Testiautomaatio ja testityökalujen valinta ketterässä ohjelmistokehityksessä

Janne Korhonen

Pro gradu - tutkielma

ITÄ-SUOMEN YLIOPISTO

Tietojenkäsittelytieteen laitos

Tietojenkäsittelytiede

Maaliskuu 2018

ITÄ-SUOMEN YLIOPISTO, Luonnontieteiden ja metsätieteiden tiedekunta, Joensuu
Tietojenkäsittelytieteen laitos
Tietojenkäsittelytiede

Opiskelija, Janne Korhonen: Testiautomaatio ja testityökalujen valinta ketterässä ohjelmistokehityksessä
Pro gradu - tutkielma, 70 s
Kandidaatintutkielman ohjaaja: FT Markku Tukiainen, Maaliskuu 2018

Tiivistelmä:
Suurin syy testiautomaation toteuttamiselle on se, että manuaalinen testaaminen vie liikaa aikaa. Etenkin regressiotestit tulisi automatisoida, jotta saadaan nopeaa palautetta mahdollisista virheistä toiminnallisuuksissa. Myös testidatan generointi voidaan automatisoida.

Testiautomaation perustaminen vaatii kuitenkin suuren investoinnin. Ohjelmakoodi olisi hyvä olla testivetoisesti kirjoitettu, ja testiautomaation opettelu vie aikaa, mikäli tiimillä ei ole siitä aikaisempaa kokemusta. Myös varsinaisten testien suunnittelu tulisi olla hyvällä tasolla.

Testipyramidin avulla testit voidaan jakaa kolmeen kerrokseen. Suurin pohjakerros sisältää yksikkö – ja komponenttitestit, jotka ovat tärkeimpiä automatisoida. Toinen kerros sisältää bisnespainotteiset testit. Kolmas kerros sisältää graafisen käyttöliittymän kautta ajettavat testit, jotka ovat hitaita ajaa ja antavat pienimmän investoinnin palautusarvon. Testipyramidi auttaa tutkimaan, miten testiautomaatio voi auttaa ketterää kehitystiimiä.

Automatisoitavat testaustyypit ovat yksikkö – ja komponenttitestit, API – ja web – palvelutestit, graafisen käyttöliittymän ja sen takana ajettavat testit, ja performanssitestit. Esimerkiksi käyttökokemukseen tai graafiseen ulkoasuun liittyviä testejä ei kuitenkaan voida automatisoida.

Testityökalujen valinnassa voidaan harkita itse tehtyjä, avoimen lähdekoodin ja kaupallisia työkaluja, joissa jokaisessa on hyvät ja huonot puolensa. Yksi työkalu ei todennäköisesti sovi kaikkiin tarkoituksiin, joten riittää, että valitaan vain tärkeimpiin tarpeisiin sopiva työkalu mahdollisuuksien mukaan. Testityökalun valinnassa listataan kaikki tarpeet ja kriteerit, ja työkaluja voidaan kokeilla käytännössä. Automatisoitava testaustyyppi vaikuttaa työkalun valintaan.

Avainsanat: Testiautomaatio, ketterä ohjelmistokehitys, testityökalut

UNIVERSITY OF EASTERN FINLAND, Faculty of Science and Forestry, Joensuu
School of Computing
Computer Science

Student, Janne Korhonen: Test automation and test tool selection in agile software development
Master's Thesis, 70 p
Supervisor of the Bachelor's Thesis: PhD Markku Tukiainen, March 2018

Abstract:
The most important reason to implement test automation is because manual testing takes too long. Especially regression tests should be automated to receive feedback about possible bugs in the application's features quickly. Test data generation can also be automized.

However, implementing test automation requires a large investment. It would be better if the source code of the application was written in a test – driven manner. Learning test automation also takes time, if the team does not have any prior experience. Test planning should also be on a good level.

Tests can be divided in three layers as depicted by the test pyramid. The bottom layer includes unit tests and component tests, which are the most important tests to be automated. The second layer consists of business – facing tests. The third layer consists of tests ran through the graphical user interface which are slow and have the smallest return of investment. By using the test pyramid, it is possible to see how test automation can help an agile development team.

Unit tests and component tests, API and web – service tests, tests run through and behind the graphical user interface and performance tests are test types which can be automatized. Tests related to, for example, user experience or graphical style can not be automatized.

When selecting test tools, self – made tools, open source tools and commercial tools can be considered. Each of them has their benefits and pitfalls. Most likely one tool does not fit every requirement, so the tool fitting the most important requirements should be selected. All requirements and criteria should be listed, and test tools can be tried out. The test type to be automatized also affects tool selection.

Keywords: Test automation, agile software development, test tools

Lyhenneluettelo

API: Application Program Interface

TDD: Test – Driven Development

IDE: Integrated Development Environment

Sisällysluettelo

1 Johdanto 1

2 Testiautomaatio ketterässä ohjelmistokehityksessä 3

2.1 Testiautomaation hyödyt 3
2.2 Esteet testiautomaatiolle 9

3 Ketterä testiautomaatiostrategia 15

3.1 Mitä voidaan automatisoida? 20
3.2 Testityypit 23
3.3 Mitä ei tulisi automatisoida? 26
3.4 Mitkä asiat ovat vaikeasti automatisoitavia? 28
3.5 Testiautomaatiostrategian kehittämisen aloitus 29
3.6 Ketterien periaatteiden soveltaminen testiautomaatiossa 36
3.7 Syötedatan muodostaminen 40

4 Testiautomaatiotyökalut 45

4.1 Työkalujen arviointi 45
4.2 Työkalujen valinta 50
4.2.1 Itse tehdyt työkalut 50
4.2.2 Avoimen lähdekoodin työkalut 51
4.2.3 Kaupalliset työkalut 52
4.3 Testityökalujen vertailu – Watir ja Selenium 53
4.4 Oman testityökalun rakentaminen 58

5 Testiautomaation toteutus ja hallinta 63

5.1 Automaation toteuttaminen 63
5.2 Automatisoitujen testien ja testitulosten hallinta ja organisointi 64

6 Yhteenveto 67

Viitteet 70

1 Johdanto

Tämä pro gradu – tutkielma on tehty aiheista testiautomaatio ja testityökalujen valinta ketterässä ohjelmistokehityksessä. Tutkielman tutkimusmenetelmänä on käytetty kirjallisuuskatsausta. Päädyin käyttämään kirjallisuuskatsausta, koska testiautomaatiosta ja testityökalujen valinnasta on saatavilla paljon valmista tutkimusta ja alan ammattilaisten kirjoittamia teoksia. Tutkielmassa käytetty kirjallisuus on pääosin peräisin alan teoksista ja tieteellisistä artikkeleista. Tieteelliset artikkelit löytyivät ACM:n ja IEEE:n digitaalikirjastoista. Pidän löytämiäni lähteitä luotettavina. Käyttämiäni hakusanoja olivat muun muassa ”test automation”, ”test automation agile” ja ”test automation tools”.

Tämän tutkielman tutkimusongelmana on selvittää useita testiautomaatioon ja testityökalujen valintaan liittyviä kysymyksiä. Miten testiautomaatiota tulisi hyödyntää ketterässä ohjelmistokehityksessä? Missä tapauksissa testiautomaatiota kannattaa käyttää, ja missä ei? Millaisia testityyppejä voidaan automatisoida? Miten testiautomaation toteutus ja hallinta tulisi hoitaa? Miten testiautomaatiotyökalut tulisi valita? Onko kannattavaa lähteä rakentamaan omaa testityökalua? Pyrin saamaan vastauksen näihin kysymyksiin ammattilaisten kirjoittamista teoksista ja tieteellisistä artikkeleista. Työskentelen ohjelmistotestaajana, joten kerron myös omia näkemyksiäni tutkielmassa esiteltäviin asioihin.

Tutkielman toinen luku käsittelee testiautomaatiota ketterässä ohjelmistokehityksessä yleisesti, testiautomaation hyötyjä, ja esteitä testiautomaation käyttöönottoon. Kolmas luku käsittelee ketterää testiautomaatiostrategiaa. Tähän kuuluvat muun muassa testauksen eri tyypit, asiat mitä voidaan ja mitä ei voida automatisoida, sekä syötedatan muodostaminen testejä varten. Neljäs luku käsittelee testityökalujen arviointia ja valintaprosessia, sekä esimerkkinä kahden testityökalun vertailuprosessin. Lisäksi mukana on esimerkkitapaus oman testityökalun rakentamisesta ja sen kannattavuudesta. Lopuksi viides luku käsittelee testiautomaation toteutusta ja testien hallintaa.

Crispin ja Gregory toteavat, että testiautomaatio on yksi ketterän toimintatavan tärkeimmistä osista. Ketterät projektit luottavat testiautomaatioon, ja hyvä automaatio jättää ohjelmistotiimille usein aikaa tuottaa korkealaatuista ohjelmakoodia. Se tarjoaa kehyksen, jonka avulla tiimi voi toimia mahdollisimman nopeasti ja silti ylläpitää korkeaa laatua. Lähdekoodin hallinta, automatisoidut koontiversiot ja testiympäristöt, asennus, monitorointi ja erilaiset skriptit ja työkalut vähentävät manuaalista työtä, takaavat luotettavuuden ja sallivat tiimien tehdä jatkuvasti mahdollisimman hyvää työtä (Crispin ja Gregory, 2009).

Collins et al. totesivat yrityksessään, että ketterässä kehityksessä ei ole välttämätöntä määritellä erikseen testaustiimiä ja kehitystiimiä. Testaajat voivat osallistua Scrum – tilaisuuksiin, kuten päivittäisiin tilannekatsauksiin, sprintin suunnittelukokoukseen ja sprintin jälkikatsantoon. Testaajien vastuita ovat tehtävien suunnittelu ja arviointi, toiminnallisuuksien hyväksymiskriteerien määrittely ja testityökalujen käyttö iteraation testauksen nopeuttamiseksi (Collins et al., 2012).

Myös itselläni on kokemusta juuri tämänlaisesta työskentelystä ja työnjaosta. Testaajat kuuluivat omaan laadunvarmistustiimiin, mutta käytännössä testaajat oli jaettu tuotetiimeihin erikseen. Toisin sanoen toimin tuotetiimissä yhdessä toisen testaajan ja ohjelmoijien kanssa. Osallistuimme Scrum – tilaisuuksiin kehittäjien kanssa yhdessä, ja testaajina myös meillä oli vaikutus tehtäviin ratkaisuihin.

Automaatio on hyvin laaja aihe. Siihen kuuluvat tehtävät, kuten yksinkertaisten kuoriskriptien kirjoittaminen, sessio – ominaisuuksien asettaminen ja vakaiden automatisoitujen testien luominen. Automatisointityökalujen määrä on kasvanut eksponentiaalisesti samalla, kun ohjelmistoja opitaan tekemään yhä paremmin (Crispin ja Gregory, 2009).

2 Testiautomaatio ketterässä ohjelmistokehityksessä

Tämä luku käsittelee testiautomaation perusteita ketterässä ohjelmistokehityksessä. Luku 2.1 esittelee testiautomaatiosta koituvia hyötyjä ja luku 2.2 käsittelee esteitä testiautomaation käyttöönottoon.

2.1 Testiautomaation hyödyt

Miksi testejä, koontiprosessia, asennusta ja muita tehtäviä automatisoidaan? Ketterät tiimit pyrkivät siihen, että toimiva ohjelmisto on aina saatavilla, jolloin tuotantovalmis ohjelmisto voidaan julkaista tarpeen vaatiessa milloin tahansa. Tämän tavoitteen saavuttaminen vaatii jatkuvaa testausta. On olemassa useita syitä, miksi automatisoinnin perustaminen on olennaista toimivassa ketterässä kehitysprosessissa (Crispin ja Gregory, 2009).

Collins et al. toteavat, että automatisoidulla ohjelmistotestauksella saavutetaan suurin testikattavuus, luotettavuus ja laatu käytettävissä olevassa ajassa. Manuaaliseen testaamiseen verrattuna automatisoiduilla testeillä on useita etuja. Testien suorittamisella on alhaisempi hinta ja vanhoja testisarjoja voidaan toistaa uusilla ohjelmistoversioilla, jolloin testien regressio on ilmaista. Lisäksi ohjelmiston performanssi – ja stressitestausta voidaan suorittaa pitkään (Collins et al., 2012).

Crispinin ja Gregoryn mukaan yksi pääsyy siihen, miksi tiimit haluavat automatisoida testejä, on yksinkertaisesti se, että tarvittavan testaamistyön manuaalinen suorittaminen vie liikaa aikaa. Kun sovelluksen koko kasvaa, testaamiseen vaadittava aika kasvaa jopa eksponentiaalisesti riippuen testattavan sovelluksen monimutkaisuudesta. Ketterät tiimit pystyvät julkaisemaan tuontantovalmiin ohjelmiston jokaisen lyhyen iteraation lopussa, koska tuotantovalmis ohjelmisto on koko ajan saatavilla. Useiden läpäisevien regressiotestitapausten ajaminen vähintään päivittäin on tarpeellista, eikä siihen pystytä manuaalisella regressiotestauksella (Crispin ja Gregory, 2009).

Regressiotestitapausten manuaalinen ajaminen vie yhä enemmän aikaa jokaisella iteraatiolla. Jotta testaaminen pysyisi samassa aikataulussa ohjelmointityön kanssa,

myös ohjelmoijat joutuvat käyttämään aikaa manuaaliseen regressiotestaamiseen. Vaihtoehtoisesti tiimiin on palkattava lisää testaajia. Lopulta niin tiimin tekninen velka, kuin turhautuminen kasvavat. Jos ohjelmakoodin ei tarvitse edes läpäistä automatisoitua yksikkötason regressiotestiajoa, testaajat joutuvat todennäköisesti käyttämään paljon aikaa tutkimiseen ja yksinkertaisten bugien toistamiseen ja raportointiin. Tällöin testaajille jää vähemmän aikaa mahdollisesti vakavien järjestelmätason virheiden etsimiseen. Mikäli tiimi ei harjoita testivetoista kehitystä, on ohjelmakoodi heikommin testattavissa, eikä se välttämättä tarjoa haluttua toiminnallisuutta (Crispin ja Gregory, 2009).

Lukuisten erilaisten skenaarioiden manuaalinen testaaminen voi viedä paljon aikaa etenkin silloin, kun dataa syötetään käyttöliittymän kautta. Testidatan perustaminen erilaisille monimutkaisille skenaarioille voi olla ylivoimainen tehtävä, jos sen nopeuttamiseksi ei käytetä automatisointia. Lopulta voidaan testata vain muutamia skenaarioita, ja tärkeät virheet jäävät huomaamatta (Crispin ja Gregory, 2009).

Manuaalinen testaus käy itseään toistavaksi etenkin silloin, kun testit tehdään kirjoitettujen testitapausten mukaisesti. Manuaalinen testaaminen käy siis nopeasti tylsäksi ja epämiellyttäväksi työksi. Tämä johtaa siihen, että virheiden tekeminen ja yksinkertaisten virheiden huomaamatta jääminen yleistyvät. Testivaiheita saattaa jäädä kokonaan välistä. Jos tiimillä on tiukka aikataulu, saatetaan kattavasta testauksesta luistaa. Koska manuaalinen testaaminen on hidasta, testaamista voidaan harjoittaa jopa keskiyöllä kehitysiteraation viimeisenä päivänä. Tällöin virheiden huomaaminen heikentyy huomattavasti (Crispin ja Gregory, 2009).

Automatisoitu koonti, julkaisu, versionhallinta ja monitorointi pienentävät riskejä ja tekevät kehitysprosessista johdonmukaisempaa. Testiskriptien automatisointi karsii virheiden mahdollisuutta, koska jokainen testi suoritetaan aina samalla tavalla (Crispin ja Gregory, 2009).

Testivetoisen koodin kirjoittaminen auttaa ohjelmoijia ymmärtämään vaatimuksia ja suunnittelemaan ohjelmistoa sen mukaan. Kiinnostavampaan tutkivaan testaamiseen jää enemmän aikaa, kun koontiversioiden yhteydessä ajetaan jatkuvasti yksikkötestejä ja funktionaalisia regressiotestejä. Tutkivien testitapausten alkuasetusten automatisointi antaa vielä enemmän aikaa tutkia järjestelmän mahdollisesti heikkoja

osia. Kun pitkäveteisten manuaalisten skriptien suorittamiseen ei kulu aikaa, testaaja voi suoriutua paremmin, pohtia eri skenaarioita ja oppia enemmän siitä, miten sovellus toimii (Crispin ja Gregory, 2009).

Mikäli jatkuvasti ajatellaan, kuinka automatisoida testejä jotakin korjausta tai uutta ominaisuutta varten, testattavuus ja laatusuunnittelu nousevat pintaan sen sijaan, että tehtäisiin hataraksi osoittautuva nopea korjaus. Tämä johtaa laadukkaampaan ohjelmakoodiin ja laadukkaampiin testeihin. Testien automatisointi voi auttaa ylläpitämään yhdenmukaisuutta koko sovelluksen osalta. Projektit onnistuvat, kun osaavat tekijät vapautuvat tekemään parhaiten hallitsemaansa työtä. Tämän mahdollistavat automatisoidut regressiotestit, jotka havaitsevat muutokset olemassa olevaan toiminnallisuuteen ja tarjoavat välitöntä palautetta (Crispin ja Gregory, 2009).

Usealle ohjelmistoalalla työskentelevälle on tullut tilanne, jossa vastaan tulee virheen korjaus tai uuden toiminnallisuuden toteuttaminen huonosti suunniteltuun ohjelmakoodiin, jolle ei ole olemassa automatisoituja testejä. Kun ohjelmakoodin automatisoidut regressiotestit ovat tarpeeksi kattavia, syntyy luottamuksen tunne. Vaikka muutos voi saada aikaan odottamattoman vaikutuksen, se havaitaan yksikkötasolla minuuteissa ja funktionaalisella tasolla tunneissa. Muutosten tekeminen testivetoisesti tarkoittaa muuttuneen käyttäytymisen pohtimista ennen koodin kirjoittamista ja verifioivan testin kirjoittamista, mikä lisää luottamusta (Crispin ja Gregory, 2009).

Kun automatisoitua testiympäristöä ei ole toimimassa turvaverkkona, ohjelmoijat voivat sen sijaan alkaa nähdä testaajat turvaverkkona. Ohjelmoija saattaa ajatella, että automatisoidun yksikkötestauksen toteuttaminen on turhaa, koska se toistaisi testaajan joka tapauksessa tekemää manuaalista työtä. Yksikkötestausten automatisoinnin poisto aiheuttaa sen, että testaajat joutuvat kiirehtimään enemmän (Crispin ja Gregory, 2009).

Tiimit, joilla on hyvä testikattavuus automatisoitujen regressiotestien myötä, voivat tehdä muutoksia ohjelmakoodiin vapaammin. Testit kertovat välittömästi, mikäli jokin menee pieleen. Tällaiset tiimit voivat edetä nopeammin kuin tiimit, jotka luottavat pelkästään manuaaliseen testaukseen (Crispin ja Gregory, 2009).

Kun tiettyyn toiminnallisuuteen kuuluva automatisoitu testi menee läpi, täytyy testin mennä jatkossakin läpi, kunnes toiminnallisuutta muutetaan tahallisesti. Kun sovellukseen suunnitellaan muutoksia, testit voidaan muuttaa vastaamaan näitä muutoksia. Jos automatisoitu testi epäonnistuu odottamattomasti, voidaan olettaa, että ohjelmakoodin muutos tuotti regressiovirheen. Automatisoidun testisarjan ajaminen joka kerta, kun uutta ohjelmakoodia lisätään, auttaa varmistamaan, että regressiovirheet saadaan havaittua nopeasti. Koska kirjoitetut muutokset ovat vielä tuoreita ohjelmoijan mielessä, nopea palaute mahdollistaa nopeamman vianetsinnän. Jos virhe löytyisi vasta viikkoja myöhemmin, vian syyn etsintä olisi paljon hitaampaa. Virheet ovat nopeammin korjattavissa, kun testit epäonnistuvat nopeasti (Crispin ja Gregory, 2009).

Automatisoituja testejä suoritetaan säännöllisin väliajoin ja ne toimivat usein muutoksenhavaitsijoina. Ne antavat tiimille tilaisuuden tietää, mikä on muuttunut edellisen koontiversion jälkeen, kuten sen, tuliko koontiversion mukana joitain negatiivisia sivuvaikutuksia. Jos automaatiotestisarja on tarpeeksi kattava, voidaan kauaskantoiset vaikutukset saada helpommin selville verrattuna manuaaliseen testaamiseen. Useimmiten regressiotestejä ei ajeta jokaisella iteraatiolla, saati päivittäin, jos testejä ei ole automatisoitu. Tämä aiheuttaa ongelman esimerkiksi julkaisuvaiheessa, kun tiimin täytyy saada ajettua kaikki regressiotestit. Tällöin ilmaantuu bugeja, jotka olisi voitu löytää jo aikaisemmassa vaiheessa. Useat aikaisesta testaamisesta saatavat hyödyt menetetään (Crispin ja Gregory, 2009).

Testeistä on hyötyä myös varsinaista ohjelmointia vetävänä voimana. Yksikkötestit ja asiakaspohjaiset testit ovat hyödyllisiä etenkin automatisoituina, jolloin ne muodostavat vahvan regressiotestiympäristön. TDD (testivetoinen kehitys) ja SDD (toiminnallisuuspohjainen testivetoinen kehitys) saa tiimit ajattelemaan ratkaisuja testien kautta (Crispin ja Gregory, 2009). Myös Collins et al. toteavat, että ohjelmoijien kokemus tehdä testiautomaatiota komponenttitasolla ja käyttää testivetoista kehitystä estää virheiden ilmaantumisia (Collins et al., 2012). Suunnittelukokouksissa pohditaan, miten testit voidaan toteuttaa parhaalla mahdollisella tavalla. Tiimit suunnittelevat ohjelmakoodin ensisijaisesti niin, että testit menevät läpi, jolloin testattavuudesta ei tule koskaan ongelmaa. Automatisoitu testiympäristö kasvaa lähdekoodin mukana, tarjoten turvaverkon jatkuvaan refaktorointiin. Tällöin on

tärkeää, että koko tiimi harjoittaa testivetoista kehitystä ja kirjoittaa jatkuvasti yksikkötestejä. Muutoin turvaverkkoon jää aukkoja. Tiimille ei myöskään jää niin sanottua teknistä velkaa, ja tiimin työskentelynopeus pysyy vakaana tai jopa kasvaa ajan myötä. Tämä seikka on yksi syy, miksi esimiesten tulisi antaa ohjelmistotiimeille aikaa harjoittaa hyviä käytäntöjä oikealla tavalla (Crispin ja Gregory, 2009).

Myös Collins et al. toteavat, että testiautomaatiota pidetään tärkeänä osana ohjelmistokehitystä, mikä näkyy etenkin eri kehitysmalleissa, kuten TDD:ssä. Testiautomaatio on ketterän testaamisen avaintekijä, ja sen hyötyjä ohjelmistoprojektissa voidaan havaita pienelläkin alkukustannuksella pitkällä aikavälillä. Ketterät menetelmät tulevat edukseen lähinnä siksi, koska niitä harjoitettaessa on kyky mukautua nopeasti alkuperäisten vaatimusten muutoksiin ja priorisoida toiminnallisuuksien kehitys suoritettavan ohjelmakoodin kautta sen sijaan, että kirjoitettaisiin mittavaa dokumentaatiota. Ketterät menetelmät antavat myös mahdollisuuden tehdä yhteistyötä asiakkaan kanssa sen sijaan, että noudetettaisiin kankeita suunnitelmia ja sopimusneuvotteluja. Koska ketterät prosessit ovat iteratiivisia, testauksen täytyy seurata ohjelmistoprojektin iteraatioita. Testaus on myöskin suoritettava tehokkaasti automaattisilla testaustyökaluilla, jolloin saavutetaan nopea palaute vaatimusten muutoksiin ja tehokas vuorovaikutus kehitystiimin kanssa (Collins, Dias-Neto, de Lucena Jr., 2012).

Ketterät ohjelmistokehitystiimit käyttävät testejä ohjaamaan kehitysprosessia. Kun halutun toiminnan kuvaamiseen käytettävät testit automatisoidaan, niistä muodostuu käytännöllinen dokumentaatio järjestelmän tietystä toiminnallisuudesta. On hyvä, jos toiminnallisuuksille on olemassa kertova dokumentaatio, mutta tietyllä syötteellä suoritettavat, selkeän tuloksen näyttävät testit ovat myös tarpeellisia (Crispin ja Gregory, 2009).

Kuten tavanomaisessa dokumentaatiossa, myös testeissä ylläpitäminen on tärkeää. Jos automatisoituja testejä ei päivitetä järjestelmän muuttuessa, testit epäonnistuvat. Tällöin täytyy suorittaa korjaavia toimenpiteitä, jotta koontiprosessi näyttäisi ohjelmiston olevan taas kunnossa. Automatisoidut testit ovat aina tarkka kuva siitä, miten ohjelmakoodi toimii (Crispin ja Gregory, 2009).

