MODERNIEN VERKKOPOHJAISTEN KÄYTTÖLIITTYMÄTEKNOLOGIOIDEN AUTOMATISOITU TESTAUS

PRO GRADU– TUTKIELMA

Marko Rantala

Ihmisen ja teknologian välinen vuorovaikutus

Viestintätieteiden tiedekunta

Tampereen Yliopisto

Ohjaaja
Päivi Majaranta

11.5.2018

TIIVISTELMÄ

Tampereen yliopisto, Viestintätieteiden tiedekunta
Master's Degree Programme in Human-Technology Interaction
RANTALA, MARKO
Modernien verkkopohjaisten käyttöliittymäteknologioiden automatisoitu testaus.
Pro gradu –tutkielma
Filosofian maisterin tutkinto
Toukokuu 2018
Sivumäärä 55, liitteet 6 sivua

Tämä tutkielma esittää suosituksia siitä, kuinka toteuttaa modernien verkkopohjaisten käyttöliittymäteknologioiden avulla toteutettuihin sovelluksiin yksikkötestausta ja päästä-päähän –testausta. Lisäksi tutkielma perehtyy tutkielman toimeksiantajan Solita Oy:n tämän hetkisiin testaus-käytäntöihin haastattelun avulla ja esittää suosituksia siitä, miten testausta pystyttäisiin kehittämään tulevaisuudessa yrityksen ohjelmistoprojekteissa.

Tutkimus on toteutettu kvalitatiivisena eli laadullisena tutkimuksena. Tutkielma perehtyy kirjallisuuskatsauksen avulla verkkopohjaisten käyttöliittymien yksikkötestaukseen ja päästä-päähän –testaukseen. Automatisoidulla testauksella pyritään saavuttamaan sovelluksessa laaja testauskattavuus ja testauksen varianssi, joiden avulla pystytään varmistumaan sovelluksen kaikkien komponenttien toimivuudesta koko sovelluksen elinkaaren ajan.

Yksikkötestauksella tarkoitetaan eriytettyä testausta, joka kohdistuu ainoastaan sovelluksen pienimpiin toiminnallisiin osiin, eli esimerkiksi funktioihin. Yksikkötestauksella pyritään selvittämään, toimiiko kyseinen funktio tai komponentti oikein ilman riippuvuutta muihin sovelluksen komponentteihin.

Päästä-päähän –testauksessa toteutetaan emuloidun käyttöliittymän avulla käyttöliittymäkomponentin vaatimusmäärittelyyn, kuten käyttötapaukseen, perustuvaa

vuorovaikutusta käyttöliittymälle. Päästä-päähän –testauksella pyritään testaamaan komponentin vaatimusten täyttyminen ja niiden oikeellinen toiminta sovelluksen elinkaaren ajan.

Asiantuntijahaastattelun perusteella Solitan tulisi projekteissaan automatisoida käyttöliittymätestausta mahdollisimman laajasti, etenkin elinkaareltaan pidempiaikaisissa ohjelmistoprojekteissa. Yksikkötestejä tulisi toteuttaa kaikille sovellukseen toteutetuille toiminnallisille käyttöliittymän osille. Lisäksi käyttöliittymän komponenteille tulisi toteuttaa *Snapshot*-testejä, joilla pyritään varmistumaan komponenttien oikeellisesta tilasta. Snapshot –testeillä tarkoitetaan testejä, joissa verrataan halutun tilan DOM-puun, eli html-dokumentin rakenteen, representaatiota testauksen aikaiseen representaatioon DOM-puusta. Päästä-päähän –testausta projekteissa tulisi toteuttaa käyttötapausten perusteella ja päästä-päähän –testejä tulisi toteuttaa ainakin kaikille sovelluksen kriittisille toiminnallisuuksille, jotta pystytään varmistumaan sovelluksen kokonaistoimivuudesta.

SISÄLLYS

1. JOHDANTO 1

2. VERKKOSOVELLUSTEN AUTOMATISOIDUT TESTAUSMENETELMÄT: KIRJALLISUUSKATSAUS 6

2.1 Modernien verkkosovellusten testaus 6

2.2 Verkkosovellusten päästä-päähän –testaus 11

2.3 Verkkosovellusten käyttöliittymien yksikkötestaus 14

2.4 Yhteenveto 16

3. TESTAUKSEN TOTEUTUS 19

3.1 Verkkosovellus 19

3.2 Testaustyökaluista yleisesti ja työkalujen valinta 20

3.3 päästä-päähän –testaus 20

3.3.1 Nightwatch.js yleisesti 20

3.3.2 Toteutetun sovelluksen päästä-päähän –testaus 23

3.3.3 Nightcloud – testaus pilvestä 25

3.4 Yksikkötestaus 26

3.4.1 Jest 26

3.4.2 Enzyme 26

3.4.3 Yksikkötestien toteutus 28

3.5 Yhteenveto 31

4. ASIANTUNTIJAHAASTATTELUT 33

4.1 Tutkimuksen toteutus 33

4.2 Haastatteluiden tulokset 35

4.3 Haastattelun tuloksien tarkastelu ja analysointi 37

5. YHTEENVETO JA POHDINTA 42

5.1 Testaus Solitan kannalta tulevaisuudessa 43

5.1.1 Miksi testata? 44

5.1.2 Mitä tulisi testata? 46

5.1.3 Käytettävät teknologiat ja tulevaisuus 49

5.1.4 Lopuksi 52

LÄHDELUETTELO 54

LIITTEET 56

LIITE 1 Toteutetun ohjelmiston lähdekoodi 56

LIITE 2 Päästä-päähän –testaukseen käytetyn Nightwatch.js:n asennus ohjeet. 57

LIITE 3: Haastattelun runkoa ja kysymyspohja. 59

1. JOHDANTO

Aihe tähän maisterintutkielmaani syntyi kesän 2017 aikana, kun huomasimme työnantajani Solitan kehittäjäyhteisössä, että käyttöliittymäteknologioiden testaamiseen liittyvissä asioissa olisi parantamisen varaa. Pienimuotoisten kyselyiden perusteella huomasimme, että noin puolet kehittäjistä toteuttaa automatisoitua testausta nykyisissä projekteissa. Tästä kyselystä heränneessä jälkikeskustelussa useat kollegani Solitalla toivoivat, että he saisivat tietää enemmän käyttöliittymäteknologioiden testaamisesta ja siitä, kuinka testejä pystytään toteuttamaan moderneihin verkkopohjaisiin sovelluksiin. Etsin itse kyseiseen aikaan aihetta tähän tutkimukseen ja päädyin valitsemaan tämän aiheen, jolle on yrityksemme sisällä selvästi kiinnostusta ja tutkimuksen pohjalta toteutetut yrityksen sisäiset tietoiskut todennäköisesti hyödyttävät useita työntekijöitä. Toinen suuri motivaatio tälle tutkimukselle on se, että modernien verkkopohjaisten käyttöliittymien testaamisesta ei löydy yleispätevää tutkimusta. Tutustuessani testaukseen liittyvään kirjallisuuteen huomasin, että harvassa tutkimuksessa kerrotaan mitä käyttöliittymissä tulisi testata tai kuinka näitä testejä yleensä toteutetaan moderneilla teknologioilla. Useimmiten tutkimukset ja tieteelliset artikkelit esittelevät jonkin tietyn testausteknologian ja siihen toteutettuja algoritmeja tai jonkin tietyn ohjelmistokehyksen testaukseen liittyviä toteutuksia. Nämä tietyn ohjelmistokehyksen tarpeisiin tehdyt tutkimukset kuitenkin monesti jäävät pintapuolisiksi raapaisuiksi itse testauksen kannalta.

Nykypäivän verkkopohjainen ohjelmakehitys jaetaan pääsääntöisesti kahteen osaan; taustalogiikan kehitykseen (*back-end*) ja käyttöliittymien toteuttamiseen (*front-end*). Tämä tutkielma käsittelee testausta käyttöliittymätoteutusten näkökulmasta ja rajaa aiheen verkkopohjaisiin JavaScript-teknologioihin. Ohjelmistojen testauksella voidaan yleisesti ajatella tarkoitettavan toimia, joilla pyritään kartoittamaan ohjelmistossa esiintyviä ohjelmointivirheitä tai puutteita ja takaavan näin sovelluksen laadun ja että sovellus täyttää sille asetetut vaatimukset. (IEEE–729, 1983)

Ohjelmointivirheet voivat aiheuttaa virhetilanteita ohjelmistokoodin suorituksessa tai voivat aiheuttaa mahdollisia tietoturva-aukkoja, joita hypoteettiset tunkeutujat pystyvät hyödyntämään esimerkiksi kirjautumistietojen varastamiseen. Käyttöliittymätestauksessa voidaan ajatella harvoin esiintyvän suoranaisia tietoturva-aukkoja. Käyttöliittymätaso vastaa lopulta sovelluksen ulkonäöstä ja itsessään sovelluksen toiminnallisuuksien esittämisestä, joten taustalogiikan tulisi pitää sisällään tiedonkäsittelyyn liittyvät turvallisuudet. Tämä johtaa siihen, että käyttöliittymän ei tulisi ottaa kantaa siihen, onko käyttöliittymän käyttäjällä mahdollisesti oikeuksia nähtävän datan saamiseen. Esimerkiksi tapauksessa jossa käyttöliittymälle annetaan taustalogiikasta virheellistä dataa näytettäväksi, tulee käyttöliittymän näyttää näkymä, vaikka käyttäjällä ei olisikaan oikeuksia nähdä tätä dataa. Tässä tapauksessa taustajärjestelmän oikeuksienkäsittelyssä on tapahtunut virhe. Käyttöliittymiin on kuitenkin pakollista tehdä esimerkiksi erilaisia lomakkeiden syötteen validointeja, sillä käyttöliittymästä on turha lähettää aivan vääränlaista tietoa käsiteltäväksi, jos se on helposti estettävissä. Tämä liittyy olennaisena osana käyttöliittymätason tietoturvaan. Käyttöliittymätestauksen voidaan ajatella useimmiten liittyvän läheisesti käytettävyystestaukseen, sillä sen ensisijaisena tavoitteena on estää käyttöliittymän virhetilanteet, jotka aiheuttavat epäloogisuuksia toiminnassa, tai jopa aiheuttavat ohjelmiston suorituksen keskeytymisen.

Käytettävyystestaus on olennainen osa käyttöliittymien testausta, ja se kuuluu aina ohjelmistoprojekteihin, ainakin silmämääräisesti kehittäjän toimesta. Rajaan tutkielmani aiheen tässä yhteydessä pois käytettävyystestauksesta, sillä kyseessä olisi tiedon ja tehdyn tutkimuksen perusteella laajuudeltaan aivan oma aihe tutkielmalle. Tämä tutkielma on rajattu automatisoituun testaukseen ja siinä käyttöliittymäteknologioiden toteuttamisen aikaisiin testeihin. Käyttöliittymätestausta voidaan suorittaa myös manuaalisesti. Esimerkiksi jos käyttöliittymään toteutetaan painike, niin yleensä sen toimivuus testataan ainakin kehittäjän toimesta. Kehittämisen aikaisella automatisoidulla testauksella pyritään ennen kaikkea säästämään testaukseen käytettyä aikaa ja tarjoamaan laajempi testauksen monimuotoisuus toteutetuille ominaisuuksille. Testauksen monimuotoisuudella tarkoitetaan tässä tapauksessa esimerkiksi syötteen monimuotoisuutta; jos

testaaja käy manuaalisesti syöttämässä 1000 eri variaatioita salasanasta, kuluu siihen resursseja huomattavasti enemmän, kuin jos toteutetaan testiautomaationa nuo 1000 eri variaatiota. Testisyötteet generoidaan aina uusien muutosten yhteydessä satunnaisesti ja syötetään kirjautumislomakkeeseen automaattisesti, jolloin niitä ei tarvitse syöttää manuaalisesti lomakkeeseen.

Kehityksenaikainen testaus voidaan yksinkertaisimmillaan jaotella neljään eri kategoriaan, jotka ovat: Yksikkötestaus, päästä-päähän –testaus, integraatiotestaus ja visuaalinen testaus. Yksikkötestauksella tarkoitetaan ohjelman pienimmän osan, kuten esimerkiksi yksittäisten funktioiden testausta. Päästä-päähän –testauksella tarkoitetaan menetelmää, jossa testataan käyttöliittymäobjektien avulla kaikki sovelluksen toiminnallisuudet ja kuinka nämä erillään toteutetut toiminnallisuudet toimivat yhdessä. Integraatiotestauksella tarkoitetaan käytännössä sitä, että ohjelmistosta testataan se, että eri komponentit toimivat saumattomasti yhteen ja toimivat oikeellisesti. Integraatiotestaus voidaan usein helposti mieltää päästä-päähän –testauksen kanssa samaksi asiaksi, sillä molemmissa testausmenetelmissä pyritään selvittää kuinka sovelluksen toiminnallisuudet tai funktiot toimivat yhdessä. Suurin ero integraatiotestauksen ja päästä-päähän –testauksen välillä on se, että päästä-päähän –testaus toteutetaan käyttöliittymäobjekteja manipuloimalla, kun integraatiotestaus toteutetaan useimmiten yksikkötestauksen kaltaisesti suoraan funktiokutsuilla. Visuaalisella testauksella tarkoitetaan visuaalisesti esimerkiksi ruudunkaappausten avulla suoritettavaa vertailua käyttöliittymässä tapahtuvista muutoksista. Tutkielman ajatuksena on kuitenkin rajata aihe vielä suppeammaksi ja valita näistä testausparadigmoista ainoastaan kaksi; yksikkötestaus ja päästä-päähän –testaus. Tutkielman kontekstissa, Solitan ohjelmistokehityksessä, automatisoitu käyttöliittymätestaus toteutetaan useimmiten päästä-päähän –testien ja yksikkötestien avulla.

Tutkimusongelman sijasta, tutkielman on tarkoitus esitellä nykypäivän teknologioita ja selventää testausparadigmojen eroja ja tarjota ohjenuoria tämän hetken käyttöliittymäkehittäjille siihen, miksi heidän tulisi ylipäätään testata toteuttamiaan verkkosovelluksia ja kuinka testejä toteutetaan.

Tutkielma on tyypiltään laadullista tutkimusta, jonka tavoitteena on kerätä verkkopohjaisten käyttöliittymäteknologioiden testaukseen liittyvää materiaalia aineistoksi. Aineiston on tarkoitus kertoa, kuinka käyttöliittymätestausta toteutetaan nykypäivän verkkokehityksessä ja kuinka Solita pystyisi kehittämään omia käytäntöjään nykyisistä. Tutkimuksen teko on toteutettu kirjallisuuskatsauksen avulla, jossa on kerätty aineistoa tämän hetkisestä testauksesta. Lisäksi tutkielma pitää sisällään ohjelmistokoodin toteutusta, joka auttaa tutkielmaan syventyviä ymmärtämään testien toteutusta käytännön näkökulmasta, sekä asiantuntijahaastattelu testauksesta Solitan sisältä. Haastatteluosuuden avulla saadaan tietoa, kuinka työelämän asiantuntijat näkevät tämän hetkisen testauksen tilan. Haastatteluiden avulla saadaan myös selvyys kuinka asiantuntijat toteuttavat testausta projekteissaan.

Tutkielma koostuu neljästä osasta. Ensimmäisenä on kirjallisuuskatsaus, jossa perehdytään kirjallisuuden avulla yleisesti automatisoituun käyttöliittymätestaukseen, sekä tarkennetaan käsiteltäviä paradigmoja. Luvussa 2.1 kerrotaan modernien verkkopohjaisten sovellusten testauksesta ja osio antaa yleiskatsauksen nykyisiin teknologioihin. Yleisen osuuden jälkeen tutkielmassa syvennytään tarkemmin omissa osioissaan kahteen eri testausparadigmaan; luvussa 2.2 päästä-päähän –testaukseen (*end-to-end tests*) ja luvussa 2.3 yksikkötestaukseen (*unit tests*). Toinen osio pitää sisällään testien käytännön toteutusta, sillä testaukseen ei riitä pelkkä teoria, vaan testien toteuttamiseen tarvitaan myös käytännön esimerkkejä. Luku 3. siis käsittelee päästä-päähän –testausta ja yksikkötestausta toteutuksen näkökulmasta. Luku esittelee esimerkeillä, kuinka nykypäivänä käyttöliittymille toteutetaan testausta modernien JavaScript pohjaisten testauskehysten avulla. Tutkielman kolmannessa osiossa eli luvussa 4. syvennytään Solitan tämän hetkisiin käyttöliittymien testauskäytäntöihin ja selvitetään, kuinka testaustapoja pystytään jatkossa kehittämään yrityksen sisällä. Osion tarkoituksena on kartoittaa Solitan sisäisiä käytäntöjä asiantuntijahaastatteluiden avulla ja selvittää näin, mitä mieltä kokeneet ohjelmistokehittäjät ovat testauksen tarpeellisuudesta, kuinka he itse toteuttavat testausta projekteissaan ja miten he tulevaisuudessa kehittäisivät testauskäytäntöjä yrityksen sisällä. Viimeisessä osiossa, luvussa 5

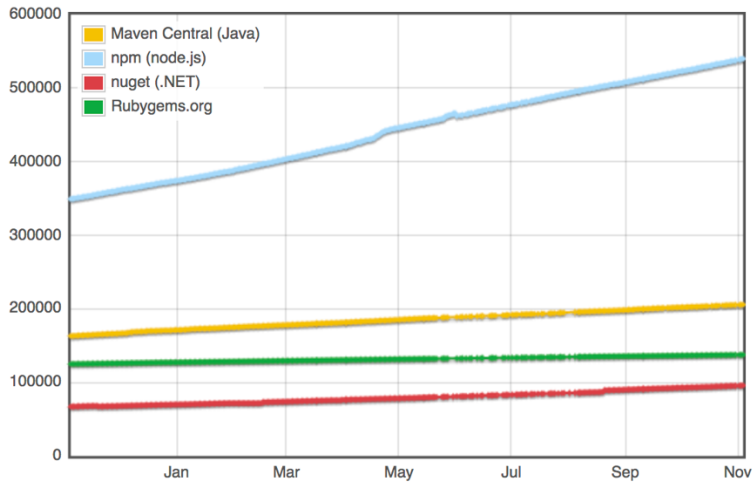
katsotaan hieman tulevaisuuteen ja tutkielma selvittää mihin suuntaan nykyistä käyttöliittymätestausta tulisi kehittää Solitan sisällä ja minkälaisia käytäntöjä käyttöliittymätestauksessa Solitan tulisi ottaa tulevaisuudessa huomioon.

