Satakunnan ammattikorkeakoulu
Satakunta University of Applied Sciences

OSKARI ALAJÄRVI

Ohjelmiston E2E-testaus

SÄHKÖ- JA AUTOMAATIOTEKNIIKKA
IT-AUTOMAATIO
2022

Tekijä(t)
Alajärvi, Oskari

Julkaisun laji
Opinnäytetyö, AMK

Päivämäärä
04-2022

Sivumäärä
27

Julkaisun kieli
Suomi

Julkaisun nimi
Ohjelmiston E2E-testaus

Tutkinto-ohjelma
Insinööri (AMK), Sähkö- ja automaatiotekniikka

Tiivistelmä

Tämän opinnäytetyön aiheena oli luoda ohjelmiston End-To-End testaus Hubble Oy:n ostolaskujen lukuohjelmalle. Tavoitteena oli luoda testijärjestelmä, joka testaa ohjelmiston toimivuuden perustasolla.

Työssä kuvataan testien rakentamisen kulku alkuvaiheen selvitystöistä lähtien valmiiseen lopputulokseen saakka. Työssä käytetään paljon kuvakaappauksia koodista ja avataan näiden kautta monimutkaisimmatkin vaiheet.

Opinnäytetyön tuloksena saavutettiin haluttu testauskattavuus koko tuotteelle, joka auttaa Hubblea virheiden hallinnassa automatisoinnin avulla.

Avainsanat
E2E-testaus, End-To-End-testing, Software testing, Cypress

Author(s) Alajärvi, Oskari	Type of Publication Bachelor's thesis / ThesisAMK	Date 04-2022
	Number of pages 27	Language of publication: Finnish

Title of publication
Software E2E-testing

Degree programme
Electrical and Automation Engineering, Bachelor of Engineering

Abstract

The purpose of this thesis was to create End-To-End testing for Hubble Oy purchase order reading software. The goal was to create the testing system, which tests software's functionality as basic level.

This thesis will describe the complete path of creating tests, from beginning till the end. This thesis will use lot of screenshots of the code for describing the whole process.

There will be used lot of a screenshots of the code for describing the most complicated steps as well. As a result of the thesis, the planned testing coverage was achieved to the entire software, which helps Hubble in error handling with automation.

Keywords
E2E-testaus, End-To-End-testing, Software testing, Cypress

SISÄLLYS

TERMI- JA LYHENNELUETTELO

Alusta – Käyttöjärjestelmä, jossa ohjelmistoa käytetään
Applikaatio – Ohjelmisto
BETA-versio – Lähes valmiin lopputuotteen testiversio
BETA-testaus – Lähes valmiin lopputuotteen testaamista kohderyhmällä
Buildaus – Koodattavan ohjelmiston rakennus käytettävään muotoon
E2E / End To End – Päästä päähän ulottuva testausmenetelmä
Flow – Tehtävä toiminto, joka koostuu useammasta toimesta
Käyttöliittymän margin
Response – Paluuviesti palvelimelta
Safari – Applen verkkoselain
Package manager – Paketinhallintajärjestelmä

1 JOHDANTO

Ohjelmistotestaus on muodostunut nykyään yhä tärkeämmäksi osaksi ohjelmistoja, kun on huomattu paremmin sen kannattavuus taloudellisuuden, laadun ja asiakastyytyväisyyden kannalta. Vuonna 2002 tehdyn tutkimuksen mukaan yhdysvaltalaiset ohjelmistotalot menettivät 21.2 miljardia dollaria puutteellisen tai vajavaisen testauksen aiheuttamina tappiona ja tuotannonmenetyksinä (Kasurinen 2013, 11.)

Ohjelmat ovat nykyään yhä suurempia, kattavampia ja ne pyörivät useilla eri alustoilla. Tällöin myös testauksen rooli kasvaa merkittäväksi. Tämän opinnäytetyön tavoitteena on rakentaa automatisoitu testausjärjestelmä Hubble Oy:n tuotteelle Silma.io, joka lukee ostotilauksia. Hubble Oy tuottaa ohjelmistoratkaisuja yrityksien alati laajeneviin tarpeisiin ja kehittää omaa Silma.io ostolaskujen lukuohjelmistoa.

Tällä hetkellä Silma.io lukuohjelman testaus on toiminut manuaalisesti ohjelmistokehittäjien tekemällä manuaalitestauksella. Tarkoituksena ja opinnäytetyön tarpeena on

poistaa mahdolliset ihmisen epätäydellisyydestä johtuvat virheet testauksessa ja automatisoida koko testaus.

Opinnäytetyössä käyn läpi teoriaa testauksesta, testien tarpeellisuudesta ja niiden suunnittelusta. Olen dokumentoinut mahdollisimman tarkasti ohjelmistotestien rakentamisen testikirjastojen asentamisesta testien suorittamiseen saakka. Lopuksi käyn läpi johtopäätöksiä ja pohdintaa. Sen lisäksi, että Hubble Oy saa automatisoidun testauksen opinnäytetyön tuloksena, tarkoituksenani on tuoda myös muille testien tekijöille apua testien tekemiseen.

2 LÄHTÖKOHDAT JA TARVE

2.1 Hubble Oy

Hubble on vuonna 2017 perustettu ketterän kehityksen ohjelmistotalo, jossa halutaan ratkaista asiakkaan arjen ongelmia, toisin sanoen tutustua asiakkaan liiketoimintaan ja ratkaista aidosti ratkaisua vaativat asiat. Hubblen missio on tehdä yritysten arjesta helpompaa ja päästä eroon teknologian äärellä olevasta mystiikasta. Halu tehdä asiat selkärankaisesti ja rehdisti, ajoi yrityksen perustajajäsenet perustamaan Hubblen. Hubble työllistää tällä hetkellä kuusi henkilöä ja liikevaihto nousee vuosittain. Pitkään mielessä muhinut oma tuote päätettiin aloittaa vuoden 2021 aikana ja opinnäytetyöni liittyykin juuri tähän Hubblen omaan tuotteeseen.

Hubblen arvomaailmaan kuuluu vahvasti suoraselkäisyys ja avoimuus. Asiakkaalle tehtävä koodi kuuluu asiakkaalle, eikä sitä piilotella. Asiakas voi halutessaan seurata työn edistymistä, vaikka ihan kooditasolla, jos siihen halukkuutta riittää. Hubblen kanssa työskentely, oli kyseessä sitten työsuhde tai asiakkuus, on aina helppoa,

mukavaa sekä ennen kaikkea avointa. Hubblella asiat tehdään kuten luvataan, ei tehdä eikä laskuteta turhia asioita. Joskus asiakkaalle myös kerrotaan, että asia olisi ratkaistavissa helpommin ja halvemmin, mikä ei ole ollenkaan yleinen tapa ohjelmistoalalla.