Edellä mainitut syyt testiautomaatiolle edustavat testiautomaation hyötyjä ja perusominaisuuksia. Automaatio tuo projekteihin säännöllisyyttä, ja se antaa tiimeille tilaisuuden suorittaa laadunvalvontaa erilaisella tavalla ja rikkoa sovelluksen rajoja. Automaation myötä testaajat saavat enemmän aikaa manuaaliseen testaamiseen ja tiimin jäsenet voivat keskittyä tuottamaan oikeanlainen tuote aikataulun mukaisesti (Crispin ja Gregory, 2009).

Testiautomaation tuomista hyödyistä yksi tärkeä ominaisuus on tapa, miten virheitä korjataan. Manuaaliseen testaukseen nojaavat tiimit tapaavat löytää virheitä kauan sen jälkeen, kun virheen sisältävä ohjelmakoodi kirjoitettiin. Tiimit korjaavat tällöin jatkuvasti yksittäisiä virheitä pureutumatta siihen, mikä virheen alun perin aiheutti tai miten ohjelmakoodin voisi suunnitella paremmin uudelleen. Kun automatisoitu testiympäristö ajetaan ohjelmoijan omassa hiekkalaatikkoympäristössä, automatisoidut regressiotestit löytävät virheet ennen kuin ohjelmakoodi merkataan lopullisesti valmiiksi. Näin ohjelmakoodin hiomiseen jää aikaa, tekninen velka vähenee ja ohjelmakoodista saadaan vakaampaa (Crispin ja Gregory, 2009).

Yksi esimerkki testiautomaation tuomista hyödyistä käytännössä on oma kokemus työskennellessäni testaajana ohjelmistoprojektissa. Projekti oli jatkuvasti kehiteltävä web – palvelin, jota ylläpidettiin Amazonin pilvessä. Usea asiakas käytti palvelua myös paikallisena asennuksena. Kyseessä oli Javan Spring – ohjelmistokehyksellä toteutettu palvelinohjelma, jonka graafinen käyttöliittymä oli tehty käyttämällä GWT:ta (Google Web Tools). Tuotteella oli suhteellisen paljon yksikkötestejä, sillä ohjelmoijat pyrkivät kirjoittamaan testit jokaiselle toiminnallisuuden osalle. Toiminnallisuuksilla oli kuitenkin suurempi testikattavuus varsinaisen testiautomaatiosarjan vuoksi. Testiautomaatio oli toteutettu Python – kielellä käyttäen sen behave – kirjastoa, jolla pystytään kirjoittamaan käyttäytymisvetoista testiautomaatiota. Behave – kirjaston avulla luodaan toiminnallisuutta kuvaavia testitapauksia selkokielisellä Gherkin – kielellä omiin feature – tiedostoihin. Varsinaiset testien toteutukset kirjoitetaan python – tiedostoihin.

Toiminnallisuuksien automaattiset funktionaaliset testit suoritettiin enimmäkseen graafisen käyttöliittymän kautta, mutta myös palvelun julkisen ja sisäisen rajapinnan kautta. Käyttöliittymän kautta suoritettavat testit käyttivät Pythonin Selenium

Webdriver – kirjastoa selaimen automatisointiin. Testiautomaatio oli linkitetty Jenkins – ohjelmaan, joka suoritti jatkuvaa integrointia. Muutokset palvelun lähdekoodissa käynnistivät koontiprosessin, ja koontiversion onnistuessa Jenkins käynnisti automaattisen testisarjan ajon. Koska testit ajettiin jokaisen ison muutossarjan jälkeen, palautetta saatiin nopeasti ja mahdolliset virheet pystyttiin korjaamaan. Testaajien tehtäväksi jäi automatisoinnin ylläpito ja uusien testien kirjoittaminen, sekä tutkiva testaus ja uusien ominaisuuksien manuaalinen verifiointi. Uusien ominaisuuksien perustoiminnallisuuksien testit pyrittiin automatisoimaan mahdollisimman nopeasti niiden valmistuessa, jolloin toistuvimmat, tylsimmät testitapaukset pystyttiin jättämään automaation varaan, mikäli toiminnallisuus ei vaatinut ihmissilmää virheiden havaitsemiseen. Kyseisessä projektissa ainoastaan testaajat olivat tietoisia automaation sisällöstä. Testitapausten selkokielinen esitystapa Gherkin – kielellä mahdollistaisi myös sen, että muut kuin tekniset osapuolet pystyisivät arvioimaan ja kirjoittamaan testitapauksia. Seuraavaksi käsitellään esteitä testiautomaatiolle.

2.2 Esteet testiautomaatiolle

Testiautomaation ongelmista on kirjoittanut aiemmin muun muassa Bret Pettichord vuonna 2001. Nämä ongelmat ovat yhä voimassa, mutta ne koskevat tiimejä, jotka eivät käytä automaatiota kehityksen osana. Crispin ja Gregory esittelevät Bretin tekemän listan ongelmista, joita voi ilmaantua, mikäli automaatiota ei hyödynnetä päivittäin. Jos testiautomaatioon käytetään vain ylimääräinen aika, se ei saa tarpeeksi testiautomaation vaatimaa huomiota. Selkeitä tavoitteita tai kokemusta ei välttämättä ole. Testiautomaation tarve saattaa syntyä tietynlaisesta epätoivon tunteesta, jolloin automaatio voi jäädä pelkäksi toiveeksi realistisen, toteutettavan ehdotuksen sijaan. Automaatiota ei välttämättä ajatella nimenomaan testauslähtöisesti, vaan itse automatisointia pidetään mielekkäämpänä kuin testaamista. Myöskin tietynlaisen teknisen ongelman ratkaiseminen saattaa aiheuttaa sen, että automaation toteuttajalla ei ole enää näkemystä siitä, vastaako tulos varsinaista testaustarvetta (Crispin ja Gregory, 2009).

Crispin ja Gregory esittelevät oman listan esteistä testiautomaation onnistuneeseen toteutukseen. Lista pohjautuu Crispinin ja Gregoryn omiin kokemuksiin ketterissä tiimeissä työskennellessään ja muihin heidän tuntemiensa tiimien kokemuksiin. Näitä

esteitä ovat ohjelmoijien asenne, alkuinvestointi, alati muuttuva ohjelmakoodi, vanhentuneet järjestelmät, pelko ja vanhat tottumukset (Crispin ja Gregory, 2009).

Crispinin ja Gregoryn mukaan perinteisissä, ei-ketterissä työympäristöissä erilliset, näkymättömät laadunvalvontatiimit hoitavat kaiken testaamisen. Ohjelmoijat, jotka ovat tottuneet työskentelemään tällaisissa ympäristöissä, eivät välttämättä osaa huomioida funktionaalisten testien automatisointia. Jotkut ohjelmoijat eivät viitsi testata, koska he pitävät laadunvalvontatiimiä turvaverkkona virheiden nappaamiseen ennen julkaisua. Pitkät vesiputouskehitysjaksot tekevät testaamisesta entistä etäisempää ohjelmoijille. Kun testaajat ovat saaneet työnsä tehtyä, ohjelmoijat ovat jo siirtyneet seuraavan julkaisun pariin. Virheet pinoutuvat myöhemmin korjattaviksi suurella kustannuksella, eikä kenelläkään ole vastuuta niiden tuottamisesta. Jopa testivetoisen kehityksen omaavat ohjelmoijat, jotka ovat tottuneet automatisoimaan testejä yksikkötasolla, eivät välttämättä ajattele sitä, miten hyväksymistestit suoritetaan yksikkötasoa syvemmällä (Crispin ja Gregory, 2009).

Testiautomaation oppiminen on vaikeaa, etenkin kun yritetään oppia miten toteuttaa automaatio tavalla, joka tuottaa riittävän vastineen automaatioon käytettyihin resursseihin nähden. Testiautomaation perustamisen alkuun liittyy tietynlainen oppimiskäyrä, jonka ohjelmoijat ja testaajat joutuvat ylittämään automaation onnistuneen käyttöönoton saavuttamiseksi. Uusien tiimien odotetaan usein ottavansa käyttöön käytäntöjä, kuten testivetoinen kehitys ja refaktorointi. Nämä käytännöt ovat kuitenkin vaikeita oppia. Ilman riittävää ohjausta, aikaa oppia uusia kykyjä ja vahvaa hallintohenkilöstön tukea, nämä käytännöt jäävät helposti oppimatta. Jos tiimeillä on ylimääräisiä oppimiseen vaikuttavia esteitä, kuten työskenteleminen huonosti suunnitellun vanhentuneen koodin parissa, testiautomaation sisäistäminen voi olla mahdotonta (Crispin ja Gregory, 2009).

Tämä oppimiskäyrä voi ilmaantua, kun luodaan palvelinpohjaista testikehystä tai opetellaan käyttämään uutta funktionaalitestityökalua. Tiimi on ylittänyt oppimiskäyrän silloin, kun automaatiosta tulee luonnollinen ja pinttynyt prosessi. Crispin on työskennellyt kolmessa tiimissä, jotka ottivat testivetoisen kehityksen ja funktionaalisen testiautomaation onnistuneesti käyttöön. Jokainen tiimi tarvitsi paljon

aikaa, koulutusta, sitoutumista ja rohkeutta harjoittelun sisäistämiseksi (Crispin ja Gregory, 2009).

Automaatio vaatii suurta investointia, vaikka koko tiimi työskentelisi sen eteen. Käytetty vaiva ei välttämättä tuota tulosta heti. Testikehysten valinta ja päätös siitä, tehdäänkö työkalut sisäisesti vai käytetäänkö ulkoisia työkaluja, vaatii aikaa ja tutkimusta. Myöskin uudelle laitteistolle ja ohjelmistolle voi syntyä tarve (Crispin ja Gregory, 2009).

Crispinin ja Gregoryn mukaan moni on kokenut, että testiautomaatio ei ole tuottanut tulosta. Nämä tahot ovat kenties ostaneet nauhoita – toista – työkalun, antaneet sen laadunvalvontatiimille, ja odottaneet sen ratkovan kaikki automaatio – ongelmat. Tulosten puuttuessa nämä työkalut jäävät käyttämättä. Kyseiset tahot ovat toisaalta voineet tuottaa tuhansia rivejä graafisen käyttöliittymän testiskriptejä, eikä tahoilla ole enää jäljellä ketään työntekijää, joka tietäisi, mitä kyseiset skriptit tekevät. Vaikeasti ylläpidettävät skriptit ovat voineet jäädä hyödyttömiksi (Crispin ja Gregory, 2009).

Gregory kertoo esimerkkitapauksen, jossa hän oli astunut uuteen organisaatioon laadunvalvontapäällikkönä. Yksi hänen tehtävistään oli arvioida nykyisten automatisoitujen testiskriptien tila ja lisätä testien kattavuutta. Organisaatio oli jo aiemmin hankkinut lisensoidun työkalun, mutta alkuperäisen testiympäristön kirjoittaneet testaajat eivät enää työskennelleet kyseisessä organisaatiossa. Yksi uusista testaajista yritti opetella työkalun käyttöä ja hän lisäsi uusia testejä (Crispin ja Gregory, 2009).

Gregory pyysi testaajaa tekemään arvion testiympäristöstä nähdäkseen, mikä sen testikattavuus on. Testaaja käytti viikon ymmärtääkseen, miten testit oli organisoitu. Myös Gregory alkoi tutkia asiaa ja havaitsi, että testit olivat huonosti suunniteltuja ja toivat vain vähän arvoa. Tämän myötä uusien testien lisääminen lopetettiin, ja alettiin pohtia, mikä kyseisen testiautomaation tarkoitus oli. Lisensoitu työkalu ei vastannut tarkoituksen tarpeisiin, joten lisenssiä ei enää jatkettu. Kyseinen organisaatio alkoi sen sijaan käyttää tarkoitukseen sopivaa avoimen lähdekoodin työkalua. Uuden työkalun opettelu vei aikaa, mutta aikaa olisi kulunut joka tapauksessa, koska kukaan tiimistä ei tiennyt miten käyttää alkuperäistä työkalua (Crispin ja Gregory, 2009).

Myös Raulamo-Jurvanen et al ovat huomanneet, että testityökaluihin liittyvät kriteerit ovat osa päätöksentekoprosessia siitä, pitäisikö ohjelmistotestausta automatisoida ylipäätään. Testityökaluihin liittyviin kriteereihin kuuluu päätös käytettävästä työkalusta, työkalun kustannusten korvaaminen, tarkoitukseen sopivan työkalun saatavuus ja työkalun testaamisen positiiviset tulokset (Raulamo-Jurvanen et al, 2017).

Automaation onnistumiseen vaikuttaa suuresti testien suunnittelun taidot. Heikot toimintatavat tuottavat testejä, joita on hankala ymmärtää ja ylläpitää. Ne voivat tuottaa myös aikaa vieviä, vaikeasti tulkittavia tuloksia tai epätosia virheitä. Tiimit, joilla ei ole riittävää koulutusta tai taitoja, saattavat todeta, että automaatioon käytetty investointi ei ole siihen käytettävän ajan arvoista. Sen sijaan hyvät testien suunnittelun toimintatavat tuottavat yksinkertaisia, hyvin suunniteltuja ja jatkuvasti paranneltavia ja ylläpidettäviä testejä. Testimoduulien – ja objektien kirjastot kasaantuvat ajan kuluessa ja tekevät uusien testien automatisoinnista nopeampaa (Crispin ja Gregory, 2009).

Tilastojen puute voi vaikeuttaa testiautomaation käyttöönottoa. Tilastojen tuottaminen ei kuitenkaan ole helppoa. On esimerkiksi mahdotonta vertailla, miten paljon aikaa kuluu automatisoitujen testien kirjoittamiseen ja ylläpitämiseen sen sijaan, että ajettaisiin aina samat regressiotestit manuaalisesti. On myöskin mahdotonta vertailla, miten paljon virheiden korjaaminen maksaa heti niiden syntyessä sen sijaan, että etsii ja korjaa virheet vasta iteraation lopussa. Useat tiimit eivät käytä aikaa kerätäkseen näitä tilastoja. Tiimien on hankala vakuuttaa johtoporras siitä, että investointi automaatioon kannattaa tehdä, koska ei ole lukuja näyttämään, että automatisointi vie vähemmän vaivaa ja tuottaa enemmän arvoa. Tilastojen puute tekee myös tiimeistä vastahakoisempia muuttamaan toimintatapojaan (Crispin ja Gregory, 2009).

Käyttöliittymän kautta ajettavat automatisoidut testit ovat hankalia, koska käyttöliittymä yleensä muuttuu useita kertoja kehityksen aikana. Tämän vuoksi yksinkertaiset nauhoita – toista – tekniikat eivät ole yleensä hyvä valinta ketterään ohjelmistoprojektiin (Crispin ja Gregory, 2009).

Myös API:n tasolla suoritettavia automatisoituja testejä voi olla hankala ylläpitää, jos tiimillä on vaikeuksia saada aikaan hyvä suunnittelu bisneslogiikalle ja tietokantaoperaatioille, ja jos suuria muutoksia joudutaan tekemään usein. Voi olla

hankalaa ja kallista löytää keino testien automatisoimiseksi, jos järjestelmän suunnittelun ohella käytetään vain vähän vaivaa testaamiseen. Ohjelmoijien ja testaajien täytyy tehdä yhteistyötä saadakseen aikaan testattava sovellus (Crispin ja Gregory, 2009).

Vaikka varsinainen ohjelmakoodi, toteutus ja graafinen käyttöliittymä tapaavat usein muuttua ketterän kehityksen aikana, ohjelmakoodin tarkoitus harvoin muuttuu. Kun testikoodi organisoidaan sovelluksen tarkoituksen mukaan, eikä sen implementaation mukaan, testaaja pysyy varsinaisen kehityksen mukana (Crispin ja Gregory, 2009).

Crispinin ja Gregoryn kokemuksen mukaan on paljon helpompaa lähteä toteuttamaan automaatiota, kun kirjoitetaan uutta ohjelmakoodia, jonka arkkitehtuuri on suunniteltu huomioimaan myös testit. Testien kirjoittaminen ohjelmakoodille, jolle ei ole tehty yhtään tai vain vähän testejä, on vaikea tehtävä. Tämä pätee etenkin silloin, kun tiimillä ei ole kokemusta testiautomaatiosta tai ketteristä menetelmistä. Se johtaa monesti ristiriitaisiin tilanteisiin. Jos halutaan automatisoida testit vanhentuneen koodin refaktoroimiseksi ja vanhentunut ohjelmakoodi ei ole suunniteltu helposti testattavaksi, on hankalaa automatisoida testejä edes yksikkötestitasolla. Jos tiimi kohtaa tällaisen haasteen, eikä käytä tarpeeksi aikaa ongelman ylittämiseksi, testien automatisoinnin aloittaminen tehokkaasti on vaikeaa (Crispin ja Gregory, 2009).

Testiautomaatio on pelottava ajatus varsinkin niille, jotka eivät ole koskaan oppineet hallitsemaan sitä kunnolla, mutta myös niille, jotka hallitsevat sen. Ohjelmoijat voivat olla hyviä kirjoittamaan tuotantokoodia, mutta he eivät välttämättä ole kokeneita testien kirjoittajia. Sen sijaan testaajilla ei aina ole vahvaa ohjelmointitaustaa, eivätkä he luota potentiaalisiin testiautomaatiokykyihinsä. Tällaiset testaajat saavat yleensä vaikutelman, että heillä ei ole paikkaa ketterässä ohjelmistokehitysmaailmassa. Crispin ja Gregory uskovat kuitenkin toisin. Yhdenkään testaajan ei pitäisi huolehtia siitä, miten tehdä testiautomaatiota. Automaatio on tiimin ongelma, ja tiimistä yleensä löytyy useita ohjelmoijia, jotka voivat auttaa. Tärkeintä on vastaanottaa uusien ideoiden oppiminen päivä kerrallaan (Crispin ja Gregory, 2009). Olen huomannut, että mikäli testiautomaatiolle löytyy valmista toteutusta, sitä voidaan käyttää apuna testiautomaation oppimiseen ja uusien testien rakentamiseen.

Tiimin jäsenet voivat panikoida, kun iteraatiot eivät edisty tasaisesti ja tiimi ei saa kaikkia ohjelmointi – ja testaustehtäviä valmiiksi iteraation loppuun mennessä. Crispin ja Gregory ovat havainneet, että kun ihmiset alkavat panikoida, he palaavat vanhoihin, turvallisiin tapoihin, vaikka nämä tavat eivät olisi koskaan tuottaneet hyviä tuloksia. Tiimi saattaa todeta, että automatisoiduille testeille ei ole aikaa, mikäli ohjelma halutaan saada valmiiksi julkaisupäivämäärään mennessä. Manuaalista testamista suoritetaan niin paljon, kuin on aikaa, ja toivotaan parasta. Testit automatisoidaan vasta myöhemmin (Crispin ja Gregory, 2009).

Tämä ajattelu johtaa vain huonoihin tuloksiin. Joitain manuaalisia testejä voidaan saada suoritettua, mutta tärkeät, virheitä löytävät tutkivat testit voivat jäädä tekemättä. Nämä virheet voivat maksaa yritykselle paljon rahaa. Jos automatisointitehtäviä ei hoidettu, nämä tehtävät siirtyvät seuraavalle iteraatiolle. Kun iteraatiot etenevät, tilanne pysyy yhä samana (Crispin ja Gregory, 2009).

Ketterä koko tiimin lähestymistapa toimii perustana automaation tuomien haasteiden ylittämisessä. Ohjelmoijat, jotka eivät ole käyttäneet ketteriä menetelmiä, ovat todennäköisesti tottuneet tulemaan palkituiksi virheellisenkin ohjelmakoodin julkaisemista, kunhan projekti on aikataulussa julkaisupäivämäärään nähden. Testivetoinen kehitys on suuntautunut enemmän suunnitteluun kuin testaamiseen, joten bisnespainottuvat testit eivät välttämättä tule ohjelmoijien mieleen. Jotta kaikki saataisiin pohtimaan, miten kirjoittaa, käyttää ja ajaa sekä teknologiapainottuvia, että bisnespainottuvia testejä vaatii johtamiskykyä ja tiimin omistautumista laadulle. Koko tiimin saaminen osallistumaan testiautomaatioon saattaa olla kulttuurinen haaste (Crispin ja Gregory, 2009). Seuraavassa luvussa perehdytään tarkemmin ketterään testiautomaatiostrategiaan ja siihen, mitä ylipäätään voidaan automatisoida.

3 Ketterä testiautomaatiostrategia

Tämä luku käsittelee ketterää testiautomaatiostrategiaa. Luvussa 3.1 pohditaan, mitä voidaan automatisoida, luku 3.2 esittelee testauksen eri tyypit, luvussa 3.3 pohditaan, mitä ei tule automatisoida ja luku 3.4 käsittelee vaikeasti automatisoitavia testejä. Luku 3.5 käsittelee testiautomaatiostrategian kehittämisen aloittamista, luku 3.6 ketterien periaatteiden soveltamista testiautomaatiossa ja luku 3.7 syötedatan muodostamista.

Testaajien täytyy määritellä ja analysoida ohjelmistoprojektin testiautomaatiostrategia harkiten, koska tehtävän onnistuminen vaatii työtä testityökalujen tuntemiseksi ja testitulosten verifioimiseksi. Näin voidaan varmistua siitä, että automaation kustannukset ovat pienempiä, kuin pelkkien manuaalisten testien suorittamisessa (Collins et al., 2012).

Perinteinen ohjelmistotestaus keskittyy lähinnä sellaisiin testaustyyppeihin, kuten tutkiva testaus, käytettävyystestaus, hyväksymistestaus, alfa – ja betatestaus, performanssitestaus ja tietoturvatestaus. Funktionaaliset testit kritisoivat tuotetta, mutta eivät juurikaan tue tuotteen luomista millään tavalla. Lisäksi testausaktiviteetit sijoittuvat kehitysprosessin jälkimmäiseen vaiheeseen, jolloin virheiden estämisen sijaan keskitytään virheiden löytämiseen (Collins et al., 2012).

Ketterässä testaamisessa tiimi paikantaa ja myös estää virheitä. Ketterä testaus on haaste perinteisissä vesiputousprojekteissa työskenneille. Tämä johtuu siitä, että heidän ei tarvitse odottaa järjestelmän toimitusta projektin loppuvaiheessa aloittaakseen testausaktiviteetit, vaan heidän tulee olla oma – aloitteisia ja aloittaa testaustyö projektin alussa yhdessä kehittäjien kanssa (Collins et al., 2012).

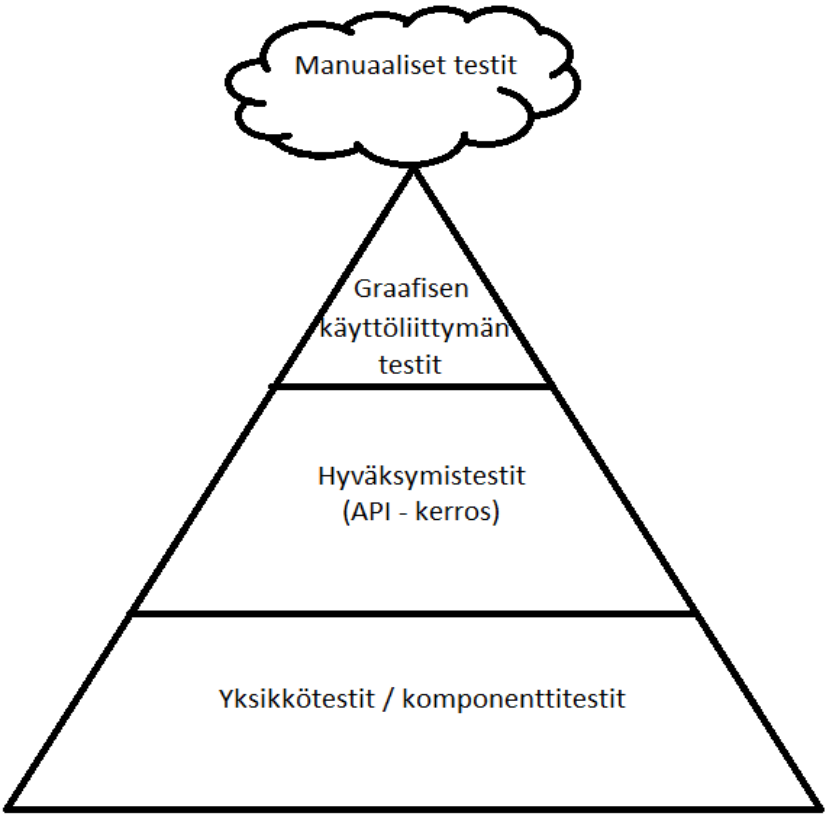
Crispin ja Gregory tarkentavat, että testiautomaatiostrategialle ei ole mitään varmaa keinoa tai vastausta, joka toimisi jokaiselle tiimille. He kehottavat lähestymään automaatioon liittyviä ongelmia samalla tavalla kuin muihinkin ongelmiin. Ensiksi ratkaistava ongelma täytyy määritellä hyvin (Crispin ja Gregory, 2009).

