OULUN AMMATTIKORKEAKOULU

Joni Väisänen

TESTIAUTOMAATION JATKOKEHITTÄMINEN

TESTIAUTOMAATION JATKOKEHITTÄMINEN

Joni Väisänen
Opinnäytetyö
Kevät 2019
Tietotekniikan tutkinto-ohjelma
Oulun ammattikorkeakoulu

TIIVISTELMÄ

Oulun ammattikorkeakoulu
Tietotekniikan tutkinto-ohjelma, laite- ja tuotesuunnittelu

Tekijä: Joni Väisänen
Opinnäytetyön nimi: Testiautomaation jatkokehittäminen
Työn ohjaaja: Veijo Väisänen OAMK, Pasi Heikkinen CGI Suomi Oy
Työn valmistumislukukausi ja -vuosi: Kevät 2019
Sivumäärä: 41 + 1 liite

Opinnäytetyön aiheena oli jatkokehittää testiautomaatio järjestelmää CGI Suomi Oy:lle. Testiautomaatio oli tarkoitus saada päivittäiseen käyttöön. Työ piti sisällään uusien testitapauksien luonnin, luotettavuuden parantamisen ja dokumentoinnin CGI:lle.

Testiautomaation avulla pystytään automatisoimaan testausprosessi, joka vapauttaa työntekijöiltä tunteja muuhun testaukseen. Testaaminen on tärkeä varmistettaessa tuotteen laatua. Testiautomaation avulla saadaan välitön palaute, jos testattavassa kohteessa on jokin vika.

Testit tehtiin Ranorex-ohjelmistolla, jonka ohjelmointikielenä toimii C#. Testiajosta huolehtii Jenkins-sovellus, jonka avulla määritellään, mitkä testeistä ajetaan milloinkin.

Opinnäytetyössä saavutettiin useita sille asetettuja tavoitteita. Päivittäiset testiajot pyörivät useassa ympäristössä, luotettavuuden laatua saatiin nostettua myös ja toimintaohjeita saatiin tehtyä työntekijöille, jotka työskentelevät testiautomaation parissa.

Opinnäytetyötä pystytään hyödyntämään testiautomaation aloittamisessa ja kehittämisessä. Työssä käydään tärkeimpiä osa-alueita, jotka vaikuttavat testiautomaation onnistumiseen. Myös testitapauksien automatisointia on mietitty opinnäytetyössä.

Asiasanat: testiautomaatio, kehitys, C#, testaus

ABSTRACT

Oulu University of Applied Sciences
Information Technology, Option of Equipment and Product Design

---

Author: Joni Väisänen
Title of thesis: Test automation upgrade
Supervisor: Veijo Väisänen OAMK, Pasi Heikkinen CGI Suomi Oy
Term and year when the thesis was submitted: Spring 2019
Number of pages: 41+1

The aim of this thesis was to upgrade test automation for CGI Suomi. Goal was to get test automation run daily, add more test cases and improve test result reliability. Thesis also contained documenting work process for CGI.

With test automation it is possible to automate test process which spares tiime to do other testing. Testing is important when verifying software quality. With help of test automation, we can get fast feedback if software contains bugs.

Test cases are done with Ranorex automation software which uses C Sharp (C#) language. The company use Jenkins to determine when test cases are run and which ones.

In the end of thesis, test automation was working daily in multiple environments, the quality of test cases increased, and we made new directive for working with test automation.

This thesis can be utilized when working or starting work with test automation.

Keywords: Test automation, testing, C#, Ranorex

ALKULAUSE

Tämän opinnäytetyön on tehty CGI:lle. Yrityksen edustajana toimi Pasi Heikkinen ja ohjaavana opettajana lehtori Veijo Väisänen. Iso kiitos ohjaajille valvojille, joitten opastuksella sain nostettua opinnäytetyön sisällön tasoa.

Oulussa 4.3.2019
Joni Väisänen

SANASTO

API (Application Programming Interface) - Sovelluksen rajapinta

CI (Continuous Integration) - Jatkuva integraatio

C# - Microsoftin kehittämä ohjelmointikieli

E2E (End-To-End) - Päästä päähän -testausmuoto

GUI (Graphical User Interface) - Graafinen käyttöliittymä

Testitapaus -Yksittäinen testitapaus

Testiketju - Testikokoelma sisältää useita testitapauksia

UI (User Interface) - Käyttöliittymä

SISÄLLYS

1 JOHDANTO

Opinnäytetyön tavoitteena oli jatkokehittää olemassa olevaa testiautomaatiota CGI Suomi Oy:lle. Työ on jatkoa opinnäytetyölle nimeltään “Testiautomaatioprosessin luominen”, jonka on kirjoittanut Antti Ristolainen. Pääsin kesätyön ohessa työskentelemään testiautomaation parissa, josta kiinnostus aiheeseen tuli. Testiautomaatio kattaa ison alueen, josta piti rajata osa pois. Opinnäytetyössä keskitytään uusien testien luontiprosessiin, testiautomaation ylläpitoon, testiautomaatiossa käytettyihin ohjelmistoihin, testituloksien analysointiin ja luotettavuuden tarkasteluun.

Opinnäytetyölle asetettiin seuraavia tavoitteita. Testiautomaatio pitää saada päivittäiseen ajoon. Testituloksien pitää olla luotettavia. Myös uusia testejä pitää saada. Jokainen osa-alue on tärkeä testiautomaatiossa. Testien pitää olla aktiivisessa ajossa, että mahdolliset virheet huomataan nopeasti. Tämä myös vaatii sen, että yrityksestä löytyy testitapauksia, mistä löytää virheitä. Myös testiautomaation tuloksiin pitää pystyä luottamaan.

Opinnäytetyössä käydään läpi, minkälaisia testitapauksia kannattaa automatisoida. Myös uusien testitapauksien luontiprosessiin tutustutaan tarkemmin, mitä kaikkea pitää huomioida uusien testitapauksien luonnin yhteydessä. Työssä tutustutaan myös siihen, mistä testiautomaatio koostuu.

CGI perustettiin vuonna 1976 Quebecissä Kanadassa. Yrityksen perustivat Serge Godin ja André Imbeau. Alussa yritys oli kahden hengen kokoinen. Nykyään CGI työllistää yli 70 000 työntekijää. CGI:llä on satoja toimipisteitä ympäri maailmaa. Vuonna 2016 CGI:n markkinaosuus oli 14 prosenttia ja se oli Suomen toiseksi suurin it-palveluyritys. (CGI 2018.)

2 OHJELMISTON TESTAUS

2.1 Määritelmä

Testaus on laadunvalvontaa, jonka avulla varmistetaan ohjelmiston toimivuus. Testauksen yhteydessä varmistetaan, että ohjelmiston pystyy tekemään asiat, joita siltä odotetaan. Ohjelmiston ei myöskään saa tehdä mitään, mitä sen ei kuuluisikaan tehdä. (ATR Soft 2018.)

Testaamisen voidaan suorittaa kahdella eri tavalla. On olemassa manuaalisesti suoritettava testaus, jonka testaaja suorittaa. Toinen mahdollinen tapa on automaattinen testaaminen, jonka tietokone suorittaa. Kummassakin menetelmässä on omat hyödyt ja haitat, mutta molemmilla on yhteinen tavoite laadunvarmistuksessa. (Bartlett 2018.)

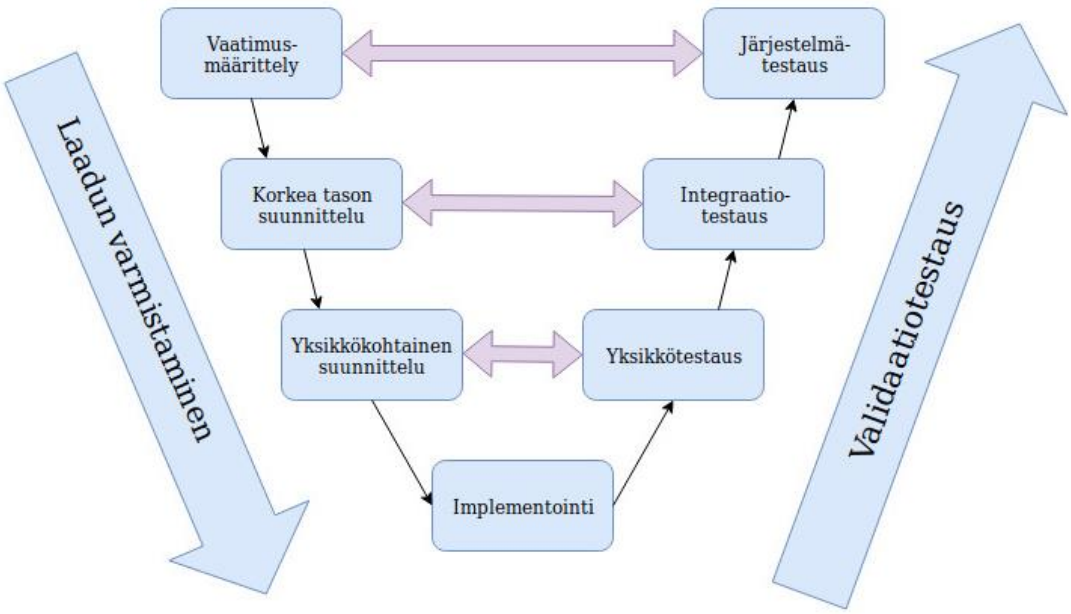
Ohjelmistoa voidaan testata usealla eri tasolla, joita ovat esimerkiksi UI (User Interface) ja API (Application Programming Interface). UI:llä tarkoitetaan testaamista, joka tapahtuu käyttöliittymän kautta. Käyttöliittymä voi myös olla graafinen (GUI eli Graphical User Interface). API:lla testaaminen tapahtuu rajapintojen kautta.

Testaaminen voi olla myös tutkivaa testausta, jossa korostuu testauksen vapaus ja vastuu. Testaajat eivät seuraa valmiita testitapauksia, vaan lähestyvät testattavaa kohdetta tutkivalla asenteella löytää mahdollisia virheitä. Tutkiva testaus koostuu kolmesta osa-alueesta: oppiminen, suunnittelu ja toteutus. (Roberts 2019.)

2.2 Testauksen tasot

Testaus voidaan jakaa karkeasti neljään tasoon, joihin lukeutuu yksikkö-, integraatio-, järjestelmä- ja validaatiotestaus. Tasot eivät nykypäivän ketterässä kehityksessä edusta erillisiä vaiheita, vaan kulkevat monesti rinnakkain projektin läpi. Tasot on jaettu käytännössä testattavan yksikön koon mukaan. (SysTech 2018d.)