2. VERKKOSOVELLUSTEN AUTOMATISOIDUT TESTAUSMENETELMÄT: KIRJALLISUUSKATSAUS

Tämä osion tarkoituksena on kartoittaa yleisesti modernien verkkosovellusten testaukseen liittyviä menetelmiä nykypäivän ohjelmistokehityksessä, sekä mitä teknologioita siihen yleisesti käytetään. Tämän jälkeen lukijalle esitellään tarkemmin yksikkötestausta ja päästä-päähän –testausta sekä perehdytään syvällisemmin siihen, mitä kyseiset testausmenetelmät tarkoittavat. Osion tiedonetsintä on suoritettu pääsääntöisesti seuraavilla tietokantapalveluilla; The ACM Digital Library, GoogleScholar, IEEE/IET Electronic Library, ScienceDirect (Elsevier), Google-haku ja npm-paketinhallinta-työkalun (*node package manager*, *https://www.npmjs.com/*) haulla.

2.1 Modernien verkkosovellusten testaus

Nykypäivän verkkosovelluskehitys kehittyy huimaa vauhtia niin teknologioiden kuin myös erilaisten ohjelmistokehysten, komponenttikirjastojen ja muiden JavaScript-kirjastojen osalta (Kaavio 1). JavaScript pohjaisessa verkkokehityksessä kaikista yleisimmin käytettyyn npm-paketinhallintajärjestelmään tulee keskimäärin 518 uutta pakettia päivässä. (Debill, 2017)



Kaavio 1. Tilastograafi suosituimmista paketinhallintajärjestelmien pakettien määrästä vuodelta 2017. (Lähde: http://www.modulecounts.com/)

Pakettien lisääntyessä käytössä olevia teknologiavaihtoehtoja tulee lisää ja kehittäjä voi valita näistä omiin tarpeisiinsa soveltuvimman. Tämä teknologioiden monimuotoisuus on kuitenkin johtanut myös ongelmiin esimerkiksi testauksen parissa; kuinka testata kaikki käytössä olevat ohjelmistokomponentit ja mitä teknologiaa testaukseen tulisi käyttää? Tällä hetkellä yleisimmin käytössä olevat teknologiat perustuvat Node.js ja Selenium–WebDriver teknologioiden päälle kehitettyihin sovelluksiin.

Node.js on JavaScript kehitysympäristö (*develeopment environment*), joka tarjoaa ajonaikaisen suorituksen JavaScriptille palvelinpuolelle ja mahdollistaa tämän avulla JavaScriptin suorituksen ja kääntämisen ennen kuin se saapuu selainpuolelle suoritettavaksi (Tilkov, 2010). Node.js on nykypäivän verkkosovellusten kehityksessä oleellinen osa, sillä se mahdollistaa modernien JavaScript pohjaisten ohjelmistokehysten suorittamisen ja sitä kautta verkkosovelluksien kehityksen suuressa mittakaavassa. (NodeJS, dokumentaatio)

Selenium–WebDriver on avoimen lähdekoodin MIT-lisenssillä toimiva testauskehys. WebDriverin tarkoituksena on mahdollistaa testien toteuttaminen teknologiasta, alustasta ja selaimesta riippumattomasti. Selenium–WebDriver mahdollistaa selaimen kaltaisen vuorovaikutuksen ja niiden manipuloinnin toteutetuille DOM-elementeille (*document object model*) ja näin ollen simuloi selainten avulla tehtävää vuorovaikutusta. DOM-elementeillä tarkoitetaan verkkosivun HTML-dokumentin esitystä objektipohjaisena mallina, jonka avulla dokumentin sisältöä pystytään ohjelmointikielen, kuten JavaScriptin avulla manipuloimaan dynaamisesti. (W3C, DOM). Selenium–WebDriverin pohjalta toteutettua W3C WebDriveria on ehdotettu W3C:n standardiksi, joka on tällä hetkellä vielä ehdotusvaiheessa. Ehdotusvaihe on viimeinen vaihe ennen virallista julkaisua standardiksi tai pääsyä W3C-organisaation viralliseksi suositukseksi. W3C perustelee standardoinnin pohjana käytettävää teknologiaa sillä, että Selenium–WebDriver on pitkäaikainen teknologia, jolla on testaukseen hyvät ja kattavat ominaisuudet, sekä teknologiaan perehtynyt, laaja käyttäjäkunta. (W3C, WebDriver)

Selenium-WebDriverin avulla voidaan siis toteuttaa ohjelmointikielen kaltaisella syntaksilla toteutettuja testejä, jolloin esimerkiksi DOM-elementit valitaan erilaisten yksilöintitietojen, kuten HTML-elementtien id- tai class-attribuuttien kautta. Toinen testien toteuttamisen tapa, mihin Selenium-WebDriver tarjoaa käyttöliittymän on *nauhoitus*. Nauhoituksen avulla testaaja pystyy nimensä mukaisesti nauhoittamaan, esimerkiksi selaimeen asennettavalla lisäosalla, vuorovaikutusta sivustolla; testaaja käy vuorovaikuttamassa eri elementtien kanssa sivustolla, kuten klikkaamassa eri elementtejä tai antamassa syötteitä syöte-kenttiin. Tämä vuorovaikutus tallennetaan Selenium-WebDriverin käyttöliittymän avulla, joka pystytään myöhemmin ajamaan uudestaan identtisenä vuorovaikutuksena sivustolla, aina uusien toimintalogiikan muutosten jälkeen. (Huggins et al., 2009)

Ohjelmistokehityksen muuttuessa testauksesta on tullut myös yhä tarpeellisempaa esimerkiksi nykyisten versionhallintatyökalujen arkipäiväistyessä ja ohjelmointiteknologioiden kehittyessä. Nykisin valloillaan oleva ohjelmistokehityksen malli, jatkuva integraatio (*Continuous Integration CI*) vaatii nykypäivän ohjelmistokehitykseltä jatkuvaa testausta. Syynä tähän on, että ohjelmistoprojekteissa useat kehittäjät tekevät yhteiseen versionhallintaan useita muutoksia (*commits*) päivässä ja tehdyt muutokset ovat usein yhteydessä toisiinsa. Samaan ohjelmistokoodipohjaan tehtyjen muutosten, mutta kuitenkin erillisesti toteutettujen ominaisuuksien yhteentoimivuuden varmistaminen on erittäin kriittistä yhteisen ympäristön toimivuuden kannalta, joka pohjautuu versionhallinnasta saatavaan yhteiseen koodisäilöön. (Fowler, 2006)

Yleisesti modernien verkkopohjaisten käyttöliittymäteknologioiden testaukseen liitetään vahvasti paradigma nimeltään TDD (*Test-driven development*) (Haviv, 2014). TDD-paradigmalla tarkoitetaan lähestymistapaa, jossa testit implementoidaan ennen itse ohjelmiston logiikkaa. Eli yksi lähestymistapa tämän toteutukselle on esimerkiksi se, että vuorovaikutussuunnittelija (IX Designer) toteuttaa sovelluksen toiminnoille käyttäjätarinat tai käyttötapaukset siitä, miten käyttäjät tulevat mahdollisesti sovellusta käyttämään ja kuinka sovellus käyttäytyy näissä eri vuorovaikutustilanteissa. Tämän jälkeen kehittäjä toteuttaa testit, jotka mallintavat käyttäjätarinoissa ilmentyvää vuorovaikutusta, jonka jälkeen alkaa vasta itse ohjelmiston komponenttien toteutus. Ohjelmiston

logiikka tulee toteuttaa, että logiikka täyttää testien avulla toteutetut käyttäjätarinoiden määrittelyt. Useasti TDD:n kaltaisen ohjelmistokehityksen voidaan ajatella olevan hyvä lähtökohta ohjelmistoprojektille.

TDD:stä sovellettu malli ATDD (*Acceptance test driven development*) on myös yleinen käytössä oleva paradigma modernissa ohjelmistokehityksessä (Solis & Wang, 2011). ATDD tarkoittaa käytännössä ohjelmiston pilkkomista osiin toteutettujen hyväksymistestien perusteella, jolloin yksi ohjelmiston kokonaisuus tai osa on valmis sen täyttäessä ja läpäistessä sille määritellyt ja toteutetut hyväksymistestit. ATDD:n voidaan ajatella soveltuvan hyvin ja on myös paljon käytetty ketterän ohjelmistokehitykseen projekteissa, johtuen paradigman tavasta jakaa ohjelmisto pienempiin osiin.

Kolmas testausparadigma, joka on laajentunut yleisesti nykypäivän ohjelmistokehityksen käyttöön, etenkin ketterissä ohjelmistoprojekteissa, on nimeltään BDD (*Behaviour driven development*) (Solis & Wang, 2011). BDD:tä pidetään usein hyvin soveltuvana mallina ketterään ohjelmistokehitykseen, johtuen BDD:ssä käytössä olevista lyhyistä kehityskierroksista (Solis & Wang 2011). Lyhyet kehityskierrokset tai pienet kehityskokonaisuudet, nimeltään sprintit ovat oleellinen osa ohjelmistoyrityksissä usein käytössä olevien ketterän kehityksen menetelmien ja kehyksien, kuten Scrum-kehyksen, metodologiaa. Ajateltuna BDD:tä yksinkertaistettuna, paradigman voidaan huomata koostuvan eri vaiheista. BDD:n ensimmäisessä vaiheessa toteutetaan jonkinlainen vaatimusmäärittely. Tämä vaatimusmäärittely toteutetaan useimmiten liiketoimintalähtöisesti, määritellen yrityksen tavoittelemia liiketoiminnallisia hyötyjä ja sitä kuinka tärkeinä näitä ohjelmiston ominaisuuksia pidetään. Tämän määrittelyn perusteella, kehittäjille määritellään toteuttava työjono ja arvotetaan toteuttavat ominaisuudet. Tämä tapahtuu niin, että bisneskriittisinten ominaisuuksien tulisi valmistua ensimmäisenä. Tämän jälkeen BDD:ssä toteutetaan käyttäjätarinat, jotka toimivat pohjana koko BDD:lle. Käyttäjätarinoiden avulla pystytään määrittämään ohjelmistolle vaadittavat ominaisuudet, eli niiden pohjalta toteutettavat hyväksymistestit ja näin ollen myös määrittely sille, milloin jokainen komponentti ja ohjelmiston osa ovat valmiita. Käyttäjätarinoiden pohjalta toteutetaan myös hyväksymistestit, jotka toimivat

ATDD-mallin kaltaisesti sovelluksen osien hyväksymisvaatimuksina. BDD:ssä oleellisena osana kuuluu hyväksymistestien toteuttaminen automatisoituna testauksena ja toteutettuihin testeihin luodaan yhteys dokumentaatioon. Tämä voidaan tehdä esimerkiksi nimeämisten kautta niin, että testien nimet tai testifunktioiden nimet vastaavat vaatimuksiin kirjattuja kohtia. Näin testitoteutuksen ja testimäärittelyn välinen yhteys ovat aina suoraan selvillä. Esimerkiksi käyttäjätarinan osan odotetaan olevan seuraavan kaltainen: ”Käyttäjä kirjoittaa ikä-kenttään 30, kentän perään ilmestyy vihreä oikeinmerkki ja käyttäjälle aukeaa ikä-kentän alle uusi kenttä, jossa lukee lempiväri”. (Fowler, 2013)

BDD-malli ehdottaa GivenWhenThen-lähestymistapaa itse testien toteuttamiseen, jonka tarkoituksena on selventää ja tuoda näkyviin mitä testataan. Ideana tässä mallissa on esittää funktion nimenä skenaarion alkutilanne, mitä tehdään ja lopuksi lopputulos. Esimerkiksi edellä mainittua käyttäjätarinaa voidaan analysoida pidemmälle. Lähtö tilanne on, että käyttäjän tule antaa ikänsä, eli yksikkötestaus funktion nimen alku voitaisiin ajatella olevan `syotaIka`. Tämän jälkeen määritellään, että mitä halutaan testata kyseisessä testissä. Tässä tapauksessa ikä voidaan määritellä seuraavasti: iän tulee olla positiivinen luku. Funktiomme nimen seuraava osa voisi olla `suurempiKuinNolla`. Viimeisenä vaiheena halutaan lisätä nimeen tieto testin lopputuloksesta, eli mikä kyseisen testin haluttu lopputulos on. Tässä tapauksessa funktion paluuarvona tulee olemaan totuusarvo: `true` tai `false`, ja koska haluamme olettamuksen `ikä > 0` niin palautusarvo olisi true, tällöin funktion nimeksi muodostuisi: `syotaIka_suurempiKuinNolla_true`. (Solis & Wang, 2011)

Viimeinen BDD:n kohta liittyy ohjelmistokoodin toteutukseen, jonka tulisi toimia itsenäisenä dokumentaationa (*Readable Behaviour Oriented Specification Code*). Tämä lähestymistapa tarkoittaa sitä, että kirjoitettaessa ohjelmistokoodia, kehittäjien tulisi käyttää riittävissä määrin kommentointia, sekä ohjelmiston metodien ja muuttujien tulisi olla niin selkeästi nimettyjä ja toteutustaan kuvaavia, että erillistä dokumentaatiota ohjelmistokoodin rakenteesta ja mitä tiedostoissa tapahtuu, ei tarvita. (Solis & Wang, 2011)

2.2 Verkkosovellusten päästä-päähän –testaus

päästä-päähän –testauksella tarkoitetaan testausparadigmaa, jonka tarkoituksena on testata sovelluksen toiminnallisuus alkupäästä aina loppupäähän asti ja tarkistaa, että ohjelmisto toimii oikein. Moderneissa verkkosovelluksissa päästä-päähän –testaus toteutetaan useimmiten ainoastaan käyttöliittymän manipulaation avulla (*View, MVC-malli*) (Fat et al., 2016). Yksinkertaistettuna tämä tarkoittaa, että tarkoituksena on testata kaikki asiat mitä käyttäjä näkee sivustolla ensilatauksesta aina käyttösession loppuun asti. Kooditasolla tämä tarkoittaa, että testataan edellä mainittujen komponenttien yhteen toimivuus saumattomasti yhteen. Käytännössä päästä-päähän –testaus toteutetaan niin, että valittu testaustyökalu käy emuloidun selaimen avulla kaikki sovellukseen toteutetut näkymät läpi ja käy toteuttamassa näiden näkymien elementteihin halutut vuorovaikutukset. Vuorovaikutus voi olla painikkeiden käyttöä, pyyhkäisyjä tai generoitujen syötteiden antamista niille tarkoitetuille elementeille tai testattavan käyttöliittymän tilaobjektien tämän hetkisen tilan oikeellisuuden tarkistamista (Van Deursen, 2015). Syötteiden annon jälkeen testaustyökalu tarkistaa, että tapahtuuko kyseisistä toiminnoista käyttäjälle haluttu palaute, kuten hyväksymispalaute toiminnon onnistumisesta tai näkymän siirtyminen seuraavaan. Yleisimpiä testattavia tilanteita ovat käyttöliittymäkomponenttien renderöinti ja HTTP-protokollan tarjoamat kutsut (esimerkiksi POST ja GET), joiden avulla välitetään muokattavaa ja näytettävää dataa käyttöliittymän ja taustajärjestelmän välillä. Yllä oleva voidaan muuttaa päästä-päähän –testitapaukseksi, esimerkiksi verkkokaupan ostotapahtumaksi:

1) Käyttäjä selaa tuotteita (Sivulla on näkyvissä lista käyttöliittymäkomponentteja, jotka esittävät tuotteita)
2) Käyttäjä painaa tuotteen ”Lisää koriin” –painiketta (Halutun tuotteen tiedot lisätään ostoskorin tilaan ja yksilöidään tunnuksella)

3) Käyttäjä näkee, että tuote on siirtynyt ostoskoriin ja ostoskorissa olevien tuotteiden määrä vaihtuu suuremmaksi (Käyttäjälle näytetään muuttunut ostoskorin tilaobjekti, jossa on yksi tuote)
4) Käyttäjä painaa ”Ostoskoriin” –painiketta (Käyttäjän näkymä vaihtuu ostoskoriin)
5) Käyttäjä näkee ostoskori-näkymässä lisäämänsä tuotteen ja sen jälkeen painaa ”Siirry tilaamaan” –painiketta (Käyttäjän näkymä vaihtuu yhteystietonäkymään, jossa on useita käyttöliittymä komponentteja. Sivulle lisätään yhteystieto-komponentit, sekä maksutapa-komponentit sen perustella, onko tuotteelle saatavissa kyseinen maksutapa)
6) Käyttäjä valitsee ”Nimi” –kentän ja kirjoittaa siihen nimensä. (Käyttäjän antamat syötteet tarkistetaan ja tallennetaan maksutapahtuman asiakastiedot-objektin tilaan, kun käyttäjä on ne kenttään lisännyt.)
7) Täytettyään kaikki yhteystiedot, käyttäjä valitsee maksutavan, painamalla ”Lasku” –painiketta maksutavat kohdassa. (Käyttäjän valitsema maksutapa tallentuu maksutapahtuman laskutavaksi, käyttäjälle näytetään ”Vahvista tilaus” –painike kun kaikki pakolliset tiedot ovat täytetty oikein)
8) Käyttäjä painaa ”Vahvista tilaus” –painiketta (Käyttäjän antamat tiedot muutetaan POST-kutsun parametreiksi ja lähetetään taustajärjestelmälle käsiteltäväksi. Taustajärjestelmästä palautettu vastaus käsitellään ja käyttäjän näkymä päivitetään joko, hyväksymissivuun tai epäonnistunut tilaussivuun, riippuen siitä onko kyseinen tilaus mennyt läpi)
9) Käyttäjä näkee yhteenvedon tekemästään tilauksesta ja ilmoituksen ”Tekemäsi tilaus on otettu vastaan onnistuneesti”.