Collins et al. listaavat useita näkökulmia ja opetuksia, mitä testiautomaatiostrategiassa tulisi harkita. Testiautomaatio vaatii, että tuotetta voidaan testata kattavasti.

Ohjelmistokehyksen käyttö tekee testitapausten kehittämisestä helpompaa ja saa automaation kestämään tulevia muutoksia. Testiautomaatio komponenttitasolla vaatii mukautuksia tuotteen arkkitehtuuriin. Tiimin jäsenten pyytäminen lisäämään uusia testejä joka kerta, kun uusi virhe löydetään, vaatii jatkuvaa monitorointia ja muistutuksia. Testiautomaatio vaatii sille omistautuneita resursseja, ja testien suoritusnopeus on olennaista. Testityökaluista tulee helposti tekosyy jokaiseen ongelmaan. Testiautomaation hyötyjä on vaikea saavuttaa projektin ensimmäisen julkaisun aikana, joten se tulisi esitellä pieninä askelina. Collins et al. mukaan testiautomaation vaikein haaste on minimoida ylläpidosta koituvia kustannuksia (Collins et al., 2012).

Crispin ja Gregory esittävät kuvan Mike Cohnin esittelemästä testiautomaatiopyramidista, jonka pohjakerros koostuu teknologiapohjaisista yksikkö – ja komponenttitesteistä (Kuva 1). He tiedostavat, että moni tiimi kamppailee tämän idean kanssa, koska se vaikuttaa päinvastaiselta verrattuna siihen, mitä monissa muissa organisaatioissa ja tiimeissä on käytetty. Useille testitiimeille on opetettu testauksen V – malli, jossa aktiviteetit, kuten komponentti, - järjestelmä – ja julkaisutestaus, tehdään peräkkäin ohjelmointiaktiviteettien jälkeen. Toiset tiimit ovat kääntäneet pyramidin ympäri tehden suurimman osan testeistä funktionaalisessa kerroksessa tai esittelykerroksessa (Crispin ja Gregory, 2009).

Ketterä testiautomaatiopyramidi esittää kolmea eri automatisoitujen testien kerrosta. Pohjimmainen kerros on perusta, joka tukee kaikkia muita kerroksia. Se on koottu lähinnä tiimiä tukevista robusteista yksikkö – ja komponenttitesteistä, eli teknologiapainottuvista testeistä. Kyseinen kerros edustaa automatisoitujen testien ydintä. Testit ovat yleensä kirjoitettu samalla kielellä kuin itse testattava järjestelmä. Kun tiimi hallitsee testivetoisen kehityksen, nämä testit ovat nopeimpia ja halvimpia kirjoittaa. Ne tarjoavat myös nopeimman palautteen, joten ne tuovat paljon arvoa. Pohjimmaisen kerroksen testeillä on suurin investoinnin palautusarvo (Crispin ja Gregory, 2009).



Kuva 1. Testiautomaatiopyramidi, joka koostuu kolmesta päätasosta (Crispin ja Gregory, 2009).

Ketterässä kehityksessä pyritään saamaan mahdollisimman moni testi alimmalle tasolle. Vaikka bisnespohjaiset testit tapaavat mennä yhdelle ylemmistä tasoista, niiden toteuttaminen yksikkötasolla on järkevää. Tämä vaihtoehto on hyvä, jos testit ovat sellaisia, joita asiakkaiden ei tarvitse pystyä lukemaan, ja jotka voidaan ohjelmoida nopeammin kuin yksikkötestit. Toisenlaiset teknologiapainottuvat testit, kuten suorituskykytestit, ovat myös mahdollisia toteutettaviksi yksikkötasolla (Crispin ja Gregory, 2009).

Pyramidin keskitason kerros sisältää suurimman osan tiimin tukemiseksi kirjoitetuista automatisoiduista bisnespohjaisista testeistä. Näihin kuuluvat funktionaaliset testit, jotka varmentavat sen, että projekti valmistuu oikeanlaisesti. Kerroksen testeihin voivat lukeutua myös ominaisuustestit, hyväksymistestit ja testit, jotka kattavat suurempia toiminnallisuuden osia kuin yksikkötestikerros. Nämä testit toimivat API:n tasolla, eli niin sanotusti graafisen käyttöliittymän takana. Toiminnallisuutta testataan suoraan taustasovelluksen kautta sen sijaan, että käytettäisiin graafista käyttöliittymää.

On mahdollista kirjoittaa testitapauksia, jotka asettavat syötteitä tuotantokoodiin, hyväksyvät tulosteet ja vertaavat niitä odotettuihin tuloksiin. Koska nämä testit ohittavat esittelykerroksen, ne ovat halvempia kirjoittaa ja ylläpitää kuin testit, jotka ajetaan graafisen rajapinnan kautta (Crispin ja Gregory, 2009).

Koska nämä testit pyritään kirjoittamaan sellaisella kielellä, jota asiakkaat voivat ymmärtää, ne teettävät enemmän työtä kuin yksikkötason testit. Testejä voidaan yleensä ajaa hitaammin, koska jokaisen testin kattavuus on suurempi kuin yksikkötesteillä ja ne voivat päästä käsiksi tietokantaan tai muihin komponentteihin. Testien tarjoama palaute ei ole yhtä nopeaa kuin yksikkötason testeissä, mutta palaute on silti nopeampaa kuin mitä saavutettaisiin käyttöliittymän kautta. Näiden testien investoinnin palautusarvo ei ole niin korkea kuin testeillä, jotka muodostavat pyramidin pohjan. Investoinnin palautusarvo on kuitenkin suurempi kuin ylimmällä kerroksella. Esimerkkejä pyramidin keskitason työkaluista ovat Fit ja FitNesse (Crispin ja Gregory, 2009).

Pyramidin ylin taso edustaa pienintä automatisointityötä, koska sen testit tuottavat yleensä pienimmän investoinnin palautusarvon. Nämä testit tehdään graafisen käyttöliittymän kautta, joten ne käytännössä toimivat esitystasolla. Testit kirjoitetaan sen jälkeen, kun ohjelmointityö on valmis, joten niiden tarkoitus on yleensä kritisoida tuotetta. Nämä testit ovat osa regressiotestiympäristöä (Crispin ja Gregory, 2009).

Ylimmän tason testit ovat perinteisesti kalliimpia kirjoittaa. Nykyään on kuitenkin saatavilla uusia työkaluja, jotka auttavat vähentämään tarvittavaa investointia. Koska käyttöliittymän komponentit muuttuvat usein, nämä testit ovat hauraampia kuin testit, jotka toimivat toiminnallisella tasolla tai yksikkötasolla. Esimerkiksi pelkästään HTML – elementtien uudelleennimeäminen voi aiheuttaa virheen testiskriptissä. Käyttöliittymän kautta ajettavat testit ovat myös hitaampia verrattuna pyramidin alemman tason testeihin, jotka ajetaan täysin tuotantokoodin päällä. Ylimmän tason testit antavat tärkeää palautetta, mutta graafisen käyttöliittymän testiympäristön ajaminen voi kestää tunteja, kun taas yksikkötason testiympäristöjen ajaminen vie muutamia minuutteja samassa järjestelmässä. Ylimmän tason testien määrää olisi siis minimoitava, jolloin nämä testit muodostaisivat vain pyramidin kärjen (Crispin ja Gregory, 2009). Crispinin ja Gregoryn esittämää ongelmaa pystytään kuitenkin

vähentämään sillä, että graafisen käyttöliittymän kautta ajettavat testit ajetaan osana jatkuvaa integrointia omalla testikoneella, jolloin testien hitaus ei ole haitaksi. Riittävä kommunikointi ja suunnittelu testaajien ja ohjelmoijien välillä taas auttaa varautumaan käyttöliittymän muutoksista aiheutuviin testien muutoksiin.

Vaikka automatisoituja testejä olisi kuinka paljon tahansa, useimmat järjestelmät tarvitsevat myös manuaalisia testaustoimintoja, kuten tutkivaa testausta ja hyväksymistestausta. Näitä kuvaa pyramidin huipulla oleva pilvi. Suurin osa regressiotesteistä pitää automatisoida, tai manuaalinen testaus ei anna hyvää investoinnin palautusarvoa. Patrick Wilson-Welsh on antanut kuvaavan ulottuvuuden testiautomaatiopyramidille eräänlaisella ”kolme pientä porsasta” – metaforalla. Sen mukaan alin perustakerros on tehty tiilestä, koska kerroksen testit ovat varmalla pohjalla. Keskimmäinen kerros on tehty tikuista, koska riittävän vahvoina pysyäkseen testit tarvitsevat enemmän uudelleenjärjestelyä, kuin alemman kerroksen testit. Ylimmän kerroksen testit ovat tehty oljista, sillä testejä on hankala pitää kasassa. Jos oljista on tehty liikaa testejä, vie testien takaisin kuntoon saaminen paljon aikaa (Crispin ja Gregory, 2009).

Useat uudet ketterät tiimit eivät aloita tällaisella pyramidilla, vaan pyramidi on käännetty ympäri. Tämä johtuu kokemuksesta aiempien projektien parissa. Graafisen käyttöliittymän testityökalut ovat usein helpompia opetella, joten tiimit aloittavat usean testinsä ylimmältä ”olkikerrokselta”. Testiautomaation hallitsemiseen liittyvän jyrkän oppimiskäyrän vuoksi tiimi todennäköisesti aloittaa vain muutamalla pohjakerroksen testillä. Keskitason funktionaalisten testien automatisointi on helppo toteuttaa, jos järjestelmä on suunniteltu kyseiset testit mielessä. Tällöin tämän kerroksen testit saattavat kohota pyramidissa korkeammalle kuin pohjakerroksen testit. Kun tiimit alkavat hallita testivetoista kehitystä ja yksikkötestien automatisointia, pohjakerros alkaa kasvaa. Testivetoista kehitystä hyödyntävä tiimi kykenee tällöin rakentamaan testipyramidin pohjakerroksen nopeasti (Crispin ja Gregory, 2009).

Testipyramidin tarkastelu on hyvä keino alkaa tutkimaan, miten testiautomaatio voi auttaa ketterää ohjelmistokehitystiimiä. Ohjelmoijat tapaavat keskittyä pyramidin pohjakerrokseen, ja he tarvitsevat paljon aikaa ja koulutusta ylittääkseen jyrkän

oppimiskäyrän ja päästäkseen pisteeseen, jossa testivetoinen kehitys tapahtuu luonnollisesti ja nopeasti. Perinteisissä tiimeissä testaajilla ei ole yleensä muita vaihtoehtoja kuin automatisoida testit graafisen käyttöliittymän tasolla. Ketterien tiimien käyttämä koko tiimin lähestymistapa tarkoittaa sitä, että testaajat työskentelevät yhdessä ohjelmoijien kanssa ja auttavat heitä tulemaan paremmiksi testien kirjoittamisessa. Tämä taas vahvistaa pyramidin pohjakerrosta. Koska kehitys on testivetoista, koko tiimin suunnittelu pohjautuu jatkuvasti testattavuuteen ja pyramidi voi kasvaa oikean muotoiseksi (Crispin ja Gregory, 2009).

Ohjelmoijat tekevät testaajien kanssa yhteistyötä automatisoidakseen funktionaalitason testejä täyttäen keskimmäisen kerroksen pyramidista. Esimerkiksi testaaja ja asiakas voivat valmistaa jopa satojen rivien suuruisen taulukon testitapauksista web – palveluun. Ohjelmoijat voivat auttaa löytämään keino automatisoida kyseiset testit. Tiimin eri jäsenet voivat olla erikoistuneet alueisiin, kuten testidatan generoiminen, tai työkalujen käyttämiseen. Kun tiimi työskentelee yhdessä, tietämys näistä alueista leviää tiimin keskuudessa ja tiimi löytää parhaat yhdistelmät työkaluista, testitapauksista ja testidatasta (Crispin ja Gregory, 2009).

Kun ohjelmoijia käytetään löytämään kustannustehokkaita keinoja ylimmän tason graafisen käyttöliittymän testien automatisoimiseksi, voidaan saavuttaa useita hyötyjä. Nämä ponnistelut voivat saada ohjelmoijat ymmärtämään paremmin järjestelmän kokonaiskuvaa, ja testaajat voivat oppia tekemään taipuisampia, vähemmän hataria graafisen käyttöliittymän testejä. Mitä enemmän tiimi kykenee tekemään yhteistyötä ja jakamaan tietoa, sitä vahvemmaksi tiimi, sovellus ja testit tulevat (Crispin ja Gregory, 2009). Seuraavassa luvussa käsitellään, mitä ylipäätään voidaan automatisoida.

3.1 Mitä voidaan automatisoida?

Suurin osa testityypeistä hyötyy automaatiosta. Manuaaliset yksikkötestit eivät vie pitkälle estääkseen regressiovirheitä, koska manuaalisen testiympäristön ajaminen jokaisen koodimuutoksen jälkeen ei ole käytännöllistä. Ohjelmakoodia ei voi myöskään suunnitella testivetoisesti manuaalisten yksikkötestien kautta. Kun ohjelmoijat eivät voi ajaa testejä nopeasti nappia painamalla, he eivät ole tarpeeksi

motivoituneita ajamaan testejä ollenkaan. Eri ohjelmakoodiyksiköiden yhteentoimivuus voidaan testata manuaalisesti, mutta automatisoidut komponenttitestit ovat paljon tehokkaampi turvaverkko (Crispin ja Gregory, 2009).

Manuaalinen kokeileva testaaminen on tehokas keino löytää toiminnallisia virheitä, mutta jos automatisoituja bisnespainotteisia regressiotestejä ei hyödynnetä tarpeeksi, kaikki aika tulee todennäköisesti kulumaan manuaaliseen regressiotestaukseen. Automatisoitujen testien ajamiseksi tarvitaan jonkinlainen automatisoitu ohjelmistokehys, joka sallii ohjelmoijien lähettää, testata ja koota ohjelmakoodia (Crispin ja Gregory, 2009).

Automatisoinnin kohteena voi olla periaatteessa mikä tahansa toistuva tai pitkäveteinen ohjelmistokehitykseen liittyvä tehtävä. Automatisoidun ohjelmiston koontiprosessin perustaminen on tärkeää, eikä automatisoitua testipyramidia voi rakentaa ilman sitä. Tiimi tarvitsee yksikkötason testien välitöntä palautetta pysyäkseen oikealla radalla. Myös testaajat hyötyvät siitä, että he voivat saada kaikki muutokset listaavat tiedotteet automatisoiduista koontiversioista. Näin testaajat tietävät, milloin koontiversio on valmis testattavaksi ilman, että heidän tarvitsee jatkuvasti häiritä ohjelmoijia (Crispin ja Gregory, 2009).

Yksi Crispinin ja Gregoryn esittämä ongelma testaajien näkökulmasta liittyy koontiversioiden myöhäiseen saatavuuteen. On normaalia, että perinteisessä ympäristössä testaajat joutuvat odottamaan pitkään saadakseen vakaan koontiversion. Ketterässä ympäristössä testaajien on taas pysyttävä ohjelmoijien mukana, tai muuten ominaisuudet testataan myöhässä. Jos ohjelmoijat eivät saa palautetta, kuten kehitysehdotuksia tai löytyneitä virheitä, testaajat voivat menettää ohjelmoijien uskottavuuden. Virheitä ei löydetä ennen kuin kehittäjät ovat jo seuraavan ominaisuuden parissa. Virheet kasautuvat ja automaatio kärsii, koska sitä ei saada valmiiksi. Myös kehitysiteraation nopeus hidastuu, koska ominaisuutta ei voida merkata valmiiksi ennen testaamista. Tämä tekee seuraavan iteraation suunnittelemisesta hankalampaa. Julkaisusyklin lopussa ominaisuuden testaaminen tapahtuu liian myöhään, jolloin onnistunut julkaisu voi jäädä toteutumatta (Crispin ja Gregory, 2009).

Automatisoitu koontiprosessi nopeuttaa testaamista ja vähentää virheiden määrä. Jatkuvan integroinnin ja koontikehyksen toteutus voi olla suhteellisen helppo ja nopea tehdä, vaikka se vaatiikin jatkuvaa huolenpitoa. Jotkut tiimit taas voivat kokea enemmän vaikeuksia, kuten sellaiset tiimit, jotka tekevät suuria, monimutkaisia järjestelmiä. Crispin ja Gregory ovat puhuneet tiimien kanssa, joilla koontiaika kestää kaksi tuntia tai enemmän. Tämä tarkoittaa sitä, että ohjelmoijan täytyy odottaa jopa kaksi tuntia ohjelmakoodinsa palautuksen jälkeen saadakseen validoinnin siitä, rikkoiko palautus mitään aiempaa toiminnallisuutta (Crispin ja Gregory, 2009).

Crispinin ja Gregoryn mukaan useat ketterät tiimit pitävät kahdeksasta kymmeneen minuuttiin kestävää koontiaikaa toimimattomana. Jo 15 minuuttia on liian pitkä aika palautteen saamiseksi, koska tällöin palautukset alkavat pinoutua ja testaajien on odotettava pitkä aika saadakseen käsiinsä tuorein koontiversio. Tämä on hankalaa myös kehittäjille. Jos ohjelman koonti kestää kaksi tuntia iteraation tai julkaisusyklin lopulla, tietyn toiminnallisuuden särkyessä täytyisi odottaa kaksi tuntia saadakseen selville, korjasiko ohjelmakoodin muutos virheen vai ei (Crispin ja Gregory, 2009). Itselläni on kokemusta noin 20 minuutin koontiajoista. Koska koonti oli osa jatkuvaa integrointia, ohjelmakoodin muutosten ollessa valmiita koontiprosessi alkoi automaattisesti. Muiden testiaktiviteettien takia en ole kuitenkaan kokenut pitkää koontiaikaa suureksi ongelmaksi. Toisinaan koontiaikojen lyhyestä kestosta olisi kuitenkin hyötyä.

Usein pitkät koontiajat johtuvat joko tietokantayhteyksien käytöstä tai rajapinnan kautta tehtävien testien vuoksi. Jos suurta ohjelmakoodia vasten ajetaan tuhansia testejä, koontia ajavalta tietokoneelta syödään paljon resursseja. Tässä tapauksessa testit voidaan profiloida ja tarkastella, mikä aiheuttaa pullonkaulan. Jos vika on esimerkiksi tietokantayhteyksissä, voidaan oikeaa tietokantaa jäljitellä käyttämällä muistinsisäistä tietokantaa. Koontiprosessia pystytään säätämään niin, että testit hajautetaan useiden koneiden kesken. Eri ohjelmistoilla voidaan parantaa resurssienhallintaa. Lisäksi on mahdollista tuoda eksperttejä tiimin ulkopuolelta lisäavuksi (Crispin ja Gregory, 2009).

Jatkuvan integroinnin ja koontiprosessin nopeuttamiseksi on tärkeää ottaa yksi askel kerrallaan. Muutokset voidaan esitellä yksittäin, jolloin jokainen onnistuminen

kyetään arvioimaan erikseen ja pysytään oikealla radalla. Aluksi voidaan siirtää eniten aikaa vaativat testit ajettaviksi joka yö sen sijaan, että ne ajettaisiin jokaisen koonnin yhteydessä. Nopeasti ajettava jatkuva integrointi ja koontiprosessi tarjoavat suurimman investoinnin palautusarvon kaikista automatisointitöistä. Se on ensimmäinen kohde, jonka jokaisen tiimin tulisi automatisoida. Tämän jälkeen tiimin täytyy löytää keino saada nopeaa palautetta automatisoiduilta testeiltä (Crispin ja Gregory, 2009). Seuraavassa luvussa esitellään erilaisia testityyppejä, jotka voidaan automatisoida.

3.2 Testityypit

Ensimmäinen automatisoitava testityyppi on yksikkö – ja komponenttitestit. Crispinin ja Gregoryn mukaan yksikkötestien automatisoinnin tärkeyttä ei voida korostaa tarpeeksi. Jos ohjelmoijat käyttävät testivetoista kehitystä testien kirjoittamisen keinona, ohjelmoijat eivät pelkästään luo suurta regressiotestiympäristöä, vaan he käyttävät sitä suunnitellakseen korkealaatuista ja kestävää ohjelmakoodia. Jos tiimi ei automatisoi yksikkötestejä, se menettää mahdollisuutensa onnistumiseen pitkällä tähtäimellä. Tämän vuoksi yksikkötason testiautomaatio ja jatkuva integrointi tulisi olla tärkein prioriteetti (Crispin ja Gregory, 2009).

Toinen testityyppi on API – tai web – palvelutestaus. API:n tai web – palvelun testaaminen on helpointa, kun käytetään jonkinlaista automaatiota. Kirjoittajista Gregory on ollut tiimeissä, jotka ovat onnistuneesti käyttäneet helposti kirjoitettavia ja ylläpidettäviä data – vetoisia testejä. Gregoryn tiimi esimerkiksi käytti Ruby – kieltä lukeakseen kaikista syötteiden permutaatioista ja yhdistelmistä muodostetun taulukon ja vertasivat tulosteita odotettuihin tuloksiin toisessa taulukossa. Yksi Gregoryn asiakas käytti Rubyn IRB – ominaisuutta (Interactive Ruby Shell) testatakseen web – palveluja hyväksymistesteissä. Tiimi oli halukas jakamaan skriptinsä asiakastiimin kanssa, mutta testaajat halusivat myös nähdä mitä tapahtuu, kun syötteitä muutetaan lennosta. Tämän mahdollisti testien ajaminen interaktiivisesti semiautomatisoituun tapaan (Crispin ja Gregory, 2009).

Kolmas ryhmä on graafisen käyttöliittymän takana ajettavat testit. Ne ovat helpompia automatisoida, kuin varsinaiset käyttöliittymätestit. Nämä testit ovat vakaita, koska

esityskerrokseen tehtävät muutokset eivät vaikuta testeihin ja ne toimivat vakaammalla bisneslogiikkaohjelmakoodilla. Tämänlaiseen testaukseen tarjottavat työkalut antavat yleensä keinoja testien kirjoittamiseen esittelevässä muodossa käyttämällä taulukkoja. Testitapaukset, jotka saavat tuotantokoodin toimimaan testisyötteillä ja palauttamaan tulokset, voidaan yleensä kirjoittaa nopeasti. Tällä keinolla voidaan kirjoittaa bisnespohjaisia testejä, joita sekä asiakkaat, että kehitystä ajavat ohjelmoijat ymmärtävät (Crispin ja Gregory, 2009). Juuri tähän pohjautuu myös aiemmin esillä ollut BDD eli käyttäytymisvetoinen kehitys.

Neljäs kategoria on varsinainen graafisen käyttöliittymän testaus. Jopa ohuet, ilman mitään bisneslogiikkaa sisältävät graafiset käyttöliittymät tulee testata. Ketterän kehityksen tahti on nopeaa ja uusia toiminnallisuuksia tuotetaan jokaisella iteraatiolla. Tämän vuoksi useimmissa projekteissa tarvitaan automatisoituja regressiotestejä graafisen käyttöliittymän tasolla (Crispin ja Gregory, 2009).

Graafisen käyttöliittymän automatisoimiseksi on tärkeää valita oikeat työkalut. Automatisoitujen skriptien tulisi olla mukautuvia ja helposti ylläpidettäviä. Gregory on käyttänyt onnistuneesti Ruby – ja Watir – kieliä, kun testikehys on toteutettu käyttäen hyviä ohjelmointikäytäntöjä tuotantosovellusten tapaan. Kirjastojen kehittämiseen käytetiin aikaa, jolloin ohjelmakoodia ei tarvinnut toistaa tai työstää uudelleen. Tarvittavat muutokset voitiin tehdä yhdessä paikassa. Myös testien investoinnin palautusarvo kasvoi, kun ohjelmakoodista tehtiin helposti hallittavaa. Tällöin ohjelmoijien tulisi kuitenkin nimetä oliot tai antaa niille tunnusluvut. Jos luotetaan vain järjestelmän generoimiin tunnuslukuihin, testit joudutaan muuttamaan joka kerta, kun uusi olio lisätään sivulle, koska tällöin tunnusluvut muuttuvat (Crispin ja Gregory, 2009).

Graafisen käyttöliittymän testien tulisi varmentaa varsinaista rajapintaa, kuten varmistaa, että napit toimivat ja suorittavat sen, mitä niiden pitäisi tehdä. Näillä testeillä ei tulisi testata varsinaista bisneslogiikkaa. Myös linkkien tarkastuksen testit ovat helppo automatisoida. Tällöin ei ole tarvetta tarkastaa manuaalisesti, että jokaisen sivun jokaisesta linkistä päästään oikealle sivulle. Ensin tulisi automatisoida yksinkertaisemmat asiat, ja vasta myöhemmin käyttöliittymän haasteellisemmat osat (Crispin ja Gregory, 2009).