Yksi tapa visualisoida testauksen eri vaiheita on 90-luvulla kehitetty V-malli (kuva 1). Nykypäivänä V-mallia ei sovelleta sellaisenaan, mutta sen sisältämät testaamisen tasot ovat kuitenkin yleisesti käytössä alan kielessä. Termejä käytetään myös ristiin, mikä voi aiheuttaa ristiriitoja. (SysTech 2018c; SysTech 2018d.)



KUVA 1. Testauksen V-malli (mukaillen SysTech 2018d)

2.2.1 Yksikkötestaus

Yksikkötestauksella tarkoitetaan pienimmän mahdollisen ohjelman osan, esimerkiksi metodin, toiminnan testaamista. Testeillä varmistetaan, että ohjelman pienimmät osat toimivat oikein, ja ehkäistään niitten aiheuttamat virhetilanteet. Esimerkkinä jos metodi saa parametrina jotain sille kuulumatonta, se ei kaada koko ohjelmaa, eikä metodi saa tehdä mitään sellaista, jota sen ei kuuluisi tehdä. Tämän vuoksi on tärkeää, että yksikkötestejä ajettaessa tarkkaillaan lopputulosta kokonaisvaltaisesti. (SysTech 2018c.)

Yksikkötestauksen hyödyt näkyvät kehitysprosessin aikana, kun kirjoitettuun koodiin tehdään muutoksia. Automatisoiduilla yksikkötesteillä pystytään nopeasti todentamaan muutoksien aiheuttamat

virheet. Monissa ketterissä menetelmissä hyödynnetään testivetoista kehitysmallia, jossa yksikön ohjelmointi aloitetaan kirjoittamalla sille testit ja koodaamalla sen jälkeen. (SysTech 2018c.)

Testitapausten pitää olla itsenäisiä ja testata yhtä asiaa kerralla. Yksikkötestaaminen on aina tarpeellinen jollain tasolla, eikä sitä voi ohittaa kokonaan. Yksikkötestaamisen suorittaa yleensä kehittäjä. (Guru99 2018e.)

2.2.2 Integraatiotestaus

Integraatiotestauksessa testataan useiden komponenttien yhteistoimintaa tavoitteena löytää virheitä. Testeissä suoritetaan tiettyjä suorituspolkuja, jotka hyödyntävät useita eri yksiköitä tai laajempia komponentteja. Projektin edetessä ja yksiköiden valmistuessa testattavien polkujen määrä lisääntyy ja samalle ne pitenevät. (SysTech 2018c.)

Integraatiotestauksessa testataan eri ohjelmisto-osia yhdessä. Tyypillinen ohjelmistoprojekti koostuu useista osista, jotka voivat olla eri kehittäjän kehittämiä. Integraatiotestauksessa keskitytään datan liikkumiseen ohjelmisto-osien välillä. (Guru99 2018b.)

Jatkuva integrointi eli Continuous Integration (CI) on toimintatapa ohjelmistokehityksessä, missä kehittäjä päivittää versiohallintaa useita kertoja päivässä. Jokaisen päivityksen yhteydessä suoritetaan käännös ohjelmistolle. Käännöksille on useasti olemassa automaattisia testitapauksia, jonka avulla varmistetaan, ettei uudet ominaisuudet hajota olemassa olevaa ohjelmistoa. (Nair 2018.)

2.2.3 Järjestelmätestaus

Järjestelmätestauksessa testataan sovellusta kokonaisvaltaisesti, ja tarkastellaan, täyttääkö ohjelma sille asetetut vaatimukset ja käyttötarkoitukset. Testausta voidaan pitää ohjelmiston tulikokeena, jossa sitä käytetään siinä käyttötarkoituksessa ja ympäristössä, johon se on suunniteltu. (SysTech 2018c.)

Tässä vaiheessa on hyvä keskittyä ohjelmiston hienosäätöön ja toimintojen varmistamiseen. Jos kehityksen aikaisemmissa vaiheissa ei ole kiinnitetty tarpeeksi huomiota testaamisen ja virheiden

ehkäisyyn, se huomataan viimeistään tässä vaiheessa. Virheiden korjaus järjestelmätestausvaiheessa yhteydessä on aikavievää ja kallista. (SysTech 2018c.)

Muutoksia ja korjauksia tehtäessä on olennaista, että ohjelmistoa voidaan testata kattavasti automatisoitujen eritasoisten testien avulla. Näin voidaan varmistaa olennaisten toiminnallisuuksien toiminta ilman ylenmääräistä manuaalista testausta. (SysTech 2018c.)

Järjestelmätestaus koostuu toiminnallisesta ja ei-toiminnallisesta testauksesta. Toiminnallisella testaamisella varmistetaan, että ohjelmisto toimii toivotulla tavalla ja sille asetetut vaatimukset täyttyvät. Testaus tehdään yleensä lopullisen käyttäjän näkökulmasta ja käytetään käyttöliittymää. (Pesonen 2018b.)

Ei-toiminnallinen testauksessa testataan osa-alueita, jotka eivät ole käyttäjälle näkyvissä. Testaukseen lukeutuu suorituskyky-, tietoturva-, luotettavuus- ja kapasiteettitestausta. Osa-alueitten toimivuudella on merkittävä rooli sovelluksen laatuun. (SysTech 2018c; Pesonen 2018b.)

2.2.4 Validaatiotestaus

Validaatiotesteillä varmistetaan ohjelman toiminta vaatimusten mukaan. Testitapaukset kirjoitetaan yleensä ohjelman vaatimusmäärittelyn pohjalta ja ohjeena toimivat suunnitteluvaiheessa luodut käyttötapaukset. Automatisointia kannattaa mahdollisuuksien mukaan hyödyntää, sillä kaikkia muutosten yhteydessä syntyneitä virheitä ei välttämättä ole löydetty alemman tason testeillä. (SysTech 2018c.)

2.3 Psykologiset näkökohdat

Testaustavan ja -tapojen valintaan vaikuttaa useita eri tekijöitä käytetystä kehitysmenetelmästä ohjelman alustaan ja kohderyhmään. Testauksessa on aina taustalla tiettyjä psykologisia ilmiöitä, joiden tiedostaminen saattaa vaikuttaa suuresti testauksen tehokkuuteen. (SysTech 2018b.)

Tavoitteiden asettaminen on yksi tärkeimmistä huomioon otettavista psykologisista ilmiöistä. Jos tavoitteeksi asetetaan osoittaa, että ohjelmisto täyttää sille asetetut vaatimukset, testaaja voi jättää kirjoittamatta testitapauksia, jonka avulla voitaisiin löytää vakava virhe. Tällä tavalla testit täyttävät

tavoitteensa, mutta eivät anna haluttua lopputulosta ohjelman toimivuuden osalta. Tämänkaltainen tavoite jättää ulkopuolelle mahdollisuuden, että ohjelma tekee jotain sellaista, mitä se ei saisi tehdä. Tämä voi olla käytön kannalta myös ongelmallista kuin se, että ohjelma ei toimi niin kuin pitäisi. (SysTech 2018b.)

Kehittäjän näkökulmasta on tärkeää, ettei testaamisella ajatella syyllisen etsimistä. Tarkistuksissa ja katselmoinnissa, missä kehittäjän koodia katselmoidaan, on olemassa riski, että kehittäjä kokee asemansa ja ylpeytensä olevan uhattuna. Tämän vuoksi on erittäin tärkeää, että katselmointitilaisuuksissa vallitsee avoin yhdessä tekemisen henki eikä ohjelman virheistä puhuta kehittäjän virheinä. (SysTech 2018b.)

2.4 Testauksen kannattavuus

Ohjelmiston testauksessa on lähes aina kyse resurssien käytön jakamisesta oikein. Raha ja aika riittävät harvoin kokonaisvaltaiseen tavoitteluun. Hyviin testauskäytänteisiin panostamalla pystytään saamaan aikaan luotettava ohjelmisto kohtuullisella panostuksella. (SysTech 2018a.)

Hyvällä testaamisen suunnittelulla ja kattavilla integraatio- ja yksikkötesteillä pystytään rakentamaan koko kehitysprosessin läpi tukena toimiva testikokoelma. Tämän avulla voidaan havaita monet ongelmakohdat aikaisessa vaiheessa, jolloin ne ovat yleensä korjattavissa pienillä kustannuksilla. Tällöin myöhemmässä vaiheessa pystytään keskittymään enemmän ylemmän tason testaukseen ja ohjelmiston hiomiseen. Virhetietokantaa kannattaa myös ylläpitää, sillä sen avulla on helppo seurata syntyvien, korjattavien ja korjattujen virheitten määrää. Se voi myös toimia hyvänä indikaattorina projektin etenemisestä. (SysTech 2018a.)

Mitä aiemmassa projektin vaiheessa virhe havaitaan, sen halvempaa sen korjaaminen on. Virheiden ehkäisyyn painottuvien kehityskäytänteiden, esimerkiksi pariohjelmoinnin, kustannukset voivat tuntua suurilta. Cigital Inc. -yhtiön vuonna 2007 tekemän tutkimuksen mukaan kehityksen aikana löytynyt virhe maksaa keskimäärin yritykselle 977 dollaria, mutta ylläpitovaiheessa havaitun virheen korjaamisen kustannus keskimäärin on 14 102 dollaria. (SysTech 2018a.)

2.5 Manuaalinen testaaminen

Manuaalitestaaminen on prosessi, missä suoritetaan lista erilaisia tehtäviä ja tarkistetaan, että tulokset ovat oikeita. Se on periaatteessa ohjelmiston käyttämistä asiakkaan tavoin, millä voidaan varmistaa toimintojen oikea toimivuus. (Barlett 2017.)

Manuaalitestaaminen vaati paljon työtä, eikä sitä voida täysin ohittaa tai automatisoida. Automaattitestaamisella ei voida kattaa kaikkia osa-alueita. Ohjelmistot tulevat ihmisten käyttöön ja täten olisi tärkeää, että ihmiset osallistuvat myös testaamisen. Manuaalitestaamisella löydetään enemmän käytettävyysvirheitä kuin automatisoiduilla testeillä. Manuaalitestaaminen antaa testaajalle liikkumatilaa testauksen yhteydessä, jota automaatiolla ei saada. (Barlett 2018.)

