Petri Timperi

# VAATIMUKSET TESTIAUTOMAATIOKEHYKSELLE TOIMINNALLISESSA TESTAUKSESSA

# TIIVISTELMÄ

Timperi, Petri
Vaatimukset testiautomaatiokehykselle toiminnallisessa testauksessa
Jyväskylä: Jyväskylän yliopisto, 2015, 40 s.
Tietojärjestelmätiede, kandidaatin tutkielma
Ohjaaja(t): Halttunen, Veikko

Tutkielmassa käsitellään ohjelmistojen toiminnallisen testauksen automaatiossa käytettävää kehystä joka toteuttaa testiautomaation arkkitehtuurin ja sen peruspalvelut. Kehyksistä tarkastellaan niiden toiminnallisuuksia, arkkitehtuuriratkaisuja ja suunnitteluperiaatteita sekä listataan vaatimukset, joita kehyksen tulisi toteuttaa. Työssä perustellaan miksi testiautomaatio on tärkeää ketterässä kehityksessä ja listataan merkittävimmät ongelmat, joita testiautomaation käyttöönotossa tyypillisesti kohdataan. Testiautomaatiokehyksen rooli ongelmien ratkaisemiseksi käydään läpi ja perustellaan, miksi sen kehitykseen ja käyttöönottoon kannattaa panostaa.

Tutkielma on tehty kirjallisuus- ja artikkelikatsauksena. Siinä on pyritty löytämään eri lähteistä ratkaisuja testiautomaation ongelmiin testiautomaatiokehyksen näkökulmasta sekä löytämään vaadittavat toiminnallisuudet ja vaatimukset jotka kehyksen tulee toteuttaa.

Tutkielmassa tarkastellaan toiminnallisten testien testiautomaatioratkaisuja, niiden suunnitteluperiaatteita ja arkkitehtuuri-ratkaisuja. Suunnitteluperiaatteista käydään läpi modulaarisuus, kirjastojen käyttö, avainsana- ja datapohjaisuus, sekä näiden yhdistelmä hybridi testiautomaatiokehys. Tapaustutkimusten pohjalta havaitaan että tämän päivän kehykset ovat tyypillisesti hybridi-kehyksiä ja niissä erotellaan kehyksen ja testausvälineen osuudet eri tasoille arkkitehtuurissa. Kehys toteuttaa tyypillisesti testien hallinnan, testidatan käsittelyn, tulosten raportoinnin sekä testiympäristön alustuksen. Testausvälineen rooli on toteuttaa testiskriptit, jotka ajetaan kehyksen toimesta testattavaa järjestelmää vasten.

Tutkielmaan on koottu testiautomaatiokehysten yleisiä vaatimuksia joita voidaan hyödyntää sovellusaluekohtaisten vaatimusten määrittelyssä. Arkkitehtuurivaatimusten lisäksi, kehyksen tärkeimmät vaatimukset olivat laaja räätälöitävyys, testausvälineriippumattomuus, tuki hajautetuille testeille sekä kattava testien hallinta niin raportoinnin kuin testien valinnan ja muokkauksen suhteen.

Asiasanat: testiautomaatiokehys, testiautomaatio, toiminnallinen testaus, avainsanapohjainen testaus

# ABSTRACT

Timperi, Petri
Requirements for Test Automation Framework of Functional Testing
Jyväskylä:University of Jyväskylä, 2015, 39 s.
Information Systems, Bachelor's Thesis
Supervisor: Halttunen, Veikko

The study goes through the functionalities, design principles and architecture solutions of test automation frameworks. It explains the principles behind frameworks and collects their generic requirements. The importance of test automation is elaborated especially in Agile environments. The typical adoption issues of test automation are covered and the solutions that test automation framework design can offer to solve those issues. The study is based on literature review where these issues are addressed.

The study walks through the solutions of test automation frameworks for functional testing. What are their design principles and architectural solutions. The modularity, library usage, keyword- and data-driven design principles are discussed as well as the combination of these, the hybrid design. Based on various case studies, the most common framework type today is hybrid. It implements the design principles mentioned and separates the testing tool and framework parts in the architecture. The framework's role is to implement test management, test data handling, results reporting and test configuration features. Testing tool's responsibility is to implement the actual test scripts executed against the system under test.

The generic requirements of test automation frameworks were collected in the study. In addition to architecture requirements, the most important requirements were full configurability, testing tool independency, support for distributed tests and extensive test management functionalities, including test selection, editing and reporting. Based on this study, application area specific requirements can be processed.

Keywords: test automation framework, test automation, functional testing, keyword-driven testing

# KUVIOT

# SISÄLLYS

# 1 JOHDANTO

Ohjelmistojen automaattisen testauksen (myöhemmin testiautomaatio) merkitys on kasvanut koko 2000-luvun ajan, osin ketterien menetelmien nousun, osin lisääntyneiden tehokkuusvaatimusten johdosta. Ketterät menetelmät yleisesti ovat johtaneet ohjelmistokehityssyklin lyhenemiseen niin, että ohjelmistotuotteiden julkaisuja, joko kehityksen aikaisia tai tuote-julkaisuja, tehdään useammin. Näiden julkaisujen yhteydessä tehtävä ohjelmiston integrointi on mahdollistanut virheiden löytymisen aiempaa aikaisemmin tuottaen kehittäjälle nopeamman palautteen tehtyjen muutosten onnistumisesta. (Crispin & Gregory, 2009, 255-264.)

Ketterän kehittämisen myötä tihentynyt integrointisykli on asettanut uusia vaatimuksia ohjelmistojen testaukselle. Erityisesti ohjelmistomuutosten vaikutusten todentaminen laaja-alaisesti koko ohjelmistotuotteessa on kohdannut uusia haasteita. Uusien toiminnallisuuksien lisäksi myös vanha toiminnallisuus on pitänyt testata aiempaa lyhyemmässä ajassa. Perinteisesti vanhan toiminnallisuuden toimivuuden tarkistus, regressiotestaus, on voinut viedä suurimman osan ohjelmiston uuden julkaisun testausajasta, etenkin jos laadusta ei ole haluttu tinkiä. Nyt kun ketterien menetelmien avulla on pyritty saamaan nopeaa palautetta toteutetuista vaatimuksista ja näin vastaamaan nopeammin markkinoiden odotuksiin, on testaus ollut herkästi pullonkaula ketterässä kehityssyklissä etenemiseen. (Crispin & Gregory, 2009, 258-259.)

Yksi tärkeimmistä testauksen tehostamisen ja nopeuttamisen keinoista on automaattisen testaamisen lisääminen. Testiautomaatio tarkoittaa testausvälineen avulla, ilman testaajan toimenpiteitä, tapahtuvaa testausta. Testiautomaatiota on kehitetty testauksen eri vaiheille yksikkö- ja komponenttitesteihin ja edelleen toiminnallisiin testeihin. Testaustavoitteiden erilaisista vaatimuksista johtuen testausmetodit ja testiautomaatioratkaisut ovat olleet erilaisia eri testausvaiheille. (Crispin & Gregory, 2009, 98-100.) Tässä tutkielmassa tarkastellaan toiminnallisten testien testiautomaatiota, ja erityisesti sen automaatiokehysratkaisuja.

Toiminnallisten testien automaatiokehys (myöhemmin testiautomaatiokehys) tarkoittaa testiautomaatioratkaisun arkkitehtuuria ja siihen toteutettuja

peruspalveluja. Sillä kontrolloidaan toiminnallisia, käyttöliittymän tai ylätason rajapinnan kautta tapahtuvia, testejä. Tutkielmassa selvitetään erityyppisten toiminnallisen testauksen testiautomaatiokehyksien ominaisuuksia ja suunnitteluperiaatteita: Mitä eroja kehyksillä on ja miten ne sijoittautuvat testiautomaatioratkaisujen arkkitehtuurissa? Millaisia olemassa olevia ratkaisuja on olemassa ja mitä toiminnallisuuksia kehyksissä tyypillisesti on? Tutkimusongelmat ovat seuraavat:

- Mitä toiminnallisen testauksen testiautomaatiokehykset ovat ja mikä niiden rooli on testiautomaatioratkaisuissa?
- Mitkä ovat vaatimukset testiautomaatiokehykselle jonka avulla voidaan rakentaa kustannustehokas ja ylläpidettävä toiminnallisen testauksen automaatio niin, että sen rakentaminen on kannattavaa?

Tutkielma toteutetaan kirjallisuuskatsauksella. Katsauksessa pyritään löytämään eri lähteiden ja niissä olevien tapausesimerkkien avulla toiminnallisen testauksen testiautomaatiokehyksien toiminnallisuudet ja niiden arkkitehtuuriratkaisut. Tutkielma pyrkii kuvaamaan testiautomaatiokehyksen hyödyt ja perustelemaan miksi sen toteuttamiseen kannattaa panostaa heti automaatiota aloitettaessa. Se koostaa tärkeimmät vaatimukset jotka tulee huomioida kehyksen rakentamisessa jotta siitä saadaan modulaarinen, useita eri testausvälineitä tukeva alusta, jonka päälle on helppo rakentaa kannattavasti lisää testiautomaatiota järjestelmän elinkaaren aikana.

Kehyksen hyötyjen kuvaaminen on tärkeää koska sillä on tärkeä rooli testiautomaation onnistumisessa. Kehyksen toteuttamisen haaste on siinä, että ainoastaan kehyksen kuvaamalla arkkitehtuurilla ei vielä voida suorittaa testejä, vaan kehyksen rooli on tarjota alusta jolle testiautomaatiota voidaan tehokkaasti toteuttaa. Usein testiautomaation rakentamisessa halutaan kuitenkin tuloksia mahdollisimman pikaisesti, jolloin kehyksen toteutukseen ja käyttöönottoon ei uhrata riittävästi aikaa. Myöhemmin huonosti suunniteltu kehys voi vaarantaa koko testiautomaation kannattavuuden kun ylläpitoon ja modulaarisuuteen ei ole kiinnitetty riittävästi huomiota. (Graham & Fewster, 2012, 9-10.)

Tutkielmaan koottuja testiautomaatiokehyksen vaatimuksia voidaan hyödyntää määriteltäessä oman sovellusalueen vaatimukset kehykselle. Vaatimuksia voidaan käyttää eri kehystuotteiden arvioinnissa ja päätettäessä niiden käyttöönotosta. Se palvelee myös alustavana listana vaatimuksista omaan kehitykseen pohjautuvalle kehykselle tai avoimen lähdekoodin tuotteen räätälöinnille vastaamaan asetettuja vaatimuksia.

Tutkielma rakentuu niin, että ensimmäisen johdantoluvun jälkeen kuvataan luvussa kaksi toiminnallinen testiautomaatio yleisesti. Testiautomaatiosta selvitetään sen tarkoitus, mistä komponenteista se koostuu sekä mikä on kehyksen rooli kokonaisuudessa. Luvussa käsitellään myös testiautomaation haasteita ja periaatteita joilla haasteita voi ratkaista.

Luvussa kolme käydään läpi testiautomaatiokehysten tyypit ja niiden arkkitehtuuriratkaisut. Mitkä periaatteet automaatioratkaisun arkkitehtuurin tulee

toteuttaa sekä mitä yleisiä testiautomaatiota helpottavia ominaisuuksia eri ratkaisuista löytyy.

Seuraavassa luvussa neljä tarkastellaan testiautomaatiokehyksen vaatimuksia. Siinä käydään läpi testiautomaatiostrategian vaikutukset vaatimuksiin sekä listataan eritasoisia vaatimuksia kehykselle. Vaatimukset on jaettu lähteiden pohjalta eri kategorioihin joita voi tarkastella omaa testiautomaatioratkaisua valittaessa.

Luku viisi koostaa yhteenvedon tutkimuksen tuloksista ja kuvaa soveltamiskohteita. Siinä pohditaan testiautomaation merkitystä nykyaikaisessa ohjelmistokehityksessä sekä testiautomaatiokehyksen roolia kokonaisuudessa.

# 2 AUTOMATISOINTI TOIMINNALLISESSA TESTAUKSESSA

Tässä pääluvussa käydään läpi testiautomaation merkitystä modernissa ohjelmistokehityksessä ja perustellaan sen käytön tarpeellisuutta. Luvussa taustoitetaan tutkimusaluetta ja kuvataan testauksen vaiheet ja testiautomaation rooli niissä. Lopuksi käsitellään testausvälineen ja testiautomaatiokehyksen suhdetta ja määritellään niiden vastuualueet testiautomaatioratkaisussa.

## 2.1 Testiautomaatio yleisellä tasolla

Testiautomaatio tarkoittaa ohjelmistotestausta jossa testiautomaatiovälineen avulla suoritetaan ohjelmistoa testaavia testejä ilman testaajan puuttumista testin suoritukseen. Tyypillisesti testiautomaatio ajaa testin, analysoi testitulokset sekä päättää menikö testi läpi vai ei. (Dustin ym., 2009, 3-21.) Testiautomaatio voidaan toteuttaa esimerkiksi siten, että testiautomaatioväline syöttää arvoja käyttöliittymän kenttiin, painaa painikkeita sekä valitsee arvoja valintalistalta. Alemman tason testissä voidaan testata yksittäistä luokan metodia niin, että tarkistetaan esimerkiksi metodin suorittaman laskennan oikeellisuus. (Crispin & Gregory, 2009, 97-103.)

Testiautomaatiota voidaan tehdä usealla eri testaustavoitteella jolloin sen luonne on erilainen tavoitteesta riippuen. Testausta voidaan automatisoida muun muassa teknisen toiminnallisuuden, liiketoimintalogiikan, suorituskyvyn, rinnakkaisen suorituksen tai luotettavuuden varmistamiseksi. (Dustin ym., 2009, 38-49.)

