteutuksen laadukkuudella on merkitystä erityisesti silloin, kun testataan järjestelmiä, joiden täytyy olla toimintavarmoja. Itseään toistavan manuaalisen testaamisen vähentämiseksi on ohjelmistojen testaamiseen sovellettu testiautomaatiotyökaluja. Tässä diplomityössä käsitellään testiautomaatiota, joka toimii Internet-selaimen käyttöliittymästä käsin ja testaa sovelluksen toimintaa päästä päähän (end to end). Käyttöliittymän logiikan lisäksi myös sovelluksen taustajärjestelmien liiketoimintalogiikka on osana testattavassa kokonaisuudessa eli järjestelmää testataan kokonaisuutena. Diplomityön tarkoitus on selvittää, millaisilla parannuskeinoilla web-sovellusten päästä päähän -testiautomaatiosta olisi mahdollista saada nykyistä tehokkaampaa niin, että se olisi entistä kannattavampi laadunvarmistuskeino web-sovelluksille.

Päästä päähän -testiautomaatioon liittyviä ongelmia ja kokemuksia tutkittiin haastattelemalla sellaisia ohjelmistoalan ammattilaisia, jotka olivat työskennelleet päästä päähän -testiautomaatiota hyödyntäneissä web-sovellusprojekteissa. Haastattelujen tuloksista johdettiin parannuskeinoja, joita hyödyntämällä aiemmissa projekteissa kohdattuja ongelmia voidaan jatkossa välttää.

Tutkimuksessa havaittiin, että ongelmia aiheuttivat hitaus, epävakaus, kirjoittamiskäytännöt ja teknologiavalinnat. Hitausongelma käsitti virheen havaitsemisen, virheen korjaamisen ja testien suorittamisen hitaudesta. Näistä yhdessä aiheutui se, että sovellukseen saatettiin luoda uusia virheitä ennen kuin entisiä ehdittiin korjata. Epävakausongelman aiheuttaja oli asynkronisuuden hallinnan vaikeus ja joissakin tapauksissa ongelma testaustyökalun ja selaimen yhteistoiminnassa. Testien epävakaus laski luottamusta testien tulosta kohtaan. Muita mielenkiintoisia havaintoja tehtiin muun muassa siitä millä tavalla, kuinka paljon ja missä vaiheessa kehitystyötä testejä luotiin, kuinka syvästi järjestelmää pyrittiin testaamaan ja koettiinko päästä päähän -testiautomaatio kannattavaksi.

Diplomityö esittää keinoja, joilla haastattelututkimuksessa löytyneet ongelmat voidaan ratkaista tai ongelmien vaikutusta voidaan lievittää. Kehitysideoihin lukeutuvat muun muassa testien suorituksen nopeuttaminen, ylläpidettävyyden parantaminen, kirjoittamiskäytäntöjen kehittäminen, vakauden parantaminen, testaustasojen suhteuttaminen ja osaamisen kehittäminen.

# ABSTRACT

**VILLE KÄHKÖNEN**: Efficient end-to-end test automation for web applications
Tampere University of Technology
Master of Science Thesis, 44 pages, 1 Appendix page
January 2017
Master's Degree Programme in Information Technology
Major: Pervasive Systems
Examiner: Professor Hannu-Matti Järvinen

Keywords: end-to-end testing, e2e-testing, web application testing, testing, test automation

Software testing is not a trivial task. The quality of design and implementation of testing is significant when it comes to testing applications that needs to work reliably. Test automation tools have been invented to reduce repetitive manual testing. This master's thesis deals with test automation which operates through web browser user interface and tests web application from end-to-end. End-to-end test automation targets to both front end and back end logic so the system is being tested as a whole. The purpose of the thesis is to find out what kind of improvements are needed in order to make end-to-end test automation more efficient so that end-to-end testing of web applications will be more profitable in the future.

The problems behind applying end-to-end test automation for web application projects were investigated by interviewing software designers and architects. The persons interviewed had worked on web application projects that involved the use of end-to-end test automation tools. The findings of the interviews were utilized to find ways to avoid problems that had been encountered in previous projects.

The study shows that the problems were caused by slowness and instability of tests as well as test design and technology choices. Tests were considered to be slow because of the time consumed before an error is detected, corrected and the test suite is run. This allowed more errors to be introduced before previous ones were fixed. The instability issue was caused by the difficulty in managing an asynchronous system, and in some cases, by a flaw in test automation tool and browser cooperation. The instability issue made test results unreliable. Other interesting findings were related to things such as in which manner, how many and in which state of development tests were created, how thoroughly the system was intended to be tested, and whether the interviewees considered end-to-end test automation to be profitable.

The thesis aims to find and present ways to solve or alleviate the problems found in the investigation. The thesis makes suggestions on how to speed up the tests, improve the sustainability of the tests, improve the way of writing the tests, make tests more stable, balance testing levels, and enhance skill development.

# ALKUSANAT

Tämän diplomityön aihe syntyi keväällä 2016. Solita Oy:n sisällä oli nähty tarvetta web-sovelluksen päästä päähän -testiautomaation tutkimiselle, sillä yrityksessä on tehty aiemmin ja tullaan todennäköisesti myös jatkossa tekemään sellaisia sovelluksia, joissa selaintestiautomaatiolla on paikkansa. Työntekijöiden näkemykset poikkesivat toisistaan melko paljon riippuen käytännön kokemuksista ja projektitaustasta.

Työ aloitettiin keväällä 2016, jolloin suoritettiin haastattelututkimus mahdollisten ongelmien ja ratkaisujen kartoittamiseksi. Diplomityö valmisteltiin pääosin syksyn 2016 aikana ja viimeisteltiin vuoden 2017 vaihteessa.

Haluan kiittää erityisesti kaikkia haastattelututkimukseen osallistuneita rohkeudesta ja tämän työn tekemisen mahdollistamisesta. Myös muut keskusteluissa mukana olleet kollegat ja läheiset ovat kiitoksen ansainneet. Lopuksi suuret kiitokset työtä ohjanneelle Petri Sirkkalalle ja työn tarkastaneelle professori Hannu-Matti Järviselle.

Tampereella, 16.1.2017

Ville Kähkönen

# SISÄLLYSLUETTELO

# LYHENTEET JA MERKINNÄT

| | |
|---|---|
| Asynkroninen tiedonsiirto | Synkronisen tiedonsiirron vastakohta. Asynkronisessa tiedonsiirrossa pyytävä osapuoli ei jää odottamaan vastausta vaan jatkaa normaalia toimintaansa ja käsittelee mahdollisen vastauksen sen saapuessa tai myöhemmin. |
| CI | Jatkuva integraatio (Continuous Integration), on toistuva tapahtuma, jossa kootaan kaikkien kehittäjien versionhallintaan tekemät muutokset yhteen, tehdään sovelluksesta ajettava käännös ja suoritetaan testit sitä vasten. |
| CSS | CSS (Cascading Style Sheets) on kieli, jolla määritellään HTML- ja XML-dokumenttien kuvantaminen kuvaruudulle, paperille, puheeksi tms. [1] |
| CSS-valitsin | Puurakenteen, esimerkiksi HTML, solmua vastaava mallinne tai kaava, jota voidaan käyttää osoittamaan elementtejä XML-dokumentissa. [1] |
| DOM | Dokumenttioliomalli on alusta- ja kieliriippumaton ohjelmointirajapinta, joka mahdollistaa dokumenttien dynaamisen sisällön, rakenteen ja tyylien käsittelyn ja päivittämisen. [2] |
| HTML | Hyper Text Markup Language on standardi merkkauskieli, jota käytetään verkkosivujen luomiseen. HTML-elementit ovat verkkosivujen rakennuspalikoita. |
| HTTP | Hyper Text Transport Protocol on yleinen tiedonsiirtoprotokolla, jota käytetään esimerkiksi Internet-selaimen ja verkkopalvelimen väliseen tiedonsiirtoon. |
| IDE | Integroitu ohjelmointiympäristö (Integrated Development Environment) on ohjelmistosovellus, joka tarjoaa hyödyllisiä työkaluja sovelluskehitykseen. |
| Käyttötapaus, skenaario | Konkreettisen tuloksen tuottava tapahtumasarja käyttäjän ja järjestelmän välisessä vuorovaikutuksessa. [3] |
| Mock-toteutus | Testausta varten luotu versio ohjelmistokomponentista, joka normaalin käyttäytymisen sijaan tarjoaa ennalta määriteltyjä tuloksia. [4] |

| | |
|---|---|
| Päästä päähän -testaus | Järjestelmän testaaminen kokonaisuutena päästä päähän (end-to-end). Testauksessa ovat mukana kaikki järjestelmän osat aina käyttöliittymästä tiedon pysyväistalletukseen. |
| Savutesti | Komponentin tai järjestelmän päätoiminnallisuuden kattava kaikista määritellyistä/suunnitelluista testitapauksista valittu osajoukko, jolla varmistetaan, että kaikkein kriittisimmät ohjelman toiminnot toimivat, mutta pienempiin yksityiskohtiin ei kiinnitetä huomiota. [3] |
| Testitapaus | Syötearvojen, suorituksen esiehtojen, odotettujen tulosten ja suorituksen jälkiehtojen muodostama kokonaisuus, joka on muodostettu tiettyä tavoitetta tai testauksen kohdetta varten, esimerkiksi tietyn ohjelmapolun testaukseen tai vaatimustenmukaisuuden varmistamiseksi. [3] |
| Toiminnallisuus | Kuinka hyvin ohjelmistotuote pystyy tuottamaan toiminnot, jotka täyttävät määrättyjen käyttöolosuhteiden edellyttämät tarpeet. [3] |
| Toiminnallisuustestaus | Komponentin tai järjestelmän toiminnallisuusmäärittelyihin pohjautuva testaus. [3] |
| Verkkosivu | HTML- tai muulla vertaisella merkkauskielellä kirjoitettu dokumentti, joka voi sisältää tekstiä, grafiikkaa, videota, ääntä tai muuta mediaa ja linkkejä toisille verkkosivuille. Verkkosivuja selataan Internet-selaimella. |
| Versionhallintajärjestelmä | Järjestelmä joka pitää kirjaa tiedostoon tai tiedostoihin tapahtuvista muutoksista siten, että palaaminen aiempaan määrättyyn versioon on mahdollista myöhemmin. |
| Virtuaalikone | Erillisenä tietokoneena isäntäkäyttöjärjestelmän sisällä isolaatiossa esiintyvä sovellus tai käyttöjärjestelmä, joka kykenee suorittamaan ohjelmia itsenäisen tietokoneen tapaan. |
| XPath-valitsin | Kuten CSS-valitsin, mutta polku XML-dokumentissa olevaan elementtiin ilmaistaan XML Path -kielellä. |
| Yksikkötestaus | Yksittäisten ohjelmistokomponenttien testaus. [3] |

# 1. JOHDANTO

Ohjelmistoista on tullut ajan saatossa yhä parempia käytettävyydeltään ja saatavuudeltaan. Tavoitteena on tuottaa sovelluksia, jotka ovat mahdollisimman hyviä käyttää. Ohjelmiston käytettävyyteen liittyy toiminnallisten ominaisuuksien lisäksi myös ei-toiminnallisia ominaisuuksia kuten saatavuus. Saatavuudella tarkoitetaan sitä, kuinka varmasti sovellus on käytettävissä ajasta, paikasta ja laitteistosta riippumatta. Ajalle trendikästä on, että korkeaa saatavuutta tavoitellaan tarjoamalla sovelluksen käyttöliittymä tietoverkon yli sellaisessa muodossa, että sen käyttäminen olisi mahdollista millä tahansa Internet-yhteydellä varustetulla laitteella ilman ylimääräisien lisäosien asennuksien tekemistä. Yleinen ratkaisu nykyään on tehdä palvelu, johon käyttäjä yhdistää Internet-selaimellansa ja saa vastauksena selaimen ymmärtämän sovelluskoodista ja merkkauksesta koostuvan kokonaisuuden, josta on mahdollista piirtää esitys laitteen ruudulle. Taustalla ovat web-standardit [5], joiden perusteella on mahdollista tehdä sovellus, joka on toimiva kaikissa standardeja noudattavissa Internet-selaimissa. Tähän web-maailmaan rakennetaan nykyään viihteen ohella myös toimintakriittisiä järjestelmiä, jotka vaativat testaamista sillä tasolla, että voidaan todeta niiden täyttävän liiketoiminnan asettamat vaatimukset. Manuaalisen testaustyön vähentäjäksi, mutta ei niinkään korvaajaksi, on pyritty valjastamaan testiautomaatiota, jolla sovelluksen toimivuudesta halutaan tietoa jatkuvasti, kun sovellukseen tehdään muutoksia. Tämä diplomityö on tehty siksi, että selaimen kautta operoivan testiautomaation ongelmia ja kehityskohteita saataisiin kartoitettua ja mahdollisiin ongelmiin voitaisiin löytää ratkaisuja.

Web-sovellusten käyttöliittymästä käsin toimivan testiautomaation on tunnettu olevan hidasta, vaikeata ja epästabiilia [6] [7]. Tämä tutkielma pyrkii vastaamaan siihen, miten päästä päähän -testiautomaation tekemistä voitaisiin nykyisestä tilanteesta parantaa. Samalla mietitään sitä, onko ylipäätänsä kannattavaa tehdä selaimen kautta operoivaa järjestelmää kokonaisuutena testaavaa testiautomaatiota, sillä ratkaisevana tekijänä on keskiössä oleva liiketoiminta ja sen kannattavuus. Välttämättä testiautomaation tekemiseen käytetylle rahalle ja vaivalle ei saada vastinetta. Yhtä hyvin pelkkä manuaalinen testaaminen ja esimerkiksi tutkiva testaaminen voivat olla kannattavampia tapoja tehdä laadunvarmistusta. Tavoitteena on tukea sekä kehittäjien että projekteista vastuussa olevien henkilöiden ammatillista päätöksentekoa web-ohjelmistoprojektien eri vaiheissa.

Ensin työssä käsitellään testaamista yleisellä tasolla tarkoituksena kasvattaa ymmärrystä siitä, mitä ohjelman testaamisella tarkoitetaan, mihin sillä pyritään ja miten testiautomaatio liittyy testaamiseen. Sitten kuvaillaan päästä päähän -testausta sen pääpiirteiden osalta kokonaiskuvan hahmottamisen helpottamiseksi ja täsmennetään, mitä päästä päähän -testauksella ja testiautomaatiolla tarkoitetaan tässä työssä. Seuraavaksi luvussa

kolme kerrotaan työssä käytetyistä tutkimusmenetelmistä, tutkimuksessa hankitun tietoaineiston hyödyntämisessä käytetystä teoriasta, aineiston hankkimiseksi suoritetun haastattelututkimuksen toteutuksesta ja lopuksi esitellään haastatteluissa kysytyt kysymykset. Luvussa 4 esitellään haastattelututkimuksen seurauksena saadut tulokset ja luvussa 5 tarkastellaan tuloksia ja kehitysideoita yhdistäen niitä alalla vallitsevaan tietoon mahdollisten parannuskeinojen löytämiseksi. Lopuksi vedetään yhteen ne ajatukset, joiden avulla päästä päähän -testiautomaatiota voidaan jatkossa kehittää ja viimeiseksi arvioidaan tutkimuksen onnistumista.

# 2. TAUSTAT JA LÄHTÖKOHDAT

Tässä luvussa on tarkoitus käydä läpi testaamisen periaatteita yleisellä tasolla ja luoda pohjaa sille, että mitä päästä päähän -testaamisella tarkoitetaan ja miettiä millaisia haasteita päästä päähän -testaamisessa on, kun sitä verrataan testauksen teoriaan. Lisäksi käsitellään sitä, minkä tyyppisien järjestelmien testaamisesta tämän työn piirissä on suurin piirtein kyse, jotta työssä saadut tulokset yhdistetään tyypiltään kuvatun kaltaisiin järjestelmiin, eikä joihinkin muihin, joihin tulokset eivät sovellu.

## 2.1 Testaus

Aluksi käsitellään ohjelman testaamista yleisellä tasolla siten, että ymmärretään mihin testauksella pyritään, mitä se tarkoittaa ja mitkä sen haasteet ovat. Lopuksi kerrotaan, miten automaatio liittyy nykyisellään testaamiseen ja mitä tarkoitetaan sanalla testiautomaatio.

### 2.1.1 Tarkoitus

Testauksen tavoitteena on virheiden löytäminen. Lähtökohta on, että onnistunut testi havaitsee virheen ohjelman toiminnassa. Epäonnistunut testi puolestaan on sellainen, ettei se havaitse ohjelmassa piileviä virheitä. [8]

Toisinaan esiintyvä harhaluulo on, että testaamisessa varmistetaan, että ohjelma tekee sen, mitä sen oletetaan tekevän. Tällöin kuitenkin tullaan testanneeksi vain osa ohjelmasta. Tämä osa ohjelman toimintaa on usein juuri se osa, jonka ihminen haluaa nähdä, koska asioiden rikkominen ei ole kaikille ihmisille luontaista. Ohjelmien testaamisessa kuitenkin pitäisi keskittyä rikkomiseen. Pitää keksiä keino, jolla ohjelman voidaan osoittaa toimivan väärin. [8]

Mitä tarkoittaa se, että ohjelma toimii väärin? Ohjelma voi toimia väärin monella tavalla. Se voi ensinnäkin tehdä jotakin, mitä sen ei pitäisi tehdä. Toiseksi se voi jättää tekemättä jotakin, mitä sen pitäisi tehdä. Kolmanneksi se voi toimia toisin, kuin ohjelman määrittelyssä mainitaan. Neljäntenä ohjelmassa voi olla jokin muu ongelma kuten vaikeakäyttöisyys, hitaus tai jokin muu syy miksi ohjelma toimii käyttäjän mielestä väärin. Testaamista on se, että pyritään osoittamaan, että jokin edellä mainituista seikoista on tosi. Toisin sanoen testauksessa ohjelmaa suoritetaan siinä tarkoituksessa, että ohjelmasta löydetään virheitä [8]. [9]

Ohjelmaa voi testata sekä toiminnallisten että ei-toiminnallisten ominaisuuksien osalta. Kun ohjelmalle tehdään toiminnallista testaamista, keskitytään vain siihen, että ohjelma tekee määrittelyn mukaisesti kaiken sen, minkä sen pitää tehdä, eikä tee jotakin ylimääräistä, mitä sen ei pitäisi tehdä. Toisin sanoen keskitytään siihen, mitä ohjelma

tekee. Ei-toiminnallisten ominaisuuksien testaamisessa keskitytään siihen, miten ohjelma tekee sen, mitä se tekee. Tutkitaan, millaisien rajoituksien puitteissa ohjelma toimii.

Testauksessa vertailukohtana on ohjelmalle asetetut vaatimukset [9]. Pitää tietää, mitä järjestelmältä odotetaan. Sellaista ohjelmaa, jonka vaatimuksia ei tunneta tai jolla vaatimuksia ei ole, on mahdotonta testata, koska silloin ohjelman toiminnasta tehdyillä havainnoilla ei ole lainkaan vertailukohtaa.

Löydettyjen virheiden avulla on mahdollista parantaa ohjelman laatua. Löydetyn virheen korjaamisen jälkeen ohjelmalle asetetut vaatimukset täyttyvät paremmin ja ohjelman laatu paranee heti. [9]

### 2.1.2 Haasteet

Jopa pienissä sovelluksissa kaikkien virheiden löytäminen on mahdotonta. Tämä johtuu siitä, että olemassa olevien syötteiden ja mahdollisten suorituspolkujen määrä kasvaa nopeasti valtavaksi. Jos otetaan esimerkiksi yksinkertainen maksimissaan 20 kertaa pyörivä silmukka, jonka sisällä on 5 eri suorituspolkua, erilaisten suorituspolkujen lukumäärä on tuolloin 5^20 + 5^19 + … + 5^1 ≈ 100 triljoonaa [8, pp. 10-11]. Huomataan, että jo hyvin yksinkertaisen sovelluksen läpikotaisin testaaminen on mahdotonta. Voidaan todeta, ettei ole mahdollista osoittaa ohjelman täydellistä virheettömyyttä[1].

Testausmenetelmä vaikuttaa testauksen haasteellisuuteen. Mustalaatikkotestaamisessa täydelliseen testikattavuuteen pääsemiseksi olisi testattava kaikki mahdolliset sisääntulot, koska kyseessä on nimensä mukaisesti testausmenetelmä, jossa testaaja ei näe ohjelman sisäiseen toteutukseen. Lasilaatikkotestaamisessa puolestaan testaaja näkee ohjelman toteutuksen. Tällöin riittää periaatteessa kaikkien mahdollisten suorituspolkukombinaatioiden testaaminen. Kuitenkin siitä huolimatta, että kaikki suorituspolut voitaisiin testata, ei ensinnäkään tiedetä vastaako toiminto vaatimuksia. Toiseksi ei tiedetä puuttuuko ohjelmasta tarvittavia suorituspolkuja. Kolmanneksi pelkät suorituspolut eivät riitä, koska voi olla esimerkiksi ohjelmointikielestä riippuvia tilanteita, joissa virhe saadaan aikaan tietyllä syötteellä. [8] Esimerkiksi JavaScript-kielessä yhden alkion taulukolla [”10”].map(parseInt) antaa oikean tuloksen [10], mutta neljän alkion taulukolla [”10”, ”20”, ”50”, ”100”].map(parseInt) tulos on järjetön [10, NaN, NaN, 9][2].

