Jatkuvan testauksen automatisointi: menetelmien tarkastelu

Turun yliopisto
Tietotekniikan laitos
LuK-tutkielma
Tietojenkäsittelytieteet
Maaliskuu 2025
Verneri Sirva

TURUN YLIOPISTO
Tietotekniikan laitos

Verneri Sirva: Jatkuvan testauksen automatisointi: menetelmien tarkastelu

LuK-tutkielma, 33 s.
Tietojenkäsittelytieteet
Maaliskuu 2025

Jatkuva testaus on olennainen osa modernia ohjelmistokehitystä, sillä se mahdollistaa virheiden havaitsemisen ja korjaamisen tehokkaasti. Automatisoimalla ohjelmistotestaus voidaan saavuttaa merkittäviä hyötyjä. Testejä voidaan ajaa useasti, manuaalisen testauksen aiheuttamia pysähdyksiä ohjelmistokehityksessä ei tule, sekä testikattavuus ja mutaatioiden löytäminen saattaa olla automaattisissa testeissä parempaa manuaaliseen testaukseen verrattuna.
Tutkielmassa tarkastellaan seuraavia ohjelmistotestauksen osa-alueita: testaustasot, testaustyypit ja testausmenetelmät. Näiden soveltuvuutta jatkuvaan automaattiseen testaukseen arvioidaan kattavuuden, kustannustehokkuuden ja virheiden löytämisen kyvykkyyden näkökulmasta. Tutkielman havainnot osoittavat, että automaattinen testaus parantaa testausprosessin tehokkuutta ja laatua, mutta vaatii merkittäviä alkuinvestointeja. Automaattisen jatkuvan testauksen yhdistäminen organisaation tarpeisiin tarjoaa mahdollisuuden optimoida ohjelmistokehitysprosessia ja varmistaa ohjelmistojen laatu koko ohjelmiston elinkaaren ajan. On kuitenkin huomioitava, että automaattinen testaus saattaa vaatia ohelle manuaalista testausta. Esimerkiksi turvallisuustestauksessa tarvitaan automaattisen testauksen ohella ihmisen luovuuteen ja oppimiskykyyn perustuvaa manuaalista testausta.

Asiasanat: jatkuva testaus, automaattinen ohjelmistotestaus, ohjelmistotestaus, ohjelmistokehitys, jatkuvat menetelmät

Sisällys

1 Johdanto **1**

2 Ohjelmistotestauksen tausta **4**

2.1 Keskeiset termit 4

2.2 Kustannukset ja jatkuva testaus 7

3 Ohjelmistotestauksen menetelmät **10**

3.1 Testaustasot 11

3.1.1 Yksikkötestaus 11

3.1.2 Integraatiotestaus 14

3.1.3 Hyväksyntätestaus 16

3.2 Testaustyypit 19

3.2.1 Toiminnallinen testaus 19

3.2.2 Ei-toiminnallinen testaus 21

3.3 Regressiotestaus 27

3.4 Analyysi 29

4 Yhteenveto **32**

Lähdeluettelo **34**

1 Johdanto

Ohjelmistot ovat alttiita virheille, ja tuotannossa ilmenevät virheet voivat johtaa merkittäviin kustannuksiin. Ohjelmistotestaus auttaa tunnistamaan virheet jo ohjelmiston kehitysvaiheessa, ja on siksi keskeinen osa ohjelmistokehityksen elinkaarta (engl. Software Development Life Cycle) ja laatuprosessia (engl. Quality Process) [1], [2]. Tässä tutkielmassa tarkastellaan automaattista jatkuvaa testausta (engl. Automated Continuous Testing), jossa ohjelmistoa testataan koko elinkaaren ajan, tavoitteena löytää virheet mahdollisimman varhaisessa vaiheessa ja välttää testauksen aiheuttamia pysähdyksiä ohjelmistokehityksessä [3]. Tämä tukee organisaatioiden pyrkimyksiä nopeuttaa ohjelmistokehitysprosessia täyttäen silti tiukat vaatimukset laadusta, tietoturvasta ja kustannuksista [4].

Ohjelmistotestaus voidaan toteuttaa automaattisesti tai manuaalisesti. Manuaalinen testaus on usein hidasta, kallista ja laajoissa ohjelmistoissa jopa mahdotonta toteuttaa [5]. Manuaalisen testauksen osuus halutaan yleisesti minimoida ja hyödyntää ainoastaan, kun automatisointi ei ole mahdollista tai testi suoritetaan vain muutaman kerran. Tästä syystä tutkielma on rajattu tarkastelemaan automaattista jatkuvaa testausta.