Myös performanssitestaus on yksi tärkeä testauksen osa – alue. Kaikkia testityyppejä ei voida suorittaa ilman automaatiota. Manuaaliset performanssitestit eivät yleensä ole sopivia tai tarkkoja, vaikka näitä testejä koitetaan suorittaa usein. Performanssitestaus vaatii monitorointityökalujen käyttöä ja keinon ohjata testattavan järjestelmän toimintoja. Performanssitestaukseen voidaan liittää myös tietoturvatestaus. Ilman mitään ohjelmistokehystyökalua ei ole mahdollista tehdä suurta hyökkäystä sen verifioimiseksi, voiko web – sivuston hakkeroida tai pystyykö sivusto käsittelemään suurta datakuormaa (Crispin ja Gregory, 2009).

On helpompaa tarkastaa ASCII – tiedoston tuloste järjestelmäprosessilla visuaalisesti, kun ensin jäsentää tiedoston sisällön ja näyttää sen helposti luettavassa muodossa. Manuaalisen vertailun sijaan on parempi tehdä skripti, joka vertaa tulostetiedostoja keskenään varmistaakseen, ettei mitään tarkoituksettomia muutoksia ole tullut. On olemassa useita tiedoston vertailutyökaluja aina ilmaisista ”diff” – työkaluista maksullisiin työkaluihin, kuten WinDiff. Myös lähdekoodin hallintatyökalut ja IDE:t sisältävät sisäänrakennettuja vertailutyökaluja. Nämä työkalut ovat tärkeitä myös testaajille. Skriptien tekeminen on suositeltavaa myös silloin, kun halutaan vertailla tietokantatauluja testattaessa datavaraston tai datan migraatioprojekteja (Crispin ja Gregory, 2009).

Kun työskennellään asiakkaiden kanssa oppiakseen ymmärtämään heidän liiketoimintaansa paremmin, voidaan nähdä tilaisuuksia automatisoida joitain asiakkaiden tehtäviä. Crispinin yritys tarvitsi lähettää jokaiselle asiakkaalle sähköpostia esikirjeen kanssa useassa muodossa. Ohjelmoijat pystyivät generoimaan pohjat nopeasti ja liittämään ne esikirjeisiin tehden postitustyöstä paljon nopeampaa. Toinen testaaja sen sijaan kirjoitti taulukkomakron, joka suoritti monimutkaisia laskelmia eläköitymisvarojen sijoittamiseksi. Aiemmin hallintohenkilöstö oli tehnyt tätä tehtävää manuaalisesti. Monet manuaaliset listaukset voidaan korvata automatisoiduilla skripteillä, joten automatisointi ei päde pelkästään testaukseen (Crispin ja Gregory, 2009). Testaajien automaatio – osaamista on siis mahdollista hyödyntää myös muissa automatisointitehtävissä ja mittauksissa. Esimerkiksi itse ajauduin tekemään testaustehtävien ohessa skriptin lisenssin kulutuksen suoritusajan mittaamista varten.

Yksi osa – alue automatisoinnin hyödyntämiseen on datan luonti tai esivalmistelu. Jos testidatan valmisteluun menee jatkuvasti paljon aikaa, olisi suositeltavaa automatisoida kyseinen prosessi. Usein on tarve toistaa jotain tiettyä toimintoa useita kertoja saadakseen jokin virhe toistettua. Jos tehtävän voi automatisoida, voidaan samasta tuloksesta olla varmoja joka kerta. Testidatan puhdistus on yhtä tärkeää kuin sen luominen. Datan luontityökalussa pitäisi olla keinoja purkaa testidata, jottei se vaikuttaisi eri testeihin tai estäisi samojen testien uudelleenajamista (Crispin ja Gregory, 2009). Seuraavana pohditaan, mitä ei tulisi automatisoida.

3.3 Mitä ei tulisi automatisoida?

Osa testauksesta vaatii ihmisen silmien, korvien ja tietämyksen hyödyntämistä. Käytettävyystestaus ja tutkiva testaus ovat esimerkkejä tähän kategoriaan kuuluvista testityypeistä. Muita testejä, jotka eivät välttämättä oikeuta investointia testiautomaatioon, ovat yksittäiset testit ja sellaiset testit, jotka eivät epäonnistu koskaan (Crispin ja Gregory, 2009).

Käytettävyystestaus vaatii sitä, että joku varsinaisesti käyttää ohjelmistoa. Automatisointi voi olla hyödyllistä silloin, kun pohjustetaan skenaarioita käytettävyyden peräkkäiseen tarkasteluun. Käyttäjien tarkkailu käytännön tilanteissa, kyselyn suorittaminen käyttäjien kokemuksista ja tulosten analysointi ovat esimerkkejä tehtävistä, jotka kertovat siitä, että ohjelmiston käytettävyyttä ei voida automatisoida. Käyttäjän toimintojen kirjaus on hyödyllistä käytettävyystestauksessa (Crispin ja Gregory, 2009).

Gregory mainitsee esimerkkitapauksen manuaalisen käytettävyystestauksen tarpeellisuudesta. Gregoryn tiimi oli arvioinut useita graafisen käyttöliittymän työkaluja ja he olivat päätyneet käyttämään Ruby – kieltä ja sen Watir – selainautomatisointikirjastoperhettä. Testit olivat rajoitettu ainoastaan graafisen käyttöliittymän toiminnallisuuksiin. Yksi testeistä tarkisti, että käyttäjälle näytetään oikeat validointiviestit. Gregory ajoi testejä ja sattui katsomaan ruutua, koska hän ei ollut nähnyt kyseistä toisen testaajan luomaa testiä. Gregory huomasi jotain outoa, joten hän päätti toistaa testin, vaikka testi meni läpi. Yksi ohjelmoijista oli lisännyt ruudulle ”$” – merkin, ja virheviesti näkyi tämän vuoksi väärässä paikassa. Itse viesti

oli kuitenkin oikea. Tässä tapauksessa testien seuraaminen manuaalisesti oli erittäin arvokasta, koska julkaisu lähestyi, eikä ongelmaa olisi muuten havaittu ajoissa (Crispin ja Gregory, 2009).

On mahdollista automatisoida testejä, jotka varmistavat, että graafinen käyttöliittymä pysyy muuttumattomana. Tällöin on kuitenkin otettava huomioon siitä koituva hinta. Halutaanko todella välittää siitä, että napin asema on siirtynyt yhdellä pikselillä? Oikeuttavatko tulokset niiden tuottamiseksi vaadittavan työn? Crispin ja Gregory ovat sitä mieltä, että ulkoasuun liittyvää mielipidetestausta ei pitäisi automatisoida, koska automatisoitu skripti voi nähdä vain sen, mitä skriptille on saneltu. Automatisoinnilta menisi visuaaliset, ihmisen välittömästi havaitsemat ongelmat kokonaan ohi (Crispin ja Gregory, 2009).

Tutkivaa testausta voidaan nopeuttaa skripteillä, jotka luovat testidatan ja käyvät läpi joitain asetusvaiheita. Tämä kuitenkin vaatii, että osaava testaaja suunnittelee ja suorittaa testit. Yksi tutkivan testauksen tärkeimmistä tarkoituksista on oppia tuotteesta enemmän kokeilemalla tuotteen toimintoja. Tästä saatua tietoa voidaan käyttää tulevan kehityksen parantamiseen. Se ei ole mahdollista automatisoiduilla skripteillä. Usein tutkivaan testaukseen ei kuitenkaan jää aikaa ilman, että muita testejä automatisoidaan runsaasti (Crispin ja Gregory, 2009).

Crispinin ja Gregoryn mukaan on olemassa argumentti, jonka mukaan sellaisia testejä ei tarvitse automatisoida, jotka eivät koskaan epäonnistu. Jos vaatimus on niin selkeä, että sen toteuttamiseen on vain yksi keino, virheen esitteleminen kyseiseen toteutukseen on hankalaa. Esimerkiksi osoitekenttiä sisältävän lomakkeen tapauksessa voidaan pohtia, tarvitaanko automatisoitua regressiotestiympäristöä sen verifioimiseksi, että toinen osoiterivi ei ole pakollinen? Miten todennäköistä on, että manuaalisen verifioinnin jälkeen joku muuttaa kentän vahingossa pakolliseksi? Vaikka näin kävisi, se ei olisi merkittävästi haitallinen tapahtuma. Joku muu huomaisi sen, ja virhe pystyttäisiin vain sivuuttamaan, kunnes se korjataan (Crispin ja Gregory, 2009).

Toisaalta testin sisällyttäminen kyseiseen toiminnallisuuteen olisi helppoa. Ohjelmoijien kopioi – liitä – virheitä tapahtuu jatkuvasti. Automatisointia ei tarvita, mikäli koetaan, että yksittäinen manuaalinen testi on riittävän hyvä tarkoitukseen, ja

että tulevaisuuden virheiden uhka ei oikeuta automatisoituja regressiotestejä. Jos päätös osoittautuu vääräksi, automatisoinnin voi toteuttaa myöhemmin. Mikäli asiasta ei olla varmoja ja automatisoinnin toteuttaminen olisi helppoa, on automatisointi suositeltavaa. Kriittistä järjestelmää testattaessa yksikin pieni mahdollisuus regressiotestin epäonnistumisesta voi olla liikaa. Tällöin testien automatisoinnin päätöksenteon tukena kannattaa hyödyntää riskianalyysia (Crispin ja Gregory, 2009).

Usein voi olla tarpeellista suorittaa vain kertaluontoinen testi, jolloin automaatiota ei tarvitse toteuttaa. Esimerkkinä tästä on sellaisen toiminnallisuuden testaus, missä tulokseen vaikuttaa ajonaikainen päivämäärä. Ikävät tehtävät on kuitenkin hyvä automatisoida, vaikka niitä ei suorittaisi usein. Automatisoinnin aiheuttamat kulut tulisi arvioida testien manuaaliseen suorittamiseen kuluvaa aikaa vastaan. Jos testin suorittaminen manuaalisesti on helppoa, eikä automatisointi olisi nopeaa, testin suoritus kannattaa pitää manuaalisena (Crispin ja Gregory, 2009). Seuraavana pohditaan, mitkä asiat ovat vaikeasti automatisoitavissa.

3.4 Mitkä asiat ovat vaikeasti automatisoitavia?

Kun ohjelmakoodi ei ole kirjoitettu testivetoisesti tai testiautomaatio huomioiden, automatisoinnin toteuttaminen on hankalaa. Vanhat järjestelmät kuuluvat yleensä tähän kategoriaan. Myös sellaista uutta ohjelmakoodia tuotetaan paljon, joka sisältää piirteitä, joita ei voi testata (Crispin ja Gregory, 2009).

Testiautomaation toteuttaminen jo valmiille ohjelmakoodille on vaikeaa, mutta silti mahdollista. Vanhentuneessa ohjelmakoodissa voi olla tietokantaan, bisneslogiikkaan, käyttöliittymään ja tiedon käsittelyyn liittyvää koodia sekaisin. Tällöin ei ole välttämättä selkeää, mistä testiautomaation tekeminen kannattaisi aloittaa. Koska esittelykerros sisältää paljon ohjelman logiikkaa, kaikkien testien automatisointi graafisen käyttöliittymän alapuolisilta tasoilta ei ole suotavaa (Crispin ja Gregory, 2009).

Crispinin ja Gregoryn mukaan on olemassa ainakin muutamia erilaisia, hyväksi todettuja lähestymistapoja automatisointiin. Oppimiskäyrä voi olla pelottava, mutta kuitenkin mahdollinen ylittää, jonka jälkeen testiautomaatiosta tulee paljon

helpompaa. Jokin valmis ohjelmakoodin osa voidaan refaktoroida testiautomaation mahdollistamiseksi, jonka jälkeen osan ympärille voidaan tehdä testiympäristö. Jopa vanhentuneelle ohjelmakoodille voidaan näin kirjoittaa testejä suojelemaan uusien ongelmien ilmestymistä vastaan. Tämä lähestymistapa voi toimia myös sellaisilla järjestelmillä, joilla ei ole kunnollista rakennetta tai oliopohjaisuutta (Crispin ja Gregory, 2009).

Kuten missä tahansa automaatioprojektissa, vaikeasti automatisoitavaa koodia kannattaa lähestyä pala palalta aloittaen kriittisimmistä alueista. Crispinin tiimi päätti käyttää erilaista, mutta yhtä lailla tehokasta, lähestymistapaa. Tiimin jäsenet alkoivat työstää vanhentunutta ohjelmakoodia kirjoittamalla kaikki uudet toiminnallisuudet uuteen testiystävälliseen arkkitehtuuriin. Vanhaa ohjelmakoodia alettiin korvaamaan testivetoisella koodilla pala palalta. Kun oli tarve työskennellä vanhan ohjelmakoodin parissa bugien korjaamiseksi tai koodin päivittämiseksi, pystyttiin vain lisäämään yksikkötestit kaikelle muutetulle ohjelmakoodille. Graafisen käyttöliittymän testiympäristö kattoi kaikki kriittiset toiminnot siltä vanhentuneen järjestelmän osalta, jolla ei ollut yksikkötestejä (Crispin ja Gregory, 2009). Seuraavassa luvussa käsitellään testiautomaatiostrategian kehittämistä.

3.5 Testiautomaatiostrategian kehittämisen aloitus

Yksinkertainen, askel askeleelta käytävä lähestymistapa ei alustavasti vaikuttaisi sopivalta mihinkään automaatiostrategiaan. Ketterässä testaamisessa pyritään kuitenkin ensin ymmärtämään ongelmat. Päätös siitä, miten ja mistä aloittaa automatisointi, vaatii pohdintaa ja keskustelua tiimin kesken. Tiimin pohtiessa testaukseen liittyviä haasteita on harkittava sitä, missä kaikkialla automaatio on edes sopivaa. Ennen kuin aletaan etsimään jotain tiettyä automatisointityökalua, tulee vaatimukset tiedostaa riittävän tarkasti (Crispin ja Gregory, 2009).

Ensin on siis pohdittava ongelmaa, joka halutaan ratkaista ja sitä, mitä halutaan automatisoida. Jos testiautomaatio puuttuu kokonaan, ei kannata heti hankkia kallista, kaupallista testityökalua ajatellen, että kyseinen työkalu automatisoisi kaikki tarvittavat funktionaaliset testit. Sen sijaan tulisi pohtia, mikä on suurin tavoite, joka voidaan saavuttaa. Se on todennäköisesti yksikkötestien toteutus, jonka ohjelmoijat

pystyvät tekemään. Sen sijaan, että aloitettaisiin testipyramidin huipulta, voidaan aloittaa pohjalta ja varmistaa, että peruspalikat ovat kunnossa. Tällöin tulisi myös pohtia eri tyyppisiä testejä, joita tarvitsee automatisoida ja sitä, milloin tarvittavat työkalut tulisi olla valmiina (Crispin ja Gregory, 2009).

Kun tiimi pohtii mihin keskittyä automatisoinnissa seuraavaksi, on otettava huomioon mikä osa – alue on vaikein tai tylsin testata. Saadaanko ohjelmakoodia edes valmiiksi sen testaamista varten? Tuntevatko tiimin jäsenet olonsa tarpeeksi itsevarmoiksi muuttaakseen ohjelmakoodia, vai puuttuuko heiltä automatisoitujen testien tuoma turvaverkko? Tiimin jäsenet saattavat olla edistyneempiä testivetoisen kehityksen osaajia, ja heillä voi olla täysi yksikkötestiympäristö. Tiimillä ei kuitenkaan välttämättä ole hyvää ohjelmistokehystä bisnespohjaisiin testeihin, tai testien automatisointi on hankalaa. Jos graafisen käyttöliittymän testejä on jo olemassa, saattavat ne olla hitaita ja kalliita ylläpitää (Crispin ja Gregory, 2009).

Jos kaikki aika käytetään jo testattujen ominaisuuksien uudelleentestaamiseen, uusien ominaisuuksien testaukseen ei ole aikaa, ja on tarve lisätä yhä enemmän testausta, kärsitään vakavasta testiautomaation puutteesta. Tämä tilanne tarkoittaa sitä, että testaajilla ei ole aikaa osallistua suunnittelu – ja toteutuskeskusteluihin, regressiovirheitä voi ilmaantua huomaamatta, ja testaus ei pysy enää kehityksen perässä, jolloin testaajat jäävät jumiin. Tällöin ohjelmoijat eivät enää työskentele bisnespainotteisten testien parissa, eikä testaajilla ole aikaa pohtia parempaa keinoa ratkaista testauksen ongelmia. Tiimi voi korjata tämän tilanteen kehittämällä automatisointistrategian. Siihen kuuluu ohjelmakoodin kehittäminen testivetoisesti ja sopivien automaatiotyökalujen valinta ja toteuttaminen. Testaajat saavat näin tilaisuuden kehittää teknisiä taitojaan (Crispin ja Gregory, 2009).

Automatisointityö on paras aloittaa siitä osa – alueesta, joka tuntuu vaikeimmalta. Jos tiimillä on esimerkiksi vaikeuksia tuottaa edes käyttöönotettavaa ohjelmakoodia, tiimin tulisi toteuttaa automatisoitu koontiprosessi. Tällöin testaajien ei tarvitse odotella turhaan saadakseen käsiinsä testattavaa ohjelmakoodia. Jos ohjelman suorituskyvyssä taas on puutteita, täytyy performanssitestauksesta tulla tärkein tehtävä. Crispinin ja Gregoryn mukaan oleellista on pyrkiä ymmärtämään, mitä

ongelmaa aiotaan ratkaista. Riskianalyysi on yksi apukeino tilanteeseen (Crispin ja Gregory, 2009).

Gregory kertoo esimerkkitapauksen, jossa hän työskenteli vanhentuneen järjestelmän parissa yrittäen parantaa laatuongelmia ja samalla lisätä uusia ominaisuuksia tärkeää asiakasta varten. Käytössä olevalla sovelluksella ei ollut automatisoituja yksikkötestejä tai funktionaalisia testejä ja ohjelmakoodi täytyi refaktoroida laatuongelmien korjaamiseksi. Tiimin jäsenet päättivät käydä projektin kimppuun osa osalta. Kun tietty toiminallisuus valittiin refaktoroitavaksi, ohjelmoijat kirjoittivat yksikkötestit ja varmistivat, että ne menivät läpi. Tämä jälkeen he kirjoittivat ohjelmakoodin uudelleen ja varmistivat taas, että samat testit menivät läpi. Refaktoroinnin loputtua tuloksena oli testattavissa oleva, hyvin kirjoitettu ohjelmakoodi ja siihen yhdistettävät testit. Testaajat kirjoittivat samaan aikaan korkeamman tason funktionaalisia testejä. Suurin osa heikkolaatuisesta vanhentuneesta ohjelmakoodista oli uudelleenkirjoitettu vuoden kuluessa ja tiimi saavutti hyvän testikattavuuden ratkomalla ongelmat yksi kerrallaan (Crispin ja Gregory, 2009).

Testiautomaatio ei maksa itseään takaisin, ellei käytössä ole hyväksi todettuja kehitysmenetelmiä. Ensimmäinen askel muiden testien automatisointiin on jatkuvan integroinnin perustaminen ketterän yksikkötestiympäristön säännölliseen ajamiseen. Kun ohjelmakoodia refaktoroidaan jatkuvasti ylläpidettävyyttä ja hyvää suunnittelua varten, myös automaation investoinnin palautusarvo kasvaa. Refaktorointia ei voida tehdä ilman hyvää yksikkötestikattavuutta. Näitä kehityskäytäntöjä täytyy soveltaa myös automatisoituihin funktionaalisiin testiskripteihin (Crispin ja Gregory, 2009).

Testipyramidikonsepti on auttanut tiimejä sijoittamaan automaatioon vaadittavan työtaakan sinne, missä työtä saadaan eniten aikaiseksi. Yleensä halutaan maksimoida parhaan investoinnin palautusarvon tuottavien testien määrä. Jos järjestelmän arkkitehtuuri on suunniteltu testattavuutta varten, testiautomaation kustannukset käyvät pienemmiksi etenkin yksikkötasolla (Crispin ja Gregory, 2009).

Testipyramidin pienikokoisella huipulla sijaitsevilla käyttöliittymän läpi ajettavilla testeillä on yleensä matalin investoinnin palautusarvo, koska ne ovat tarpeellisuudestaan huolimatta kalliita ylläpitää. On mahdollista automatisoida osa

näistä testeistä, mutta suurin osa graafisen käyttöliittymän testeistä on määritelty bisnestermein. Tällöin voi olla parasta jättää testit osaksi manuaalista testausta (Crispin ja Gregory, 2009).

Pyramidin keskimmäinen kerros edustaa funktionaalisia testejä, jotka toimivat suoraan tuotantokoodin kanssa ilman graafista käyttöliittymää tai muuta välikerrosta. Vaikka ne antavat palautetta hitaammin ja ovat kalliimpia automatisoida kuin yksikkötason testit, voidaan oikeilla työkaluilla saada aikaiseksi hyvä investoinnin palautusarvo. Testien arvoa lisää myös se, että ne voidaan kirjoittaa sellaisella kielellä, jota bisnesekspertit ymmärtävät (Crispin ja Gregory, 2009).

Sovellukset sisältävät useita eri kerroksia, joita voidaan testata itsenäisesti. Kun pyritään testaamaan kaikkia sovelluksen kerroksia erikseen, on silti varmistettava, että kerrokset toimivat hyvin keskenään. Tämä voi vaatia vähintään yhden bisneslogiikan testin esityskerroksen kautta (Crispin ja Gregory, 2009).

Kun arvioidaan automatisointityön hyötyjä, tulee harkita myös vähemmän konkreettisia näkökulmia, kuten sitä, tukeeko käytettävä työkalu yhteistyötä eri teknisten tiimien ja asiakastiimien välillä. Testien kirjoittamisen yksi pääsyistä on kehityksen ohjaamisen auttaminen. Jos automatisoitujen hyväksymistestien kirjoittamisprosessilla saavutetaan täydellinen bisnesvaatimusten ymmärtäminen, saadaan jo reilusti palautetta, vaikka testit eivät ikinä havaitsisi yhtään regressiovirhettä (Crispin ja Gregory, 2009). Olen itse huomannut työympäristössä, että testiautomaation toteuttaminen vähemmän tutulle toiminnallisuudelle on auttanut merkittävästi toiminnallisuuden ymmärtämisessä.

Myös testien suunnittelua ja ylläpidettävyyttä tulisi harkita tarkkaan. Yksi harkinnan kohde on se, olisiko aiemmin kirjoitettuja manuaalisia testiskriptejä voitu automatisoida. Kun näitä testejä aletaan muuttaa automaattisiksi, voidaan huomata, että prosessi on melko helppo. Esimerkiksi testitapausten lisääminen FitNesse – testityökaluympäristöön on vaivatonta. Tästä on hyötyä, kun erilaisia testattavia permutaatioita on useita. Tällöin tulee myös todennäköisesti testattua useampia ehtoja, kuin mitä manuaalinen testaus kattaisi. Crispin kertoo esimerkin, jossa hänen tiiminsä kirjoitti eläkelainasovelluksen uudelleen ja pystyi testaamaan satoja erilaisia lainan maksuprosessiin liittyviä testitapauksia FitNesse – testien avulla. Näihin tapauksiin

lukeutuivat esimerkiksi ne, missä kolme lainanmaksua käsitellään samana päivänä, missä joku ei suorita maksuja kolmeen kuukauteen ja suorittaa tämän jälkeen ison lyhennyksen, ja tarkastaessa, lasketaanko ja toteutetaanko lainan korko oikein. Näille tapauksille oli helppo kirjoittaa automatisoitu ratkaisu (Crispin ja Gregory, 2009).

Tämä menetelmä tuo suuren hyödyn, mutta siinä on myös huonoja puolia. Tässä vaiheessa tiimillä olisi kymmeniä, ellei satoja ylläpidettäviä testitapauksia. Entä jos lainanmaksun koron määrän laskennan säännöt muuttuvat vuoden kuluttua? Tämä vaatisi jokaisen testin päivittämistä. Jos käytettävällä testityökalulla on hankala tehdä muutoksia valmiina oleviin testeihin, isosta automatisoidusta testiympäristöstä voi tulla hankala ongelma. Varsinkin koko sovelluksen käyttökulun alusta loppuun suorittamiseen liittyvä testaus on hankalaa, koska nämä testit tarvitsevat eniten ylläpitoa bisneslogiikan muuttuessa. Automatisoinnin tarve tulisi tasapainottaa automatisoinnista aiheutuvien kulujen kanssa (Crispin ja Gregory, 2009).