Testitapauksella tarkoitetaan syötearvojen, suorituksen esiehtojen, odotettujen tulosten ja suorituksen jälkiehtojen muodostamaa kokonaisuutta, joka on muodostettu tiettyä tavoitetta varten, esimerkiksi tietyn ohjelmapolun testausta tai vaatimustenmukaisuuden varmistamista varten. Ohjelman monimutkaisuudesta johtuvan suorituspolkujen valtavan lukumäärän valossa on siis järkevintä muodostaa jokin äärellinen joukko testitapauksia, joilla saadaan löydettyä virheitä mahdollisimman tyydyttävästi. Tosin löydettyjen virheiden lukumäärä yksinään ei vielä kerro paljoa testauksen onnistumisesta, vaan

[1] Ihan pienimmille ohjelmille, kuten esimerkiksi taulukon järjestämisalgoritmille, on mahdollista tehdä matemaattinen oikeaksi todistaminen. Tässä työssä käsitellään kuitenkin paljon suurempia kokonaisuuksia.
[2] Kokeiltu selaimilla Microsoft Internet Explorer 11.0.9600.18282 ja Google Chrome 49.0.2623.112 sekä NodeJS-ajoympäristön versiolla 5.1.0.

siihen vaikuttaa se, mitkä sovelluksen toiminnan piirteet ovat käyttäjän kannalta arvokkaita. Toisin sanoen kannattaa miettiä, mitkä ovat niitä asioita, jotka tuottavat sovellukselle arvoa. Kun tunnistetaan arvokkaita piirteitä, niiden osalta voidaan tehdä priorisointia. Esimerkiksi eri käyttötapauksien välillä voi vallita erilainen prioriteetti. Tietoturvallisuuden prioriteetti puolestaan riippuu muun muassa siitä, kuinka arkaluonteista dataa sovellus käsittelee. Kolmas priorisoinnin piiriin kuuluva ominaisuus voisi olla esimerkiksi sovelluksen käytettävyys.

Testitapausten suunnittelu on tärkeä osa testaamista. Testitapausten suunnittelussa auttaa se, että päästään katsomaan ohjelman sisälle, jotta testitapauksia varten voidaan tehdä järkeviä oletuksia [8]. Jos laskin osaa laskea 1+2, niin ei ole välttämättä järkevää testata sokeasti, että positiivisten lukujen yhteenlasku toimii myös kaikilla muilla arvoilla. Sen sijaan ohjelman sisäistä toteutusta tarkastelemalla voidaan keksiä tilanteita, joissa laskin ei noudata oikein laskusääntöjä. Jos keksitään rajatapaustilanteita, kannattaa testata niitä. Esimerkiksi, jos on tiedossa laskimen lukuarvorajoitteet, kannattaa testata laskimen toimintaa niiden läheisyydessä.

Testitapauksien lukumäärän rajallisena pitämisessä auttaa se, että ei turhaan testata samaa ohjelman suorituspolkua useasti. Yksi tapa rajoittaa testitapauksia on jakaa sovelluksen toiminta ekvivalenssiluokkiin. Ekvivalenssiluokkiin jakamisella tarkoitetaan sitä, että jaetaan syöteavaruus jollakin ehdolla kahteen tai useampaan ryhmään. Esimerkiksi laskimen yhteenlaskun toiminnan testaamisen voisi jakaa positiivisien lukujen ja negatiivisien lukujen yhteenlaskuun. Lisäksi erikoistapauksena nollan esiintyminen osana yhteenlaskua. [9]

Päästä päähän -testaus on vain yksi sovelluksien testaamisen muoto muiden joukossa. Muita muotoja ovat esimerkiksi yksikkötestaus ja integraatiotestaus. Näiden erona on se, että ne testaavat sovellusta erikokoisten kokonaisuuksien osalta. Yksikkötesti, kuten sen nimikin sen sanoo, testaa vain pientä yksikköä. Integraatiotestit puolestaan testaavat useamman yksikön yhteistyötä ja järjestelmäintegraatiotestit järjestelmän osien keskustelun toimivuutta. Näitä eri testausmuotoja yhdistelemällä voidaan suhteellisen pienellä testimäärällä saavuttaa suuren testikattavuuden, kun mukaan otetaan oletus, että suurempaa kokonaisuutta testaava testiautomaatio voi luottaa siihen, että toimintoa on testattu tarkemmin pienemmässä mittakaavassa yksikkötasolla.

### 2.1.3 Testiautomaatiosta

Aina kun automaatiosta on kyse, on hyvä muistaa, ettei automaatio itsessään ratkaise ongelmia. Automaatio ei ole hopealuoti. Automaatio on työkalu, jonka avulla on mahdollista saada ennalta määritelty toiminto toistettua ilman käsityötä. Testiautomaatiolla tarkoitetaan tietokoneen ymmärtämässä muodossa ilmaistuja ohjeita ja sääntöjä, joita noudattamalla tullaan tarkistaneeksi, että ohjelma täyttää ohjeet ja säännöt laatineen ihmisen mielikuvan ohjelman oikeasta toiminnasta.

Jos testaamisen ymmärtää ihmisen tekemänä luovana aivotyönä, testiautomaation suorittamisessa ei ole täysin kyse testaamisesta, vaan tarkisteen toistamisesta. Varsinainen testaaminen tapahtui oikeastaan tarkisteen luomisen aikana tai sitä ennen. Näin on ainakin silloin, kun muutoksia ei tapahdu. Jos testiä muutetaan, niin kyse on taas selvästi testaamisesta. Jos testauksen kohdetta muutetaan, on kyse siitä, havaitseeko testi muutosta.

Automaation rakentamiseen tarvitaan työtä ja aikaa. Tämän lisäksi automaation rakentamisen aikana keskittyminen ei ole pelkästään testaamisessa, sillä testitapausten luominen testiautomaatiotyökalulla on oma haasteensa. Siksi aina kannattaa miettiä olisiko testiautomaation rakentamisen sijaan jollakin toisella tavalla mahdollista saavuttaa parempia tuloksia.

## 2.2 Päästä päähän -testaus

Tässä aliluvussa käsitellään päästä päähän -testausta yleisesti. Kokonaiskuvan hahmottumista varten on ensiksi tarkoituksena tarkastella asiaa pääpiirteiltään ennen haastattelututkimuksen tekemistä. Myöhemmin aliluvussa 2.2.2 esitellään yhdenlainen esimerkkiarkkitehtuuri, jonka mukainen testauksen kohteena oleva web-sovelluksen toteuttava järjestelmä voisi olla. Samalla sivutaan lyhyesti päästä päähän -testaukseen soveltuvia testiautomaatioteknologioita. Aliluvun tärkein tarkoitus on kertoa, mitä päästä päähän -testauksella tarkoitetaan tässä työssä.

### 2.2.1 Päästä päähän -testauksen piirteitä

Tämä työ keskittyy web-sovellusten päästä päähän -testiautomaation tekemisen tehostamiseen. Päästä päähän -testejä, kuten muitakin testejä, tai tarkemmin testitapauksia, voidaan periaatteessa luoda monella tavalla. Ihmisen suoritettavaksi testitapauksia voisi kuvailla esimerkiksi luonnollisella kielellä. Tätä kirjoitettaessa tunnetuimpia tapoja toteuttaa tietokoneen suorittamaksi tarkoitettuja päästä päähän -testitapauksia on kuvailla ne ohjelmointikielellä tai nauhoittaa siihen tarkoitetulla työkalulla testaajan käyttäytymistä. Kolmas tapa voisi olla luoda testaustyökalulla satunnaissyötettä. Tässä työssä käsitellään testiautomaatiota, joka tarkoittaa ihmisen jollakin ohjelmointikielellä tai muilla tavoin luomia testitapauksia, jotka suorittaa tietokone. Tämän aliluvun tarkoituksena on antaa mahdollisimman hyvä kokonaiskuva päästä päähän -testauksen mahdollisuuksista, rajoitteista ja yleisestä ongelmakentästä.

Päästä päähän -testaaminen on järjestelmän toiminnallisuustestaamista, eli siinä testaus pohjautuu järjestelmän toiminnallisuusmäärittelyihin. Päästä päähän -termillä tarkoitetaan sitä, että testaus ulottuu järjestelmän käyttöliittymästä aina syvälle liiketoimintalogiikkaa ja tiedon talletusta sisältäviin osiin. Tällöin testejä suoritettaessa koko järjestelmä on toiminnassa mukana: palvelintietokone, käyttöjärjestelmä, Internet-yhteys, tietokanta, sovellus itse ja niin edelleen.

Päästä päähän -testeissä on mahdollista testata yleisen testausperiaatteen mukaisesti sitä, että järjestelmä tekee sen, mitä sen pitäisi tehdä ja ainakin jossakin määrin myös sitä, ettei se tee jotakin, mitä sen ei pitäisi tehdä. Se, että aina ei voida testata, mitä järjestelmän ei pitäisi tehdä, johtuu siitä, että testaajalla, oli se sitten robotti tai ihminen, on virheraporttia varten saatavilla yleensä vain käyttöliittymässä näkyvä virheilmoitus. Testaaja myös varmistaa järjestelmän toimintaa yleensä vain käyttöliittymästä. Testaaja ei näe esimerkiksi sitä, tallentuiko tietokantaan oikea määrä oikeaa tietoa tapahtuman yhteydessä tai lähettikö järjestelmä sähköpostia oikeisiin osoitteisiin.

Sillä, että testien suorituksessa on osallisena koko järjestelmä, on sekä hyvät että huonot puolensa. Hyvä puoli on se, että testeillä on mahdollista löytää virheitä monesta eri järjestelmän osasta. Testeillä saadaan siis kasvatettua nopeasti testikattavuutta. Esimerkiksi sovelluspalvelimen konfiguraatioon tehdystä virheellistä toimintaa aiheuttavasta muutoksesta voidaan saada palautetta testien suorituksessa. Virheelliset tietokantakyselyt käyvät ilmi. Rikkoutunut käyttöliittymä tai rajapinta voidaan huomata ja niin edelleen. Huonona puolena koko järjestelmän osallisuudesta on selvästikin se, ettei virheen sattuessa välttämättä ole lainkaan selvää, mistä virhe aiheutui, koska kuten edellä mainittiin, testien suorittaja tarkistaa järjestelmän toimintaa yleensä vain käyttöliittymästä. Käyttöliittymässä näkyvä virheilmoitus, jo pelkästään tietoturvallisuussyistä, ei anna tarkkaa kuvaa siitä, mitä järjestelmän sisällä tapahtui.

Päästä päähän -testauksessa hyödynnetään sekä lasilaatikko- että mustalaatikkotestausmenetelmiä [10]. Testejä kirjoitettaessa voidaan hyödyntää ohjelman koodia, mikä tekee siitä lasilaatikkotestausta. Ohjelmakoodin hyödyntäminen ei kuitenkaan ole lainkaan pakollista, koska testejä voi tehdä esimerkiksi pelkän web-selaimeen asennettavan liitännäisen avulla. Joka tapauksessa kyse on myös mustalaatikkotestaamisesta, sillä testitapauksia suunniteltaessa ei tarvita mitään tietoa järjestelmän sisäisestä toteutuksesta toisin kuin puhtaassa lasilaatikkotestauksessa. Koska kyse ei ole puhtaasta lasi- eikä mustalaatikkotestaamisesta, päästä päähän -testaamista voidaan kutsua harmaalaatikkotestaamiseksi [9].

Yksi näkökulma voisi olla kirjoittaa testejä loppukäyttäjän näkökulmasta [10]. Silloin tavoitteena on jäljitellä käyttäjän mahdollisia tekemisiä käyttöliittymässä. Taustalla on idea, että testatuksi tulisivat järjestelmän loppukäytössä odotettavissa olevat käyttötapaukset. Tätä lähestymistapaa varten on otettu käyttöön käyttäytymislähtöisiä (behaviour driven) menetelmiä, joissa testit pyritään kirjoittamaan helposti luettavan tarinan muotoon. Näin luodut tarinat, skenaariot, voivat toimia myös kommunikaatiovälineenä kehittäjien ja muiden projektin jäsenien tai sidosryhmien välillä, sillä ohjelman toiminnan kuvausta voi lukea silloin hyvin kirjoitetuista testeistä myös ohjelmointitaidoton. Yhtenä ongelmana tällaisessa lähestymistavassa voisi nähdä sen, että se ajaa testien kirjoittajan helposti käsittelemään vain niin sanottuja onnellisia tapauksia, joissa ei testata lainkaan sitä, ettei järjestelmä tee jotakin mitä sen ei pitäisi tehdä. Perusteena tälle on se, että on luonteikasta kirjoittaa ja lukea yksinkertaisia skenaarioita. Helposti ensimmäisenä tulee mieleen skenaario, jossa annetaan lähtöasetelma, suoritetaan toiminnot ja lopuksi tarkistetaan, että toimintojen vaikutuksen kohteena olleet osat ovat oikeassa tilassa. Yleisen

testausperiaatteen mukaisesti skenaariossa pitäisi tarkistaa myös se, ettei järjestelmä tee jotakin mitä sen ei pitäisi tehdä. Esimerkiksi pankkisovelluksessa tilisiirto voidaan testata kirjoittamalla skenaario ”A kirjautuu sisään ja siirtää B:n tilille 5 euroa. A:n tilillä on nyt 5 euroa vähemmän rahaa. B kirjautuu sisään ja huomaa, että tilillä on 5 euroa enemmän rahaa kuin ennen.”. Testitapaus on järkevä ja se kuvaa sitä, mitä järjestelmässä piti tapahtua, mutta siitä puuttuu tarkistus, että kaikilla muilla tileillä on yhtä paljon rahaa lopussa kuin oli alussa ja ettei mitään muita asiaan kuulumattomia tapahtumia tapahtunut.

Toisena esimerkkinä päästä päähän -testaamisesta voitaisiin käyttää yksinkertaista ajanvarausjärjestelmää, jossa käyttäjät voivat tehdä ajanvarauksia web-käyttöliittymästä tapaamisia varten. Yksinkertainen todellisen käyttäjän näkökulmasta tehty testitapaus voisi olla sellainen, että käyttäjä kirjautuu sisään järjestelmään, siirtyy ajanvarausnäkymään, tekee ajanvarauksen ja lopettaa käytön. Tällöin järjestelmä pitää asettaa ensin testin suorittajan kannalta tunnettuun tilaan. Sitten testin suorittajan pitää tarkistaa, että kirjautuminen onnistuu, ajanvarausnäkymään siirtyminen onnistuu, ajanvarauksen tekeminen onnistuu ja se todella tallentuu pysyvästi ja lopulta uloskirjautuminen onnistuu. Jokaisen toiminnon yhteydessä tarkistetaan, ettei sovellus anna yhtään virheilmoitusta. Testataan siis, että koko käyttötapaus toimii vaatimusten mukaisesti.

Tällaisen järjestelmän päästä päähän -testaaminen on periaatteessa yksinkertaista, koska käyttötapaukset ovat yksinkertaisia. Todellisuudessa järjestelmät ovat yleensä paljon monimutkaisempia. Monimutkaisemmissa järjestelmissä hyvien testitapausten suunnitteluun kannattaa kiinnittää huomiota. Testitapauksien suunnittelu on haastava ajatteluprosessi ja sillä on suuri merkitys lopputuloksen kannalta. Huolellisella testitapauksien suunnittelulla voi vaikuttaa virheiden löytämisen ohella myös testien toistettavuuteen ja ylläpidettävyyteen.

Tarvittavien testitapausten määrä kasvaa hyvin nopeasti järjestelmän monimutkaistuessa. Jos edellä mainittuun ajanvarausjärjestelmään lisätään esimerkiksi käyttöoikeushallinta, tulee jokaista käyttöoikeusrajattua toimintoa kohden lisää mahdollisia käyttötapauksia. Hyvin nopeasti päästään tilanteeseen, ettei projektilla ole resursseja tehdä ja ylläpitää niin suurta määrää testejä, että voisi sanoa testauksen testiautomaation osalta olevan kovin kattavaa. Tällöin kysymykseen voisivat tulla niin sanotut savutestit, joiden ei ole tarkoituskaan testata useimpia toimintoja, vaan niiden pääpaino olisi pelkästään tärkeimpien tai kriittisimpien toimintojen testaamisessa.

Lisähaasteen aiheuttaa myös se, jos sovelluksen halutaan toimivan useilla selaimilla, päätelaitteilla, alustoilla, eri virtuaaliajoympäristön versioilla, eri kielillä ja niin edelleen. Tällöin testauksen tekeminen eri ympäristön variaatioilla voisi tulla kysymykseen. Erityisesti eri selaimilla testauksen tekeminen kannattaa, koska valtavirtaa edustavia selaimia on useita ja selaimet toimivat standardeista [5] huolimatta hieman eri tavalla [8, p. 195]. On myös mahdollista, että sama selain toimii toisessa käyttöjärjestelmässä hieman eri tavalla. On tietenkin tilanteita, joissa järjestelmän käyttäjäkunta on suppea ja käyttö tapahtuu pelkästään yhdellä käyttöjärjestelmällä ja yhdellä selaimella. Harvemmin näin kuitenkaan on.

Jos järjestelmän halutaan toimivan usean tyyppisillä päätelaitteilla, on näytön pikselitarkkuuden huomioon ottaminen tärkeää etenkin nykyisin, kun web-selain on usein mukana PC-tietokoneissa, puhelimissa, kannettavissa tietokoneissa, sormitietokoneissa, kotiteatterilaitteistoissa, televisioissa, pelikonsoleissa ja niin edelleen. Web-sovelluksen näkymien testaaminen voisi tulla kysymykseen siis myös eri näytön ko'oilla. Myös muiden ei-toiminnallisien ominaisuuksien kuten käytettävyyden tai suorituskyvyn mittaaminen voisi tulla kysymykseen. Toiminnallisten ominaisuuksien lisäksi myös ei-toiminnallisten ominaisuuksien testaaminen laajastikin on mahdollista, mutta on eri asia, onko se rajallisten resurssien näkökulmasta katsottuna kannattavaa.

Testien suorittamisessa kuluu aikaa riippumatta siitä, suoritetaanko ne automaattisesti vai ihmisen käsin tekemänä. Tämä johtuu tietenkin siitä, että sovelluksen suorituksessa kuluu aikaa. Web-sovelluksien suorituksessa aikaa kuluu tyypillisesti pyyntö-vastaus tietoliikenteeseen, tietokantakyselyyn, taustajärjestelmän ohjelman suoritukseen ja selaimessa käyttöliittymälogiikan suorittamiseen ja näkymän piirtämiseen. Päästä päähän -testit ovat hitaita. On selvää, että testitapausten määrän tulee pysyä siinä määrin siedettävänä, että testien ajamiseen ei kulu kohtuuttoman paljon aikaa. Testitapauksien suunnittelussa kannattaa pitää huoli, ettei tehdä turhia päällekkäisiä testitapauksia, joissa sama asia testataan useasti.

Testien ajamiseen kuluvalla ajalla on vaikutusta siihen, kuinka nopeasti virheet saadaan korjattua, koska mitä kauemmin testien ajamisessa kuluu aikaa, sitä kauemmin kehittäjällä kestää saada palautetta testien suorituksen onnistumisesta. Tähän liittyy myös sellainen riski, että pitkän aikaa vievän testien ajon aikana versionhallintajärjestelmään tallentuu uusia muutoksia, jotka rikkovat jonkin toisen ominaisuuden. Näiden ominaisuuksien toimivuudesta saadaan palautetta vasta sen jälkeen, kun sovelluksesta luodaan uusi ajettava käännös ja kaikki testit suoritetaan uudelleen. Ongelmallista on erityisesti se, että jos useita uusia muutoksia kasaantuu yhteen testiajoon, virheitä voi olla vaikea yhdistää niitä aiheuttaneisiin muutoksiin.

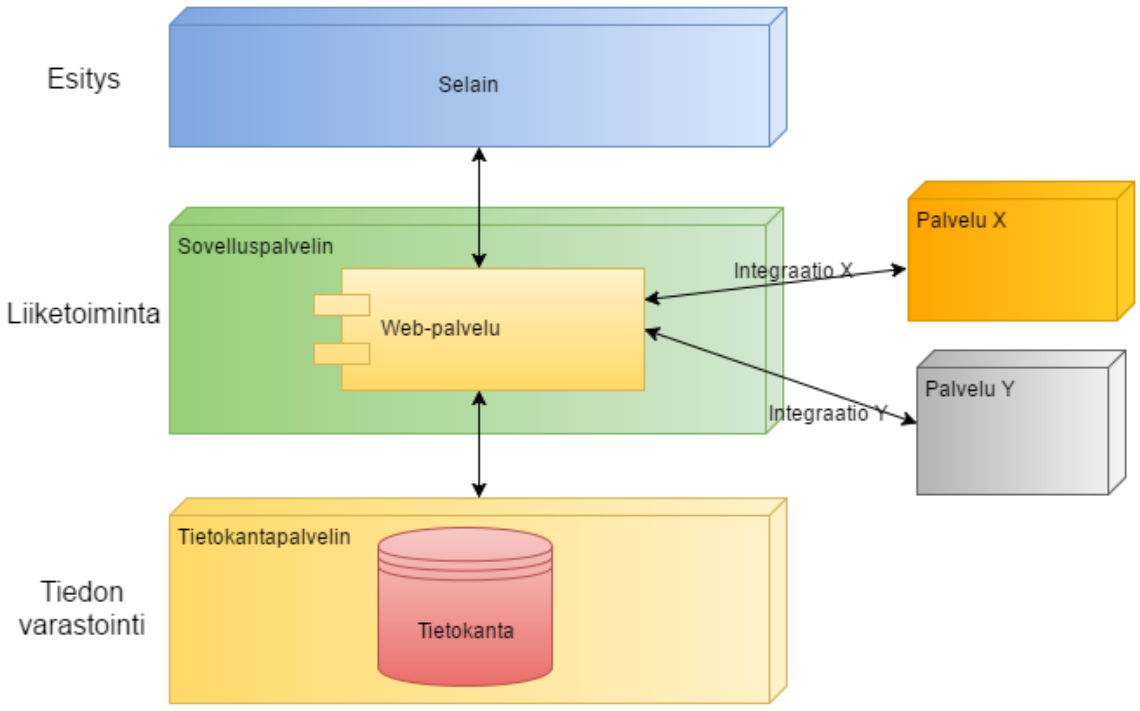
On vaikeaa sanoa, kuinka paljon testien suorittamiseen saa kulua aikaa, koska se on riippuvainen monesta tekijästä. Ainakin projektin koolla ja kestolla voisi helposti kuvitella olevan merkitystä asiaan. On aivan eri asia kestääkö päästä päähän -testiautomaation suorittamisessa kahvitauon vai kokonaisen työpäivän verran aikaa [6]. Jos suuren muutoksen yhteydessä rikotaan vahingossa järjestelmän muita osia ja testien suorittamiseen kuluu aikaa kokonainen yö, muutoksesta rikkoontumisen kautta takaisin toimivaan tilaan siirtymiseen kuluu kaksi päivää. Tämä johtuu siitä, että muutoksen aiheuttamasta rikkoontumisesta saadaan tieto vasta muutoksen jälkeisenä päivänä ja jos korjaus ehditään tehdä samana päivänä, niin tieto korjauksen onnistumisesta saadaan sitä seuraavana päivänä. Näin siis silloin, jos oletetaan, että mikä tahansa muutos voi rikkoa minkä tahansa osan sovelluksesta. Tämä kuitenkin voi olla harvinaista käytännössä. Yleensä ominaisuuden korjauksen yhteydessä riittää, että suoritetaan kyseistä ominaisuutta koskevat testit uudestaan.