Hubblen asiantuntijat koostuvat alan vahvoista ammattilaisista ja työ kuin työ saadaan aina kunnialla maaliin. Hubblella uskalletaan ottaa vastaan vaativatkin työt, joita muut voisivat kavahtaa. Hubblen ajatukseen kuulukin se, ettei ole olemassa asiaa, mitä ei voitaisi ratkaista. Lisäksi Hubblen sivuilla (2022) tuodaan hyvin esille, että työkaverit ovat tasavertaisia eikä silloin välitetä toisten titteleistä tai taustoista. Myös ilmaus ”toimistolla eivät kravatit tai korkkarit kurista” kuvastaa hyvin toimistolla vallitsevaa ilmapiiriä ja Hubblen halua välttää turhaa hierarkiaa.

2.2 Opinnäytetyön tarve

Opinnäytetyön tarve kohdistuu Hubblen oman tuotteen Silma.io:n testaamiseen. Silma.io on ostotilauksia lukeva ohjelmistorobotti, jonka tehtävänä on helpottaa toimistotyöntekijöiden arkea nopeuttamalla ja helpottamalla työntekoa sekä säästämällä yrityksen kustannuksia manuaalisesta työskentelystä ostotilauksien parissa. Ohjelma mahdollistaa ostotilauksia käsitteleville henkilöille vapautuvaa aikaa muiden töiden suorittamiseen, kun turhat manuaaliset työvaiheet poistuvat kokonaan.

Silma.io ohjelmistolla voidaan lukea ostolaskuja. Ohjelmalle opetetaan ostolaskun pohjalta malli, josta se osaa tunnistaa nimen, tilausnumeron, osoitteen sekä tuoterivit. Asiakkaalla on tuotteella yleensä eri tuotekoodi kuin valmistajalla. Kun asiakkaalta tulee ostotilaus tuotteen valmistajalle, joutuu ostolaskun käsittelijä tekemään manuaalista työtä tarkistaessaan omasta vastaavuustaulukostaan oikean tuotteen. Silma.io ohjelmaan tallennetaan vastaavuustaulukko, jolloin ohjelma muuntaa tilauksen automaattisesti datamuotoon oikeilla tuotteilla vastaavuustaulukon mukaan. Tämä helpottaa valtavasti käsittelijän tekemää manuaalista työtä.

2.2.1 Ohjelman toiminta

Ohjelma voidaan asentaa paikallisesti asiakkaan toimintaympäristöön On Premise -asennuksena, tai sitä voidaan käyttää pilvipalveluna, jonka onkin ylivoimaisesti suosituin käyttömuoto. Silma.io:lle voidaan syöttää ostotilauksia manuaalisesti valitsemalla niitä tietokoneelta ja mobiililaitteelta tai vastaanottamalla niitä suoraan sähköisesti sille rakennetun sähköpostin kautta.

Ostotilauksissa on usein asiakkaan omat tuotekoodit, jolloin tavarantoimittajan on tarvinnut käsin katsoa taulukosta heidän vastaavat tuotteet, ja muodostaa heidän omaan järjestelmäänsä tästä tilaus. Silma.io ohjelmaan lisätään asiakaskohtaiset muuntotaulukot, jolloin ostotilaus konvertoituu automaattisesti oikeanlaiseksi.

2.3 Opinnäytetyön tarkoitus

Hubble Oy:n ostolaskujen lukemiseen tarkoitettuun Silma.io -ohjelmaan tehtyjen koodimuutosten jälkeen tulee suorittaa testauksia, jotta voidaan varmistaa ohjelman perustoimintojen olevan kunnossa. Testaukset kuitenkin on jouduttu tähän mennessä tekemään aina käsin, jolloin testaamiseen on kulunut paljon aikaa. Testauksen automatisoinnilla saadaan parannettua tuotteen laatua sekä säästettyä aikaa ja resursseja.

Tämän opinnäytetyön tarkoituksena on rakentaa Silma.io -ohjelmaan kattavat testit perustoiminnoille, jolloin saadaan mahdolliset kehitysvaiheen virheet kiinni ennen niiden päätymistä viralliseen tuotantoversioon. Testi rakennetaan mukailemaan mahdollisimman tarkkaan käyttäjän tapaa käyttää ohjelmaa, jolloin ollaan lähempänä käyttötapaa, jossa virheet esiintyisivät. Ohjelma testataan E2E testauksen mukaisesti käyttöliittymältä kantaan saakka, kirjautumisesta perustoimintojen käyttöön asti.

3 OHJELMISTOTESTAUS

3.1 Miksi ohjelmistoja testataan?

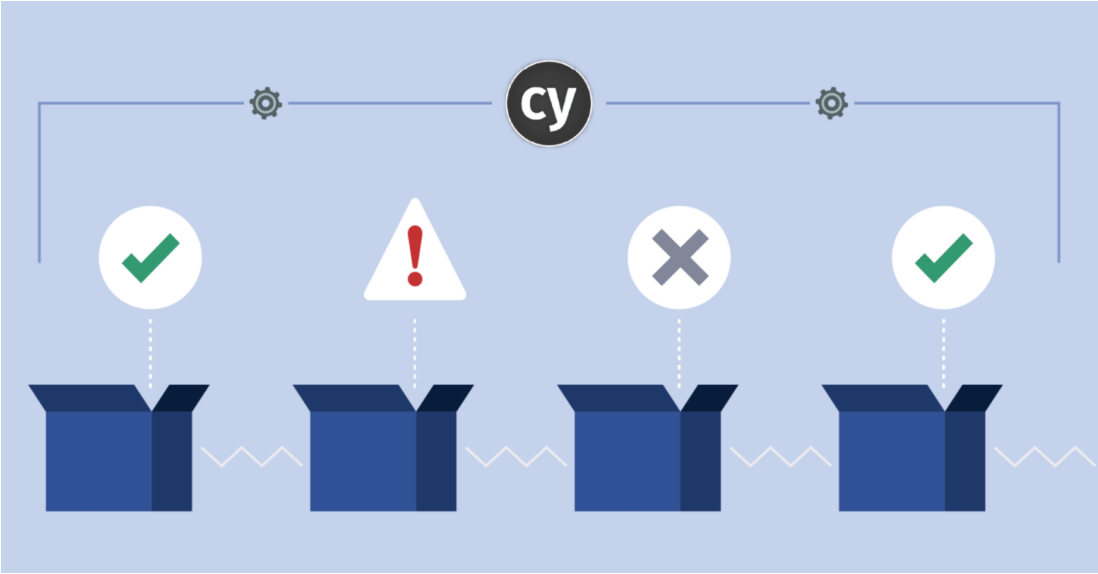
Ohjelmistotestaus on työtä, jonka tarkoituksena on varmistaa, että toteutettava ohjelmistotuote on toivotunlainen ja kaikki siihen kuuluvat ominaisuudet toimivat niin kuin on tarkoitus. Testauksessa tarkastetaan tehty työ, jotta se vastaa sitä, mitä on ollut tarkoituksena tehdä ja tunnistaa ne asiat, joissa tuotos jostain syystä poikkeaa suunnitelmasta. Testaus on kokonaisuutena hyvin laaja asia, laajempi kuin esimerkiksi tekninen kirjoittaminen tai ohjelmointityö. Testaaja joutuu tekemään monia erilaisia asioita ja työvaiheista riippuen voi päätyä koodin kirjoittamisesta dokumentointiin ja koekäyttäjien haastattelemiseen. (Kasurinen 2013, 10.)