Yllä olevassa esimerkissä kuvataan verkkosovelluksen toimintalogiikkaa. Tarkoituksena on kertoa mitä käyttäjä tekee sivustolla ja tämän jälkeen sulkujen sisällä on käyttöliittymätasologiikan yksinkertaistettu kuvaus siitä, mitä kyseinen toiminto tekee ohjelmallisesti. Käyttöliittymätason logiikka on käytännössä se, mihin testaus päästä-päähän –testauksessa halutaan kohdentaa.

päästä-päähän –testauksen tarkoituksena onkin tarkastaa, että nämä kaksi järjestelmää toimivat saumattomasti yhteen. Esimerkiksi jos taustajärjestelmä välittääkin vääränlaisen JSON-objektin (*JavaScript Object Notation*) käyttöliittymälle, jonka avulla sovellus rakentaa käyttöliittymäkomponentit, tulee näkymä näyttämään virheelliseltä. Vaihtoehtoisesti voidaan myös testata, palauttaako käyttöliittymä vääränlaisen JSON-objektin lomake-datasta, jolloin tietojen tallennus taustajärjestelmässä epäonnistuu tai tapahtuu virheellisesti. (Haviv, 2014)

päästä-päähän –testauksella tavoitellaan automatisoitua käyttöliittymän käytön testausta. Sen sijaan, että kehittäjä tai testaaja kävisi ohjelmistokoodin muutosten jälkeen itse kokeilemassa manuaalisesti eri selaimilla esimerkiksi kirjautumislomakkeen toimintaa. Päästä-päähän –ohjelmistokehys toteuttaa sen automaattisesti joka kerta, esimerkiksi vietäessä muutokset CI-ympäristöön tai vaihtoehtoisesti aina ohjelmakoodin käännöksen yhteydessä (esimerkiksi ajamalla testit npm:n tai muun suorittajan avulla). Edellä mainitussa esimerkkitapauksessa toteutetut testit testaisivat seuraavia asioita: Kävisivät kirjautumislomakkeen syötekentät läpi, validoisivat syötteen oikeellisuuden, verifioisivat välitettävän JSON-objektin tai listan muodon, jotta se olisi kaikissa testitapauksissa oikeellinen, suhteessa haluttuun muotoon ja lopuksi tarkastaisivat vastauksen oikeellisuuden. Oikeellisuuden tarkistamisessa varmistetaan menikö testaus kirjautuminen läpi vai ei. Lisäksi tarkistetaan näytettiinkö käyttäjälle oikeanlainen palaute kyseisestä kirjautumistoiminnosta käyttäjälle. Muita yleisiä testauskohteita komponenttien renderöinnin, validoinnin, verifioinnin ja interaktiivisten elementtien toiminnan testaamisen lisäksi ovat sivuston linkkien ja sitä kautta sivuston reittien toimivuuden testaus ja tyylien oikeellinen asettuminen.

Linkkien ja reittien testauksella tarkoitetaan päästä-päähän –testauksessa sitä, että automatisoidut testit käyvät kaikki linkit läpi ja varmistavat, että käyttäjä pääsee näistä kaikista linkeistä käsiksi johonkin sivuston sisältöön. Reittien testaamisella tarkoitetaan, että käyttäjälle tarjotut linkkipolut toimivat ja ovat loogisia. Esimerkiksi voidaan ajatella verkkokaupan ostotapahtumaputkea reittinä, jossa eri näkymät päivittyvät ja samoin URL-osoite:

1) Käyttäjä siirtyy ostoskoriin (/ostoskori/)
2) Käyttäjä siirtyy ostoskorista tilaussivulle täyttämään tilaajatiedot (/yhteystiedot/)
3) Käyttäjä siirtyy tilaussivulta maksutavan valintaan (/maksutapa/)
4) Käyttäjä siirtyy maksamaan (käyttäjä siirtyy pankin omaan maksupalveluun)
5) Käyttäjä palaa maksamasta vahvistussivulle (käyttäjä siirtyy pankin sivulta takaisin ostotapahtumasivulle esimerkiksi: (/tilausvahvistus?success=true)

Valittu esimerkki on reitti joka sisältää useita eri näkymiä, mutta useimmiten reitit sisältävät 1-2 eri näkymää jotka muodostuvat käyttäjän valinnoista.

Tyylien oikeellisella asettumisella tarkoitetaan testauksessa sitä, että elementille määritellyt tyylit tulevat voimaan halutulle elementille. Karkea esimerkki tästä on, että jos elementille määritellään luokka `.f-red` testaustyökalu tarkistaa, että elementin tekstin väri on punainen. Hyödyllisempää tyylien testaus on tilanteissa, joissa tyylit esimerkiksi piilottavat tai tuovat esiin sivuston osia. Esimerkiksi tilanne, jossa käyttäjällä on ”Näytä lisää”–painike, jota painamalla käyttäjälle esimerkiksi tuodaan sisältöelementti näkyviin. Joissain tapauksissa tämänkaltainen toiminnallisuus voidaan tehdä tyylien avulla esimerkiksi tyylisäännöllä `display: none;`, joka piilottaa elementin näkyvistä. Esimerkin kaltaisissa tilanteissa tyylisääntöjä on pakollista testata, johtuen siitä, että tyylien oikeellinen asettuminen vaikuttaa itse sisällön näkymiseen ja käyttäjän käytettävyyteen kriittisellä tavalla. Tämän kaltainen toiminta on kriittistä, verrattuna esimerkiksi siihen onko jokin ohjelmassa esiintyvä tekstinpätkä lihavoitu tai kursivoitu tai näkyykö jokin painike pyöreillä reunoilla vai ei. Yleensä värit ja muotoilut tyyleissä eivät kuitenkaan estä käyttöliittymän käyttöä, vaan ovat pikemminkin kosmeettisia ongelmia (Nielsen, 1995).

2.3 Verkkosovellusten käyttöliittymien yksikkötestaus

Yksikkötestauksella (*unit testing*) tarkoitetaan ohjelmiston yksittäisten komponenttien tai esimerkiksi yksittäisten funktioiden testausta automaattisesti ja mielellään monimuotoisella, esimerkiksi generoidulla syötteellä. Yksikkötestauksella on tarkoitus varmistua siitä, että kyseinen ohjelmiston osa vastaa sen alustavaa määrittelyä ja toimii niin kuin sen on tarkoitettu toimivan. Testit toteutetaan ohjelmallisesti kehittäjien toimesta ja ne toteutetaan puhtaasti, eli tarkoituksena on testata osat riippumatta muista ohjelmiston osista ja pyrkiä siihen, että esimerkiksi kyseinen funktio tekee juuri sen mitä halutaan (Runeson, 2006). Erona päästä-päähän –testauksen ja yksikkötestauksen välillä on se, että yksikkötesteillä on pääsy ohjain- (*controller, MVC-malli*) ja mallitasoille (*model, MVC-malli*) ohjelmiston rakenteessa (Fat et al., 2016). Pääsy taustalogiikkaan mahdollistaa ohjelmiston moduulien testaamisen suoraan esimerkiksi funktioille välitetyillä arvoilla, ilman käyttöliittymätason manipulointia. Yleisesti puhuttaessa ohjelmistotestauksesta kokonaisuutena, usein tarkoitetaan pelkkää yksikkötestausta. Mitä yksikkötestaus sitten käytännössä oikein on? Esimerkiksi testauksen tarkoituksena on tarkistaa, jos funktion tarkoituksena on palauttaa jokin tietty arvo, riippumatta siitä onko se oikeellinen vai ei. Kaaviossa 2 on yksinkertaistettu esimerkki yllä olevan kaltaisesta tilanteesta esitettynä ”tyhmänä funktiona” (joka palauttaa pelkän arvon), josta voidaan testata, palauttaako osio aina luvun kolme.

```
const getThree = () => { return 3 }
```

Kaavio 2. Funktio-getThree palauttaa kaikissa tapauksissa numeron kolme.

Kaavion 2 esimerkki voisi olla esimerkiksi ohjelmistokoodin osa verkkopohjaisesta laskimesta, joka palauttaa aina luvun kolme, kun käyttäjä painaa painiketta tai painaa näppäimistön näppäintä jossa on kolme. Tämä tarkoittaa, että myös kyseinen funktio tulisi testata ja tarkistaa, että kaikissa tapauksissa palautetaan numero kolme.

Jos ajatellaan oikeiden ohjelmistoprojektien kehitystä, niin yksikkötestauksen tarkoituksena on useimmiten testata monimutkaisempia funktioita jotka muokkaavat tai muuttavat funktion sisällä

jotain dataa. Jos sovelletaan tätä yllä mainittuun laskin-esimerkkiin, niin multiply-funktiolle voitaisiin oletettavasti välittää parametrina X ja Y. Tällöin funktion tarkoitus on tarjota kutsujalle välitettyjen parametrien, eli tekijöidensä tulo. Tämän funktion tapauksessa tulisi testata, palauttaako funktio kaikilla parametreina välitetyillä numeroarvoilla tekijöidensä tulon. Yksikkötestauksessa tulee myös selvittää mitä funktiolle tapahtuu, jos sinne välitetäänkin parametrina esimerkiksi null tai väärän muotoinen arvo. Muita tarkistuksia pitää myös tehdä; kuinka kyseinen funktio käsittelee desimaalilukuja tai JavaScriptin käyttämän suurimman luvun (2^{53} - 1) ulkopuolisia arvoja. (IEEE–754, 1985) Nämä kaikki erilaiset arvot toteutetaan syötteellä, joka ottaa huomioon suuren määrän vaihtoehtoja, ovat monimuotoisia ja käyvät mahdolliset eri tilanteet läpi tarpeeksi kattavasti.

Yksikkötestauksen yhtenä ongelmana voidaankin pitää pienimmän testattavan osan määrittelyä eli yksikköä. Usein kehittäjät saattavatkin ajatella yksiköksi osa-alueita, jotka sisältävät useita toiminnallisia komponentteja tai funktioita. Tutkimuksen mukaan yksikkötesteillä ajatellaan olevan vaikea testata monimutkaisista algoritmeista ja tietorakenteista riippuvaisia komponentteja, sekä datan tai tilan käsittelyyn erikoistuneita osia. Paradoksina tai haasteena yksikkötestauksen osalta voidaan ajatella olevan se, kuinka ohjelmiston sisällä pystytään testaamaan toisistaan riippuvaisia osia erillään toisistaan. Muita vaikeuksia mitä yksikkötestauksessa ajatellaan olevan ovat: Testauskehyksen integrointi ohjelmistoon mukaan, määrittely milloin yksikkötesti on oikeasti läpäisty hyväksytysti, sekä yksikkötestien ylläpitäminen ajan tasalla. (Runeson, 2006)

Tarkempia esimerkkejä testauskehysten integroinnista ja testien toteuttamisesta saamme luvussa 3.

2.4 Yhteenveto

Aiemmin luvussa 2 esittelin useamman eri suunnittelumenetelmän, joiden toteutukseen testaus liittyy vahvasti osana. Esittelin myös yleisesti mitä tarkoitetaan päästä-päähän – ja yksikkötestauksella, sekä esimerkkien kautta syvennyin mitä näillä paradigmoilla pitäisi testata.

Testauksella pyritään siis ensi sijaisesti välttämään ohjelmiston toteutuksessa tapahtuvia virheitä. Virheet voivat olla joko toimintalogiikkaan liittyviä, jolloin ne saattavat vaarantaa ohjelmiston tietoturvaa tai käytettävyyttä estämällä ohjelmiston suorituksen tai virheet voivat olla pelkästään kosmeettisia, jolloin ne yleensä eivät aiheuta muuta kuin esteettisiä ongelmia.

Automatisoidulla testauksella pyritään ennen kaikkea vähentämään testaukseen käytettävää aikaa, sekä laajentamaan tehtävien testien lukumäärää. Automatisoidun testauksen ei ole tarkoitus tehdä perinteistä käytettävyystestausta tarpeettomaksi, vaan pikemminkin toimia itse kehitysvaiheessa kehittäjien testausvälineenä, sekä välineenä jolla pystytään vakioimaan ominaisuuksien toiminnallisuudet. Tällä tarkoitan sitä, että toteutettaessa ominaisuutta sille on esimerkiksi toteutettu päästä-päähän –testaus hyväksymistestauksen osana, jolloin kyseinen käyttöliittymäkomponentti on valmis sen läpäistessä hyväksymistestit. Jatkuvan integraation mallissa (CI) ohjelmiston osat toteutetaan pienemmissä osissa ja julkaistaan aina sopivan kokonaisuuden valmistuttua, kuten esimerkiksi Lean-ajattelutavan mukaisessa ohjelmistokehityksessä MVP:n (*minimum viable product*) valmistuttua (Ries, 2009). Tämä MVP tarkoittaa käytännössä sitä, että toteutettaessa kokonaisuutta, suhteutetaan kokonaisuuden vaatima työmäärä halutun julkaisuaikataulun kanssa. Toteutettava kokonaisuus pilkotaan sopivan kokoisiksi osiksi (Scrum-mallissa 1-2 viikon toteutusta vastaaviksi osakokonaisuuksiksi). Aikataululle valitaan näistä pilkotuista osakokonaisuuksista tärkeimpiä (esimerkiksi BDD-mallissa toiminnallisuuksien kannalta tärkeimmät) osia niin paljon kuin annetussa ajassa ehditään toteuttaa, toteutetaan ne ensin ja julkaistaan ne. Tämän jälkeen otetaan työjonosta seuraavaksi tärkeimmät ominaisuudet, toteutetaan ne ja julkaistaan. Tämän esimerkin kannalta tärkeää on, että aikaisemmin toteutetut ominaisuudet toimivat oikein myös tulevia osia kehitettäessä. Tässä kohtaa edellä mainitsemani ohjelmiston eri osien testien vakioiminen on tärkeässä osassa, jotta ohjelmiston toimivuus pystytään varmistamaan ja kaikki aiemmin toteutetut komponentit pysyvät toimivina ja komponenttien toteutukset eivät riko toisiaan. Tätä on käytännössä mahdotonta toteuttaa manuaalisesti, etenkään yhtään laajemmissa ohjelmistoprojekteissa. Automaattisten testien ajaminen kehityksen yhteydessä tarjoaa tähän siis

kustannustehokkaan työkalun, jolla huomataan mahdolliset sivuttaisvaikutukset toteutettujen komponenttien toiminnassa.

3. TESTAUKSEN TOTEUTUS

Luvussa 3. syvennytään tarkemmin, kuinka luvussa 2. esiteltyjä yksikkötestejä ja päästä-päähän – testejä toteutetaan moderneille verkkosovelluksille. Luvussa 3.1 toteutetun sovelluksen lähdekoodi löytyy versionhallinasta, johon löytyy linkki Liitteestä 1.

3.1 Verkkosovellus

Testauksen toteuttamiseen tarvitaan sovellus, jonka toimintalogiikkaa voidaan testata. Tämän kappaleen mahdollistamiseksi osana tutkielmaa on toteutettu SPA–sovelluksen (*single page application*) käyttöliittymä, jonka toimintalogiikan varmistamiseksi toteutetaan päästä-päähän – ja yksikkötestit.

Sovellus on toteutettu JavaScript-ohjelmointikielellä, jossa on käytetty ECMAScript 6-syntaksia. Sovelluksen toteuttamiseen on käytetty React-komponenttikirjastoa ja Create-react-app-rakennustyökalua, joka auttaa kehittäjiä luomaan ja toteuttamaan React-pohjaisia verkkosovelluksia ilman, että kehittäjän tarvitsee konfiguroida ohjelmistonasetukset ja riippuvuudet manuaalisesti. (Facebook, Inc. React.js)

Toteutettava sovellus on monisivuinen lomake (*multi step form*), joka sisältää tavallisia syötekenttiä, joihin käyttäjä pystyy lisäämään nimimerkin, sähköpostiosoitteen, ja kirjoittamaan palautetta. Lähetyksen jälkeen käyttäjälle näytetään palautelomakkeen lähetyksen onnistumisesta tai epäonnistumisesta. Koska kyseessä on pelkkä käyttöliittymätoteutus, on viestin lähetyksen toteutettu satunnaisesti arvottava palaute. Toteutus arpoo jokaisen lähetyksen kohdalla onnistuuko palautteen lähetys vai ei. Tarkoituksena toteutetulla sovelluksella on ainoastaan tarjota yksinkertainen alusta missä käyttäjä pystyy lisäämään vapaata tekstisyötettä (nimimerkki, sähköposti ja palaute), tarkoin määriteltyä tekstisyötettä (sähköposti muotoa: x@x.x), sekä käyttämään painiketta (”Lähetä”–painike). Yksinkertaisen lomakkeen syötteen validoinnit ja sen pohjalta näytettävät käyttöliittymäelementit, kuten virheviestit ja ”Lähetä”–painikkeen käytön salliminen, soveltuvat

hyvin niin päästä-päähän –testauksen, kuin myös yksikkötestien testitapauksien toteuttamiseen ja oikeiden testitoteutusten simulointiin.

3.2 Testaustyökaluista yleisesti ja työkalujen valinta

Modernien verkkopohjaisten testaustyökalujen määrä on valtava ja lähes kaikki alkavat huomattavasti muistuttaa nykypäivänä jo toisiaan. Yksikkötestien toteuttamisteknologiaksi valikoitui *Jest* ja siihen toteutettu työkalu nimeltään *Enzyme* (lisää luvussa 3.3.2). Tässä tapauksessa voisimme käyttää myös aivan yhtä hyvin *Mocha* ja *Chai*, *Karma* tai *Jasmine* nimisiä työkaluja testien toteuttamiseen. Tässä tutkielmassa Jest-teknologian valintaan suurin syy on siinä, että se toimii saumattomasti yhteen *create-react-appin* kautta. Lisäksi teknologia on samoilta toteuttajilta kuin itse React-komponenttikirjasto. Enzymen käyttöön suurin syy oli Leland Richardsonin artikkeli (Richardson, 2016), jossa kerrotaan mitä ongelmia Airbnb-yhtiö koki testauksessa Reactin kanssa ja miksi yhtiö päätyi toteuttamaan testauksen avuksi Enzyme-työkalun. Päästä-päähän –testauksessa Nightwatch.js:n vaihtoehdoksi oli ainoastaan Googlen toteuttama *Puppeteer*-testauskehys. Tämän tutkielman rajauksena on katsoa testauskehyksiä etenkin Solita Oy:n näkökulmasta ja Nightwatch.js päästä-päähän –testauskehys on yrityksessä käytössä useammassa projektissa, joten kyseisen teknologian valintaan suurin syy oli siinä.

3.3 päästä-päähän –testaus

3.3.1 Nightwatch.js yleisesti

Nightwatch.js on päästä-päähän –testaukseen tarkoitettu testaukehys, joka on toteutettu Node.js teknologialla ja joka toteuttaa Selenium–WebDriveriin pohjautuvaa W3C WebDriver API-standardia. Nightwatch.js käyttää tätä standardiin pohjautuvaa REST–arkkitehtuuria, sillä sen avulla testauskehys pystyy kommunikoimaan palvelimen kanssa ja näin tarjoamaan

selaimentoimintoihin liittyviä toiminnallisuuksia, kuten linkkien avaamisen. (http://nightwatchjs.org/)

Nightwatch.js tarvitsee toimiakseen Selenium–WebDriverin, joka tarjoaa ajon aikaisen ympäristön Nightwatch.js:n ajamiseen. Lisäksi testien ajamiseen tarvitaan selainajurit (*browser drivers*) kullekin selaimelle, jota vastaan tehdyt testit halutaan suorittaa. Tarkemmat ohjeet Nightwatch.js käyttöönottoon löytyvät liitteenä liitteestä 1.