Kaiken testaaminen on mahdotonta, joten käytössä oleva aika kannattaa käyttää siihen, että luodaan testejä, joilla ohjelmasta löydetään tehokkaasti virheitä [8]. Testauksessa voidaan esimerkiksi keskittyä eliminoimaan virheitä niiden luonteen perusteella. Testauksen kohteita voidaan priorisoida. Jos halutaan välttää vakavia virheitä, niin kannattaa testata ainakin korkealla priorisoituja ominaisuuksia. Esimerkiksi liiketoiminnan jatkumisen kannalta elintärkeiden sovelluksen osien testaaminen voisi olla korkealle priorisoitua.

Edellä esitetyistä ajatuksista päästäänkin siihen kysymykseen, miten web-sovelluksia voidaan päästä päähän -testata automaattisesti, järkevästi ja tehokkaasti, jotta mahdollisimman paljon virheitä saataisiin löydettyä mahdollisimman pienellä vaivalla. Yleinen testausperiaate on, ettei kaikkea kyetä testaamaan, koska tarvittavien testitapausten määrä kasvaa nopeasti liian suureksi, eikä automaatio ole ratkaisu ongelmiin [9]. Tämä työ keskittyy etsimään ratkaisuja siihen, miten päästä päähän -testiautomaatiota voitaisiin tehdä tehokkaammin.

### 2.2.2 Arkkitehtuurit ja teknologiat

Web-sovellusten maailmassa on hyvin yleistä, että pohja-arkkitehtuurina on asiakas-palvelin-malli, jossa keskustelu tapahtuu asiakkaan tekemien pyyntöjen ja palvelun tarjoamien vastauksien avulla. Toinen yleinen suunnitteluperiaate on kolmikerrosmalli, joka muodostuu esitys-, liiketoiminta- ja tiedon varastointikerroksista. Kuvassa 1 on esitetty karkealla tasolla arkkitehtuuri sellaiselle järjestelmälle, jota tässä työssä käsitellyllä testiautomaatiolla saatettaisiin haluta testata. [8]




Kuva 1. Testattavan järjestelmän esimerkkiarkkitehtuuri

Selaimen kautta tehtävän testiautomaation toiminnan kannalta on mielenkiintoista, miten sovelluslogiikka jakautuu selaimen ja sovelluspalvelimen välillä. Jos sovelluslogiikka ja tila ovat pelkästään palvelimen puolella, testien suorituksessa pyyntöjen seurausten jälkeiset tarkisteet voidaan tehdä heti, kun palvelimelta saatu staattinen HTML-dokumentti on ladattu ja tulkittu selaimessa. Jos kyse on dynaamisesta web-sovelluksesta, kuten nykyään on tapana, testien suorittaminen on hieman monimutkaisempaa. Dynaamisen web-sovelluksen tapauksessa palvelin lähettää selaimelle HTML-dokumentin lisäksi selaimessa suoritettavaa koodia, joka tekee asynkronisia HTTP-pyyntöjä ja dokumentin rakenteen muokkausta. Monimutkaisuuden aiheuttaa se, että on vaikeampaa tietää varmaksi, missä kohtaa kaikki tarvittavat pyynnöt on tehty ja näkymä on valmis. Tätä varten päästä päähän -testaustyökaluissa on menetelmiä, joilla testeissä voi ilmaista mitä käyttäjälle esitettävän dokumentin tulee sisältää silloin, kun näkymän pitäisi olla valmis. Ennen tarkisteiden tekemistä voidaan esimerkiksi odottaa, että taulukkoon ilmestyy rivejä tai että jonkin painikkeen aktiivisuuden määrittävä CSS-tyyliluokka muuttuu.

Testauksessa pitää jollakin tavalla rajata se, kuinka syvälle testit ulottuvat. Mitä syvemmälle testit ulottuvat, sitä epävakaammiksi testit tulevat verkon epävakauden ynnä muiden ongelmien vuoksi. Tästä johtuen voi olla kannattavaa rajata testeissä järjestelmään liittyvien kolmansien osapuolien toiminnallisuutta pois, sillä niiden toiminnan testaaminen on muiden vastuulla. Pois rajatuille integraatioille voidaan tehdä testejä varten minimaalinen mock-toteutus, joka muistuttaa toiminnaltaan alkuperäistä järjestelmää. Jos sovelluksen testaamisen kannalta mock-toteutus ei jostakin syystä ole riittävä, on päästä päähän -testausta tehtävä myös integraation yli. Näin ollen päästä päähän -testaus kuvan 1 esimerkkiarkkitehtuurin tapauksessa tarkoittaisi selaimesta sovelluspalvelimen kautta tietokantaan tai integraatioihin ja takaisin.

Kun puhutaan web-sovelluksien päästä päähän -testiautomaatiosta, pohjateknologiana on nykyisin usein Selenium 2, jota kutsutaan usein Selenium WebDriver -nimellä[3]. Selenium WebDriver on yhdistelmä kahdesta aiemmasta projektista, jotka pyrkivät ratkaisemaan web-selaimessa toimivan sovelluksen testaamisen ongelmaa. Projektit olivat vuonna 2004 syntyneet Selenium ja vuonna 2006 alkunsa saanut WebDriver. [11]

Siinä missä Seleniumin toiminta perustui siihen, että se ajoi skriptejä selaimen JavaScript-moottorissa, WebDriver keskusteli suoraan selaimen kanssa käyttäen selaimen ja käyttöjärjestelmän välistä rajapintaa. Selenium-projekti oli aiemmin törmännyt suuriin ongelmiin sen takia, että selaimet monimutkaistuivat ja niiden tietoturvaominaisuudet paranivat ajan myötä, minkä vuoksi selaimen JavaScript-hiekkalaatikon kautta operointi vaikeutui. Näiden projektien yhdistelmä Selenium WebDriver pyrkii yhdistämään molempien projektien hyvät puolet: WebDriverin suora keskusteluyhteys selaimen kanssa ja Seleniumin laajempi tuki eri selaimille. Projektit yhdistyivät vuonna 2008. [11]

Selenium-työkalupakkiin kuuluu edellä mainittujen lisäksi Selenium IDE ja Selenium-Grid. Selenium IDE on nopeaan prototyyppien rakentamiseen tarkoitettu työkalu,

[3] Muitakin lähestymistapoja on. Esimerkkinä avoimen lähdekoodin selaimen pohjalta kehitetty testaustyökalu Cypress.io [29].

jolla voi ilman ohjelmointitaitoa luoda testiautomaatiota. Testien luonti tapahtuu taltioimalla käyttäytymistä käytön aikana Firefox-selaimeen asennettavan liitännäisen avulla. Selenium-Grid puolestaan mahdollistaa hajauttamisen useisiin ympäristöihin, mistä johtuen testien ajamiseen kuluu lyhempi aika, koska testejä voidaan ajaa rinnakkain. [11]

Selenium WebDriverin päälle on rakennettu paljon testausohjelmistokehyksiä eri ohjelmointikielillä. Lähes jokaiselle ohjelmointikielelle, joilla tehdään web-sovelluskehitystä, on omat kehyksensä, joiden on tarkoitus yksinkertaistaa testien kirjoittamista. Esimerkkeinä Selenium WebDriveria hyödyntävistä työkaluista JavaScript-ohjelmointikielellä toimiva testaustyökalu WebdriverIO ja Python-kielellä toteutettu Robot Framework [12] [13].

Lisäksi on olemassa käyttöliittymälogiikkaa varten suunniteltuja ohjelmistokehyksiä, joille on rakennettu oma päästä päähän -testiautomaatiokehys. Näistä esimerkkinä on AngularJS-ohjelmistokehystä varten rakennettu Angular Scenario Runner, jonka toiminta perustui Seleniumin aiemman version tavoin selaimen JavaScript-hiekkalaatikon window-objektin kautta operoimiseen. Angular Scenario Runner on vanhentunut, eikä sen käyttöä suositella. [14]

Tässä työssä päästä päähän -testiautomaatiolla tarkoitetaan sellaisia testejä, jotka on rakennettu suoraan Seleniumin työkaluilla tai niitä apunaan käyttävillä päästä päähän -testausohjelmistokehyksillä tai jollakin muulla selaimen kautta operoivalla testauskehyksellä, kuten Angular Scenario Runner. Kyse on siis selaimen kautta suoritettavista testeistä, joissa on mukana jokin sovellukselle olennainen taustajärjestelmä, jonka kanssa selain keskustelee.

# 3. TUTKIMUSASETELMA

Tässä luvussa kerrotaan ensin tutkimuksessa käytetyistä menetelmistä ja teoriasta, jota tutkimuksessa hankitun tietoaineiston hyödyntämisessä on käytetty. Lisäksi kerrotaan haastattelututkimuksen toteutuksesta ja lopuksi listataan haastattelussa kysytyt kysymykset.

## 3.1 Tutkimusmenetelmät

Aiheeseen liittyvien ongelmakohtien ja kehitysideoiden kartoittamiseksi suoritettiin aineiston hankinta. Aineiston hankkimiseksi tutkielman osana suoritettiin haastattelututkimus, jolla pyrittiin löytämään päästä päähän -testiautomaatioon liittyviä ongelmakohtia ja keksimään kehitysideoita, joilla web-sovelluksien selaimesta käsin suoritettavaa testiautomaatiota voitaisiin parantaa. Tutkimuksen taustalla yhtenä tutkimusmenetelmänä on ankkuroitu teoria (grounded theory), joka on tutkimusstrategia, joka edellyttää sitoutumista tietynlaiseen etenemiseen ja samalla aineiston analysointimenetelmä, jota voidaan käyttää laadullisten tutkimusten osana. ”Ankkuroidun teorian tavoitteena on tuottaa empiiristä aineistoa tutkimalla teoriaa tai käsitteellisiä malleja aihepiiristä, josta ei ole vielä tuotettu jäsenneltyä tietoa tai josta ei ole olemassa vakiintuneita teorioita” [15]. Tiivistettynä ankkuroitu teoria on ajatus siitä, että teoria piilee aineistossa [16].

Haastattelututkimuksessa pyrittiin mittaamaan sekä kvalitatiivisia että kvantitatiivisia aiheeseen liittyviä ominaisuuksia. Esimerkkeinä näistä laadullisena ominaisuutena testien kirjoittamisen hyötyjen ja haittojen kuvaileminen ja määrällisenä ominaisuutena testien ajamiseen kuluva aika. Kokemuksiin liittyvän kvalitatiivisen tutkimusmenetelmän käyttöä osana ohjelmistotuotannon ongelman käsittelyä on tutkinut Seaman todeten, että paras tulos on mahdollista saavuttaa yhdistelemällä kvalitatiivisia ja kvantitatiivisia menetelmiä. Kvalitatiivinen data on rikkaampaa kuin kvantitatiivinen data, joten kvantitatiivisien menetelmien käyttö kasvattaa datasta saatavan informaation määrää. Kvalitatiivinen tieto kasvattaa myös datan moninaisuutta parantaen tuloksien luotettavuutta. [17]

Haastattelututkimuksessa data kerättiin talteen tekemällä mahdollisimman kattavat muistiinpanot haastattelujen aikana. Muistiinpanot tehtiin taulukkomuotoon, jossa kysymykset olivat vaakariveillä ja vastaukset sarakkeissa siten, että käsittelyssä olleille projekteille ja luonteeltaan yleisille vastauksille oli omat sarakkeensa. Yleisille vastauksille oli paikkansa siksi, että haastatelluilla oli ajatuksia, jotka koskivat kaikkia menneitä projekteja, tai sitten tietoa oli saatu muista lähteistä, esimerkiksi niin sanottuna perimätietona.

Haastattelujen kulku ja niissä ilmi tullut tieto olisi saatu paremmin talteen myöhempää tarkastelua varten joko kuvaamalla tai äänittämällä haastattelut, mutta näitä keinoja haluttiin tietoisesti välttää, jotta haastateltujen näkemykset saataisiin mahdollisim-

man rehellisesti selville [18]. Näin tehtiin siksi, että aihepiiriin liittyy muun muassa kokemuksen mukanaan tuomia voimakkaita tunteita sekä puoleen että toiseen [7]. Myös tästä johtuen yksikään tässä työssä esitetty näkemys ei ole yksilöitävissä tiettyyn haastateltuun ja haastateltujen nimet ovat mainittu liitesivulla A satunnaisessa järjestyksessä.

Monissa ohjelmistotuotannon tutkimuksissa data voidaan jakaa tapauksiin. Tässä tutkimuksessa data jaettiin tapauksiin projekteittain. Jokainen haastatteluissa ilmi tullut projekti, jossa oli tehty päästä päähän -testiautomaatiota, oli oma tapauksensa. Näistä projekteista saatuihin tietoihin sovellettiin jatkuvan vertailun menetelmää (constant comparison method), jossa muistiinpanoja ja merkintöjä käydään läpi tutkimuksen edetessä aika ajoin ja saatujen tietojen pohjalta tehdään vertailua niissä toistuvasti esiintyvien avainsanojen ja ilmiöiden välillä. Lisäksi projektien välillä tehtiin ristianalyysia (cross-case analysis) siitä, millaisia kokemuksia päästä päähän -testiautomaation soveltamisesta oli eri projekteissa. [17]

## 3.2 Haastattelututkimuksesta

Haastattelut suoritettiin Solita Oy:n tiloissa Tampereen toimistolla kevään 2016 aikana. Haastattelustrategiana oli puolistrukturoitu haastattelu. Kokemuksiin pohjautuvaa puolistrukturoitua haastattelua osana ohjelmistotuotannon ongelmien tutkimista ovat käsitelleet Hove ja Anda todeten, että puolistrukturoidun haastattelun pitäminen on hintavaa ja että saadun datan laatu riippuu haastattelusuorituksen onnistumisesta. Haastattelusuorituksen onnistumiseen voi vaikuttaa haastattelun suunnittelulla, jonka keskiössä olevat neljä pääaluetta ovat tarvittavan vaivan määrän arviointi, tarvittavien aihetta koskevien tietojen selvittäminen etukäteen, haastattelijan ja haastateltavan välisen sujuvan kanssakäymisen varmistaminen ja sopivien työkalujen käyttäminen. [18]

Haastattelua varten oli laadittu etukäteen kysymykset, joiden mukaan haastattelun oli tarkoitus edetä, mutta tilanne salli myös keskustelun käymistä. Esimerkiksi jatkokysymyksiä saatettiin esittää tilanteen mukaan. Myös kysymysjärjestys saattoi vaihdella hieman sen mukaan, miten haastateltava luontaisesti toi asiat esille.

Haastateltavien johdattelemista vältettiin kaikin keinoin. Mahdollisella haastattelun osana käydyillä keskusteluilla ei haluttu johdatella haastateltavaa mihinkään suuntaan, vaan tarkoituksena oli elävöittää ja kontrolloida tilannetta siten, että vastauksien kattavuus saadaan riittäväksi haastattelulle varatun tunnin aikana [19].

Eräät asiat, kuten projekteissa käytössä olleet teknologiat, eivät tulleet selväksi aina täysin haastattelun aikana. Epäselviksi jääneisiin asioihin kysyttiin jälkeenpäin tarkennuksia pikaviestimien avulla.

### 3.2.1 Haastattelun kohderyhmän valinta

Haastateltavien löytämiseksi lähetettiin sähköpostilla noin 250 henkilölle kutsu haastatteluun. Lähtökohtaisesti haastattelututkimukseen pyrittiin saamaan erityyppisiin tehtäviin

erikoistuneita henkilöitä – ohjelmistosuunnittelijoita, projektipäälliköitä, käytettävyysasiantuntijoita ynnä muita. Kohderyhmää ei haluttu turhaan rajata liian tiukasti, sillä pyrkimyksenä oli saada mahdollisimman totuutta vastaava otanta erilaisissa projekteissa ja erilaisissa tehtävissä työtä tehneiltä henkilöiltä. Suurin osa haastateltavista löydettiin kutsusähköpostin ja vielä myöhemmin muistutuksena lähetetyn sähköpostin avulla. Pieni osa tuli mukaan muuta kautta levinneen tiedon myötä.

Haastatteluun osallistui pääasiassa vain kokeneita ohjelmistosuunnittelijoita tai arkkitehtejä, joilla oli omakohtaisia kokemuksia päästä päähän -testiautomaation tekemisestä menneissä tai tutkimuksen tekohetkellä meneillään olleissa web-sovellusprojekteissa. Muut eivät ehkä kokeneet kuuluvansa päästä päähän -testiautomaation tekemisestä kokemusta saaneiden tai asiaan muuten perehtyneiden joukkoon, kuten kutsusähköpostissa toivottiin. Periaatteessa haastattelututkimukseen oli kutsuttu myös ohjelmointityötä tekemättömät henkilöt, mutta omakohtainen kokemus päästä päähän -testiautomaation rakentamisesta oli vaatimuksena haastatteluun osallistumiselle.

### 3.2.2 Haastattelukysymykset

Haastattelun kesto oli tunti ja siinä ajassa käytiin läpi 20 alla lueteltua kysymystä. Osalla haastateltavista oli tarjota vastauksia kahden eri projektin osalta. Näissä tapauksissa yhden tunnin mittainen tuokio osoittautui haasteelliseksi, mutta tehokkaalla ajankäytön tarkkailulla vastaukset ehdittiin keräämään kaikkiin kysymyksiin.

- Millaisissa projekteissa olet tehnyt päästä päähän -testiautomaatiota?
- Millaisilla teknologioilla?
- Tehdäänkö jokaiselle ominaisuudelle testit?
- Kuinka tarkasti ominaisuudet testataan? Testataanko eri selaimilla tai eri ympäristöissä?
- Onko mielestäsi tarpeellista testeissä matkia oikeaa käyttäjää?
- Onko tarkoitus varmistaa, että käyttöliittymä on ulkoasultaan oikeanlainen?
- Kauanko testien ajamisessa menee yleensä aikaa?
- Kuinka usein testit ajetaan?
- Mikä on CI-palvelimen rooli testauksessa?
- Missä vaiheessa testit kirjoitetaan?
- Kauanko testien kirjoittamiseen käytetään aikaa?
- Onko testejä mielestäsi helppo kirjoittaa käyttämilläsi teknologioilla?
- Miten kuvailisit päästä päähän -testien ylläpidettävyyttä?
- Millaisia hyötyjä päästä päähän -testiautomaatiolla on saavutettu?
- Millaisiin ongelmiin olet törmännyt päästä päähän -testiautomaation kanssa?
- Kuinka kauan arvioisit, että projektissasi kestää CI-palvelimella havaitusta hajonneesta testistä päästään tilanteeseen, jossa kaikki testit menevät läpi?

- Miten näkisit päästä päähän -testiautomaation tekemisen vaikuttavan liiketoimintaan?
- Onko päästä päähän -testiautomaatiosta enemmän hyötyä kuin haittaa?
- Miten päästä päähän -testiautomaation tekemistä voitaisiin tehostaa?
- Onko jotakin muuta vielä mielessä?

Haastattelukysymyksillä haluttiin pääpiirteittäin saada selville, miten ja millaisia testejä kirjoitetaan, miten aika käsitteenä liittyy päästä päähän -testiautomaatioon, millaista on testien kirjoittaminen ja ylläpidettävyys, onko testien tekemisestä hyötyä vai haittaa ja miten niitä voisi jatkossa tehostaa.

Kysymyksien muotoilu haluttiin sellaiseksi, ettei niihin olisi helppoa vastata pelkästään kyllä tai ei. Tämä ei tosin yhdessäkään haastattelussa ollut ongelma, vaan kysymyksiin vastattiin poikkeuksetta hyvin laajasti. Haastateltujen kokemus ja ammattitaitoisuus näkyivät vastauksissa positiivisella tavalla.

Kysymykset pyrittiin valitsemaan siten, että päästä päähän -testiautomaatioon liittyvää problematiikkaa käytäisiin haastattelussa mahdollisimman laajanäköisesti läpi. Kysymyksien järjestys suunniteltiin siten, että haastateltavat ehtisivät palauttaa mieleensä päästä päähän -testiautomaation rakentamiseen liittyvät seikat ennen kaikista mielenkiintoisimpiin kysymyksiin siirtymistä: liiketoiminta, hyödyt, haitat ja mahdollinen tehostaminen.

# 4. HAASTATTELUTUKIMUKSEN TULOKSET

Haastattelututkimukseen osallistui seitsemän henkilöä, joista suurin osa oli pitkään ohjelmistokehitystyötä tehneitä ohjelmistosuunnittelijoita, arkkitehtejä tai vanhempia ohjelmistosuunnittelijoita. Kaikki haastatellut toimivat itse sovelluskehittäjinä projekteissaan. Mukaan mahtui niin käyttöliittymän kuin taustajärjestelmän kehittämiseen keskittyviä kehittäjiä. Käytännössä lähes kaikki kuitenkin tekivät kehitystä sovelluksen kaikilla osa-alueilla.

Haastattelututkimukseen osallistui henkilöitä kaikkiaan kahdeksasta projektista. Taulukossa 1 on esitelty, kuinka haastatellut henkilöt jakautuivat eri projekteihin. Projektit on numeroitu siinä järjestyksessä kuin ne haastatteluissa esiintyivät.

Taulukko 1. Haastatellut henkilöt projekteittain.

| Projekti | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Haastatellut henkilöt | H1 | H1 | H2 | H2 | H3, H7 | H3 | H4 | H5, H6 |

Projekteissa 5 ja 8 on edustettuna kummassakin kahden eri henkilön näkökulmat. Haastatelluilta H1, H2 ja H3 on saatu vastaukset kysymyksiin kahden eri projektin osalta. Muilla haastatelluilla vastaukset kohdistuvat yhteen projektiin. Kaikilta haastatelluilta saatiin myös yleisen tason vastauksia, jotka eivät suoraan kohdistuneet mihinkään määrättyyn projektiin.

Haastattelun piiriin kuuluneet projektit vaihtelivat toteutusteknologioiltaan jokseenkin paljon. Taulukossa 2 on esitelty projektien toteutusteknologiat pääpiirteittäin.