Testiautomaation rooli on kasvanut siirryttäessä perinteisistä menetelmistä ketteriin ohjelmistonkehitysmenetelmiin. Perinteisten menetelmien pitkät julkaisusyklit mahdollistivat paremmin manuaalisen testauksen julkaisusyklin ollessa esimerkiksi puoli vuotta tai vuosi. Ketterien menetelmien käyttöönoton myötä syklit ovat lyhentyneet jolloin testauksen työmäärä on noussut regressiotestien vuoksi. Jokaisesta julkaisusta pitää varmistaa, että myös vanha toi-

minnallisuus toimii, etteivät tehdyt muutokset ole rikkoneet sitä. (Leffingwell, 2007, 75-85)

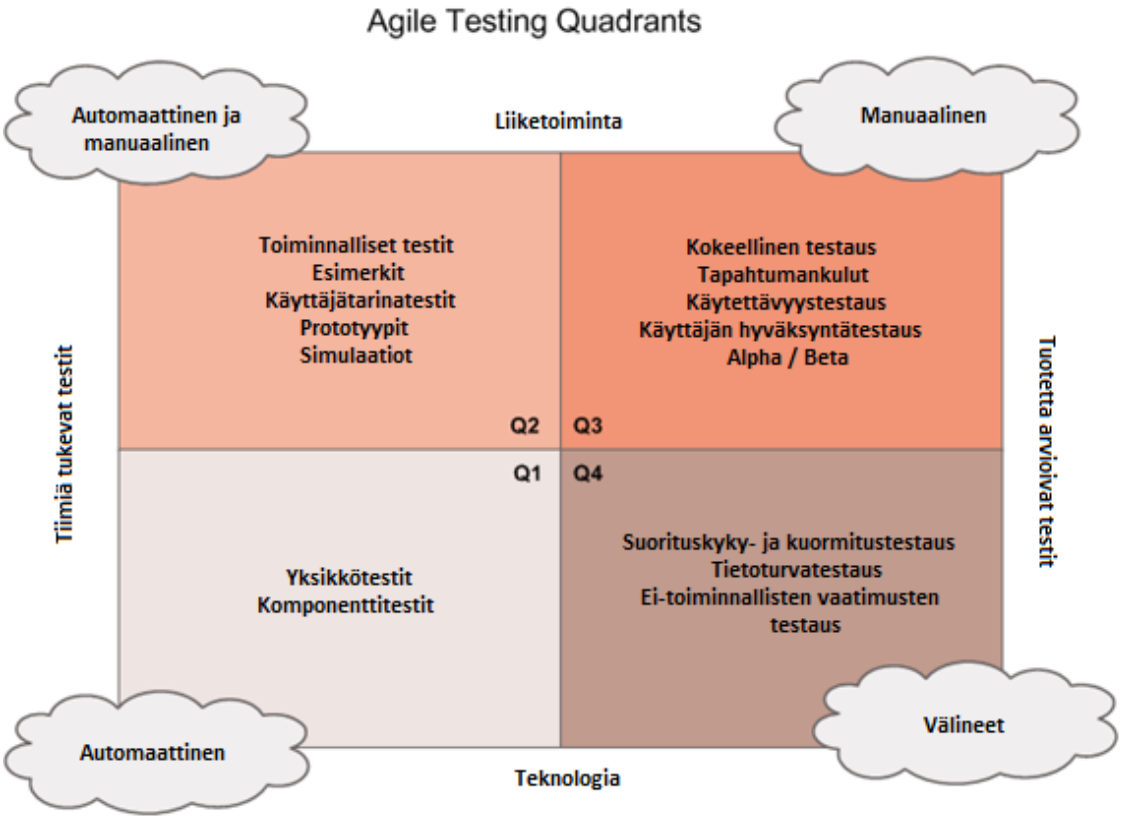
Testaamiseen on ketterissä menetelmissä kehitetty käytäntöjä kuten Test Driven Development (TDD), jossa automaattiset yksikkö- ja komponenttitestit tehdään aina, kun kehitetään uutta toiminnallisuutta ja ne ajetaan jatkuvan integroinnin käytäntöjen mukaisesti jokaisessa koonnossa (build). Myös käyttöliittymä- ja rajapintatesteissä on alettu hyödyntää testiautomaatiota, koska lyhyen julkaisusyklin ja testausajan vuoksi kattava manuaalinen testaus ei ole mahdollista. (Leffinwell, 2007, 155-175.)

## 2.2 Testausvaiheet ja toiminnallinen testaus

Testausvaiheiden jaolle on esitetty useita eri malleja. Suosittu malli on V-malli jossa kutakin ohjelmistokehityksen vaihetta vastaa vaihe testauksessa (ISTQB 2015). ISTQB:n mukaiset testausvaiheet ovat yksikkötestaus, komponenttitestaus, integrointitestaus, systeemi-integrointitestaus, systeemitestaus sekä hyväksyntätestaus.

Myös muita testauksen vaihejako-malleja on olemassa esimerkiksi iteraatiiviseen ja ketterään kehitykseen mutta niissä kaikissa voidaan löytää edellä mainitun kaltaisia vaiheita joissa testauksen luonne ja tavoite on erilainen. (Hass, 2008, 8-11.)

Testiautomaatiosta puhuttaessa edellä kuvattua vaihejako-mallia ei tyypillisesti ole käytetty. Yleisesti jako on tehty testauksen suoritustavan mukaan, minkä tyyppisiä automatisoituja testejä toteutetaan. Crispin ja Gregory (2009) ovat jakaneet eri testausvaiheet liiketoiminta- ja teknologiajaon mukaan. Tässä jaossa teknologialähtöiset testit tarkoittavat white-box tyyppistä testausta jossa testaus tehdään tuntien ohjelmiston sisäinen rakenne ja yksittäisten luokkien ja metodien rooli rakenteessa. Teknologialähtöiset testit ovat tyypillisesti yksikkö ja komponenttitestejä joita tehdään kehittäjien toimesta xUnit-tyyppisillä työkaluilla. Liiketoiminta-lähtöiset testit taas ovat pääasiassa black-box testausta jossa ohjelmiston sisäistä rakennetta ei tunneta, tai gray-box testausta jossa sisäinen rakenne tunnetaan osin ja käytetään tietoa hyväksi esimerkiksi testin läpimenon arvioimiseksi. Liiketoimintalähtöiset testit perustuvat usein käyttäjälle näkyviin toiminnallisuuksiin käyttöliittymässä joita voidaan verifioida toiminnallisia vaatimuksia vasten. (Gregory & Crispin, 2009, 97-102.) Kuva 1 on esitetty kyseinen jako.




Kuva 1. Testauksen vaihejako ja testiautomaatio (Crispin & Gregory, 2009, 98)

Dustin ym. (2009, 15) taas listaa hieman samalla ajatuksella eri testaustavoitteet joissa testiautomaatiolla on merkillepantava rooli:

- Yksikkötestaus (Unit testing)
- Regressiotestaus (Regression testing)
- Toiminnallinen testaus (Functional testing)
- Tietoturvatestaus (Security testing)
- Suorituskykytestaus (Performance testing)
- Stressitestaus (Stress testing)
- Rinnakkaistestaus (Concurrency testing)

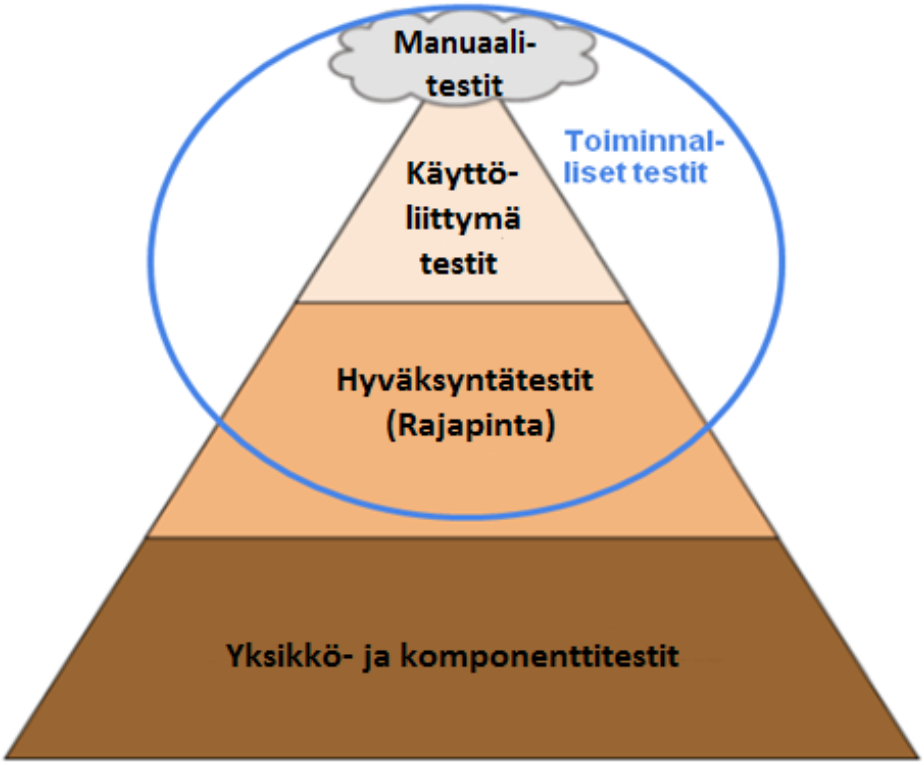
Tutkielman aiheena olevan testiautomaatiokehyksen vaatimukset toiminnallisessa testauksessa rajaus koskee testiautomaatiota, jolla varmistetaan gray-box-tyyppisesti ohjelmiston toiminta asetettuja toiminnallisia vaatimuksia vasten. Vaatimukset kyseiselle testiautomaatiolle poikkeavat osin vaatimuksista koskien white-box-tyyppistä testausta jotka edellä mainituissa lähteissä on rinnastettu lähinnä yksikkö- ja komponenttitason testeihin. Toiminnallinen testaus termin käyttöä puoltaa myös yleinen testiautomaatiovälineiden jako, jossa käyttöliittymä- ja rajapintatestauksen välineet tyypillisesti niputetaan tämän termin alle. (Hass, 2008, 385-386.)

Kuitenkin on hyvä huomata, että sinällään testiautomaatiokehyksen ei tarvitse rajautua palvelemaan vain tiettyä testausvaihetta, mutta usein rajanveto

on järkevää, jotta kunkin testausvaiheen erityispiirteet ja vaatimukset tulevat huomioiduksi parhaalla mahdollisella tavalla. Usein rajanvetoon vaikuttaa myös kehykseen tehtyjen testien omistajuus, yksikkötestauksen omistajuus on tyypillisesti kehittäjillä, kun taas toiminnallisen testauksen omistajuus testaajilla. Myös muiden testaustavoitteiden, kuten suorituskyky- tai tietoturvatestauksen, vaatimukset voivat osin toteutua sekä yksikkö- ja komponentti-, että toiminnallisen testauksen testejä soveltamalla ja hyödyntämällä. (Crispin & Gregory, 2009, 110-120.)

V-mallin kannalta voidaan sanoa, että toiminnallista testausta tehdään pääsääntöisesti integrointi-, systeemi- ja hyväksyntätestaus-vaiheissa yleensä gray-box-tyyppisesti. White-box-tyyppinen testaus taas suoritetaan yksikkö- ja komponenttitestausvaiheissa. (Hass, 2008, 1-23.) Tässä tutkielmassa käytetty jako toiminnalliseen ja yksikkötestaukseen on tehty testauksen suoritustavan mukaan niin, että toiminnallisessa testauksessa testit ajetaan käyttöliittymää tai testattavaa rajapintaa vasten, kun taas yksikkö- ja komponenttitestauksessa testit ajetaan kutsumalla alimman tason luokkien metodeja ilman laajempaa integraatiota muihin ohjelmiston luokkiin ja komponentteihin.

Kuva 2 on esitetty Cohnin (2009) mallia soveltaen toiminnallisten testien rajaus. Kuvassa kolmio kuvaa suositeltua automaatiotestien määrää testaustavoitteittain. Yksikkö- ja komponentti testejä on syytä olla eniten, ja ne ovat muun automaation perusta. Rajapinta- ja käyttöliittymätestit täydentävät ja niitä on määrällisesti vähemmän johtuen niiden toteutuksen ja ylläpidon suuremmasta työmäärästä. (Cohn, 2009.)




Kuva 2. Toiminnallisten testien rajaus Cohnia (2009) soveltaen.

## 2.3 Toiminnallisen testiautomaation haasteet

Yksikkö- ja komponenttitestit ovat tulleet testilähtöisen kehityksen (TDD, test driven development) ja jatkuvan integraation (CI, continuous integration) käytäntöjen, ja yleisesti ketterän kehittämisen, myötä kiinteäksi osaksi ohjelmistojen kehitystä. Testit tehdään tyypillisesti vakiintuneiden käytäntöjen mukaan xUnit yksikkö- ja komponenttitestaus kehyksillä. (Meyer, 2014.)