Testaamista on hyvin monenlaista tarpeen mukaan ja testejä voidaan toteuttaa eri tavoin. Testaamista tehdään, kun on halu ja tarve reagoida nopeasti ohjelmistossa esiintyviin virheisiin. Testaamisen tärkein tavoite on tuoda esille ohjelmistossa olevat virheet, jotta ne voidaan mahdollisimman nopeasti korjata. Tällä säästetään omaa aikaa ja rahaa sekä loppukäyttäjälle saadaan laadukkaammin toimivan ohjelmiston kautta parempi kokemus (Guru99 2011).

Helposti tulee ajatelleeksi virheiden olevan kehityksessä aiheutuneita virheitä. Kuitenkin kehityksessä on mukana asioita, jotka eivät ensinnäkään ole välttämättä kehittäjän aiheuttamaa, tai voi olla, että koko asiaa ei ole tullut kehitysvaiheessa eteen ja se päästään korjaamaan vasta vian ilmentyessä. Vikoja voivat aiheuttaa monet muutkin seikat kuin kehittäjän kirjoittamat koodivirheet. Niitä voi olla kolmansien osapuolien lisäosien päivittyminen tai selainversiot. Samoin uudet käyttöjärjestelmät, joiden mukana tulleita uusia tai muuttuneita ominaisuuksia ei ole ollut mahdollista huomioida kehityksessä tai toisin päin, uudemmille käyttöjärjestelmille kehitetty ohjelmisto ei ole sopiva aiempien ohjelmistojen kanssa. Tämän vuoksi esimerkiksi monet ohjelmantarjoajat tuottavat omat versiot esimerkiksi eri Windowsin käyttöjärjestelmille.

Tietysti yllä olevat käyttöliittymään kohdistuvat tyylit, kuten marginin toimimattomuus, ei näy automaattitestauksessa vaan vaatii käyttöliittymän käsin testausta

ihmisen toimesta erilaisilla laitteilla, mobiililla sekä työpöytäversiolla. Automaattitestauksella testataan perusrakennetta, esimerkiksi toimiiko kirjautuminen, löytyykö käyttöliittymältä määrättyjä painikkeita, tapahtuuko painikkeen painosta määrätyt asiat. Ohjelma käy läpi kaikki kirjoitetut testit ja koostaa niistä raportin.



Kuva 1. Testitulokset (Zubair 2020)

Yllä olevan kuvan avulla on helppo hahmottaa testien suorituksen tulokset. Jokainen testi on kuvassa oma laatikkonsa, ja yllä olevien ikonien mukaan näemme minkä suorittamisessa on tapahtunut mitäkin. Testauksessa oikein -ikoni on merkki onnistuneesta suorituksesta, kun taas huutomerkki -ikoni on merkki epäonnistumisesta ja taas X-ikoni on merkki suorittamattomuudesta. Testin ansiosta ohjelmistokehittäjä näkee virheen toisessa testissä ja pääsee korjaamaan virheen ennen kuin se päätyy käyttäjälle. Ohjelman laatu ja asiakastyytyväisyys kasvaa, sekä ohjelmantoimittaja säästää rahaa ja aikaa virheen voitaessa korjata aikaisemmassa vaiheessa, näin säästytään myös sivuhaitoilta, mitä saattaa esiintyä, ellei käyttäjä voi käyttää määrättyä toiminnallisuutta.

Testien rakentamisen myötä saatu ajan säästö parantaa ohjelman tuottavuutta, mutta saattaa myös liiallisena aiheuttaa merkittävää rasitetta. Testien rakentamisen voi myös viedä liian pitkälle, jolloin siitä ei hyödy enää kukaan vaan lopputuloksena on erilaisten testien tolkuton kaaos ja niiden ylläpito vie enemmän aikaa kuin koko ohjelmiston kehitystyö. (Georgian, 2021.) Kuitenkin Kasurinen (2013, 12) tuo hyvin esille, että testaustyö on tärkein työvaihe kannattavuuden näkökulmasta katsottuna. Kun

vertaillaan tuotteiden kannattavuutta ja testausta keskenään, on näillä selkeä yhteys toisiinsa. Yritykset, jotka tekevät ohjelmistotestaamista huolellisesti tuotteilleen, saavat paremman katteen tuotteilleen verrattuna yrityksiin, jotka tekevät testaukset huolimattomasti.

3.2 Miten ohjelmistoja testataan?

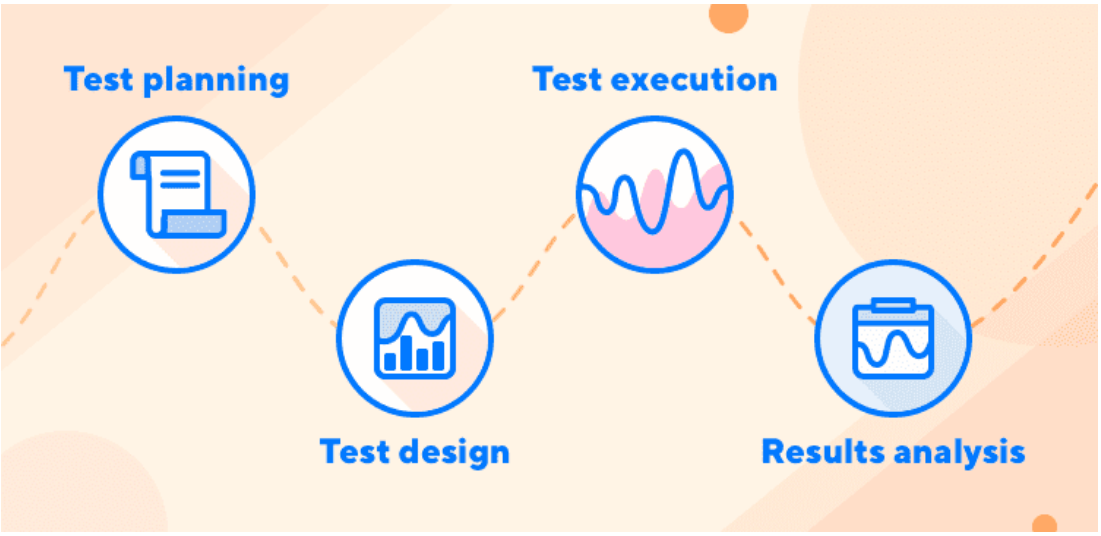
Ihmisen käsin suorittamalla kliksuttelu -testauksella ei päästä koskaan yhtä nopeaan testausnopeuteen, ja ihmisen epätäydellisyyden vuoksi moni asia saattaa jäädä huomaamatta. Tosin tätä manuaalista ihmisen tekemää testausta ei saa suoraan verrata ohjelmiston ”taustatestaamiseen”, koska niillä haetaan esille erityyppisiä ongelmia. Siinä missä taustatestaamisella haetaan esille yksittäisten moduulien tai jopa kokonaisen ohjelma flow’n ongelmat, manuaalisella käsin läpikäynnillä huomataan käyttöliittymän ulkonäölliset ja käytettävyyden ongelmat, sekä osataan huomioida prosessia kokonaisena elementtinä.