Taulukko 2. Toteutusteknologiat projekteittain.

| Projekti | Taustajärjestelmän toteutusteknologiat | Käyttöliittymän toteutusteknologiat | Päästä päähän -testiautomaatioteknologiat |
|---|---|---|---|
| 1 | PHP | Vanilla JS, jQuery, Backbone.js | Selenium IDE, Selenium WebDriver |
| 2 | .NET | AngularJS, Ramda JS, Deku, D3.js | Protractor, Chai Assertion Library, Mocha |
| 3 | Java | JavaServer Pages, Backbone.js (osa toteutuksesta) | REST-assured, JBehave, FluentSelenium |

| | | | |
|---|---|---|---|
| 4 | Java | AngularJS | Protractor, Jasmine |
| 5 | Clojure | AngularJS | Clojure WebDriver |
| 6 | Java | AngularJS | Angular Scenario Runner, Protractor |
| 7 | Clojure | Knockout.js | Robot Framework |
| 8 | Java | Wicket | Selenium WebDriver, JBehave |

Lähes kaikissa projekteissa arkkitehtuurillinen lähestymistapa oli se, että sovelluspalvelin on tilaton eli se ei muista aiempia pyyntöjä. Tämän lisäksi sovelluspalvelin tarjoaa käyttäjän selaimelle JavaScript-sovelluksen, joka toteuttaa käyttöliittymälogiikan ja keskustelee sovelluspalvelimen rajapinnan kanssa. Tällaisissa ratkaisuissa sovelluslogiikkaa on paljon asiakkaan puolella. Poikkeuksena projekti 8 on toteutettu Java-kielellä toteutetulla Wicket-käyttöliittymäkirjastolla, jossa HTML-dokumentit rakennetaan palvelinpäässä valmiiseen selaimessa esitettävään muotoon. Samoin projektissa 3 oli alun perin käytetty JavaServer Pages -palvelinpään tekniikkaa, jota oli myöhemmin lähdetty uudistamaan Backbone.js-kirjaston avulla selainpään JavaScript-sovellusta kohti.

Palvelinpäässä sijaitsevan käyttöliittymälogiikan Java-toteutuksissa, projekteissa 3 ja 8, on kummassakin käytetty Selenium WebDriveria hyödyntävää JBehave-kirjastoa, joka kehottaa kehittäjää kirjoittamaan testit helppolukuiseen käyttäjän käyttäytymistä mukailevaan muotoon. Projektissa 5 selaintestit on kirjoitettu ohuen Clojure-kielellä toteutetun Selenium WebDriver -kääreen avulla. Toisessa Clojure-projektissa selaintesteille on valittu alun perin Nokia Networksin kehittämä yleiskäyttöinen testauskehys Robot Framework, joka myös ohjaa JBehaven tavoin abstrahoimaan tekniset yksityiskohdat piiloon varsinaiselta käyttötapauksen kuvaukselta määrittelemällä avainsanoja, jotka ovat funktioiden kaltaisia kääreitä alemman abstraktiotason toiminnoille. Esimerkiksi *Input username* voisi olla avainsana, jolle parametrina voisi antaa kirjautumissivulle syötettävä käyttäjänimi.

Puolessa projekteista selainpään käyttöliittymän toteuttavana teknologiana oli aikanaan hyvin suosittu Googlen kehittämä AngularJS-ohjelmistokehys. Näistä projekteista kolmessa oli käytössä Protractor, joka on Selenium WebDriverin päälle rakennettu testauskehys, joka on suunniteltu nimenomaan AngularJS-sovellusten testaamiseen JavaScript-kielellä. Protractor-testeissä ei yleensä ole JBehaven kaltaista tarina-abstraktiota, vaan tekniset yksityiskohdat pyritään piilottamaan testattavia sivuja kuvastavien Page-olioiden alle. Esimerkiksi sisäänkirjautumissivun loginPage-olio voisi tarjota inputUsername-funktion, joka kätkee sisällensä HTML-elementin hakemisen ja sen arvon asettamisen selaimen dokumenttioliomalliin.

Projekti 1 poikkeaa kaikista muista sillä tavalla, että siinä osa selaintesteistä tehtiin kirjoittamisen sijaan tallentamalla testin tekijän käyttäytymistä selaimessa myöhemmin toistettavaan muotoon. Näiden lisäksi kyseisessä projektissa oli myös Selenium 2 -testejä jossakin määrin.

Projektien teknologiat vaikuttavat jonkin verran siihen, kuinka yksinkertaista käyttöliittymälogiikan testaaminen on. Vaikkakin kyseessä on kaksi hyvin erilaista lähestymistapaa, Wicket ja AngularJS ovat kummatkin suunniteltu siten, että käyttöliittymälogiikalle on yksinkertaista tehdä yksikkötestejä. Kummassakin käyttöliittymälogiikkaa voi testata ilman selaimen ja taustajärjestelmän välistä keskustelua. Wicketissä yksikkötestejä voi tehdä ilman selainriippuvuutta ja AngularJS:ssä ilman selain- tai taustajärjestelmäriippuvuutta. Muissa projekteissa käyttöliittymän testaaminen oli joko osittain tai kokonaan selaimen läpi tehtävien päästä päähän -testien varassa. Myös näissä projekteissa käyttöliittymän yksikkötestaaminen olisi ollut mahdollista, mutta tutkimuksessa ei selvinnyt syytä miksi näin ei oltu tehty.

Haastatelluilta kysyttiin millaisissa projekteissa he ovat tehneet päästä päähän -testiautomaatiota. Kysymyksen ohessa pyrittiin kartoittamaan projektien kokoluokkaa arvioimalla projekteissa työskennelleiden kehittäjien määrää ja aktiiviseen kehittämiseen käytettyä aikaikkunaa vuosina. Näiden kertolaskulla saatiin indeksi, jolla voi hyvin suurella varauksella yrittää vertailla haastattelun piiriin kuuluneiden projektien kokoa toisiinsa nähden. Tuloksien tarkkuuteen vaikuttaa muun muassa se, että haastatellut eivät olleet aiemmin tulleet ajatelleeksi asiaa ja se, että osa projekteista oli loppunut jo vuosi tai pari sitten, joten asian muistaminen oli hankalaa. Tulokset on esitetty taulukossa 3. Jos haastateltu antoi vastauksena lukuarvovälin, taulukkoon on merkitty niiden keskiarvo. Jos siis haastateltu on esimerkiksi kertonut projektissa työskennelleen 6–7 kehittäjää, taulukkoon on merkitty keskiarvo 6,5.

Taulukko 3. Projektien kokoarviointia.

| Projekti | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Kehittäjien lukumäärä | 4,5 | 3 | 2 | 3 | 6,5 | 2 | 8 | 12 |
| Aktiivinen kehittäminen vuosina | 3 | 3 | 4 | 2 | 2 | 3 | 4 | 3 |
| Kokoindeksi | 13,5 | 9 | 8 | 6 | 13 | 6 | 32 | 36 |

Taulukosta käy ilmi se, ettei yksikään haastattelun piiriin kuulunut projekti ollut kovinkaan lyhytkestoinen, sillä jokainen projekti oli kestänyt arviolta noin kaksi vuotta tai enemmän. Projektit 7 ja 8 ovat selvästi kokoluokaltaan suurimmasta päästä. Projektit 1 ja 5 ovat tutkimuksen kannalta keskikokoisia ja loput niitä pienempiä.

## 4.1 Testien suorittamiseen kuluva aika

Haastattelututkimuksessa selvitettiin, kuinka kauan testien suorittamiseen kuluu yleensä aikaa. Tuloksina saadut ajat tarkoittavat projektikohtaisella jatkuvan integraation (CI) palvelimella testien suorittamiseen kuluvaa aikaa projektia 2 lukuun ottamatta. Projektissa 2 testejä ajettiin vain kehittäjien omilla koneilla, vaikkakin tahtotila oli saada ne tulevaisuudessa suoritukseen jatkuvan integraation palvelimelle.

On selvää, että tuloksiin vaikuttaa ajoympäristön laskentateho, kuorma, verkkoviiveet ynnä muut seikat, mutta tässä tutkimuksessa kiinnostavinta oli se, kuinka kuluva aika vaikuttaa projektityöskentelyyn ja mitä tekoja sen parantamiseksi olisi mahdollista tehdä. Taulukossa 4 on esitetty testien suorittamiseen kuluva aika projekteittain.

Taulukko 4. Testien suorittamiseen kuluva aika projekteittain

| Projekti | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Testien suorittamiseen kuluva aika (min) | 10 | 3 | 40 | 30 | 20 | 5 | 110 | 60 |

Haastatteluissa pyrittiin selvittämään, kuinka laaja-alaisesti ja tarkasti ominaisuudet haluttiin testata eri projekteissa. Projekteissa 2 ja 6 oli päätetty tehdä testit vain kaikista kriittisimmille ominaisuuksille eli niin sanotut savutestit. Tämä näkyy selvästi myös testien suoritusajoissa, jotka ovat näiden projektien kohdalla 3 ja 5 minuuttia. Kaikissa muissa projekteissa testejä pyrittiin tekemään laajemmin siten, että ainakin perustoiminnoille oli testit.

Projektin 7 verrattain pitkä testien suoritusaika johtuu kolmesta seikasta. Ensinnäkin projekti on laajuudeltaan vertailuryhmän suurimmasta päästä ja toisekseen siinä pyrittiin siihen, että lähes jokaista ominaisuutta koskee jollakin tavalla päästä päähän -selaintesti, mikä puolestaan johtui osittain siitä, ettei käyttöliittymäkoodia testannut mikään muu testiautomaatio. Näin ollen projektissa 7 päästä päähän -testiautomaatiolla tehtyjä testejä on paljon, ehkä eniten vertailuryhmässä.

Osassa projekteista oli tietoisesti pyritty rajoittamaan suoritukseen kuluva aika johonkin määrättyyn rajaan, jotta palautesykli pysyisi kohtuullisena. Ongelmaksi muodostui tällöin kuitenkin priorisoinnin vaikeus. Oli vaikea päättää, mikä testitapaus on tärkeä ja mikä ei. Projektissa 8 oli kokeiltu jakaa testejä niin sanottuihin nopeaan ja hitaaseen ajoon. Tästä oli kuitenkin jouduttu luopumaan, koska jaottelu näiden kahden välillä oli vaikeaa.

Yleisesti ottaen päästä päähän -testien suorittamisessa kului aikaa kohtuullisen pitkään. Projekteja 1, 2 ja 6 lukuun ottamatta ei voi puhua enää lyhyen kahvitauon mittaisesta kestosta. Näissä projekteissa kuluva aika vaihteli 20 minuutista aina lähes kahteen tuntiin.

Kun haastattelussa kysyttiin päästä päähän -testiautomaation tehostamiskeinoja, haastatellut H2, H3, H4 ja H6 mainitsivat testien ajamisen rinnakkaistamisen yhtenä keinona. Käytännössä jokainen haastatelluista oli sitä mieltä, että päästä päähän -testit ovat hitaita ja että niiden nopeammasta suorittamisesta olisi hyötyä. Näin sovelluskoodiin tehtyjen muutoksien jälkeen olisi nopeampaa saada palautetta siitä, rikkoontuiko jokin aiemmin toiminut ominaisuus muutoksen yhteydessä. Haastatellut H3 ja H5 olivat miettineet testien rinnakkaistamista, mutta projekteilla ei ole ollut resursseja sen toteuttamiseen, joten se oli jäänyt tekemättä.

## 4.2 Testien suorituksessa havaitun virheen korjaaminen

Tutkimukseen osallistuneiden henkilöiden projektit vaihtelivat sen osalta, ajettiinko testit automaattisesti heti versionhallintaan tehtyjen muutoksien jälkeen vai ei. Kahta projektia lukuun ottamatta kaikissa tutkimukseen osallistuneiden henkilöiden projekteissa oli käytössä jatkuvan integraation menetelmä (continuous integration), jossa ideana on hajautettua versionhallintaa hyödyntäen koota kehittäjien lähdekoodiin tekemät muutokset yhteen, kääntää sovellus ajettavaksi binääritiedostoksi ja suorittaa testit sovelluksen viimeisintä kehitysversiota vasten. Tähän tarkoitukseen on yleensä valjastettu erillinen palvelin, joka tutkii tasaisin väliajoin, onko sovelluksesta tullut versionhallintaan uudempi versio. Tutkimuksen suurimmissa projekteissa tätä menetelmää oli vielä tuettu siten, että projektiryhmän tiloihin oli sijoitettu radiaattoriksi kutsuttu näyttö, jonka tarkoitus oli esittää mahdollisimman reaaliaikaisesti CI-palvelimen suorittamien käännösten ja testien tulokset.

Projekteissa 2 ja 6 testit suoritettiin ainoastaan kehittäjien omilla koneilla kehittäjän omasta halusta. Näissä projekteissa virheen havaitseminen on kiinni siitä, että kehittäjät muistavat ja haluavat suorittaa testit riittävän usein. Projektissa 6 tästä ei ollut aiheutunut ongelmia lähinnä testien luonteen takia, sillä niitä suoritettiin öisin vikasietoisuuden testaamisen merkeissä. Projektissa 2 puolestaan tämä oli vaikuttanut heikentävästi virheiden havaitsemiseen. Joskus oli unohduksen vuoksi voinut käydä siten, että testien rikkoontumisesta kuullaan vasta toiselta kehittäjältä. Tästä johtuen korjaamiseen saattoi pahimmillaan kulua päiviä.

Haastatelluilta kysyttiin, kuinka kauan he arvioivat, että CI-palvelimella havaitusta virheestä kestää päästä siihen tilanteeseen, että ongelma on korjattu. Kesto riippuu arvatenkin ongelman suuruudesta, mutta mielenkiintoista on löytää syitä siihen, miksi toisinaan ongelman korjaamiseen kuluu turhauttavan paljon energiaa. Käytännössä kaikki vastanneet olivat sitä mieltä, että korjaukseen kuluu aikaa yleensä puolesta tunnista pariin tuntiin, mutta pahimmillaan ongelman selvittämisessä voi tuhraantua kokonainen päivä.

Projekteissa 5, 7 ja 8 oli ongelmana se, että luottamus jatkuvan integraation palvelimella suoritettujen testien tulokseen kärsi siksi, että testien suorituksessa oli epävakautta. Toisin sanoen testien suorituksesta voitiin saada eri tulos eri ajokerralla ilman, että sovelluksen lähdekoodia tai testejä muutettiin. Testeissä havaitun ongelman korjaamiseen tämä vaikutti siten, että tuloksessa näkyvästä virheestä ei välttämättä oltu juurikaan kiinnostuneita tai päädyttiin odottamaan, menisivätkö testit läpi seuraavalla ajokerralla. Testien vakauteen liittyvää ongelmaa käsitellään omana kokonaisuutenaan tässä työssä.

Haastateltu H6 totesi, että testien korjaaminen olisi helpompaa, jos testiraportista näkisi paremmin, mistä syystä virhe aiheutui. Pelkkä testiautomaatiokehyksen antama virheilmoitus ei yleensä vielä yksistään anna kovin tarkkaa kuvaa ongelmasta. Virheilmoitus voisi olla esimerkiksi, ettei kuvailtua käyttöliittymäelementtiä ollut näkyvissä testitapausta suoritettaessa. Korjaamisen kannalta mielenkiintoista on lähinnä se, miksi kyseinen käyttöliittymäelementti ei ollut näkyvissä. Ainakin kahdessa projektissa oli käytössä työkalu, joka otti virhetilanteesta aina kuvankaappauksen. Sen avulla oli toisinaan keksitty ongelman syy. Aiemman esimerkin tapauksessa kuvakaappauksesta olisi voinut olla nähtävissä, ettei käyttöliittymäelementti ollut näkyvissä, koska testin suoritus oli väärässä näkymässä.

Yhteen vedettynä päästä päähän -testiautomaation korjaamisessa on haastateltujen projekteissa ollut pahimmillaan kolme eri vaihetta: virheen huomaaminen, siihen reagoiminen ja korjaaminen. Tehokkaimmillaan virhe huomataan ja se korjataan. Reagointia odottava vaihe välissä on tarpeeton ja se johtuu yleensä luottamuksen puutteesta saatua testitulosta kohtaan, koska testien suorituksessa on epävakautta muun muassa vaihtelevista verkkoviiveistä tai -ongelmista johtuen. Päästä päähän -testiautomaation kanssa työskentelyä voisi tehostaa kasvattamalla saadun testituloksen luotettavuutta.

## 4.3 Testien ylläpidettävyys

Tutkimuksessa pyrittiin kartoittamaan sitä, kuinka ylläpidettäviä testit ovat ja millä keinoilla niiden ylläpidettävyyttä on kyetty parantamaan. Testien ylläpidettävyydellä tarkoitetaan sitä, kuinka yksinkertaisesti tai kuinka pienellä muokkauksella olemassa olevasta testistä saadaan sellainen, että se testaa ohjelmaa siihen tehdyn muutoksen jälkeen vähintään yhtä hyvin kuin ennen muutosta. Hyvin siinä mielessä, että muuttunutta ohjelman osaa testataan yhtä kattavasti kuin aiemmin. Testien ylläpidettävyyden arviointi on subjektiivista ja siihen vaikuttavat monet tekijät kuten projektin vaihe, muutoksien suuruus, osaaminen, aikataulupaineet ja niin edelleen. Tästä johtuen on mielekästä keskittyä erityisesti keinoihin, joilla ylläpidettävyyttä on saatu parannettua.

Testien ylläpitäminen koettiin haastateltujen joukossa jokseenkin haasteelliseksi. Positiivista oli, että ylläpidettävyyteen oli käytännössä kaikissa projekteissa kiinnitetty huomiota ja sitä oli kyetty usein parantamaan. Tiivistettynä eniten ylläpito-ongelmia tuottivat testien kirjoittamiskäytännöt ja teknologiavalinnat.

Projektissa 1 päästä päähän -testiautomaation tekemiseen käytettiin Internet-selaimeen asennettavaa Selenium IDE -laajennosta, jolla testit tehtiin nauhoittamalla testaajan käyttäytymistä. Kyseisessä projektissa testit oli nauhoitettu sovellukseen tehtyjen muutoksien jälkeen uudestaan. Selenium IDE mahdollistaa nauhoitettujen testien muokkaamisen selainlaajennoksen graafisessa taulukkonäkymässä. Toinen vaihtoehto on muokata käsin testistä tallennettua lähdekoodimerkkausta [11]. Haastattelussa ei selvinnyt, oliko testit nopeampaa vain nauhoittaa suoraan uudelleen vai olivatko muutokset aina niin suuria, ettei nauhoitetun testin muokkaaminen olisi ollut mielekästä. Joka tapauksessa tässä projektissa selaintestien ylläpidettävyys koettiin surkeaksi. Sama haastateltava H1 koki myös projektin 2 päästä päähän -testiautomaation surkeaksi testien verrattain vähäisestä määrästä huolimatta. Tässä projektissa JavaScript-kielellä kirjoitettujen testien koodin laadussa olisi ollut parantamisen varaa.

Haastateltava H2 otti kantaa projektien 3 ja 4 päästä päähän -testiautomaation ylläpidettävyyteen. Projektissa 3 ei hitaan jatkokehityksen ja ylläpidon aikana ollut juurikaan ongelmia testien ylläpidon kanssa lukuun ottamatta sitä, että joidenkin testien vakautta olisi voinut parantaa. Projektissa 4 testit oli pyritty kirjoittamaan siten, ettei esimerkiksi pieni syötekentän paikan muuttuminen vaatisi suuria muutoksia testeissä. Tätä varten testejä varten oli kirjoitettu sivunäkymää kuvaavia, teknisiä yksityiskohtia abstrahoivia Page-olioita, joiden johdosta pienet muutokset kohdistuvat yhteen paikkaan. Tästä on ainakin hyötyä silloin, kun useat testitapaukset käyttävät yhden sivun samaa käyttöliittymäelementtiä. Käyttöliittymäelementtiin tapahtuvan muutoksen yhteydessä voi riittää muuttaa yhtä Page-oliota usean testitapauksen sijaan.

Haastateltava H3 otti projektien 5 ja 6 osalta esille osaamisen, CSS-valitsimien kirjoittamisen ja Page-olion käyttöönoton. Projektien alussa testien ylläpidettävyyttä ei osattu ajatella riittävästi. Testien kirjoittamiskäytännöt kehittyivät projektien edetessä siten, että testien ylläpidettävyys parani. Yksi parannuskeino oli kirjoittaa CSS-valitsimet siten, että niissä viitataan CSS-tyyliluokkiin, jotka kuvailevat HTML-käyttöliittymäelementin ulkoasun. Perusteluna tälle oli se, että käyttöliittymäelementin ulkoasu pysyy yleensä vakaampana kuin sen sijainti näkymässä.

Projektissa 7 oli ainakin testien suorittamiseen kuluvaa aikaa silmällä pitäen luultavasti eniten päästä päähän -testiautomaatiota. Haastateltu H4 kuvaili ylläpitoa työlääksi, mutta sen helpottamiseksi oli kirjoitettu apufunktiokirjastoja, jotka abstrahoivat teknisiä yksityiskohtia ja ovat käyttökelpoisia ympäri sovellusta. Esimerkiksi pudotusvalikosta arvon valitsemista varten on funktio, jolle riittää kertoa elementin CSS-valitsin ja valittavaa arvoa kuvaava teksti sen sijaan, että testiä kirjoitettaessa tarvitsisi miettiä, millä tavoin haluttua arvoa saadaan klikattua pudotusvalikosta. Tässä projektissa käyttöliittymäelementtien paikantamisen helpottamiseksi oli annettu testien kannalta kiinnostaville HTML-elementeille test-id-attribuutin arvo, jolla elementtiin päästiin testeissä käsiksi tarkoitukseen tehdyn apufunktion avulla. Tästä testien kirjoittamiskäytännöstä on myös sellainen hyöty, että näkymän muuttaja huomaa jo HTML-merkkauksessa näkyvästä test-id-attribuutista, että kyseiseen kohtaan käyttöliittymää kohdistuu jokin päästä päähän -testi, jota voi joutua muuttamaan muutoksen yhteydessä.