Toiminnallisessa automaatiotestauksessa haasteet ovat yksikkötestejä suuremmat. Kun yksikkötesteissä kehittäjä tekee toteutusta vastaavat testit kehityksen yhteydessä, tulee testien yhteensopivuus toteutuksen kanssa varmistettua. Tähän lisättynä jatkuvan integraation käytänteet, jossa testejä ajetaan jokaisen muutoksen yhteydessä, on myös testien ylläpito kiinteä osa ohjelmiston kehitystä. Rikkoontuneet testit korjataan heti ja varmistetaan toimivuus ajamalla testit uudelleen. (Meyer, 2014.) Toiminnallisen automaatiotestauksen puolella kehityksen ja testien välinen sidos ei ole yhtä vahva. Testejä kehitetään tyypillisesti sovelluksen testaajien toimesta jolloin kehityksessä toteutettu muutos voi jäädä huomaamatta ja testiskripti päivittämättä. Lisäksi toiminnallisessa automaatiotestauksessa testausvälineiden ongelmat ovat etenkin graafisten käyttöliittymien testauksessa lisätyötä aiheuttavia tekijöitä. (Crispin & Gregory, 2009, 293-297.)

Välineisiin ja yhteistyöhön liittyvien ongelmien lisäksi, toiminnallisen testiautomaation haasteena on sen kehityksen luonne verrattuna testaajien osaamiseen. Testaajien osaamisalue on usein liiketoiminta-osaamisessa, testausprosesseissa sekä testiympäristöjen ylläpidossa. Testiautomaation kehittäminen taas vaatii ohjelmistokehityksen tuntemusta, ohjelmointikielien, -ympäristöjen ja ohjelmistoarkkitehtuurien osaamista. Tästä johtuen testiautomaation kehitys voi keskittyä jonkin kaupallisen valmistuotteen käyttöönottoon ja sillä tehtyjen testien toistamiseen. Testiautomaation arkkitehtuurin ja eri välineiden muodostaman kokonaisuuden miettiminen on jäänyt vähemmälle, kun on pyritty nopeisiin hyötyihin hankitun välineen avulla. (Dustin ym., 2009, 69-97.)

Usein testiautomaation hyödyt jäävät toteutumatta. Uuden välineen käyttöönotto saatetaan tehdä hätäisesti ilman testiautomaatio-strategian ja tavoitteiden miettimistä. Välineellä esimerkiksi vain tallennetaan käyttöliittymässä tehtäviä toimenpiteitä ja ajetaan tallennettu skripti hetken päästä uudestaan eri testiympäristöissä. Käytännössä törmätään Dustin ym. (2009, 72-96) mukaan lukuisiin ongelmiin:

- Tallennetut testiskriptit eivät ole toistettavia. Esimerkiksi käyttöliittymän dynaamisten komponenttien lataus tehdään suorituskerrasta ja ajoympäristöstä johtuen hieman eri tavalla jolloin testi ei toimi toivotulla tavalla.

- Testiskripti ei tarkista automaattisesti testitulosten oikeellisuutta. Automaatiolla tuotetun testidatan tarkistus testaajan toimesta vie liikaa aikaa.
- Testiskriptiä ei voi ajaa uudestaan eri arvoilla vaan tallennuksessa annetut arvot ovat kiinteitä.
- Testiskriptiä ei voi ajaa uudestaan toiseen testiympäristöön muokkaamatta sitä.
- Testiskriptit eivät ole ylläpidettäviä. Esimerkiksi useammassa testitapauksessa käytettävä Kirjautumis-toiminto on tallennettu useaan kertaan. Kun Kirjautumistoiminto muuttuu, joudutaan muuttamaan skriptejä useaan eri testitapaukseen.
- Testeissä ei ole toteutettu virheenkäsittelyä ja virheestä toipumista. Esimerkiksi käyttöliittymä-komponenttien lataus saattaa kestää joskus oletettua kauemmin jolloin testiskriptin pitäisi odottaa komponentin lataamista ilman, että testistä raportoidaan virhe.
- Testien liittäminen osaksi jatkuvan integraation käytäntöjä ei onnistu ja tehdyt testiskriptit jäävät käyttämättä regressiotestauksessa.

Testausvälineen ja testien ajoon liittyvien ongelmien lisäksi haasteita on organisaatiotasolla siinä miten testiautomaatioon suhtaudutaan sekä miten kypsiä ohjelmistokehityksen käytännöt ovat. Crispin ja Gregory (2009, 265-270) listaavat tähän liittyviä ongelmia:

- Ohjelmoijien asenteet eivät tue testiautomaatiota toiminnallisten testien tasolla. Ohjelmoijat eivät ota huomioon toiminnallisia testejä kehityksessä joka johtaa testien ylläpidon ongelmiin ja pitkiin korjausaikoihin.
- Oppimiskäyrän jyrkkyys ja siitä johtuva lisätyö voivat aiheuttaa epäuskoa automaation onnistumiseen. Uusien välineiden opettelu ottaa aikaa ilman, että näkyviä tuloksia syntyy.
- Pitkä takaisinmaksu aika voi johtaa automaation hylkäämiseen. Organisaation päättäjät eivät ymmärrä miten paljon panostusta onnistunut automaatio vaatii esimerkiksi testiautomaatiokehyksen ja skriptien kehityksen osalta.
- Testattava sovellus on jatkuvan päivityksen kohteena, jolloin käyttöliittymä elää ja sen kautta tehtyjä testejä pitää päivittää.
- Perinnejärjestelmien automatisointi voi olla hankalaa koska niitä ei ole suunniteltu ja toteutettu testattaviksi automaatiolla.
- Pelko, ettei osaaminen riitä automaation kehitykseen. Testaajilla puuttui usein ohjelmointitaitoja ja ohjelmoijilta testaustaitoja.
- Vanhoista käytännöistä luopuminen on vaikeaa. Ohjelmiston julkaisupäivän lähestyessä saatetaan helposti palata vanhoihin käytäntöihin ja todeta ettei testiautomaatiolle ole aikaa.

Kaikista edellä mainituista ongelmista johtuen testien säännöllinen ajo testiympäristössä ei käytännössä onnistu automaattisesti. Manuaalinen työ on-

gelmien selvitykseen, testikonfiguraation muuttamiseen ja tulosten tarkastamiseen vie liikaa aikaa mitätöiden automaattisista testeistä saavutetun hyödyn. (Crispin & Gregory, 2009.)

## 2.4 Testiautomaatiokehys

Tutkielmassa tarkasteltava toiminnallisen testauksen testiautomaatiokehys on määritelty Kentin (2007) toimesta niin, että se on testiautomaatioratkaisun ohjelmistoarkkitehtuuri. Ohjelmistoarkkitehtuuri taas tarkoittaa lähdekoodin rakennetta ja siinä olevien komponenttien keskinäisiä suhteita. Tyypillisesti kehyksellä tarkoitetaan kehittyneen testiautomaatioratkaisun arkkitehtuuria. (Kent, 2007.)

Fewster ja Graham (1999) kuvaavat käsitteellä ”testware” kaikkia automaatioratkaisuun liittyviä artefakteja kuten testiskriptit, testidata, dokumentaatio, tiedostot, testiympäristö ja miten nämä artefaktit liittyvät toisiinsa. Kent (2007) tulkitsee heidän käyttämän testware-termin tarkoittavan juuri testiautomaatiokehystä. Hän myös toteaa, että testiautomaatiokehys on tällä hetkellä yleisin käytetty termi.

Dustin, Garret ja Gauf (2009, 146-149) liittävät testiautomaatiokehyksen määritelmän esimerkkitapauksessa käytettyyn työkaluun (STAF), joka toteuttaa testiautomaatioarkkitehtuurin eriyttämällä testien suorituksen ja testausvälineet alemmalle tasolle arkkitehtuurissa. Testiautomaatiokehys toteuttaa arkkitehtuurin ylemmän tason jolla hoidetaan testidatan käsittely, testien suorituksen hallinta, testien raportointi sekä tyypillisesti testiympäristön alustus ja palautus toimet. Kehyksen lisäksi tarvitaan testausväline jota kehys ohjaa ja jolla testin suoritus käytännössä tapahtuu.

Tässä tutkielmassa testiautomaatiokehyksellä tarkoitetaan edellä määriteltyä toiminnallisen testauksen automaatiokehystä jolla voidaan kontrolloida toiminnallisia, käyttöliittymän tai ylemmän tason rajapinnan (esim. web service) kautta tapahtuvia testejä. Se ei tarkoita yksikkö- ja komponenttitestaukseen käytettäviä kehyksiä (esim. xUnit), joilla testataan yksittäisten luokkien metodeja, eikä kokonaistoiminnallisuuksia.

## 2.5 Testausvälineet ja testiautomaatiokehys

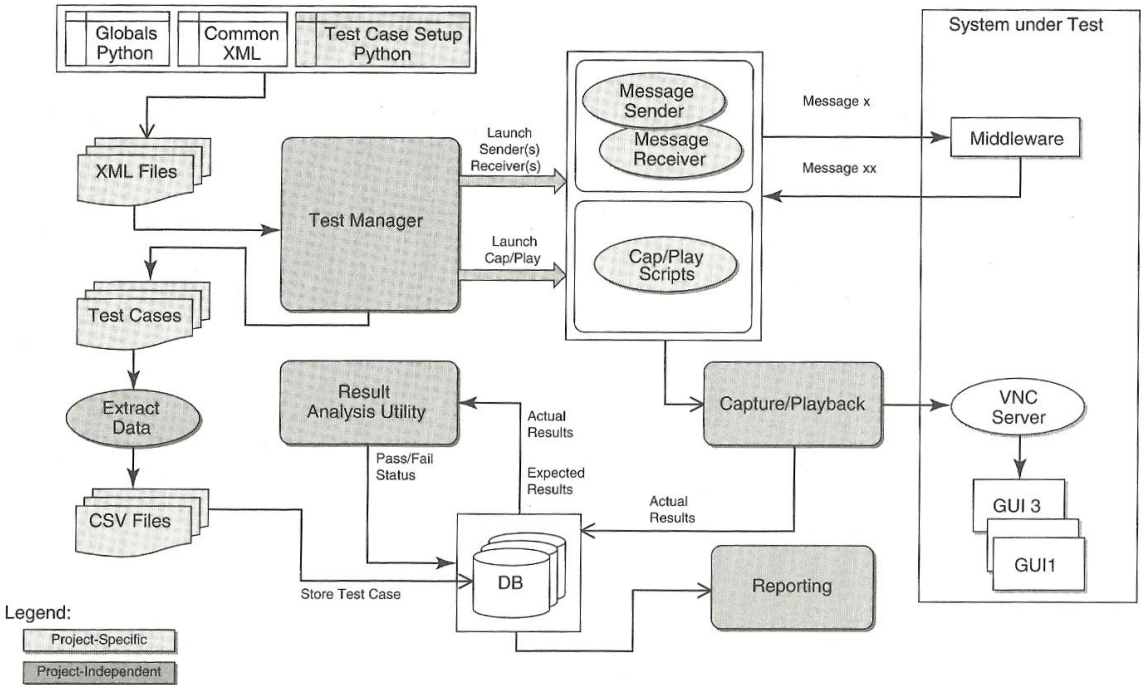
Testiautomaatiokehyksen toimintaa ja rajausta on kuvattu aiemmin luvussa 2.4. Tässä luvussa kuvataan kehyksen ja testausvälineen suhde toisiinsa ja mitkä niiden roolit ovat testiautomaatio-ratkaisussa.

Testiautomaatiokehys on automaatio-arkkitehtuurin ylempi taso jonka tehtävänä on kontrolloida testien suoritusta. Testausväline on loogisesti erillinen testiautomaatiokehyksestä, ja sen rooli arkkitehtuurissa on ajaa automaattiset testit testauksen kohteena olevaan järjestelmään. Testiautomaatiokehys on

alusta jonka päällä yksittäiset testausvälineet voivat suorittaa testejä yhteneväisellä tavalla välineriippumattomasti. Testausvälineen ja testiautomaatiokehyksen suhde toisiinsa voi kuitenkin hämärtyä käytännön toteutuksissa, sillä kehyksen mukaisen arkkitehtuurin voi toteuttaa myös jollain käytetyistä testausvälineistä. Toisaalta kehys voidaan toteuttaa myös valmiilla siihen tarkoitetulla välineellä tai tehdä se itse jollain skriptauskielellä. Voidaankin sanoa että testausvälineen ja testiautomaatiokehyksen erottaa niiden sijainti testiautomaation arkkitehtuurissa. (Nagle, 2015.)

Testiautomaatiokehyksen kannalta on oleellista, että kehyksen tulee kyetä ajamaan eri testausvälineillä kehitettyjä testejä saman testijoukon suorituksessa (Dustin ym., 2009, 146). Testitapauksen automaation suorituksessa voidaan esimerkiksi ensin ajaa osa testistä rajapinnan kautta ja sen jälkeen jatkaa sitä web-käyttöliittymän kautta toisella testausvälineellä.

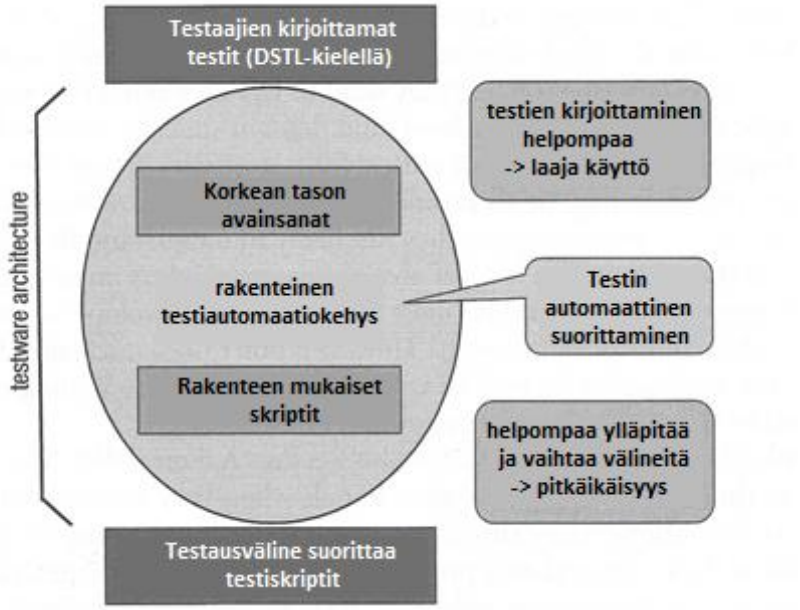
Dustinin ym. (2009) kirjassa on esimerkki testiautomaatiokehyksen ja testausvälineen sijoittumisesta arkkitehtuurista (Kuva 3). Kuvassa testausprojektista riippumattomat (project-independent) osat ovat kehyksen toiminnallisuutta, kun taas projektista riippuvat (project-specific) osat ovat testausvälineen osuutta, testiskriptejä sekä testeihin liittyvää testidataa ja muuttujia. Kehys toimii ajoalustana eri testausvälineissä toteutetuille testiskripteille.