2.6 Automaattinen testaaminen

Automaattitestaaminen tarkoittaa, että käytetään automaattista työkalua testien ajamiseen. Automaatio testaamisen tavoitteena on vähentää testaajien työkuormaa, mutta tarkoitus ei ole korvata manuaalista testaamista. Automaatiotestit syöttävät sovellukselle ennalta määrättyä syötteitä, jonka kautta automaatio saa sovelluksesta ulos tietoa, jota se vertailee odotettuun tulokseen. (Guru99 2018a.)

On olemassa kahdenlaista automaatiotestaamista: automatisoidut testit ja testiautomaatio. Yleensä ihmiset sekoittavat termit keskenään ja pitävät niitä samoina. Automatisoidulla testauksella viitataan siihen, että automaatiotyökalu ajaa testit. Testiautomaatiolla tarkoitetaan, kun automatisoidaan koko testausprosessi testien käynnistämisestä tuloksien tallentamiseen. (McMeekin 2017.)

2.7 Testitapauksen luominen

Uuden testitapauksen aloittaminen vaatii, että testaaja tutustuu ohjelmiston tai ominaisuuden vaatimuksiin. Kun vaatimukset ovat selvillä, tiedetään, mihin ohjelmiston pitää pystyä ja mikä luokitellaan virheeksi. Tämä on tärkeä vaihe testaamisessa, koska päätavoite on saada ohjelmisto toimivaksi. (Barlett 2018.)

Kun ymmärtää vaatimukset, voi aloittaa testisuunnitelman kirjoittamisen. Hyvä testisuunnitelma kattaa laajan alueen ohjelmistosta ja on selkeä testata. Sen pitää myös olla helposti uudelleen toistettava, jonka avulla muut testaajat pystyvät testaamaan kohteen myös. (Barlett 2018.)

Testisuunnitelman valmistuessa varsinaisen testauksen pystyy aloittamaan. Testauksen aikana olisi hyvä pitää kirjaa tuloksista testauksen yhteydessä. Näin testitulokset ja vaiheet ovat tuoreessa muistissa, mikä helpottaa virheen uudelleen paikantamista. Kehittäjille on tärkeä antaa ohjeet, kuinka virheet saadaan esille. Tämä auttaa kehittäjiä korjaamaan ohjelmistossa esiintyviä virheitä. (Barlett 2018.)

Onnistunut testitapaus on sellainen, joka löytää virheen ohjelmasta, eikä ole sellainen, joka todistaa, ettei virhettä tapahdu. Tämä pieni näkökulmaero saattaa vaikuttaa suuresti testauksen tehokkuuteen. Erityinen riski pyrkimys osoittaa toimivuus virheiden löytämisen sijaan on silloin, kun testataan itse tuotettua koodia ja ohjelmaa. Tämän takia testaaminen ja arviointi tulisi jättää ulkopuoliselle taholle. (Jyväskylän yliopisto 2018b.)

2.8 Regressiotestaus

Regressiotestauksen tarkoitus on varmistaa, ettei uusi muutos ohjelmistossa vaikuta olemassa oleviin ominaisuuksiin. Regressiotestauksella tarkoitetaan olemassa olevien testitapauksien uudelleen testaamista varmistaakseen olemassa olevien toiminnallisuuksien toiminnot. Tämän avulla pystytään varmistamaan, ettei uudella ohjelmistomuutoksella ole sivuvaikutuksia. (Guru99 2018g.)

Testiautomaatiota hyödynnetään paljon regressiotestauksessa. Regressiotestaus on säännöllistä ja aika vievää, minkä vuoksi regressiotestit sopivat hyvin automatisoitavaksi. Testien automatisointi antaa testaajille enemmän aikaa uusien ominaisuuksien testaamiseen. (Guru99 2018g.)

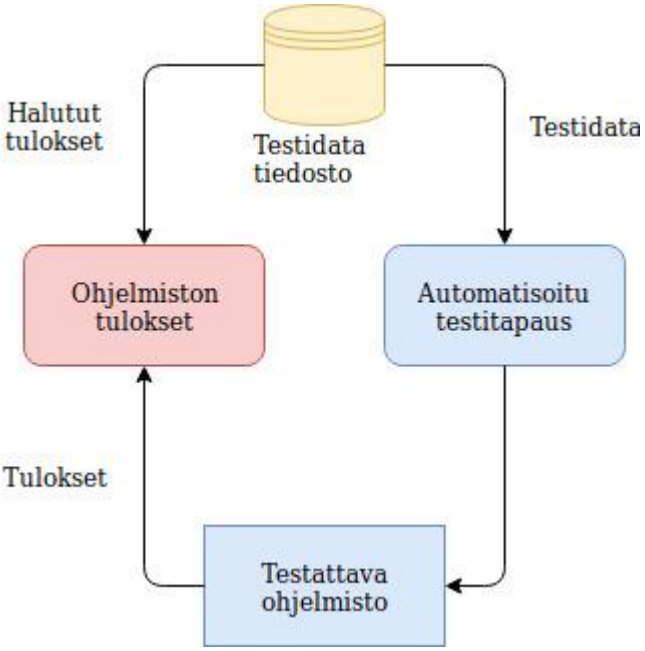
2.9 Smoke-testaus

Smoke-testaus tehdään ohjelmiston käännöksen yhteydessä, jonka avulla varmistetaan tärkeimpien ominaisuuksien toiminnallisuus. Smoke suoritetaan aina ennen, kuin laajempi testaus aloitetaan. Testitapaukset valitaan tärkeitten ominaisuuksien perusteella. (Guru99 2018c.)

Smoke-testauksen voi tehdä manuaalisesti tai automaattisesti. Testiketjun automatisointi yhdistetään yleensä samaan prosessiin, joka suorittaa käännöksen. Tällä tavalla saadaan Smoke ajettua heti käännöksen perään. (Guru99 2018c.)

2.10 Data-Driven-testaus

Data-Driven-testauksella tarkoitetaan saman testitapauksen ajamista useilla eri syötteillä. Testauksessa testaajan haluamat syötteet kirjataan tiedostoon tai tietokantaan (kuva 2). Automatisoitu testitapaus yhdistetään tiedostoon, jonka avulla pystytään ajamaan sama testitapaus käyttämällä jokaista vaihtoehtoa, jolla halutaan testata. (Guru99 2018f.)



KUVA 2. Data-Driven Framework (mukaillen Guru99 2018f)

On olemassa testitapauksia, joita halutaan suorittaa useilla eri syötteillä. Jos jokaiselle tehtäisiin oma testitapaus, se olisi aikaa vievää, pitkästyttävää ja vaikeasti ylläpidettävä, esimerkkinä testitapaus missä halutaan testata ohjelmiston sisäänkirjautumista usealla tunnuksella. (Guru99 2018f.)

Ensimmäinen vaihtoehto voisi olla tehdä jokaiselle tunnukselle oma testitapaus. Toisena vaihtoehtona voidaan vaihtaa tunnus jokaisen ajon jälkeen manuaalisesti. Kolmantena ratkaisuna tallennetaan jokainen käyttäjätunnus tiedostoon, josta testitapaus hakee uuden tunnuksen jokaisen ajon jälkeen. (Guru99 2018f.)

Kahta ensimmäistä vaihtoehtoa ei ole järkevä toteuttaa, koska ne vievät paljon aikaa. Saman testitapauksen ajaminen monta kertaa eri syötteillä voi myös olla pitkäveteistä, minkä ansioista se soveltuu hyvin automatisoitavaksi.

3 TESTIAUTOMAATIO

3.1 Määritelmä

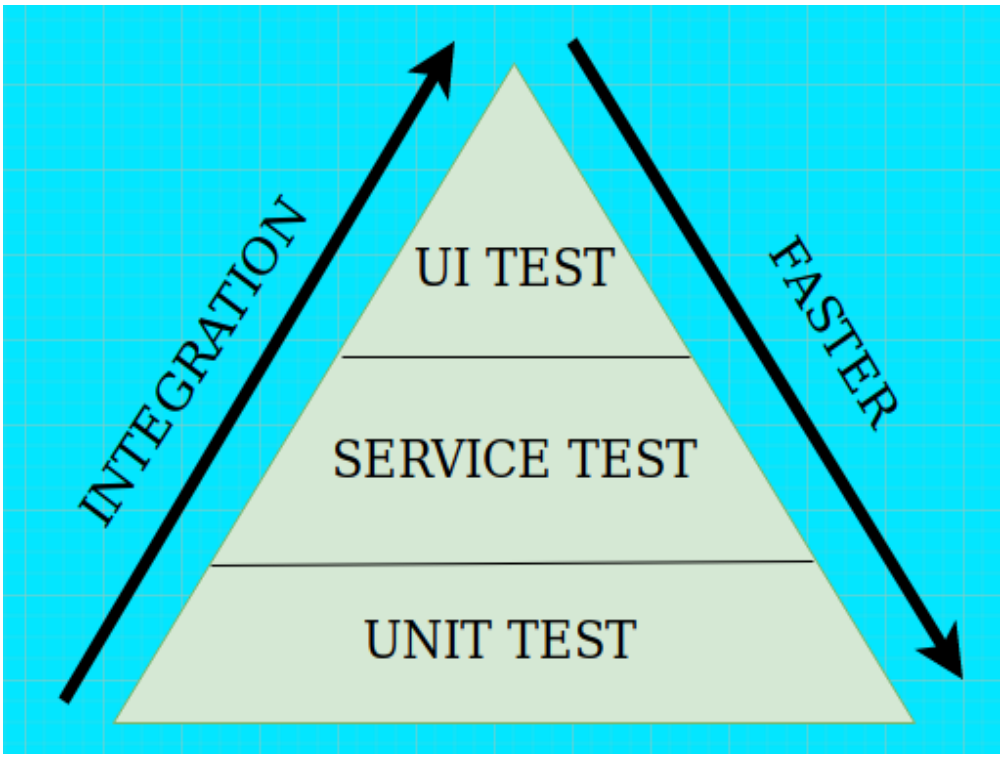
Testiautomaatiolla tarkoittaa, että testaus on automatisoitu täysin. Se osaa itsenäisesti käynnistää testit, ajaa testit ja tallentaa tulokset halutulla tavalla. Hyvin toteutettu testiautomaatio on täysin itsenäinen eikä tarvitse ulkopuolista käyttäjää.