Testien suunnittelu tulisi aloittaa testattavan ominaisuuden pienestä siivusta. Automatisointia tulisi lähestyä samalla tavalla, kuin ohjelmoijat lähestyvät ohjelmointityötä. Ensin saadaan yksi pieni testiyksikkö toimivaksi ja sitten siirrytään seuraavaan. Kun koko siivu on saatu katettua, palataan takaisin, hiotaan kokonaisuus kuntoon ja valitaan testikaava harkiten. Vain matalimman mahdollisen tason tarpeellisimmat testitapaukset tulisi automatisoida. Jokaisen testitapauksen laajuus on rajoitettava yhteen testiehtoon tai bisnesvaatimukseen. Testin tarkoitus olisi aina ymmärrettävä. Myös testien välisiä riippuvuuksia olisi vältettävä, koska testeistä tulee nopeasti monimutkaisempia ja kalliimpia ylläpitää (Crispin ja Gregory, 2009).

Automatisoitu testi omaa sitä paremman investoinnin palautusarvon, mitä matalammalla tasolla testi sijaitsee. Toisin sanoen testiautomaatio tulisi toteuttaa niin alhaalla testipyramidissa kuin mahdollista. Jos yksikkötesteillä ja ohjelmakoodin integraatiotesteillä on hyvä testikattavuus, ei ole tarvetta automatisoida yhtä montaa funktionaalista testiä. Kun alemmilla tasoilla testien kattavuus on hyvä, on riittävää tehdä ohjelman ajon alusta loppuun suorittavat testit manuaalisesti järjestelmän käyttäytymisen verifioimiseksi. Riskianalyysia voidaan jälleen käyttää apuna päätöksenteossa (Crispin ja Gregory, 2009).

Joissain tilanteissa testiautomaation rooli on kriittinen myös graafisen käyttöliittymän tasolla. Tiimi saattaa käyttää kolmannen osapuolen graafisen käyttöliittymän ohjainta, eikä sen käyttäytymisestä olla välttämättä varmoja. Investointi kannattaa tehdä, jos riskianalyysi ja investoinnin palautusarvon analyysi tukee suurta automatisoinnin määrää graafisen käyttöliittymän tasolla (Crispin ja Gregory, 2009).

Jos automatisointi toteutetaan korkeammilla tasoilla, ei tulisi automatisoida jokaista mahdollista järjestelmän käyttäytymispolkua. Regressiotestiympäristössä ei tarvitse säilyttää jokaista kehitysvaiheessa tehtyä automatisoitua testiä. Sen sijaan tulisi saavuttaa tasapaino ajansäästön ja virheiden löytämisen mahdollisuuksien välillä. Parhaimman hyödyn saavuttamiseksi tulisi kattaa jokainen tärkeä ohjelmakoodin polku yksikkötasolla, integraatiotasolla ja funktionaalisella tasolla (Crispin ja Gregory, 2009).

Tasapainon saavuttaminen ei ole pelkästään yksi ketterän kehityksen arvoista, vaan järkevää myös muulla tasolla. Vaikka olisi välitön tarve saada tarpeeksi hyvä ratkaisu aikaiseksi, ei ratkaisun tarvitse silti olla täydellinen. Työkalu on sopiva käytettäväksi, jos se antaa tarvittavia tuloksia ja tuo riittävän hyödyn automatisointiin vaadittaviin resursseihin nähden. Aikaa muiden vaihtoehtojen tarkasteluun voidaan varata myöhemmin. Automaatioympäristöä voidaan aina parantaa ajan kanssa. Tärkeintä on selvittää, sopivatko automaatiotyökalut kyseiseen tilanteeseen (Crispin ja Gregory, 2009).

Ei pidä ajatella, että jollakin työkalulla voidaan generoida joukko skriptejä, saada välittömät testitarpeet täytettyä ja refaktoroida skriptit ylläpidettävään muotoon myöhemmin. Vaikka ideaalisinta automaatioratkaisua ei tarvitse jatkuvasti etsiä, tarvitaan kuitenkin sellainen ratkaisu, joka ei lisää tiimin teknistä velkaa. Eleganteimman, hienoimman ratkaisun ja halvalla ylläpidettävän, vain tarvittavat virheet löytävän ratkaisun välillä olisi löydettävä tasapaino (Crispin ja Gregory, 2009).

Usein testiautomaatio minimoidaan graafisen käyttöliittymän tasolla, mutta on myös tilanteita, joissa graafisen käyttöliittymän automaatiota tarvitaan enemmän. Jos käyttäjä tekee muutoksen tietyssä paikassa, ei aina voida tietää, mikä muu osa sovelluksen käyttäytymisessä muuttuu. Jotkut ongelmat esiintyvät vain graafisen käyttöliittymän tasolla. Esimerkiksi Crispin testasi bugikorjausta, joka koski

sovelluksen logiikan ongelmaa, jossa eläköitymissuunitelmaan osallistuneilta pyydettiin rahaa heidän tililtään. Vaikka yksikkötestit käsittelivät tätä muutosta, graafisen käyttöliittymän regressiotestit epäonnistuivat, koska lomake ei tullut pyynnöstä esiin. Kukaan ei ollut odottanut, että muutos sovelluksen logiikassa voisi vaikuttaa graafiseen käyttöliittymään, joten manuaalista testausta olisi tuskin suoritettu. Tämän vuoksi tarvitaan myös graafisen käyttöliittymän regressiotestejä (Crispin ja Gregory, 2009).

Vaikka tallennus – ja toistotyökalut eivät aina ole ideaalisia, ovat ne kuitenkin tietyissä tilanteissa sopivia. Tallennus – ja toistotyökalua saatetaan käyttää hyvään tarkoitukseen esimerkiksi silloin, kun vanhentuneelle ohjelmakoodille on jo luotu automatisoitu testiympäristö, tiimillä on paljon kokemusta työkalusta tai organisaation hallinto haluaa jostain syystä käyttää kyseistä työkalua. Tallennettuja skriptejä voidaan käyttää aloituspisteenä. Tämän jälkeen skriptit voidaan jakaa moduuleihin, korvata kovakoodattu data tarvittaessa parametreilla, ja koota testit käyttäen moduuleja rakennuspalikoina. Pienelläkään ohjelmointikokemuksella ei ole hankalaa tunnistaa ne skriptin osat, jotka tulisi olla tietyssä moduulissa, kuten sisäänkirjautumisessa (Crispin ja Gregory, 2009).

Tallennus – ja toisto voi olla sopivaa myös vanhentuneille järjestelmille, jotka on suunniteltu niin, että yksikkötestaus on vaikeaa ja testiskriptien kirjoittaminen käsin alusta asti on liian kallista. Oikealla suunnittelulla ja tallennetun interaktion helposti luettavan formaatin käytöllä on mahdollista rakentaa toistettavat testit ennen kuin ohjelmakoodi on koottu (Crispin ja Gregory, 2009).

Jotkut ketterät tiimit saavat arvoa kaupallisista tai avoimen lähdekoodin testityökaluista ja toiset taas suosivat täysin räätälöityä lähestymistapaa. Moni testaaja voi havaita hyötyjä kirjoittamalla yksinkertaisia skriptejä jollakin skriptauskielellä automatisoidakseen yksinkertaisia, mutta tarpeellisia tehtäviä, generoidakseen testidataa tai ajaakseen muita työkaluja. Ratkaistavaa ongelmaa on ensin tarkasteltava tiimin kesken ja päätettävä helpoin ja tehokkain tapa ongelman ratkaisuun. Välillä tulee arvioida käytettyjä työkaluja ja pohtia, onko tiimi edelleen tyytyväinen työkaluihin vai jääkö ongelmia huomaamatta siitä syystä, ettei käytössä ole oikeita työkaluja. Aikaa tulisi käyttää myös uusien työkalujen tutkimiseen ja sen

harkitsemiseen, täyttävätkö työkalut kaikki aukot tai korvaavatko ne jo käytössä olevat sopimattomat työkalut (Crispin ja Gregory, 2009).

Jos ketterä kehitys on uusi konsepti tiimille tai tiimi työskentelee uuden projektin parissa, saatetaan työskennellä korkean riskin toiminnallisuuksien parissa samaan aikaan, kun valitaan työkaluja ja pystytetään testiympäristöjä ensimmäisten iteraatioiden aikana. Jos testi – infrastruktuurin luonti on vielä kesken, ei kannata odottaa, että pystytään tuottamaan paljoa bisnesarvoa. Työkalujen arviointiin, koontiprosessien pystyttämiseen ja erilaisten testauksen lähestymistapojen kokeilemiseen tulisi käyttää paljon aikaa (Crispin ja Gregory, 2009). Testityökalujen valintaa pohditaan enemmän luvussa neljä. Seuraavaksi käsitellään ketterien periaatteiden soveltamista testiautomaatiossa.

3.6 Ketterien periaatteiden soveltaminen testiautomaatiossa

Jokaisella tiimillä, projektilla ja organisaatiolla on erityinen tilanne ja erityiset automaatiohaasteensa. Oli tiimin tilanne mikä tahansa, voidaan ratkaisujen etsimisen helpottamiseksi käyttää ketteriä periaatteita ja arvoja. Crispinin ja Gregoryn mukaan käsitteet, kuten rohkeus, palaute, yksinkertaisuus, kommunikointi, jatkuva kehittäminen ja muutoksiin mukautuminen eivät ole pelkästään ketteriä ideoita, vaan ominaisuuksia, joita löytyy kaikilta menestyviltä tiimeiltä (Crispin ja Gregory, 2009).

Ketterän periaatteen ajatus toteuttaa yksinkertaisin toimiva asia pätee ohjelmakoodin lisäksi myös testeihin. Testisuunnitelma tulisi pitää yksinkertaisena ja mahdollisimman pienenä. Tehtävän suorittamiseen voidaan käyttää yksinkertaisinta työkalua. Yksinkertaisuus ei aina tarkoita helpointa lähestymistapaa. Se sisältää syvän ajatustyön siitä, mitä juuri nyt tarvitaan. Tämän jälkeen otetaan ensimmäiset askeleet tavoitetta kohti. Kun tavoite pidetään yksinkertaisena, ei vääristä valinnoista jouduta pahasti sivuraiteille, ennen kuin virhe huomataan (Crispin ja Gregory, 2009).

On helppoa syventyä johonkin kiehtovaan haasteeseen ja luistaa perusteista. Investoinnin palautusarvo tulisi arvioida ennen jokaista automaatiotehtävää. Automaatio on hauskaa, kun siihen pääsee kunnolla kiinni. On houkuttelevaa yrittää tehdä jotain haastavaa vain siksi, koska se on mahdollista. Kuten kaikki muut ketterän

kehitysprojektin testaamisen osa – alueet, ainoa tapa pysyä tahdissa on tehdä ainoastaan minimi vaadittu työ (Crispin ja Gregory, 2009).

On suositeltavaa käyttää yksinkertaisinta työkalua, millä haluttu tehtävä saadaan suoritettua. Asiakaspohjaiset testit kannattaa automatisoida yksikkötasolla, mikäli se on yksikkötasolla helpointa. Kun esimerkiksi testitapauksia kirjoitetaan FitNessella, Crispinin ja Gregoryn mukaan voidaan huomata, että ohjelmoijat automatisoivat samat testitapaukset nopeammin JUnit – testeillä. Joskus taas ohjelmoijat käyttävät FitNessea testivetoiseen kehitykseen JUnitin sijaan, koska heidän kirjoittamansa ohjelmakoodi auttaa testaamaan FitNessen testiympäristössä (Crispin ja Gregory, 2009).

Lyhyet iteraatiot mahdollistavat erilaisten automaation lähestymistapojen kokeilemisen, tulosten arvioimisen ja lähestymistavan muuttamisen mahdollisimman nopeasti. Automaatiotyöhön tulisi sitoutua esimerkiksi kehittämällä organisaation sisäinen testikehys tai toteuttamalla avoimen lähdekoodin työkalu ainakin muutaman iteraation ajaksi. Jokaisen iteraation jälkeen voidaan tarkastella, mikä toimii ja mikä ei. Ratkaisuja ongelmien ylittämiseksi voidaan pohtia ja hyödyntää seuraavassa iteraatiossa. Jos ratkaisu ei ole oikea, voidaan muutaman iteraation ajan kokeilla jotain muuta. On kuitenkin vaara joutua tilanteeseen, jossa työkaluun on käytetty niin paljon resursseja ja niin moni testi on riippuvainen kyseisestä työkalusta, että työkalun vaihtaminen ei ole enää hyvä vaihtoehto. Koska ohjelmoijat kykenevät kirjoittamaan omia testityökaluja, ja valmiita kaupallisia ja avoimen lähdekoodin työkaluja on niin paljon, ei ole syytä tyytyä mihinkään muuhun kuin optimaalisimpaan työkaluun. Iteraatioiden askelmaista lähestymistapaa voidaan siis hyödyntää muuttamalla käytettävää työkalua tarvittaessa (Crispin ja Gregory, 2009).

Ketterä kehitys ei toimi ilman automaatiota. Koko tiimin lähestymistavalla voidaan hyödyntää suurempaa joukkoa kykyjä ja resursseja hyödyllisen automaatiostrategian löytämiseksi ja toteuttamiseksi. Ongelman ratkaisu tiimin kesken johtaa todennäköisemmin siihen, että ohjelmakoodi suunnitellaan testattavuutta ajatellen. Ohjelmoijat, testaajat ja muut tiimin jäsenet tekevät yhteistyötä testien automatisoimiseksi tarjoten useita näkökulmia ja kykyjä (Crispin ja Gregory, 2009).

Koko tiimin lähestymistapa auttaa testien automatisoinnin suuren työtaakan aiheuttamaan pelkoon. Tiimi saa rohkeutta, kun mukana on erilaisia taitoja ja kokemuksia omaavia jäseniä. Kun tiimin jäsenet kykenevät auttamaan muita ja kysymään toisilta apua, he saavat itseluottamusta riittävän testiautomaation kattavuuden saavuttamiseksi (Crispin ja Gregory, 2009).

Myös Collins et al. ovat todenneet tutkimuksessaan, että paras testiautomaatiostrategia ketterässä kehityksessä on se, että ohjelmoijat ja testaajat tekevät yhteistyötä automatisoidakseen yksikkötestit ja järjestelmätestit. Riskien minimoimiseksi tämä strategia vaatii kuitenkin tiettyjä toimenpiteitä. Tiimin yhteistyö vaatii, että testaajat ovat mukana ympäristön konfigurointiin ja jatkuvaan integrointiin liittyvissä tehtävissä. Tällöin tiimit eivät erkaannu toisistaan, vaan strategia rohkaisee tiimejä yhteistyöhön (Collins et al., 2012).

Erikoistuneet teknologiapohjaiset testit, kuten tietoturvatestit ja performanssitestit, saattavat tarvita tiimin ulkopuolista asiantuntemusta. Joillakin yrityksillä on erikoistuneita tiimejä, jotka ovat saatavilla jaettuna resurssina tuotantotiimien käyttöön. Vaikka näitä resursseja hyödynnettäisiin, tulisi ketterien tiimien silti ottaa vastuu erityyppisten testien suorittamisesta. Tiimit voivat myös yllättyä siitä, että tiimin jäsenet saattavat omata tarvittavia taitoja käytettäessä luovaa lähestymistapaa (Crispin ja Gregory, 2009).

Joillain organisaatioilla on itsenäisiä testaustiimejä, jotka tekevät kehityksen jälkeistä testausta. Tiimit saattavat suorittaa erikoistunutta testaamista, kuten järjestelmien välistä testausta tai laajaa performanssitestausta. Kehitystiimien tulisi työskennellä läheisesti testitiimien kanssa ja hyödyntää testaamisesta saatavaa palautetta ohjelmakoodin suunnittelun parantamiseksi ja automaation hyödyntämiseksi (Crispin ja Gregory, 2009).

Ongelmien selvittäminen ja hyvien ratkaisujen toteutus vievät aikaa. Aikaa vieviä prosesseja ovat etenkin ideointi, ratkaisut, koulutus ja työssäoppiminen. Usein on tarve auttaa johtoryhmää ymmärtämään, että ilman riittävää aikaa tehdä asiat oikealla tavalla tekninen velka kasvaa ja kehityksen vauhti hidastuu. Ratkaisujen toteuttaminen oikealla tavalla vie aluksi paljon aikaa, mutta säästää aikaa pitkällä tähtäimellä (Crispin ja Gregory, 2009).

Organisaatioiden johtoryhmät ovat luonnollisesti kiinnostuneita tuottamaan tuloksia mahdollisimman nopeasti. Jos johtoryhmä ei ole halukas antamaan tiimille aikaa automaation toteuttamiseen, tulisi hyvät ja huonot puolet käydä selkeästi läpi johtoryhmän kanssa. Toiminnallisuuksien julkaisu lyhyellä aikavälillä tulee kalliiksi, jos mukana ei ole automatisoituja regressiotestejä toiminnallisuuksien varmentamiseksi. Teknisen velan kertyessä tiimin on hankalampi toimittaa johtoryhmän vaatimaa bisnesarvoa. Tällöin voidaan tehdä kompromissi, jossa toiminnallisuuden laajuutta leikataan ja toteutetaan automaatio laadukkaamman tuotteen julkaisemiseksi ja ylläpitämiseksi (Crispin ja Gregory, 2009).

Yleensä tiimeillä on kiire tiukkojen aikarajojen vuoksi. Tiimeillä saattaa olla jatkuva houkutus palata vanhoihin toimintatapoihin, kuten regressiotestien suorittamiseen manuaalisesti, vaikka tiimi tiedostaisi, ettei kyseinen toimintatapa ole tehokas. Takaisin palaamiseen ja ongelmien korjaamiseen ei tunnu ikinä olevan tarpeeksi aikaa. Iteraation suunnittelupalaverissa tiimit voivat varata aikaa automatisoinnin edistämiseksi. Yleensä testaajilla on työn alla useita eri tehtäviä, joten tehtävien yhtäaikaista suorittamista tulisi välttää. Uuden työkalun opetteluun tai uusien testien automatisointiin tulisi varata paljon aikaa ja keskittyä vain näihin tehtäviin. Tämä saattaa olla hankalaa, mutta ei kuitenkaan yhtä hankalaa, kuin tehtävien jatkuva vaihtelu (Crispin ja Gregory, 2009).

Testit ovat yhtä arvokkaita kuin itse tuotantokoodi. Tuotantokoodin arvo laskee, jos sillä ei ole koodia tukevia testejä. Testejä tulisi kohdella samalla tavalla kuin muutakin ohjelmakoodia, kuten pitää sitä saman versionhallintaohjelmiston alla. Tämä auttaa tunnistamaan, mitkä testiskriptien versiot toimivat minkäkin ohjelmakoodin version kanssa (Crispin ja Gregory, 2009).

Kaikki hyvälaatuisen ohjelmakoodin ominaisuudet ovat myös hyvien automatisoitujen testien ominaisuuksia, kuten refaktorointi, yksinkertainen suunnittelu, modulaarisuus, oliopohjainen suunnittelu, hyvät standardit, ja testien pitäminen mahdollisimman itsenäisinä. Ketterä kehitys saattaa joskus vaikuttaa kaoottiselta, vaikka todellisuudessa se on hyvin kurinalaista. Automatisointitehtävät tulisi hoitaa suurella kurinalaisuudella pienissä askelissa. Automatisoituja skriptejä voidaan kirjoittaa testivetoisesti samalla tavalla kuin ketterä ohjelmoija kirjoittaisi tuotantokoodia.

Testien tulisi kuitenkin olla yksinkertaisia, sillä monimutkaiset, paljon logiikkaa sisältävät skriptit vaativat testausta ja ovat kalliita ylläpitää. Testit tulisi määritellä tarkasti ennen niiden ohjelmointia (Crispin ja Gregory, 2009).

Crispinin ja Gregoryn mukaan tiimityöskentelyn tärkeyttä testiautomaatiossa ei voi korostaa tarpeeksi. Tiimin jäsenten vaihteleva kokemustaso, kyvyt ja näkökulmat voivat toimia yhdessä parhaan lähestymistavan saavuttamiseksi. Tiimin täytyy olla innovatiivinen ja tehdä ratkaisut jokaisen tilanteen mukaan. Automaatiotyökalut ovat vain yksi pala kokonaisuutta. Myös testidata ja testiympäristöt ovat tärkeitä komponentteja (Crispin ja Gregory, 2009). Seuraavaksi käsitellään testien syötedatan muodostamista.

3.7 Syötedatan muodostaminen

Käytettiin testien automatisointiin mitä työkalua hyvänsä, testit käsittelevät aina dataa. Ihanteellisessa tilanteessa testit pääsisivät käsiksi realistiseen, tuotantodataa vastaavaan dataan. Yleensä tuotantotietokannat ovat kuitenkin huomattavan kokoisia ja monimutkaisia. Testit myös hidastuvat eksponentiaalisesti, kun otetaan yhteydenotto tietokantaan (Crispin ja Gregory, 2009).

On olemassa useita käteviä työkaluja testidatan generoimiseen erilaisilla syötekentillä ja rajaehdoilla. Kaupallisilla ja avoimen lähdekoodin työkaluilla voidaan generoida testitiedostoja tai dataa suoraan tietokantatauluihin. Näillä työkaluilla voidaan luoda erityyppistä dataa erilaisilla variaatioilla. On myös melko helppoa luoda testidataa itsetekoisilla skripteillä käyttäen esimerkiksi skriptikieliä kuten Ruby ja Python, kuoriskriptejä, tai työkaluja kuten Fit ja FitNesse (Crispin ja Gregory, 2009).

Testidatan generoimiseen käytettävät skriptit ja työkalut eivät tarvitse olla monimutkaisia. Esimerkiksi PerlClipin avulla voidaan yksinkertaisesti generoida tekstiä Windowsin leikepöydälle, josta sen voi liittää muualle. Mikä tahansa ratkaisu on kokeilemisen arvoinen, jos se poistaa turhaa työtä riittävästi ja antaa keskittyä mahdollisten ongelmien löytämiseen. Testidatan luomiseenkin riittää yksinkertaisin toimiva ratkaisu. Testit voidaan näin pitää mahdollisimman yksinkertaisina ja nopeina (Crispin ja Gregory, 2009).

Ensimmäisenä olisi hyvä luoda testejä, jotka ajetaan täysin muistin sisäisesti. Vaikka testidata täytyy silti pystyttää ja purkaa, sitä ei säilötä tietokannassa. Jokainen tällainen testi on itsenäinen ja nopeasti ajettavissa. Tietokantayhteyden käyttö johtaa aina levyltä lukemiseen ja kirjoittamiseen, mikä on hidasta. Jos tavoitteena on antaa nopeaa palautetta tiimille, testien tulisi suorittua mahdollisimman nopeasti. Muistin sisäinen tietokanta antaa testeille mahdollisuuden tehdä vaativaa työtä ja antaa silti nopeaa palautetta (Crispin ja Gregory, 2009).

Testiautomaation toteuttamisen aloittaminen on hankalaa. On helppoa todeta, että vaikka nykyisten testien suoritusaika kestää tuntikausia, ovat ne silti parempi vaihtoehto, kuin että testejä ei olisi ollenkaan. Tietokantayhteyden käyttö on suurimpia testejä hidastavia tekijöitä. Tätä vastaan voidaan ottaa pieniä askeleita tietokannan jäljittelemiseen ja testata ohjelmiston logiikkaa mahdollisimman paljon ilman tietokantaa. Mikäli tämä osoittautuu haastavaksi, voidaan järjestelmän arkkitehtuuria arvioida uudelleen ja tarkastella, onko se muokattavissa paremmin testattavaksi (Crispin ja Gregory, 2009).

Kun testataan ohjelmakoodissa olevaa bisneslogiikkaa, algoritmeja tai laskutoimituksia, ollaan yleensä kiinnostuneita ohjelmakoodin käyttäytymisestä tietyillä syötteillä ja siitä, että testidata vastaa oikeaa dataa. Datan alkuperä ei ole niinkään tärkeää. Tällöin voidaan luoda testien osana oleva muistinsisäinen testidata ja antaa tuotantokoodin toimia kyseisen datan perusteella. Itse testin tarkoitukseen pystytään keskittymään simuloimalla tietokantayhteyttä – ja objekteja. Tällöin testien suoritusaika on nopeampi ja testejä on helpompi kirjoittaa ja ylläpitää (Crispin ja Gregory, 2009).

Testidataa generoidessa olisi hyvä käyttää sellaisia arvoja, jotka heijastavat testin tarkoitusta. Jokaiselle testille voidaan luoda ainutlaatuisia arvoja, mikäli ollaan varmoja siitä, että jokainen testi on itsenäinen. Esimerkiksi aikaleimoja voidaan käyttää arvojen osana. Ainutlaatuinen data takaa sen, että testit eivät vaikuta toisiinsa hajadatalla. Kun tarvitaan suuria datamääriä, testidata voidaan generoida satunnaisesti. Data tulisi kuitenkin aina siistiä testin lopussa, jottei se vaikuta tuleviin testeihin. Kun taas testataan erityislaatuista dataa, satunnaisesti generoitu data voi mitätöidä testin

tarkoituksen. Satunnaisen testidatan käyttöä voidaan kuitenkin harjoittaa juuri sen verran, että jokaiselle testille annetaan ainutlaatuinen syöte (Crispin ja Gregory, 2009).