Kuva 3. Testiautomaatiokehyksen arkkitehtuuri (Dustin ym. 2009)

Graham ja Fewster (2012, 9) esittelevät kehyksen arkkitehtuurista yksinkertaistetun version jossa testausväline eriytetään alemmalle tasolle arkkitehtuurissa niin, että testaajien näkemät avainsanat toteutetaan testiautomaatiokehykselle. Avainsanojen tekninen toteutus tehdään arkkitehtuurin alemmalla tasolla olevilla testausvälineen skripteillä. Kehys toteuttaa avainsanojen käsittelyn ja vastaavien skriptien ajamisen, testidatan käsittelyn ja syöttämisen skrip-

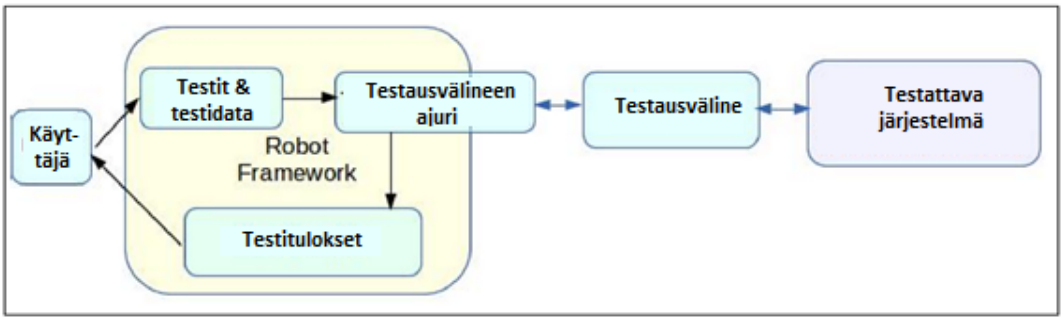
teille sekä muita kehykselle kuuluvia toiminnallisuuksia. Uusia testausvälineitä voidaan lisätä kehyksen piiriin kunhan ne täyttävät kehysarkkitehtuurin vaatimukset.




Kuva 4. Testiautomaatiokehyksen (TestWare) arkkitehtuuri (Graham & Fewster, 2012, 9)

Bisht (2013, 9-11) avaa automaatio-arkkitehtuuria Robot Framework tuotteen konseptikuvalla (Kuva 5). Siinä kehyksen ja testausvälineen suhde tulee hyvin esiin, työnjako on seuraava:

- Testit ja testidata (Test & Test Data). Testien konfiguraatio ja testit, jotka ovat kuvattuna avainsanoilla käyttäen testidataa.
- Testitulokset (Test Results). Kehyksen keräämät testitulokset sisältävät testien suoritustiedot sekä tarvittavat lokitiedostot.
- Testausvälineen ohjain (Testing tool driver). Toteuttaa kehyksen rajapinnan ja kommunikaation eri testausvälineisiin. Kullekin testausvälineelle voidaan räätälöidä oma ohjain jossa välineestä riippuvat asiat on toteutettu.
- Testausväline (Testing tool). Suorittaa testit testattavaa järjestelmää vasten testiskripteihin koodatun logiikan mukaan.
- Testiautomaatiokehys (Robot Framework). Hoitaa testien suorituksen koordinaation ja testidatan syöttämisen testiin sekä testitulosten keräämisen ja raportoinnin.




Kuva 5. Robot Framework kehystuotteen suhde testausvälineeseen (Bisht 2013, 10)

Testausvälineitä on tänä päivänä runsaasti markkinoilla. Välineitä on olemassa kaikkiin testausvaiheisiin ja –tarkoituksiin. Erilaisiin testaustavoitteisiin tarkoitettuja testausvälineitä ovat muun muassa (Hass, 2008, 373-388):

- Rajapintatestausvälineet joilla testataan esimerkiksi web service rajapinnan XML-pohjaista sanomanvaihtoa. Välineitä tähän ovat muun muassa SoapUI, SilkTest, SoapTest ja SoaTest.
- Graafisen käyttöliittymän testausvälineet joilla voidaan tallentaa ja toistaa käyttöliittymässä tehtyjä toimenpiteitä. Välineitä ovat muun muassa QuickTestPro, Abbot, eggPlant ja Ranorex.
- Web-käyttöliittymän testaukseen erikoistuneet välineet kuten Selenium, Watir, Squish ja Helium.
- Suorituskykytestausvälineet joilla voidaan testata liiketoimintalogiikan tai web/GUI käyttöliittymän kautta järjestelmän suorituskykyä. Välineitä ovat muun muassa LoadTester, SilkPerformer, IBM Rational Performance Tester ja JMeter.
- Yksikkö- ja komponenttitestausvälineet joilla kehittäjä voi testata alatason metodien toimivuutta. De facto välineenä testeihin on eri ohjelmointikielien versiot xUnit välineistä, esimerkiksi Javalle JUnit.

Tarkkaa jakoa erityyppisiin testausvälineisiin on vaikea tehdä yksiselitteisesti. Yleensä testausväline on optimoitu toimimaan tietyn tyyppisissä testeissä parhaiten mutta siihen on saatettu rakentaa tukea myös toisen tyyppiseen testaukseen. Esimerkiksi käyttöliittymän testaamiseen tarkoitettu väline voi sisältää tuen myös sanomarajapinnan testaukseen. (Hass, 2008, 370-383.)

Testiautomaatiokehyksen toimintoja toteutettavia välineitä on saatavilla avoimeen lähdekoodiin pohjautuvia. Eri lähteissä kuvattuja kehyksiä ovat:

- Concordion (Hendrickson 2008)
- FIT (Crispin & Gregory 2009)
- Fitnesse (Crispin & Gregory 2009)
- Robot framework (Bisht 2013)
- STAF/STAX (Dustin ym. 2009)
- Cucumber (Bowers & Bell 2013)

# 3 KEHYSTEN TYYPIT JA TOIMINTAPERIAATTEET

Testiautomaatiokehys voidaan rakentaa monella eri tavalla. Erilaisia kehyksiä on esitelty tutkimuksissa joissa on tehty tiettyyn ympäristöön soveltuva automaatiokehys (mm. Peltola, Sierla & Vyatkin, 2014). Kehys voidaan rakentaa itse juuri omaan ympäristöön sopivaksi, käyttää avoimen lähdekoodin tai kaupallisia tuotteita. Omassa kehityksessä hyvänä puolena on soveltuvuus omaan testiympäristöön ja täysi kontrolli siihen tehtävistä muutoksista. Myös avoimen lähdekoodin tuotteissa voidaan haluttua toiminnallisuutta rakentaa yleensä GPL-lisenssin puitteissa jolloin kontrolli kehitykseen säilyy. Toisaalta kehyksen oppimiskäyrä vaikeuttaa käyttöönottoa, ja usein valmiissa kehyksissä on ominaisuuksia joita ei omassa ympäristössä tarvita. (Crispin & Gregory, 2009, 311-316.)

Testiautomaatiokehyksien jaotteluperusteita on useita riippuen kehyksen toimintaperiaatteista. Kattava jaottelu, joiden tarkempia tapausesimerkkejä ja toimintaperiaatteita on esitelty myös muissa lähteissä, löytyy Kellyn (2003) artikkelista:

- Modulaarinen kehys (Test script modularity)
- Testikirjastopohjainen kehys (Test Library Architecture)
- Avainsanapohjainen kehys (Keyword driven)
- Datapohjainen kehys (Data driven)
- Hybridi kehys (Hybrid)

Listan neljää ensimmäistä kehys-tyyppiä voidaan pitää myös suunnitteluperiaatteina, joita noudattamalla saadaan tehtyä käyttökelpoinen ja kustannustehokas hybridi-tyyppinen kehys (Kelly 2003). Tässä pääluvussa käydään läpi edellä mainittujen kehys-tyyppien piirteet ja suunnitteluperiaatteet.

## 3.1 Modulaarinen kehys

Testiskriptien modularisointi tarkoittaa automaatiotestin jakamista pieniin mutta merkityksellisiin toiminnallisuuksiin joita yhdistämällä saadaan koostettua

laajempia testitapauksia ja testijoukkoja. Modulaarisuuden kantava ajatus on uudelleenkäyttö ja ylläpidettävyys niin, että rajatun toiminnallisuuden testaava testiskripti pidetään yllä yhdessä koodirakenteessa (Kelly, 2003, Graham & Fewster, 2012, 8-10).

Modulaarisuutta voisi havainnollistaa esimerkillä (sovellettu Kellyn 2003 esimerkistä) jossa web-sivuston testauksessa joudutaan useassa testitapauksessa ensin kirjautumaan palveluun. Kirjautumistoiminnosta tehdään modulaarinen testiskripti, jota voidaan kutsua useissa testitapauksissa. Esimerkiksi käyttäjän osoitteen muutos ja salasanan vaihto ovat erillisiä testitapauksia, jotka molemmat kutsuvat kirjautumisskriptiä aluksi ja jatkavat sitten kutsumalla muita modulaarisia toimintoja. Jos kirjautumistoiminto testattavassa ohjelmistossa muuttuu, voidaan testiskriptin muutokset tehdä vain kirjautumisskriptiin, jonka jälkeen kaikki sitä käyttävät testitapaukset toimivat taas oikein.

Testiskriptin modulaarisuus on hyvä esimerkki testiautomaation ohjelmistokehitys luonteesta, hyvät ohjelmistokehityskäytännöt pätevät suoraan myös testiautomaatiossa (Kelly 2003). Graham ja Fewster (2012, 12) liittävät modulaariseen kehitykseen myös virheistä palautumisen, konfiguroitavuuden, tulosten raportoinnin sekä riittävän dokumentaation skriptin toiminnasta.

## 3.2 Testikirjastopohjainen kehys

Testikirjastoilla voivat laajentaa modulaarisuutta testiautomaatiossa. Testikirjastoihin voidaan toteuttaa yleiskäyttöisiä funktioita joita hyödyntämällä testiskriptin toteuttaminen on helpompaa. Kirjastossa voi olla funktioita esimerkiksi graafiseen käyttöliittymätestaamiseen liittyen, kuten ikkunoiden sulkeminen, ikkunan tai kontrollin aktivointi, sekä aktiivisuuden tarkistus. Nämä toiminnot ovat kehyksen käyttämien testausvälineiden teknisiä rutiineja joita voidaan kutsua välineellä kehitetyissä skripteissä tai kehyksen kautta. (Kelly, 2003, Nagle, 2015, Kent, 2007.)

Edellisessä luvussa käytetyssä kirjautumisesimerkissä kirjastofunktiota voitaisiin käyttää esimerkiksi kirjautumissivuston latautumisen tarkistamiseen, tarkastamalla funktion avulla, että käyttäjätunnus ja salasana kontrollit ovat sivulla aktiivisena. Samaa funktiota voitaisiin käyttää myös osoitetiedon kontrollin aktiivisuuden tarkistamiseen.

## 3.3 Avainsanapohjainen kehys

Avainsanoihin pohjautuva testiautomaatiokehys tarkoittaa testausvälineen abstraktointia kehyksessä. Kehykseen toteutetaan mekanismi avainsanoilla nimettyjen testiskriptien suoritukseen niin, että skriptien tekninen toteutus piilotetaan avainsanojen taakse. Testitapaukset voidaan siten muodostaa samantyyppisesti kuin manuaalisessa testauksessa. Avainsanat koostetaan esimerkiksi

taulukkoon jonka testiautomaatiokehys käy suorituksen yhteydessä läpi ja ajaa avainsanoja vastaavat testiskriptit. (Kelly, 2003.)

Kentin (2007) artikkelissa hahmotetaan avainsanapohjaista testausta periaatteella jossa testaaja voisi avainsanoja käyttäen tehdä testitapaukset valmiiksi jo ennen sovelluksen testiskriptien valmistumista. Testin suoritusjärjestys ja toiminnallinen logiikka koostetaan avainsanojen avulla taulukkoon jota vasten sovellusta testaavat testiskriptit voidaan toteuttaa.

Avainsanapohjainen kehys perustuu tyypillisesti edellisissä luvuissa avattuihin periaatteisiin modulaarisista testiskripteistä ja testifunktioista. Avainsanaa vastaava testiautomaation toiminto koostetaan modulaarisista skripteistä. (Kelly, 2003.) Alla olevassa kuvassa on esimerkki taulukkoon koostutuista avainsanoista jotka muodostavat ajettavan testijoukon.