Testiautomaation tarkoitus ei ole korvata manuaalista testaamista vaan tukea sitä. Testiautomaation avulla pystytään automatisoimaan yksinkertaiset ja itseään toistavat testitapaukset, minkä ansioista testaajille jää enemmän aikaa testata vaativampia testitapauksia. (Pesonen 2018b.)

Automatisoinnin avulla testaajille saadaan enemmän aikaa muuhun testaukseen. Jatkuvalla automatisoidulla testaamisella saavutetaan säästöjä pitkällä aikavälillä. Päivittäisellä testiautomaatioajolla saadaan nopeaa palautetta. (Pesonen 2018b.)

Tyypillisiä testitapauksia automatisoitavaksi ovat regressiotestaus ja suorituskykytestaus. Automatisoitu testitapaus on etukäteen suunniteltu. Se on aina suunniteltu jotain tiettyä tilannetta varten, eikä se osaa itse löytää uusia käyttökohteita. (ATR Soft 2018.)

Kuvassa 3 on testiautomaatiossa käytetty pyramidi, jonka on kehittänyt Mike Cohn. Kattava yksikkötestaus luo tukevan pohjan testiautomaatiolle. Yksikkötestauksen jälkeen voidaan siirtyä pyramidin ylemmille tasoille. Mitä ylemmäs pyramidista mennään, sen hitaammaksi ja laajemmiksi testitapaukset menevät. (Cohn 2009.)



KUVA 3. Testiautomaatio-pyramidi (mukaillen Cohn 2009)

3.2 Hyödyt

Automaattiset tarkistukset ovat hyvä tapa varmistaa ohjelmiston toimivuus muutoksien jälkeen. On mahdollista, että lisättyjen ominaisuuksien mukana tulee virheitä, jotka vaikuttavat olemassa oleviin toimintoihin. Automaattisella regressiotestauksella pystytään tarkistamaan nopeasti, vaikuttavatko uudet muutokset negatiivisella tavalla olemassa oleviin muutoksiin. On tärkeää ajaa regressiotestitapauksia joka kerta, kun ohjelmistoa päivitetään. (Gharai 2018a.)

Automaattisella testauksella saadaan nopeaa palautetta ohjelmiston toimivuudesta. Tämä on tärkeää kehittäjille, että he pystyvät korjaamaan virheet ohjelmistosta nopeasti. Nopea palaute saadaan ainoastaan yksikkö- ja API-testauksella. Jos toiminnallisuudet tarkistetaan UI- tai systeemitasolla, kestää kauemmin saada tulokset, mikä on yksi haitta testien automatisoinnissa. (Gharai 2018a.)

Vaikka testien automatisointiin menee aikaa, ne ovat yleensä nopeampia kuin manuaalisesti ajettu. Tämä auttaa saamaan nopeaa palautetta kehittäjille. Tämä on myös tärkeää Data-Driven -testauksessa, jossa suoritetaan samaa testitapausta useasti eri syötteillä. (Gharai 2018a.)

3.3 Haasteet

Automaattiset testitapaukset tarkistavat ainoastaan ne asiat, jotka testiin on ohjelmoitu. Testitapauksen kaikki tarkistukset voivat mennä läpi, mutta ohjelmistossa voi silti löytyä virheitä, koska testitapaukset eivät tarkista näitä kohtia. Ongelman välttämiseksi testitapaukset pitää suunnitella huolellisesti ennen aloittamista. Manuaalitestaus ja tutkiva testaus olisi myös hyvä suorittaa ennen testin automatisointia. (Gharai 2018a.)

Testitapaus voi keskeytyä useaan tekijään, vaikka testattavassa ohjelmistossa ei ole varsinaista virhettä. Testi voi kaatua, jos ympäristö on alhaalla tai testattavaan ohjelmistoon on tullut muutoksia. On tärkeää saada testiautomaatiosta johtuvat virheet pois ajoista, että testeistä saadaan luotettavia tuloksia. (Gharai 2018a.)

Automatisoidut testitapaukset vaativat jatkuvaa ylläpitoa, kun testattavaa ohjelmistoa päivitetään. Jos testitapauksia ei ylläpidetä, testit alkavat hajota, jonka seurauksena testejä ei pysty enää toistamaan. Testiautomaatio ei ole kertaluonteista, vaan se vaatii jatkuvaa ylläpitoa. Ylläpidon helpottamiseksi kannattaa tehdä uudelleen käytettäviä moduuleita, jota kaikki testitapaukset pystyvät käyttämään. (Gharai 2018a.)

3.4 Automatisoitavat testitapaukset

Milloin testitapaus kannattaa automatisoida ja mitä hyötyä siitä saadaan? Useasti automaattitestitapauksen tekemisen yhteydessä keskittyminen siirtyy uuden testitapauksen läpimenoon. Tämän vuoksi osa tarkastuksista saattaa jäädä huomioimatta. Sprintien aikana työntekijöillä on paineita suorittaa tehtävät rajatulla ajalla. Yleensä aikaa ei jää tarpeeksi testatakseen kaikkea tai kirjoittaa testitapauksia uusille ominaisuuksille. (Gharai 2018c.)

Yhtenä syynä voidaan pitää testien kattavuuden parantamista, mutta miten testien kattavuus pystytään määrittelemään? Testiautomaation avulla pystytään ajamaan paljon testitapauksia lyhyeen aikaan. Tämän ansioista pystytään tekemään lisää testitapauksia, jonka avulla pystytään varmistamaan ohjelmiston toimivuus. Testauksessa ja erityisesti automaatio testauksessa isompi testimäärä ei oikeastaan kerro paremmasta laadusta. (Gharai 2018c.)

Ajan säästäminen on myös keskeinen tekijä. Automatisoinnin avulla pystytään keskittymään paremmin tutkivaan testaukseen. Uusia ominaisuuksia ei kannata välttämättä ensimmäisenä automatisoida, koska ne voivat olla vielä työn alla, jonka vuoksi ominaisuuden vaatimukset voivat muuttua. Tämän vuoksi ominaisuudet kannattaa alustavasti testata manuaalisesti. On tärkeä huomioida ajan säästämisessä, että testitapauksien tekeminen vaatii panostusta ja katselmointia, jonka avulla varmistetaan testien laatu. (Gharai 2018c.)

Testiautomaation avulla pystytään saamaan nopeaa palautetta. Nopean palautteen ansiosta nähdään nopeasti, jos ohjelmistoon on päässyt virhe. Automaattitestaaminen itsenään ei paranna laatua, eikä se estä virheitten pääsyä tuotantoon. (Gharai 2018c.)

Testitapauksien toistettavuus on isoin syy, miksi testejä automatisoidaan. Testit automatisoidaan, koska niitä halutaan toistaa uudelleen ja uudelleen. Automatisoitko testiä, jonka ajaisit yhden kerran ja unohtaisit sen? Ajalla minkä käyttäisit testin automatisointiin ehtisit ajaa testin manuaalisesti.

3.5 Testiketjut testiautomaatiolle

Automaattiset regressiotestit ovat keskeinen tekijä testiautomaatio-strategiassa. Regressiotestien avulla varmistetaan, että olemassa olevat toiminnot eivät ole hajonneet uusien muutoksien yhteyksissä.

Smoke-regressiopaketin avulla testataan ohjelmiston tärkeimpien ominaisuuksien toimivuus. Testien tarkoitus on saada nopeasti selville, että tärkeimmät ominaisuudet toimivat ennen laajemman testauksen aloittamista, jonka vuoksi testien kesto tulisi olla lyhyt. Smoke-paketti voidaan ajaa jokaiseen käännökseen ja voi pitää sisällään API- ja GUI-testejä. (Gharai 2018b.)

Funktionaalisen regressiopaketin tarkoitus on testata ohjelmiston osa-alueita tarkemmin kuin Smoke-testit. Ideaalisesti jokaisella tiimillä on regressiotestit omaan osa-alueeseen ohjelmistosta. Paketteja olisi hyvä ajaa useamman kerran päivän aikana ja kestoltaan olisi hyvä olla alle 30 minuuttia. (Gharai 2018b.)

E2E-testien (End-To-End) tarkoituksena on testata erilliset ohjelmiston osat, jotka ovat yhdistetty tietokantoihin ja kolmannen osapuolen sovelluksiin. E2E-regressiopaketti testaa ohjelmistoa kokonaisuudessaan.
Testauksen tarkoitus ei ole testata kaikkia toiminnallisuuksia, koska osa testataan funktionaali-regressiopaketissa. Testeissä testataan siirtymistä yhdestä tilasta toiseen ja tärkeimpiä skenaarioita. Testit tulisi testataan GUI:n kautta mallittaen, kuinka käyttäjä käyttäisi sovellusta. Testit ajetaan yleisesti kerran päivässä tai yössä. (Gharai 2018b.)

3.6 Versionhallinta

3.6.1 Määritelmä

Versiohallinta on ohjelmistotyökalu, jonka avulla kehittäjätiimi hallinnoi koodia. Versiohallinnan avulla pystytään ylläpitämään muutoksia koodissa. Jokaisesta muutoksesta versiohallinta päivittää nykyisen version, jonka avulla on helppo palata taaksepäin virheen sattuessa. Kaikissa ohjelmistoprojekteissa lähdekoodi on tärkeä osa projektia, ja sitä pitää pystyä suojelemaan ja ylläpitämään. Versiohallinta suojaa lähdekoodia merkittäviltä ja inhimillisiltä virheiltä. (Atlassian 2018.)

On mahdollista kehittää ohjelmistoa käyttämättä mitään versiohallintaa, mutta yksikään ammattitaitoinen henkilö ei suosittele tekemään niin. Kysymyksen ei pitäisi olla, käytetäänkö versiohallintaa vaan, mitä versiohallintatyökalua pitäisi käyttää. (Atlassian 2018.)

3.6.2 Hyödyt

Versiohallinnan avulla pystyy ylläpitämään jokaisen tiedoston muutoshistoriaa. Tämä tarkoittaa, että jokainen muutos joka on tehty mihin tahansa tiedostoon on tallennettu. Se pitää sisällään tiedostojen luonnin, poistamisen ja sisällön muuttaminen. Loki pitää sisällään myös muutoksen tekijän, päivämäärän ja muutoksen syyn, jos sellainen on annettu muutoshetkellä. (Atlassian 2018.)