Jatkuva testaus on metodologiana melko laajasti tutkittu ja yleisesti sovellettu käytännön ohjelmistokehitysprojekteissa [6]. Akateeminen kirjallisuus tuntee useita jatkuvan testauksen menetelmiä. Ohjelmistoprojektiin tulee siis valita yhdistelmä

eri menetelmiä ja tehdä valintoja myös menetelmien sisällä[1]. Menetelmistä muodostetaan ohjelmistokehitykseen ja muihin prosesseihin sopiva kokonaisuus ja integroidaan jatkuva testaus tiiviiksi ja iteratiiviseksi osaksi ohjelmistokehitystä [8]. Hyvin toteutetussa jatkuvassa testauksessa, testejä tehdään koko ohjelmistokehityksen ja ohjelmiston elinkaaren ajan. Huolimatta siitä, että automaattisen jatkuvan testauksen käyttö ohjelmistokehityksessä on yleistynyt, automatisoidun jatkuvan testauksen ottaminen osaksi ohjelmistokehitysprosessia vielä nykyistä laajemmin on suositeltua [9]. Jatkuvan testauksen mahdollisten merkittävien hyötyjen ja laajan mielenkiinnon vuoksi on hyödyllistä tehdä tutkielma automaattiseen jatkuvaan testaukseen soveltuvista menetelmistä. Tässä tutkielmassa pyritään vastaamaan seuraaviin tutkimuskysymyksiin:

TK1: Mitä jatkuvan testauksen menetelmiä kirjallisuus tuntee?

TK2: Miten jatkuvan testauksen menetelmien soveltuvuutta automaattiseen jatkuvaan testaukseen voidaan mitata?

TK3: Mitä haasteita tai etuja valituilla menetelmillä on näillä mittareilla?

Tutkielmassa hyödynnetään akateemista kirjallisuutta. Akateemisen kirjallisuuden haussa on hyödynnetty Google Scholarin, IEEE Xploren ja Web of Sciencen hakutoimintoja. Muitakin hakualustoja on testattu, mutta valituista hakualustoista saatiin yleisesti kattavimmat ja laadukkaimmat hakutulokset.

Jatkuvan testauksen menetelmien identifioimiseksi, on käytetty hakusanoja *continuous testing*, *continuous testing methodology* ja *continuous testing method*. Jatkuvan testauksen menetelmien listauksen jälkeen on haettu kyseisten menetelmien englanninkielisillä termeillä lisää vastaavaa kirjallisuutta sekä lisätty sana *automated* ennen menetelmän nimeä. Lisäksi tavallista Google-hakua on käytetty testauskehysten löytämiseksi.

[1]Esimerkiksi yksikkötestauksessa on kaksi toisistaan eroavaa koulukuntaa: klassinen ja Lontoon koulukunta [7]

Tämä tutkielma rakentuu seuraavasti. Luvussa 2 käydään läpi jatkuvan testauksen keskeistä taustaa, kuten määritellään keskeiset termit, jatkuvan testauksen kustannushyödyt ja jatkuvan testauksen liittymistä muihin jatkuviin metodologioihin. Luvussa 3 käydään läpi kirjallisuudesta löytyvät jatkuvan testauksen menetelmät sekä arvioidaan menetelmien soveltuvuus automaattiseen jatkuvaan testaukseen. Luku 4 on tutkielman yhteenveto.

2 Ohjelmistotestauksen tausta

Ohjelmistotestaus on keskeinen osa ohjelmiston laatuprosessia. Ohjelmistotestaus voidaan määritellä prosessiksi, jossa seurataan ohjelmiston suorituskykyä ja toimintaa tarkoituksena varmistaa sen toiminta halutulla tavalla sekä tunnistaa mahdolliset virheet [10]. Ohjelmistotestaus on melko laaja termi ja kattaa useita erilaisia testausmenetelmiä, jotka toimivat konkreettisina työkaluina testausprosessin toteuttamiseen. Testausmenetelmät määrittelevät käytännön testien toteutuksen, jotka voidaan suorittaa manuaalisesti tai automaattisesti. Testausmenetelmien valinta vaikuttaa siihen, kuinka tehokkaasti testaus pystyy varmistamaan ohjelmiston toimivuuden, suorituskyvyn ja käyttäjäystävällisyyden.