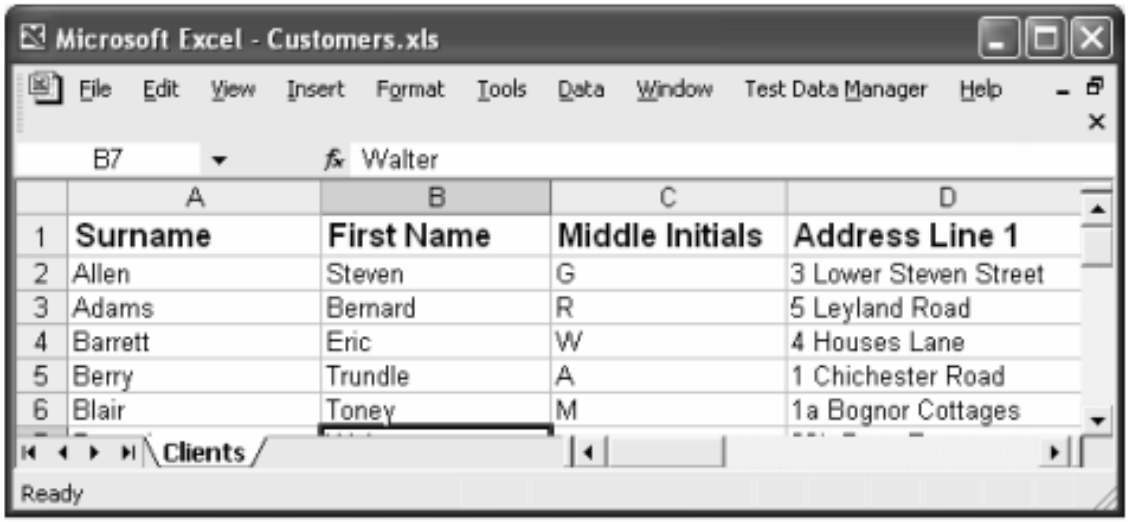
| #Supplier_Add | Sup Name | Description | Type | External Flag |
|---|---|---|---|---|
| Supplier_Add | SC-SUP00 | SC-SUP00 Supplier Company | | OFF |
| Supplier_Add | SC-SUP01 | SC-SUP01 Supplier Company | | ON |
| # Loc_Add | Loc Name | Description | Corp | Corporate Y/N |
| Loc_Add | SC-SXX-1 | Test LO Stock Room only | ON | OFF |
| # | SI Name | Item Description | Name | External Flag |
| Stock_Item_Add | SC-STCK00 | CABLE | COAX | Test SI. |
| Stock_Item_Add | 99 | PUBLICATION | BOOK | Wind in the ... |
| # | SI Name | Description | Type | Quantity |
| Stock_Loc_Add | SC-STCK00 | SC-SXX-1 | 1-01 | 10 |
| Stock_Loc_Add | 99 | SC-SXX-1 | 1-01 | 10 |

Kuva 6. Avainsanapohjaisen testiautomaation taulukko (Kent, 2007)

## 3.4 Datapohjainen kehys

Datapohjaisen kehyksen periaate on, että käytettävä testidata on erotettu testiskriptin toimintalogiikasta. Testiskriptin käyttämä data luetaan tiedostosta tai tietokannasta testin suorituksen yhteydessä ja sijoitetaan skriptin käyttämiin muuttujiin. Kehyksen tarkoitus on mahdollistaa saman testin suoritus helposti useilla eri arvoilla, esimerkiksi raja-arvotestauksessa. Datapohjaista kehystä voidaan hyödyntää myös suorituskykytestauksessa koska sillä voidaan helposti generoida useita ajettavia testitapauksia. (Kelly, 2003.)

Kentin (2007) mukaan datapohjaisella kehyksellä voidaan saavuttaa parempi testikattavuus kun voidaan helposti testata sekä sallitut, että epäkelvot arvot. Myös lokalisointi testauksessa voidaan saavuttaa etuja kun eri kielillä olevat tiedot, esimerkiksi maiden nimet, voidaan tarkistaa datapohjaisella testauksella (Boisschot, 2012). Oheisessa kuvassa on esimerkki data-pohjaisen testauksen tauluko€sta jonka kautta testien eri arvot syötetään käyttöliittymän kenttiin.



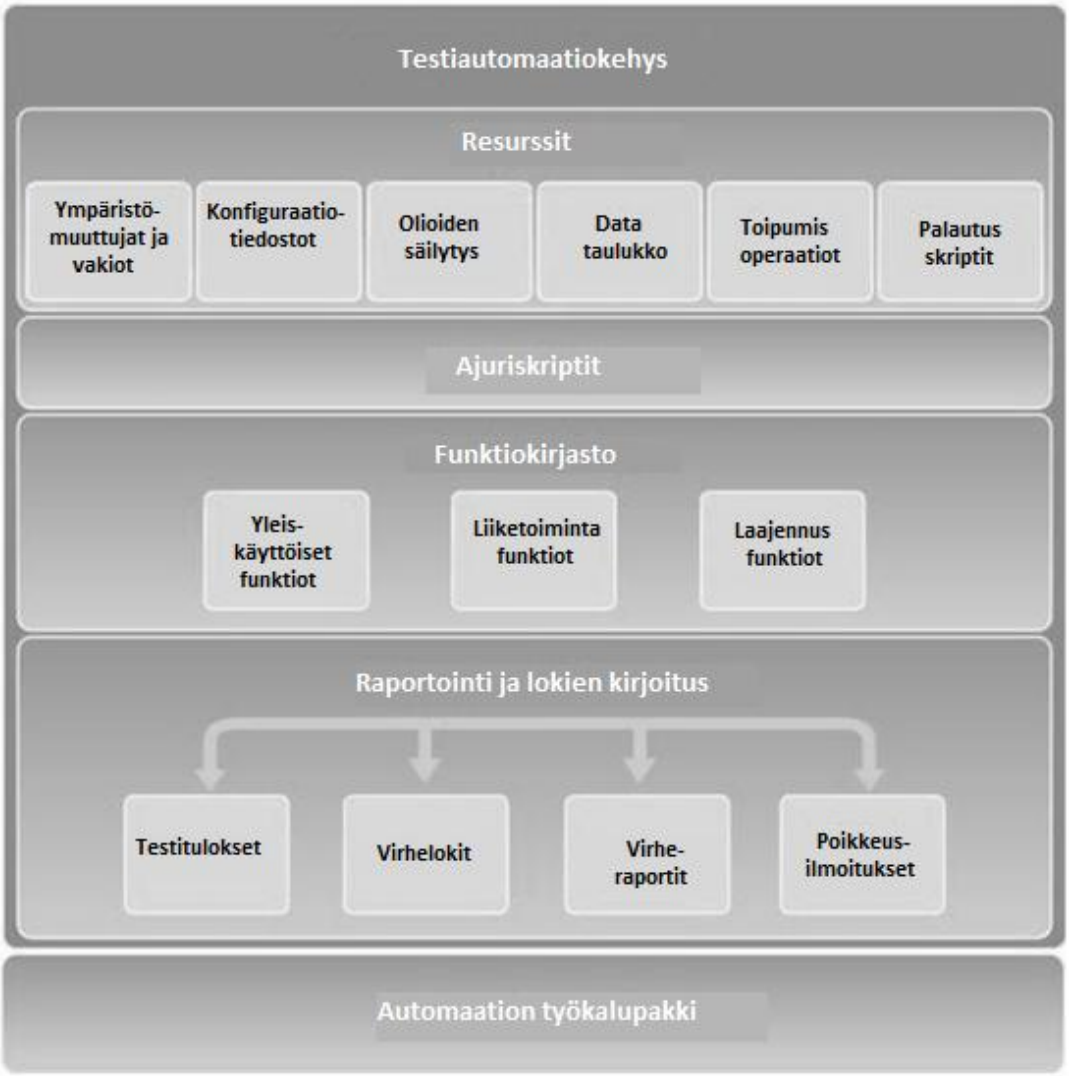

Kuva 7. Datapohjaisen testauksen arvojoukko Excel-tiedostossa (Kent, 2007)

## 3.5 Hybridi kehys

Hybridi kehys ei aiemmista kehystyypeistä poiketen tuo uusia suunnitteluperiaatteita kehyksen toteutukseen, vaan on yhdistelmä aiemmin mainituista. Kehys käyttää hyväkseen kaikkien aiemmin mainittujen kehystyyppien suunnitteluratkaisuja tuottaakseen monipuolisimman toteutuksen. Siinä testiskriptit tehdään modulaarisesti ja tiettyjä rutiineja kirjastoidaan uudelleenkäytettäviksi osiksi. Hybridi toteuttaa testidatan hallinnan niin, että sitä voidaan varioida helposti ja ajaa testejä useilla eri testidatoilla. Se abstraktoi skriptien tekniset yksityiskohdat avainsanoihin joita käyttämällä saadaan liiketoimintalogiikan skenaario testattua. (Kelly, 2003, Kent, 2007)

Tänä päivänä hybridi-kehys on kaikkien yleisin kehystyyppi. Siinä toteutuu lukuisissa eri projekteissa vuosien saatossa havaitut hyvät suunnitteluratkaisut. Testattavasta järjestelmästä riippuu mitkä suunnitteluperiaatteet ovat hallitsevia ja mihin panostetaan eniten, mutta modernissa kehyksessä näistä kaikista löytyy piirteitä. (Owaise, 2013.) Hybridin suosiota kuvastaa se, että useissa käytetyissä kirjallisuuslähteissä viitataan edellä läpikäytyihin suunnitteluperiaatteisiin puhuttaessa tehokkaasta testiautomaatiokehyksestä (esimerkiksi Dustin ym., 2009, 139-149, 227, Graham & Fewster, 2012, 8-12, Li & Wu, 2005, 16-17.) Useissa lähteissä myös viitataan avainsanapohjaiseen kehykseen pohjaoletuksella, että siinä on jo modulaarisuus ja testikirjasto periaatteet käytössä (mm. Pajunen, 2011).

Vaikka hybridi kehys ei tuokaan testikehyksen toiminnalliseen arkkitehtuuriin uusia suunnitteluperiaatteita, voidaan hybridi kehykseen ajatella kuuluvaksi uutena elementtinä muu testikehyksen toiminnallisuus kuten tulosten raportointi, testiympäristön konfigurointi, testidatan generointi sekä virhetilanteista toipuminen (Owaise, 2013). Alla olevassa kuvassa on esimerkki testiautomaatiokehyksestä jossa edellä mainittuja suunnitteluperiaatteita ja toiminnallisuuksia on huomioitu.




Kuva 8. Esimerkki testiautomaatiokehys jossa suunnitteluperiaatteita otettu huomioon (AutomationPlace, 2014)

# 4 VAATIMUKSET TESTIAUTOMAATIOKEHYKSELLE

Tässä pääluvussa kuvataan testiautomaatiostrategia sekä sen vaikutus testiautomaatiokehyksen vaatimuksiin. Kehykselle asetetut vaatimukset listataan eri lähteistä ja perustellaan niiden merkitys automaatioratkaisun toteutuksessa.

## 4.1 Testiautomaatiostrategia ja vaikutukset vaatimuksiin

Ennen toiminnallisen testiautomaatiokehyksen vaatimusmäärittelyä, tulee testiautomaation strategia olla selvillä. Pitää tietää mitä testiautomaatiolla tavoitellaan, mitä osa-alueita järjestelmän testauksesta automatisoidaan ja mitä jätetään manuaaliseen testiin. Miten eri testauksen tasot yksikkötesteistä hyväksyntätesteihin aiotaan suorittaa automaation kannalta. Halutaanko, että kaikki toiminnalliset testi ovat automaation piirissä vai riittääkö, että yksikkötestit ovat kattavat ja käyttöliittymän kautta testataan vain perustoiminnallisuus. Miten eri testaustasojen automatisoinnit vaikuttavat toisiinsa, onko esimerkiksi järkevää tavoitella raja-arvotestejä käyttöliittymän kautta jos ne joka tapauksessa testataan yksikkötesteissä. (Crispin & Gregory, 2009, 273-324.) Crispin ja Gregory (2009, 280-283) analysoivat testauksen eri tehtävien ja vaiheiden kautta mitä kannattaa automatisoida. He listaavat seuraavat alueet joita on syytä pohtia:

- Jatkuva integrointi, paketointi ja asentaminen
- Yksikkö- ja komponenttitestit
- Rajapintatestaus
- Käyttöliittymän testaus
- Suorituskykytestaus
- Testidatan generointi

*Jatkuva integrointi, paketointi ja asentaminen* ovat Crispin ja Gregoryn mukaan keskiössä kun mietitään testiautomaatiota. Ohjelmiston koonto täytyy tapahtua automaattisesti niin, että kaikki ohjelmistokomponentit saadaan integroitua ohjelmistojulkaisuksi niin, että oikeat, keskenään yhteensopivat komponentit käännetään ja muodostetaan asennettavaksi paketiksi. Automaattinen testiympäristöön asentaminen on tärkeä osa prosessia jolla varmistetaan, että julkaisu voidaan samassa jatkuvan integroinnin prosessissa myös testata alustavasti.

*Yksikkö- ja komponenttitestit* ovat tärkeä osa testausstrategiaa jolla varmistetaan muutosten jälkeinen nopea palaute kehittäjille. Ilman näitä alemman tason testejä, toiminnallinen testiautomaatiokaan ei onnistu kannattavasti ohjelmiston heikon laadun vuoksi.

*Rajapintatestaus* on toiminnallisen testauksen alue joka on melko helposti automatisoitavissa. Testiautomaatiostrategiaa mietittäessä, sen merkitys korostuu koska testit ovat käyttöliittymätestejä helpommin ylläpidettäviä.