Isommissa ohjelmistokokonaisuuksissa testien tekijät ovat usein eri henkilöitä kuin ohjelmiston kehittäjät. Välillä näkeekin työnhakupalstoilla haettavan ohjelmistotaloihin henkilöstöä nimikkeellä ”Software testing engineer”.

Yleisesti ottaen testit rakennetaan, kun ohjelmiston rakenne on valmiina käytettäväksi, eli ammattitermein kun toiminnallisuudet ovat kunnossa. Käyttöliittymän ulkonäöllä ei sinänsä ole tässä asiassa merkitystä. Testit ajetaan erikseen tai buildaamisen yhteydessä aina ennen ohjelmiston julkistamista tai tuotantoversion käyttöönottoa. Tällä menetelmällä saadaan ohjelmiston laatua kasvatettua ja virheiden määrää vähennettyä tuotantoversiossa.

Moni on varmasti nähnyt julkaistuja sovelluksia, jossa lukee ”BETA-versio”. Kyseessä on vielä kehityksen alla oleva tuote, josta on kuitenkin julkaistu versio betatestausta varten. Betatestauksessa ohjelmisto sisältää jo tarvittavat asiat lähtökohtaisesti, mutta saattaa sisältää virheitä, joiden onkin tarkoitus tulla esille betaversion käytössä, jolloin saadaan testattua ohjelmaa ennen virallisen version julkaisemista.

Testaamistapoja on monenlaisia, mutta keskityn tässä työssä niin sanottuun päästä päähän testaamiseen eli E2E-testiin.



Kuva 2. Testien elinkaari (Katalon)

Yllä olevassa kuvassa näkyy hyvin ohjelmistotestin elinkaari, joiden eri osa-alueita käyn läpi seuraavissa kappaleissa. Elinkaari koostuu ohjelmiston suunnittelusta, esimerkkitapauksien suunnittelusta, testien suorittamisesta sekä niiden tuloksien läpikäynnistä. Jokainen osa-alueensa on todella tärkeä, eikä mitään vaihetta voida pitää toista huonompana, jotta testistä saadaan mahdollisimman järkevä ja kattava. (Bose, 2020.)

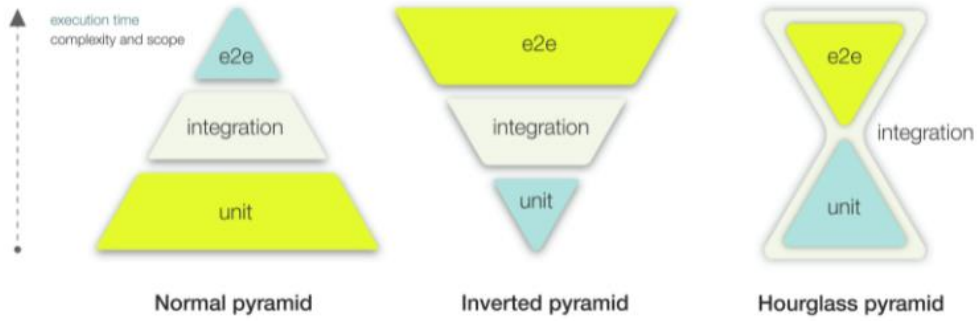
3.3 Testauksen suunnittelu

Ennen ohjelmiston testien tekemisen aloittamista on hyvä suunnitella mitä testataan ja millä tavalla. On myös tärkeää asettaa rajaukset testattaville asioille, sillä liian monimutkainen testirakenne aiheuttaa enemmän vaivaa kuin hyötyä, joten on syytä välttää testien teossa liiallisuutta. Tärkeää on testata kriittiset perustoiminnot ohjelman toiminnallisuuden kannalta. Testauksen suunnittelussa testattavaa materiaalia käydään läpi yleistasolla, mitä aiotaan testata ja millä menetelmällä.

E2E testimenetelmässä saadaan testattua kokonainen polku. Sen kääntöpuolena on kuitenkin hitaus ja suoritus kestää yleensä 5-10min, isoissa kokonaisuuksissa vielä

kauemmin. Sen avulla voidaan testata paljon toiminnallisuuksia yhdellä kertaa ja ennen kaikkea samalla tavalla, kuin käyttäjä ohjelmaa käyttäisi, joka on sen suurin etu (Marketing Ranorex, 2021). Suunnittelijan on todella ymmärrettävä ohjelman toiminta kokonaisuudessaan, sekä osattava ajatella asiakkaan silmin sen käyttöä. Tärkein ajatus on simuloida mahdollisimman tarkasti testissä todellisen käyttäjän tekemä vastaava operaatio (Georgian 2021).

Suunnitellessa testien kattavuutta päästään asiaan, joka voi mennä pahasti pieleen. Tällöin pitää harkita vakavasti tarvitaanko tätä testiä, jos tarvitaan niin, kuinka paljon. Liian kattava E2E testijärjestelmä kuluttaa resursseja enemmän, mitä se vapauttaa. Varsinkin testaajan uraa aloittelevat henkilöt kirjoittavat testejä liikaa, sillä oikea balanssi löytyy kokemuksen ja ajan myötä. (Martinez, 2020.)