Nightwatch.js tarjoaa käyttäjälleen mahdollisuuden automatisoituun käyttöliittymätestaukseen. Tämä tarkoittaa käytännössä sitä, että Nightwatch.js:n avulla simuloidaan DOM-puuta manipuloimalla käyttäjän vuorovaikutusta toteutettuun sovellukseen. Nightwatch.js tarjoaa tähän oman ohjelmointirajapinnan, eli *api*n, jota käyttämällä testien toteuttaja pystyy helposti ajamaan testejä, eli manipuloimaan käyttöliittymäelementtejä DOM-puun avulla jokaisella kerralla samalla tavalla. Otetaan esimerkiksi tavallinen HTML-käyttöliittymään toteutettu painike: `<button id=”button” type=”submit”>Submit</button>`. Painikkeilla on aina jokin tarkoitus ja edellä oleva html-komponentti toimii lähetä-painikkeena. Toteutettaessa esimerkiksi tämän painikkeen käyttöliittymää ja sen toiminnallisuutta, tulisi painikkeen toiminnallisuus testata joka kerta, kun sen riippuvuuteen tulee muutoksia. Manuaalisesti tämä tapahtuu niin, että ohjelmistokehittäjä tekee muutoksen ohjelmistokoodiin, tallentaa koodimuutokset, siirtyy selaimeen, käy klikkaamassa käyttöliittymässä olevaa painiketta ja varmistaa, että painike tekee jotain. Tämä voi tapahtua kehitysprosessin aikana kymmeniä tai jopa satoja kertoja. Tähän tulee avuksi päästä-päähän –testaus ja tässä tapauksessa Nightwatch.js:llä kyseinen painikkeen painallus tapahtuisi seuraavasti:

```
browser.click("#submit", function(response) { ... });
```

Kaavio 3. Esimerkki funktio lomakkeen lähetyksestä, toteutettuna Nightwatch.js:llä

Kaaviossa 3 `browser.click` on funktio, joka toimii toimintakäskynä: selain, paina. ”#submit” tarkoittaa HTML-elementin yksilöivää tunnusta, eli id:tä, jolla vuorovaikutus pystytään kohdistamaan ainoastaan tähän painikkeeseen. Osassa ”`function(response){…}`” määritellään vapaaehtoinen takaisinkutsu-funktio (*callback*), eli ohjelmiston osa joka välitetään parametrina toiselle funktiolle. Tässä tapauksessa Nightwatch.js:n tarjoamalle `browser.click`-funktiolle. Takaisinkutsu-funktio suoritetaan sen jälkeen, kun ensisijainen funktio on suoritettu ja sille välitetään useimmiten ensimmäisen funktion paluu-arvo. Paluuarvosta voidaan tarkistaa ensimmäisen funktion vastaus, tai tehdä muuta ensimmäisestä funktiosta riippuvaisia toimenpiteitä. Eli kaavion esimerkissä submit-napin painalluksesta voidaan ajatella seuraavan POST-kutsu, josta voitaisiin haluta tietää kyseisen HTTP-kutsun paluuarvo, otsikot (*request headers*) tai vaikkapa käyttöliittymä tason muutos, kuten onnistumisviestin näyttäminen lähetyksen onnistumisesta.

Automatisoitutestaus pystytään nimensä mukaisesti automatisoimaan ja näin suorittamaan tehtävät automaattisesti. Tehtävät voidaan suorittaa esimerkiksi joka kerta kun kehittäjä tallentaa jonkin kooditiedoston tai kun kehittäjä lähettäessä (*push*) versionhallintaan muutoksensa. Tällöin ohjelmistokehittäjän ei tarvitse muutosta tehdessään käydä itse vuorovaikuttamassa käyttöliittymän kanssa. Pelkästään yksittäisen painikkeen toteutukselle ei ole ajankäytöllisesti järkevää toteuttaa automatisoituja testejä. Jos kyseessä on kuitenkin monisivuinen lomake, mihin tulee täyttää tietyn tyyppiset syötteet ja testata näiden kenttien validointia, niin testauksen toteutus on järkevä vaihtoehto. Tämä johtuu siitä, että useiden kymmenien sekuntien käyttäminen lomakkeen testaamiseen voidaan automatisoida ja suorittaa joka kerta muutamassa sekunnissa tai sekunnin sadasosassa sen sijasta, että aikaa käytettäisiin kymmenen sekuntia syötteiden antamiseen manuaalisesti. Todellisen ohjelmistoprojektin aikana tämä tarkoittaisi tuon kymmenen sekunnin käyttämistä lomakkeen täyttämiseen useita kymmeniä tai satoja kertoja. Lisäksi automatisoidun testauksen suorittaminen ja testitapaukset eivät ole enää kehittäjän muistin varassa. Näin pystytään varmistumaan, että samat testit suoritetaan joka kerta.

3.3.2 Toteutetun sovelluksen päästä-päähän –testaus

Toteutetulle sovellukselle on toteutettu Nightwatch.js:n avulla päästä-päähän –testit, jotka löytyvät kaaviosta 4.

```
1  module.exports = {
2    'Form input test' : function (browser) {
3      browser
4        .url('http://localhost:3000/')
5        .waitForElementVisible('body', 1000),
6      browser.pause(1000)
7      browser.setValue('.name--input', 'nightwatch');
8      browser.expect.element('.name--error').to.have.css('display').which.equals('none');
9      browser.setValue('.email--input', 'nightwatch');
10     browser.expect.element('.email--error').to.have.css('display').which.equals('block');
11     browser.clearValue('.email--input')
12     browser.pause(500)
13
14     browser.assert.attributeContains('.submit-button', 'class','disabled')
15     browser.expect.element('.submit-button').to.have.css('pointer-events').which.equals('none');
16
17     browser.setValue('.email--input', 'nightwatch@nightwatch.fi')
18     browser.expect.element('.email--error').to.have.css('display').which.equals('none');
19     browser.setValue('.form--textarea', 'I got nothing to say')
20     browser.expect.element('.submit-button').to.have.css('pointer-events').which.equals('auto');
21
22     browser.assert.cssClassNotPresent('.response-wrapper','show-response')
23     browser.click(".submit-button", function(response) {
24       console.log(response)
25     });
26     browser.assert.cssClassPresent('.response-wrapper','show-response')
27     browser.pause(200)
28       .end();
29   }
30 };
```

Kaavio 4. Sovellukselle toteutettu päästä-päähän –testauksen lähdekoodi

Kaavion 4 toteutus toimii käytännössä selaimessa seuraavina käyttäjän toiminnallisuuksina:

1. Avaa selain.
2. Siirry osoitteeseen ’http://localhost:3000/’.
3. Odota kunnes sivu on latautunut.
4. Lisää Nimi–kenttään teksti ’nightwatch’.
5. Lisää Sähköposti–kenttään teksti ’nightwatch’.
6. Tyhjennä Sähköposti–kenttä.
7. Lisää Sähköposti–kenttään teksti ’nightwatch@nightwatch.fi’.
8. Lisää Palute–tekstialueelle teksti ’I got nothing to say’
9. Paina Lähetä–painiketta.”

Testin tarkoituksena on siis täyttää palautelomakkeelle nimi, sähköposti ja palaute, ja lähettää kyseinen lomake. Seuraavaksi avaan tarkemmin mitä itse ohjelmistokoodi tasolla tapahtuu, sekä samalla sen, mitä ominaisuuksia Nightwatch.js tarjoaa

Kaavion 4 rivillä yksi määritellään uusi moduuli, jossa on määritelty (rivi 2) ja toteutettu (rivit 3-28) päästä-päähän –testit. Nightwatch.js:n avulla tehdyissä toteutuksissa alussa tulee määritellä testattavan sivun osoite. Kaaviossa 4 selaimen siirtyminen kehitysympäristön lokaalille palvelimelle tapahtuu rivillä 3.

Nightwatch.js tarjoaa testien ajamiseen selaimelle apu-funktioita, jotta testien suorittaminen olisi käytännössä mahdollista. Apu-funktioita ovat rivillä 5 nähtävä `waitForElementVisible()` ja rivillä 6 `pause()`. Funktio `waitForElementVisible()` tarjoaa testaukseen työkalun, jolla pystytään odottamaan selaimen latausaikoja. Funktio käytännössä haravoi selaimeen tulostuvaa DOM-puuta niin kauan kuin se löytää halutun elementin, joka on kaavion 4 tapauksessa `<body>`-elementti. Funktiolle pystytään välittämään parametrina myös aika, kuinka kauan elementtiä odotetaan. Valinnaisina parametreina funktiolle voidaan antaa totuusarvo, keskeytetäänkö etsintä vai ei jos elementti ei muutu näkyväksi tai sitä ei löydy, sekä teksti-muuttuja, jolla voidaan antaa viesti mikä tulostetaan toiminnon epäonnistuessa. Epäonnistuminen tapahtuu, jos kyseistä elementtiä ei löydy annetussa ajassa. Kaavion 4 tapauksessa valinnaisia parametreja ei ole käytetty ja ajaksi on määritelty yksi sekunti (1000 millisekuntia). Funktio `pause()` pysäyttää testin suorituksen selaimessa halutuksi ajaksi, kaavion 4 testeissä pysäytykset (riveillä 6, 12 ja 27) on lisätty ainoastaan selaimessa tapahtuvaa havainnointia varten, jotta kehittäjä pystyy tarkistamaan testien oikeellisuuden. (Rusu, Nightwatch.js)

Nightwatch.js:n funktion `setValue()` (Kaavio 4, rivit 7, 9, 11 ja 19) avulla pystytään asettamaan HTML-syötekenttiin syötettä. Funktion avulla pystytään myös käyttämään esimerkiksi näppäin-oikoteitä, jos kyseiseen ohjelmaan on sellaisia toteutettu. `SetValue()` ottaa ensimmäisenä parametrinaan vastaan HTML-valitsimen, joka kaavion 4 kaikissa kohdissa on luokan-nimi ja toisena parametrina itse syötteen. Esimerkiksi kaavion 4 rivillä 7, testi-toteutus etsii HTML-

elementin, jolta löytyy luokannimi ”name--input" ja syöttää kyseiseen syötekenttään tekstin ”nightwatch”. Nightwatch.js tarjoaa myös funktion `clearValue()` (Kaavion 4 rivi 11), joka toimii käänteisesti `setValue()`:n verrattuna, se tyhjentää syötekentän tai tekstialueen. (Nightwatch.js)

Funktioiden lisäksi Nightwatch.js tarjoaa tarkistuksia (asserts) joiden avulla pystytään esimerkiksi selvittämään, onko jollekin HTML-elementille lisätty attribuutti (attributeContains, Kaavion 4 rivi 14) tai onko elementillä jokin tietty tyylisääntö (cssClassPresent, Kaavion 4 rivi 26). Nightwatch.js:n on myös lisätty BDD-mallin mukaisesti Expect-laajennos, jonka avulla tarkistuksia pystytään tekemään BDD-mallin odottamalla tavalla. Esimerkiksi Kaavion 4 rivillä 26 kuvattu `cssClassPresent()` tarkistus, jonka avulla tarkistetaan onko palautteen modaali näkyvissä vai ei. Tämä voitaisiin kääntää BDD-mallin mukaisesti myös Nightwatch.js:llä seuraavaan muotoon: `browser.expect.element('.class').to.be.visible`. BDD-mallin kaltainen nimeäminen tekee testauksen toteutuksesta luettavampaa ja se sitoutuu paremmin myös käyttäjätarinaan, jos käyttäjätarinan ajatellaan esimerkiksi tässä tapauksessa olevan ”Käyttäjälle tulee näkyviin viesti lähetyksen onnistumisesta”-määrittely. BDD:n kaltaista nimeämistä käytetään Kaavion 4 rivillä 10. Rivillä 9 testi syöttää virheellisen sähköpostiosoitteen (muotoa ’nightwatch’, vaikka sähköpostin pitäisi sisältää tekstijonon muotoa ’xxx@xxx.xx’, eli sisältää ’@’- ja ’.’-merkit), jonka jälkeen voidaan olettaa ohjelmiston näyttävän virheilmoituksen. Näin ollen sähköpostiosoite on tässä tapauksessa virheellinen. Päästä-päähän –testien tulisi myös testata sitä, toimiiko ohjelmiston logiikka oikein, eli käytännössä tämä tapahtuu syöttämällä virheellinen arvo, josta pitäisi seurata virheviestin näyttäminen. (Rusu, Nightwatch.js)

3.3.3 Nightcloud – testaus pilvestä

Nightcloud on Nightwatch.js:n kehittäjiltä tuleva pilvipalvelu, jonka avulla pystytään toteuttamaan päästä-päähän –testausta Nightwatch.js:n avulla. Nightcloud.io lupaa tarjota käyttäjilleen mahdollisuuden toteuttaa etätestejä ilman konfiguraatiota, ja mahdollisuuden nauhoittaa testien tulokset (Rusu, Nightcloud). Lisäksi Nightcloud kertoo pystyvänsä itse analysoimaan ja

tunnistamaan yleisimpiä ongelmatilanteita ohjelmistokoodissa oletettavasti tekoälyn avulla (Rusu, Nightcloud). Nightcloud on kuitenkin tällä hetkellä (30.3.2018) vielä salaisessa beta -testausvaiheessa, jota pääsevät kokeilemaan ja testaamaan ainoastaan kutsutut tahot. Nightcloud on sisällytetty ja mainittu tässä osiossa sen takia, että pilvipohjainen testaus saattaa olla tulevaisuudessa erittäin merkittävässä asemassa verkkopohjaisten sovellusten päästä-päähän –testauksessa.

3.4 Yksikkötestaus

3.4.1 Jest

Jest on Facebookin ja Reactin kehittäjien toteuttama testaustyökalu, jonka avulla pystytään toteuttamaan testejä ilman asetuksien konfiguraatiota. Jest-teknologian valinnan syinä tähän on, että sen automatisoitu integraatio create-react-app-työkalun kanssa sekä se, että Jest on yksi suosituimmista työkaluista React-pohjaisiin verkkosovelluksiin. (http://facebook.github.io/jest/)

Jest tarjoaa jQuery (pitkäaikainen Javascript kirjasto verkkopohjaisten käyttöliittymäelementtien manipulointiin) tyylisen tavan päästä käsiksi Reactin toteuttamaan DOM-puuhun. Jestin rajapinta on toteutettu TDD-tyylisellä syntaksilla, jolloin testien lähdekoodin lukeminen on sujuvampaa ja se on hyvin samankaltainen kuin esimerkiksi luvun 3.4.2 Nightwatch.js kehyksen esimerkeissä. Esimerkiksi `expect(component.state{'numberOne'}).toBeGreaterThan(0)`–funktiosta on suoraan syntaksista luettavissa, että halutun komponentin tila nimeltään ”numero yksi” odotetaan olevan suurempi kuin arvo 0. (Facebook Inc. Jest)

3.4.2 Enzyme

Enzyme on Airbnb-yhtiön kehittämä työkalu Jest-ohjelmistokehyksen päälle. Enzyme mahdollistaa kehittäjälle helppokäyttöisemmän tavan muokata ja käsitellä React-pohjaisten sovellusten tuottamaa virtuaalista DOM-puuta, tarjoamalla niin sanotun *shallow rendering* ominaisuuden. Ominaisuuden avulla, testauksessa pystytään testaamaan ohjelmistoon toteutettuja komponentteja kokonaisuuksina, vaikka ne sisältäisivätkin pienempiä osia (http://airbnb.io/enzyme/). Enzyme siis tarjoaa työkalun komponenttien eristämiseen ja mahdollistaa testien toteuttamista komponenttipohjaisesti. Näin

Reactin tarjoama komponenttipohjainen ajattelumaailmaa pystytään soveltamaan myös testauksessa (http://airbnb.io/projects/enzyme/). Enzyme on saatavilla myös muihin testauskehyksiin (kuten chai, mocha, karma etc.), jotka helpottavat nykypäivän modernissa verkkopohjaisessa ohjelmistokehityksessä käytössä olevan virtuaalisen DOM-puun testausta. Kuten luvussa 2.1 kerroin, tavallinen DOM-puu koostuu useista eri solmuista, eli elementeistä.

Virtuaalinen DOM-puu on virtuaalinen kopio sivustolle tulostetusta DOM-puusta. Virtuaalisen DOM-puun ideana on, että DOM-puusta pidetään kopiota tallessa virtuaalisena muistissa, jolloin muistissa sijaitsevan virtuaalisen puun elementeille pystytään tallentamaan elementin tila. Elementin tilan tallentamisella pyritään siihen, että käyttöliittymätoteutuksissa pystytään tarkkailla, onko elementin tila muuttunut (esimerkiksi onko elementin sisältämä data muuttunut, tai onko käyttäjä vuorovaikuttanut kyseisen elementin kanssa) (Esch, Virtual-dom). Elementin tilaa tarkkailemalla pystytään siis havaitsemaan muutokset, joita käyttäjä mahdollisesti tekee vuorovaikuttaessa elementin kanssa. Elementin tilan muuttuessa, virtuaalisen DOM-puun avulla pystytään päivittämään juuri halutut, muuttuneet DOM-puun elementit dynaamisesti, ja lisäksi pystytään pitämään kirjaa näiden elementtien tilasta. Virtuaalisen DOM-puun puute tekee DOM-elementtien tilan hallinnasta hankalaa, sillä tavallinen DOM-puu on aina staattinen tekstirepresentaatio sivuston rakenteesta. (Esch, Virtual-dom). Virtuaalisen DOM-puun esitys on tärkeä osa komponenttipohjaista ajattelua, johon Enzyme juuri pyrkii tarjoamaan apua. Enzyme tarjoaa testikehykselle mahdollisuuden päästä käsiksi jo virtuaaliseen DOM-puuhun, jolloin pystytään testaamaan ja käsittelemään paremmin yksittäisiä komponenttien toteutuksia, eikä pelkästään staattista esitystä toteutetuista komponenteista. (http://airbnb.io/enzyme/)

Tämän tutkielman yhteydessä toteutetun sovelluksen kannalta Enzyme ei olisi pakollinen, sillä kyseinen palautelomake ei sisällä monimutkaista komponenttilogiikkaa, jolloin edellä mainitun kaltaisille pinnallisille kopioille ei ole tässä yhteydessä tarvetta. Enzyme on kuitenkin otettu tässä tutkielmassa käyttöön, jotta laajemmat ja todelliset projektitoteutukset voisivat hyötyä myös tässä tutkielmassa esitetyistä esimerkeistä, esimerkiksi snapshot-testauksesta puhuttaessa.