*Käyttöliittymän testaus* voidaan Crispin ja Gregory mukaan jakaa käyttöliittymä-kerroksen alla tapahtuvaan ja käyttöliittymässä tapahtuvaan testaukseen. Käyttöliittymä-kerroksen alla tapahtuva automaatiotestaus on helpommin ylläpidettävää koska käyttöliittymän pienet muutokset eivät vaikuta testeihin.

*Suorituskykytestauksen* läpivieminen vaatii tyypillisesti myös automatisointia koska kuormitustilanteiden luominen ilman automaatiovälinettä saattaa olla mahdotonta. Automaatiokehyksen vaatimuksissa myös suorituskykytestit on syytä huomioida.

*Testidatan generointi* voi usein vaatia paljon manuaalista työtä. Siksi datan luonti ja alustus ovat hyviä kandidaatteja automatisoitavaksi. Myös testidatansiivous testin jälkeen on syytä ottaa huomioon kehyksen vaatimuksissa.

Baxter, Flynn, Wills ja Smith (2012, 469) tapauskuvauksessa testauksen automatisoitavaa osuutta lähestytään testitapausten luonteen näkökulmasta. Lähtökohta heidän listaukselleen on toiminnallisessa testauksessa jonka automatisoivia kohteita voidaan listauksen avulla tunnistaa:

- Onko todennäköistä, että testi toistuu usein?
- Tarvitaanko testin suoritukseen useita henkilöitä?
- Kuinka paljon aikaa voi säästyä automatisoinnilla?
- Onko testaustehtävä kriittisellä polulla?
- Mikä on riski sille, että testattava vaatimus muuttuu?
- Onko testi yhteinen toisen projektin kanssa?
- Onko testin suoritus herkkä inhimilliselle virheelle?
- Onko testi paljon aikaa vievä?
- Vaatiiko testin suoritus erikoisosaamista, jota on pyydettävä toiselta sidosryhmältä?

Tässä luvussa mainitut testiautomaatio-strategiaan liittyvät näkökulmat on hyvä ottaa tarkasteltavaksi toiminnallisen testauksen automaatiokehyksen suunnittelussa. Vaikka kehyksen päätarkoitus on suorittaa toiminnallisia testejä, on sen hyödyntäminen jatkuvan integroinnin prosessissa, suorituskykytestauk-

sessa, testidatan generoinnissa sekä muissa testauksen osa-alueissa, otettava huomioon.

## 4.2 Testiautomaatiokehyksen vaatimukset

Tutkielmassa valittu näkökulma testiautomaation vaatimuksiin ei lähteiden perusteella ole yleinen. Tyypillisesti kirjallisuudessa ja tutkimusartikkeleissa tarkastellaan testiautomaation vaatimuksia kokonaisuutena jossa testausvälineen osuus on merkittävä. Kuitenkin systemaattisen ja tuloksellisen testiautomaation edellytys on, että erilaisia testiautomaation välineitä voidaan käyttää saman kehyksen yhteydessä saumattomasti. (mm. Peltola, 2014, Dustin ym., 2009, 306-324.)

Tässä keskitytään tarkastelemaan vaatimuksia testiautomaatiokehyksen kannalta. Usein vaatimukset ovat osin päällekkäisiä testausvälineen vaatimusten kanssa. Merkittävämpänä erona testausvälineen ja kehyksen välillä on niiden sijainti testiautomaation arkkitehtuurissa. Testausvälineen vastuulla on toteuttaa testin suoritusta testattavaa järjestelmää vasten. Kehyksen rooli on kontrolloida testien suoritusta ylemmällä tasolla niin, että yksittäiset toiminnat ja testitapaukset tulevat suoritetuiksi oikeassa järjestyksessä raportoiden tuloksia keskitettyyn paikaan. Kehys voi hoitaa myös muuttujien välitystä testitapausten ja testijoukkojen välillä jotka voivat olla toteutettu eri testausvälineillä. (Dustin ym., 2009.) Kehyksen ja testausvälineen suhdetta on kuvattu perusteellisemmin luvussa 2.5.

Tässä luvussa ja aliluvuissa on jaoteltu vaatimukset kehyksen eri toiminnallisuuksien pohjalta ottaen huomioon myös testausvälineen ja kehyksen rajapinnassa olevat vaatimukset. Täysin kattavaa yleistä listausta ei ole mahdollista tehdä, vaan vaatimukset ovat aina osin sovellusaluekohtaisia. Kunkin testiympäristön vaatimukset tuleekin analysoida aina tapauskohtaisesti hyödyntäen esimerkiksi tässä tutkielmassa esitettyä vaatimuslistausta. Aliluvuissa on käytetty vaatimuksille seuraavaa jaottelua:

- Liiketoimintavaatimukset
- Arkkitehtuurivaatimukset
- Testien ajamisen vaatimukset
- Testien hallinnan ja seurannan vaatimukset
- Tulosten raportoinnin vaatimukset
- Testiympäristön hallinnan vaatimukset
- Muut vaatimukset

### 4.2.1 Liiketoimintavaatimukset

Liiketoimintavaatimukset määrittävät yleisesti organisaation tavoitteleman hyödyn testiautomaatiosta. Nämä korkeamman tason vaatimukset ovat yhteisiä

sekä testiautomaatiokehykselle, että suoritusvälineille, ja ohjaavat automaatio-arkkitehtuuria sekä välinevalintoja. (Dustin ym., 2009, 51-53.)

Dustin ym. (2009, 53-55) painottavat saavutettuja hyötyjä liiketoiminnalle lyhyemmän testaussyklin (nopeampi markkinoille tulo), testauksen kustannuksen laskun sekä laadun parantumisen ansiosta. Nämä liiketoiminta-lähtöiset vaatimukset toistuvat useissa lähteissä (mm. Cervantes, 2009; Peltola, 2014) ja muodostavat ylimmän vaatimustason testiautomaatiolle.

Tarkempi liiketoimintavaatimusten lista löytyy Gregoryn ja Crispinin (2014, 256) kirjasta jossa määritellään seuraavat ylemmän tason vaatimukset automaatiolle:

- Automaation tulee mahdollistaa julkaisujen tahdissa pysyminen.
- Automaation käytön tulee onnistua erilaisten tiimien toimesta ilman erikoisosaamista.
- Testiautomaation tulee olla joustava ja sen tulee voida vastata muutoksiin testiympäristössä, testidatassa ja teknisissä rajoitteissa.
- Automaattisen testitapauksen luonti tulee olla suhteellisen helppoa ja intuitiivista.
- Automaattisten testien tulee olla helposti luettavia ja ymmärrettäviä kelle tahansa.
- Testien tulee olla helposti saatavilla eri tiimeille ja niiden jäsenille. Kuka tahansa voi lisätä, katsella ja ajaa testejä.
- Testien suoritus pitää olla nopeaa ja luotettavaa.

Bisht (2013, 8) lähestyy ylemmän tason vaatimuksia hyväksyntätestauksessa saavutetun hyödyn kannalta. Kirjassa listataan seuraavat vaatimukset ja odotukset testiautomaatiolle:

- Sovelluksen virheiden paikantaminen. Automaatiolla voidaan löytää virheet toiminnallisuudessa ja suorituskyvyssä.
- Testauksesta aiheutuvien virheiden väheneminen. Manuaalinen testaus on herkkää inhimillisille virheille jotka voidaan automaatiolla pois sulkea.
- Usein toistuvien töiden väheneminen. Automaatiolla voidaan korvata usein toistuvat työt ajamalla kerran rakennettuja testejä toistuvasti uudelleen.
- Mittausdatan tuottaminen. Testiautomaation raporttien dataa voidaan käsitellä tuottamaan hyödyllisiä mittareita joilla toimintaa voidaan ohjata.

Liiketoimintavaatimukset tulee ottaa huomioon sovellusaluekohtaista vaatimuslistaa tehdessä. Niistä ei saa suoraan testiautomaatiokehyksen vaatimuksia vaan ne pitää johtaa arvioimalla vaatimuskategorioittain mitkä ovat vaikutukset.

### 4.2.2 Arkkitehtuuri vaatimukset

Kehyksen arkkitehtuuriin liittyviä vaatimukset ovat sen sisäiseen rakenteeseen liittyviä ratkaisuja joilla tuetaan luvussa 3 kuvattuja kehyksen toiminnallisuuksia. Näitä ovat muun muassa modulaarinen ja laajennettava rakenne, avainsana- ja datapohjaiset suunnitteluperiaatteet sekä uudelleenkäytettävien toimintojen kirjastointi. Arkkitehtuurin vaatimukset liittyvät vahvasti testiautomaation ylläpidettävyyteen. (Dustin ym., 2009, 226-228.)

Testiautomaation kannattavuus saavutetaan parhaiten mitä aiemmin se aloitetaan järjestelmän elinkaaressa. Järjestelmän muuttuessa testien ylläpidon helppous nousee tärkeäksi (Crispin & Gregory, 2009, 269-311). Jotta testiskriptit olisi mahdollista päivittää muuttunutta toteutusta vastaavaksi kustannustehokkaasti, tulee automaatiokehyksen suunnittelussa noudattaa seuraavia periaatteita (Verma, Singh, Verma & Rao, 2010):

- Toteuta arkkitehtuuri uudelleenkäytön näkökulmasta.
- Toteuta uudelleenkäytettäviä toimintoja joita voi hyväksikäyttää eri sovellusten testauksessa.
- Suunnittele testiskriptit komponenteiksi joita voi uudelleenkäyttää.
- Vältä toteuttamasta kertakäyttöisiä automaatiotestejä.

Dustin ym. (2009, 226-228) painottaa, että testiautomaation kehitys on ohjelmistokehitystä joten siinä tulee soveltaa sen parhaita käytäntöjä. Arkkitehtuurin suunnittelu on tärkeää jotta hyväksi havaitut suunnittelumallit tulevat toteutetuiksi. He listaavat seuraavat arkkitehtuuriin liittyvät vaatimukset:

- Avainsanapohjainen arkkitehtuuri. Avainsana piilottaa teknisen toteutuksen alleen ja käytettävää teknologiaa tai testivälinettä voidaan vaihtaa ilman vaikutusta testin logiikkaan.
- Datapohjainen arkkitehtuuri. Testissä käytettävä data luetaan erillisestä tiedostosta ja käytetään avainsanat toteuttavissa modulaarisissa testiskripteissä.
- Modulaarinen kehyksen arkkitehtuuri. Mahdollistaa uusien komponenttien, esimerkiksi testausvälineiden, lisäämisen arkkitehtuuriin.
- Modulaarinen testiskriptien kehitys. Samaa testiskriptin toiminnallisuus toteutetaan vain yhdessä paikassa jossa se on helppo ylläpitää.
- Modulaarinen graafisen käyttöliittymän navigointi. Käyttöliittymässä navigointi tulee erottaa erilliseksi komponentiksi arkkitehtuurissa niin, että testin logiikka voi pysyä samana vaikka käyttöliittymään tulisi muutos.
- Uudelleenkäytettävät toiminnot testikirjastoon. Eri testien suoritus vaatii usein samoja toimintoja jotka kannattaa kirjastoida uudelleenkäytettäviksi.
- Uudelleenkäytettävien ulkoisten kirjastojen käyttö. Tietyllä testivälineellä tehtävä perustoiminto voi olla saatavilla ulkopuolisesta palvelusta jolloin se kannattaa käyttää hyväksi omassa kehityksessä.

- Automaattinen testiautomaatio koodin generointi. Testattavan järjestelmän rajapintojen määrittelytiedostojen (IDL tai C-header) hyväksikäyttö testiautomaatiokoodin generoimiseksi rajapintatestiin.

Edellä kuvatut arkkitehtuuriin liittyvät vaatimukset on syytä huomioida automaatiokehyksessä. Niitä vasten voidaan tarkastella valmiin kehyksen toiminnallisuuksia tai määritellä itse tehtävän kehyksen teknisiä vaatimuksia.

### 4.2.3 Testien suorituksen vaatimukset

Testien suorituksen vaatimukset ovat hajautettuun ajamiseen, ajastukseen, muuttujien välitykseen sekä kehyksen kontrollirakenteisiin liittyviä vaatimuksia. Dustin ym. (2009, 306-324) listaa seuraavia testien suoritukseen liittyviä vaatimuksia:

- Testien rinnakkainen suoritus testiskenaarion aikaansaamiseksi, sekä synkronointi testin aikana.
- Mahdollisuus ajaa testejä eri palvelimelta kuin missä testattava järjestelmä sijaitsee.
- Testin suoritus ei saa vaikuttaa testattavaan sovellukseen tai testiympäristöön.
- Testien ajo eri käyttöjärjestelmiä vasten eri testausvälineillä.
- Testin suoritus jatkuvan integroinnin prosessin yhteydessä eräajona.
- Korkean tason ohjelmointikielen käyttö testien suorituksen kontrollointiin, esimerkiksi prosessien käynnistys, lopetus ja kysely sekä ajonaikaisten muuttujien käyttö
- Testien ajastettu suoritus. Mahdollisuus asettaa ajastin milloin testin suoritus alkaa.

Crispin ja Gregory (2009, 316-317) mainitsevat ohjelmointikielen käytön kehyksessä jolloin testien suoritusta voidaan kontrolloida parhaalla mahdollisella tavalla. Muuttujien käyttö testidatan syöttämiseksi testiskripteille ja tiedon välitykseen testien välillä tulee esiin Cervantesin (2009) STAF-kehystä käsittelevästä artikkelista. Erityisesti globaalien muuttujien käyttö mainitaan tapana jakaa tietoa eri palvelimilla ajettavien hajautettujen testien välillä. Ohjelmointikielen käyttö mahdollistaa myös testien tuloksien analysoinnin kehyksen tasolla ja päätöksen miten testien suoritusta jatketaan riippuen aiempien testien tuloksista. Lisäksi Cervantesin (2009) artikkelissa kuvataan muita käyttökelpoisia STAF-kehyksen testien suoritukseen liittyviä ominaisuuksia:

- Mahdollisuus ajaa testejä usealla eri palvelimilla niin, että testien käynnistys voidaan tehdä testikoneelta.
- Prosessien käynnistys etäpalvelimelle testikoneelta.
- Testien synkronointi testipalvelimien yhtäaikaisen resurssien käytön estämiseksi.