Projektissa 8 oli kirjoitettu sivu- tai näyttöpohjaisia yleiskäyttöisiä osia helpottamaan ylläpitoa. Haastateltu H5 koki, että jotkin testikoodikokonaisuudet olivat paisuneet suuriksi ja että paremmilla apukirjastoilla testeistä olisi voinut saada yhtenäisempiä. Sekä H5 että H6 olivat lopulta sitä mieltä, että testien ylläpidettävyys koodin osalta oli lopulta riittävän hyvä.

Yksi merkittävä ongelma, jonka projektin 8 osalta haastatellut nostivat esille, oli kaikenlaisen integroidun ohjelmointiympäristön tuen puute JBehave-abstraktiolle, jonka tarkoitus on mahdollistaa luonnollista kieltä mukailevat testitapausten kuvaukset ja se, että testitapauksia voisivat lukea ja kirjoittaa myös muut kuin ohjelmointitaitoiset sovelluskehittäjät. Java-ohjelmointikielelle on rakennettu integroituja ohjelmointiympäristöjä (IDE, Integrated Development Environment), jotka koostuvat vähintään tekstieditorista ja kääntäjästä. Näissä on usein toiminto, jonka avulla on mahdollista siirtyä koodissa esiintyvän funktion kutsusta tai olion esiintymästä suoraan sen määritelmään. Esimerkiksi Java-ohjelmointikielellä kirjoitettujen päästä päähän -testien tapauksessa tästä on hyötyä silloin, kun kehittäjä haluaa siirtyä nopeasti katsomaan, miten testissä käytetty apukirjastofunktio on toteutettu. Kyseisessä projektissa kehittäjien keskuudessa käytössä olleessa integroidussa ohjelmointiympäristössä ei ollut tukea sille, että testitapauksen kuvailevasta tarinasta olisi voinut siirtyä nopeasti määritelmään eli toteutusyksityiskohtiin. Sen sijaan funktiomäärittely jouduttiin etsimään tekemällä vapaa tekstihaku projektin tiedostoihin tai etsimällä relevantteja tiedostonimiä hakemistopuusta.

Toinen ongelma projektissa 8 oli haastatellun H6 mukaan se, ettei ollut olemassa mitään järkevää tapaa ajaa yksittäistä testitapausta. Usein oli monta testitapausta yhdessä tarinassa. Jos yhtä testitapausta piti muuttaa, saattoi joutua muuttamaan samalla koko tarinaa, jotta ongelma korjaantuisi. Testit olivat suuria kokonaisuuksia. Ei ollut esimerkiksi yksittäistä Java-kooditiedostoa, jossa olevia testejä olisi voinut suorittaa joko yhdessä tai yksitellen.

Yleisesti ottaen, jos sovellukseen jouduttiin tekemään suuria muutoksia, vaadittiin testeihin yleensä myös suuria muutoksia. Pienet muutokset, esimerkiksi yksittäisten käyttöliittymäelementtien paikkojen muuttaminen näkymässä, eivät tuota ongelmia, jos tämä on testejä luotaessa osattu ottaa huomioon. Useat olivat sitä mieltä, että tavalla jolla HTML-elementit paikannettiin, oli suuri vaikutus testien ylläpidettävyyteen. Koettiin, että HTML-elementteihin kiinni pääsemiksesi kirjoitettujen CSS- tai XPath-valitsimien suunnittelulla on merkitystä. Elementtien paikantamisen lisäksi myös muut testien kirjoittamiskäytännöt kehittyivät projektien edetessä. Kuten haastateltu H3 totesi, testien ylläpidettävyyttä ei ehkä osattu ajatella riittävästi vielä projektin alussa. Jos ylläpidettävyyteen olisi kiinnitetty huomiota heti alusta saakka, lopputulos olisi voinut olla parempi.

## 4.4 Testien kirjoittamiskäytännöt

Päästä päähän -testiautomaation avulla tavoiteltu testikattavuus vaihteli projekteittain. Projekteissa 2 ja 6 testejä kirjoitettiin vain kaikista kriittisimmille ominaisuuksille. Lähes kaikissa muissa projekteissa periaate oli se, että tärkeimmille käyttötapauksille on ainakin

laillisia syötteitä käyttäviä testejä. Missään projektissa päästä päähän -testiautomaatiolla ei ollut tarkoitus testata useille ominaisuuksille sekä laillisia että laittomia tapauksia. Yleinen kommentti oli, että ainakin niin sanotuille perusominaisuuksille halutaan tehdä testejä. Useimmissa projekteissa pyrittiin lähtökohtaisesti siihen, että jokaista ohjelman näkymää koskisi jokin päästä päähän -testi. Näiden lisäksi osassa projekteista oli myös sellaisia testejä, jotka testasivat määrättyä ominaisuutta melko tarkasti, kuten esimerkiksi projektissa 4 oli lomakkeen syötevalidoinnille tehty. Projektissa 7 oli myös sellaisia päästä päähän -testejä, jotka käyttivät ulkoisia integraatioita esimerkiksi kirjautumiseen.

Kaikissa projekteissa päästä päähän -testiautomaatiota ei tehty heti toteutusprojektin alusta lähtien. Projekteissa 2 ja 8 päästä päähän -testit tulivat mukaan myöhemmin. Projektissa 2 ne otettiin mukaan vasta tuotantoon menemisen jälkeen ja projektissa 8 noin vuoden mittaisen kehitystyön jälkeen, jolloin ohjelman toteutukseen ei enää tullut suuria muutoksia. Kaikissa muissa projekteissa päästä päähän -testiautomaatio oli mukana projektin alkuhetkistä saakka.

Ohjelmaan kuuluvan ominaisuuden kehittämisen kannalta ajankohta, jona testejä kirjoitettiin, vaihteli jonkin verran projektien välillä. Projekteissa 3, 4, 5, 6, 7 ja 8 testit kirjoitettiin pääasiassa uuden ominaisuuden toteuttamisen yhteydessä. Kaikissa tapauksissa testejä ei laitettu versionhallintaan heti toteutuksen yhteydessä, vaan saatettiin odottaa, että toteutusta kokeillaan käytännössä ja todetaan, ettei siihen tule heti muutoksia. Projektissa 7 hajautettua versionhallintaa oli sovittu käytettävän siten, että testit kirjoitetaan samaan versiohaaraan ominaisuuden toteutuksen kanssa. Projektissa 2 testit kirjoitettiin varsinaisen toteutustyön jälkeen ja projektissa 1 sitä mukaa, kun paljastuu ominaisuuksia, jotka menevät toistuvasti hajalle muutoksien yhteydessä.

Joissakin tapauksissa testejä pyrittiin kirjoittamaan jopa rinnakkain toteutuksen kanssa. Haastateltavat H3 ja H4 kertoivat aloittavansa testien kirjoittamisen ennen ominaisuuden täydellistä valmistumista. Esimerkiksi ominaisuutta koskevaan näkymään navigoimiselle saatettiin haluta kirjoittaa ensiksi testi. Asteittain testit kehittyvät testaamaan sellaisia pieniä asioita, joita juuri kyseisessä kohdassa kehitystyötä on saatu toteutettua. Tällä tavalla ominaisuuden toteutuksesta on testit mukaan lukien mahdollista saada valmista jälkeä pala kerrallaan.

Haastatellut H1, H4 ja H6 mainitsivat, että päästä päähän -testejä kirjoitettiin usein lisää rikki menneen ominaisuuden korjauksen yhteydessä. Tällä tavalla haluttiin välttyä siltä, että sama ominaisuus menisi huomaamatta uudestaan rikki jonkin toisen muutoksen yhteydessä. H5 nosti esille, että joskus uudet ominaisuudet ovat sidoksissa entisiin siten, että testit saa tehtyä laajentamalla olemassa olevia testejä.

Projektissa 8 oli testikoodin abstraktiotason nostamisella pyritty siihen, että testejä voisivat lukea ja kirjoittaa myös ne projektin jäsenet, jotka eivät varsinaisesti ohjelmoi. Toisaalta oli myös tavoitteena, että asiakas voisi lukea testejä ja nähdä niistä mitä toiminnallisuuksia on toteutettu. Käytännössä tällaista käyttötarkoitusta JBehave-tarina-abstraktio ei kuitenkaan palvellut. Testien kirjoittaminen vaati tarina-abstraktiosta huolimatta myös ohjelmointitaitoja ja teknisten yksityiskohtien tuntemusta, eivätkä testit lopulta olleet kovin luettavia muille kuin ohjelmoijille. Tämä yhdistettynä aliluvussa 4.3 kuvattuun

integroidun ohjelmointiympäristön tarjoaman JBehave-abstraktiotuen puutteeseen asetti tarina-abstraktiosta saadun hyödyn kyseenalaiseksi.

Moni haastatelluista totesi Page-olioabstraktion auttaneen testien ylläpidettävyyden lisäksi testien luettavuutta. Tämä johtui siitä, että testin toiminnan ymmärtämisessä ei tarvinnut jatkuvasti keskittyä testin käyttämän verkkosivun rakenteen yksityiskohtiin. Projektissa 5 Page-oliota ei osattu alussa ottaa kovin vakavasti. Myöhemmin kuitenkin sen hyöty ymmärrettiin ja käytössä siitä saatiin hyviä kokemuksia.

Testien kirjoittaminen koettiin yleisesti melko haasteelliseksi. Osassa projekteista yhdeksi haasteeksi koettiin elementtien valitsimien kirjoittaminen. Valitsimien miettimisen lisäksi aikaa vei myös asynkronisuus ja ajoituksen hallinta. Konkreettisia tuntimääriä testien kirjoittamiseen kuluvasta ajasta oli hyvin vaikea arvioida johtuen muun muassa siitä, että kuluvaan aikaan vaikuttaa se, oliko vastaavanlaiselle ominaisuudelle jo aiemmin tehty testejä tai oliko kyseessä uuden ominaisuuden luominen vai vanhan muokkaaminen. Jos olemassa oli valmiita aputyökaluja, joilla tarvittaviin käyttöliittymäelementteihin pääsi käsiksi, testien kirjoittaminen oli nopeaa. H2 totesi, että harvemmin ominaisuuden päästä päähän -testejä on saatu yhdessä tunnissa valmiiksi, ellei kyse ole hyvin pienestä uudesta ominaisuudesta tai muutoksesta. Yleensä kyse on tunneista ja pahimmillaan päivästä, jos törmätään edellä kuvattuihin ongelmiin.

Haastateltu H4 totesi, että yleiskäyttöisiä osia voisi lainata projektista toiseen, mutta ongelmana on työkaluriippuvuus, eli osat eivät sovi sellaisenaan käytettäväksi toiseen projektiin, koska esimerkiksi ohjelmointikieli ei ole yhteensopiva. Vaikka ohjelmointikielikin olisi yhteensopiva, ei esimerkiksi tähän tutkimukseen saaduissa projekteissa ole montaa sellaista, jotka voisivat vaihtaa helposti osia keskenään, sillä myös testiautomaation tekemisessä käytetyt kirjastot ja ohjelmistokehykset vaihtelevat.

## 4.5 Testien vakaus

Noin puolet haastatelluista nosti esille yhtenä ongelmana testien epävakauden. Epävakaudella tarkoitettiin satunnaisesti ilmeneviä ongelmia, joiden seurauksena testien suorituksen tulos saattoi vaihdella ajokerran mukaan. Epävakaudesta johtuen satunnaisesti saatiin väärä negatiivinen tulos. Väärällä negatiivisella tuloksella tarkoitetaan tilannetta, jossa testit raportoivat ongelman, vaikka ohjelmassa ei ole virhettä. Väärän negatiivisen tuloksen haitta on se, että sellaisen selvittämiseen käytetty aika menee mahdollista oppimista ja tutkimisen yhteydessä havaittuja muita ongelmia lukuun ottamatta täysin hukkaan.

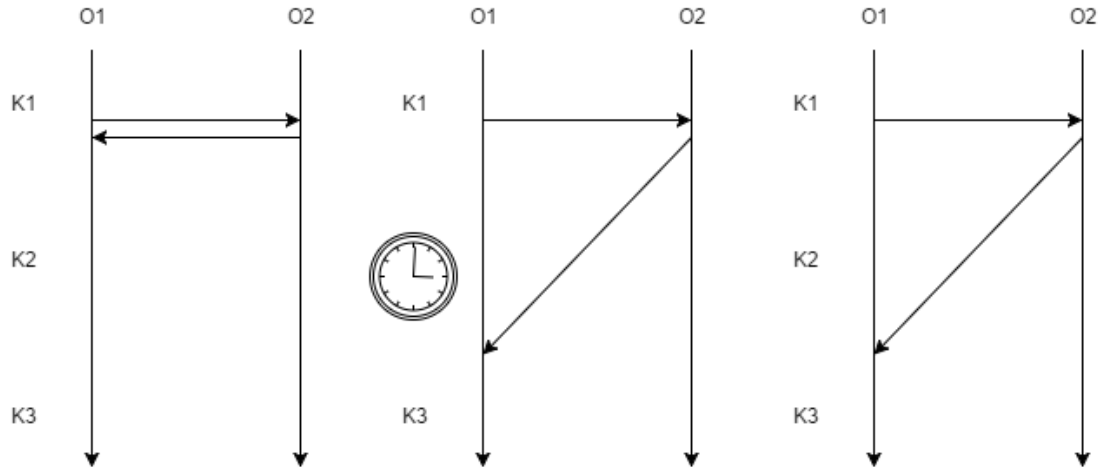
Projektissa 5 oli kärsitty haastateltujen H3 ja H7 mukaan paljon epävakausongelmista. Päästä päähän -testitapauksia oli tehty kohtuullisen paljon, noin 100 kappaletta, vaikkei kyseessä ollut H3:n mukaan kovinkaan suuri web-sovellus. Päästä päähän -testien paljous selittyi osittain sillä, ettei projektin käyttöliittymälogiikalle oltu tehty lainkaan yksikkötestejä. Projektin edetessä saavutettiin tilanne, jossa päästä päähän -testitapauksia oli niin paljon, että riski satunnaiselle epäonnistumiselle kasvoi suureksi. Testien vakautta yritettiin parantaa muun muassa alustamalla selainistunto uudestaan testikokonaisuuksien välissä, mutta tulokset eivät parantuneet. Kuitenkin Page-olio-abstraktion käyttöönotto

helpotti ongelman kanssa. H7:n mukaan testien epävakaus asetti päästä päähän -testiautomaation hyödyn kyseenalaiseksi.

H3:n toisessa projektissa Protractor-testauskehyksen kanssa asynkronisuuteen liittyviä ongelmia ei ollut läheskään yhtä paljon. Haastateltu arvioi, että Clojure WebDriverin harvinaisempi käyttö maailmalla ja siten vähäisempi tiedon ja avun saatavuus saattoi vaikuttaa siihen, että toisessa projektissa onnistuttiin paremmin.

Myös muissa projekteissa oli havaittu muun muassa asynkronisuudesta ja verkkoviiveistä johtuvaa epävakaisuutta, mutta ei niin suurissa määrin kuin projektissa 5. H2 mainitsi, että joskus oli yksittäisiä testejä, jotka olivat epävakaita ja joiden vakautta olisi voinut parantaa kirjoittamalla testi paremmin asynkronisuutta huomioivaksi. Projektissa 8 törmättiin siihen, ettei Selenium ymmärrä milloin asynkronisiin HTTP-pyyntöihin on saatu vastaus ja näkymä on valmis. Tämä oli saatu korjattua kuitenkin siten, että testeistä käsin hyödynnettiin Wicket-käyttöliittymäkirjaston tarjoamaa ominaisuutta, joka mahdollisti sen, että testikoodista käsin pystyi kysymään, onko näkymän kannalta ohjelman suoritus vielä kesken.

Ainakin H2, H3, H4 olivat törmänneet tilanteeseen, jossa ohjelma läpäisee testit kehittäjän omassa kehitysympäristössä, mutta jatkuvan integraation palvelimella ei. Tätä ongelmaa varten projektissa 7 oli tehty kehitystä varten aputyökalu, jolla verkkoviivettä sai simuloitua kehittäjän omassa ympäristössä. Kehittäjän omassa ympäristössä selain, sovelluspalvelin ja tietokantapalvelin voivat sijaita kaikki samassa fyysisessä laitteessa, mistä johtuen verkkoviiveet ovat olemattomia. Tällöin ohjelman on mahdollista toimia samalla tavalla kuin se toimisi, jos pyyntö olisi synkroninen. Kuvassa 2 on demonstroitu synkronisen ja asynkronisen tiedonsiirron eroa. Alaspäin osoittavat nuolet kuvastavat ohjelman suorituksessa kuluvaa aikaa. Kuvassa on kolme tapausta, joissa jokaisessa tehdään kohdassa K1 pyyntö ohjelmalle O2. Pyyntöön saatavaa vastausta tarvitaan kohdassa K3.




Kuva 2. Synkroninen ja asynkroninen tiedonsiirto

Vasemmalla kuvassa on synkronisen tiedonsiirron esimerkki, jossa osapuolen O1 suoritusta jatketaan vasta sitten, kun pyyntöön on saatu vastaus osapuolelta O2. Keskellä

kuvassa on esimerkki asynkronisen tiedonsiirron tapauksesta, jossa ohjelma O1 on suunniteltu siten, että se odottaa pyyntöön vastauksen ennen kohdan K3 suorittamista. Oikealla kuvassa on tapaus, jossa pyynnön asynkronisuutta ei ole otettu huomioon, mutta ohjelma toimii samalla tavalla kuin keskimmäisessä tapauksessa. Näin voi käydä silloin, kun tietoverkon aiheuttama viive ja ohjelman O2 vastauksen muodostamiseen käyttämä suoritusaika ovat vähäisiä verrattuna ohjelman O1 kohdassa K2 kuluvaan suoritusaikaan. Kohdassa K2 voisi tapahtua esimerkiksi raskas käyttöliittymän päivittäminen, jonka aikana ohjelmalta O2 tarvittava vastaus voisi saapua. Tällöin testattava sovellus voisi toimia näennäisesti oikein.

Ajoitukseen liittyvien ongelmien lisäksi ainakin yksi haastateltava oli törmännyt epävakausongelmaan, johon vaikutti se, oliko selainikkuna aktiivisena testien suorituksen aikana. Ongelma voi johtua testauksessa käytettävässä selaimessa tai Selenium WebDriverissa olevasta virheestä. H7 arvioi, että jos selainikkunan aktiivisuus vaikuttaa ongelmallisesti testien suoritukseen, se voisi hankaloittaa testien rinnakkaistamista. Rinnakkain suorituksessa olevat Firefox-selainikkunat toimivat eri prosesseissa ja vain yksi ikkuna voi olla aktiivinen kerrallaan.

## 4.6 Osaamisen kehittäminen

Yleinen haastatteluissa esille tullut kommentti on ollut se, että jos heti projektin alussa testejä olisi osattu tehdä yhtä hyvin kuin osattiin tehdä projektin lopussa, ongelmia olisi ollut huomattavasti vähemmän. Nopeasti uusiutuvista ohjelmistokehyksistä ja työkaluista johtuen lähes jokaisen projektin alussa tekeminen hakee muotoaan. Jos testejä lähdetään tekemään heti alusta asti väärin, eikä korjaavaa liikettä tehdä riittävän ajoissa, ongelmiin törmätään todennäköisesti jossakin vaiheessa.

Kaikilla ei ole kokemusta päästä päähän -testiautomaation tekemisestä, eikä kukaan ole seppä syntyessään. Toimiviksi todetuista käytännöistä ja yleisistä suden kuopista olisi hyvä saada tietoa ennen testien kirjoittamiseen ryhtymistä.

Joillakin päästä päähän -testiautomaatiosta on huonoja kokemuksia. Aiemmissa projekteissa koettujen epäonnistumisten takia asenteet ovat saattaneet kääntyä negatiivisiksi päästä päähän -testiautomaatiota vastaan. Tästä johtuen voi olla tapauksia, ettei epäonnistumisen jälkeen enää haluta ratkaista ongelmia tai oppia tekemään paremmin, vaan todetaan teknologioiden ja keskeneräisten työkalujen olevan syypäitä vaikeuksiin.

## 4.7 Kokemuksia päästä päähän -testiautomaation kannattavuudesta

Yleisesti lähes kaikki haastatellut näkivät yhtenä päästä päähän -testien hyötynä sen, että nähtiin kokonaisten käyttötapausten ja toimintoputkien toimivan halutulla tavalla. Monet kokivat, että yksikkötestien lisäksi oli tarvetta testiautomaatiolle, joka testaa käyttötapauksia kokonaisuutena. Tällä tavalla vähennettiin tarvetta manuaaliselle regressiotestaukselle ja saatettiin säästää aikaa ja rahaa, jota voitiin kohdistaa uusien ominaisuuksien

testaamiseen ja tutkivaan testaamisen. Käytännössä kaikissa projekteissa selaintesteillä oli löydetty virheitä ja sitä kautta saatu toimitettua parempaa laatua asiakkaalle. Rahalliset hyödyt ovat voineet olla suuriakin. On halvempaa havaita ohjelmassa oleva virhe kehityksen yhteydessä kuin asiakkaan käytössä. Yhtä haastateltua lukuun ottamatta kaikki olivat selkeästi sitä mieltä, että päästä päähän -testiautomaatiosta oli enemmän hyötyä kuin haittaa. Hyötyjä voi olla kuitenkin mahdotonta mitata, sillä yhdessäkään tutkimukseen saadussa projektissa ei oltu toimittu niin, että olisi poistettu kaikki päästä päähän -testiautomaatio, lopetettu niiden tekeminen, jatkettu kehitystä ja tutkittu seurauksia.

Päästä päähän -testiautomaatiosta todettiin, että kokonaisuuden testaamisesta seuraa myös sellainen hyöty, että projektiin uusina tulevilla voi olla matalampi kynnys lähteä tekemään muutoksia, kun on ainakin jonkinlaista luottoa sille, ettei vahingossa riko täysin aiemmin toimineita ominaisuuksia. Esimerkiksi johonkin ennen toimineeseen näkymään navigoimisen rikkoontuminen näyttää käyttäjän näkökulmasta pahalta, vaikka sovelluskoodimielessä kyse on pienestä ongelmasta ja se olisi voitu välttää tekemällä edes yhden käyttötapauksen testaava päästä päähän -testi kyseiselle ominaisuudelle.