3.4.3 Yksikkötestien toteutus

Yksikkötestit on toteutettu siis Jest-testauskehyksen avulla, joka käyttää Enzyme-työkalua. Tutkielman yhteydessä toteutettujen testien tarkoituksena ei ole syventyä esimerkiksi Enzymen tarjoamaan *shadow rendering*-ominaisuuden avulla tehtävään virtuaalisen DOM-puun käsittelyyn ja kulkemiseen, vaan näyttää yksinkertaisen esimerkin avulla, kuinka sovelluksille ylipäätään on mahdollista toteuttaa yksikkötestejä. Sovellukseen toteutetuilla yksikkötesteillä on haluttu näyttää, kuinka yksittäistä ominaisuutta voidaan testata. Toteutetussa palautelomake-sovelluksessa on valittu testattavaksi lomakkeen validointi ja tässä tapauksessa sähköpostiosoitteen tarkistaminen, jossa täyttyvät toiminnallisuus, syötteen validointi, sekä ohjelmiston tilan päivittyminen sen perusteella millainen syöte kyseiselle funktiolle eli ohjelman ”pienimmälle osalle” on annettu.

```
1  import React from 'react';
2  import {shallow} from 'enzyme'
3
4  import App from './Form';
5
6  test('App renders without crashing', () => {
7    const wrapper = shallow (<App/>)
8  });
9
10 test('testing email validation, incorrect string, state: false', () => {
11   const app = shallow (<App/>)
12   app.setState({emailField : 'testing'})
13   app.instance().formValidate()
14   expect(app.state('emailValid')).toBeFalsy()
15 });
16
17 test('testing email validation, email address, state: true', () => {
18   const app = shallow (<App/>)
19   app.setState({emailField : 'a@a.fi'})
20   app.instance().formValidate()
21   expect(app.state('emailValid')).toBeTruthy()
22 });
```

Kaavio 5. Yksikkötestien toteutuksen lähdekoodi esimerkki.

Kaaviossa 5 on esitetty kolme yksikkötestiä luvussa 3.1 kuvatulle palautelomakkeelle. Kaavion rivillä 1 yksikkötestiin tuodaan mukaan React, jotta kyseistä kehystä pystytään testata, rivillä 2 tuodaan mukaan Enzymen tarjoama *shallow rendering*, sekä rivillä 4 tuodaan itse sovellus. Riveillä 6, 10 ja 17 esitellään yksikkötestit. Ensimmäisen yksikkötestin tarkoituksena on ainoastaan tarkistaa, että itse sovellus muodostuu sivulle oikein. Tällä tarkistuksella pyritään siihen, että mahdollisista testien epäonnistumisista voidaan poissulkea se, että koko sovellus ei ole lähtenyt käyntiin.

Toinen toteutettu yksikkötesti testaa itse lomakkeen validointia. Rivillä 10, määritellään Jest-rajapinnan avulla `test()`-funktio ja nimetään se BDD:n vaatimalla tavalla kuvaavasti sisältäen syötteen kuvauksen ja odotetun lopputuloksen testille. Tämän yksikkötestin tapauksessa halutaan testata sähköpostin validointia. Testissä halutaan antaa väärän muotoinen syöte, joka on tekstityyppiä. Testissä oletetaan, että sähköpostin validoinnin tulisi olla epätosi. Rivillä 11 valitaan koko sovellus käsiteltäväksi. Rivillä 12, asetetaan sovelluksen tilaan sähköpostiosoite-kentän arvoksi

sähköpostikentän tilaa vastaava objekti. Objekti sisältää attribuutin nimeltä `emailField` ja se sisältää teksti-tyyppisen muuttujan, joka sisältää tekstin ”testing”. Rivillä 13 kutsutaan manuaalisesti validointi-funktiota `formValidate()`, jonka tarkoituksena on tarkistaa lomakkeen kenttien oikeellisuus ja laukaista mahdolliset lomakkeen tai sen kenttien tilan päivitykset. Funktiokutsu manuaalisesti Enzymen avulla tapahtuu niin, että kutsutaan yksikkötestin alussa haettua juuri-elementtiä ja Enzymen sille tarjoamaa funktiota: `instance()`. Kun lomakkeen syötteiden validointi ja tilan päivitys on käynnistetty, voidaan rivillä 14 ensin hakea sähköpostinkentän validoinnin tila `app.state('emailValid')` ja olettaa virheellisen syötteen annon ja validoinnin jälkeen arvon (Runeson, 2006) olevan epätosi seuraavasti: `expect().toBeFalsy()`.

Rivin 17 yksikkötesti on käytännössä muuten sama, mutta käänteisenä. Sähköpostikentän tilaksi asetetaan tekstityyppisenä muuttujana oikeellinen sähköpostiosoite, joka sisältää ’@’-merkin. Tällöin rivillä 21 voidaan olettaa sähköpostikentän tilan olevan oikeellinen ja läpäisseen validoinnin: `expect().toBeTruthy()`.

```
PASS src/Form.test.js
  ✓ App renders without crashing (32ms)
  ✓ testing email validation, incorrect string, state: false (12ms)
  ✓ testing email validation, email address, state: true (4ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        1.715s
Ran all test suites related to changed files.
```

Kaavio 6. Jestillä toteutettujen yksikkötestien tuloste komentorivillä.

Kaavio 6 näyttää miten yksikkötestien tulokset näkyvät käyttäjälle ja millä tavalla hyväksytysti läpi menneet testit näkyvät. Kaavion 6 kuvasta voidaan myös huomata, että Jestin tarjoama snapshot –ominaisuus ei ollut käytössä. Snapshot-ominaisuus Jestissä tarkoittaa sitä, että Jest ottaa niin kutsutut kuvankaappaukset oikeellisesti toimivasta komponentista, jonka jälkeen, aina snapshot-testit ajettaessa, sen hetkistä kuvankaappausta verrataan tähän haluttuun tulokseen ja pyritään

varmistumaan kaiken toimivan yhä oikeellisesti. Jestin tapauksessa kyseessä ei ole kuitenkaan oikeista kuvankaappauksista, vaan kyseessä on representaatio komponentin virtuaalisesta DOM-puusta. Tämä representaatio pidetään Jestin toimesta tallessa, ja verrataan sitten tulevaisuudessa, aina Snapshot-testit ajettaessa, tallessa olevaa sen hetkiseen representaatioon DOM-puusta ja näin pyritään säilyttämään komponenttitason muuttumattomuus. (Facebook Inc. Jest)

3.5 Yhteenveto

Kuten toteutetuista yksikkötesteistä ja päästä-päähän –testeistä havaitaan, teknologiat ovat toteutukseltaan ja syntaksiltaan toistensa kanssa hyvin samankaltaisia. Nykypäivänä, puhuttaessa yhä enemmän BDD:n kaltaisista suunnittelumalleista ja siirryttäessä niiden ehdottamiin hyviin käytäntöihin (esimerkiksi GivenWhenThen) on tämä yhtenäistyvä kehityssuunta täysin ymmärrettävää.

Tässä luvussa esitetyissä esimerkeissä huomataan suurimpana erona yksikkötestien ja päästä-päähän –testaukselle, että yksikkötesteissä pystytään suoraan muokkaamaan komponentin tilaa ja asettamaan sille käytännössä mitä tahansa arvoja (`app.state()`). Päästä-päähän –testauksessa kaikkien syötteiden tulee kulkea aina selaimen ja sovelluksen käyttöliittymän kautta kutsumalla selainta ohjelmallisesti `browser.setValue()`. Näin ollen komponentin tilan ja sen ominaisuuksien muokkaaminen on päästä-päähän –testauksessa lähempänä luonnollista käyttötilannetta ja ohjelmallinen manipulointi ei ole suoranaisesti mahdollista. Tämänkaltainen tilanne voi olla merkittävä, esimerkiksi syöte-kentässä, joka ottaa vastaansa ainoastaan numero-tyyppistä sisältöä. Tällaiseen tekstikenttään ei pysty syöttämään muuta kuin numeroarvoja, joten tekstin syöttäminen itse kentän tilaan ei pitäisi olla mahdollista. Vaikka kyseisen toiminnon ei pitäisi olla mahdollista, niin sovellus on silti hyvä testata ja myöskin toteuttaa niin, että numeroarvot ovat ainoat mitä tälle kentälle voidaan syöttää.

Yksikkötestejä voidaan ajatella tarvittavan etenkin silloin kun sovelluksessa on oikeasti paljon toiminnallisuutta ja funktioita, joiden toiminta on tärkeää koko ohjelmiston toimintalogiikan kannalta. Päästä-päähän –testejä taas on hyvä olla myös silloin kun itse toimintalogiikka ei ole niin monimutkainen, vaan tärkeämpänä kriteerinä voidaan pitää käyttöliittymäelementtien määrää, etenkin jos ne sisältävät toiminnallisuuksia tai niiden toiminnot ovat riippuvaisia toisistaan. Yksinkertaistettu esimerkki tästä on kaavion 4 rivillä 10 (luku 3.3.2) esiintyvä toteutus, kuinka testata tuleeko virheviesti näkyviin, jos sähköposti on virheellinen. Tässä tapauksessa siis yksittäisen tekstikentän näkyvyys riippuu käyttäjän oikean muotoisesta syötteestä ja siitä validoidaanko se oikeellisesti toteutetussa validointifunktiossa. Luvun 3.3.3 yksikkötestausesimerkissä huomataan, että kyseisestä vaihtoehdosta pystytään testaamaan ainoastaan validointifunktion toimivuus ja se onko validointi mennyt läpi. Yksikkötestien pääsyy rajoittuu siis tähän, ohjelmalliseen toteutukseen, jolloin yksikkötesteillä ei pystytä saamaan täyttä varmuutta siitä, tuleeko kyseinen virheviesti oikeellisesti näkyviin vai ei. Osion lopuksi on vielä syytä huomauttaa, että tyylien tarkastaminen on myös Jestin avulla yksikkötesteissä käytännössä mahdollista. Luvussa 2.3 esitetyn yksikkötestauksen määrittelyn mukaisesti (pääsy ainoastaan controller ja model tasolle, ei view), yksikkötestien tulisi testata ainoastaan ohjelmiston toiminnallisia osia, joihin tyylit eivät tässä tapauksessa kuulu. Tähän taas vastaa yksikkötesteinä toteutetut snapshot-testit, jolloin pystytään varmistumaan komponenttien samankaltaisuudesta, myös view-tason representaationa.

4. ASIANTUNTIJAHAASTATTELUT

Tämän tutkielman oleellisena osana on asiantuntijahaastattelu, sillä tutkielman tarkoituksena on tuottaa tutkielman toimeksiantajan, Solitan kannalta tietoa siitä, miten yrityksessä toteutetaan testausta tällä hetkellä. Tutkielma pyrkii myös selvittämään, kuinka testausta voitaisiin tulevaisuudessa parantaa yrityksen toteuttamissa projekteissa.

Asiantuntijahaastatteluiden toteuttamismuodoksi valikoitui ryhmähaastattelu (*focus group*) sillä haastateltaviksi valittiin Solitalla vanhemmassa asemassa (senior) toimivia ohjelmistokehittäjiä. Suurin syy tutkimusotteen valintaan on se, että haastateltavilla on oletettavasti paljon enemmän tietoa aihepiiriin liittyvistä asioista. Näin ollen he saattavat kokemuksensa ja osaamisensa pohjalta keskustella spontaanisti, sekä tuoda esiin huomioita ja kysyä kysymyksiä aiheeseen liittyen, joita yksittäishaastatteluissa ei välttämättä osattaisi kysyä tai ei ymmärrettäisi näiden olevan oleellisia asioita itse tutkimuksen kannalta. Haastateltavilla voidaan olettaa olevan testauksesta ja ohjelmistokehityksestä kokemusta yhteensä useita kymmeniä vuosia, kun taas haastattelun toteuttajan kokemus testausten toteutuksesta on minimaalinen ja tieto aiheeseen liittyen on kerätty pääsääntöisesti viimeisen vuoden aikana. Tämän tutkielman olosuhteiden ja edellä kuvatun kaltaisessa tilanteessa ryhmähaastattelu on hyvä tapa saada laajasti selville osallistujien mielipiteitä ja selvittää lisää tietoa aihepiiriin liittyvistä asioista (Kitzinger, 1995). Ryhmähaastattelun tavoitteena onkin myös rohkaista haastatteluun osallistujia keskustelemaan avoimesti mielipiteistään ja suhtautumisestaan testaukseen ja yleisesti testausteknologioihin (Kitzinger, 1995). Myös ryhmähaastatteluun kuluva aika, verrattuna useisiin yksilöhaastatteluihin käytettävään aikaan vaikutti tutkimusotteen valintaan.

4.1 Tutkimuksen toteutus

Tutkimus toteutettiin ryhmähaastatteluna Solitan sisällä ja viidestä kutsutusta osallistujista tähän tilaisuuteen osallistui 3 vanhemmassa asemassa toimivaa ohjelmistokehittäjää (Osallistuja 1, 2 ja 3).

Osallistujille kerrottiin ennen haastatteluun osallistumista tutkimuksen tarkoituksesta ja mainittiin tutkimuksen eettisistä käytännöistä ja siitä, että osallistujia käsitellään anonyymisti. Kaikki osallistujat olivat sukupuoleltaan oletettavasti miespuolisia ja heidän iät ovat 30– ja 45–ikävuosien välillä. Vanhemmassa asemassa toimivat ohjelmistokehittäjät ovat Solitalla usein olleet valitsemassa projektissa käytössä olevia teknologioita ja mahdollisesti tehneet uransa aikana näin useissa eri projekteissa. Ennen haastattelua, haastattelukutsussa, haastatteluun osallistujia pyydettiin valmistautua kertomaan hieman tämän hetkisen projektin testauksesta (mitä testataan, missä vaiheessa testit tehdään, kuinka paljon testejä toteutetaan ja miksi juuri kyseinen teknologia on valittu projektiin) ja mahdollisuuksien mukaan näyttämään joitakin heidän valitsemiaan kooditason esimerkkejä projektiinsa toteutetusta testauksesta. Toteutuksen esittely valittiin ryhmähaastatteluun tehtäväksi, jotta haastattelun osallistujat pääsisivät näkemään toistensa toteutuksia ja vertaamaan omia toteutuksiaan toisiin ja tätä kautta herättämään spontaania keskustelua aiheesta. Haastattelulla pyrittiin selvittämään, kuinka Solitan toteuttamissa ohjelmistoprojekteissa toteutetaan testausta, mitä teknologioita on valittu käyttöön, sekä ennen kaikkea vastaamaan siihen miten testauksen oletetaan tulevaisuudessa muuttuvan ja kuinka Solitan tulisi mahdollisesti muuttaa nykyisiä käyttöliittymätestauksen tapojaan, pyrkiessään vastaamaan tähän muutokseen.

Haastattelun kulkuun oli alun perin suunniteltu alle puolentunnin esittely vaihe, jonka jälkeen alettiin esittää tarkentavia kysymyksiä aiheesta ja tarjota aiheita keskusteluun. Haastattelutilanteessa esittely osion kuitenkin annettiin venyä 45 minuutin mittaiseksi, sillä osallistujat keskustelivat ahkerasti toistensa toteutuksista ja tarjosivat huomioita ja kysymyksiä kustakin esiteltävästä kokonaisuudesta. Haastattelu kesti kokonaisuudessaan hieman yli 60 minuuttia. Huomiot ja kysymykset esitysten aikana kattoivat monia asioita alun perin suunnitelluista haastattelukysymyksistä (Liite 2). Lisäksi haastattelija päätti ensimmäisen osion venymisen vuoksi herättää keskustelua kysymyksillä jo esittelyvaiheessa, jotta saataisiin esiteltävistä asioista tietoa niin, että niihin ei tarvitsisi palata myöhemmin uudestaan.

4.2 Haastatteluiden tulokset

Haastatteluun osallistujista Osallistuja 1:n ja Osallistuja 3:n nykyisissä projekteissa on toteutettu automatisoitua testausta ja he ovat itse suunnitelleet myös siihen toteutetut testausmenetelmät ja kuinka testausta tulisi tehdä. Kummankin osallistujan projekteissa oli toteutettu sekä yksikkötestausta että päästä-päähän –testausta.

Osallistujan 1 projektissa oli käytössä samat teknologiat kuin edellisessä luvussa esitellyssä testauksen toteutuksessa, eli päästä-päähän –testaukseen käytetään Nightwatch.js-testauskehystä ja yksikkötestaukseen Jest-, sekä Enzyme-teknologioita. Osallistuja 1:n projektissa testaus painottuu yksikkötestien toteutukseen, joita tehtiin kaikille funktioille, jotka oli tuotu näkyviin. Testit on jaettu sovelluksen Utility-osien ja komponenttiosien testaukseen. Ohjelmiston *Utility*-osilla tarkoitetaan ohjelmiston osia, joita käytetään tarjoamaan käyttöliittymäkomponenteille erilaisia toiminnallisuuksia ja apufunktioita. Utility-funktiot liittyvät usein tiedonkäsittelyyn tai ovat funktioita joiden toiminnallisuutta tarvitaan useissa eri komponenteissa, jolloin toteutus on jaettu usean komponentin kesken. Osallistujan 1 projektissa komponenttien testaus on toteutettu pääsääntöisesti Snapshot-testauksena, joiden tarkoituksena on tallentaa näkymien tila jaksotettuna (*serialized*) JSON-esityksenä. Esitystä verrataan aina testit ajettaessa nykyiseen, eli nykyisten snapshot-testien tilaan. Kriittisille komponenteille, jotka ovat erikoistuneet datankäsittelyyn (välitetään dataa tai vastaanotetaan dataa), toteutetaan myös perinteisiä yksikkötestejä, joiden tarkoituksena on varmistaa komponentin tilanhallinnan oikeellisuus. Komponenttien testit on rakennettu komponenttien yhteyteen, joita ajetaan jatkuvasti kehityksen yhteydessä. Osallistuja 1:n projektissa päästä-päähän –testit ajetaan ainoastaan ns. Smoke-testeinä, jolla tarkoitetaan esimerkiksi tuotantoon tai CI:n testiympäristön viennin yhteydessä ajettavia hyväksymistestejä. Näillä testeillä pyritään varmistumaan siitä, että mikään ominaisuus ei rikkoudu uuden version toimituksen, eli ns. buildin yhteydessä, vaan kaikki vanhat toteutetut ominaisuudet toimivat ja ohjelmisto toimii kokonaisuudessaan oikein. Usein smoke-testauksen kaltainen hyväksymistestaus myös estää toimituksen tuotantoon, jos hyväksymistesti, eli tässä tapauksessa päästä-päähän –testi ei

mene läpi. Osallistujan 1 projektissa ei ole käytössä minkäänlaista TDD:n kaltaista prosessia, jossa testit toimisivat komponenttien toteuttamisen hyväksymisvaatimuksina. Haastattelusta kuitenkin käy ilmi, että projektin päästä-päähän –testit toteutetaan käyttötapausten pohjalta, jotka myös toimivat alun perin toteutettujen komponenttien suunnitteluperustana. Kysyttäessä TDD:n käytöstä projektissa, Osallistuja 1 kertoo osan kehittäjistä kokeilleen TDD:n kaltaista lähestymistapaa testien toteuttamiseen. Projektissa kuitenkin todettiin, että testien määrittelyn ei pitäisi olla etukäteen tarkasti määriteltyä ja tämän toimivan paremmin heidän projektinsa tarpeisiinsa.