### 4.2.4 Testien hallinnan ja seurannan vaatimukset

Testien hallinta ja ajon seuranta helpottavat testiautomaation kehitystä. Kehykselle voidaan asettaa seuraavia vaatimuksia (Hoffman, 1999):

- Voitava ajaa valittu joukko testejä koko testijoukosta.
- Testin suorituksen aloitus mistä kohdasta tahansa testijoukkoa.
- Testien käynnistyksen jälkeen ei tarvetta puuttua suoritukseen.

Dustin ym. (2009) mainitsee lisäksi vaatimukset:

- Mahdollisuus muokata ajettavia testiskriptejä suoraan testitapauslistalta.
- Testitapausten selailu ja valinta testijoukkoon.
- Testien suorituksen seuranta testien ajon aikana
- Mitkä testeistä on ajettu ja mitkä ajamatta.
- Ajettujen testien tulokset.

Myös muissa lähteissä painotetaan testien hallinnan merkitystä. Bansal ym. (2013) toteaa, että kehyksen tulee tukea tyypillisesti erilaisissa testauksen hallinta välineissä olevia standardi testienhallinta ominaisuuksia. Cervantes (2009) mainitsee graafisen käyttöliittymän jossa ajettavia testitapauksia voi hallita ja tarvittaessa suorittaa esimerkiksi kesken jäänyt tai virheeseen päättynyt testi uudelleen. Crispin ja Gregory (2009, 311 – 321) mainitsevat lisäksi testien versionhallinnan tärkeyden jotta tiedetään aina testattavaa sovellusta vastaava testiautomaation konfiguraatio. He myös listaavat testien hallintaan ja testiautomaatioon liittyviä kysymyksiä jotka on huomioitava kehyksen testien hallinnassa s. 321:

- Mitkä testit on jo automatisoitu ja mitkä ei?
- Mitkä testit ajetaan osana regressiotestejä tällä hetkellä?
- Mitä sovelluksen toiminnallisuuksia automatisoidut testit kattavat?
- Kuinka tietty toiminnallisuus toimii?
- Kuka kirjoitti testin tai muutti sitä viimeksi?
- Kuinka kauan testiä on ajettu regressio-testijoukossa?

### 4.2.5 Tulosten raportoinnin vaatimukset

Testiautomaatiossa tulosten raportointi ja automaattinen analysointi on keskeistä koska testien suorituksesta tulee usein paljon analysoitavaa dataa, jonka tarkistus ja raportointi manuaalisesti vievät liian paljon aikaa. Kun automaattinen testi suoritetaan, ei testaaja ole havainnoimassa systeemin tilaa ja näkemässä miten testin suoritus onnistui. Automaatiotestien tulee siis itsenäisesti kyetä päättelemään testin suorituksesta, sen tuottamista lokitiedostoista ja herätteistä suoritettiinko testi onnistuneesti vai ei. (Dustin ym., 2009, 13-14; Bansal ym., 2013.)

Testitulosten analysointi on etenkin reaaliaikaisissa järjestelmissä hyvä suorittaa testien suorituksen yhteydessä. Tällöin on todennäköistä, että kaikki tieto analyysin tekemiseksi on olemassa. Analyysin tekeminen saattaa vaatia huomattavan osan testiautomaatioon käytettävästä työstä. (Bansal ym. 2013.) Bansal ym. (2013) listaa seuraavia vaatimuksia tulosten analysoimiseksi:

- Testitulosten organisointi ihmisen ymmärtämään muotoon.
- Testitapauksen helppo ja visuaalinen yhdistäminen testituloksiin.
- Testitulosten hyväksikäyttö testausmetriikoissa.
- Testien tulosten raportointi eri tasoilla niin, että testien onnistumisen näkee helposti mutta tarvittaessa voidaan tarkastella virhetilanteen tietoja raportista.
- Testitulosten tulee olla jossain yleisesti käytetyssä muodossa, esimerkiksi PDF- tai HTML-muodossa.

Dustin ym. (2009, 306-328) esittelemässä tapauksessa käytettiin avoimen lähdekoodin STAF testiautomaatiokehystä jota räätälöitiin vastaamaan raportoinnin vaatimuksiin:

- Testien raportointi ja lokitus testitapaus ja testiaskeleen tasolla.
- Testien tulosten raportointi testitapauskohtaisesti, onnistunut / epäonnistunut suoraan testien hallintatyökaluun.
- Testitulosten automaattinen analysointi ja niitä vastaavat raportit.
- Alustavan virheraportin luominen suoraan virheenhallintavälineeseen testin suorituksen perusteella. Seuraavilla tiedoilla
    - Testitapaus ja testiaskel jossa virhe tapahtui
    - Tarvittavat lokitiedostot
    - Käytetty testidata
    - Virheen kuvaus sisältäen näytönkaappauksen ja muut tarvittavat tiedot
- Mahdollisuus sähköpostin lähettämiseen.
- Eri tason lokien generointi ja tallennus testattavasta järjestelmästä.
- Testitulosten kerääminen hajautetussa ympäristössä yhteen paikkaan.
- Tuki testitulosten pakkaamiseen (zipping).
- Testitulosten linkitys vaatimuksiin, mitkä vaatimukset on testattu onnistuneesti.

Edellä mainittujen vaatimusten lisäksi Gregory ja Crispin (2014, 99, 390-391) asettavat korkeamman tason vaatimuksia raportoinnille. He painottavat testiraporttien helppoa saatavuutta kaikille kehitystiimin jäsenille jotta kaikki osapuolet näkevät tilanteen ja voivat luottaa testauksen laatuun ja kattavuuteen. Lisäksi pääsy testituloksiin, esimerkiksi lokitiedostoihin, on oleellista virhetilanteiden analysoinnin helpottamiseksi. Myös muille kohderyhmille, kuten tuoteomistajalle ja johdolle olisi hyvä pystyä tuottamaan tilannetietoa toiminnallisuuksien ja julkaisujen testauksen tilanteesta.

### 4.2.6 Testiympäristön hallinnan vaatimukset

Testiympäristön konfigurointiin ja hallintaan liittyvien vaatimusten merkitys kasvaa ajettaessa testiautomaatiota toistuvasti testattavaan järjestelmään. Kehyksen tulisi kyetä tarkistamaan testiympäristön valmius testien suoritukseen ja tarvittaessa konfiguroimaan ympäristöä niin, että valittu testijoukko voidaan ajaa testiympäristössä onnistuneesti. (Dustin ym., 2009, 306-328).

Cervantes (2009) kuvaa STAF kehyksen tapaustutkimuksessaan useita testiympäristöön liittyviä toimenpiteitä joita kyseisellä kehyksellä voi tehdä. Kehyksellä voidaan kopioida testissä käytettävät konfiguraatiotiedostot testattavan järjestelmän palvelimille sekä luoda väliaikaisia hakemistoja testituloksia varten. Lisäksi kehyksessä on tuki testattavan järjestelmän tietokannan ja prosessien käynnistämiselle ja lopetukselle sekä testien ajastukselle.

Dustin ym. (2009, 154-156) listaa automaattisesti suoritettavia tarkastuksia ja konfigurointeja ylemmällä tasolla. Testiautomaation on syytä tarkistaa ennen testien ajoa verkkoyhteyksien toimivuus, tietokannan tila, käytettävien ohjelmistojen tila ja versiot sekä laitteisto. Tarkistuksista on hyvä tehdä raportti joka kertoo kattavasti testiympäristön tilan ennen testien suorittamista jolloin ympäristöön liittyvät ongelmat voidaan sulkea pois testien epäonnistuessa. Dustin ym. (2009) listaa alueita jotka on syytä tarkistaa ennen testien suoritusta:

- Laitteiston tila
- Testattavat ohjelmistot
- Käyttöjärjestelmän sovellukset
- Verkkokonfiguraatio
- Käyttöjärjestelmän ydin
- Kiintolevyjen tilat
- Käyttäjän tiedot

Keräämällä tietoa testiautomaation lokitiedostoon, tiedetään aina testituloksia tarkasteltaessa testiympäristön tila. Dustin ym. (2009) ehdottaa myös, että tarkastetusta testiympäristöstä tallennetaan tilannekuva (snapshot) johon vertaamalla voidaan havaita testiympäristön ongelmat.

Vaikka Dustin ym. (2009) ei kuvaakaan samalla tarkkuudella vaatimuksia kuin Cervantes (2009), voidaan ylemmän tason tarpeista johtaa samantyyppisiä alemman tason vaatimuksia kuin Cervantesin (2009) artikkelissa. Lisäksi voidaan tehdä päätelmä, että automaatiokehyksellä tulee olla pääsy yleisesti käyttöjärjestelmän palveluihin jotta se voi tarkastella testiympäristön tilaa ja tehdä tarvittavia konfigurointeja testejä varten.

### 4.2.7 Muut vaatimukset

Edellä kuvattuihin vaatimuskategorioihin kuulumattomia vaatimuksia ovat kehyksen lisensointiin, räätälöintiin, testausprosessiin ja käytettävyyteen liittyvät vaatimukset. Nämä vaatimukset ovat hyvin sovellusaluekohtaisia riippuen

millaisessa testiympäristössä testejä ajetaan ja millainen on testattavana oleva järjestelmä. Tähän lukuun on koottu mitä muita vaatimuksia eri lähteissä on kehykselle asetettu. Tarkat sovellusaluekohtaiset vaatimukset voidaan johtaa näitä vaatimuksia soveltaen.

Lisensointiin ja kehyksen räätälöintiin liittyvät vaatimukset on huomioitava testiautomaatio-kehyksen hankinnassa. Avoimeen lähdekoodiin perustuvat kehykset ovat parhaiten räätälöitävissä omiin tarpeisiin sopivaksi. Usein käy niin, ettei mikään valmis kehys tarjoa kaikkia ominaisuuksia tietylle sovellusalueelle, jolloin mahdollisuus muokata kehystä nousee tärkeäksi ominaisuudeksi. Etenkin uusien testiautomaatiovälineiden integrointi testiautomaatiokehykseen saattaa vaatia kehyksen toiminnallisuuden muokkaamista. Avoimen lähdekoodin välineet ovat myös joustavampia, kun suunnitellaan hajautettuja testejä joissa testaussolmuja on useita ja jokaisessa niistä tulee olla lisenssi kehyksen ajamiseksi. (Dustin ym., 2009, 306-311.)

Crispin ja Gregory (2009, 314-315) vertailevat avoimen lähdekoodin ja kaupallisten tuotteiden eroja. He listaavat avoimen lähdekoodin tuotteiden tyypillisiä piirteitä:

- Välineet on usein tehty testiautomaatio-tiimin toimesta joka on rakentanut ratkaisun omaan käyttöönsä. Tällöin ne palvelevat usein hyvin muidenkin tarpeita.
- Ominaisuudet ovat yleensä kattavia ja ne tukevat usein sekä testaajan, että ohjelmoijan toiveita.
- Osa avoimen lähdekoodin välineistä on huonosti dokumentoituja mutta variaatiota on paljon.
- Valinta kannattaa kohdistaa välineeseen jossa on aktiivinen käyttäjäryhmä joka kehittää välinettä edelleen.

Kaupallisista tuotteista Crispin ja Gregory (2009, 314-315) esittävät että ne ovat periaatteessa varma valinta koska dokumentaatio ja tuki ovat kunnossa. Lisäksi käyttöönotto on usein helpompi etenkin vähemmän teknisille henkilöille. Toisaalta ne eivät yleensä ole ohjelmoija-ystävällisiä, vaan käyttävät usein välinekohtaisia skriptauskieliä jotka tekijöiden pitää opetella alusta alkaen. (Crispin & Gregory, 2009)

Testausprosessiin liittyvistä vaatimuksista Dustin ym. (2009, 159-164) käsittelevät testien linkittämistä vaatimuksiin sekä automaattista virheiden kirjausta. He painottavat vaatimusten linkittämisen tärkeyttä muihin järjestelmäkehityksen lopputuotteisiin. Testausvälineiden tulisi kyetä pitämään yllä vaatimusten jäljitettävyys matriisia (requirements traceability matrix), josta nähdään helposti mitkä järjestelmän vaatimuksista on jo testattu. Matriisissa linkitetään järjestelmän vaatimukset, testitapaukset ja testin tulos. Tuloksena saadaan testauskattavuus vaatimusten osalta, kuinka moni vaatimuksista on testien piirissä ja kuinka monta niistä on testattu onnistuneesti.

Toinen testausprosessiin linkittyvä vaatimus on automaattinen virheiden raportointi. Mahdollisuus automaattiseen virheraporttiin tulisi ottaa huomioon testausvälineissä ja testiautomaatiokehyksessä. Rajapinta virheenkäsittely-

järjestelmään mahdollistaa virheraportin pohjan täyttämisen automaattisesti. Virheiden raportointia ei ole kuitenkaan syytä täysin automatisoida vaan pitää prosessissa kohta jossa testaaja hyväksyy ehdotetun raportin tallentamisen käsittelyjärjestelmään. Näin vältytään kaksoiskappaleiden ja väärien virheiden (false positive) raportoinnilta. (Dustin ym., 2009, 164.) Gamba (2012, 414) mainitsevat myös virheraporttien tuottamisen automaattisesti testiraportin perusteella, jolloin kaikki testiin liittyvä tieto on mahdollista liittää osaksi virheraporttia.