Versioitten haaroittaminen ja yhdistäminen on helpompaa versiohallinnan avulla. Tiimin jäsenet työstävät projektia samanaikaisesti. Versiohallinnan avulla pystytään haaroittamaan kehitys useampaan haaraan. Oman haaran avulla pystytään parantamaan työskentelyä, etteivät toisten kehittäjien keskeneräiset muutokset versiohallinnasta häiritse omaa työskentelyä. Uusille ominaisuuksille tehdään yleensä omat haarat, missä niitä työstetään. Ominaisuuden valmistuessa se on helppo yhdistää takaisin päähaaraan. (Atlassian 2018.)

4 TYÖN ETENEMINEN

4.1 Kehitystyökalut

4.1.1 Jira

Jira on Atlassianin kehittämä tehtävienhallintaohjelma. Nimi tulee japanikielestä sanasta Gojira, joka tarkoittaa Godzilla. Yleisimmät käyttökohteet sovellukselle ovat virheiden ja projektien seurannalle. (Guru99 2019.)

Nykypäivänä ohjelmistoprojekteja tehdään Agile- tai Scrum-menetelmillä. Menetelmissä kehittäjätiimi seuraa suunnitteluja ominaisuuksia uusille julkaisuille. Jiran avulla pystytään seuraamaan tehtäviä ja niiden tilaa. (Guru99 2019.)

4.1.2 C# (CSharp)

C# on olio-ohjelmointikieli, jonka on kehittänyt Microsoft 2000-luvulla. Kieli tehtiin, koska Sun (myöhemmin Oracle) ei halunnut Microsoftin tekevän muutoksia Javaan, jonka vuoksi Microsoft päätti luoda oman kielen. C# on kasvanut paljon sen luontihetkestä, koska Microsoft tarjoaa laajan tuen kielelle. Nykypäivänä C# on yksi yleisimmistä ohjelmointikielistä ympäri maailman. (Mkhitaryan 2017.)

Kieltä käytetään yleensä ohjelmistojen kehitykseen Microsoft-alustoille ja vaatii .NET Frameworkin Windowsilla toimiakseen. C#:n avulla pystytään tekemään mitä tahansa, mutta se on erittäin vahva Windows-alustojen ohjelmistojen kehityksessä. Kielen avulla pystytään myös luomaan selainpohjaisia sovelluksia ja mobiilipuolella suosio on kasvussa. (Mkhitaryan 2017.)

C#:lla on paljon ominaisuuksia, minkä ansioista se on helppo oppia. Se on korkeatasoinen kieli ja helppolukuinen. Monet monimutkaiset tehtävät ovat siirretty pois kehittäjän vastuulta. Esimerkiksi muistinhallinnasta huolehtii .NET. (Mkhitaryan 2017.)

Kieli on tehokas, joustava ja hyvin tuettu, minkä ansioista kieli on yksi suosituimmista ohjelmointikielistä. Tänä päivänä C# on neljänneksi suosituin ohjelmointikieli ja kehittäjistä 31 % käyttää sitä säännöllisesti. Sillä on myös kolmanneksi isoin yhteisö StackOverflowissa, joka on myös rakennettu C#-kielellä. (Mkhitaryan 2017.)

4.1.3 Ranorex

Ranorex on monipuolinen työkalu testien automatisointiin. Se on GUI-testiautomaatiokehys (Framework), jonka avulla pystytään testaamaan selainpohjaisia, työpöytä- ja mobiilisovelluksia. Ranorex käyttää C#:a ja VB.NET:iä, koska sillä ei ole omaa ohjelmointikieltä. Se tukee useita eri tekniikoita, muun muassa Silverlight, .NET, Winforms, Java, SAP, WPF, HTML5, Flash, Flex, Windows Apps, iOS ja Android. (The Economic Times 2018.)

Ranorexilla on hyvä nauhoitusominaisuus, jonka avulla monet pääsevät tutustumaan helposti automatisointiin ilman ohjelmointitaustaa. Testitapauksia pystyy myös kehittämään tuetuilla C#- ja VB.NET-kielellä. (The Economic Times 2018.)

Jokaisen testiajon jälkeen Ranorex luo raportin testituloksista. Raportti pitää sisällään kaiken, mitä testitapauksessa on validoitu. Testitapauksiin voidaan myös lisätä omia kommentteja eri vaiheista, joitten avulla raportista saadaan selkeä kuva, mitä on testattu.

Ranorex on kaupallinen tuote ja sen käyttö vaatii lisenssin. Lisenssit voivat olla kelluvia tai työasemakohtaisia. Lisenssityyppi kannattaa määritellä sen mukaan, miten Ranorex-työkalua käytetään yrityksen sisällä. (Ranorex 2019.)

Validointi

Testitapaukset koostuvat useista validoinneista, joitten avulla verrataan saatua tulosta haluttuun. Ranorexin avulla pystytään suorittamaan validointeja vertailemalla elementtien arvoja, olemassaoloa ja kuvavertailua. Ilman validointia automaattiset testit eivät vertaile testattavaa ohjelmistoa millään tasolla.

Validoineissa olisi hyvä pääsääntöisesti pyrkiä käyttämään elementtiarvojen vertailua eikä kuvien. Testattavan ohjelmiston päivittyessä ohjelmiston ulkoasu voi muuttua, minkä vuoksi kuvavertailu saattaa antaa virheellisiä tuloksia. Kuvia saa näin olla päivittämässä enemmän kuin elementtivertailuja.

Kuvassa 4 on esimerkkiohjelma, jossa testauksen alla on yksinkertainen laskinohjelma. Ensimmäisessä validoinnissa tarkistetaan, että laskin on näkyvillä. Toisessa tarkistetaan, että laskin on tunnistanut oikein painallukset. Testitapaus tulostaa myös raportille tulokset, jotka ovat nähtävillä kuvassa 6.

```
void ITestModule.Run()
{
    Mouse.DefaultMoveTime = 300;
    Keyboard.DefaultKeyPressTime = 100;
    Delay.SpeedFactor = 1.0;

    var repo = testiRepository.Instance;

    //Tarkistetaan, että laskin on näkyvillä
    Validate.Exists(repo.calculator.Self, "Tarkistetaan laskimen näkyvyys");

    //Otetaan talteen painetut napit myöhempää tarkistusta varten
    string validointi = "";

    //Etsitään kaikki nappaimet jotka alkavat 1-9 numeroilla ja painetaan niitä
    IList<Button> lukuNappaimet = Host.Local.Find<Button>(repo.calculator.Self.GetPath() + "//button[@accessibleName~'^[1-9]']");
    for(int i = 0; i < lukuNappaimet.Count;i++)
    {
        //Haetaan napin teksti elementiltä
        string numero = lukuNappaimet[i].GetAttributeValue<string>("accessiblename");
        Report.Info("Painetaan nappia missä lukee: " + numero);
        lukuNappaimet[i].Click();
        validointi += numero;
    }

    //Tarkistetaan näytöltä, että sinne lisättiin kaikki painallukset
    Validate.AttributeEqual(repo.calculator.resultInfo, "accessiblevalue", validointi, "Tarkistetaan, että näytöllä on oikea luku.\r\n Löydetty arvo: {2} Odotettu arvo: {3}");
}
```

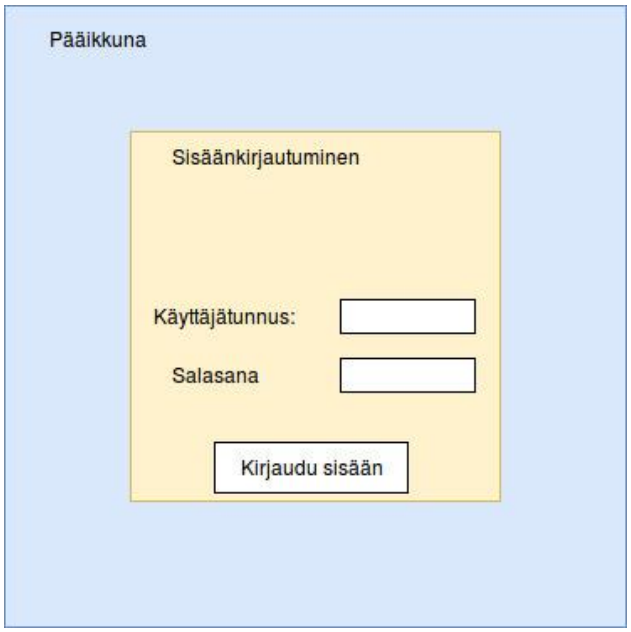
KUVA 4. Validointi-esimerkkejä

UI-Elementit

Ranorex käyttää RanoreXPath-lausekieltä löytääkseen elementit. RanoreXPath on samankaltainen kuin XPath-lausekieli. Ne jakavat saman syntaksin ja loogisen käytöksen. RanoreXPath koostuu aina adapterista, attribuutista ja arvosta. Adapteri luokitellaan elementtityypin mukaan, joka voi esimerkiksi olla tekstikenttä tai painettava nappi. Attribuutilla haetaan haluttua adapteria kyseisillä arvoilla. (Walter 2016.)

Elementin haku koostu kolmesta osasta: **<adapter>[@<attribute>=<value>]**. **<adapter>** on halutun elementin tyyppi, joka voi olla tekstikenttä, painike tai ikkuna. **[@<attribute>=<value>]** avulla pystytään määrittelemään adapterin attribuutti arvot. (Walter 2016.)

Kuvassa 5 on yksinkertaistettu sovelluksen sisäänkirjautumisnäkymä. Kuvasta nähdään, että siinä on kaksi ikkunaa: pääikkuna ja sisäänkirjautuminen. Sisäänkirjautumisella on useampi tekstikenttä ja yksi painike. Kirjaudu sisään -painike voidaan hakea usealla tavalla RanoreXPathin avulla.