Haastatellut H3 ja H5 mainitsivat, että hyvin kirjoitettuna päästä päähän -testit voivat toimia osittain toimintojen dokumentaationa. Testeistä voikin olla hyvä lähteä selvittämään itsellensä tapoja, joilla omaisuuden tulisi ainakin toimia. Tästäkin voi olla hyötyä projektiin uusina tuleville kehittäjille, sillä sovellukseen tutustuminen voi olla hyödyllistä aloittaa kokonaisia käyttötapauksia testaavasta testikoodista.

Selaintestiautomaatiota ei kuitenkaan pidetty itsestään selvänä tapana tehdä laadunvarmistusta. H1, H6 ja H7 miettivät olisiko olemassa tehokkaampiakin tapoja saada virheitä havaittua. Jos olisi tiedossa jokin tehokkaampi tapa, joka ratkoo samaa ongelma-aluetta kuin selaintestiautomaatio, kannattaisi tietenkin käyttää sitä. Haastatelluilla ei ollut tutkimushetkellä tiedossa nykyisen kaltaisen päästä päähän -testiautomaation korvaajaa.

# 5. TULOSTEN TARKASTELU

Tässä luvussa tarkastellaan haastattelututkimuksessa saatuja tuloksia ja pyritään yhdistämään niitä alalla vallitsevan tiedon kanssa siten, että löydetään kehittämisideoita, joita hyödyntämällä olemassa olevat ja tulevat web-sovelluksia suunnittelevat ohjelmistoprojektit voivat pyrkiä parantamaan päästä päähän -testiautomaatiotansa.

## 5.1 Rinnakkaistaminen ja palautesyklin nopeuttaminen

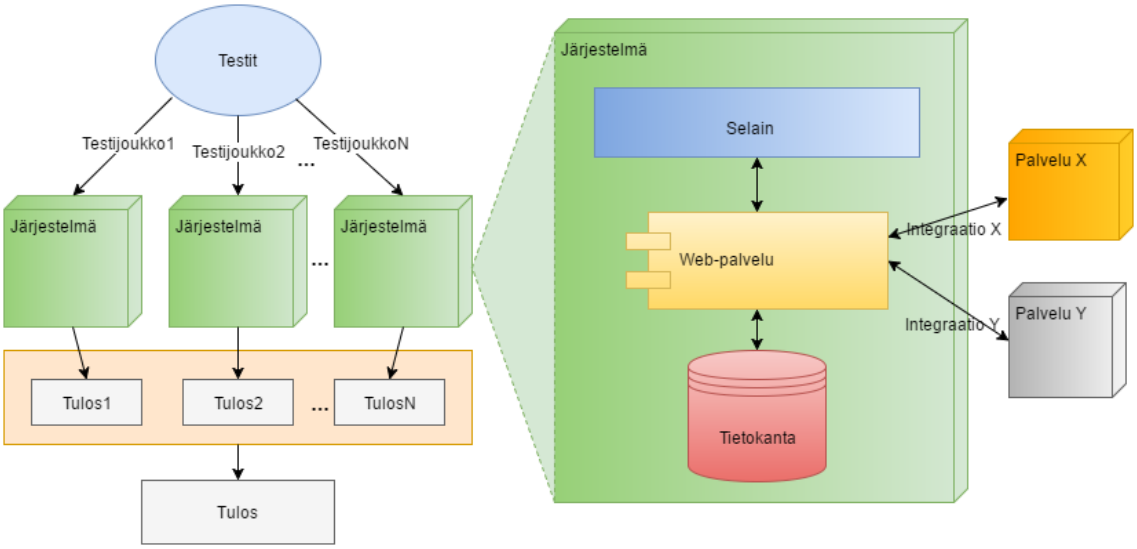
Kuten aliluvussa 4.1 kerrottiin, valtaosa haastatelluista mainitsi testien suorituksen rinnakkaistamisen yhtenä keinona tehostaa päästä päähän -testiautomaatiota, sillä hitaus koettiin ongelmalliseksi. Käytännössä on siis havaittu, että päästä päähän -testit skaalautuvat korkeussuunnassa heikosti, eli niiden määrän rajaton kasvattaminen ja peräkkäin suorittamisen jatkaminen voi johtaa lopulta ongelmiin. Ongelmaksi oli koettu se, että sovelluskehitystyö alkoi kärsiä siitä syystä, että testien hitaan suorituksen johdosta palautesykli heikentyi. Samalla kasvoi riski sille, että testien suorituksen aikana versionhallintaan tallentui uusia virheitä aiheuttavia muutoksia, mistä johtuen virheiden alkuperän selvittäminen vaikeutui. Tavoiteltiin palautteen saamisen nopeuttamista ja palautteen kohdistumista hallittuun määrään muutoksia.

Kasvattamalla testejä suorittavan järjestelmän suorituskykyä ei ole odotettavissa riittävää suoritusajan pienentymistä, koska tällä ei voida vaikuttaa esimerkiksi tietoliikenneviiveisiin. Suorittimen suorituskyvyn kaksinkertaistaminen ei siis tule johtamaan siihen, että testien suorittamiseen kuluva aika puolittuu, eikä suorittimen suorituskykyä ole kaikissa tilanteissa halpaa kasvattaa. Tästä päästään siihen, että testien tulee skaalautua leveyssuunnassa, eli testien lukumäärän lisäämisellä ei pitäisi olla suoraa vaikutusta niiden suorittamiseen kuluvaan aikaan. Tämä tarkoittaa sitä, että testejä pitää suorittaa samanaikaisesti eli rinnakkain. Periaatteessa rinnakkaistamisella on mahdollista päästä lähes vakiosuoritusaikaan, jossa kokonaissuoritusajan määrittää sovelluksen käynnistämiseen kuluvan ajan ja hitaimman testin suoritusajan summa.

Jotta testit voitaisiin suorittaa rinnakkain, pitää ensin ratkaista muutama ongelma. Ensimmäinen ongelma on se, miten testejä voidaan ajaa rinnakkain siten, etteivät ne sotke toistensa dataa. Toisin sanoen toisen testin tekemä tarkiste ei saa rikkoontua siitä, että toinen testi muuttaa tietokantaan tallennettua tietoa. Tietokannan tilan täytyy säilyä sellaisena kuin testiä tehdessä on oletettu. Rinnakkain ajettavilla testijoukoilla täytyy näin ollen olla jokaisella oma tietokanta, joka voidaan testejä varten alustaa tunnettuun tilaan. Tämän tietokannan tila saa riippua vain kyseisen testijoukon käyttäytymisestä. Toinen mahdollinen ongelma on se, ettei selain tue sitä, että selaimesta on samassa käyttöjärjestelmässä useita ikkunoita tai välilehtiä aktiivisena siten, että niihin voi rinnakkaisesti jokaista vastaava testijoukko keskustella ilman ongelmia. Voi myös olla, ettei selaintesti-

automaatiotyökalu sovellu tällaiseen toimintaan. Näin ollen vähintään tietovaraston ja selaimen pitää olla testijoukkojen välillä eriytettyjä toisistaan. Ihanteellisinta olisi se, että testejä rinnakkain suorittavat yksiköt vastaisivat toisiansa kaikilta muilta osin paitsi niissä vallitsevan sovelluksen tilan ja testien suorituksen myötä saatujen tuloksien osalta.

Yksi tapa rinnakkaistaa testien suoritus on jakaa testit joukkoihin ja suorittaa ne erillisissä fyysisissä ympäristöissä, joihin jokaiseen on asennettu sama versio web-palvelusta, sen tietovarastoista ja joiden ympäristöt vastaavat muutenkin toisiansa. Suorittamalla testijoukkoja kokonaan toisistaan erillisissä fyysisissä järjestelmissä välttyy edellä käsitellyiltä ongelmilta kokonaan. Kuvassa 3 on järjestelmäkokonaisuus, jolta eräänlaisen web-palvelun rinnakkaistettu päästä päähän -testauskokoonpano voisi näyttää. Kuvassa oikealla puolella on suurennoksena esitys siitä, miltä yksittäinen suoritusyksikkö voisi sisältä näyttää.




Kuva 3. Testien suorituksen rinnakkaistaminen

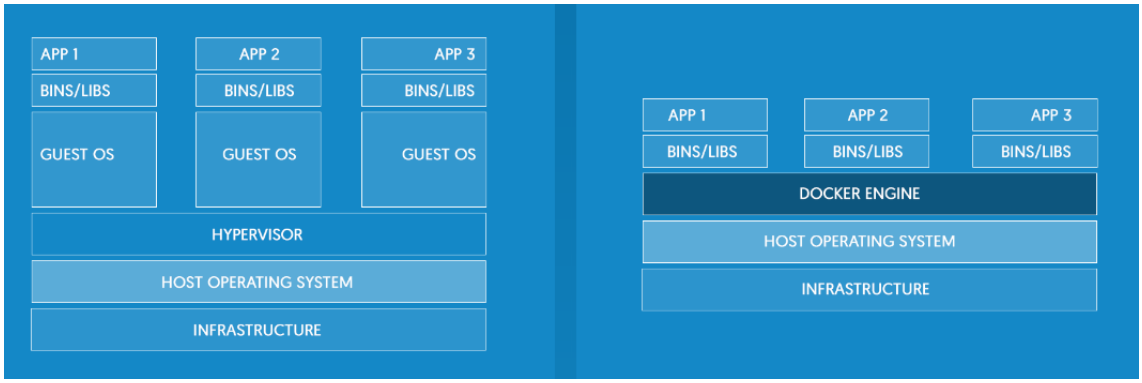
Kuvan kokoonpanossa on jokaisessa järjestelmäyksikössä ajossa sama versio web-palvelusta ja tietovarasto on kaikissa vastaavanlainen. Osiin pilkottu testijoukko suoritetaan suorittajayksiköissä ja tuloksena saadaan joukko testituloksia, joista voidaan koostaa lopullinen kaikki testit kattava tulos. Kuvaa tarkastellessa huomataan, että viimeistään rinnakkaistuksen tullessa ajankohtaiseksi kannattaa miettiä, voiko ulkoisia integraatioita korvata mock-toteutuksella päästä päähän -testien ajamisen ajaksi, sillä rinnakkain ajettuna testauksen aikana integraatioihin voi lähteä valtavasti pyyntöjä lyhyessä ajassa, mistä voi aiheutua ongelmia sekä testeille että ulkoisille palveluille. Selaimen läpi tehdyillä päästä päähän -testeillä tuskin halutaan testata järjestelmien suorituskykyä, sillä sen tekemiseen on parempia menetelmiä olemassa.

Kuvassa 3 esitellyllä kokoonpanolla voitaisiin tehdä jatkuvaa integraatiota, eli hajautettuun versionhallintaan tehtyjen muutoksien jälkeen voitaisiin automaattisesti ajaa päästä päähän -testit läpi rinnakkaistetusti. Tässä on kuitenkin sellainen ongelma, että testejä rikkovien ominaisuuksien pääsemisen jatkuvan integraation palvelimelle asti on havaittu aiheuttavan ongelmia. Testien pitäisi siis suoriutua nykyistä nopeammin myös kehittäjän omassa ympäristössä, jotta kehittäjä voisi itse suorittaa ainakin relevanteimmat

testit ennen muutoksien lähettämistä hajautettuun versionhallintaan. Nykyään voidaan virtualisoinnin avulla suorittaa yhden tietokoneen ja käyttöjärjestelmän sisällä useita virtuaalikoneita (virtual machine). Virtuaalikone on tietokoneen isäntäkäyttöjärjestelmän alla suoritettava normaalia omalla käyttöjärjestelmällä varustettua tietokonetta vastaava kokonaisuus, jolla on omat virtuaaliset liitännät. Se siis voi suorittaa sovelluksia ja keskustella ulkomaailman kanssa isäntäkäyttöjärjestelmään asennetun virtualisoinnin mahdollistavan sovelluksen eli hypervisorin kautta. Yksi esimerkki hypervisorista on VirtualBox [20]. Tällaisella järjestelyllä kehittäjien omilla tietokoneilla voisi suorittaa eri virtuaaliympäristöissä yhtäaikaisesti eri testijoukkoja. Ongelmaksi tässä muodostuu kuitenkin se, että testien ajaminen rinnakkaisesti useassa virtuaalikoneessa voi vaatia niin paljon resursseja, että muu kehitystyö voisi kärsiä. Jokaista suoritusyksikköä varten pitää isäntäkoneen muistista olla varattuna tilaa kokonaiselle käyttöjärjestelmälle. Tehokkaalla palvelinkoneella tällainen menettely tosin voisi olla toimiva.

Yksi merkittävä kysymys, joka usean rinnakkaisen suoritusalustan nousee esille, on se, että miten järjestelmät saadaan asennettua sovelluksien ja säätöjen osalta jokaisen suoritusyksikön osalta täysin samanlaiseksi. Tätä varten on olemassa konfiguraation hallintajärjestelmiä, joiden avulla voi määritellä tilan, johon virtuaalikone asettuu, kun se pystytetään. Yksi tällainen moderni ympäristön pystyttämisen automatisointisovellus on Ansible [21]. Konfiguraationhallintaa ei kuitenkaan käsitellä tässä työssä sen enempää.

Nykyään on olemassa konttiteknologia (container), jota oikein käyttämällä jokaista suoritusyksikköä varten tarvittavaa käyttöjärjestelmää ei tarvitse monistaa isäntäkoneen muistiin useaan kertaan. Konttiteknologia hyödyntää virtualisointia siten, että sovelluksia voi laittaa suoritukseen eristettyinä prosesseina, joilla on oma virtuaalinen muistiavaruus, omat verkkoliitännät ja oma levy. Pohjalla kontit jakavat saman käyttöjärjestelmäytimen. Kuvassa 4 on vasemmalla kokoonpano, jossa virtuaalikoneilla ajetaan useita sovelluksia hypervisorin päällä ja oikealla konttiteknologialla tehty sama toteutus.




Kuva 4. Virtuaalikoneiden ja konttiteknologian ero [22].

Konttiteknologialla käynnistetyt sovellukset näkyvät isäntäkoneen käyttöjärjestelmälle itsenäisinä prosesseina. Tällä tavalla käynnistetyt sovellukset käyttävät samaa yhteen kertaan keskusmuistiin ladattua järjestelmäpohjaa, joka ei muutu. Näin resursseja säästyy paljon ja ensimmäisen sovelluksen käynnistämisen jälkeen toisten sovelluksien käynnistäminen on todella nopeaa, koska pohjalle tarvittava järjestelmä on jo pystytetty.

Konttiteknologialla saavutaan siis sama kuin virtuaalikoneilla, mutta vähemmillä resursseilla ja huomattavasti lyhyemmällä käynnistysajalla. Testausta varten voitaisiin pystyttää web-palvelimen ja tietokantapalvelimen muodostamia konttipareja, jotka on asetettu keskustelemaan toistensa kanssa. Samalla tulee varmistettua se, että ajoympäristöt ovat samanlaiset jokaisella suoritusyksiköllä, koska prosessit on pystytetty samoilla konttimäärittelyillä ja niiden käyttämät yhteiset resurssit ovat jaettuja. [22]

Pilvipalveluissa palvelimen resurssit ovat erittäin skaalautuvia. Päästä päähän -testiautomaatiota voisi suorittaa pilvipalvelussa esimerkiksi rinnakkaistamisen hyötyjen maksimoimiseksi. Sitä mukaa kun testien määrä kasvaa, voi rinnakkaisten virtuaalikoneiden tai konttien määrää kasvattaa ilman huolta siitä, että palvelimen resurssit loppuvat. Pilvipalveluissa kustannukset määräytyvät yleensä käytön mukaan, joten riippuu paljon projektista, onko pilviratkaisun käyttäminen kannattavaa. Suuret IT-jättiläiset tarjoavat usein pilvipalveluja ja yksi esimerkki tällaisesta on Amazon Web Services [23].

Testien suorituksen palautesykliä voi parantaa myös luopumalla testauksen suoritusalustana toimivan selaimen graafisesta puolesta kokonaan. Esimerkiksi PhantomJS toteuttaa selaimien tapaan useimmat web-standardit kuten DOM-hallinnan ja CSS-valitsimet, mutta ei käytä aikaa näkymän piirtämiseen [24]. Tällainen ratkaisu voisi olla toimiva varsinkin silloin, kun sovelluksen ulkoasullisia seikkoja ei ole tarvetta testata esimerkiksi vertailemalla näkymästä otettuja kuvakaappauksia. Toisaalta jos päädytään pelkästään tällaiseen graafista toteutusta vailla olevaan ratkaisuun, luovutaan myös siitä, että saadaan tieto, toimiiko sovellus jollakin oikeasti käytössä olevalla selaimella.

Osassa tutkimukseen osallistuneista projekteista testien ajamisen rinnakkaistamista oli mietitty, mutta jokaisessa näistä projekteista oli törmätty siihen, että rinnakkaistamisen toteuttamiseen ei ole aikaa eikä resursseja. Tästä johtuen kynnystä rinnakkaistamisen toteuttamiselle olisi syytä madaltaa esimerkiksi tekemällä mallitoteutus, jonka pohjalta voisi lähteä liikkeelle. Hankaluuksia aiheuttaa tietenkin se, että projektien teknologiat ja käytännöt ovat hyvin erilaisia, joten malli voi olla huonosti yhteensopiva toisenlaisen projektin kanssa. Siitä huolimatta mallin pohjalta on helpompi ja nopeampi lähteä tekemään projektin tarpeisiin soveltuvaa toteutusta.

## 5.2 Ylläpidettävyys

Testien ylläpidettävyyteen kannattaa kiinnittää huomiota projektin alusta asti. Asiaa voi miettiä siitä näkökulmasta, miten helppoa ulkopuolisen olisi tulla projektiin ja tehdä sovellukseen ja testeihin muutoksia. Ylläpidettävyyteen panostamalla kehittäjä helpottaa myös omaa työtänsä jatkossa, kun sovelluksen ominaisuuksiin tehdyt muutokset vaativat muutoksia myös päästä päähän -testeihin.

Testien luettavuuteen voi vaikuttaa samoin kuin sovelluksen lähdekoodin luettavuuteen. Jos lähdekoodeille suoritetaan katselmointeja, kannattaa myös niihin liittyvät päästä päähän -testit olla katselmoinneissa mukana. Kun ulkopuolinen katselmoi testejä, niiden laatua ja kattavuutta tullaan tarkastelleeksi toisesta näkökulmasta, joka saattaa poi-

keta alkuperäisen kehittäjän näkemyksestä. Saman asian toistaminen, saman asian testaaminen monesti, sovituista käytännöistä poikkeaminen ja muut epäjohdonmukaisuudet on hyvä korjata ylläpidettävyyden parantamiseksi. Ylipäätänsä projektin käyttämät testien kirjoittamiskäytännöt vaikuttavat testien ylläpidettävyyteen merkittävissä määrin. Kirjoittamiskäytäntöjä on käsitelty aliluvussa 5.3.

Sen lisäksi, että testit ovat luettavia, ne on pidettävä ajokelpoisina kaikissa ajoympäristöissä. Testien pitää suoriutua samalla tavalla kehittäjien omilla koneilla ja esimerkiksi jatkuvan integraation palvelimella. Internet-selaimien päivittyminen voi vaatia myös Selenium WebDriverin päivittämistä, sillä selaimen toiminnan muuttaminen aiheuttaa usein myös sitä käskyttävän Selenium WebDriverin ja selaimen välisen rajapintaan muutoksia. Kun selaimeen tulee uusi päivitys, ei sen kanssa toimivaa versiota Selenium WebDriverista ole välttämättä välittömästi saatavilla. Tästä johtuen ei selaimen ole aina järkevää antaa päivittyä automaattisesti eri ajoympäristöissä. Jos testauksessa käytettyjen selaimien antaa päivittyä automaattisesti, on odotettavissa tilanteita, joissa jossakin ympäristössä Selenium WebDriverin ja testien ajamiseen käytettävän selaimen välillä vallitsee yhteensopivuusongelma. Sekä selaimen että Selenium WebDriverin päivittäminen kaikkiin ympäristöihin voi olla raskas prosessi, eikä sitä ole kannattavaa liian usein tehdä, ellei juuri kaikista uusimman selainversion käyttäminen päästä päähän -testien ajamisessa ole jostakin syystä vaatimuksena. Jos päästä päähän -testiautomaatiolla halutaan saavuttaa lähinnä toiminnallista testaamista, eli esimerkiksi sovelluksen ulkoasun täydellinen ruudulle piirtyminen ei ole tarkistettava asia, testien ajoalustana toimivan selaimen versio voi olla syytä jäädyttää johonkin toimivaksi tunnettuun kaikissa ympäristöissä. Selainversion päivitys voidaan näin ollen tehdä tarvittaessa tai esimerkiksi ajoittain, jotta käytössä olisi kuitenkin tarpeisiin nähden riittävän uusi versio.

## 5.3 Testien kirjoittamiskäytännöt

Haastattelujen perusteella erilaisia tapoja kirjoittaa päästä päähän -testejä oli vähintään yhtä paljon kuin projekteja. Yhtä oikeaa tapaa ei varmastikaan ole. Tärkeintä kuitenkin olisi, että projektin kesken sovitaan yhteiset toimintatavat, jotta testeistä ja toteutuksesta tulee kauttaaltaan yhtenäiset. Yhden asian ilmaisu on helpommin tunnistettavissa koodissa, kun samaa asiaa ei ole tehty kautta projektin monella eri tavalla. Esimerkiksi HTML-elementtien valitsimien kirjoittaminen kuuluu tähän kategoriaan.

Yksi sovittava asia on se, kuka tai ketkä päästä päähän -testejä kirjoittavat. Projektissa 5 kaikki toteutustyötä tehneet eivät olleet osallistuneet testien kirjoittamiseen, mikä oli aiheuttanut epäselvyyttä. Muissa projekteissa lähtökohta oli, että kaikki kirjoittavat toteuttamillensa ominaisuuksille myös päästä päähän -testejä, jos tarve vaatii. Joissakin projekteissa tavoitetila oli, että myös ominaisuuksien toteutustyötä tekemättömät kirjoittaisivat testejä. Jokainen näistä vaihtoehdoista voi olla toimiva tapa, jos siitä on projektiin osallistuvien kanssa yhdessä sovittu. Suositeltavaa kuitenkin olisi, että kaikki kehittäjät osallistuisivat päästä päähän -testien kirjoittamiseen, sillä se on yksinkertaisin tapa saada

jotakin varmuutta sille, että testit tulevat kirjoitettua niille ominaisuuksille, joille se nähdään tarpeelliseksi. Ei siis ole tarvetta lähteä erikseen selvittämään, mitkä ominaisuudet ovat vailla päästä päähän -testejä sen takia, ettei ominaisuuden toteuttaneella kehittäjällä ollut tarkoitusta niitä tehdä.