2.1 Keskeiset termit

Ohjelmistotestaus voidaan jakaa eri osa-alueisiin. Kattokäsitteenä toimii testausmetodologia, joka tässä tapauksessa on jatkuva testaus. Tämän alla ohjelmistotestaus jaetaan neljään eri testaustasoon: yksikkö-, integraatio-, järjestelmä- ja hyväksymistasoihin [11]. Ohjelmistotestaus voidaan myös jakaa testaustyyppeihin. Testaustyyppien alla on toiminnallinen testaus ja ei-toiminnallinen testaus. Toiminnallinen testaus varmistaa ohjelmiston toiminnan suunnitellulla tavalla. Ei-toiminnallinen testaus taas arvioi esimerkiksi ohjelmiston suorituskykyä, käytettävyyttä ja turvallisuutta [12]. Kolmantena ohjelmistotestaus voidaan jakaa yksittäisiin testausmenetelmiin. Tässä tutkielmassa testausmenetelmien alle on asetettu menetelmät, jotka

eivät suoraan sovi edellä mainittuihin kategorioihin. Näiden pohjalta seuraavaksi esiteltävät kategoriat ja termit ovat pohjana automaattisen jatkuvan testauksen tarkasteluun tässä tutkielmassa.

Testausmetodologia on laajempi lähestymistapa testaukseen, joka sisältää useita testausmenetelmiä ja -käytäntöjä. Metodologia pyrkii vastaamaan kysymykseen siitä, miten testaus integroituu osaksi ohjelmiston kehitysprosessia. Esimerkkejä tällaisista metodologioista ovat jatkuvan testauksen metodologia ja ketterä ohjelmistotestaus.

Testaustasot Määrittävät mihin ohjelmistotestauksen osa-alueeseen testaus kohdistuu ja kuvaavat miten laajaa ohjelmistokokonaisuutta testataan. Testaustasot jaetaan neljään päätasoon [11]:

- **Yksikkötestaus** testaa yksittäisiä ohjelmistokomponentteja tai funktioita tyypillisesti eristettynä ja varmistaa niiden oikean toiminnan.
- **Integraatiotestaus** tarkistaa ohjelmistokomponenttien välisen yhteensopivuuden ja toiminnan, esimerkiksi ohjelmistorajapintojen kautta.
- **Järjestelmätestaus** keskittyy ohjelmiston toimivuuden testaamiseen kokonaisuutena käyttäjän näkökulmasta.
- **Hyväksymistestaus** varmistaa, että ohjelmisto täyttää sille asetetut käyttäjä- ja liiketoimintavaatimukset.

Testaustyyppi Testauksen osa-alue, joka keskittyy ohjelmiston toiminnalliseen tai ei-toiminnalliseen ominaisuuteen. Määrittelee mitä ohjelmiston osa-alueita ja ominaisuuksia testataan. Esimerkkejä ovat suorituskykytestaus, käytettävyystestaus ja turvallisuustestaus.

Testausmenetelmä viittaa yksittäiseen testaustapaan, jota käytetään testausprosessin konkreettiseen toteuttamiseen. Testausmenetelmä kuvaa mitä konkreet-

tisia tapoja käytetään testauksen suorittamiseen.

Edellä mainitut keskeiset termit muodostavat toisiinsa nivoutuvan kokonaisuuden, joka ohjaa ohjelmistotestauksen toteutusta. Testausmetodologiat ovat korkean tason strategioita, jotka määrittelevät laajan lähestymistavan testaukseen koko ohjelmistokehitysprosessin aikana. Kyseessä on siis kattokehys, jonka alla sovelletaan erilaisia testaustasoja, -menetelmiä ja -tyyppejä. Testaustasot jakavat ohjelmistotestauksen neljään hierarkkiseen osa-alueeseen: yksikkö-, integraatio-, järjestelmä- ja hyväksymistasoon. Testaustasot määrittelevät mihin ohjelmiston osa-alueeseen testaus keskittyy, kuten yksittäisiin komponentteihin, niiden väliseen yhteistoimintaan tai ohjelmiston kokonaisuuteen käyttäjän näkökulmasta.