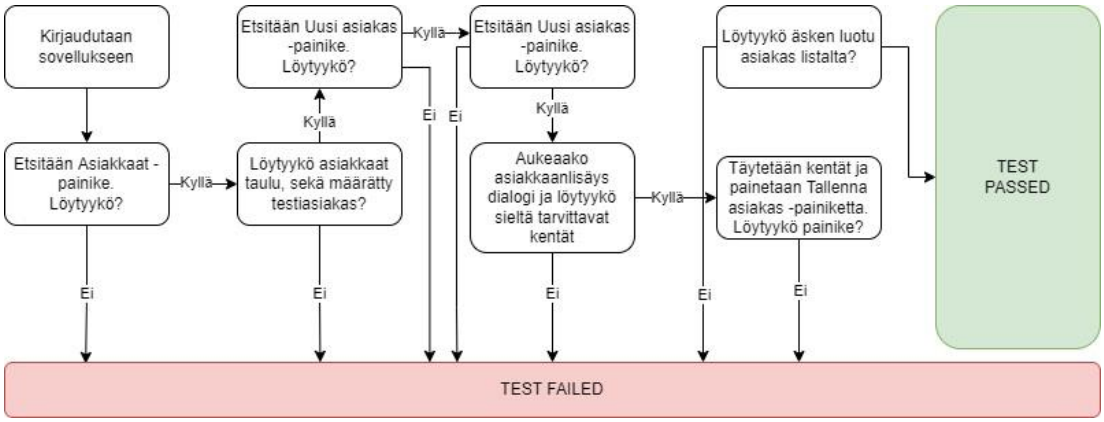


Kuva 3. Esimerkkejä testipyramidin rakenteista (Georgian 2021)

Pyramidi (kuva 3) kertoo yleisen suuntaviivan testien jakautumiselle, miten niiden olisi hyvä jakautua. Kuitenkin tämä on asia, jota ei noudateta kovinkaan tarkasti, vaan tilanteen mukaan rakennetaan testit, joita ohjelmistokehittäjän tai testikoodarin mielestä ikinä tarvitaankaan. Useimmiten ohjelmistotestien jakautuminen muistuttaa rakenteeltaan enemmän tiimalasia. (Georgian, 2021.)

3.4 Esimerkkitapauksien suunnittelu

Testauksen suunnittelussa on hyvä käydä ohjelmaa läpi ja piirtää siitä testauskaavio (laita alle kuva), jonka mukaan testejä on helppo rakentaa. Testauksen suunnittelu on lähes välttämätön osa testien tekoa.



Kuva 4. Asiakasnäkymän testisuunnittelu

4 OHJELMISTOTESTIEN RAKENTAMINEN

Valitsin projektiin Cypressin, joka on JavaScript pohjainen End-to-End testikirjasto ja siinä käytetään Mochan bdd -syntaksia, jolla testit kirjoitetaan. Syntaksi on yleinen testikirjastoissa, joten sen oppiminen auttaa testitöissä myös yleisellä tasolla. Mochan mukana tulee myös tärkeä async -tuki, joka odottaa määrätyn ajan testimetodissa haettavaa elementtiä (Cypress www-sivu, 2022). Haettaessa elementtiä käyttöliittymältä, saattaa testin alun ja elementin esiintulon välillä olla useita sekuntia renderöintinopeudesta riippuen. Tällöin ilman async -tukea testi päättyisi virheeseen, koska haettua elementtiä ei löytynyt.

4.1 Testikirjaston asentaminen

Cypress asennetaan samaan kansioon käyttöliittymän kanssa, tässä tapauksessa ui -kansioon. Asennus suoritetaan terminaalissa komennolla npm install cypress –save-dev, joka käynnistää asennusprosessin. Prosessi etenee itsekseen, eikä käyttäjältä vaadita tässä kohtaa mitään toimenpiteinä. Cypressin asentuessa siitä tulee ilmoitus ja tavallinen terminaalinen kirjoitusrivi aukeaa jälleen.

Asennuksen jälkeen Cypressiin asetetaan perussäädöt, joka kertoo testiohjelmalle esimerkiksi missä kansiossa testit sijaitsevat, missä kansiossa support -tiedosto sijaitsee ja millaisin komennoin testiä voidaan ajaa. Cypress käy jokaisen käynnistymisen yhteydessä läpi asetustiedostot, joiden mukaan se käynnistyy. Cypress kansion sisältä löytyy plugins -kansio, jossa sijainnit sisältävä index.js -tiedosto sijaitsee. Tämä tiedosto pitää sisällään määritykset kansio- ja tiedostosijainnille, joita testissä käytetään.

```
module.exports = (on, config) => {
  // on('file:preprocessor', webpack({
  //  webpackOptions: require('@vue/cli-service/webpack.config'),
  //  watchOptions: {}
  // }))

  return Object.assign( target: {}, config,  source2: {
    fixturesFolder: 'cypress/fixtures',
    integrationFolder: 'cypress/integration',

    screenshotsFolder: 'cypress/screenshots',
    videosFolder: 'cypress/videos',
    supportFile: 'cypress/support/index.js'
  })
}

```

Kuva 5. Index.js -tiedosto plugins -kansiossa

Yllä olevassa kuvakaappauksessa index.js tiedostosta, määritellään sijainnit datatiedostoille (fixtures), integrationtiedostoille (testit), screenshot kuville (kuvakaappaukset testistä), videotiedostoille (testien taltiointi), sekä support -tiedostolle (konfiguraatiotiedosto). Tässä projektissa perusasetukset näiden osilta pidetään ennallaan, koska tarvetta muutoksille ei ole. Ainoat perusasetusmuutokset, jota tein on testien ajoon tarkoitettujen komentojen muutokset package.json tiedostoon, joka sisältyy aina jokaiseen Vue -projektiin. Sieltä löytyy ohjelmat skriptit (komennot) ja muut projektin asetukset. Voidakseni käyttää projektista tuttua npm run -komentoa testeille, on luotava skriptit package.json tiedostoon. Komennot ovat test:e2e ja test:e2e-h, joista jälkimmäinen on headlessina (ilman käyttöliittymää, terminaalissa suorittaen) ajettava testi. Edellä mainitut komennot ovat itse määrittämiäni, ja tekevät alla olevan kuvan mukaiset toimenpiteet. Projektiin on asennettu Vue CLI Service, jolloin voi käyttää näitä komentoja. Normaalisti Cypress käynnistettäisiin npx cypress run käyttäen npm package manageria tai yarn cypress run käyttäen yarn package manageria.

```
"test:e2e": "vue-cli-service test:e2e --mode development",
"test:e2e-h": "vue-cli-service test:e2e --mode development --headless",
```

Kuva 6. Package.json -tiedosto skriptit

4.2 Suunniteltujen testien rakentaminen

Testien kirjoittamisen aloitin sisään- ja uloskirjautumisen metodeista, sekä tämän asian testaamisesta. Ohjelmaan voi kirjautua sisään ja näkyville tulee määritellyt asiat, sekä uloskirjautuminen toimii. Alla sisään- ja uloskirjautumisen metodit toimintoineen, jotka kerron seuraavaksi. Kirjautumisen metodit tein omiin metodeihinsa, koska Cypress vaatii kirjautumisen jokaisessa testissä, joten tätä kokonaisuutta ei tarvitse kirjoittaa jokaiseen testiin, vaan voidaan kutsu pelkkää metodia.

```
export function login() {
  const username = '[redacted]';
  const password = '[redacted]'
  cy.visit('/#/admin');
  cy.get('[data-cy=admin-username]').type(username, options: { sensitive: true });
  cy.get('[data-cy=admin-password]').type(password, options: { sensitive: true });
  cy.get('[data-cy=admin-login]').click();
  cy.wait(3000);
}
```