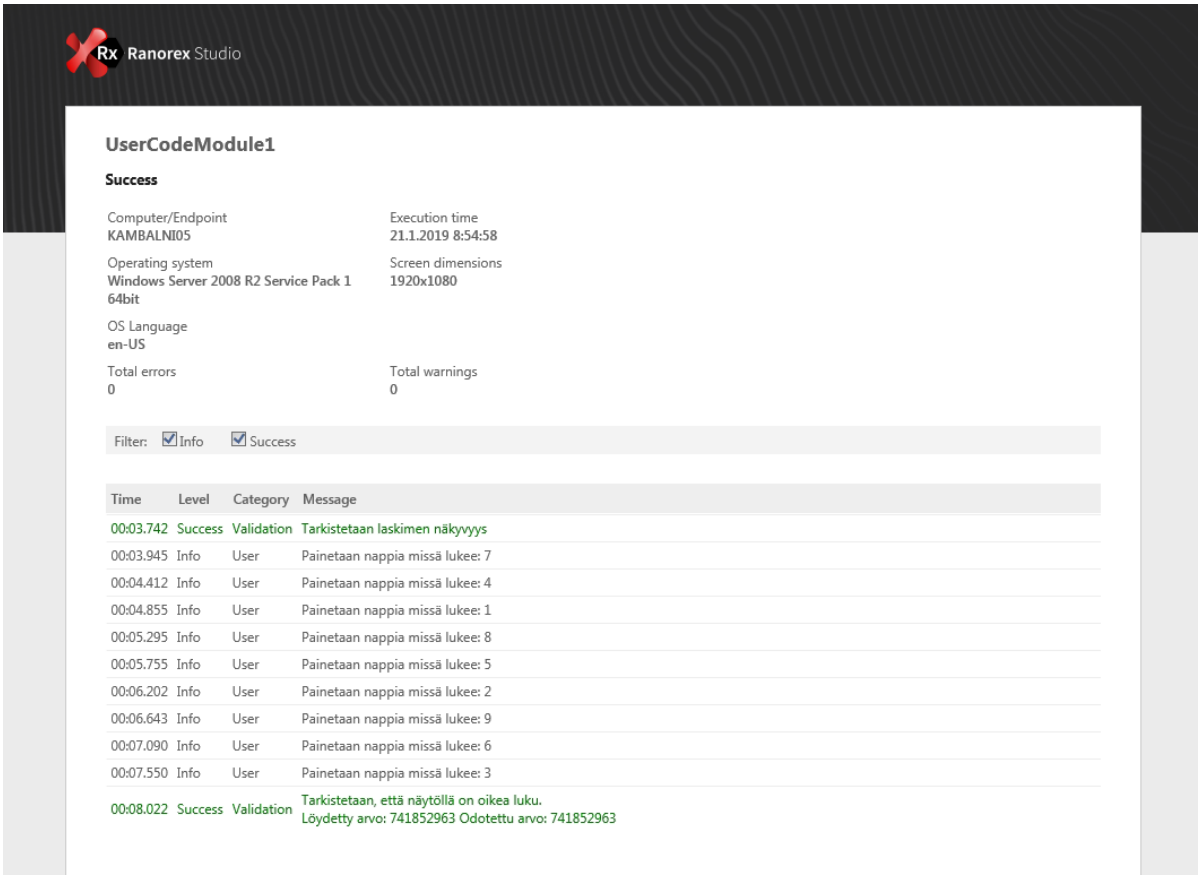
KUVA 5. Sisäänkirjautuminen

Painike pystytään hakemaan koko polulla, mihin lukeutuu myös ikkunat joitten alla painike löytyy: /window[@text='Pääikkuna']/window[@text='Sisäänkirjautuminen']/button[@text=Kirjaudu sisään']. Toinen vaihtoehto on, että haetaan kaikki napit, missä lukee Kirjaudu sisään: //button[@text='Kirjaudu sisään'].

Ensimmäisessä vaihtoehdossa nappi haetaan useamman elementin kautta. Tarkemmilla hauilla pystytään ehkäisemään riskiä löytää väärä elementti. Toisessa vaihtoehdossa haetaan kaikki napit, joissa lukee "Kirjaudu sisään". Ranorexilla pystytään hakemaan löydetyt elementit kahdella eri tavalla. Ensimmäinen vaihtoehto on hakea ensimmäinen, joka sopii polkuun. Toisena vaihtoehtona on hakea kaikki, jotka sopivat polkuun.

Raportointi

Ranorex luo testitapauksen lopussa raportin testitapauksesta. Raportti pitää sisällään kaiken, mitä testitapauksessa on validoitu ja tulostettu. Ranorexin Report- ja Validate-luokkien avulla pystytään luomaan sisältöä raportille. Kuvassa 6 on esimerkki, miltä raportti voi näyttää.



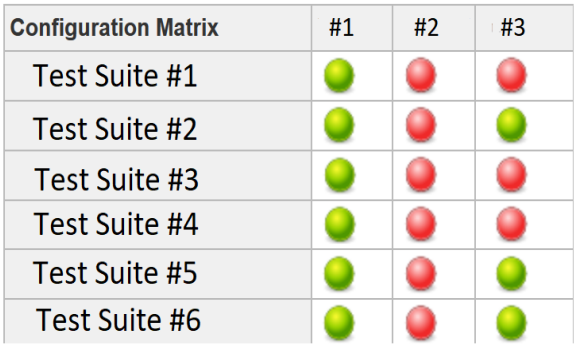
KUVA 6. Raportti-esimerkki

Raportin tuloksia voidaan myös suodattaa useampaan eri luokkaan riippuen, mitä halutaan tulostaa. Liian yksityiskohtainen raportti voi aiheuttaa sekaannusta ulkopuoliselle käyttäjälle, mutta virhetilanteissa tarkemmasta raportista on hyötyä. Kuvassa 6 oleva raportti on peräisin kuvan 4 esimerkistä.

4.1.4 Jenkins

Jenkins on suosittu automaatioserveri. Maailmanlaajuisesti sillä on yli 1,6 miljoonaa käyttäjää. Sen on alun perin kehittänyt Kohsuke Kawaguchi 2004 vuonna. Tänä päivänä Jenkinsiä käytetään kaikkialla pienistä yhden hengen projekteista isoihin yrityksiin. (Cloudbees 2018.)

Kuvassa 7 on esimerkki, miltä Jenkinsin projekti voi näyttää. Kuvassa on kolme eri ympäristöä vaakatasolla. Pystytasolla on kuusi erilaista testiketjua, mitä ajetaan ympäristöjä vasten. Näkymästä nähdään helposti, mitkä testit ovat kaatuneet missäkin testiympäristössä.



KUVA 7. Jenkins-projektin näyttö

Jenkinsin avulla saadaan automatisoitua testien aloitusajankohdat sekä pystyy määrittelemään, mitä testiketjuja ajetaan, missä ajetaan ja milloin ajetaan. Jenkinsin avulla ei tarvita ulkopuolista henkilöä valitsemaan testiketjuja ja käynnistämään testiajoa.

4.2 Lähtötilanne

Opinnäytetyön alussa testiautomaatio oli aikaisessa kehitysvaiheessa. Ranorexilla oli muutama testitapaus valmiina, joita oli ajettu Jenkisissä manuaalisesti. Testiautomaation ympäristöt olivat valmiina, minkä ansioista lähtökohdat kehitystä varten olivat erinomaiset.

Työprosessin alussa määriteltiin tavoitteita tärkeysjärjestykseen, mikä helpottaa työvaiheitten järjestämiseen. Listalla oli muun muassa: saada testit päivittäiseen ajoon, uusien testitapauksien lisääminen, uusien testiympäristöjen lisääminen ja luotettavuuden parantaminen.

4.3 Päivittäiset Jenkins-ajot

Sain Jenkinsin avulla testit pyörimään päivittäin ilman, että niitä tarvitsee itse manuaalisesti käynnistää. Tein Jenkinsissä uuden projektin testiautomaatiolle, jossa pystyin antamaan parametreina tarvittavat tiedot Ranorexin testitapauksiin. Jenkins-projekteille pystytään määrittelemään monenlaisia herätesignaaleja, jotka käynnistävät projektin ja testit.

Asetin herätesignaalit kellonajan mukaan, minkä seurauksena testit lähtevät käyntiin asetetuilla ajankohdilla. Lisäsin kaikki testitapaukset saman projektin alle. Testien pyöriessä automaattisesti pystyin seuramaan pienellä vaivalla olemassa olevien testitapauksien laatua.

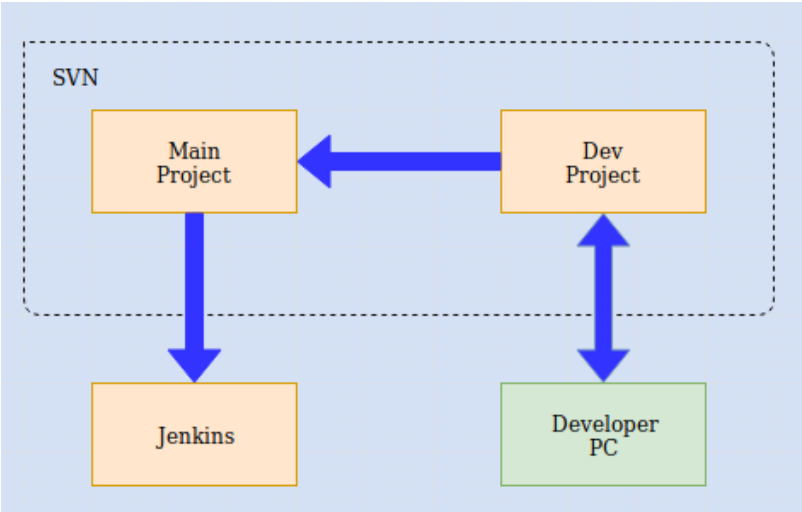
Testien lisääntyessä huomasin, että päivittäisen ajon sekaan oli päässyt myös rikkinäisiä testitapauksia. Rikkinäisellä testitapauksella tarkoitetaan, että testi kaatuu huonosti toteutettuun testiin eikä testattavasta ohjelmistosta löydetyn virheen takia. Testien kasvaessa halusin alkaa erottelemaan testitapauksia toisistaan, minkä avulla pystyttäisiin helpommin erottelemaan rikkinäiset testit toimivista.

4.4 Versiohallinnan jakaminen

Työn alussa meillä oli kehitysympäristö ja Jenkins-ajot oli yhdistetty samaan paikkaan versiohallinnasta. Tämä ratkaisu ei ole toimiva pitemmällä tähtäimellä. Tähän on useita syitä, miksi kehitysympäristö kannattaa pitää erillään testiautomaatioajoista. Jos versiohallinnassa on testin kaatava virhe, se pääsee seuraavan Jenkins-ajoon mukaan ja voi kaataa testit.

Minimoidakseni kyseiset virheet jaoin projektin kahteen osaan: kehitysympäristöön ja tuotantoympäristöön. Tämän tarkoitus on varmistaa, että Jenkinsissä on aina toimivat testitapaukset eivätkä kehitysympäristössä olevat keskeneräiset testitapaukset ja yleisessä käytössä olevat ominaisuudet vaikuta Jenkins-ajoihin.