Osallistuja 3:n projektissa aikaisemmin päästä-päähän –testaus on suoritettu Nightwatch.js:llä, mutta kyseisestä teknologiasta on sittemmin siirrytty Cypress-testauskehykseen. Yksikkötestaus projektissa on toteutettu Karma-, Mocha- ja Chai-testaustyökalujen avulla. Testauksessa Karma hoitaa testien ajamisen (*test runner*), Mocha taas on testauskehys, joka tarjoaa itse testausrajapinnan, ja Chai on kirjasto, joka mahdollistaa TDD:n GivenWhenThen-syntaksin kaltaisen tavan kirjoittaa testausta. Osallistuja 3:n projektissa pääpaino on ollut yksikkötesteissä ja yksikkötestauksen tarkoituksena on ollut testata kaikki toteutetut funktiot. Testien toteutuksen pohjana on pyritty käyttämään TDD-mallin mukaista ohjelmistokehitystä, eli testit toteutetaan ensin, jonka jälkeen aletaan tehdä vasta itse toteutusta. TDD:n pohjalla heidän projektissa on ollut niin kutsuttu liikennevalomalli, jossa ensimmäisenä vaiheena on toteuttaa käyttötapauksen pohjalta testit. Tällöin testi on toteutettu, mutta itse toteutusta ei ole tehty ja liikennevalo on silloin punaisella. Punainen-tila säilyy niin kauan kuin tehty testi ei mene läpi hyväksytysti. Kun testin läpäisevä toteutus on tehty, vaihtuu komponentin testin tila tällöin keltaiselle. Keltainen väri merkitsee sitä, että tällöin on tehty pienin mahdollinen toimiva ratkaisu, jolla testi on mahdollista läpäistä. Kun keltainen-tila on toteutunut, aletaan ohjelmistototeutuksen refaktorointivaihe, jolloin pyritään tekemään paras mahdollinen toteutus ja saavuttamaan vihreä-tila. Viimeinen vaihe sisältää myös usein koodikatselmointeja tai parikoodausta, jolloin myös toteutuksen vertaisarviointi tulee osaksi toteutetun ohjelmistokoodin laatua. Osallistuja 3 mainitsee, että projektissa käytössä olleesta TDD-

mallista ollaan alettu poiketa, johtuen uusien kehittäjien tulosta mukaan projektiin, sekä siitä, että kaikki mukana olijat eivät ole vakuuttuneita mallin tarpeellisuudesta.

Osallistuja 2 oli haastatteluun osallistujista ainoa, kenen projekteissa ei tällä hetkellä tehdä automatisoitua testausta käyttöliittymätoteutukselle. Osallistuja 2:n tämän hetkinen projekti eroaa myös muiden osallistujien projekteista siinä, että kyseessä ei ole dynaaminen verkkosovellus tai –sivusto vaan pikemminkin sisällöntuotannollinen verkkosivusto (CMS, *content management system*), joka sisältää pienempiä dynaamisia käyttöliittymäkomponentteja. Osallistuja 2:n projektiin on projektin edellisten, Solitan ulkopuolisten, toteuttajien toimesta toteutettu yksikkötestausta. Kyseiset testit eivät Osallistuja 2:n mielestä ole kuitenkaan vastanneet testauksella pyrittävään tarkoitukseen, vaan testit ovat lähinnä toteutettu projektin testauksen minimikriteerien täyttämiseksi. Nykyiseen projektiin on myös harkittu viimeisen vuoden aikana automatisoidun testauksen toteutuksesta ja tästä on myös tehty esimerkkitoteutus prototyypin tapaisesti ja näytetty minkälaista testausta uusilla testauskehyksillä pystyttäisiin toteuttamaan projektissa. Solitan ulkopuoliset tahot ovat kuitenkin suositelleet modernien teknologioiden sijasta käytettäväksi vanhempia ja heille tutumpia työkaluja testien toteuttamiseksi. Vanhentuneen teknologian ja käytäntöjen takia testejä ei ole sitten koskaan haluttu toteuttaa Solitan päässä. Osallistuja 2:n projektissa on kuitenkin käytössä aktiivisesti käytettävyystestaus ja vähintään kerran vuodessa tehdään laaja-alaisempi testaus, jossa selvitetään tilastojen avulla käytetyimpiä laitteita ja selainsovelluksia, joita vastaan toteutetaan käytettävyystestaus näillä laitteilla.

4.3 Haastattelun tuloksien tarkastelu ja analysointi

Haastattelun osallistujat 1 ja 3 näkivät, että heidän nykyisten projektien tiimit ovat tehneet hyvää työtä testauksen toteutuksessa ja he eivät oikeastaan muuttaisi mitään nykyisissä käytännöissään. Heidän toteutuksensa olivat rakenteeltaan samankaltaisia ja noudattivat sitä myös testauksen päämääriltä, vaikka käytössä olivat eri teknologiat. Molemmissa projekteissa pääsääntöisesti

käytettiin yksikkötestausta koodin laadun varmistamiseen ja sitä on tehty molemmissa projekteissa laajassa mittakaavassa. Kysyttäessä osallistujilta heidän projektiensa testauksen kattavuudesta, molemmat kertovat olevansa tyytyväisiä tämän hetkiseen tilaan. Molempien osallistujien projekteissa on käytössä myös testien seurantatyökalu, jonka avulla pystytään seuraamaan testattujen komponenttien määrää projektissa, sekä sitä kuinka moni testeistä menee tällä hetkellä läpi tilastollisesti ja visuallisesti. Osallistujan 1 ja Osallistujan 3 projektien päästä-päähän –testauksesta voidaan löytää myös samankaltaisuuksia. Vaikka Osallistujan 3 projektissa käytetäänkin päästä-päähän –testausta TDD:n kaltaiseen hyväksymistestaukseen, toimii se projektissa myös komponenttien laadun varmistajana. Kaikki osallistujat mainitsivat haastattelun yhteydessä, että eivät halua käyttää tai tehdä testejä Seleniumia vastaan, sillä he pitävät teknologiaa vanhentuneena. Kerrottaessa, että heidän käyttämät teknologiat pohjautuvat myös Selenium-teknologian pohjalta toteutetulle W3C:n WebDriver API:lle, vaikuttivat kaikki osallistujat hyvin yllättyneiltä. Tämän voidaan ajatella johtuvan siitä, että kaikki osallistujat viittasivat Seleniumiin vanhentuneena teknologiana, jota on käytetty jo vuosia sitten, eikä WebDriverin kautta uudistettuun ja sen kanssa yhdistettyyn Seleniumin uuteen spesifikaatioon.

Yhteisenä kehityskohteena molempien osallistujien projektissa nousi esiin data-mallien hallinta ja niiden testaus. Data-mallien testauksesta molemmilla nousi esiin tyypityksen mahdollinen käyttöönotto, joka osallistujien mukaan voisi auttaa data-mallien muuttujien oikeellisuuden varmistamisessa. Tämänlaisia vaihtoehtoja voisivat esimerkiksi olla TypeScriptin kaltaiset laajennokset JavaScript-kieleen, jotka tarjoavat tyypityksen tuomat edut JavaScriptiin, joka itsessään on heikosti tyypitetty ohjelmointikieli (Microsoft, TypeScript). Tyypitetyt kielet ja niiden tarjoamat tyyppiliteraalit, saattaisivat siis ratkaista useampia heidän kohtaamia ongelmiaan. Osallistujan 1 projektissa käytössä on HL7-datamalli, jota käytetään Suomessa monissa terveydenhuollon järjestelmissä datan yhdenmukaiseen toteuttamiseen (esimerkiksi: https://www.hl7.org/fhir/, http://www.kanta.fi/fi/web/ammattilaisille/hl7). Osallistuja 1:n projektissa syynä miksi tyyppejä ei ole integroitu mukaan projektiin on se, että kokemukset tyyppitestaukseen tarkoitettua työkalusta

(https://flow.org/). Käyttöönottovaiheessa testien tekemiseen koettiin menevän aivan liian paljon aikaa ja testien toteuttaminen ja muuttaminen tukemaan tyypitystä koettiin liian työlääksi. Tämä tapahtui etenkin intensiivisessä kehitysvaiheessa, jossa käytetyn HL7-standardin mukaisen datamallin sisältö saattaa vielä muuttua ja lopullinen datamallin rakenne hakee vielä muotoansa. Datamallin mukaisen laadun varmistamiseksi projektissa on toteutettu laajat ja tarkat syötteiden validoinnit. Validoinnit on toteutettu niin, että virheellisten tyyppien syöttäminen ja niiden läpi pääseminen tarkastuksista on validointien avulla estetty. Riittävän validoinnin ansiosta projektissa ei ole nähty tarpeelliseksi testata muuttujien tyypitystä ja datan oikeellisuuden varmistamista tyyppitestaus-kirjastolla sen tarkemmin, sillä he saavat tiedon omien toteutuksiensa datasta jo muiden testien kautta. Osallistuja 1 kuitenkin mainitsee, että he aikovat tästä kaikesta huolimatta vielä parantaa projektinsa data-kerrosta ja samalla myös varmentaa projektin testausta GraphQL–kielen avulla.

GraphQL on kyselykieli, jonka avulla pystytään käyttämään ohjelmointirajapintaa. (Facebook Inc. 2015). GraphQL tarjoa REST-rajapinnan (Representational state transfer) kaltaisen tavan hakea, esittää ja käyttää välitettävää dataa. GraphQL:n yksi suurimmista eroista esimerkiksi REST-rajapintaan on tyypitetyt muuttujat.

```
Type Tutkielma {
     Otsikko: String,
     Sisältö: String,
       Tekijä: Kirjoittaja
   }
Type Kirjoittaja {
Opiskelijanumero: ID,
        Nimi: String,
       Syntymäpäivä: Date
}
```

Kaavio 7. Esimerkki tutkielman rakenteesta esitettynä GraphQL:n avulla.

Kaaviossa 7 on kuvattuna Tutkielma-tietomalli. Tutkielma pitää sisällään otsikon ja sisällön jotka ovat aina tekstityyppisiä muuttujia. Tutkielma pitää sisällään myös Kirjoittajan. Kirjoittajalla on aina yksilöivänä tunnuksena opiskelijanumero, nimi teksityypin muuttujana, sekä syntymäpäivä, joka on päivämäärätyyppiä. Kuten kaaviosta 7 käy ilmi, että GraphQL:n avulla data-malliin pystytään määrittämään mitä tyyppiä jokin muuttuja on ja samalla pystytään mallintamaan tyypit ja varmistumaan tiedon oikeellisuudesta siis jo kyselyvaiheessa.

Osallistujilta kysyttäessä, mitä he pitävät tärkeänä testauksesta Solitan ja sen toteuttamien projektien kannalta oli se, että testausta toteutetaan ylipäätänsä. Oli testauksen toteutuksena sitten pelkästään testi siitä, että sovellus käynnistyy oikein, tulisi se tehdä. Jo yksi testi saattaisi ennalta ehkäistä monia mahdollisia virhetilanteita projekteissa. Kysyttäessä syitä testauksen puuttumiselle haastatteluun osallistujat eivät osanneet suoraan vastata, mikä testaamattomuutta aiheuttaa. Osallistujat arvelivat, kuten Osallistuja 2:n tapauksessa, syiden usein liittyvän resursseihin. Resurssien puuttuminen voidaan ajatella jakautuvan kolmeen eri kategoriaan, joko projektiin toteutettaville ominaisuuksille ei ole budjetoitu tarpeeksi työtunteja testauksen toteuttamiselle, projektiin toteutettaville työtunneille ei ole budjetoitu tarpeeksi toteuttajia tai projektiin toteutettaville ominaisuuksille ei ole varattu tarpeeksi aikaa, jotta testit ehdittäisiin toteuttaa ennen julkaisua. Toinen syy testauksen puuttumiselle on se, että kehittäjät eivät koe testauksesta saatavien hyötyjen kohtaavan toteutuksen määrän kanssa, esimerkiksi ajankäytön näkökulmasta. Haastatteluista käy ilmi, että aina testauksen toteutusten hyödyt eivät olet verrattavissa mahdollisesti

siihen käytettyyn aikaan, jolloin testaus jää helposti usein pois. Näin voidaan ajatella käyvän myös projekteissa joissa testausta ei ole toteutettu lainkaan. Ajateltaessa esimerkiksi projektia, joka sisältää ainoastaan muutamia dynaamisia käyttöliittymäkomponentteja alkuperäisessä toteutusvaiheessa. Näin pienessä projektissa kehittäjä saattaa ajatella yksittäisten komponenttien testaamisen olevan yksinkertaista ja nopeaa myös manuaalisesti, verrattuna automatisoidun testauksen opettelemiseen ja pystyttämiseen käytettävään aikaan. Tällaisen ajattelutavan voidaan ajatella pätevän etenkin pienen mittakaavan projekteissa, joissa alkuperäiset määrittelyt eivät anna ymmärtää enempää projektin tulevaisuudesta. Liikeyritysten tavoitteena on kuitenkin pyrkiä talouskasvuun, joten pieniä projekteja pyritään usein kasvattamaan, jolloin pienet projektitoteutukset laajenevat ja kasvavat suuremmiksi toteutuksiksi. Tällöin myös projektien testauksen tulisi kehittyä projektin vaativuuden mukaan. Jos projekti kuitenkin sisältää jo valmista, testaamatonta toteutusta, on tähän valmiiseen toteutukseen huomattavasti vaikeampaa toteuttaa testausta jälkikäteen. Tämä voi esimerkiksi johtua kehittäjän, tai jopa koko toimittajan vaihtumisesta tai mahdollisesti uuden toteutuksen tiukasta aikataulusta, jolloin kehitysprojektiin lisätään jo alussa niin sanottua teknistä velkaa. Teknisellä velalla tarkoitetaan ohjelmistotuotannossa ratkaisuja, joilla useimmiten pyritään saavuttamaan taloudellista tai aikataulullista etua projektissa, mutta nämä edut tehdään laadun ja hyvien käytäntöjen kustannuksella. Tämä tarkoittaa käytännössä sitä, että tehdään teknisesti huonompia ratkaisuja tai jätetään joitain asioita kuten testausta tekemättä, jotta saavutetaan esimerkiksi projektin alkuperäinen aikataulu tai saavutetaan annetut työmääräarviot. Teknisen velan ottoa tulisi pyrkiä välttämään suurissa määrin, etenkin jos velan takaisinmaksulle ei tule olemaan todellisia perusteita. Usein tällaisessa tilanteessa velkaa jää pysyväksi osaksi ohjelmiston toteutusta.

5. YHTEENVETO JA POHDINTA

Tämän tutkielman kirjallisuuskatsauksessa perehdyttiin automatisoituun testaukseen ja siihen, mitä se on, kuinka sitä tehdään ja mitä teknologioita siihen yleisesti käytetään. Tutkielman aihe myös rajattiin koskemaan yksikkötestausta ja päästä-päähän –testausmenetelmiin, jotka ovat useimmiten käytössä Solitan käyttöliittymäteknologioiden testauksessa. Kirjallisuuskatsaus esitteli lukijalle Selenium–WebDriver –teknologian ja sen, kuinka teknologiat ovat nyt siirtyneet W3C:n alaiseksi spesifikaatioksi. Kirjallisuuskatsauksessa esiteltiin myös joukko erilaisia suunnittelu ja testausparadigmoja. Tämän tutkielman 2. luvussa käsiteltiin tarkemmin paradigmoja, joita olivat TDD, BDD ja ATDD. Luvussa perehdyttiin myös siihen, kuinka paradigmojen avulla toteutetaan ohjelmistoprojekteja ja kuinka niiden avulla pyritään toteuttamaan laadullisesti parempia ohjelmistoprojekteja. Kuitenkin tarkemmin ajateltuna TDD:n kaltainen ohjelmistokehitys voi monissa pienemmän mittakaavan projekteissa (tässä tapauksessa pienellä ohjelmistoprojektilla tarkoitetaan projekteja jotka kestävät alle 4 kuukautta) olla jopa turhan aikaa vievä lähestymistapa. Esimerkiksi vuorovaikutussuunnittelijan käyttäjätarinoiden tutkimiseen ja toteuttamiseen ja sen pohjalta toteutettuihin testeihin käytetty aika ei välttämättä ole koherenttia verrattuna projektin kokonaistyömäärään. Monesti näissä lyhemmän aikajänteen projekteissa kyseessä on yksinkertaisia ja joka päiväisiä toteutustehtäviä mitkä ovat helppo verifioida yksinkertaisimmilla ja käytännönläheisemmillä menetelmillä. Ohjelmistoa on kuitenkin toivottavaa testata ja tällöin myös toteutuksen jälkeinen testien implementaatio on hyvä tapa toimia.

Nykypäivän ohjelmistokehityksessä harvoin käytetään mitään mallia suoraan, vaan projekteissa on käytössä usein jokin tietty muunnelma jostain yleisesti käytetystä mallista. Mallista käytetty variaatio nähdään juuri kyseiseen projektiin paremmin sopivana, verrattuna esimerkiksi valmiiseen Scrum-kehykseen tai BDD-malliin. Johtuen siitä, kuten termitkin sen sanovat, ne ovat ainoastaan malleja ja viitekehyksiä itse toteutukselle. BDD-mallin kaltaisten automatisoitujen hyväksymistestausten toteutuksiin Nightwatch.js:n kaltaiset automatisoidut päästä-päähän –testit soveltuvat hyvin. Näitä testejä voidaan ajaa esimerkiksi niin kutsuttujen Smoke-testien yhteydessä tai

myös kehityksen yhteydessä. Vaikka Nightwatch.js ei virallisesti olekaan hyväskymistestaustyökalu, eikä BDD-paradigmaan kuuluva testauskehys, niin sen helppo- ja monikäyttöisyys, verrattuna esimerkiksi Solisin ja Wangin käyttämiin ja testaamiin testauskehyksiin, on kehittäjän näkökulmasta erityisen tärkeää. (Solis & Wang, 2011) Esimerkiksi Cucumber-niminen testauskehys on tehty ja toteutettu BDD-malli edellä, eikä esimerkiksi ohjelmistokehityksen tai testauksen näkökulmat edellä. Tämä on johtanut siihen, että kyseisen teknologian tarjoamat ominaisuudet ja käytön helppous eivät välttämättä ole parasta mahdollista kehittäjien arjen kannalta. Näin myös nykypäiväisen ohjelmistokehityksen ja sen projekteihin mukaan integroinnin kannalta virallisten kehysten käyttö ei ole kaikissa tapauksissa järkevää.