Kuva 7. Login -metodi

Login -metodissa määritellään sisäänkirjautumiseen tarvittavat muuttujat, username (käyttäjätunnus) ja password (salasana). Tämän jälkeen mennään pääkäyttäjän kirjautumissivulle, etsitään elementit joihin on määritetty html datamuuttuja data-cy admin-username ja admin-password. Näihin elementteihin, jotka ovat käyttöliittymällä tekstikentät, kirjoitetaan (.type()) muuttujiin määritellyt merkkijonot sensitive asetuksella, jolloin kirjoitetut käyttäjätunnus ja salasana eivät näy testilogissa. Kirjautumistietojen kirjoittamisen jälkeen etsitään kirjautumispainike, jota painetaan ja odotetaan 3000 millisekuntia, jonka aikana käyttöliittymä kerkeää latautumaan.

```
export function logout() {
  cy.visit('/');
  cy.get('[data-cy=user-menu]').should( chainer: 'be.visible').click();
  cy.get('[data-cy=user-menu-logout]').should( chainer: 'be.visible').click();
}
```

Kuva 8. Logout -metodi

Logout -metodissa mennään ohjelman juuriosoitteeseen, etsitään menuvalikko, jolle on määritetty html datamuuttuja user-menu. Jos menuvalikko löytyy, klikataan se auki. Tämän jälkeen etsitään uloskirjautumispainike, jolle on määritetty html datamuuttuja

user-menu-logout. Jos uloskirjautumispainike löytyy, klikataan sitä. Alla kuva, jossa näkyy nimen html datamuuttuja määritellään elementille.

```
<v-textarea
  data-cy="item-description"
  v-model="item.description"
  :dense="$vuetify.breakpoint.xsOnly"
  :disabled="saving"
  :label="$t('items.description')"
  rows="3"
  validate-on-blur/>
<template v-for="attribute in attributes">
  <AttributeInput :key="attribute.id" v-model="attribute.value" :attribute="attribute" :disabled="saving"/>
</template>
```

Kuva 9. Html datamuuttuja asetus elementille

Kirjautumistestien jälkeen ollaan hyvällä perustasolla, josta voidaan lähteä testaamaan yksittäisiä näkymiä. Kun saadaan rakennettua yhden näkymän testit hyvällä tasolle, on sitä helppo kopioida ja muokata muille näkymille. Testaamassani ohjelmassa on asiakasyrityksiä (party), joidenka näkymän toimivuutta testaan. Suunnitelmani mukaan katsotaan, että asiakastaulu näkyy sivulla, sinne saadaan luotua uusia kohteita, sekä niitä saadaan poistettua.

```
import 'cypress-keycloak';
import {login} from "../../components/login";
import {customVisit} from "../../components/visit";

const viewports = [
  {
    name: 'Mobile',
    size: [390, 844]
  },
  {
    name: 'Desktop',
    size: [2560, 1600]
  }
];

const partyName = 'cypress-name-1';
const partyCode = 'cypress-code-1';
const conversionTableItemCode = '224466';
```

Kuva 10. Asiakasyrityksien testin vaihe 1/6

Testiin tuodaan login ja customVisit metodit, joita voidaan kutsua testin edetessä. CustomVisit -metodin käyn tarkemmin läpi tullessani vaiheeseen, jossa sitä kutsutaan näin ollen saadaan pidettyä loogisuus ja järjestelmällisyys kohdillaan. Cypressissä on asetettuna oletus viewport (näkymäkoko), jonka oletustakin voi muuttaa, mutta tarve on kuitenkin testata sekä työpöytänäkymällä että mobiililla, joten nämä määritellään testissä. Asetetaan viewport objektit arrayn (lista) sisälle. Määritän sinne nimen ja koon. koska nimeä taas tarvitaan testilogissa testien selkeyttämiseksi onko testi mobiilinäkymällä vai työpöytänäkymällä. Koko asetetaan pikseleinä leveys x korkeus ja arvoiksi on määritelty yleisimmät ruutukoot. Tämän jälkeen määritellään partyName (asiakasyrityksen nimi), partyCode (asiakasyrityksen koodi) ja conversionTableItemCode (muuntotaulukon koodi).

```
21  viewports.forEach((viewport :{name: string, size: number[]} ) => {
22
23    describe( title: `▼ Parties - ${viewport.name.toUpperCase()} ▼`, fn: () => {
24
```

Kuva 11. Asiakasyrityksien testin vaihe 2/6

Viewportit käydään läpi yksitellen forEach() -metodilla, jossa määritellään tehtävät asiat milläkin ruutukoolla. Testi alkaa describe määritteellä, joka kertoo testiluokan nimen. Kaikki testiluokan alla olevat alkavat määritteellä it. Testin describe ja it -määritteet periytyvät myös aiemmin mainitun Mochan syntaxista (kirjoitustapa). Testin nimi määritellään englannin kielellä Parties (asiakasyritykset), jonka jälkeen asetetaan viewportin nimi objektista. Nimessä käytettävä viewportin nimi asetetaan isoiksi kirjaimiksi käyttämällä Javascriptin metodia .toUpperCase(), lähinnä testirakenteen selkeyttämisen vuoksi.

Kuva 12. Asiakasyrityksien testin vaihe 3/6

Testiluokan jälkeen määritellään varsinaiset testit. Kuvassa 12, rivillä 25 asetetaan testiehto, jos viewportin nimi on Desktop, eli vuorossa on työpöytäkoon viewport, suoritetaan sen sisällä olevat testi. Jos nimi on jokin muu kuin Desktop, suoritetaan else sisällä olevat testi. Molemmissa näkymissä tehdään samat asiat, työpöytä- ja mobiiliversion menurakenteet ovat erilaiset, joten testi on tehtävä eri tavalla koska menu on eri työpöytä- ja mobiiliversiossa.

Määritellään nimi yksittäiselle testille Party view works correctly (Asiakasyritysnäkymä toimii oikein). Nimen mukaisesti testissä testataan asiakasyritysnäkymän perusasiat, sinne pääsee menusta navigoimalla ja asiakasyritystaulu on näkyvissä. Ensin asetetaan viewport oikean kokoiseksi käyttämällä vy.viewport() -metodia. Tämän jälkeen kirjaudutaan sisään haetulla metodilla login(). Kirjautumisen jälkeen etsitään menun pääpainike datamuuttujalla menu-basic-info, jonka jälkeen avautuvasta valikosta painetaan datamuuttujalla menu-parties määritettyä painiketta. Ennen asiakasyritystaulun etsintää suoritetaan 3000 millisekunnin odotusaika cy.wait() -metodilla. Metodeilla cy.get() haetaan haluttua asiaa, joka on käytännössä html datamuuttuja. Haussa voidaan käyttää class -tyylielementtiä tai id -elementtiä. Class ja id voivat kuitenkin olla samalla sivulla samanlaisia, joten tätä tapaa ei suositella käytettäväksi. Metodilla should() määritetään asia mitä pitää olla, tässä tapauksessa sen pitää olla näkyvissä, be.visible. Haettua elementtiä voidaan myös klikata metodilla .click(), jota käytetään esimerkiksi rivillä 29 klikatessa menun painiketta.