Kuvassa 8 nähdään, millä tavalla tiedostot liikkuvat testiautomaation sisällä. Kehittäjä hakee kehitysympäristön omalle työasemalle ja työskentelee sen parissa. Uusien muutoksien valmistuessa kehittäjä päivittää muutokset versiohallintaan, jonka kautta ne voidaan siirtää tuotantoympäristöön. Näin Jenkinsillä on oma projekti erotettu kehitysympäristöstä.



KUVA 8. Testiautomaation versiohallinta

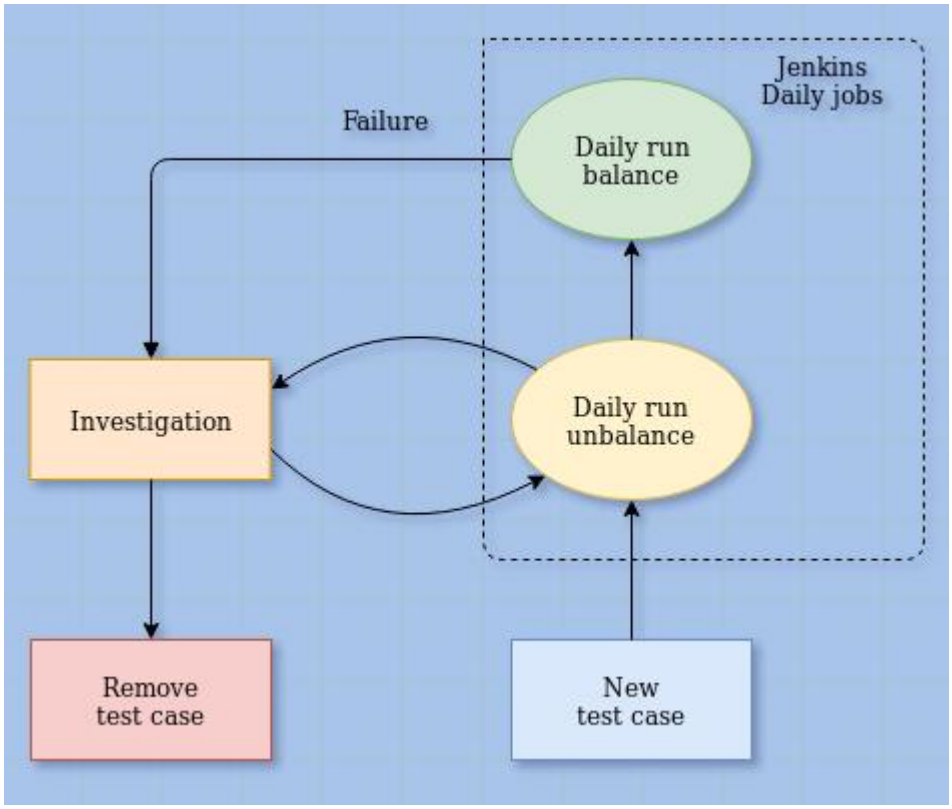
Uusien ominaisuuksien valmistuessa kehitysympäristössä niitten toiminnallisuus tulee tarkastaa, että ne toimivat niin kuin pitää ja eivät vaikuta negatiivisella tavalla olemassa olevaan koodiin. Projektin jakamisella avulla parannettiin testien laatua ja luotettavuutta.

4.5 Testitapauksien luokittelu

Yhtenä tavoitteiden osana oli testien laadun ja luotettavuuden parantaminen. Yhtenä ratkaisuna oli jakaa testitapaukset kahteen ryhmään: toimivat ja rikkinäiset testitapaukset. Toimiviksi testitapauksiksi luokiteltiin ne testit, jotka toimivat oikein kerta toisensa jälkeen. Rikkinäisiksi testit luokitellaan silloin, kun testit kaatuvat säännöllisesti ja tulokset muuttuvat, vaikka ympäristö pysyisi samana.

Rikkinäisillä ja toimivilla testeillä on omat Jenkins-projektit. Omat projektit auttavat seuraamaan rikkinäisten testitapauksien kehittämistä. Pystyin helposti seuramaan alkaako jokin testeistä toimimaan oikein. Silloin voidaan siirtää kyseinen testitapaus toiseen ajoon. Samalla meillä on toimivat testitapaukset omassa ajossa, jonka ansioista muutos testituloksista on helposti huomattavissa.

Kuvassa 9 nähdään visuaalisesti, kuinka testitapaukset voivat liikkua testiautomaation sisällä. Meillä on kaksi projektia, joissa ajetaan testejä päivittäin. Virheen sattuessa testi menee tutkintaa, jossa määritellään virheen syy. Testitapaus voidaan tässä tapauksessa siirtää rikkinäisten testitapauksien ajoon, jossa sitä seurataan tarkemmin. Jos testitapaus on vanha eikä sillä ole mitään arvoa, testitapaus voidaan myös poistaa.



KUVA 9. Testitapauksien liikkuminen testiautomaation sisällä

4.6 Testiympäristöjen lisäys

Yksi testiympäristö ei aina riitä testaamiseen, vaan useasti tarvitaan enemmän kuin yksi. Tähän voi olla monia syitä, esimerkiksi sovellusta halutaan testata eri alustoilla, kuten Linux, Windows ja Mac. Yrityksessä olemme asentaneet sovelluksen eri versioita koneille, joita haluamme testata.

Opinnäytetyössä minun ei tarvinnut lisätä uusia ympäristöjä Jenkinsille, koska ne oli tehty valmiiksi aikaisemmin. Meidän piti ainoastaan saada testattavan sovelluksen asetukset ja ympäristö asetettua testiautomaatiolle sopivaksi.

Samat testitapaukset ajetaan useampaa versiota vasten, jonka ansioista pitää pystyä määrittelemään testitapauksessa testattava versio. Tämän tiedon avulla testiautomaatio osaa ajaa testit oikealla tavalla, koska versioilla voi olla eroavaisuuksia toiminnallisuuksissa. Testiympäristön määrittely voidaan tehdä Jenkinsin avulla. Jenkinsin kautta pystytään antamaan parametreja Ranorexille, jolla pystytään määrittelemään testattavan sovelluksen versio.

4.7 Luotettavuuden parantaminen

Tavoitteena oli saada testitapauksien tuloksien luotettavuutta parannettua. Aikaisessa kehitysvaiheessa testituloksissa oli huomattu eroavaisuuksia, vaikka ympäristö oli pysynyt samana. Testien tuloksiin vaikuttaa testitapauksien laatu, testiympäristö ja testattava ohjelmisto.

Testitapauksien laadun parantamiseksi suunniteltiin useita ratkaisuja. Yhtenä ratkaisuna oli versiohallinnan jakaminen. Jakamisen avulla pystyttiin varmistamaan, etteivät keskeneräiset muutokset pääse vaikuttamaan olemassa oleviin testitapauksiin. Toisena ratkaisuna oli katselmointi. Katselmoinnin toteuttajana tulisi olla eri henkilö ja tulisi varmistaa, että testi toimii niin kuin pitää ja tarkistaa tarvittavat asiat.

Testiympäristö voi myös vaikuttaa testien tuloksiin negatiivisella tavalla. Jos testiympäristössä on yhteysongelmia, voi testattava sovellus toimia hitaasti ja aiheuttaa ongelmia testitapauksille. Testiympäristö voi myös olla alhaalla, jonka seurauksena testit eivät pääse pyörimään ollenkaan ja aiheuttavat automaattisesti virheelliset testitulokset.

5 TULOKSET

5.1 Lopputulos

Viikolla 40 testiautomaatiolle oli automatisoitu 13 erillistä testitapausta. Työn lopussa testiautomaatiolle oli automatisoitu 29 erillistä testitapausta. Aikaisempia testejä päivitettiin myös ajan kanssa kattamaan laajemman kokonaisuuden, jonka vuoksi testitapauksien määrään ei kannata kiinnittää suurta huomiota.

Lisäsin opinnäytetyön edetessä uusia testiympäristöjä, mitä vasten testejä ajetaan päivittäin. Viikolla 40 testitapauksia ajettiin yhteensä 91 kertaa. Viimeisellä viikolla testitapauksia ajettiin yhteensä 502 kertaa. Testitapauksien kokonaisajoon vaikuttaa suuresti, monessako ympäristössä niitä ajetaan samanaikaisesti.

Testien kasvaessa kasvaa myös testiajon pituus. Viikolla 40 testiajo kesti keskimäärin 1 h 55 min, kun taas projektin lopussa keskimääräinen kesto venyi 8 h 48 min. Testitapauksia tuli kaksinkertaisesti lisää ja testiajon kesto kasvoi suunnilleen nelinkertaiseksi. Tässä täytyy myös huomioida, että kaikki testitapaukset eivät ole samanpituisia vaan kesto voi vaihdella 10–60 min.

Alkuvaiheessa testien luotettavuuden kanssa oli ongelmia. Viikolla 40 päiväajoissa meni keskimäärin 52 % testeistä läpi yhdessä ympäristössä. Työn loppuvaiheessa testeistä meni 82 % läpi. Luvut on otettu päivittäisten ajojen tuloksista suoraan, jonka vuoksi niissä voi olla oikeita virheitä ja rikkinäisiä testitapauksia.

Aluksi testiautomaatiolle oli yksi ympäristö valmiina. Projektin lopussa testiautomaatiolle oli käytössä neljä ympäristöä ja jokaisessa oli eri versio, jota testataan. Ympäristöjen lisäys oli nopea tehdä, kun työasemat löytyivät valmiina. Ainoastaan testattavaan ohjelmistoon piti tehdä versiokohtaiset muutokset testiautomaatiota varten.

Ranorexilla oli vaikeuksia tarkistaa yhden elementin olemassaolo. Validointi saattoi kestää useita minuutteja, koska elementti jumitti Ranorex-työkalun. Tämän vuoksi halusimme keksiä toisen ratkaisun elementin olemassaolon tarkistamiseksi. Ongelman ratkaisuksi tein uuden metodin (liite 1),

jonka avulla pystytään paikantamaan elementti kuvankaappauksen avulla. Uuden ratkaisun avulla elementin paikantamiseen meni sekunteja.

5.2 Kehitettävät osa-alueet

Testiautomaatio käyttää neljää työasemaa testien ajoon. Työasemilla on ennalta asennettu versio, jonka perusteella määritellään kone, missä testiketju halutaan ajaa. Tämä rajoittaa testien ajamista eri koneilla. Tulevaisuudessa voitaisiin kehittää, että testiautomaatio osaisi itse poistaa ohjelmiston työasemalta ja asentaa oikean version tilalle ennen testien aloittamista. Tämän ominaisuuden avulla pystyttäisiin hyödyntämään kaikkia työasemia tasapuolisesti.