Toinen sopimista vaativa asia on se, missä vaiheessa kehitystyötä testit kirjoitetaan. Tähän vaikuttaa paljon se, millainen on meneillään olevan kehitystyön luonne. Jos projekti on alkuvaiheilla, muutoksia näkymiin ja toimintoihin on luultavasti odotettavissa, sillä koko sovelluksen ulkoasu ja olemus hakevat silloin vielä muotoaan, ellei esimerkiksi käyttöliittymäprototyypeillä ole onnistuttu takaamaan, että määritelty toiminto on hyvä sovitun kaltaisena. Ylipäätänsä muutosuhan alla olevalle ominaisuudelle päästä päähän -testien tekeminen on riskialtista, sillä testien korjaaminen muutoksien jälkeen on osoittautunut monesti työlääksi. Tällaisissa tilanteissa testien kirjoittamista kannattaa lykätä siihen asti, kunnes toteutusta on kokeiltu ja siihen ollaan tyytyväisiä. Tilanne on toinen silloin, kun projekti on edennyt esimerkiksi jatkokehitysvaiheeseen, jolloin muutokset ovat pienempiä tai muuten ollaan tietoisia siitä, että toteutus on lukkoon lyöty, eikä muutoksia ole heti tiedossa. Tällöin päästä päähän -testit kannattaa esitellä muutoksien yhteydessä tai välittömässä läheisyydessä, koska testien kirjoittamisen lykkäämiselle ei ole tällaisessa tilanteessa syytä.

Projektin jäsenien kesken olisi hyvä sopia myös siitä, miten tarkkoja tai laajoja testeistä on tarkoitus tehdä. Ensinnäkin testataanko monilla erilaisilla syötevariaatioilla vai luotetaanko siihen, että yksikkötesteissä tämä on huolehdittu. Toisekseen testataanko vain onnistuneita käyttötapauksia vai otetaanko mukaan myös laittomat syötteen arvot. Entä missä määrin sovelluksen ulkoasulle halutaan tehdä tarkisteita? Haastateltujen keskuudessa oli koettu hyväksi ideaksi pitää päästä päähän -testien määrä maltillisena. Näin ollen esimerkiksi erilaisilla syötevariaatioilla testaaminen on kannattavaa pitää minimaalisena. Toisaalta virhetilanteille olisi hyvä olla tarkisteita, sillä näin myös virheviestien näkyminen käyttäjälle tulee testattua. Ei ole kannattavaa tehdä ainoastaan onnellisia testitapauksia, joissa käyttäjä suorittaa tehtävät juuri kuten sovelluskehittäjä olettaisi. Sovelluksen ulkoasullisien seikkojen testaamisen tarkkuus riippuu projektista. Yhtä lukuun ottamatta kaikissa projekteissa hyväksi todettu ulkoasun testauksen taso oli tarkistaa, että käytön kannalta kriittiset käyttöliittymäkomponentit olivat näkyvissä ja käytettävissä. Päästä päähän -testien määrää suhteessa muihin testiautomaation tasoihin on käsitelty laajemmin aliluvussa 5.5.

Päästä päähän -testiautomaatiolle on hyvä sopia myös testauksen rajat. Pitää löytää järkevä rajaus sille, mitkä järjestelmät ovat mukana testauksessa ja mitkä korvataan testauksen ajaksi mock-toteutuksella, joka jäljittelee oikean järjestelmän toimintaa. Sisäiset järjestelmän osat eivät ole ongelma, koska niiden toimintaan ja ympäristöön projektiryhmä voi itse vaikuttaa. Ongelmia aiheuttavat ulkoiset kolmansien osapuolien hallitsemat järjestelmät, jotka voivat muuttua yllättävästi ja sijaitsevat eri lähiverkossa. Näiden järjestelmien päästä päähän -testaukseen mukaan ottaminen ei ole suositeltavaa, sillä riski muun muassa verkkohäiriöistä johtuville väärille negatiivisille testituloksille on kasvanut.

Tällaisille integraatioille kannattaa tehdä taustapalveluun mock-toteutus, joka on käytössä päästä päähän -testien suorituksen aikana. Integraation toimintaa kannattaa testiautomaatiolla tarkistaa, mutta Internet-selaimen läpi tehtävien päästä päähän -testien sijaan siihen on suositeltavaa käyttää integraatiotestejä.

Projekteissa oli pientä vaihtelua sen suhteen, missä määrin erilaisia abstraktioita haluttiin käyttää ja pidettiinkö niistä. Page-olio-abstraktio, joka piilottaa teknisiä yksityiskohtia tehden testitapauksista luettavampia, oli kaikkien suosiossa ja sen käyttö on jatkossakin erittäin suositeltavaa. Sen sijaan projekteissa 3, 7 ja 8 käytössä ollut tarina-abstraktiotaso jakoi hieman mielipiteitä. Pitää kriittisesti tarkastella sitä, missä kohdassa abstraktiotason nostaminen tuottaa todellista hyötyä. Jos työkalut ovat kunnossa ja ne poistavat ongelman, jossa tarinasta toteutukseen navigointi on haastavaa, ei teknisiä syitä tarina-abstraktion karttamiselle ole. Haastattelututkimus antoi vahvaa vihiä siitä, ettei abstraktiotason nostaminen suoraan mahdollista sitä, että päästä päähän -testejä kirjoittaisivat myös ne, jotka eivät tee varsinaista ohjelmointityötä projektissa, joten pelkästään siinä toivossa tarina-abstraktiota ei kannata ottaa käyttöön. Päätös kannattaa tehdä lopulta projektin jäsenien oman mieltymyksen mukaan.

Yksi näkemyksiä jakanut seikka oli myös se, miten HTML-elementit otettiin käsiteltäväksi dokumenttioliomallista. Rakenteellisien HTML-dokumenttien elementtien ja attribuuttien osoittamiseen on yleensä käytössä joko CSS- ja XPath-valitsimet. Siinä missä CSS-valitsimet perustuvat web-dokumenttien tyylittelyssä käytettyyn Cascading Style Sheets -mekanismiin, XPath on yleisesti rakenteisille dokumenteille luotu tapa osoittaa dokumentin osia [25]. Käytetyllä elementin valitsinmenetelmällä ei ollut juurikaan merkitystä päästä päähän -testien kirjoittamisen yksinkertaisuuden kannalta, sillä menetelmät eivät kehittäjän näkökulmasta eroa juurikaan. Sen sijaan merkittävää oli se, millä tavalla valitsimessa ilmoitettiin se, mistä nimenomaisesta HTML-elementistä on kyse. Jos valitsimesta tekee sellaisen, että se kuvaa navigaatiopolun, jota seuraamalla haluttu elementti löydetään elementtipuussa, tullaan ottaneeksi kantaa dokumentin rakenteeseen. Tällöin näkymään tehty pienikin dokumentin elementtihierarkiaa muokkaava muutos voi rikkoa valitsimen. Asia on toisin silloin, jos valitsin osoittaa suoraan johonkin halutun elementin uniikkiin ominaisuuteen, joka ei ole välittömän muutosuhan alla. Tutkimuksessa kävi ilmi kaksi eri tapaa ratkaista tämä ongelma. Ensimmäinen tapa oli kuvailla valitsimessa halutulle elementille kuuluvat CSS-tyyliluokat. Perusteluna tälle tavalle oli se, että jossakin vaiheessa projektia sovelluksen ulkoasu ei juurikaan koe enää muutoksia, joten elementin CSS-tyyliluokat pysyvät melko muuttumattomina. Toinen tapa oli antaa kiinnostaville HTML-elementeille testi-id-attribuutissa arvo, johon testeissä pystyttiin suoraan viittaamaan. Sekä test-id-attribuuttiin että CSS-tyyliluokkiin osoitetulla viittauksella vältytään siltä ongelmalta, että osoitetun elementin paikan muuttuessa valitsin rikkoontuisi. On kuitenkin kaksi seikkaa, joiden perusteella test-id-attribuutin käyttäminen elementin osoittamiseen on parempi tapa. Ensimmäinen on se, että test-id-attribuutti on luotettavammin uniikki, sillä samassa näkymässä voi olla toinen samoilla tyyleillä varustettu elementti. Toinen on se, että näkymään muutoksia tekevä kehittäjä tietää joidenkin päästä päähän -testien vaativan mahdollisesti muutoksia nähtyään

muutoksiin liittyvien elementtien test-id-attribuutit. Huonoa test-id-attribuutin käytössä on se, että HTML-dokumentin merkkaukseen tulee sovelluksen toiminnan kannalta täysin hyödyttömiä lisäyksiä. Jos tästä pienestä saastuttavasta vaikutuksesta ei ole haittaa, on test-id-attribuutin käyttö suositeltava tapa päästä HTML-elementteihin käsiksi testeissä.

Hyvien käyttöliittymäsuunnitteluperiaatteiden noudattamisesta seuraa se, että käytetyt käyttöliittymäkomponentit ovat eri näkymissä ulkoasultaan ja toimintaperiaatteeltaan yhdenmukaisia. Tästä johtuen useissa eri näkymissä tarvittaville käyttöliittymäkomponenteille on järkevää tehdä testeihin työkaluja, joiden avulla samojen asioiden toistaminen testeissä vähenee helpottaen testien kirjoittamista ja lukemista. Esimerkiksi pudotusvalikosta arvon valitsemista ei kannata tehdä jokaisessa testissä erikseen siten, että ensin haetaan pudotusvalikkoelementti valitsimilla, sitten klikataan elementtiä, jotta pudotusvalikko aukeaa ja lopulta klikataan pudotusvalikosta riviä, jonka teksti vastaa haluttua arvoa. Tällaisen toiminnon suorittamista varten kannattaa aputyökaluihin tehdä funktio, jolle syötteenä annetaan pudotusvalikkoelementin valitsin ja arvo, joka pudotusvalikosta halutaan valita.

Projektissa 1 testejä ei kirjoitettu vaan ne nauhoitettiin Selenium IDE:n avulla. Tämän kaltaisten nauhoitettavien testien hyötynä on se, että niitä voi tehdä täysin ohjelmointitaidotonkin henkilö. Huonona puolena on se, että sovellukseen kohdistuvan pienenkin muutoksen jälkeen koko testitapauksen voi joutua nauhoittamaan uudestaan. Erityisen hyvin tämä voi soveltua kuitenkin niin sanottuihin savutesteihin, joilla pyritään vain testaamaan kevyesti, että sovellus toimii pääpiirteittäin. Hyvä soveltuvuus johtuu siitä, että tämä on ohjelmointikielellä tehtävää testien kuvausta kevyempi tapa toteuttaa testi.

## 5.4 Testien vakauden parantaminen

Päästä päähän -testiautomaatio ei ole luonteeltaan determinististä. Lähes mikä tahansa järjestelmän piirissä tapahtuva kohina voi vaikuttaa lopputulokseen. Näitä ovat esimerkiksi verkkoviiveet, Selenium WebDriverin yhteistoiminta eri selaimien ja selainversioiden kanssa ja haastatteluissa ilmi tullut ongelma, jossa Firefox-selainikkunan epäaktiivisuus oli aiheuttanut satunnaisesti testien väärän negatiivisen tuloksen. Ongelmakentän voi oikeastaan jakaa kahteen osaan: On ongelmia, joihin voi vaikuttaa ja ongelmia, joihin ei voi suoraan vaikuttaa, mutta niitä voi pyrkiä ratkaisemaan löytämällä kiertoteitä.

Ensinnäkin on ongelmia, joihin voi vaikuttaa. Yksi tyypillinen kehittäjän vaikutuspiirissä oleva ongelma on se, etteivät testit odota luotettavasti sivun täydellistä valmistumista. Tällaiseen ongelmaan voi löytää apua esimerkiksi käyttöliittymän toteutuksessa käytetystä ohjelmistokehyksestä tai apukirjastoista. Dynaamisesti dokumenttioliomallia muokkaavien JavaScript-ohjelmistokehyksien tapauksessa on syytä käyttää kehyksen tarjoamaa tapaa odottaa näkymän valmistumista. Projektin 5 päästä päähän -testiautomaatiokoodeista kävi ilmi, että ”odota sivun latautumista” -funktio luotti siihen, että dokumenttioliomallin document.readyState-muuttujan arvo ”completed” kertoo sen, että sivu

on täysin latautunut. On totta, että sivu on tuolloin selaimen näkökulmasta täysin latautunut, mutta käyttöliittymäohjelmistokehyksen harjoittama dynaaminen näkymän muokkaaminen voi olla vielä kesken. Kyseisessä AngularJS-sovelluksessa näkymän valmistumisen voi tarkistaa angular.element(document).ready-funktion avulla [26].

Toinen ongelma, johon kehittäjä itse voi vaikuttaa on Selenium WebDriverin toiminta epävakaustilanteissa, sillä Selenium on avointa lähdekoodia ja kaikkien saatavilla GitHub-palvelussa [27]. Esimerkiksi selaimen aktiivisuustilasta johtuvien ongelmien korjaaminen olisi mahdollista. Muutenkin Selenium testauskehyksen ja ekosysteemin laatua olisi mahdollista parantaa sillä, että sitä kehittävä yhteisö saisi osakseen lisää taitavia ohjelmoijia. Tosin haastattelujen perusteella kehittäjillä on harvoin tai pahimmillaan ei koskaan aikaa osallistua päivätyön ja projektin ohessa tässä työssä käsiteltyjen testiautomaatiotyökalujen parantamiseen. Tästä johtuen avoimen lähdekoodin testiautomaatiotyökalujen parantaminen vaatisi yritystasolla resurssien käyttämistä työkalujen kehitystyöhön. Myös työnantajan puolelta tuleva kannustus vapaa-ajalla tapahtuvaan avoimen lähdekoodien projektien kontribuutioon voisi parantaa esimerkiksi testiautomaatiokehyksien toimintaa ja työntekijöiden ammattitaitoa. Tällaisesta toiminnasta on jo kokeiluja alalla [28].

Yllä mainittuja epävakaudesta johtuvia ongelmia voi toisaalta ratkaisemisen lisäksi yrittää kiertää. Jos testien väärä negatiivinen tulos johtuu aidosti satunnaisuudesta ja väärä positiivinen tulos on todella harvinainen, ohjelman toimivuutta voi arvioida tilastollisesti. Aivan kuten osa haastatelluista oli monesti ensimmäisenä toimenpiteenään suorittanut testit uudelleen sen jälkeen, kun ensimmäisen kerran virhe oli havaittu, jokainen epäonnistunut testi voitaisiin jo lähtökohtaisesti yrittää suorittaa toiseen kertaan tai useammin. Tällöin yksi onnistunut suoritus jokaista testiä kohti riittäisi olettamukseen, että ohjelma toimii testeissä tarkoitetulla tavalla. Yksi vaatimus tälle tietenkin on se, että testit ovat jaettu siten itsenäisiin moduuleihin, että niiden suorittaminen riippumatta toisistaan on mahdollista. Tätä vaatimusta ei ole, jos suoritetaan kaikkia testejä erillisissä, mutta toisiaan vastaavissa ympäristöissä rinnakkain ja muodostetaan saaduista osatuloksista lopputulos. Esimerkiksi projektissa 4 suoritetussa kokeilussa jatkuvan integraation palvelimella suoritettiin samalla sovellusversiolla 122 kertaa käännös lähdekoodeista, sovelluksen käynnistäminen ja testiautomaation ajo sitä vasten. Näistä 12 tuotti negatiivisen tuloksen päästä päähän -testeissä havaitun virheen takia. Oletetaan, että testit suoriutuvat toisistaan riippumatta. Näin ollen todennäköisyys sille, että kaikki testit suoriutuvat onnistuneesti oli $\frac{122-12}{122} \approx 0{,}9016$. Päästä päähän -testien lukumäärän ollessa 99 saadaan yksittäisen testin läpi menolle todennäköisyys $\sqrt[99]{0{,}90164} \approx 0{,}9990$. Todennäköisyys lopulta sille, että yksi testi antaisi väärän negatiivisen tuloksen kahdesti peräkkäin olisi näiden tietojen perusteella siis $(1 - 0{,}9990)^2 \approx 1 * 10^{-6}$. Saman testien useaan kertaan toistamisen menetelmä voisi olla toimiva, mutta ensisijaisesti kannattaisi yrittää ratkaista yllä olevat ongelmat siten, että testeissä ilmenevä epävakaus on mahdollisimman harvinaista. Ongelmien ratkaiseminen kiertämisen sijaan on tietenkin etusijalla. Jos nimittäin edellä mainittu virheiden satunnaisuus ei päde, eli on määrättyjä testejä, jotka antavat

toisinaan negatiivisen tuloksen, voi ongelma johtua sovelluksessa olevasta virheestä, joka tapahtuu harvoin. On hyvin uskottavaa, että paljon asynkronista toimintaa sisältävissä moderneissa web-sovelluksissa on mahdollista luoda tilanteita, joita sovelluskehittäjät eivät ole osanneet ottaa huomioon.

Toisekseen on ongelmia, joihin ei suoraan voi vaikuttaa. Esimerkiksi verkkoyhteys voi katketa kesken testien suorituksen. Lisäksi jos verkkoyhteys katkeaa juuri sellaiseksi aikaa, etteivät kehittäjät sitä itse huomaa, mutta testeissä päädytään aikakatkaisuun, testiautomaatiokehyksen antama virheilmoitus voi olla harhaanjohtava. Tällaisten ongelmien ratkaisemiseen vaaditaan testien suorittajalta älykkyyttä. Ihminen testatessaan järjestelmää jäisi odottamaan verkkoyhteyden palaamista ja suorittaisi testin uudelleen sen jälkeen. Nykyiset päästä päähän -testiautomaatiokehykset ilmoittavat tässä kohdassa, ettei testissä vaadittua HTML-elementtiä löytynyt dokumenttioliomallista, nostavat lipun epäonnistumisen merkiksi ja jatkavat seuraavaan testitapaukseen. Moderneilla koneoppimismetelmillä, kuten digitaalisilla neuroverkoilla, olisi mahdollista luoda malleja, jotka määrittelisivät, millä tavalla testiautomaation tulisi käyttäytyä eri tilanteissa. Esimerkiksi väärien negatiivisten tuloksien määrää voisi vähentää sillä tavalla, että koneoppimisjärjestelmälle annettavassa opetusdatassa olisi testien suorituksen aikana ympäristöstä saatua tietoa ja varmojen väärien negatiivisten tapauksien virheilmoituksia. Jos järjestelmä tunnistaa väärän negatiivisen tuloksen tarkoittaa se, että testi pitää suorittaa uudelleen. Riittävän suuren opetusdatajoukon kerääminen voi kuitenkin osoittautua ratkaisevaksi ongelmaksi. Koneoppimisjärjestelmän suoriutuminen riippuu opetusdatan määrästä ja laadusta. Haastattelun piiriin kuuluneissa projekteissa riittävän suuren opetusdatajoukon kokoon saaminen olisi ollut epätodennäköistä.

## 5.5 Testaustasojen suhde

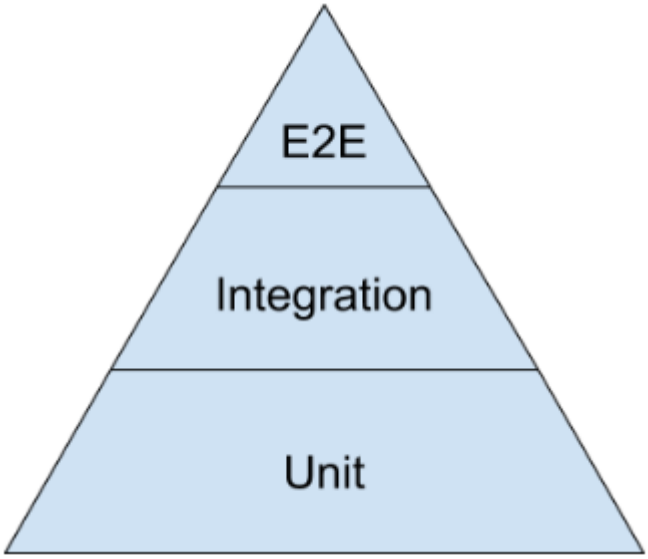
Osassa projekteista yksikkötestien vähyyttä tai täydellistä puuttumista on yritetty korvata suuremmalla määrällä päästä päähän -testiautomaatiota. Tästä seurauksena on pitkä palautesykli, koska päästä päähän -testien suorittaminen on yleensä hidasta, ellei asiaa ole ratkaistu esimerkiksi rinnakkaistamisella. Toinen seuraus voi olla huono testikattavuus, sillä selaintestien kirjoittaminen on koettu jokseenkin haasteelliseksi. Haasteellisesti kirjoitettavista testeistä tuskin tehdään maltillisessa ajassa kovin kattavia. Hitailla, epävakailla ja hankalasti kirjoitettavilla päästä päähän -testeillä käyttöliittymä- tai taustasovelluslogiikan monipuolinen testaaminen ei ole mielekästä.

Sellaista toimintalogiikkaa, joka sijaitsee ainoastaan taustapalvelussa, ei ole järkevää yrittää testata laajasti selaimesta käsin. Taustapalvelussa yksikkötasolla tarkasti testattua ominaisuutta riittää testata päästä päähän -tasolla muutamalla arvojoukolla, esimerkiksi sallitut arvot ja ei-sallitut arvot, sillä yksikkötestien tarkoitus on pitää huoli siitä, että ominaisuus toimii oikein myös muilla parametreilla. Niin kuin moni haastatelluista mainitsi, tarkoitus on testata kokonaisia käyttötapauksia tai ylipäätänsä yksikkötestejä suuremman mittakaavan toimintopolkuja.