Kuva 12. Asiakasyrityksien testin vaihe 4/6

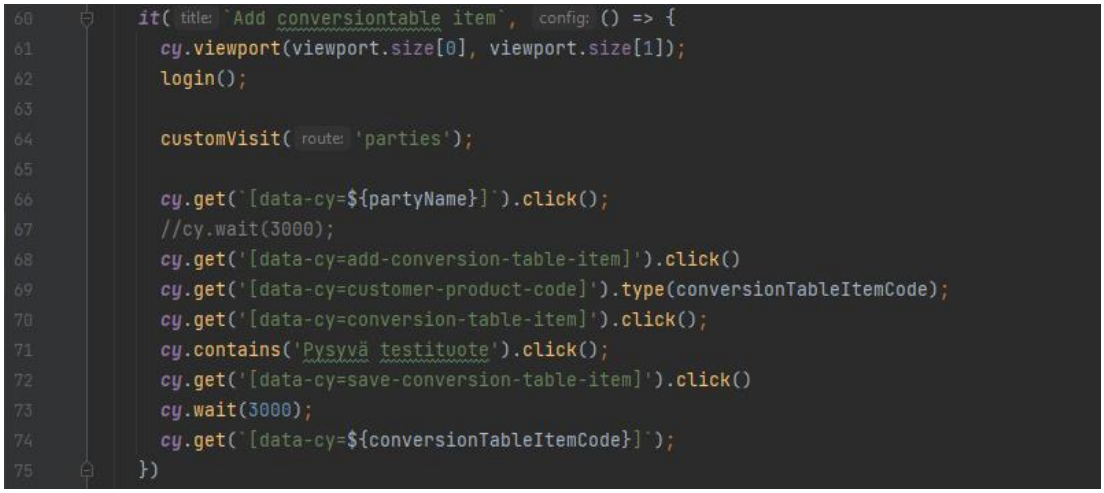
Kuvassa 12 käydään läpi asiakasyrityksen luontioperaatio. Cypress tyhjentää selaustiedot jokaisen testin jälkeen, joten kirjautuminen on suoritettava aina ennen testin muita osioita. Seuraavaksi navigoidaan asiakasyritykset sivulle suoraan, eli ei mennä menun kautta, kuten aiemmassa testissä kuvassa 12. Silloin testattiin menurakenteen toimivuutta, jolle ei ole enää toista kertaa tarvetta.

Kuva 13. CustomVisit -metodi asiakasyrityksien testissä

CustomVisit -metodissa haetaan local storagesta (selaimen muisti) ohjelman tenant (käyttäjä) tieto, jota tarvitaan routen (polku) asettamisessa. Määritellään itemKey-Name (local storagesta etsittävä avain), jonka arvolle annetaan nimi tenant. Tämän jälkeen on mahdollista navigoida uuteen osoitteeseen, kun kaikki siihen tarvittavat tiedot ovat saatavilla, sillä route tuodaan metodille sitä kutsuttaessa (kutsutaan myös passaamiseksi). Nämä tiedot asetetaan cy.visit() metodille annettuun linkkiin muuttujina.

Tämän jälkeen kuvassa 12, rivillä 51, haetaan käyttöliittymältä uuden asiakasyrityksen luontiin tarkoitettua painiketta, jolle on määritetty data-cy muuttuja create-new-party. Klikataan tuota elementtiä ja etsitään kenttä nimelle ja koodille, jonka jälkeen

kirjoitetaan niihin aiemmin määrityt muuttujat .type() -metodilla. Painetaan tallennuspainiketta, jolle on määritetty data-cy muuttuja save. Seuraavassa vaiheessa riveillä 55-57 asetetaan odotus, jolloin asiakasyritys kerkeää tallentumaan, ja katsotaan .contains() -metodilla, että asiakasyritystaulu sisältää äsken luodun asiakasyrityksen nimen.



Kuva 14. Asiakasyrityksien testin vaihe 5/6

Asiakasyritystaulussa jokaisella asiakasyritysriville on annettu data-cy nimeksi asiakasyrityksen nimi, joten voidaan hakea elementtiä asiakasyrityksen nimellä muuttujan mukaan. Avataan asiakasyrityksen näkymä klikkaamalla elementtiä. Tämän jälkeen lisätään muuntotaulukkoon uusi rivi klikkaamalla rivinlisäyspainiketta, jonka data-cy muuttuja on add-conversion-table-item. Etsitään kenttä, johon kirjoitetaan muuntokoodi, ja valitaan alasvetolaatikosta (data-cy muuttuja conversion-table-item) tuote nimellä Pysyvä testituote. Tämän jälkeen muuntotaulukon rivi tallennetaan ja 3000 millisekunnin odotusajan jälkeen katsotaan, että kohde löytyy taulusta.

Kuva 15. Asiakasyrityksien testin vaihe 6/6

Muuntotaulukon rivejä, sekä asiakasyrityksiä ei normaali tilanteessa poisteta kannasta koskaan, vaan niitä poistettaessa niille asetetaan päivämäärätietoa poistosta, jolloin sitä ei näytetä enää käyttöliittymällä. Tämä johtuen siitä, että saattaa olla ostotilauksia, jossa liitettynä tämä asiakasyritys. Tämän vuoksi tein api kutsun testissä luodun datan poistoon ja poistometodit backendille (ohjelmiston osa, jonne käyttöliittymältä lähetetään kutsut, ja joka hoitaa kantaa). Näin ollen kantaan ei kerry testidataa vaan se saadaan pidettyä puhtaana.