Projektin edetessä piti parantaa aikaisemmin tehtyjä ominaisuuksia. Lähdin tekemään projektia pienellä osaamisella. Kun aloin ymmärtämään testien rakenteita paremmin, halusin palata aikaisempiin testitapauksiin ja kehittää niitä. Tavoitteena oli saada paljon yhteisiä metodeja, joita kaikki testitapaukset pystyisivät käyttämään. Aikaisessa vaiheessa tähän ei osannut panostaa tarpeeksi.

6 YHTEENVETO

Opinnäytetyön tavoitteena oli kehittää olemassa olevaa testiautomaatiota eteenpäin. Aihe oli mielenkiintoinen ja uusi. Koulussa aihe ei tullut vastaan ja työssä tuli paljon uutta tietoa. Pääsin myös aikaisessa vaiheessa mukaan työstämään testiautomaatiota, minkä ansiosta pääsin osalliseksi päättämään monista asioista.

Työssä tuli paljon uutta tietoa ohjelmiston testaamisesta. Tutuksi tulivat eri testausvaiheet ja niiden vastuuhenkilöt. Myös uusien manuaalisten ja automaattisten testitapauksien luontiprosessi on paljon yksityiskohtaisempi prosessi, kuin osasi aluksi ajatella.

Testiautomaation parissa tuli erittäin paljon uutta asiaa vastaan monelta osa-alueelta. Aikaisemmin olin kuullut versiohallinnan ylläpidosta ja haaroittamisesta. C#-ohjelmointikieli tuli täysin uutena asiana vastaan, mutta C- ja C++-kielen osaamisesta sai hyvän pohjan syntaksin kanssa. Jenkins-ohjelmiston nimen olin aikaisemmin kuullut, mutta projektissa pääsin paremmin sisään, mihin kaikkeen se pystyy.

Kokonaisuudessaan olen tyytyväinen työhön. Kaikkia alussa asetettuja tavoitteita ei täysin saavutettu. Sain tehtyä hyvän pohjan testiautomaatiolle ja se osaa itsenäisesti käynnistää testitapaukset ja tallentaa raportit, niin kuin testiautomaation kuuluu osata. Testiautomaatiota ei voi jättää yksin, vaan se tarvitsee jatkuvaa seurantaa ja ylläpitoa.

LÄHTEET

Atlassian 2018. What is version control. Viitattu 8.11.2018, https://www.atlassian.com/git/tutorials/what-is-version-control.

ATR Soft 2018. Ohjelmistotestaus parantaa laatua. Viitattu 27.10.2018, https://www.atrsoft.com/ohjelmistotestaus-parantaa-laatua/.

Bartlett, J. 2017. Why Manual Testing Is Important. Test Lodge. Viitattu 6.11.2018, https://blog.testlodge.com/why-manual-testing-is-important/.

Bartlett, J. 2018. What is Manual Testing? Test Lodge. Viitattu 1.11.2018, https://blog.testlodge.com/what-is-manual-testing/.

CGI 2018. CGI:n tarina Suomessa. Viitattu 16.10.2018, https://www.cgi.fi/historia-suomessa.

Cloudbees 2018. Jenkins and Cloudbees. Viitattu 27.10.2018, https://www.cloudbees.com/jenkins/jenkins-cloudbees.

Cohn, M. 2009. The Forgotten Layer of Test Automation Pyramid. Mountain Goat Software. Viitattu 11.2.2019, https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid.

Gharai, A. 2018a. Test Automation Advantages and Disadvantages. Testing excellence. Viitattu 21.11.2018, https://www.testingexcellence.com/test-automation-advantages-and-disadvantages/.

Gharai A. 2018b. Test Automation Strategy For Agile Projects. Testing Excellence. Viitattu 21.11.2018, https://www.testingexcellence.com/test-automation-strategy-agile-projects/.

Gharai, A. 2018c. Why Would You Want To Automate a Test? Testing Excellence. Viitattu 21.11.2018, https://www.testingexcellence.com/why-would-you-want-to-automate-a-test/.

Guru99. 2018a. Automation Testing Tutorial: Process, Planning & Tools. Viitattu 1.11.2018, https://www.guru99.com/automation-testing.html.

Guru99. 2018b. INTEGRATION Testing Tutorial: Big Bang, Top Down & Bottom Up. Viitattu 6.11.2018, https://www.guru99.com/integration-testing.html.

Guru99. 2018c. Sanity Testing Vs Smoke Testing: Introduction & Differences. Viitattu 27.10.2018, https://www.guru99.com/smoke-sanity-testing.html.

Guru99. 2018d. Top 15 Automation Testing Interview Questions & Answers. Viitattu 6.11.2018, https://www.guru99.com/automation-testing-interview-questions.html.

Guru99. 2018e. UNIT Testing Tutorial – Learn in 10 Minutes. Viitattu 6.11.2018, https://www.guru99.com/unit-testing-guide.html.

Guru99. 2018f. What is Data Driven Testing? Learn to create Framework. Viitattu 6.11.2018, https://www.guru99.com/data-driven-testing.html.

Guru99. 2018g. What is Regression Testing? Test Cases, Tools & Examples. Viitattu 27.10.2018, https://www.guru99.com/regression-testing.html.

Guru99. 2019. JIRA Tutorial: A Complete Guide for Beginners. Viitattu 13.02.2019, https://www.guru99.com/jira-tutorial-a-complete-guide-for-beginners.html.

McMeekin, K. 2017. Test Automation vs. Automated Testing: The Difference Matters. QASymphony. Viitattu 8.11.2018, https://www.qasymphony.com/blog/test-automation-automated-testing/.

Mkhitaryan, A. 2017. Why Is C# Among The Most Popular Programming Languages in the World? Medium. Viitattu 30.11.2018, https://medium.com/sololearn/why-is-c-among-the-most-popular-programming-languages-in-the-world-ccf26824ffcb.

Nair, J. 2018. Introduction to Continuous Integration Testing. Test lodge. Viitattu 16.12.2018, https://blog.testlodge.com/continuous-integration-testing/.

Pesonen, T. 2018a. Ohjelmistontestaus ja laadunvarmistus. Vala Group. Viitattu 17.12.2018, https://www.valagroup.com/fi/palvelut/ohjelmistotestaus-ja-laadunvarmistus/.

Pesonen, T. 2018b. Testiautomaatio. Vala Group. Viitattu 18.10.2018, https://www.valagroup.com/fi/testiautomaatio/.

Ranorex 2019. Licensing. Viitattu 11.2.2019, https://www.ranorex.com/help/latest/ranorex-studio-system-details/licensing/.

Roberts, N. 2019. What is Exploratory Testing? Global App Testing. Viitattu 7.3.2019, https://www.globalapptesting.com/blog/what-is-exploratory-testing.

SysTech 2018a. Taloudellinen näkökulma. Jyväskylän yliopisto, Informaatioteknologian tiedekunta. Viitattu 7.11.2018, http://smarteducation.jyu.fi/projektit/systech/Periaatteet/suunnittelun-periaatteet/testaus/taloudellinen-nakokulma.

SysTech 2018b. Testauksen psykologia. Jyväskylän yliopisto, Informaatioteknologian tiedekunta Viitattu 7.11.2018, http://smarteducation.jyu.fi/projektit/systech/Periaatteet/suunnittelun-periaatteet/testaus/testauksen-psykologia.

SysTech 2018c. Testauksen tasot. Jyväskylän yliopisto, Informaatioteknologian tiedekunta Viitattu 6.11.2018, http://smarteducation.jyu.fi/projektit/systech/Periaatteet/suunnittelun-periaatteet/testaus/testauksen-tasot.

SysTech 2018d. Testaus lyhyesti. Jyväskylän yliopisto, Informaatioteknologian tiedekunta Viitattu 6.11.2018, http://smarteducation.jyu.fi/projektit/systech/Periaatteet/suunnittelun-periaatteet/testaus/testaus-lyhyesti.

The Economic Times 2018. Definition of ‘Ranorex Tool’. Viitattu 8.11.2018, https://economictimes.indiatimes.com/definition/ranorex-tool.

Walter, T. 2016. RanoreXPath – Tips and Tricks. Ranorex. Viitattu 8.11.2018, https://www.ranorex.com/blog/ranorexpath-tips-and-tricks/.

LIITTEET

LIITE 1 Kuvanvertailu-skripti

```
public Rectangle? checkPicture(Bitmap imgA, Bitmap imgB){
    var imgA_Arr = GetPixelArray(imgA);
    var imgB_Arr = GetPixelArray(imgB);

    int checkUpY = imgA_Arr.Length - imgB_Arr.Length;
    int checkUpX = imgA_Arr[0].Length - imgB_Arr[0].Length;

    for(int y = 0; y < checkUpY; y++){
        for(int x = 0; x < checkUpX; x++){
            bool checkUp = checkNeedle(imgB_Arr[0], imgA_Arr[y], x, 1);

            if(checkUp){
                for(int e = 0; e < imgB_Arr.Length; e++){
                    checkUp = checkNeedle(imgB_Arr[e], imgA_Arr[y + e], x, 0.85);

                    if(!checkUp)
                        break;
                }

                if(!checkUp)
                    continue;

                return new Rectangle(x, y, imgB.Width, imgB.Height);
            }
        }
    }
    return null;
}
```

```
public bool checkNeedle(int[] needle, int[] haystack, int startIndex, double similarity){
    double checkUp = 0;

    for(int i = 0; i < needle.Length; i++)
        checkUp = haystack[i + startIndex] == needle[i] ? checkUp + 1 : checkUp;

    if((checkUp / needle.Length) >= similarity)
        return true;



    return false;
}

private int[][] GetPixelArray(Bitmap bitmap){
 var result = new int[bitmap.Height][];

 var bitmapData = bitmap.LockBits(new Rectangle(0, 0, bitmap.Width, bitmap.Height),
                          ImageLockMode.ReadOnly, PixelFormat.Format32bppArgb);

 for (int y = 0; y < bitmap.Height; ++y){
   result[y] = new int[bitmap.Width];
   Marshal.Copy(bitmapData.Scan0 + y*bitmapData.Stride, result[y], 0, result[y].Length);
 }
 bitmap.UnlockBits(bitmapData);
 return result;
}
```