Testausmenetelmät toimivat käytännön työkaluina ja menetelminä, joita sovelletaan jokaisella testaustasolla. Ne määrittävät, miten testaus toteutetaan. Testaus voidaan suorittaa esimerkiksi manuaalisesti, automaatiota hyödyntäen tai tiettyyn erityistarkoitukseen, kuten suorituskyvyn varmistamiseen. Testaustyypit puolestaan täydentävät tätä kokonaisuutta keskittymällä siihen, mitä ominaisuuksia tai piirteitä ohjelmistosta testataan. Testaustyypit määrittelevät testauksen tarkemmat tavoitteet ja arviointikriteerit testaukselle. Ne voivat kohdistua esimerkiksi ohjelmiston toimintaan eli toiminnallisiin testeihin, suorituskykyyn, turvallisuuteen tai käytettävyyteen eli ei-toiminnallisiin testeihin. Näin ollen voidaan sanoa, että testausmetodologiat ohjaavat kokonaisuutta, testaustasot jäsentävät testauksen hierarkkisesti, testausmenetelmät konkretisoivat testauksen toteutustavat ja testaustyypit täydentävät prosessia keskittymällä ohjelmiston ominaisuuksien laadunvarmistukseen. Yhdessä nämä muodostavat yhtenäisen järjestelmän, joka varmistaa ohjelmiston kattavan ja tehokkaan testauksen. Tutkielma on rajattu automatisoituun jatkuvaan testaukseen, joka on tärkeä osa nykyistä ja tulevaisuuden ohjelmistokehitystä. Tutkielma ei siis kata kaikkia testaustasoja, -menetelmiä ja -tyyppejä.

2.2 Kustannukset ja jatkuva testaus

Kustannusten minimoimiseksi ohjelmiston virheet halutaan löytää mahdollisimman aikaisessa vaiheessa ohjelmistokehitysprosessia. Lisäksi tuotannossa olevasta ohjelmistosta löytyvät virheet voivat heikentää luottamusta ohjelmiston laatuun ja aiheuttaa myös loppukäyttäjälle taloudellisia tappiota. Huolimatta siitä, että prosessi kuulostaa melko yksinkertaiselta, käytännön toteutus on usein haastavaa. Ohjelmistot ovat yleisesti laajoja, niillä on useita riippuvuuksia ja ohjelmistossa voidaan käyttää useita eri kieliä, kirjastoja ja tietokantoja. Tästä syystä ohjelmiston testaus voi viedä jopa 50 % ohjelmiston kehitykseen käytettävistä resursseista [13]. Ohjelmiston kriittisen laadunvarmistuksen ohella ohjelmistotestaus on siis taloudellisesti merkittävä kustannus. Ohjelmistokehityksen taloudellinen kannattavuus voi riippua ohjelmistotestauksen onnistumisesta. Tästä syystä optimaalisten testausmetodologioiden valinta on tärkeää ohjelmistokehityksessä.

Jatkuvan testauksen metodologioiden hyödyntäminen ohjelmistotestauksessa on yleistynyt. Jatkuva testaus terminä ja mallina esiteltiin ensi kertaa vuonna 2003 [3]. Ohjelmistotestaus ja niin sanottu testiohjautuva ohjelmistokehittäminen (engl. test-driven development) eli ohjelmiston testaus koko ohjelmiston kehitysprosessin läpi on ollut yleistä jo ennen kuin jatkuva testaus on esitelty kirjallisuudessa. Ei-jatkuvan testauksen merkittävänä ongelmana on ollut kehityksen pysähtyminen testivaiheisiin. Saff ja Ernst [3] osoittivat, että jatkuvalla testauksella pystyttiin säästämään ohjelmistokehitykseen käytettyä kokonaisaikaa 10–15 % ja muihin testausmenetelmiin verrattuna jopa 92–98 %. Merkittävä ajallinen hukka ei-jatkuvassa testauksessa on muutoksen jälkeisten regressiovirheiden etsiminen ja korjaaminen. Regressiovirhe tarkoittaa sitä, että ohjelmiston aiemmin toiminut ominaisuus lakkaa toimimasta. Tästä syystä regressiotestaus on tärkeä osa jatkuvaa testausta. Lisäksi mikäli virhe löydetään vasta tuotantovaiheessa, on ohjelmistokehittäjä saattanut jo unohtaa kontekstin ja syyn tehdyille muutoksille. Virheellisen logiikan päälle on saatettu ra-

kentaa uusia toiminallisuuksia. Tästä syystä Saff ja Ernst [3] suosittelevat jatkuvan testauksen menetelmää, jolloin testit ajetaan säännöllisesti koko ohjelmistokehityksen ajan. Tällöin virheet korjataan nopeasti niiden luomisen jälkeen ja riski uuden toiminnallisuuden rakentamiseen virheellisen päälle on pienempi.