Kuvassa haetaan tenant -tieto vastaavalla tavalla, mitä kuvassa 13 customVisit -metodissa. Tenantin koodia tarvitaan apikutsun urlissa (kutsuosoite), sekä headersissa (kutsun mukana lähtevät headers tietueet). Kutsun jälkeen poiston responsen (palautuksen) odotetaan olevan http statuskoodilla 200 (onnistunut kutsu). Tämän jälkeen asiakasyrityksien näkymä on testattu kokonaisuudessaan.

```
@RestController
@RequestMapping("api/tenant/party")
public class PartyController {

@PostMapping("clean_test_data")
public void cleanTestData() {
  partyService.cleanTestData();
}
```

Kuva 16. Testidatan poistokutsu backendin controllerissa (ottaa vastaan apikutsut)

```
  void cleanTestData();
```

Kuva 17. PartyServicessä oleva testidatan poistoon tehty metodi

```
public void cleanTestData() {
  String description = "cypress-";
  List<Party> parties = partyRepository.findAllByCodeStartsWith(description);
  partyRepository.deleteAll(parties);
}
```

Kuva 18. PartyServiceImpl tiedostossa oleva testidatan poistoon tehty metodi

Kuvassa 16 Java backendissä kutsutaan partyservicen (asiakasyrityksien metodit) metodia cleanTestData() (kuva 17), joka taas kutsuu varsinaista poistometodia PartyServiceImpl tiedostossa (kuva 18). Määritellään merkkijonomuuttuja description, joka on cypress-. Tällä merkkijonolla alkaa kaikkien testitarkoituksena luotujen asiakasyrityksen kuvaus. Haetaan lista kaikista tällaisista asiakasyrityksistä ja kutsutaan partyRepositoryn komentoa deleteAll, jolle annetaan tuo kyseinen koostettu lista.

4.3 Testien suorittaminen

Testit voidaan ajaa alkuvaiheessa määritellysti joko Cypress käyttöliittymällä tai headlessina (terminaalissa) kuvassa 6 määritellyin komennoin. Tässä tilanteessa haluan käyttää Cypressin käyttöliittymää ja ajaa pelkästään asiakasyrityksiä koskevat testit. Ajan testit siis komennolla npm run test:e2e. Cypress käynnistää palvelimen, jolla ohjelma pyörii ja avaa käyttöliittymän testaamista varten. Suoritettuani asiakasyrityksien testit saan dataa siitä, mikä meni läpi ja mikä ei. Ohjelmassa on kaikki kunnossa, joten testi menee läpi, mutta esimerkkitapausta varten tein kirjoitusvirheen yhteen data-cy muuttujaan, jotta voisin liittää tähän dokumentin myös tilanteesta, jolloin kaikki ei suoriudu onnistuneesti. Kuvat alla.

Kuva 19. Testien ajo onnistuneesti

5 JOHTOPÄÄTÖKSET JA POHDINTA

Ohjelmistotestaus voi monelle ohjelmistokehittäjälle olla epämieluinen asia suorittaa, sillä testaus voidaan kokea ylimääräisenä asiana jo ennestään hyvin toimivalle ohjelmistolle. Kuitenkin se on erittäin tärkeä osa-alue ohjelmistokehityksessä esimerkiksi tuotteen kannattavuudenkin näkökulmasta. Tästä syystä aihealue oli opinnäytetyöksi erittäin mielenkiintoinen ja testien toteuttaminen oli hyvin opettavaista. Opinnäytetyön tavoitteena oli saada rakennettua testit Silma.io -ohjelman perustoiminnoille, joita suoritetaan eri vaiheissa ohjelman tuotantoa. Opinnäytetyön tuloksena sain rakennettua kattavat testit kaikille Silmä.io:n perustoiminnoille ja dokumentoitua testien tekemisen hyvin yksityiskohtaisesti. Testien ansiosta ohjelmaan ei pääse syntymään tuotantoversioon asti pääseviä virheitä ja virheiden löytäminen on huomattavasti helpompaa.

Testejä suunnitellessani huomasin, kuinka tärkeää on olla täydellisesti tietoinen kaikista ohjelman toiminnallisuuksista. Testaajan on tunnettava hyvin ohjelmisto ja osattava myös ajatella monia erilaisia variaatioita käytölle demonstroidessaan käyttäjän tekemää liikettä testissä, muutoin testien tekeminen on vaikeaa. Käyttäjiä ja käyttötapoja on monenlaisia, jonka vuoksi testin tekijän on osattava ajatella ohjelman käyttöä monelta erilaiselta näkökulmalta. Testaajan on ikään kuin osattava ajatella ohjelmisto muiden silmin.

Olen henkilökohtaisesti oppinut testauksesta valtavasti tämän opinnäytetyön pohjalta ja ymmärtänyt testausten tarkoituksen sekä tärkeyden ohjelmistokehittämisessä. Nykyään testaus kuuluukin osana työtehtäviäni. Yksityiskohtainen ohjelmiston testauksen kuvaaminen Cypress testikirjastolla toivottavasti antaa hyvät välineet myös opinnäytetyön lukijoille suorittaa ohjelmiston testausta.

LÄHTEET

Bose S. (2020). Viitattu 22.3.2022 https://www.browserstack.com/guide/test-planning

Georgian, S. (2021). What is End-to-End Testing and When Should You Use It? Viitattu 19.12.2021. https://www.freecodecamp.org/news/end-to-end-testing-tutorial/

Georgian, S. (2021). Why you Should Respect the Test Pyramid? [kuva 3]. Viitattu 19.12.2021. https://www.freecodecamp.org/news/end-to-end-testing-tutorial/

Cypress. (2022). Bundled Tools. Viitattu 30.1.2022. https://docs.cypress.io/guides/references/bundled-tools#Mocha

Georgian, S. (2021). What is End-to-End Testing? [kuva 4]. Viitattu 19.12.2021. https://www.freecodecamp.org/news/end-to-end-testing-tutorial/

Guru99. (2011). What is Unit Testing? – Software Testing Tutorial [kuva 2]. Viitattu 17.12.2021. https://www.youtube.com/watch?v=lj5nnGa_DIw

Hamilton, T. (2021). Unit Testing Tutorial: What is, Types, Tools & Test EXAMPLE. Viitattu 16.12.2021. https://www.guru99.com/unit-testing-guide.html

Kasurinen, J. (2013). Ohjelmistotestauksen käsikirja. 1. painos. Jyväskylä: Dosendo.

Katalon. What is End-to-End (E2E) Testing? All You Need to Know [kuva 2]. Viitattu 21.1.2022. https://katalon.com/resources-center/blog/end-to-end-e2e-testing

Marketing Ranorex. (2021). The Pros and Cons of End-to-End Testing. Viitattu 27.1.2022. https://www.ranorex.com/blog/end-to-end-testing-pros-cons-benefits/

Martinez, D. (2020). Avoid These 3 Mistakes With Your End-To-End Tests. Viitattu 28.1.2022. https://dev-tester.com/avoid-these-3-mistakes-with-your-end-to-end-tests/

Stamström, M. (2019). Making E2E testing easy with Cypress. Viitattu 15.12.2021. https://dev.to/mstamstrom/making-e2e-testing-easy-with-cypress-pk5

Stamström, M. (2019). Writing tests with Mocks [kuva 1]. Viitattu 15.12.2021. https://dev.to/mstamstrom/making-e2e-testing-easy-with-cypress-pk5

Zubair, A. (2020). Modern ways of end-to-end testing with Cypress JS [kuva 1]. Viitattu 21.1.2022. https://mattermost.com/blog/modern-ways-of-end-to-end-testing-with-cypress-js/