Kun testattava järjestelmä nojaa vahvasti tietokantaan, on tietokanta huomioitava myös testeissä. Jos testattava ohjelmakoodi lukee tai kirjoittaa dataa tietokantaan, tarvitaan regressiotestejä ohjelmakoodin tietokantakerroksen verifioimiseen. Yksi lähestymistapa testeille on lisätä testattava data skeemaan, työstää data ja verifioida tulokset tietokannassa. Lopuksi poistetaan kaikki testidata, jotta vaikutus ei näkyisi seuraavissa testiajoissa. Tämä tukee ajatusta siitä, että testit ovat toisistaan riippumattomia (Crispin ja Gregory, 2009).

Toinen vaihtoehto on ylläpitää testiskeemoja, jotka voidaan päivittää nopeasti syötetietokannan datalla. Syötedatan olisi tarkoitus edustaa oikeaa tuotantodataa. Koska datan määrä on pieni, se voidaan rakentaa nopeasti uudelleen joka kerta, kun tarvitaan ajaa regressiotestiympäristö. Tässä lähestymistavassa testien suoritusaika on hieman hitaampi johtuen regressiotestiympäristön ajon alussa tehtävistä toimenpiteistä. Vaikka testit ovat hitaampia kuin testit, jotka eivät käytä tietokantaa, ovat ne silti nopeampia kuin sellaiset testit, joiden täytyy työläästi populoida jokainen kolumni jokaisessa taulussa (Crispin ja Gregory, 2009).

Syötedatalla on useita käyttötarkoituksia. Testaajilla ja ohjelmoijilla voi olla oma testiskeema, jota voidaan tarvittaessa päivittää. He voivat suorittaa manuaalisia ja automatisoituja testejä vaikuttamatta muiden testeihin. Huolella valittu data on todenmukaisempi, kuin jokaisen testin itselleen rakentama rajattu datamäärä (Crispin ja Gregory, 2009).

Tällä lähestymistavalla on kuitenkin heikkoutensa. Syötedataa voi olla hankala ylläpitää. Kun tarvitaan uusia testiskenaarioita, täytyy tunnistaa toimiva tuotantodata tai luoda tarvittava data ja lisätä se syöteskeemaan. Tietoturvasyistä data tulisi sanitoida ihmisiä tunnistavista ominaisuuksista. Kun uusi taulu tai kolumni lisätään tuotantotietokantaan, täytyy testiskeemat päivittää sen mukaan. Tällöin päivitettävät taulut täytyy valita varovasti. Jos dataa täytyy lisätä testikattavuuden kasvattamiseksi, päivitys vie kauemmin ja koontiprosessin suoritusaika kasvaa. On tärkeää, että automatisoidut koontiprosessit antavat nopeaa palautetta. Pidemmät tietokannan päivitysajat pitkittävät palautteen antoa. Syötedatalla menetetään myös testien

riippumattomuus, jolloin yhden testin epäonnistuessa myös muut testit epäonnistuvat (Crispin ja Gregory, 2009).

Crispinin tiimin jäsenet ajavat graafisen käyttöliittymän testiympäristöjä ja osan funktionaalisista regressiotesteistä kaikkia skeemoja vastaan ja päivittävät datan syötedatalla jokaisella testiajolla. Joskus syötedatan virheellinen päivitys saa testit epäonnistumaan yllättäen. Syötedatan käytön investoinnin palautusarvo oli kuitenkin hyväksyttävällä tasolla. Gregoryn tiimi käyttää syötedataa myös ohjelman keskikerroksen testaamiseen paikallisilla koontiversioilla. Kehityssyklin aikana saadaan nopeaa palautetta. Testiympäristö käyttää kuitenkin tuotantodatan siirrettyä kopiota. Regressiotestit voidaan ajaa vain koontiversion paikallisilla kopioilla (Crispin ja Gregory, 2009).

Kyky mahdollisimman paljon tuotantoversiota muistuttavan järjestelmän testaamiseen on tärkeä useille ohjelmistokehitystiimeille. Automatisoidun regressiotestiympäristön ajaminen tuotantotietokantaa vastaan olisi kuitenkin liian hidasta tarjotakseen hyödyllistä palautetta. Koska uusia tietokannan kopioita tuodaan jatkuvasti ajan tasalla pysymiseksi, ei voida luottaa siihen, että data on pysynyt vakaana prosessin aikana. Kun puhutaan funktionaalisesta testaamisesta, usein tuotantotietokannan kopio on hyödyllisin manuaalisessa tutkivassa testauksessa (Crispin ja Gregory, 2009). Olen itse käyttänyt tuotantotietokannan sanitoitua kopiota manuaalisessa testauksessa. Lisäksi olen hyödyntänyt kopiota, jossa raskain data on jätetty pois nopeamman testauksen mahdollistamiseksi. Kyseisten kopioiden muodostus on automatisoitu niin, että kopio otetaan uusiksi viikoittain, jolloin data pysyy tuoreena. Sen sijaan kopion palautus alkuperäiseen tilaansa tapahtuu automaattisesti joka yö. Näin mahdolliset testauksen aiheuttamat muutokset eivät pilaa koko tietokantakopiota loppuviikon ajaksi. Testiautomaatio sen sijaan hyödyntää pienempää, ainutlaatuista datasarjaa.

Stressitestaus ja suorituskykytestaus ovat automaatiointensiivisiä testausmuotoja, joten ne tarvitsevat ympäristön, joka vastaa läheisesti tuotantoympäristöä antaakseen sellaisia tuloksia, jotka vastaavat todellisuutta. Myös esimerkiksi käytettävyyteen, tietoturvaan ja luotettavuuteen liittyvät testaustoiminnot tarvitsevat tuotantoympäristön kaltaisen järjestelmän. Näissä testeissä automaatio ei kuitenkaan ole yhtä vahvasti mukana (Crispin ja Gregory, 2009).

Tuotantodataa vastaavan testidatan käytöllä on sekä hyviä, että huonoja puolia. Vaikka tuotantotietokanta tarjoaa tarkinta mahdollista testidataa, on sen käyttö kuitenkin kallista ja hidasta. Vaihtoehto on pätevä, jos organisaatiolla on varaa hankkia testaamista varten tarvittava laitteisto ja ohjelmisto useiden tuotantodatan kopioiden tallentamiseksi. Pienten yritysten alhaisemmat resurssit voivat rajoittaa testiympäristöön tallennetun datan määrää. Tällöin tulisi päättää, miten suurta määrää testidataa pyritään tukemaan. Lisäksi tulisi suunnitella, miten kopioida tarpeeksi olennaista dataa, jotta testit olisivat mahdollisimman paljon todellisuutta vastaavia. On myös mahdollista sijoittaa alati halpenevaan laitteistoon, jotta voidaan tukea oikeanlaista tuotantoympäristöä. Muussa tapauksessa testitulokset voivat johtaa harhaan ja testidata joudutaan siivoamaan ennen sen käyttämistä (Crispin ja Gregory, 2009).

Järjestelmissä, joissa suoritetaan datan siirtoa, tulisi aina testata datan siirto oikeaa tietokantaa vasten. Tietokannan päivitysskriptit tulisi ajaa oikeaa dataa ja viimeisintä tietokantaskeeman julkaisuversiota vasten (Crispin ja Gregory, 2009). Esimerkiksi omalla työpaikalla pilvipalvelun päivitys testataan aina käyttämällä oikeaa dataa vastaavaa tietokantaa.

Kun testien tarkoitus ymmärretään, voidaan yksilöllinen tarve arvioida paremmin. Esimerkiksi muistinsisäisiä tietokantoja voidaan käyttää talletettujen proseduurien ja SQL – kyselyjen testaamiseen. Muistinsisäiset tietokannat toimivat juuri kuten oikeat tietokannat, mutta ne nopeuttavat testejä merkittävästi. Kun tarvitaan simuloida todellisuutta vastaavaa tuotantoympäristöä, voidaan tarvittaessa ottaa kopio koko tuotantotietokannasta. Tavoitteena on saada nopeaa palautetta, joten tasapainottelu realististen skenaarioiden testaamisen ja virheiden etsimisen välillä tulisi olla mahdollisimman tehokasta (Crispin ja Gregory, 2009). Seuraava luku käsittelee testiautomaatiotyökaluja ja niiden valintaprosessia.

4 Testiautomaatiotyökalut

Tämä luku käsittelee testiautomaatiotyökaluja. Luvussa 4.1 pohditaan työkalujen arviointia. Luku 4.2 alilukuineen käsittelee testityökalujen valintaa ja testityökalujen tyyppejä. Luvussa 4.3 esitellään esimerkkitapaus kahden testityökalun vertailusta. Luvussa 4.4 taas esitellään esimerkkikokemuksia oman testiautomaatiotyökalun rakentamisesta.

4.1 Työkalujen arviointi

Raulamo – Jurvasen, Mäntylän ja Garousin mukaan yksi merkittävä todettu syy ohjelmistoprojektien viivästymiselle on testityökaluihin liittyvät ongelmat. Testiautomaatio tarvitsee huomattavia investointeja ajankäytön ja kustannusten osalta, vaikka käytettäisiin itse tehtyjä tai avoimen lähdekoodin työkaluja. Raulamo – Jurvanen et al. tekemien havaintojen perusteella oikean testityökalun valinta ei ole ollut helppoa usean toimijan kohdalla (Raulamo-Jurvanen et al., 2017).

Automaatio – ongelmiin on saatavilla useita työkaluja. Työkalua valitessa ei kuitenkaan tulisi vaatia sen kehittyneempää ratkaisua, kuin mihin on tarve. Esimerkiksi Crispinin tiimin jäsenet huomasivat, että taulukkoratkaisu, joka hakee dataa tietokannasta ja suorittaa laskelmat järjestelmästä erillään, on tehokas työkalu niin kehityksen ohjaamiseen, kuin sovelluksen laskelmien verifioimiseen (Crispin ja Gregory, 2009).

Vaikka Crispin ja Gregory suosittelevat hallitsemaan yhden työkalun käyttöä kerrallaan, he toteavat, että miltään työkalulta ei kannata odottaa liikoja. Jokaiseen tarpeeseen on käytettävä sopivaa työkalua. Työkalu, joka toimii parhaiten yksikkötesteille, ei välttämättä ole sopiva funktionaalisten testien automatisointiin. Graafisen käyttöliittymän, performanssin ja tietoturvan testaus voivat kaikki vaatia erilaisen työkalun (Crispin ja Gregory, 2009).

Kun seuraava automaatiohaaste on päätetty, Crispin ja Cregory esittävät, että ensimmäinen askel automaatiotyökalun valintaan on tehdä lista kaikista tarpeista, jotka työkalun pitäisi täyttää. Jos käytössä on jo joitain työkaluja, tulisi uudet työkalut

olla helposti integroitavissa jo olemassa olevaan testaus – ja kehitysinfrastruktuuriin. Valintaan vaikuttavat myös muutkin seikat, kuten työkalun integroinnin mahdollisuus jatkuvaan koontiprosessiin ja laitteiston tuki. Esimerkiksi toisen koontiprosessin pystyttäminen funktionaalisia testejä varten voi vaatia uutta laitteistoa (Crispin ja Gregory, 2009).

Raulamo – Jurvanen et al. esittävät, että työkalun valinta perustuu näkyviin ominaisuuksiin tai intuitiiviseen ymmärrykseen odotettavissa olevista vaikutuksista, eikä niinkään vakiintuneisiin työkalujen arviointiin, formaaleihin kriteereihin tai vaikutusten analysointiin erityisissä projekteissa. Lisäksi projektin koko ja käytetyt kehitysprosessit ovat tärkeitä tekijöitä työkalun valinnassa tuotettavuuden kasvattamiseksi (Raulamo-Jurvanen et al., 2017).

Crispinin ja Gregoryn mukaan on myös pohdittava, miten työkäytäntö tiimin sisällä toimii. Kuka on toteutettavan testityökalun käyttäjä? Onko ohjelmoijien lisäksi muita testitapausten kirjoittajia? Haluavatko tiimin ohjelmoijat sellaisen työkalun, joka tuntuu heidän mielestä hyvälle? Kuka automatisoi ja ylläpitää testejä? Tiimin sisäiset taidot ovat tärkeitä. On pohdittava, miten paljon aikaa kuluu työkalun asennukseen ja opetteluun. Jos sovellus on kirjoitettu Javalla, Javaa skriptaukseen käyttävä työkalu voi olla sopivin. Onko tiimin jäsenillä tai mahdollisesti erillisellä testaustiimillä kokemusta tietyistä työkaluista? Mikäli tiimi alkaa vasta siirtyä ketterään kehitykseen, ja testiautomaatiolle on käytettävissä oma tiiminsä, voi olla järkevää hyödyntää heidän kokemustaan ja jatkaa heidän tuntemiensa työkalujen käyttöä. Työkalutarpeet riippuvat kehitysympäristöstä. On ongelma, jos esimerkiksi web – sovellusta testattaessa testityökalu ei tue SSL:a tai AJAX:ia. Kaikilla testityökaluilla ei voida testata web – palvelusovelluksia. Esimerkiksi sulautetun järjestelmän testaaminen voi taas vaatia täysin toisenlaiset työkalut (Crispin ja Gregory, 2009).

Automatisoitava testaustyyppi on ratkaisevassa asemassa. Tietoturvatestaus vaatii todennäköisesti hyvin erikoistuneita työkaluja. Myös performanssitestaukseen on olemassa useita kaupallisia ja avoimen lähdekoodin työkaluja. Kun yhdestä automaatiohaasteesta pääsee yli, seuraavaan haasteeseen on paljon paremmat valmiudet Crispinin tiimillä kesti muutamia vuosia kehittää ketteriä regressiotestiympäristöjä yksikkötasolla, integrointitasolla ja funktionaalisella tasolla.

Performanssitestaus oli seuraava haaste. Aiemmista automaatio – ongelmista saadut opetukset auttoivat tiimiä tunnistamaan testityökalun vaatimukset paremmin. Näitä vaatimuksia olivat tulosten raportoinnin helppous, yhteensopivuus olemassa oleviin ohjelmistokehyksiin ja käytettävä skriptauskieli (Crispin ja Gregory, 2009).

On mahdollista kirjoittaa lista kaikista tarvittavista työkalun vaatimuksista. Jotkut niistä eivät aina ole yhteensopivia keskenään. Työkalun tarvitsisi esimerkiksi olla tarpeeksi helppo, jotta asiakkaatkin voisivat määritellä testejä, tai testien pitäisi olla helposti automatisoitavissa. Kun vaatimukset on kirjattu ylös, voidaan aloittaa tutkimustyö oikean työkalun löytämiseksi (Crispin ja Gregory, 2009).

Eri työkaluja tarvitaan erilaisiin käyttötarkoituksiin. Crispinin ja Gregoryn mukaan uusien työkalujen toteutus ja työkalujen parhaan käyttötavan opettelu voi käydä nopeasti raskaaksi. Yhtä työkalua voidaan kokeilla tärkeimpään testauskohteeseen kerrallaan. Työkalun käytön tulosten arviointiin tulisi käyttää tarpeeksi aikaa. Jos työkalu todetaan toimivaksi, tulisi sen käyttö hallita täysin ennen siirtymistä seuraavaan testauskohteeseen ja työkaluun. Työkalujen moniajo voi joskus toimia, mutta yleensä uusi teknologia vaatii käyttäjän täyden huomion (Crispin ja Gregory, 2009).

Myös Raulamo – Jurvanen et al. toteavat, että oikean työkalun löytäminen annettuun kontekstiin ja tarkoitukseen on hankala käytännön ongelma. Oikean työkalun valintaprosessi vaatii teoriassa sitä, että etsitään joukko sopivia kandidaattityökaluja ja vertaillaan niitä. Lopuksi valitaan kontekstiin nähden sopivin ja tehokkain työkalu vastaamaan testaustarpeisiin – ja tehtäviin (Raulamo-Jurvanen et al., 2017).

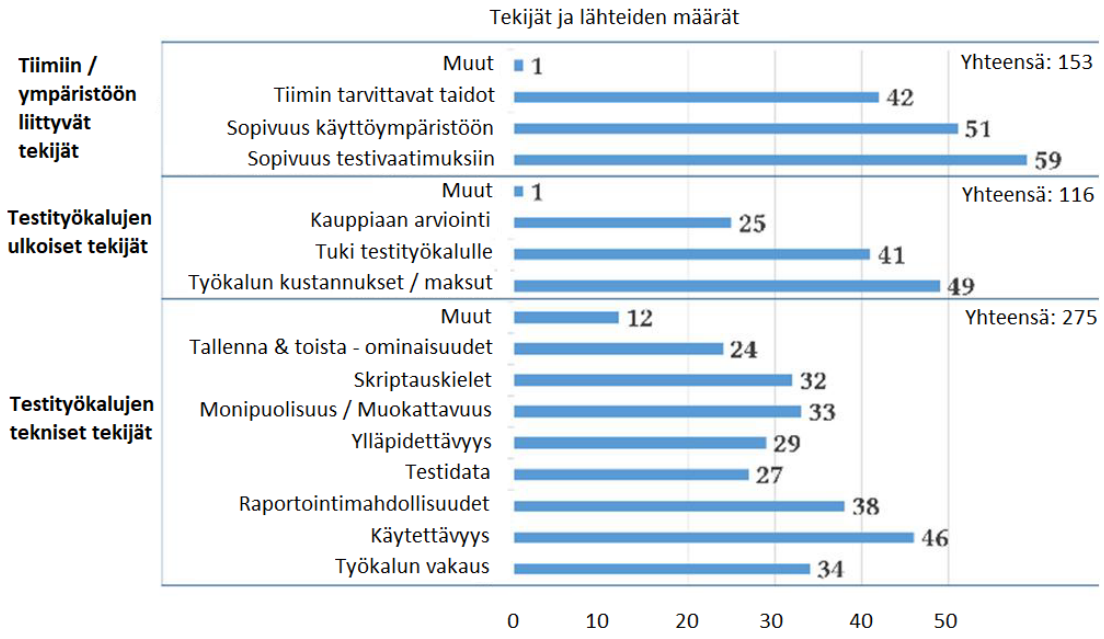
Vaikka testiautomaatiotyökaluja on saatavilla hyvin paljon, tutkimusten ja teollisten galluppien mukaan merkittävin este testiautomaatiolle on oikeiden työkalujen puute. Työkalun valintaprosessin on todettu myös rajoittavan testiautomaatiota, ja markkinoilla saatavilla olevat työkalut eivät ole vastanneet harjoittajien tarpeita. Kun keskitytään tavoitteen saavuttamiseen ja kandidaattityökaluja on vain rajallinen määrä, työkaluja käytetään väärin. Testityökaluihin ja automaatioon liittyviä ulkoisia konsultointipalveluja pidetään kysytyimpinä palveluina ohjelmistotestauksessa (Raulamo-Jurvanen et al., 2017).

Kun päädytään tarpeeseen vastaavaan työkaluun, tulisi ottaa askel taaksepäin ja tarkastella muita tarpeita. Mikä on tiimin seuraava automaatiohaaste? Toimiiko aiempaan tarkoitukseen valittu työkalu myös uudessa haasteessa, vai täytyykö aloittaa uusi testityökalun valintaprosessi (Crispin ja Gregory, 2009)?

Jos päätetään etsiä työkaluja oman organisaation ulkopuolelta, täytyy ensin löytää tarpeeksi aikaa työkalujen kokeilemiseen. Aluksi voidaan suorittaa peruslaatuista tutkimusta esimerkiksi web – hakujen ja työkaluista julkaistujen artikkelien perusteella. Löydetyistä työkaluista voidaan muodostaa lista. Jos tiimi käyttää wiki – ympäristöä tai foorumityökalua, työkaluista voidaan luoda tietoa sisältäviä viestejä ja aloittaa keskustelu työkalujen hyvistä ja huonoista puolista (Crispin ja Gregory, 2009).

Työkalujen arviointiin tulisi varata riittävästi aikaa. Jotkut tiimit pitävät muutaman kuukauden välein sprintin tai iteraation, jossa toiminnallisuuksien toteuttamisen sijaan vähennetään teknistä velkaa, päivitetään työkalujen versioita ja kokeillaan uusia työkaluja. Teknisen velan vähentäminen ja hyvän testi – infrastruktuurin perustaminen kasvattaa tiimin nopeutta tulevaisuudessa ja tarjoaa lisää aikaa tutkivaan testaukseen. Jos työkalujen päivittämiseen tai ohjelmakoodin ylläpidettävyyden lisäämiseen ei löydy aikaa, alkaa tekninen velka hidastaa tiimiä merkittävästi (Crispin ja Gregory, 2009).

Raulamo – Jurvanen et al. etsivät harmaasta kirjallisuudesta oikean testityökalun valitsemiseksi liittyvien kriteerien esiintymisiä (Kuva 2). Kriteerit luokiteltiin kolmeen osaan: testityökalujen tekniset tekijät, testityökalujen ulkoiset tekijät ja tiimiin ja ympäristöön liittyvät tekijät. Tulosten perusteella suurin esiintymismäärä oli tiimiin ja ympäristöön liittyvillä kriteereillä. Sopivuus testivaatimuksiin oli eniten esiintynyt kriteeri, mikä sopii jo aiemmin tutkielmassa todettuihin seikkoihin. Seuraavaksi eniten esiintyneitä kriteerejä olivat sopivuus käyttöympäristöön, testityökalun kustannukset ja käytettävyys (Raulamo-Jurvanen et al., 2017).



Kuva 2. Harkittavia kriteerejä oikean testityökalun valitsemiseksi (Raulamo-Jurvanen, Mäntylä, Garousi, 2017).

Kun on muodostettu lista vaatimukset täyttävistä työkaluista, tulisi vaihtoehdot karsia yhteen tai kahteen. Jäljelle jäävien työkalujen käyttöä voidaan tällöin opetella, jotta niitä päästään kokeilemaan käytännössä. Käytännön kokeilu voi olla yksinkertainen, mutta edustava skenaario. Tuloksia voidaan arvioida vaatimuksia vasten ja pitää loppukatselmointeja hyvien ja huonojen puolien kartoittamiseksi. On pohdittava, mitä resursseja työkalun implementointiin ja käyttöön tarvitaan, mitä vaikutuksia työkalulla on tiimin tuotettavuuteen ja nopeuteen, mitä riskejä työkalusta koituu, ja mitä työkalulla lopulta kyetään tekemään (Crispin ja Gregory, 2009).

Kun paras työkalukandidaatti on valittu ja sen kokeilemiseen on varattu tarpeeksi aikaa, on oltava varmoja siitä, että kaikkia kriittisiä toimintoja tulee kokeiltua. Jos esimerkiksi sovellus käyttää paljon AJAX:ia, työkalulla täytyisi pystyä automatisoimaan vastaavat testit. Jälkikatselmoinneissa pohditaan, mikä työkalussa toimi ja mikä ei. Täytyy kuitenkin varautua siihen, että työkalu ei vastaa halutunlaista, jolloin täytyy etsiä uusi työkalu. Vaikka työkalun parissa olisi jo käytetty paljon aikaa, ei ole perustetta jatkaa enää saman, sopimattoman työkalun parissa (Crispin ja Gregory, 2009). Seuraavaksi käsitellään testityökalujen valintaa.

4.2 Työkalujen valinta

Valittavissa olevia testityökaluja on nykyään jo suuri, alati kasvava joukko. Hyviä vaihtoehtoja voivat olla niin itse tehdyt, avoimen lähdekoodin, kaupalliset, tai näitä tekijöitä yhdistävät työkalut. Koska valinnanvaraa on paljon, on olennaista tietää, mistä etsiä testityökaluja ja löytää aikaa selvittää, täyttävätkö työkalut tarvittavat vaatimukset Voi olla vaikeaa arvioida jokaisen potentiaalisen ratkaisun investoinnin palautusarvo, mutta iteratiivinen ratkaisu työkalujen arviointiin auttaa oikean työkalun löytämisessä (Crispin ja Gregory, 2009). Ensimmäinen vaihtoehto on rakentaa testityökalu itse.

4.2.1 Itse tehdyt työkalut

Joskus itse tehty testityökalu saattaa olla paras valinta. Tällainen tilanne voi olla esimerkiksi se, kun sovellus tarjoaa ainutlaatuisia testaushaasteita, kuten sulautetuissa ohjelmistoissa tai ulkopuolisiin järjestelmiin integroiduissa sovelluksissa. Jos tiimin jäsenillä on taitoja, aikaa ja taipumusta kirjoittaa oma testikehys tai rakentaa testikehys jo olemassa olevan avoimen lähdekoodin työkalun päälle, on itse tehty työkalu hyvä valinta (Crispin ja Gregory, 2009).

Ketterän kehityksen onnistumiseen vaikuttaa se, miten moni ohjelmoija on testivetoinen. Nykyään saatavilla olevat kehitystyökalut – ja kielet tekevät automaatiokehyksistä helpompia rakentaa. Ruby, Groovy, Rails ja moni muu kieli ja kehys ovat hyödyllisiä automaatiossa. On mahdollista hyödyntää valmiita avoimen lähdekoodin työkaluja, kuten Fit ja HTMLUnit, ja rakentaa kustomoidut kehykset näiden päälle (Crispin ja Gregory, 2009).