Käytettävyyteen liittyviä vaatimuksia käsitellään Hendricksonin (2008) artikkelissa jossa hän painottaa liiketoiminnan ja tekniikan erottamista testin suunnittelussa. Hän korostaa testin liiketoiminta-logiikan erottamista teknisestä toteutuksesta jossa testausväline painaa esimerkiksi käyttöliittymän nappeja. Tällä erottamisella saavutetaan helppokäyttöisyyden lisäksi mahdollisuus tehdä testin suunnittelu etukäteen, jolloin sovelluksen valmistuttua saadaan automaatiotesti nopeammin ajettavaksi toteuttamalla tekninen osuus. (Hendrickson, 2008.) Myös Graham ja Fewster (2012, 9-11) korostavat testiautomaatioratkaisun helppokäyttöisyyttä ja kuvaavat ajatuksen sovellusaluekohtaisesta kielestä (domain specific language, DSL), jossa liiketoiminnan toiminnot on tulkattu avainsanoiksi joista kieli muodostuu.

Lisää käytettävyyteen liittyviä vaatimuksia on kuvattu Gregoryn ja Crispinin (2014, 256) kirjassa jossa mainitaan että automaatiotestien tekeminen pitäisi olla yksinkertaista ja intuitiivista. Lisäksi testien tulisi olla helposti luettavia ja ymmärrettäviä kenen tahansa toimesta. Niiden suunnittelu käyttäen luonnollista kieltä tukee tätä. Testien pitäisi myös olla kaikkien tiimin jäsenten käytettävissä ja muokattavissa helposti sekä testien suorittaminen pitäisi olla nopeaa ja luotettavaa, jotta nopea palaute toteutuksesta saadaan aikaiseksi.

# 5 YHTEENVETO

Tässä tutkielmassa kuvattiin testiautomaation merkitystä nykyaikaisessa ohjelmistokehityksessä sekä perusteltiin sen tärkeyttä toiminnan tehostamisessa. Sen tavoitteina oli tarkastella testiautomaatiokehyksien toiminnallisuuksia, arkkitehtuureja ja suunnitteluperiaatteita ja määrittää kehyksen rooli testiautomaatioratkaisussa. Tärkeä näkökulma oli myös kehyksen käytön hyödyt ja perustelu miksi sen käyttöönottoon ja kehitykseen on syytä panostaa. Tutkielmassa pyrittiin myös koostamaan mahdollisimman kattavasti testiautomaatiokehyksen vaatimukset jotta eri kehystuotteiden arviointi ja hankinta helpottuisi.

Tutkielmassa kuvattiin testiautomaation merkityksen kasvaminen ketterien menetelmien käyttöönoton myötä. Ketterälle kehitykselle ominaiset tiheät julkaisusyklit ovat johtaneet siihen, ettei testaus pysy syklien perässä niin, että aiempi toiminnallisuus, regressio, tulisi testattua kattavasti. Käytännössä testiautomaatio on regression varmistamiseksi ketterässä syklissä ainoa kustannustehokas ratkaisu.

Tutkielmassa tarkasteltiin testiautomaatiota ja sen suhdetta eri testaustavoitteisiin. Automaation myötä testauksen vanha V-malli ei päde, vaan jaottelu on syytä tehdä uudella tavalla. Tutkielmassa kuvattiin testiautomaation pyramidi jossa eri testaustavoitteet on edustettuina. Alimman tason laajimmat testit tehdään yksikkö- ja komponenttitesteissä, palvelurajapintatestit ovat pyramidin keskellä toiseksi laajimmalla osuudella sekä huipulla käyttöliittymän kautta tapahtuvat testit. Pyramidi kuvaa testiautomaation tavoitehierarkiaa sekä testien määrää kussakin vaiheessa. Alemman tason testit on automatisoitava ensin ennen kuin jatketaan seuraavan tason testien automatisointia. Pyramidin huipun yläpuolella on vielä manuaaliset testit joita optimi-tilanteessa olisi kaikkein vähiten, eli suurin osa testeistä tehtäisiin automaattisesti.

Testiautomaatioratkaisuja ja niiden arkkitehtuuria tarkasteltiin sisältöluvuissa laajasti. Luvuissa määriteltiin testiautomaatiokehyksen ja testausvälineen suhde toisiinsa ja sijainti testiautomaation arkkitehtuurissa. Kehyksen rooli määriteltiin testiautomaation alustaksi jolle eri testausvälineillä tehtyjä testejä voidaan viedä ajettavaksi osana suurempaa testijoukkoa. Kehyksen ominaisuuksia käytiin läpi ja listattiin niille tyypillisesti kuuluvia tehtäviä. Kehyksen

rooli on hallita testien suoritusta, syöttää testidataa modulaarisesti toteutetuille testiskripteille sekä koostaa testien tulokset raporttiin. Lisäksi kehys hallitsee testiympäristön konfiguraation ja testien alustus ja lopetustoimet. Testausvälineen rooliksi jää testien ajaminen testattavana olevaa järjestelmää vasten.

Tutkielmassa käytiin läpi testiautomaatiokehysten tyypit ja niiden suunnitteluperiaatteet. Kehystyypeistä kuvattiin modulaarinen, testikirjastopohjainnen, avainsanapohjainen, datapohjainen sekä näiden yhdistelmä hybridi kehys. Kehystyypeistä havaittiin että ne ovat enemmänkin suunnitteluperiaatteita kuin käytännön kehysratkaisuja. Erityyppisiä kehyksiä on kehitetty aiempien kokemusten perusteella ja tämän päivän onnistuneissa testiautomaatioratkaisuissa käytetään edellä mainittujen kehystyyppien yhdistelmää, hybridi kehystä. Tärkeitä periaatteita kehyksissä ovat uudelleenkäyttö, toiston välttäminen, laajennettavuus sekä alustariippumattomuus.

Testiautomaatiokehyksen vaatimuksia käytiin eri lähteiden pohjalta läpi. Tuloksissa havaittiin että kehyksen ja testausvälineen vaatimukset ovat usein päällekkäisiä ja tarkkaa rajaa, kumman vaatimuksista on kyse, on vaikea vetää. Vaatimukset listattiin testiautomaatiokehyksen ja sen testausvälinerajapinnan kannalta. Tärkeimpiä vaatimuksia edellä kuvattujen arkkitehtuuriperiaatteiden lisäksi olivat ohjelmointikielen käyttö kehyksessä testien suorituslogiikan kuvaamiseksi, pääsy käyttöjärjestelmän palveluihin, hajautettu testien ajo, testien ajastukset, testauksen hallinta ominaisuudet sekä laaja räätälöitävyys. Räätälöitävyydestä johtuen havaittiin että avoimen lähdekoodin kehykset ovat varteenotettava vaihtoehtoa kehykseksi niiden muokattavuuden ansiosta. Myös oman ratkaisun kehitys on vaihtoehto, mutta tänä päivänä on valmiita tuotteita joiden käyttöönotto on todennäköisesti pienempi työ kuin oma toteutus.

Tutkimusta voidaan soveltaa oman testiautomaatioratkaisun suunnittelussa. Tutkielmassa kuvataan testiautomaatiokehys-ratkaisuja ja niiden roolia kokonaisuudessa. Se auttaa tekemään ratkaisuja testiautomaatiostrategian suhteen mitä omalla sovellusalueella kannattaa automatisoida ja miten automatisoinnin voisi toteuttaa. Tutkielmassa listatut kehyksen vaatimukset auttavat määrittämään omia tarpeita ja niistä voidaan johtaa sovellusaluekohtaiset vaatimukset jotka on huomioitava kehystuotteen hankinnassa tai oman kehityksessä.

# LÄHTEET

AutomationPlace blogspot (2014). Automation Frameworks. Haettu 22.3.2015 osoitteesta http://automationplace.blogspot.fi/2014/12/automation-frameworks.html

Bansal A., Muli M., Patil K. (2013). Taming Complexity While Gaining Efficiency: Requirements for the Next Generation of Test Automation Tools. AUTOTESTCON, 2013, 123-129.

Baxter, M., Flynn, N., Wills, C., ja Smith M. (2012). System-of-systems test automation at NATS. Teoksessa D. Graham & M. Fewster (toim.), Experiences of Test Automation: Case studies of Software Test Automation (s. 469). Indiana: Pearson Education.

Bisht, S. (2013). Robot Framework Test Automation: Create test suites and automated acceptance tests from scratch. Birmingham: Packt Publishing.

Boisschot, B. (2012). Test automation of a SAP implementation. Teoksessa D. Graham & M. Fewster (toim.), Experiences of test automation: Case studies of Software Test Automation (s. 317-318).

Bowers, A., Bell, J. (2013). Automated testing with Selenium and Cucumber. Haettu 14.5.2015 osoitteesta http://www.ibm.com/developerworks/library/a-automating-ria/

Cervantes A. (2009). Exploring the Use of a Test Automation Framework. Aerospace conference, 2009.

Cohn, M. (2009). Succeeding with Agile: Software Development Using Scrum. Michigan: Pearson Education.

Crispin L. & Gregory J. (2009). Agile testing: A practical guide for testers and agile teams. Boston: Pearson Education.

Dustin E. & Garret T. & Gauf B. (2009) : Implementing Automated Software Testing: How to Save Time and Lower Costs While Raising Quality: How to Lower Costs While Raising Quality. Boston: Pearson Education.

Dustin E. (2012). Automating the testing of complex government systems. Teoksessa D. Graham & M. Fewster (toim.), Experiences of Test Automation: Case studies of Software Test Automation (s. 140). Indiana: Pearson Education.

Eun Ha K. & Jong Chae N. & Seok Moon R. (2009). Implementing an Effective Test Automation Framework. Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International. (s. 534-538). Seattle: IEEE.

Fewster M. & Graham D. (1999). Software Test Automation: Effective Use of Test Execution Tools. Harlow: Pearson Education.

Gamba, S. (2012). Automating through the back door (By supporting manual testing). Teoksessa D. Graham & M. Fewster (toim.), Experiences of Test Automation: Case studies of Software Test Automation (s. 140).

Graham D., Fewster M. (2012). Experiences of Test Automation: Case Studies of Software Test Automation. Indiana: Pearson Education.

Gregory J, Crispin L (2014). More Agile Testing: Learning Journeys for the Whole Team. Indiana: Pearson Education.

Hass, A. (2008). Guide to Advanced Software Testing. Norwood: Artech House.

Hendrickson, E. (2008). Agile-Friendly Test Automation Tools/Frameworks. Haettu 9.5.2015 osoitteesta http://testobsessed.com/2008/04/agile-friendly-test-automation-toolsframeworks/.

Hoffman, D. (1999). Test Automation Architectures: Planning for Test Automation. Haettu 9.5.2015 osoitteesta http://softwarequalitymethods.com/papers/autoarch.pdf.

Horwath, T. (2007). Automated Test Tools Evaluation Criteria. Haettu 9.5.2015 osoitteesta http://www.slideshare.net/basma_iti_1984/testing-tool-evaluation-criteria-presentation

ISTQB (2015). What is V-model- advantages, disadvantages and when to use it?. Haettu 28.2.2015 osoitteesta http://istqbexamcertification.com/what-is-v-model-advantages-disadvantages-and-when-to-use-it/

Kelly M. (2003) Choosing a test automation framework. Haettu 21.3.2015 osoitteesta http://www.ibm.com/developerworks/rational/library/591.html

Kent J. (2007). Test Automation: From record/playback to frameworks. EuroSTAR 2007. Haettu 21.3.2015 osoitteesta http://www.simplytesting.com/Downloads/Kent%20-%20From%20Rec-Playback%20To%20FrameworksV1.0.pdf

Laukkanen, P. (2006). Data Driven and Keyword-Driven Test Automation Frameworks. Helsinki University of Technology, Master Thesis.

Leffingwell, D. (2007). Scaling Software Agility: Best practices for Large Enterprises. Boston: Pearson Education.

Li K. & Wu M. (2005). Effective GUI Test Automation. Alameda: SYBEX Inc.

Meyer M. (2014). Continuous Integration and Its Tools. IEEE Software, Volume 31, Issue 3. s. 14-17. IEEE

Nagle C. (2015) Test Automation Frameworks. Haettu 21.3.2015 osoitteesta http://safsdev.sourceforge.net/FRAMESDataDrivenTestAutomationFrameworks.htm

Owaise, A. (2013). Test Automation Framework. Haettu 10.5.2015 osoitteesta http://blogs.hexaware.com/are-you-being-served/test-automation-framework/

Pajunen, T. (2011). Model-Based Testing with a General Purpose Keyword-Driven Test Automation Framework. IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011, s. 242-251. Berlin: IEEE.

Peltola J. & Sierla S. & Vyatkin V. (2014). Adapting Keyword Driven Test Automation Framework to IEC 61131-3 Industrial Control Applications Using PLCopen XML. Emerging Technology and Factory Automation (ETFA), 2014. s. 1-8. Barcelona: IEEE.

ThoughtWorks (2013). Guide Test Automation. Haettu 9.5.2015 osoitteesta http://www.thoughtworks.com/insights/blog/guide-test-automation

Verma, M., Singh, A., Verma, K. & Rao, S. (2010). Best Activities in Software Test Automation. ISCET, 2010. Haettu 21.3.2015 osoitteesta http://www.rimtengg.com/iscet/proceedings/pdfs/se/77.pdf