5.1 Testaus Solitan kannalta tulevaisuudessa

Kohtaavatko kaikki Solitan ohjelmistokehittäjät samanlaisia tyypitysongelmia kuin tämän tutkielman luvussa neljä esitellyissä haastatteluissa kävi ilmi? Vai kokevatko ainoastaan vanhemmat ohjelmistokehittäjät nämä tietomalleihin liittyvät ongelmat ongelmiksi? Voidaan ajatella, että etenkin virkaiältään nuorempien ja kokemattomampien ohjelmistokehittäjien ongelmat eroavat kokeneempien ammattilaisten tiedostamista ongelmista huomattavasti. Ongelman syynä voidaan pitää sitä, että noviiseilla ei välttämättä ole vielä karttunut tarvittavaa tietotaitoa ja kokemusta hahmottaa testauksen taustalla piilevien ongelmien syitä, kuten esimerkiksi tyypitystä. Nuoremmat ohjelmistokehittäjät saattavat pikemminkin huomioida itse testauskehykseen liittyviä ongelmia ja olla jopa huomioimatta tyypityksen kaltaisia syvempiä ongelmia toteuttaessaan testausta. Yksinkertaisin ratkaisu testaamattomuuteen on tietysti vaatia projekteilta, että niissä tulisi tehdä testausta. Lisäksi testaustyökalujen tulisi olla helppo ottaa käyttöön. Tähän Solitan tulisi tarjota esimerkiksi tietoiskuja uusista testauskehyksistä, sekä koulutusta kuinka yleisimpiä testauskehyksiä käytetään. Tietoiskut ovat Solitan sisäinen tapa jakaa tietoa ja hyviä tapoja kehittäjältä kehittäjille. Myös jonkinlainen selkeä ja helposti käyttöönotettava testaus- tai koodipohja voisi helpottaa testauksen käyttöönottoa projekteissa.

5.1.1 Miksi testata?

Tämän tutkielman tarkoituksena on tuottaa Solita Oy:lle ohjenuorat siihen, kuinka Solita voisi kehittää käyttöliittymäteknologioiden automatisoitua testausta tulevaisuudessa projekteissaan. Tutkielman luvun 4 perusteella selkein huomio, joka nousi haastatteluiden perusteella esiin, on toive automatisoidun testauksen käyttöön ottamisesta mahdollisimman monessa projektissa. Käyttöliittymien automatisoidun testauksen hyödyt tulevat kuitenkin parhaiten esiin monimutkaisemmissa ja laajemmissa projekteissa, joissa komponenttien välisiä riippuvuussuhteita ei voida välttää. Testit estävät etenkin muiden komponenttien rikkoutumista ja niin kutsuttuja ristikkäisvaikutuksien ilmenemistä. Ristikkäisvaikutus voi esiintyä esimerkiksi, kun muutetaan jotain jo toteutettua funktiota tällä hetkellä implementoitavan komponentin tarpeisiin sopivaksi. Tämä tehty muutos hajottaa aikaisemmin toteutetun komponentin ja sen toteutuksen. Tässä tapauksessa aikaisemmin kirjoitettu testi, tälle olemassa olevalle komponentille tai funktiolle, tulee hajoamaan. Näin tiedetään, että myös olemassa oleva toteutus tulee todennäköisesti hajoamaan ja kyseisestä funktiota ei välttämättä pysty muokkaamaan tähän tarpeeseen sopivaksi. Testien tärkeimpänä tehtävänä, edellä mainitun kaltaisessa tilanteessa, on huomata rikkoutuneet komponentit ja toteutukset, jotta ne voidaan korjata ennen niiden pääsyä loppukäyttäjän nähtäväksi.

Testien käyttöönotossa on kuitenkin huomioitava myös tarve. Jos todellista tarvetta teisteille ei ole, niin testejä on turha tällöin lähteä edes toteuttamaan. Jos taas syynä testien toteuttamatta jättämiselle on kehittäjän kokemuksen puute testien toteuttamisesta tai projektin vertaaminen muihin projekteihin, ovat lähtökohdat testien toteuttamistarpeen arvioinnille väärät.

Testien toteuttamisen ajankohdalla (ennen vai jälkeen komponentin implementaation) ei ole väliä, kunhan tarvittavat testit tehdään. Jos projektissa koetaan tarpeelliseksi ottaa luvussa 2 esiteltyjä suunnittelumenetelmiä, kuten BDD:tä tai ATDD:tä, tulisi näiden menetelmien määrittämiä ajankohtia myös noudattaa testien toteuttamisessa. Tarkoituksena tälle on se, että testien toteuttaminen ei esimerkiksi ATDD:n tapauksessa menettäisi alkuperäistä merkitystään. ATDD:n kaltaisessa hyväksymistestauksessa testien toteuttaminen ennen komponenttien

toteuttamista on tärkeää, jotta nähdään ovatko toteutetut komponentit oikeasti valmiita julkaistavaksi. Projektit ja projektien jäsenet kuitenkin eroavat toisistaan, joten myös erilaiset variaatiot näistä suunnittelumenetelmistä ovat todennäköisesti yhtä päteviä kuin alkuperäinen kehyskin tietyissä tilanteissa.

Pohtiessa tarkemmin tulisiko ohjelmistoa testata ja kuinka kattavasti sitä tulisi mahdollisesta testata, tulee aina huomioida projektin lopulliset päämäärät. Kuinka pitkäkestoinen sivuston tai sovelluksen on tarkoitus olla? Onko projektia tarkoitus jatkokehittää tai onko projekti jo ylläpitovaiheessa? Alkuperäiseen ongelmaan ei löydy yksiselitteistä vastausta, joka pätisi kaikissa tilanteissa, mutta mitä pidempi elinkaari projektilla on, sitä todennäköisemmin testausta olisi syytä tehdä. Tavanomaisissa ohjelmistoprojekteissa projektin elinkaaresta voidaan erotella karkeasti kaksi eri vaihetta: aktiivisen kehityksen vaihe ja jatkokehitys. Nämä vaiheet erottavat toisistaan ainoastaan se, että vaiheiden välissä projektin lopputuote julkaistaan ensimmäistä kertaa loppukäyttäjille. Vaiheiden kestot voidaan jakaa niin, että useimmissa ohjelmistoprojektissa aktiivista kehityksen vaihetta mitataan yleensä kuukausilla (alle 2 vuotta) ja jatkokehityksen vaihetta mitataan vuosilla (yli 2 vuotta). Tällöin aktiivisessa vaiheessa testien toteuttamiseen kuluu suhteessa enemmän aikaa verrattuna esimerkiksi projektin kokonaiskestoon. Ajan suhde toteutukseen korreloituu etenkin erittäin lyhytkestoisessa toteutusvaiheessa. Tällöin kehittäjä ei välttämättä osaa ottaa koko palvelun elinkaarta huomioon päätöksiä tehdessään. Usein ennen julkaisua tehtävä toteutus tapahtuu saman kehittäjän tai kehittäjäryhmän toimesta, mutta jatkokehitys ja ylläpitovaihe saattavat ja todennäköisesti sisältävät useita eri kehittäjiä. Näin käy etenkin yrityksen pyrkiessä tarjoamaan (kuten Solita tarjoaa) kehittäjilleen rotaatiota, tarjoamalla heille mahdollisuuden vaihtaa projektia säännöllisin väliajoin. Esimerkiksi 5 vuotta kestävässä jatkokehitysvaiheessa, saattaa olla viisi eri henkilöä ylläpitämässä projektia eri aikoihin. Tämän kaltaisessa tilanteessa alkuperäinen toteuttaja saattaa tietää minkälaisia vaikutuksia hänen toteuttamillaan eri funktioilla on. Tämä ei kuitenkaan päde enää 3 vuotta julkaisun jälkeen jatkokehitysvastuun saavan ohjelmistokehittäjän kanssa, joka ei todennäköisesti näitä eri henkilön toteuttamien funktioiden ristikkäisvaikutuksia tiedä.

Tämänkaltaisen tilanteen välttämiseksi tietysti auttaa ohjelmistojen dokumentaatio, mutta ei aina, eikä dokumentaatiota ole laajassa mittakaavassa kaikissa projekteissa edes järkevää toteuttaa. Paras vaihtoehto tämän varmistamiseen on juuri valmiin toteutuksen testaus ja se, että alkuperäiset toimintavaatimukset täyttyvät myös ylläpidon vuosien jälkeenkin. Varmistamiseen olisi hyvä käyttää esimerkiksi yksikkö- ja päästä-päähän –testausta, jotka voidaan toteuttaa hyväksymistestauksen kaltaisena testauksena. Toinen tärkeä kriteeri testauksen toteuttamiselle on se, kuinka suuret ovat mahdolliset seuraukset siitä, jos ohjelmisto ei toimi kuten sen pitäisi. Onko ohjelmisto esimerkiksi terveydenhuollon projekti, jonka toimimattomuus saattaa altistaa useita ihmishenkiä alttiiksi vaaralle? Vai estääkö komponentin toimimattomuus vain yliopiston viikoittaisen uutiskirjeen tilauksen? Ohjelmiston ja jopa komponentin käyttötarkoituksella on suuri merkitys siihen, tuleeko ohjelmistoa ja sen komponentteja testata. Edellä mainitun kaltaisissa, kriittisissä ja laajoissa järjestelmissä, voi olla jopa tarpeen arvioida yksittäisten osa-alueiden testien kriittisyyttä sen mukaan, kuinka kriittistä ja tärkeää komponenttia kyseinen testi testaa. Tämä saadaan aikaan esimerkiksi arvottamalla testit eri kategorioihin ja validoimalla ohjelmiston kehitysputki. Arvottaminen tapahtuu niin, että tietyn arvoiset komponentit eivät olisi koskaan rikki loppukäyttäjille ja tietyn arvoisten komponenttien testit saisivat olla määritetyn aikaa rikki ennen korjausta. Tämänlainen käytäntö voi olla hyväksyttävää erittäin laajoissa järjestelmissä, ainoastaan tilapäisesti ja joissain erikoistilanteissa, jos resursseja kyseisen ominaisuuden mahdollisille korjaustoimenpiteille ei juuri kyseisellä hetkellä ole saatavilla.

5.1.2 Mitä tulisi testata?

Mitä moderneissa verkkopohjaisissa käyttöliittymissä tulisi testata? Luvun 4 haastatteluissa käy selvästi esille, että kaikki mukaan tuotavat funktiot projektissa olisi hyvä testata. Tavoitteena sovelluksen testaukselle olisi saada korkea testauskattavuus, laadusta kuitenkaan tinkimättä. Kuten tutkielman luvussa 2 on selvitetty, pyritään yksikkötestauksella varmistamaan kaikkien ohjelmiston osien toimivuus oikeellisesti. Snapshot-testaus liitetään myös usein yksikkötestauksen yhteyteen,

kuten esimerkiksi luvun 3 Jestin ja Enzymen avulla tehdyissä yksikkötesteissä on tehty. Snapshot-testausta tulisi tehdä etenkin modernissa verkkopohjaisessa ohjelmistokehityksessä, jossa on käytössä virtuaaliset DOM-puut. Projekteissa tällä pystytään vertaamaan komponenttien tilan oikeellisuutta ja sitä, rakentuvatko komponentit oikein ja säilyttävätkö ne yhtenäisen tilan läpi projektin elinkaaren. Kuten kappaleen alussa mainittiin, testeillä tulisi kattaa kaikki sovelluksen funktiot tai vähintään kaikki mukaan tuodut ohjelmiston moduulit. Mukaan tuoduilla tarkoitetaan kaikkia JavaScript-moduuleja, jotka ovat tuotu import-lauseella mukaan toteutukseen. Tällä pyritään siihen, että kaikkia mukana olevia funktioita pystytään käyttämään turvallisesti. Näin pystytään varmistumaan siitä, että mahdollinen virhetilanne tapahtuu uudessa, toteutettavassa ominaisuudessa, eikä vanhassa ja testatussa funktiossa, jota uusi toteutus ainoastaan hyödyntää.

Yksikkötestejä olisi hyvä ajaa kehityksen yhteydessä, joko ennen koodin viemistä versionhallintaan tai ennen koodin viemistä testi tai tuotantopalvelimelle ja suorittaa testit Smoke-testeinä. Etenkin kehityksen aikaisella testien ajamisella pyritään välttämään turhaa työtä ja huomaamaan mahdolliset virheet ja ongelmatilanteet mahdollisimman aikaisessa vaiheessa. Valittu tapa testien ajamisen ajankohdasta riippuu todennäköisesti projektissa valitsevista olosuhteista ja siitä, minkälaisilla malleilla tai menetelmillä projekti on toteutettu. Tärkeintä projektin kannalta testien ajamisen ajankohdassa on kuitenkin se, että se sopii yhteen kehityksessä käytetyn mallin kanssa. Solitan projekteissa pyritään usein toteuttamaan yksikkötestejä, mutta testit toteutetaan pelkästään taustajärjestelmän logiikan testaukselle. Taustajärjestelmää testataan monipuolisesti ja pyritään vähintään katsomaan, että mikään sivu ei aiheuta palvelintasolla virheitä. Monesti testien toteuttajat ovat kehittämässä myös tai pelkästään taustalogiikkaa, jolloin helposti käyttöliittymätestien toteuttaminen jää taka-alalle, ja pidättäydytään vanhoissa ja *hyväksi koetuissa* tavoissa. Hyväksi koetuissa tavoissa voi olla myös havaittavissa hieman vanha kantaista ajattelua, sillä käyttöliittymien rikkoutumisen ei välttämättä koeta olevan niin paha asia, vaikka se estäisi lopputuotteen käytön osittain kokonaan. Tähän yksinkertaisimpana ratkaisuna olisi tuoda projektin testauksen toteuttamisten käytäntöjä läpinäkyvämmäksi yksittäisten projektien tasolla tai esimerkiksi

tietoiskujen avulla, koko yrityksen tasolle. Näin myös käyttöliittymien kehittäjät pystyisivät paremmin tuomaan esiin omaa osaamistaan ja tietotaitoa testaukseen liittyen.

Päästä-päähän –testauksella tulisi sovelluksessa testata käyttäjätarinat tai komponentin käyttötapaukset, joiden avulla komponentin alkuperäinen vaatimusmäärittely on tehty. Jos varsinaisia tarinoita tai käyttötapauksia ei ole tehty, olisi erittäin suositeltavaa tehdä ne viimeistään testauksen toteuttamisen yhteydessä. Komponentin toiminnan avaus kuvaukseksi voisi jo tässä vaiheessa ennalta ehkäistä väärin toteutettua toiminnallisuutta monissa projekteissa. Testausprosessin kannalta ATDD:n kaltaista ihannetilannetta voitaisiin kuvata kaavion 8. kaltaisena.

Kaavio 8. Scrum-malliin pohjautuva esimerkki hyväksymistestauksen kulusta.

Kaaviossa 8 esitellään tilanne, jossa Scrum-mallin mukainen tuoteomistaja (*Product owner*), eli PO olisi alun perin komponentin käyttötapauksen määrittelyssä mukana, yhdessä vuorovaikutus– tai

käyttäjäkokemussuunnittelijan kanssa. Vaiheen 1 ihannetilanteeksi voidaan ajatella tilannetta, jossa Tuoteomistaja pystyisi toteuttamaan hyväksymistestejä, mutta todennäköisin vaihtoehto testien toteuttajaksi olisi kehittäjä, joka toteuttaisi hyväksymistestin. Testin toteuttamisen jälkeen, 2. vaiheessa, toiselle kehittäjälle annetaan tehtäväksi toteuttaa MVP:n (*minimum viable product*) kaltainen minimitoteutus, jolla testi läpäistään. Tämän jälkeen vaiheessa 3, komponentin kehittäjä ja joko testin alkuperäinen toteuttaja tai kolmas kehittäjä parantelisivat koodin, esimerkiksi parikoodauksen avulla, komponentin parhaaksi mahdolliseksi toteutukseksi. Vaiheessa 4 tuoteomistaja suorittaisi vielä lopullisen hyväksymisen tai hylkäämisen. Vaiheen 4 perusteella, Scrum-mallin mukaisesti, ominaisuus (*backlog item*) voitaisiin siirtää, joko tuotantoon tai palauttaa takaisin työjanoon ja ottaa jatkettavaksi seuraavassa kehitysjaksossa (*sprint*).

Suurin syy miksi käyttötapauksia ei alun perin jo toteuteta projektille on se, että ajatellaan kevyen määrittelyn jo riittävän. Tällöin kuitenkin puhtaasti määritellään mitä ominaisuuksia komponentti tarvitsee toimiakseen halutulla tavalla, eikä sitä, mitä loppukäyttäjä tarvitsee komponentilta. Alkuperäisen käyttötapauksen tai suppean käyttäjätarinan kirjoittaminen olisi hyvä ja helppo tapa päästä myös päästä-päähän –testaukseen käsiksi. Kuvauksen avulla olisi helppo pystyä määrittämään kuvaus päästä-päähän –testille, joka olisi sen pohjalta helpompi toteuttaa.

5.1.3 Käytettävät teknologiat ja tulevaisuus

Testaukseen käytetyillä teknologiavalinnoilla ei periaatteessa ole merkitystä, mutta käytännössä kehittäjän arjessa valinnalla voi olla suurikin merkitys. Valittaessa moderneja ja esimerkiksi kehittäjälle ennestään tuntemattomia työkaluja, voi testauksen toteuttamista pitää mielekkäämpänä. Modernien teknologioiden rinnalle on myös hyvä valita moderneja testaustyökaluja, jotta pystytään testaamaan JavaScript-kielen uusimpia ominaisuuksia. Esimerkiksi ECMAScript-spesifikaatio, johon JavaScript-kielenä perustuu, tuo vuosittain uusia ominaisuuksia määrittelyynsä. Valittaessa moderneja työkaluja testaukseen, tulee kuitenkin muistaa, että vain pieni osa teknologioista saavuttaa pitkäaikaisen ylläpito vaiheen, jolloin työkalun ominaisuuksia päivitetään myös

tulevaisuuden standardia vastaaviksi. Tässä tapauksessa voidaan ajatella, että monien isojen yritysten tukemat teknologiat tai suositut avoimen lähdekoodin teknologiat pystyvät pysymään kehityksessä mukana, sillä niissä todennäköisesti kehitystyö ei ole yhden ylläpitäjän vastuulla. Tutkielman luvussa 4 mainitut teknologiat: Jest ja Enzyme, Nightwatch.js, Chai, Mocha, Karma ja Puppeteer ovat ainakin tällä hetkellä edellä mainitut kriteerit täyttäviä teknologioita. Haastattelun pohjalta voidaan myös todeta, että yksinkertaiset ja yleispätevät testitoteutukset soveltuvat parhaiten pitkäkestoisiin projekteihin. Yksinkertainen toteutus, ilman pitkiä luokkien-valinta listoja (ketjutettuja html-luokan nimiä) on helpompi refaktoroida uuteen toteutukseen tai testaustyökaluun myöhemmässä vaiheessa.