Itse tehdyt työkalut ovat käytettävyydeltään erittäin ystävällisiä ohjelmoijille. Jos tiimi kirjoittaa omia automaatiokehyksiä, työkalut voidaan kustomoida tarkasti kehitys – ja asiakastiimien tarpeisiin ja integroida ne valmiisiin koontiprosesseihin ja infrastruktuuriin. Myös työkalun ajo ja tulosten tulkinta voidaan tehdä helpoksi (Crispin ja Gregory, 2009).

Itse tehtyjen työkalujen toteutus ei kuitenkaan ole ilmaista. Pienellä tiimillä ei välttämättä ole resursseja kirjoittaa ja tukea näitä työkaluja tuotantokoodin kehityksen

ohella. Suuri, ainutlaatuisia vaatimuksia omaava yritys voi saada kasattua tiimin automaatioon erikoistuneita tekijöitä, jotka voivat tehdä yhteistyötä testaajien, asiakkaiden, ohjelmoijien ja muiden tahojen kanssa. Jos tarpeet ovat niin ainutlaatuisia, ettei mikään valmis työkalu tue niitä, työkalujen rakentaminen itse voi olla ainut vaihtoehto (Crispin ja Gregory, 2009). Seuraavaksi käsitellään avoimen lähdekoodin työkaluja.

4.2.2 Avoimen lähdekoodin työkalut

Monet omat työkalunsa tehneet tiimit ovat julkaisseet työkalut avoimen lähdekoodin yhteisölle. Koska näiden työkalujen kirjoittajina ovat toimineet testivetoiset kehittäjät, työkalut ovat yleensä kevyitä ja sopivia ketterään kehitykseen. Moni näistä työkaluista on toteutettu testivetoisen kehityksen mukaisesti ja testiympäristöt ovat ladattavissa lähdekoodin mukana tehden kustomoinnista helpompaa ja turvallisempaa. Työkaluissa on sekä ohjelmoijille että testaajille hyödyllisiä ominaisuuksia. Näiden työkalujen hinta on yleensä huokea, tosin jokaisen työkalun ostohinta on vain osa sen todellisesta kustannuksesta (Crispin ja Gregory, 2009).

Kaikki avoimen lähdekoodin työkalut eivät ole hyvin dokumentoituja, ja kouluttaminen työkalujen käyttöön voi olla ongelmallista. Useissa konferensseissa ja käyttäjäryhmien tapaamisissa voidaan kuitenkin nähdä seminaareja ja tutoriaaleja, jotka opastavat työkalujen käyttöön. Joillekin avoimen lähdekoodin työkaluille on kuitenkin saatavilla erinomaisia käyttöohjeita ja koulutuksia (Crispin ja Gregory, 2009).

Avoimen lähdekoodin ratkaisua harkitessa tulisi etsiä aktiivista kehittäjä – ja käyttäjäyhteisöä. Tällaisella yhteisöllä voi olla aktiivinen sähköpostilista, ja yhteisö voi julkaista uusia ominaisuuksia usein. Joillakin työkaluilla voi olla parempi tuki ja vastausaika uusien löydettyjen virheiden selvittämiseen kuin kaupallisilla työkaluilla. Tämä johtuu siitä, että työkalujen kirjoittajat ovat samalla niiden käyttäjiä, joten he tarvitsevat samoja ominaisuuksia omien tuotteidensa testaamiseen (Crispin ja Gregory, 2009). Seuraava luku käsittelee kaupallisia testityökaluja.

4.2.3 Kaupalliset työkalut

Kaupalliset työkalut koetaan usein varmana valintana. On hankala kritisoida ketään tunnetun, vuosikausia käytetyn työkalun valinnasta. Kaupallisille työkaluille on todennäköisesti tarjolla käyttöohjeet, tuki ja koulutus. Testaajille ja ei – teknisille käyttäjille työkalun käytön aloittaminen voi olla helpompaa. Jotkut työkaluista ovat vankkoja ja täynnä eri toiminnallisuuksia. Monella yrityksellä on jo vähintään yksi tällainen työkalu käytössä, kuten myös tiimi työkaluun erikoistuneita käyttäjiä (Crispin ja Gregory, 2009).

Vaikka kaupalliset työkalut muuttuvat ajan saatossa, ovat ne yleensä olleet vaikeita ohjelmoijille. Työkaluissa käytetään usein skriptauskieliä, joiden käytön opetteluun ohjelmoijat eivät halua käyttää aikaa. Työkalut ovat usein myös raskaita. Testiskriptit voivat olla hauraita ja kalliita ylläpitää, koska pienetkin sovelluksen muutokset voivat rikkoa testit. Monet näistä työkaluista ovat nauhoitusskriptejä peräkkäistä toistoa varten. Nauhoitus – ja toistoskriptit ovat pahamaineisen hintavia ylläpidettävyyden näkökulmasta (Crispin ja Gregory, 2009).

Elizabeth Hendrickson on esittänyt, että tällaiset erikoistuneet työkalut voivat luoda tarpeen testiautomaatiospesialisteille. Tämä seikka voi toimia ketteriä tiimejä vastaan. Työkalujen täytyisi olla testivetoisia eikä testivetoisuuden vastakohta. Testityökalut eivät tulisi olla muutoksen tiellä. Jos organisaatio omaa kaupalliseen työkaluun erikoistuneita työntekijöitä ja käyttää sellaista työkalua, jota vain osa tiimeistä kykenee käyttämään, on kaupallinen työkalu pätevä ratkaisu (Crispin ja Gregory, 2009).

Nykyään on saatavilla yhä parempia funktionaalisia testityökaluja ja ohjelmointiympäristöjä. Ne helpottavat testien ylläpidettävyyteen liittyvien tehtävien suorittamista ominaisuuksilla, kuten globaali tekstin haku ja korvaus. Twist on esimerkki työkalusta, joka on toteutettu Eclipse – ohjelmointiympäristön lisäosien kokoelmana. Sen käytön etuja ovat tehokkaat muokkaus – ja refaktorointiominaisuudet (Crispin ja Gregory, 2009).

Tehokkailla ketterillä testiautomaatiotyökaluilla on useita ominaisuuksia. Työkalujen tulisi tukea testiautomaatiotyön välitöntä aloittamista käyttäen testivetoista lähestymistapaa. Työkalut erottavat varsinaisen testin sen toteutuksen

yksityiskohdista. Ne tukevat ja kannustavat hyviin ohjelmointikäytäntöihin testiautomaation koodin osalta. Työkalut tukevat testiautomaatiokoodin kirjoittamista oikeilla kielillä ja oikeilla ohjelmointiympäristöillä. Lisäksi ne tukevat yhteistyötä eri toimijoiden ja tahojen kesken (Crispin ja Gregory, 2009). Seuraavassa luvussa on esimerkkitapaus testityökalujen vertailusta.

4.3 Testityökalujen vertailu – Watir ja Selenium

Tämä luku käsittelee esimerkkitapausta testityökalujen vertailusta. Vertailun kohteena olivat Watir ja Selenium. Rigzin Angmo ja Monika Sharma ovat tutkineet web – pohjaisten testityökalujen valintaa. Tutkimuksessaan Angmo ja Sharma pyrkivät löytämään parhaan työkalun Selenium – testityökaluohjelmistosta ja vertailla sitä muihin vastaaviin työkaluihin. Tutkimuksen tarkoituksena oli tehdä suorituskykyarviointi Seleniumille pohjautuen esimerkiksi suoritusnopeuteen, tallennukseen ja toistoon, raportin luontiin, selainyhteensopivuuteen, alustayhteensopivuuteen ja lokalisaatiotukeen. Tehtävään sopivimman työkalun löytämiseksi täytyi harkita useita seikkoja, kuten työkalun integrointikykyä, hintaa ja suorituskykyä. Vertailun kohteena Angmo ja Sharma päättivät käyttää Watir-WebDrive – työkalua, koska se on moderni versio Watir API:sta, joka pohjautuu Selenium webdriveriin (Watir Contributors, 2017). Tätä tarkoitusta varten kullekin työkalulle tehtiin skripti, joka automatisoi Googlen verkkohakua ja suoritti testityökalujen suorituskykyarviointia (Angmo ja Sharma, 2014).

Selenium ei ole yksittäinen työkalu, vaan avoimen lähdekoodin työkalusarja selainten automatisointiin ja web – sovellusten testaamiseen useilla eri alustoilla (Selenium Contributors). Se koostuu neljästä eri työkalusta: Selenium IDE, Selenium RC, Selenium WebDrive ja Selenium Grid. Selenium tukee useita selaimia, alustoja ja käyttöjärjestelmiä. Sitä voidaan ohjata useilla eri ohjelmointikielillä ja testikehyksillä. Selenium Framework tarjoaa standardoidun tavan kirjoittaa testejä Seleniumilla. Automaatiokehys taas lisää automaation tehokkuutta vähentämällä ohjelmointityötä alkupuolella. Kehystä käytetään useilla ohjelmointikielillä, kuten Javalla, .NET:illa ja PHP:lla, kirjoitettujen web – sovellusten automaatiotestaukseen. Se tarjoaa alustan datapohjaisten kehysten suorittamiseen käyttämällä templaatteja. Tarvittaessa

Selenium – kehys auttaa yrityksiä nopeuttamaan testaamista testien suunnittelutasolla pitäen samalla automaatiosarjan joustavana (Angmo ja Sharma, 2014).

Watir on Ruby – kielen kirjasto selaimen automatisointiin. Se on tehokas avoimen lähdekoodin automaattinen testityökalu, jonka suosio on kasvanut. Se ei ole tallenna – toista – työkalu kuten Selenium IDE, vaan se käyttää Ruby – ohjelmointikieltä. Watir – projekti koostuu useista pienemmistä projekteista, kuten Watir Classic, Watir Webdriver ja Watirspec (Watir Contributors, 2017). Watir Classic käyttää sisäänrakennettua Ruby – kielen OLE – yhteensopivuutta ja kykenee ajamaan Internet Explorer – selainta ohjelmallisesti. Watir Webdriver on Watir API:n uusin versio, joka pohjautuu Selenium 2.0 – versioon. Watirspec on Watir API:n suoritettava spesifikaatio, kuten Ruby – kielen RubySpec (Angmo ja Sharma, 2014).

Angmon ja Sharman suorittaman suorituskykyarvioinnin kriteerien parametreja oli seitsemän kappaletta (Taulukko 1). Parametreina olivat suoritusnopeus, tallennus – ja toisto, selainyhteensopivuus, alustayhteensopivuus, tuetut ohjelmointikielet, testitulosten raportointi ja tulevaisuus käytettävyyden kannalta. Suoritusnopeuteen käytettiin seuraavaa kaavaa: *Kokonaisaika / n* millisekunneissa, missä *n* on testin aikana avattujen ikkunoiden määrä ja *kokonaisaika* on lopetusajan ja aloitusajan erotus (Taulukko 2). Kokonaisaika saadaan kaapattua web – sovelluksen testaamiseen tarkoitetulla työkalulla ohjelman suorituksen alusta sivun avaamiseen saakka. Suorituskykyparametrien arvioimiseen Angmo ja Sharma käyttivät arvosanaa yhdestä viiteen, missä ”1” tarkoittaa erittäin huonoa, ”2” melko huonoa, ”3” keskivertoa, ”4” melko hyvää ja ”5” erittäin hyvää (Angmo ja Sharma, 2014).

Taulukko 1. Suorituskykyarvioinnin kriteerit.

Numero	Parametri	Selite
1.	Suoritusnopeus	Nopeus, millä laskennallinen laite voi suorittaa ohjeita. Yksikkönä millisekunti.
2.	Tallennus – ja toisto	Miten helposti testitapaus – tai sarja voidaan tallentaa ja toistaa.
3.	Selainyhteensopivuus	Testien tuki web – selaimille.
4.	Alustayhteensopivuus	Erilaisten alustojen tuki testien suorittamiseksi.
5.	Tuettu ohjelmointikieli	Testien tukema kieli.
6.	Testitulosten raportointi	Miten tulos esitetään.
7.	Tulevaisuus käytettävyyden kannalta	Kuinka helppoa työkalun käyttö on nyt ja tulevaisuudessa.

Ensin Angmo ja Sharma kirjoittivat testitapauksen Googlen toiminnallisuuden automatisoimiseksi sopivassa kehitysympäristössä käyttäen Selenium – ohjelmistoa. Testissä käytettiin Firefox – selainta ja tiettyä työkalusarjaa. Selenium RC:n ja Selenium Gridin tapauksessa täytyi ajaa Selenium – palvelinohjelma ennen testitapausten suorittamista. Toinen askel oli selvittää Selenium – ohjelmiston paras työkalu tekemällä suorituskykyarviointi ja harkitsemalla muita seikkoja. Lopuksi oli

tehtävä varsinainen suorituskykyarviointi ja vertaileva tutkimus tiettyjen kriteerien perusteella (Taulukko 3) (Angmo ja Sharma, 2014).

Taulukko 2. Suoritusnopeuden arviointi.

Työkalut	**Selenium IDE**	**Selenium RC**	**Selenium Webdriver**	**Selenium Grid**
Suoritusnopeus (ms)	3540	43396	79570	93185

Taulukko 3. Selenium – testisarjan suorituskykyarviointi.

Ominaisuudet	**Selenium IDE**	**Selenium RC**	**Selenium Webdriver**	**Selenium Grid**
Suoritusnopeus	5	3	2	1
Tallennus – ja toisto	5	0	0	0
Selainyhteensopivuus	1	5	5	5
Alustayhteensopivuus	5	5	5	5
Summa	16	13	12	11

Selenium – ohjelmiston osalta valittiin Selenium Webdriver. Vaikka Selenium IDE sai parhaimman arvosanan suorituskykyarvioinnissa, sitä ei valittu, koska se on vain Firefox – selaimen lisäosa. Selenium RC taas käyttää Selenium RC – palvelinta testien ajamiseen. Myöskään Selenium Grid ei sopinut käytettäväksi, koska sitä käytetään rinnakkaisiin testeihin ajamalla keskussolmu ja palvelinsolmu jokaisen testin alussa. Sen sijaan Selenium Webdriver ei tarvitse palvelinta, vaan se käyttää selainta suoraan. Selenium Webdriver todettiin siis parhaaksi vaihtoehdoksi lisätutkimusten osalta. Angmo ja Sharma arvioivat Selenium Webdriveria ja Watir Webdriveria keskenään (Taulukko 4). Watir Webdriver on uusi versio Watir API:sta, joka pohjautuu

Seleniumiin. Se on toteutettu kietomalla Watir – API Selenium 2.0 – API:n ympärille tehden testiautomaatiotyöstä helpompaa ja helposti ymmärrettävää (Angmo ja Sharma, 2014).

Taulukko 4. Watir Webdriverin ja Selenium Webdriverin suoritusnopeuden arviointi.

Työkalu	**Watir Webdriver**	**Selenium Webdriver**
Suoritusnopeus	8860	7800

Taulukko 5. Watir Webdriverin ja Selenium Webdriverin suorituskykyarviointi ja vertaileva tutkimus.

Ominaisuudet	**Watir Webdriver**	**Selenium Webdriver**
Suoritusnopeus	3	4
Tallennus – ja toisto	0	0
Selainyhteensopivuus	5	5
Alustayhteensopivuus	5	5
Tuki ohjelmointikielille	1	5
Testitulosten raportointi	2	4
Tulevaisuus käytettävyyden kannalta	3	4
Summa	19	27

Tulosten perusteella on selkeää, että Selenium Webdriver on Watir Webdriveria parempi käytettyjen parametrien perusteella (Taulukko 5). Työkalujen analysoinnin pohjalta voidaan todeta, että Watir Webdriver soveltuu paremmin tietynlaisiin

tilanteisiin, mutta Selenium Webdriver on parempi valinta esimerkiksi testitulosten raportointiin ja käytettäessä täsmäkieliä (Angmo ja Sharma, 2014).

Työkalun valinnassa on tärkeämpää pohtia eri parametreja pelkän kustannuksen sijaan. Selenium ja Watir ovat molemmat suunniteltu erityisesti web – testaamiseen. Watir Webdriver on hyvin vakaa testityökalu, joka käyttää hyvää selainautomaatiomoottoria eli Webdriveria puhtaalla Ruby API:lla. On kuitenkin selvää, että Selenium Webdriver on Watir Webdriveria parempi työkalu useammassa tilanteessa (Angmo ja Sharma, 2014).

Suorituskykyarviointi tehtiin vain yhdellä selaimella. Lisätutkimuksena web – sovellusta voitaisiin testata useammalla eri selaimella. Web – sovelluksia voidaan testata myös muilla kielillä, kuten .NET, PHP ja Python. Selenium IDE:n tapaan myös Watirille voidaan kehittää lisäosatoiminnallisuus. Watiriin on mahdollista lisätä myös raportointimahdollisuus (Angmo ja Sharma, 2014). Seuraavassa luvussa käsitellään oman testityökalun rakentamista.

4.4 Oman testityökalun rakentaminen

Kuten tutkielmassa on tullut ilmi, on olemassa useita kaupallisia ja avoimen lähdekoodin testityökaluja, joita eri organisaatiot voivat hyödyntää. Yksi vaihtoehto on kuitenkin rakentaa oma testityökalu. Miksi kuitenkin lähteä riskialttiille tielle rakentaa oma työkalu? Ramler ja Putschögl ovat tutkineet asiaa artikkelissaan.

Tärkein syy oman testityökalun rakentamiselle oli tekninen. Ramlerin ja Putschöglin testaama järjestelmä oli osa isoa koneistojärjestelmää, johon kuului reaaliaikainen koneistonohjaus – ohjelma. Tämä ohjelma oli yhteydessä muun muassa erilaisiin sensoreihin ja ohjelmassa oli graafinen käyttöliittymä. Tavalliset testityökalut eivät tukeneet kumpaakaan alijärjestelmää mukana olevien erityisten teknologioiden vuoksi. Esimerkiksi koko käyttöliittymä oli toteutettu ohjaimilla, jotka olivat osa sisäistä käyttöliittymäkehystä, joka oli optimoitu kosketusrajapintoja varten. Tavalliset testityökalut eivät kyenneet tunnistamaan kyseisiä ohjaimia eivätkä pystyneet olemaan vuorovaikutuksessa graafisen käyttöliittymän kanssa. Reaaliaikainen ohjelmisto kykeni toimimaan vain sovelluskohtaisilla teknologioilla suljetussa ympäristössä.

Ohjelmisto oli kuitenkin suunniteltu huomioimalla testattavuus. Kustomoidut käyttöliittymäohjaimet tarjosivat sisäisen rajapinnan asetuksien ja lokiarvojen muuttamiseen (Ramler ja Putschögl, 2013).

Teknisten ongelmien lisäksi Ramler ja Putschögl huomasivat, että testiautomaatio nojasi vahvasti olemassa oleviin prosesseihin, organisaatiorakenteisiin ja myös mukana olleiden kokemukseen ja osaamiseen. Testaajat, eli työkalun käyttäjät, olivat palvelinasiantuntijoita, ja heillä oli vuosien kokemus kehitettyjen koneiden käytöstä. He olivat vastuussa koko tuotteeseen kohdistuvasta talon sisäisestä järjestelmän hyväksymistestauksesta, johon kuului ohjelmiston ja monen muun työn osa – alueen testaus. Ohjelmistotestaus tehtiin yleensä manuaalisesti yhdessä laitteisto – ominaisuuksien kanssa. Koska käyttäjillä ei ollut erityistä taustaa testiautomaation tai ohjelmistokehityksen parissa, työkalun käyttötuen perustaminen sisälsi myös tarvittavan testiautomaatiotietouden nostamisen ja olemassa olevan testienhallinnan ja tiketti – infrastruktuurin mukautumisen. Tämä paransi ohjelmiston testattavuutta ja mukautti testiprosessia. Oman työkalun kehittäminen tarjosi erinomaisen tilaisuuden tukea käyttäjien yksilöllisiä tarpeita ja organisaation vaatimuksia (Ramler ja Putschögl, 2013).

Ramlerin ja Putschöglin oma ratkaisu tarjoaa useita samankaltaisia ominaisuuksia, jotka löytyvät suosituista graafisen käyttöliittymän testityökaluista (Li, Wu, 2004). Pääkomponentti tukee testiskriptien tallennusta, muokkaamista ja toistoa. Skriptit ovat hyvin yksinkertaisia. Jokainen rivi vastaa tiettyä toimintoa, kuten klikkausta, näppäimen lyöntiä tai sellaisten parametrien tarkistusta, jotka määrittelevät käyttöliittymän kontrollit, syötteet ja odotetut arvot. Syötearvot voidaan lukea parametritiedostoista. Testituloslokit voidaan tuoda testien hallintatyökalulla. NUnit – tuki helpottaa koonti – integraatiota, testiympäristöjä, monitasoasetusta ja kustomoitua ohjelmakoodia. Loppujen lopuksi työkalusta kehitettiin täysin yhteensopiva organisaatioiden ja testaajien yksittäisten vaatimusten kanssa. Työkalu arvioitiin automatisoimalla yhdessä testaajien kanssa noin yksi kolmannes olemassa olevista testitapauksista onnistuneesti (Ramler ja Putschögl, 2013).

Oman työkalun rakentamisessa oli useita hyötyjä. Projekti ei ollut pelkästään työkalun kehitysprojekti, vaan myös projekti kehittää tietotaitoa. Projekti auttoi saavuttamaan

kriittistä osaamista ja ymmärrystä testiautomaation erikoisuuksista vaikeiden käytännön kokemusten kautta. Sen sijaan, että työkalu olisi valittu tiettyjen valintakriteerien mukaan, oman ratkaisun ja tarpeellisen tietotaidon kehittäminen vähensi riskiä päätyä sellaisen ratkaisun pariin, joka epäonnistuisi täyttämään käyttäjän todelliset tarpeet Vaikka monet kehitetyistä ominaisuuksista olivat samankaltaisia olemassa olevien ratkaisujen kanssa, työkalu tarjosi yksilöllisen yhdistelmän, joka vastasi testaajan päivittäisen työn käytännöllisiä vaatimuksia. Vain osa kehitetyistä ominaisuuksista, kuten tuki parametritiedostoille, jäi vain vähän käytetyksi tai kokonaan käyttämättä (Ramler ja Putschögl, 2013).

Työkalu rakennettiin tiiviissä yhteistyössä testaajien ja ohjelmistokehittäjien kanssa. Yhteistyö kannusti nopeaan palautteeseen ja toi muutoksia ja käytettävyysparannuksia pienissä osissa. Mikäli testaajat lopulta kohtasivat teknisiä ongelmia, nopea ratkaisu tai väliaikainen korjaus kyettiin tekemään nopeasti. Oman työkalun käyttö ei pakottanut työkalukohtaiseen työkulkuun tai kaupallisen työkaluperheen käyttöön, vaan se johti prosessien, testienhallinnan lähestymistavan ja infrastruktuurin peräkkäisiin säätöihin ja parannuksiin (Ramler ja Putschögl, 2013).

Myös Ramler ja Putschögl toteavat, että testiautomaatio on merkittävä investointi. Kaner et al. olivat huomanneet, että testiautomaatioon kuuluva työsarka on usein hyvin suuri. Testityökalujen hinta on vain yksi osa kustannuksista. Henkilökunnan koulutuksesta ja automatisoitujen testitapausten toteutuksesta koituvat kulut ovat huomattavasti suurempia kustannuksia (Ramler ja Putschögl, 2013).

Ramlerin ja Putschöglin tapauksessa testiautomaatioprojekti vei kokonaisuudessaan 14 kuukautta ja 17 henkilötyökuukautta. Työkalun tuen kehitys on esimerkki yhdestä aktiviteetista, joka vei noin puolet kokonaistyöstä. Useat muut läheisesti toisiinsa yhteydessä olevat aktiviteetit olisivat muutenkin olleet välttämättömiä riippumatta siitä, tehtiinkö päätös rakentaa oma työkalu vai valita jokin valmis työkalu. Näitä aktiviteetteja olivat käyttäjän tarpeiden ja projektin kontekstin ymmärtäminen ja tutkiminen, työkalun toiminnallisuuden kehittäminen ja säätäminen, testaajien koulutus ja tukeminen automaatiossa ja työkalun käytössä, yhteistyö testiskriptien toteutuksessa, katselmoinnissa ja ylläpidossa, testitulosten arviointi ja visualisointi, testi – infrastruktuurin ja prosessien muodostaminen ja säätö, virtuaalisten

testiympäristöjen luominen testien suorittamiseen ilman fyysisen laitteiston tarvetta, järjestelmän testattavuuden parantaminen, virheiden löytämisen tukeminen toistamalla vain rasituksen alla esiintyviä kriittisiä virheitä ja projektin hallinta (Ramler ja Putschögl, 2013).