Pelkästään selaimessa suoritettavan käyttöliittymälogiikan testaamiseen ei kannata turhaan ottaa mukaan muiden järjestelmän osien toimintaa. Käyttöliittymälogiikalle kattavasti tehdyt yksikkötestit testaavat sen, että käyttöliittymä toimii oikein erilaisilla syötteillä ja taustapalvelun antamilla vastauksilla. Moderneille JavaScript-käyttöliittymäkirjastoille rakennetut testauskehykset tukevat usein valmiiksi sitä, että sovelluksen tekemille HTTP-pyynnöille voi kirjoittaa testiä varten oikean taustapalvelun vastauksen jäljitelmän, minkä seurauksena testejä voi suorittaa ilman, että yhteyttä taustapalveluun tarvitaan. Tällä tavalla käyttöliittymälogiikkaa voidaan testata itsenäisenä osana ilman riippuvuutta taustajärjestelmään. Selaimessa suoritettavan JavaScript-sovelluslogiikan yksikkötestien heikkous tosin on se, etteivät ne tarkista sitä, että näkymä piirtyy oikein vaadituissa selaimissa.

Herää kysymys, pitäisikö erikseen olla sellainen selaintestien muoto, jossa testaus tapahtuu selaimeen piirtyvän käyttöliittymän kautta, mutta varsinaisen taustapalvelun sijaan selainsovelluksen tekemiin HTTP-pyyntöihin vastaisi testiä varten tehty taustapalvelua jäljittelevä mock-toteutus. Tällä tavalla selaintestit eivät olisi riippuvaisia taustajärjestelmistä ja testien suorittaminen voisi olla nopeampaa ja luotettavampaa. Ongelmaksi jää kaikesta huolimatta selaintestien kirjoittamisen haasteellisuus ja uutena lisävaivana olisi taustapalvelua vastaavan mock-toteutuksen tekeminen.

On selvää, ettei kokonaisen web-sovelluksen testiautomaatiota kannata jättää yksistään päästä päähän -testien varaan. Eri testiautomaation muotoja on erittäin kannattavaa hyödyntää, mutta niiden määrä on oltava toisiinsa nähden järkevässä suhteessa. Kuvassa 5 on Googlen testiautomaatiokonferenssissa vuonna 2014 esitetty testauspyramidi, josta ilmenee ajatus testiautomaation eri muotojen suhteesta.




Kuva 5. Testauspyramidin esimerkkikuva Googlen testiautomaatiokonferenssissa vuonna 2014 [6].

Testauspyramidin idea on se, että alimmaisena olevia yksikkötestejä (unit test) kirjoitetaan muihin testiautomaatiomuotoihin verrattuna eniten. Näihin mukaan lukeutuvat

selaimessa suoritettavan sovelluksen taustapalvelun yksikkötestien lisäksi muiden järjestelmän osien yksikkötestit. Seuraavaksi eniten, mutta selvästi vähemmän kuin yksikkötestejä, on suositeltavaa olla integraatiotestejä (integration), joiden tarkoitus on testata, että järjestelmän eri osien välinen kommunikaatio on toimiva. Esimerkiksi taustapalvelun ja ulkoisien järjestelmien välisen keskustelun testaaminen on integraatiotestaamista. Muita muotoja vähemmän tarkoitus on käyttää päästä päähän -testiautomaatiota, jonka heikkouksia ovat hitaus, epävakaus ja kirjoittamisen haasteellisuus. Googlen suosittelema määrä näiden testiautomaatiomuotojen suhteelle on 70, 20 ja 10 prosenttia pyramidin alaosasta yläosaan luettuna [6].

## 5.6 Osaamisen kehittäminen

Päästä päähän -testiautomaation kirjoittaminen, kuten muukin ohjelmistoalalla työskentely on prosessi, jossa oppimisen ja kokemuksen kautta kehitytään paremmiksi tekijöiksi. Osaamisen ylläpito ja kehittäminen kuuluvat olennaisena osana myös ohjelmistojen testaamiseen. Testiautomaatio yhtenä laadunvarmistusmenetelmänä vaatii myös kriittistä tarkastelua ja kehittämistä. Osaamisen jatkuva kehittäminen on tie entistä tehokkaampiin tapoihin varmistaa laatua.

Epäonnistumisia voisi yrittää vähentää tiedon jakamisella. Osaamisen kehittämiseksi voisi esimerkiksi järjestää aiheeseen liittyviä koulutuksia. Yksi tapa voisi olla esimerkiksi esitellä päästä päähän -testiautomaatiota sellaisista projekteista, joissa siitä on ollut positiivisia kokemuksia. Koulutuksessa voisi kertoa, mitä ongelmia on onnistuttu välttämään ja millä tavoilla. Myös päästä päähän -testiautomaatioon liittyvien hyväksi todettujen projektikäytäntöjen esittely voisi myös olla hyödyllistä. Lisäksi yleiset asiat kuten elementtien valitsimien kirjoittamiskäytännöt, teknisten yksityiskohtien abstrahointiin tehdyt työkalut ja testien yleinen rakenne voisivat tulla koulutuksessa esille.

Teknologioiden kehittyessä aiemmin vaivanneisiin ongelmiin saattaa löytyä ratkaisuja. Tehokkaimpien tarjolla olevien ratkaisujen käyttö vaatii jatkuvaa kehityksen seuraamista. Selainten kehittyessä myös niihin kytketyt testausominaisuudet kehittyvät. Esimerkkinä Firefoxia aiemmin vaivannut ongelma, joka esti kokonaan testien ajamisen rinnakkain monessa ikkunassa [29] (Bug 704583). Omalta osaltaan jokainen voi vaikuttaa omaan oppimiseensa ja työkalujen toimintaan osallistumalla avoimen lähdekoodin projekteihin. Suurimmassa osassa tutkimuksen piirissä olleista projekteissa käytössä olleet Firefox-selain ja Selenium WebDriver ovat molemmat avoimen lähdekoodin projekteja.

# 6. YHTEENVETO

Tässä luvussa tarkastellaan yhteenvetona niitä keinoja, joilla päästä päähän -testiautomaatiosta saadaan tehokkaampaa ja vaivattomampaa. Lisäksi mietitään sitä, onko päästä päähän -testiautomaation rakentaminen lopulta kannattavaa web-sovellusprojekteissa. Lopuksi käsitellään tutkimuksen onnistumista.

## 6.1 Päästä päähän -testiautomaation hyödyntäminen

Päästä päähän -testiautomaatiosta voidaan saada tehokkaampaa hiomalla testeistä saatavaa palautteen saantia nopeammaksi, käyttämällä projektiin parhaiten sopivia työkaluja ja sopimalla toimintatavoista siten, että testeistä tulee yhtenäisiä, vakaita, hyviä luettavuudeltaan ja ylläpidettävyydeltään ja niitä on tarpeisiin nähden sopiva määrä.

Palautesyklin nopeuttamiseksi testien suoritus kannattaa rinnakkaistaa. Tähän valmiina tarjottavat esimerkiksi konttiteknologialla tehdyt mallitoteutukset voisivat tuoda helpotusta, mutta mallin hyödyllisyyttä kyseenalaistaa yhä se, että web-sovellusprojektit ovat usein teknologioiltaan ja toimintatavoiltaan hyvin erilaisia. Esimerkkitoteutuksesta joillakin teknologioilla arveltiin kuitenkin olevan hyötyä.

Tutkimuksen mukaan testien luettavuuteen ja ylläpidettävyyteen voi vaikuttaa luomalla projektille yhteisiä yleiskäyttöisiä työkaluja, joita käytetään laajasti eri testeissä läpi sovelluksen. Katsottiin, että sopimalla projektin sisällä toimintatavoista ja noudattamalla ja kehittämällä niitä projektin edetessä saavuttaa parhaan lopputuloksen. Todettiin myös, että testien luettavuutta ja yhtenäisyyttä voi parantaa sopivalla määrällä abstraktiota testien kuvauksissa. Tutkimuksen mukaan ainakin Page-olio-abstraktio koettiin poikkeuksetta hyödylliseksi ja sillä saatiin parannettua testien laatua. Sen sijaan tarina-abstraktio jakoi mielipiteitä haastateltujen joukossa.

Luettavuuden ja ylläpidettävyyden lisäksi testien suorituksen tulisi olla mahdollisimman vakaata, jotta turhalta satunnaisista vääristä negatiivisista tuloksista johtuvalta työltä vältyttäisiin. Asynkronisuuden huomioiminen testeissä on tärkeää, sillä tutkimuksen mukaan asynkronisuuden hallinta oli vaikeata ja sen koettiin olevan suurin syypää testien epävakauteen. Useat epävakaat testit, johtui epävakaus sitten sovelluksesta tai testistä, voivat saada koko testiautomaatiomuodon näyttämään kannattamattomalta. Jos muutoksien mukanaan tuoma testien ylläpitotyö vie paljon aikaa, on testiautomaation rakentamiseen ja virittelyyn käytetty aika vaakalaudalla päästä päähän -testiautomaatiosta saadun hyödyn kanssa.

Selaintestiautomaation kehityksessä pitää pysyä mukana, jotta tietää mitä vaihtoehtoja silloin on, kun eteen tulee projekti, jossa selaintestiautomaation hyöty selvästi nähdään. Osaamisen kehittäminen ja kehityksen mukana pysyminen on tärkeää, sillä uusia työkaluja tulee jatkuvasti ja ne tuovat mukanaan testien tekoa, ylläpitoa ja ajamista hel-

pottavia ominaisuuksia. Haastatteluun osallistuneissa projekteissa käytettyjä selaintestiautomaatiotyökaluja kehittyneempiä versioita on jo olemassa. Esimerkiksi tätä kirjoitettaessa kehitteillä oleva Cypress poikkeaa kaikista tutkimuksen projektien selaintestiautomaatiotyökaluista monella tapaa. Cypress ei käytä Selenium WebDriveria apunaan vaan on itsessään selainta ulkoasultaan ja toiminnaltaan muistuttava sovellus, jonka toteutus pohjautuu olemassa olevaan avoimen lähdekoodin selaimeen. Cypress on siis itsessään selain, jota on laajennettu testausominaisuuksilla. Se poikkeuksellisesti tarjoaa mahdollisuuden ajaa testejä askel kerrallaan ja mahdollistaa tuloksien reaaliaikaisen tarkastelun työkaluun kytketystä konsolista käsin helpottaen virheiden jäljittämistä testikoodissa tai itse sovelluksessa. Myös asynkronisuuden hallintaan ja näkymän valmistumisen odottamiseen luvataan ratkaisuja. Heikkoutena tällaisessa työkalussa on se, ettei saada takeita siitä, että testattava sovellus toimii täsmälleen yhtä hyvin käyttäjien käyttämillä selaimilla. Kuitenkin jos tutkimukseen osallistuneissa projekteissa olisi ollut käytettävissä kehittyneempien työkalujen ominaisuuksia, päästä päähän -testiautomaation tekeminen olisi varmasti ollut vaivattomampaa. [30]

Päätös päästä päähän -testiautomaation käyttöönottamisesta on tehtävä projektikohtaisesti. Esimerkiksi projektin aikataulu, tekijät, sovelluksen kohdealue ja toimintakriittisyys ynnä muut seikat vaikuttavat päätöksen tekemiseen. Riskit vaikuttavat vaadittavaan testaamisen tasoon. Jos sovelluksen toimintaan ei liity merkittävää riskiä, eli virheellinen toiminta ei aiheuta kenellekään merkittävää harmia, ei sovelluksen testaamiseen kannata käyttää paljoa resursseja. Sovelluksen toimintakriittisyyden kasvaessa kasvaa myös tarve testaamiselle. Testiautomaatio selaintestit mukaan lukien on toimiva tapa rakentaa luotettavuutta sille, että valmiiksi tehdyt ominaisuudet toimivat halutulla tavalla myös jatkossa.

Testiautomaation tekemiseen menee resursseja, olivatpa siihen käytetyt työkalut ja toimintatavat kuinka hyviä tahansa. Kannattaa miettiä tuovatko päästä päähän -testit oikeasti lisäarvoa projektille. Jos ollaan varmoja siitä, ettei päästä päähän -testiautomaation tuomat hyödyt riitä korvaamaan testien tekemiseen käytettyä aikaa, ei päästä päähän -testiautomaatiota kannata silloin ottaa turhaan käyttöön. Riskinä selaintestien kokonaan pois jättämisessä on se, että aiemmin toimineet ominaisuudet rikkoontuvat huomaamatta, ellei sovellukselle tehdä jatkuvasti kattavaa käsin testaamista. Manuaalinen testaaminen on myös työlästä, jos jokaisen sovelluksen uuden version toimituksen yhteydessä on käytävä uusien ominaisuuksien lisäksi myös aiemmin kehitettyjä ja testattuja ominaisuuksia läpi.

Suurin osa tutkitusta joukosta piti kuitenkin päästä päähän -testiautomaatiota kannattavana muun muassa siitä syystä, että se on ollut toimiva tapa luoda varmuutta sille, että vanhemmat ominaisuudet toimivat myös sen jälkeen, kun uusia kehitetään. Näin ollen voidaan todeta, että ainakin tutkimuksessa mukana olleiden kaltaisiin web-sovellusprojekteihin päästä päähän -testiautomaatio soveltuu jo nykyisellään. Menetelmien ja työkalujen parantuessa selaintestiautomaatio kasvattaa potentiaalisuuttaan yhtenä testiautomaation muotona integraatio- ja yksikkötestien ohella.

## 6.2 Tutkimuksen onnistuminen

Haastattelututkimus onnistui pääpiirteittäin hyvin, sillä haastateltujen näkemykset tulivat kattavasti esille. Haastattelukysymykset pyrkivät avaamaan aihepiiriä mahdollisimman tehokkaasti, jotta kokonaiskuva päästä päähän -testiautomaatiosta, sen tehokkuudesta ja kannattavuudesta pääsisi syntymään. Haastattelut päättyivät aina siihen, ettei haastatellulle tullut enää mieleen mitään muuta aiheeseen liittyvää sanottavaa.

Tunnin mittainen kesto haastattelulle osoittautui pienessä osassa tapauksia haasteelliseksi. Tähän vaikutti hieman se, että haastattelun tuloksien kirjaamiseen kului jonkin verran aikaa, mutta se toisaalta antoi haastatellulle aikaa miettiä asiaa ilman kiusallista odottelun tunnetta. Haastattelun aikaraamin riittävyyteen vaikutti myös kysymyksien määrä ja laatu. Osa kysymyksistä tuntui olevan sellaisia, että niihin vastaus tuli usein jo jonkin aiemman kysymyksen yhteydessä. Tällaisissa tilanteissa kysymys kuitenkin aina esitettiin mahdollisten lisäyksien ja täydennyksien saamiseksi. Suurimmassa osassa haastatteluja aika riitti hyvin. Kuitenkin joustavampi aikaraami haastattelulle olisi voinut olla hyvä asia, jotta tarpeettomalta kiirehtimiseltä olisi vältytty ja haastatteluista olisi saatu rennompia.

Haastattelukysymyksien laatimiseen käytettiin aikaa ja vaivaa. Siitä huolimatta ei ole selvää, oliko kysymysvalikoima täysin onnistunut. Arvelua herättää se, painottuiko haastattelu kysymyksiensä laadun takia liikaa muutamaan seikkaan sen sijaan, että tulos olisi kokonaisuuden kannalta ollut tasapainoisempi. Esimerkiksi olisi voinut kysyä suoraan, millä tavalla nykyisenkaltainen päästä päähän -testiautomaatio kannattaisi korvata. Toisaalta tätä kysymystä muiden ohella moni haasteltu mietti oma-aloitteisesti. Haastattelukysymyksien mahdollisia puutteita tai laatua kompensoi haastateltujen henkilöiden ammattitaitoisuus ja ratkaisuja etsivä ajattelutapa.

Haastattelujen aikana vastauksien ja ajatuksien kirjaamiseen käytetty aika maksoi itsensä takaisin sillä, että kattavien, rakenteellisesti yhtenäisten muistiinpanojen pohjalta tuloksien vertailu ja arviointi onnistuivat hyvin. Äänittämällä haastattelut ja tekemällä niistä jälkeenpäin vertailuun hyvin sopivat rakenteet olisi saatu parempi lopputulos, mutta se olisi vaatinut vähintään kaksi kertaa enemmän aikaa, sillä jokainen haastattelu olisi pitänyt käydä toiseen kertaan läpi. Vastauksien äänittämistä välteltiin myös sen takia, ettei vastauksien rehellisyys kärsisi esimerkiksi tallenteen leviämisen pelon takia.

Kokonaisuutena tutkimus onnistui hyvin, sillä tutkimuksessa löydettiin keinoja, joilla päästä päähän -testiautomaation tekemistä voidaan kehittää. Lisäksi mahdollisia jatkotutkimusta vaativia aiheita nousi esille, kuten esimerkiksi se, miten testien rinnakkain ajaminen käytännössä toteutetaan tai miten testien suorituksesta voisi tulla älykkäämpää koneoppimisen tavoin.

# VIITTAUKSET


[1] W3C, ”Selectors Level 3,” [WWW]. Saatavissa: https://www.w3.org/TR/selectors/. [viitattu 2.11.2016].

[2] W3C, ”Document Object Model,” [WWW]. Saatavissa: https://www.w3.org/DOM/. [viitattu 2.11.2016].

[3] FiSTQB, ”ISTQB:n testaussanasto v. 2.3 Suomi - Englanti,” 30.4.2015. [WWW]. Saatavissa: http://www.fistb.fi/fi/tiedostot. [viitattu 2.11.2016].

[4] Agile Alliance, ”Mock Objects,” [WWW]. Saatavissa: https://www.agilealliance.org/glossary/mocks/. [viitattu 2.11.2016].

[5] W3C, ”Standards,” [WWW]. Saatavissa: https://www.w3.org/standards. [viitattu 26.3.2016].

[6] M. Wacker, ”Just Say No to More End-toEnd Tests,” Google, 22.4.2015. [WWW]. Saatavissa: http://googletesting.blogspot.fi/2015/04/just-say-no-to-more-end-to-end-tests.html. [viitattu 17.3.2016].

[7] C. Doshi ja R. Laycock, ”P2 Magazine | We hate your broken UI tests!,” 10.4.2014. [WWW]. Saatavissa: https://www.thoughtworks.com/p2magazine/issue10/broken-ui-tests/#. [viitattu 12.5.2016].

[8] G. J. Myers, S. Corey and T. Badgett, The Art of Software testing, 3rd Edition, Wiley, 2011.

[9] M. Katara, M. Vuori ja A. Jääskeläinen, *Ohjelmistojen testaus,* Tampere: Tampereen teknillinen yliopisto, Tietotekniikan laitos, 2015.

[10] Exforsys, ”What is End-to-End Testing,” 2.8.2011. [WWW]. Saatavissa: http://www.exforsys.com/tutorials/testing-types/end-to-end-testing.html. [viitattu 29.3.2016].

[11] SeleniumHQ, ”Selenium Documentation,” Project Selenium, 16.3.2016. [WWW]. Saatavissa: http://www.seleniumhq.org/docs/#. [viitattu 17.3.2016].

[12] ”WebdriverIO - Selenium 2.0 bindings for NodeJS,” [WWW]. Saatavissa: http://webdriver.io/. [viitattu 30.3.2016].

[13] ”Robot Framework,” Robot Framework Foundation, [WWW]. Saatavissa: http://robotframework.org/. [viitattu 12.5.2016].

[14] ”AngularJS Developer Guide: E2E Testing,” [WWW]. Saatavissa: https://code.angularjs.org/1.2.16/docs/guide/e2e-testing. [viitattu 12.5.2016].

[15] Jyväskylän yliopisto, ”Ankkuroitu teoria eli grounded theory,” [WWW]. Saatavissa: https://koppa.jyu.fi/avoimet/hum/menetelmapolkuja/menetelmapolku/aineiston-analyysimenetelmat/ankkuroitu-teoria-eli-grounded-theory. [viitattu 18.1.2017].

[16] R. Koskennurmi-Sivonen, ”Grounded Theory,” 2004. [WWW]. Saatavissa: http://www.helsinki.fi/~rkosken/gt. [viitattu 3.3.2016].

[17] C. B. Seaman, ”Qualitative methods in empirical studies of software engineering,” 6 elokuu 2002. [WWW]. Saatavissa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=799955. [viitattu 17.3.2016].

[18] S. E. Hove ja B. Anda, ”Experiences from conducting semi-structured interviews in empirical software engineering research,” 1.9.2005. [WWW]. Saatavissa: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1509301. [viitattu 17.3.2016].

[19] S. Fincher ja M. Petre, Computer Science Education Research, Taylor & Francis Group plc, 2004.

[20] Oracle, ”VirtualBox,” [WWW]. Saatavissa: https://www.virtualbox.org/. [viitattu 25.10.2016].

[21] Red Hat, ”Ansible,” [WWW]. Saatavissa: https://www.ansible.com/. [viitattu 25.10.2016].

[22] Docker Inc., ”Docker - Buld, Ship, and Run Any App, Anywhere,” [WWW]. Saatavissa: https://www.docker.com/. [viitattu 26.10.2016].

[23] Amazon, ”Amazon Web Services,” [WWW]. Saatavissa: https://aws.amazon.com. [viitattu 2.11.2016].

[24] A. Hidayat, M. Svay ja J. Mason, ”PhantomJS,” [WWW]. Saatavissa: http://phantomjs.org/. [viitattu 2.11.2016].

[25] W3C, ”All Standards and Drafts,” [WWW]. Saatavissa: https://www.w3.org/TR/. [viitattu 10.10.2016].

[26] AngularJS, ”AngularJS API Docs,” Google, [WWW]. Saatavissa: https://docs.angularjs.org/api/. [viitattu 21.9.2016].

[27] SeleniumHQ, ”GitHub - SeleniumHQ,” [WWW]. Saatavissa: https://github.com/SeleniumHQ/selenium. [viitattu 27.9.2016].

[28] T. Turunen, ”Why you should pay employees for free-time Open Source contributions,” Futurice, 23 helmikuu 2016. [WWW]. Saatavissa: http://futurice.com/blog/year-2015-in-company-sponsored-open-source. [viitattu 27.9.2016].

[29] M. Foundation, ”Bugzilla,” 22 marraskuu 2011. [WWW]. Saatavissa: https://bugzilla.mozilla.org. [viitattu 5.9.2016].

[30] Cypress.io, ”Testing, the way it should be,” [WWW]. Saatavissa: https://www.cypress.io/. [viitattu 4.11.2016].

# LIITE A: HAASTATELLUT HENKILÖT

Riikka Taiminen

Timo Mihaljov

Mikko Suonio

Antti Niemenpää

Juha Siponen

Jukka Siivonen

Juha Jokimäki