Edellisessä kappaleessa mainitun verkkopohjaisten teknologioiden kehityksen nopeuden voidaan ajatella olevan myös käyttöliittymätestauksen suurin haaste. Jotta pystytään toteuttamaan moderneja verkkosovelluksia, tulisi kehityssyklien pysyä myös ohjelmointikielen kehityksen perässä. Tämä tarkoittaa sitä, että projektia tulee ylläpidon aikana jatkuvasti kehittää ja päivittää vastamaan nykyajan teknologioita. Jatkuva kehittäminen taas voi johtaa monissa projekteissa kasvaviin kustannuksiin, joita asiakas ei välttämättä ole halukas kustantamaan, joka taas johtaa teknisen velan ottamiseen koko projektin ajantasaisuuden kustannuksella. Toinen suuri haaste, joka kävi selkeästi ilmi haastatteluista, on ohjelmistossa käsiteltävän tiedon hallitseminen ja kuinka tätä tietoa pystytään testaamaan. Edellisessä luvussa mainittu GraphQL-teknologia saattaa olla, ainakin vanhempien ohjelmistokehittäjien mielestä, osa-ratkaisua nykyisiin ongelmiin. Myös haastatteluissa ilmi tullut tyypitys voi tulevaisuudessa tuoda helpotusta projektien tiedonkäsittelyyn. Tähän tarkoitukseen esimerkiksi TypeScriptin kaltaista JavaScript-kielen laajennosta voidaan pitää hyvänä vaihtoehtona.

Tulevaisuudessa pilvipohjainen testaus saattaa saada myös enemmän jalansijaa toteutuksissa. Hyötynä Nightcloudin tarjoamassa pilvipalvelussa voidaan ajatella olevan, että testikoodeja ei enää tarvitse ylläpitää manuaalisesti vaan niihin tarjotaan graafinen käyttöliittymä. Tämä voi auttaa esimerkiksi tuoteomistajia toteuttamaan vaatimusmäärittelyiden pohjalta testejä, kun testejä ei enää tarvitse ohjelmoida perinteiseen tapaan. Ongelmia kuitenkin pilvipalveluissa voidaan ajatella

löytyvän etenkin tietoturvaan ja integraatioihin liittyvissä asioissa. Jos projektin luonne on salainen tai tietoturva on erittäin tarkasti suunniteltu, kuinka tällaiseen sovellukseen sopivat testit jotka löytyvät pilvestä? Pilvestä testausta todennäköisesti ei voi ottaa kaikissa projekteissa käyttöön, joten myös itsetoteutetuilla testeillä tulee olemaan tarvetta myös tulevaisuudessa.

Miksi sitten testausta ei kuitenkaan toteuteta projekteissa, vaikka projektitasolla sille olisikin tarvetta? Solitan projekteissa yksittäisellä ohjelmistokehittäjällä on lopulta vastuu ja päätösvalta projektinsa testauksen toteutuksesta. Näin voidaan olettaa, että kaikki ohjelmistokehittäjät eivät pidä käyttöliittymätestausta tai testausta tarpeellisena. Tämä saattaa lopulta olla suurin ongelma, jonka ratkaisemalla saadaan laajempi testauskattavuus. Kuinka tätä mielipidettä voitaisiin sitten jatkossa muuttaa testausmyönteisemmäksi? Tiedon ja osaamisen lisääminen Solitan kehittäjäyhteisössä voi olla avain asemassa mielipiteiden muutokseen. Jos ajatellaan vanhemmassa asemassa olevaa ohjelmistokehittäjää, joka ei ole kymmenvuotisen uransa aikana koskaan toteuttanut projektissaan testejä. Miksi hänen tulisi muuttaa käytäntöjään ja mielipidettään testaukseen liittyen ja ottaa testit käyttöön juuri tässä projektissa? Kun Solitan kehittäjille tarjottaisiin käytössä olevia esimerkkejä testauksesta eri projekteissa ja esiteltäisiin mahdollisia käyttötapauksia tai skenaarioita, voisi se auttaa muuttamaan käsitystä myönteisempään suuntaan. Ajateltaessa tilannetta; *kuinka uusien teknologioiden käyttöönotto saa alkunsa Solitan projekteissa*, alkaa se usein yhdestä henkilöstä joka kokeilee ja esittelee teknologian muille. Tämän tietoiskun seurauksena muut kehittäjät saattavat kokeilla teknologiaa, todeta sen olevan käyttökelpoinen ja ottavan sen käyttöön uudessa projektissansa. Jos projektissa kokemukset teknologian käytöstä on tarpeeksi myönteiset, tulee se tulevaisuudessa osaksi myös muita projekteja. Jos näin ei tapahdu, teknologia useimmiten hylätään ja unohdetaan. Samankaltaista lähestymistä kaivattaisiin myös testaamiseen liittyen. Kehittäjäyhteisössä tarvittaisiin enemmän projektiesittelyitä ja oikeita käyttötapauksia, esimerkiksi; *kuinka ollaan otettu käyttöön uudessa projektissa päästä-päähän –testit tai yksikkötestit*, *kuinka vanhat testausteknologiat on vaihdettu uusiin teknologioihin*, *uutta testausteknologiaa kokeiltiin ja tässä ovat ensi vaikutelmat siitä*. Tähän käsityksen muuttamiseen saatetaan myös kaivata kokeneempien kehittäjien tarjoamaa hyväksyntää ja apua testauksen

toteuttamiselle. Etenkin kokemattomammat ohjelmistokehittäjät kuuntelevat ja ottavat mallia kokeneempien toteutuksista, jolloin testauksen toteutus leviää helpommin. Samankaltaiset tavat leviävät jo itse käyttöliittymien toteutuksissa. Kun testauksesta tulee yrityksen sisällä pikemminkin sääntö kuin poikkeus, vaikuttaa se todennäköisesti myös testausmyönteisyyden leviämiseen. Testauksen toteutuksen lisääntyessä Solitan sisällä, myös tekninen velka tulee vähentymään. Velan vähentymiseen vaikuttaa se, että testit tulisi ajatella osaksi projektin toteutusta ja sitä kautta se tulisi mukaan myös alkuperäisiin työmääräarvioihin. Näin voidaan olettaa, että testauksen toteutuksen hinta olisi helpompi perustella asiakkaalle. Lopullisesti testauksen hyödyt tulevat projekteissa näkyviin asiakkaalle tuotantoon päässeiden virheiden ja niille tehtävien korjausten määrän vähenemisenä.

5.1.4 Lopuksi

Tutkielman yhteydessä toteutettuun haastatteluun olisi voinut varata pidemmän ajan, jotta oltaisiin pystytty keskustelemaan vielä enemmän tulevaisuuden haasteista Solitan kannalta. Kuitenkin kysyttäessä suoraan tulevaisuudesta haasteista, eivät haastatteluun osallistujat nimenneet yhtä yli muiden, jota tässä tutkielmassa ei olisi jo käsitelty. Itse tutkimussuunnitelmaa olisi voinut parantaa niin, että haastatteluissa olisi voinut olla 2 ryhmää. Toinen ryhmä olisi koostunut kokemukseltaan nuoremmista (alle 5 vuotta työkokemusta alalta) ohjelmistokehittäjistä ja toinen ryhmä vanhemmista ohjelmistokehittäjistä (yli 10 vuotta työkokemusta alalta). Tällä saatettaisiin löytää tietoa siitä, ovatko samat ongelmat testauksen kanssa myös aloittelevilla ohjelmistokehittäjillä ja kokevatko he eri asiat ongelmiksi testaukseen liittyen. Tämän kaltainen tutkimusasettelu saattaisi kuitenkin johtaa tutkimukseen vanhempien ja nuorempien ohjelmistokehittäjien eroista, eikä vastaisi kysymykseen Solitan sisäisestä testauksesta. Ongelmana tämän tutkielman yhteydessä laajemman tutkimuksen toteutukselle voidaan pitää myös sitä, että puolet haastatteluun kutsutuista eivät päässeet tai eivät olleet kiinnostuneita osallistumaan tutkimukseen.

Jatkotutkimusta tämän tutkimuksen aihepiiristä voisi jatkaa tarkastelemalla tarkemmin tyypitystä verkkopohjaisissa teknologioissa tai tutkimalla testauksen vaikutuksesta teknisen velan vähenemiseen. Lisäksi jatkotutkimusta voisi tehdä tämän tutkielman haastatteluissa ilmi tulleeseen tietomallien testaukseen liittyvään ongelmaan. Ongelmaan liittyen voitaisiin lisäksi soveltaa ja tutkia tarkemmin GraphQL:n soveltumista ongelman ratkaisemiseksi.

LÄHDELUETTELO

Debill, E. (2017). Module Counts, http://www.modulecounts.com/ (Saavutettu: 4.11.2017)

Esch, M. Virtual-dom. Alkuperäinen toteutus virtuaalisesta DOM-puusta, jota esimerkiksi React käyttää. https://github.com/Matt-Esch/virtual-dom (Saavutettu: 21.1.2018)

Facebook, Inc. (2015) GraphQL. https://graphql.org/ (Saavutettu 22.3.2018)

Facebook, Inc. Jest. Dokumentaatio https://facebook.github.io/jest/

Facebook, Inc. React.js. Dokumentaatio. https://reactjs.org/ (Saavutettu: 25.11.2017)

Fat, N. et al. (2016). Comparison of AngularJS framework testing tools, Zooming Innovation in Consumer Electronics International Conference (ZINC), 2016. IEEE

Fowler, M. (2006) Continuous Integration. ThoughtWorks. Luentomateriaali. University of Alicante, Department of Computer Science & Artificial Intelligence. http://www.dccia.ua.es/dccia/inf/asignaturas/MADS/2013-14/lecturas/10_Fowler_Continuous_Integration.pdf (Saavutettu: 18.11.2017)

Fowler, M (2013) GivenWhenThen. Artikkeli BDD-kehyksen hyvistä nimeämiskäytännöistä https://martinfowler.com/bliki/GivenWhenThen.html

Haviv, A.Q. (2014). MEAN Web Development. Birmingham: Packt Publishing Ltd.

Huggins, J. et all. (2009). Selenium–WebDriver, https://frappe.github.io/charts/WebDriver, https://github.com/SeleniumHQ/selenium

IEEE 729, Standardi. (1983) Standard Glossary of Software Engineering Terminology, IEEE. Hyväksynyt: ANSI.

IEEE 754, Standardi. (1985) IEEE Standard for Binary Floating-Point Arithmetic, IEEE. Hyväksynyt: ANSI

Kitzinger, J. (1995). Introducing focus groups. British medical journal, BMJ

Microsoft. TypeScript Dokumentaatio. https://www.typescriptlang.org/docs/handbook/basic-types.html (Saavutettu 19.3.2018)

Nielsen, J. (1995). Severity Ratings for Usability Problems. https://www.nngroup.com/articles/how-to-rate-the-severity-of-usability-problems/ (Saavutettu: 24.11.2017)

NodeJS. Dokumentaatio. https://nodejs.org/en/about/ (Saavutettu: 18.11.2017)

Richardson, L. (2016), Enzyme: JavaScript Testing utilities for React. https://medium.com/airbnb-engineering/enzyme-javascript-testing-utilities-for-react-a417e5e5090f . (Saavutettu: 17.2.2018)

Ries, E. (2009) Minimum Viable Product: a guide. Startup lessons learned. *Kurssimateriaali* http://www.pbworks.com/ (Saavutettu: 16.2.2018)

Runeson, P. (2006). A survey of unit testing practices. *IEEE Software*, Vol. 23(4)

Rusu, A. Nightwatch.js, http://nightwatchjs.org/ (Saavutettu: 24.11.2017)

Rusu, A. Nightcloud, Nightwatch's own cloud-based testing platform http://nightcloud.io/ (Saavutettu: 24.11.2017)

Solis, C., Wang, X. (2011) A Study of the Characteristics of Behaviour Driven Development. 37th EUROMICRO Conference on Software Engineering and Advanced Application. IEEE

Tilkov, S. Vinoski, S. (2010). Node.js: Using JavaScript to Build High-Performance Network Programs, *IEEE Internet Computing*, Vol. 14(6). IEEE

Van Deursen, A. (2015). Testing Web Applications with State Objects. *Communications of the ACM CACM*. Vol. 58(8), *36-43*. ACM

W3C, World Wide Web Consortium. DOM. (2005). https://www.w3.org/DOM/

W3C, World Wide Web Consortium. WebDriver API, W3C-yhteisön luoma verkkostandardiehdotus, https://www.w3.org/TR/webdriver/ (Saavutettu: 2.12.2017)

LIITTEET

LIITE 1 Toteutetun ohjelmiston lähdekoodi

https://github.com/Fairydusti/feedback-form. (Saavutettu: 14.4.2018)

LIITE 2 Päästä-päähän –testaukseen käytetyn Nightwatch.js:n asennus ohjeet.

Nightwatch.js:n asennus (2.12.2017) (http://nightwatchjs.org/gettingstarted#guide)

Ensimmäisenä vaiheena Nightwatch.js:n käyttöönotossa on asentaa Node.js ja sen paketinhallintatyökalu npm, jonka avulla pystytään lataamaan ja ottamaan käyttöön Nightwatch.js.

1) Asenna Node.js esimerkiksi osoitteesta: https://nodejs.org/en/
2) Asenna npm-paketinhallinta ohjelma esimerkiksi osoitteesta: https://www.npmjs.com/get-npm
3) Asenna npm-paketinhallinta työkalun avulla Nightwatch.js komentoriviltä
   npm install nightwatch -g
4) Lataa Selenium esimerkiksi osoitteesta: http://selenium-release.storage.googleapis.com/index.html. Selenium tarvitsee toimiakseen Java versio 7 tai uudemman. Luo projektin juuri hakemisto kansion alle bin/ hakemisto johon sijoitetaan kaikki projektissa käytössä olevat binaaritiedostot.
5) Asenna selainajurit Mozilla Firefoxia käytettäessä: https://github.com/mozilla/geckodriver ja Google Chromea varten. https://sites.google.com/a/chromium.org/chromedriver/downloads. Siirrä myös nämä ajurit /bin hakemiston alle.
6) Muuta konfiguraatiotiedostossa (nightwatch.json) Seleniumin palvelinosoite osoittamaan uuteen tiedoston sijaintiin, sekä Seeleniumin cli_args kohdan WebDriver ajurien osoitteet osoittamaan ladattuihin selainajureihin bin/ hakemistossa.

Testien ajaminen manuaalisesti:

1) Käynnistää Selenium palvelin manuaalisesti siirtymällä bin/ hakemistoon ja suorittamalla komento: java –jar bin/selenium-server-standalone-{YOUR-VERSION}.jar

2) Aja nightwatch komento projektin juuressa (jossa sijaitsee nightwatch.conf.js tiedosto), jolloin testit ajetaan.

LIITE 3: Haastattelun runkoa ja kysymyspohja.

Osallistujia on etukäteen pyydetty, että he esittelisivät omia esimerkkejään toteutetuista testauksista.

Haastattelun alku: Kerro: Tutkimusetiikka, tutkimuksen tarkoitus.

Osallistujien projektiesimerkit.

(Tarkoituksena on, että toiset näkevät mitä toiset ovat tehneet/testanneet. Tarkoituksena saada kommentointia, herättää muissa osallistujissa kysymyksiä ja kerätä näistä huomioita)

Haastattelu kysymyksiä:

- Onko teillä käytössä automatisoituja testejä?
- Miksi te ylipäätään testaatte käyttöliittymiä?
- Minkälaisiin tilanteisiin end-to-end (päästä-päähän) testit soveltuvat käytettäviksi?

  (Eli käytännössä, mitä niillä halutaan testata / mihin niitä käytetään)

  - Esimerkki tilanne milloin end to end on hyvä?
  - Mihin soveltuu parhaiten?
  - Mitä teknologioita käytätte / mitä meillä on käytössä?

- Minkälaisiin tilanteisiin yksikkötestit soveltuvat käytettäviksi?

  (Eli käytännössä, mitä niillä halutaan testata / mihin niitä käytetään)

  - Mitä mieltä yksikkötestauksen määritelmästä: “Testataan ohjelmiston pienintä osaa, kuten yksittäistä funktiota”.
  - Jos tehdään yksikkötestejä -> toteutuuko yllä oleva ehto, vai ovatko ne ennemminkin integraatio testejä (esimerkiksi kahden funktion yhtenäistä toimintaa)?
    - Esimerkki tilanne?
    - Mihin soveltuu parhaiten?
  - Mitä teknologioita käytätte / mitä meillä on käytössä?

- Mitä ajattelette Solitan projektien tämän hetkisestä käyttöliittymätestauksen tilasta?
- Onko käyttöliittymätestaus Solitassa riittävällä tasolla?

- Miten määrittelette (projekteissa) mitä tulisi testata?

- Behaviour-Driven-Design (BDD) testauksen perustana, onko projektissa käytössä?
- Tehdäänkö testit ennen testattavia komponentteja? (Toimiiko testit siis komponentin hyväksymisen määrittelyinä, niin sanottu: definition of done)
    - Onko tarkoituksena helpottaa kehitystyötä
    - …vai varmistetaanko niillä ennemminkin laatua ja toimivuutta tulevaisuudessa?

- Johdatteleva kysymys: Uskotteko automatisoidun testauksen / automatisoidun testauksen vähentävän virheitä tuotantokoodissa?
- Mitä teknologioita tulevaisuudessa käytetään?
    - Onko menossa pilveen, kuten esimerkiksi Nightcloud?
    - Onko päästä-päähän –testaus menossa enemmän nauhoittamiseen ja web-käyttöliittymiin, kuin koodin kirjoitteluun?

- Jos te saisitte päättää niin miten solitan käyttöliittymä testausta pitäisi kehittää tulevaisuudessa? (Pitäisikö olla minimi vaatimukset projektien toteutukselle, että pitäisi olla X-määrä tehtyjä testejä tai jokin minimi vaatimus testauskattavuudelle.)

- Oletteko kuulleet jostain hyvistä käytännöistä, joita käytetään teidän projekteissa tai muualla talon sisällä? (Jotka eivät vielä ole tulleet esille)
- Tulisiko Solitassa panostaa enemmän automatisoituun testaukseen?

- Tulisiko Solitassa panostaa enemmän manuaaliseen käyttöliittymä ja UX-testaukseen?
- Kun te alatte testata toteutusta, mitä te ensimmäiseksi teette?
- Haluaisitko sanoa jotain yleisesti testauksesta?

Täytekysymykset

* Jos aikaa jää: Näytetään esimerkki yksinkertaisesta laskin sovelluksesta ja selvitetään yhdessä kuinka sovellusta alettaisiin testata. / sovellukselle alettaisiin toteuttaa testejä*

Jos aikaa jää: kerätään yhdessä teesejä Solitan tulevaisuuden testaukselle

Kiitä osallistujia osallistumisesta.