Työkalun rakentamisessa oli useita haasteita ja avonaisia ongelmia. Automaatiota rajoittivat vaiheet, jotka vaativat vuorovaikutusta laitteiston kanssa, kuten fyysisen napin painaminen. Osittain automatisoidut testit olivat mahdollisia, missä esimerkiksi skripti avasi dialogin, joka pyysi käyttäjältä manuaalista syötettä. Hienostuneempi ratkaisu sisälsi pienen mekaanisen robotin, joka painoi nappeja skriptin ohjaamana. Testejä suoritettaessa koneisto oli suojattava laserverholla turvallisuussyistä (Ramler ja Putschögl, 2013).

Graafisen käyttöliittymän ohjaimien testattavuudessa oli useita pieniä, mutta ärsyttäviä ongelmia. Valintaruudun klikkaus vaihtoi valintaruudun tilan sen sijaan, että se asetettaisiin johonkin tiettyyn arvoon. Listan alkioiden valinta indeksin perusteella muuttui usein epäluotettavaksi, kun alkioiden lukumäärä muuttui. Testityökalun täytyi toteuttaa sopivat ratkaisut ongelmiin (Ramler ja Putschögl, 2013).

Myös skriptien ylläpito oli ongelmallista. Yhdessä skriptissä oli keskimäärin 42,9 toimintoa ja pitkissä skripteissä jopa yli 200 toimintoa. Näin ollen tarvittiin tukea skriptien ymmärtämiseen, muokkaamiseen ja debuggaamiseen. Ramler ja Putschögl esittelivät rakenteellisia apukeinoja, kuten samaan käyttötapausskenaarion askeleeseen kuuluvien toimintojen ryhmittely ja kuvakaappausten lisääminen tuloslokiin testien epäonnistuessa (Ramler ja Putschögl, 2013).

Testiympäristön pystytyksessä huomattiin usein, että yhdellä koneella läpimenevät testiskriptit epäonnistuivat, kun ne siirrettiin toiseen samanlaiseen koneeseen, ja päinvastoin. Tämän ongelman aiheutti herkkyys ympäristön konfiguroinnin yksityiskohtiin. Näitä yksityiskohtia olivat esimerkiksi käyttöjärjestelmän kieliasetukset ja edellisestä asennuksesta jääneet tiedostot (Ramler ja Putschögl, 2013).

Viimeinen ongelmakohta oli graafisen käyttöliittymän tilan alustaminen. Ohjelmisto ylläpiti kirjaa lukuisista sisäisistä tiloista ja käyttäjän syötteistä, kuten valituista

asetuksista tai viimeaikaisista ladatuista tiedostoista. Näin ollen kaikki yksittäisen testin tekemät muutokset piti alustaa, jotta seuraava testi voitiin aloittaa puhtaalta pöydältä. Osittaisten muutosten palautus oli tarpeen myös silloin, kun testiajo keskeytyi virheen vuoksi. Ramlerin ja Putschöglin ratkaisuna oli kokonaisvaltainen asetusskripti, joka suoritettiin ennen jokaista testiä. Skripti asetti kaikki käytetyt graafisen käyttöliittymän kontrollit oletusasetuksiin. Skriptin yhdenmukaisuuden ylläpitäminen sarjalla jatkuvasti muuttuvia testejä osoittautui testiautomaation virheherkimmäksi tehtäväksi (Ramler ja Putschögl, 2013).

Oman työkalun rakentaminen oli Ramlerin ja Putschöglin mielestä menestys, ja he rakentaisivat työkalun uudelleen tilanteessa, joka olisi alkuehtoineen samankaltainen kuin projektin alussa. Kun Ramler ja Putschögl kohtasivat projektin alkuvaiheessa useita teknisiä ja organisaatiokohtaisia haasteita, yksikään mukana ollut osapuoli ei pitänyt testiautomaatiovisiota realistisena vaihtoehtona. Testaajien tarpeita tukevan mukautetun ratkaisun kehittäminen samalla, kun muodostettiin tarvittavaa osaamista automatisoitujen testien luomiseen ja ylläpitoon, oli ainutlaatuinen vahvuus, joka auttoi muuttamaan vision todellisuudeksi (Ramler ja Putschögl, 2013).

Ramler ja Putschögl eivät kuitenkaan tekisi enää samaa ratkaisua. Työkalu palvelee yhä tarkoitustaan hyvin ja Ramler ja Putschögl ovat saaneet tarpeeksi kokemusta suunnitella järjestelmät tulevaisuudessa niin, että ne olisivat testattavia tavallisilla työkaluilla. Ramler ja Putschögl olivat kuitenkin arvioineet kaupallisia työkaluja toisessa projektissa ja pettyneet näiden työkalujen ominaisuuksiin ja teknisiin kyvykkyyksiin (Ramler ja Putschögl, 2013). Seuraavassa luvussa perehdytään testiautomaation toteutukseen ja hallintaan.

5 Testiautomaation toteutus ja hallinta

Tämä luku käsittelee testiautomaation toteutusta ja hallintaa. Luku 5.1 käsittelee automaation toteutusosiota ja luku 5.2 automatisoitujen testien ja testitulosten hallintaa.

5.1 Automaation toteuttaminen

Testityökaluja arvioidessa tulisi pohtia, miten nopeasti tärkein automaatiotarve tulisi ratkaista. Mistä voidaan saada tukea automaatiotyön toteuttamiseen? Millaista koulutusta tiimi tarvitsee ja miten paljon aikaa koulutukseen voidaan käyttää? Miten nopeasti työkalun käyttö täytyisi aloittaa? Työkaluja etsiessä tulisi miettiä kaikkia näitä rajoituksia. Joskus voidaan joutua tyytymään vähemmän vakaaseen työkaluun, jotta tärkeä automaatiotyö voidaan aloittaa nopeammin. Mikään ei ole kuitenkaan pysyvää. Automaatiotyö voidaan rakentaa askel askeleelta. Moni tiimi kokee epäonnistumisia ennen kuin löydetään oikean yhdistelmä työkaluja, taitoja ja infrastruktuureja (Crispin ja Gregory, 2009).

Hyvä oliopohjainen suunnittelu ei ole ainoa avain kannattavan, ylläpidettävän testiautomaatioympäristön rakentamiseen. Testit ovat myös ajettava usein, jotta tiimi saa tarvitsemaansa palautetta. Valitut työkalut täytyy integroida osaksi koontiprosessia. Helposti tulkittavat tulokset tulisi olla automaattisesti saatavilla (Crispin ja Gregory, 2009).

Valittujen työkalujen täytyy toimia jo valmiilla alustoilla. Lisäksi niiden tulisi toimia hyvin muiden käytössä olevien työkalujen kanssa. Työkaluja tulee säätää jatkuvasti, jotta ne voivat olla avuksi kulloistenkin ongelmien kanssa. Jos koontiversion testit epäonnistuvat päivittäin, voidaan tulokset näyttää esimerkiksi yleisellä ruudulla, jotta tiimi pysyisi tietoisena koontiversion tilasta. Jos bisnespohjainen testi epäonnistuu, pitäisi olla selkeää, mikä tarkalleen meni rikki ja missä. Ongelmien eristämiseen pystytään harvoin käyttämään ylimääräistä aikaa (Crispin ja Gregory, 2009). Omaan itse kokemusta tällaisesta järjestelystä. Tiimin tiloissa on ruutu, josta näkyy jatkuvan integrointijärjestelmän Jenkinsin ajamien koontiversioiden tila. Ruudusta näkee heti,

kun testit epäonnistuvat. Ajetut testit ja epäonnistumisen syy voidaan tarkistaa ohjelman tekemistä logeista. Näin sovelluksen tilasta saadaan nopeaa palautetta.

Crispinin ja Gregoryn mukaan nämä huolet ovat tärkeä osa kokonaiskuvaa, mutta silti vain yksi osa sitä. Tarvitaan myös työkaluja, jotka auttavat tuotantoympäristöä matkivien testiympäristöjen luomisessa. On tarpeen löytää keinoja testiympäristöjen pitämiseksi itsenäisinä, jotta ohjelmoijien tekemät muutokset eivät vaikuttaisi niihin (Crispin ja Gregory, 2009).

Testi – infrastruktuurin luominen voi olla suuri sijoitus, mutta sen avulla ketterät tiimit kykenevät aloittamaan testiautomaation toteuttamisen. Laitteisto, ohjelmisto ja työkalut tulevat olla tunnistettavissa ja toteutettavissa. Projekti saattaa olla pitkäaikainen riippuen organisaation resursseista. Tiimit voivat kehitellä ideoita selviytyäkseen lyhyellä aikavälillä samalla, kun suunnitellaan, miten pystyttää tarvittava infrastruktuuri riskien minimoimiseksi, tiimin nopeuden maksimoimiseksi ja mahdollisimman laadukkaan tuotteen julkaisemiseksi (Crispin ja Gregory, 2009). Seuraavassa aliluvussa käsitellään testien ja testitulosten hallintaa.

5.2 Automatisoitujen testien ja testitulosten hallinta ja organisointi

Voi olla tilanteita, jolloin tarvitaan löytää jonkin tietyn skenaarion verifioiva testi ja ymmärtää, mitä jokainen testin osa tekee ja minkä osan sovelluksesta kyseinen testi verifioi. Automatisoituja testejä täytyy ylläpitää ja hallita aivan kuten tuotantolähdekoodia. Kun tuotantokoodi merkataan julkaistavaksi, toiminnallisuuden verifioivat testit täytyvät olla osana kokonaisuutta (Crispin ja Gregory, 2009).

Crispinin ja Gregoryn mukaan hyödyllinen esimerkki tästä voisi olla ongelma kehitettävässä ohjelmakoodissa. Onko ongelma uusi, vai onko se ollut osa ohjelmakoodia jo pidemmän aikaa ilman, että testit olisivat huomanneet sen? Tuotannossa oleva koontiversio voidaan ottaa käyttöön ja koittaa toistaa ongelma. Lisäksi voidaan tutkia syytä, miksi testit eivät havainneet virhettä. Esimerkiksi virhe, joka johtuu tietokantarajoitteen puutteesta testiskeemassa, on hankalasti selvitettävissä, ellei testikoodiversiointia yhdistetä tuotantokoodiversiointiin (Crispin ja Gregory, 2009).

Eri työkalut tarjoavat omanlaisensa ratkaisun testien organisointiin. Esimerkiksi FitNesse – työkalulla on oma wiki, hierarkkinen organisointi ja sisäänrakennettu versionhallinta. FitNesse tarjoaa tuen myös joillekin lähdekoodin hallintatyökaluille, kuten Subversionille. Muilla testityökaluilla kirjoitettuja skriptejä voidaan ylläpitää yksikkötestien tapaan samassa lähdekoodin hallintajärjestelmässä kuin tuotantokoodia (Crispin ja Gregory, 2009).

Testien hallinta auttaa tiimiä lukuisissa ongelmakohdissa. Näihin lukeutuvat tiedot testitapausten automatisoinnin tilasta ja siitä, mitä toiminnallisuuksia testit kattavat. Myös testitapauksiin liittyvä metadata on tärkeää, kuten se, milloin testitapaukset on luotu ja kuka kirjoitti ne. Lisäksi on tärkeää selvittää, mitkä testit ajetaan osana regressiotestiympäristöä ja miten kauan testit ovat olleet sen osana (Crispin ja Gregory, 2009).

Koska yksi suurimmista syistä kirjoittaa testejä on ohjata kehitystä, testit täytyy organisoida niin, että koko tiimi pystyy löytämään oikeat testit jokaiselle toiminnallisuuden osalle ja tunnistamaan helposti, minkä toiminnallisuuden osan kukin testi kattaa. Koska testejä voidaan käyttää dokumentaationa, on tärkeää, että jokainen henkilö kehitys – ja asiakastiimeissä kykenee löytämään tietyn testin nopeasti, kun halutaan tietää, miten järjestelmän kuuluisi käyttäytyä. Eri testien hallinnan tavoitteiden saavuttamiseen saatetaan tarvita useita työkaluja (Crispin ja Gregory, 2009).

Testiskriptien hallinnan menettäminen voi käydä helposti. Kun testi epäonnistuu, täytyy ongelma tunnistaa mahdollisimman nopeasti. Tällöin saatetaan joutua selvittämään, mitä muutoksia testiskriptiin on lähiaikoina tehty. Tätä tarkoitusta varten voidaan hyödyntää lähdekoodin hallintajärjestelmän historiatietoja. Asiakastiimi tarvitsee myös keinon pitää kirjaa projektin edistymisestä ymmärtääkseen, miten suuri osa ohjelmakoodista on katettu testeillä, ja mahdollisesti keinon ajaa testit itse. Testit ja testien hallintajärjestelmät tulisivat edistää kommunikointia ja yhteistyötä tiimin jäsenten välillä sekä eri tiimien välillä (Crispin ja Gregory, 2009).

Kaikki ohjelmiston julkaisemisessa mukana olevat tahot tarvitsevat helpon pääsyn testeihin ja testituloksiin. Testien hallinnan toinen osa – alue on pitää kirjaa siitä, mitkä testit ovat sellaisia, jotka ovat peräisin aiemmilta iteraatioilta ja joiden täytyy mennä

jatkuvasti läpi, ja mitkä taas sellaisia, jotka ajavat kehitystä nykyisessä iteraatiossa eivätkä välttämättä mene vielä läpi. Testejä ajetaan jatkuvalla integrointi – ja koontiprosessilla kehityksen nopeaa palautetta ja regressiovirheiden löytämistä varten (Crispin ja Gregory, 2009).

Jos kehitystä tehdään testivetoisesti ja jotkut testeistä eivät mene vielä läpi, koontiversion luonti ei saisi epäonnistua. Jotkut tiimit yksinkertaisesti pitävät uudet testit integraatio – ja koontiprosessin ulkopuolella, kunnes ne menevät ensimmäistä kertaa läpi. Tämän jälkeen kyseisten testien on mentävä aina läpi. Toiset tiimit taas käyttävät sääntöjä itse koontiprosessissa virheiden sivuuttamiseksi sellaisista testeistä, jotka ovat kirjoitettu kehityksessä olevan ohjelmakoodin kattamiseksi (Crispin ja Gregory, 2009).

Kuten millä tahansa testiautomaatiotyökalulla, testien hallinnan ongelmat voidaan ratkaista joko itse tehdyillä, avoimen lähdekoodin tai kaupallisilla järjestelmillä. Testityökalujen arviointikriteerejä voidaan soveltaa myös testien hallinnan lähestymistavan valintaan (Crispin ja Gregory, 2009).

Testien hallinta on alue, johon pätevät ketterät arvot ja periaatteet ja koko tiimin lähestymistapa. Testien ja tuotantokoodin synkronoimiseksi on löydettävä oikea yhdistelmä työkaluja muun muassa lähdekoodin hallintaan ja koontiversioiden hallintaan. Tämä tavoite voidaan saavuttaa aloittamalla yksinkertaisesti ja kokeilemalla pienin askelin. Testien hallinnan lähestymistapaa tulisi arvioida usein ja varmistaa, että se sopii kaikille testien käyttäjille. On tunnistettava toimivat ja puutteelliset osa – alueet ja suunnitella tehtävät tai ominaisuudet kokeilemaan toista työkalua tai prosessia aukkojen täyttämiseksi. Testien hallinta tulisi myös pitää mahdollisimman kevyenä ja ylläpidettävänä, jotta kaikki voivat hyödyntää sitä (Crispin ja Gregory, 2009).

6 Yhteenveto

Tutkielman aihe osoittautui lopulta hankalaksi. Vaikka aihe oli mielenkiintoinen, kritisoivia tutkimuksia testiautomaatiosta ketterässä ohjelmistokehityksessä tai testityökalujen valinnasta ei juurikaan löytynyt. Toisin sanoen aiheesta tuntuisi vallitsevan yksimielisyys tutkimuskentällä. Sain kuitenkin vastauksen kaikkiin tutkimuskysymyksiin. Testiautomaation heikkouksista olisi mahdollista lähteä tekemään jatkotutkimusta. Kuten Angmo ja Sharma totesivat artikkelissaan, myös testiautomaatiotyökalujen vertailuja voitaisiin tutkia jatkossa syvemmin.

Testiautomaatiota tulisi hyödyntää ketterässä ohjelmistokehityksessä etenkin regressiotestien automatisointiin. Näin voidaan varmistua siitä, että tärkeimpiin toiminnallisuuksiin kohdistuvista virheistä saadaan välitöntä palautetta. Ilman automaatiota testaaminen vie liikaa aikaa, jolloin resurssien puutteen vuoksi myös ohjelmoijat joutuvat testaamaan manuaalisesti. Automaatiolla saadaan vähennettyä inhimillisiä virheitä ja myös testidatan muodostaminen voidaan automatisoida. Testien jatkuva ylläpito ja päivittäminen on kuitenkin tärkeää.

Testiautomaatio vaatii kuitenkin suuren investoinnin. Kaikki ohjelmoijat eivät välttämättä ajattele testattavuutta. Automaation perustamisessa voi olla tällöin jyrkkäkin oppimiskäyrä, johon kuuluvat esimerkiksi erilaisten käytäntöjen opettelu. Testiautomaatiokoodia tulisi kuitenkin käsitellä samalla tavalla kuin varsinaista ohjelmakoodia. Testien suunnittelu tulee olla hyvällä tasolla ja testivetoisesti kirjoitettu ohjelmakoodi edistää automaation perustamista. Vanhentunutta ohjelmakoodia voidaan refaktoroida paremmin testattavaksi samalla, kun uutta koodia ohjelmoidaan testivetoisesti ja toteutetaan testiautomaatio.

Testipyramidi jakaa testit kolmeen kerrokseen. Se auttaa tutkimaan, miten testiautomaatio voi auttaa ketterää kehitystiimiä. Pohjimmainen kerros sisältää yksikkö – ja komponenttitestit, jotka ovat automatisoinnin kannalta tärkeimpiä. Toinen kerros sisältää bisnespainotteiset testit ja ylin kerros hitaasti ajettavat graafisen käyttöliittymän testit, jotka omaavat pienimmän investoinnin palautusarvon.

Koontiversiot tulisivat olla mahdollisimman nopeasti testaajien saatavilla. Tähän auttaa automatisoitu koontiprosessi jatkuvan integroinnin yhteydessä. Jatkuva integrointi auttaa myös palautteen nopeaan saamiseen. Koontiprosessia tulisi aina pyrkiä nopeuttamaan.

Automatisoitaviin testityyppeihin lukeutuvat yksikkö – ja komponenttitestit, API – ja web – palvelutestaus, graafisen käyttöliittymän ja sen takana ajettavat testit ja performanssitestaus. Kaikkea käyttökokemukseen ja käytettävyyteen liittyvää ei kuitenkaan voida automatisoida. Graafisen käyttöliittymän ulkoasuun liittyvää testausta on myös turha automatisoida. Tutkivaa testausta voidaan nopeuttaa skripteillä, mutta varsinainen tutkiva testaus tulisi olla manuaalista. Myöskään sellaisia testejä ei tarvitse automatisoida, jotka eivät koskaan epäonnistu.

On hankalaa automatisoida ohjelmakoodia, jossa testattavuutta ei ole huomioitu. Automatisointi tulee aloittaa sovelluksen kriittisimmistä alueista testipyramidin pohjalta. Hyvät kehitysmenetelmät tekevät testiautomaatiosta helpompaa. Testiautomaation kustannukset, kuten testien ylläpidettävyys, tulisi kuitenkin arvioida testiautomaation tarvetta vastaan.

Testisuunnitelma, työkalut ja ratkaisut tulisivat olla mahdollisimman yksinkertaisia. Lyhyitä iteraatioita voidaan hyödyntää erilaisten vaihtoehtojen kokeilemiseen. Sekä Crispin ja Gregory että Collins et al. ovat sitä mieltä, että koko tiimin lähestymistavalla voidaan saavuttaa hyvä automaatiostrategia. On parempi leikata tulevan julkaisun toiminnallisuuksista tai niiden laajuudesta, jotta jäljellä oleville toiminnallisuuksille saadaan riittävän kattava testiautomaatio. Testit ovat yhtä arvokkaita kuin varsinainen ohjelmakoodi, joten ne olisi pidettävä saman versionhallintaohjelmiston alla.

Testidataa on mahdollista generoida eri skripteillä ja työkaluilla. Muistin sisäisen testidatan käyttö on nopeaa. Testistä riippuen voidaan käyttää joko ainutlaatuista dataa tai satunnaisesti generoitua dataa. Stressitestaus ja suorituskykytestaus tarvitsevat tuotantoympäristöä vastaavan testiympäristön. Tuotantodatan käyttö on kuitenkin kallista ja hidasta.

Testityökalujen valinnassa on tärkeää tiedostaa, että yksi työkalu ei todennäköisesti sovi kaikkiin tarkoituksiin. Testityökalulla tulisi olla mahdollisimman vaivatonta

päivittää testit ohjelmiston logiikan muutoksia vastaaviksi. Testityökalua valitessa riittää työkalu, joka täyttää mahdollisimman monet tarpeet Esimerkiksi tallennus – ja toistotyökalut sopivat vanhentuneille järjestelmille. Tarkoitukseen sopivan testityökalun valitsemiseksi voidaan toteuttaa työkalujen keskinäinen vertailu tiettyjen kriteerien suhteen, kuten Angmo ja Sharma olivat tehneet Seleniumille ja Watirille.

Testityökaluratkaisuksi voidaan kehittää oma työkalu tai hyödyntää jotakin valmista kaupallista tai avoimen lähdekoodin työkalua. Itse tehdyt testityökalut sopivat ainutlaatuisiin testaushaasteisiin. Myös valmiille avoimen lähdekoodin pohjille on mahdollista rakentaa oma ratkaisu. Itse tehdyt työkalut vaativat kuitenkin paljon resursseja niiden toteutukseen. Kuten Ramlerin ja Putschöglin tapauksesta huomattiin, itse tehty työkalu voidaan rakentaa vastaamaan kaikkia yksittäisiä vaatimuksia.

Moni avoimen lähdekoodin työkalu on kirjoitettu testivetoisesti, mutta ne eivät välttämättä ole hyvin dokumentoituja. Avoimen lähdekoodin työkaluilla on kuitenkin usein aktiiviset yhteisöt. Suositut kaupalliset työkalut ovat monesti varma valinta testityökaluksi, mutta ne ovat yleensä raskaita ja vaikeita ohjelmoijille käytettäviksi.

Ensimmäinen askel testiautomaatiotyökalun valintaan on tehdä lista kaikista tarpeista. Automatisoitava testaustyyppi vaikuttaa työkalun valintaan. Kokemus aiemmista automaatiohaasteista auttaa valinnassa. Raulamo – Jurvanen et al toteavat, että juuri oikeiden työkalujen puute on merkittävin este testiautomaatiolle. Ulkoisten konsulttipalveluiden käyttö on suosittua testityökalujen valinnassa. Välillä tiimin tulisi pitää kehitysiteraatio, jonka aikana vähennetään teknista velkaa ja päivitetään ja kokeillaan eri testityökaluja. Työkalun käytännön kokeilu on tärkeää, kun sitä arvioidaan. Raulamo – Jurvasen et al tekemän tutkimuksen mukaan testityökalun valintaan liittyvistä kriteereistä suurimmat ovat sopivuus testivaatimuksiin ja käyttöympäristöön, kustannukset ja käytettävyys.

Viitteet

Angmo, R., Sharma, M. (2014) Performance Evaluation of Web Based Automation Testing Tools. *2014 5th International Conference – Confluence The Next Generation Information Technology Summit (Confluence)*, Noida, 731-735.

Collins, E., Dias-Neto, A., de Lucena Jr., V. (2012) Strategies for Agile Software Testing Automation: An Industrial Experience. *IEEE 36th International Conference on Computer Software and Applications Workshops*, Izmir.

Crispin, L., Gregory, J. (2009) *Agile Testing – A Practical Guide For Testers And Agile Teams.* Pearson Education, Inc., Boston, MA.

Li, K., Wu, M. (2004) *Effective GUI Testing Automation: Developing an Automated GUI Testing Tool*. Sybex, London.

Ramler, R., Putschögl, W. (2013) A Retrospection on Building a Custom Tool for Automated System Testing. *2013 IEEE 37th Annual Computer Software and Applications Conference*, Kyoto, 820-821.

Raulamo-Jurvanen, P., Mäntylä, M., Garousi, V. (2017) Choosing the Right Test Automation Tool: a Grey Literature Review of Practitioner Sources. *EASE'17 Proceedings of the 21st Internationat Conference on Evaluation and Assessment in Software Engineering*, Karlskrona, 21-30.

Selenium Contributors (2017) *Selenium – Web Browser Automation*. WWW – sivusto, http://www.seleniumhq.org (15.06.2017).

Watir Contributors (2017) *Watir is... – Watir Project – Watir stands for Web Application Testing In Ruby. It facilitates the writing of automated tests by mimicking the behavior of a user interacting with a website.* WWW – sivusto, http://www.watir.com (16.06.2017).