Automaattinen ohjelmistotestaus automatisoi muuten manuaalisen testauksen hyödyntämällä työkaluja ja skriptejä testien suorittamiseen manuaalisen ihmistyön sijaan [14]. Automaattinen jatkuva testaus voidaan toteuttaa hyödyntämällä tosiaikaista integraatiota (engl. real-time integration) kehitysympäristössä, jossa asynkronisesti ajetut testit ajetaan viimeisimmällä ohjelmistolla. Asynkroninen testaus on ohjelmistotestauksen muoto, jossa testit suoritetaan testattavasta ohjelmistosta erillisessä prosessissa kuin itse testattava ohjelmisto [3]. Tällöin testaus ei estä sovelluksen suoritusta, vaan tämä voidaan suorittaa samanaikaisesti ohjelmiston kanssa. Hyödyntäen esiteltyä asynkronisen testauksen menetelmää voidaan saavuttaa laskennallinen tehokkuus[1] ja samalla synkronisen testauksen turvallisuus. Tällöin vältytään manuaalisilta vaiheilta ja ohjelmiston laatua pystytään jatkuvasti seuraamaan tehokkaasti. Mikäli jotain ei läpäise testiä, ohjelmistokehittäjä saa ilmoituksen ja virhe pystytään korjaamaan nopeasti ja varhaisessa vaiheessa.

Automaattisen testauksen hyötyjä manuaaliseen testaukseen verrattuna ovat esimerkiksi suuri tehokkuus, laajempi testikattavuus, inhimillisten virheiden välttäminen ja resurssien säästö [14]. Manuaalisesti testejä suorittaessa aikaa kuluu tyypillisesti merkittävästi enemmän, inhimillisiä virheitä voi tapahtua ja kymmenien tai jopa satojen testien suorittaminen manuaalisesti ei ole realistisesti mahdollista. Automaattisen testauksen haasteena on puolestaan testien luomiseen ja ylläpitoon kuluva aika; testien ohjelmointi vie yleensä enemmän aikaa kuin niiden manuaalinen suorittaminen ja ne saattavat vaatia säännöllistä ylläpitoa ohjelmiston toiminnan muuttuessa. Automaattiseen testaukseen on tosin olemassa laadukkaita työkaluja,

[1]Testien ajaminen CPU:lla synkronisesti vie laskentatehoa, jonka itse ohjelmistokehitys ja ohjelmiston ajaminen vaativat.

kuten Selenium, JUnit ja Robot Framework, jotka helpottavat testien luomista ja niiden ylläpitoa. Mikäli ohjelmisto on laaja ja sitä kehitetään jatkuvasti, on todennäköistä, että automaattinen testaus on manuaalista testausta tehokkaampaa.

Automaattinen jatkuva testaus on osana suurta muiden jatkuvien metodologioiden kokonaisuutta. Jatkuvalla metodologialla tarkoitetaan ohjelmiston kehitysprosessin ja toimituksen parantamista ja tehostamista toisiinsa liittyvillä prosesseilla, joilla varmistetaan ohjelmiston laatu ja tarpeisiin vastaavuus reaaliajassa. Näihin kuuluvat esimerkiksi jatkuva integraatio (engl. Continuous Integration, CI) ja toimitus/käyttöönotto (engl. Continuous Delivery/Deployment, CD) [15]. Automaattinen jatkuva testaus muodostuu osaksi niin sanottua CI/CD-putkea (engl. pipeline). Ohjelmiston muutokset integroidaan jatkuvasti päähaaraan ja ohjelmisto toimitetaan automaattisesti tuotantoon, mikä varmistaa ohjelmiston laadun [6]. Jatkuvaa testausta tapahtuu kaikissa putken vaiheissa, aina yksikkötesteistä tuotannossa tapahtuviin testeihin. Tällöin ohjelmistossa tehdyt muutokset saadaan nopeasti tuotantoon, mikä varmistaa näiden laatu kehityksen alusta aina tuotantovaiheeseen saakka. Muita jatkuvia menetelmiä, jotka liittyvät CI/CD-putkeen ja jatkuvaan testaukseen, ovat jatkuva seuranta (engl. Continuous monitoring) ja jatkuva parantaminen (engl. Continuous improvement) [16], [17]. Jatkuva seuranta auttaa havaitsemaan ongelmia tuotantoympäristössä ja jatkuva parantaminen mahdollistaa ohjelmiston jatkuvan kehittämisen. Ilman jatkuvaa testausta muiden jatkuvien metodologioiden käyttö on siis haastavaa, sillä ohjelmiston tulee olla hyvin testattu ennen kuin se päätyy tuotantoon riskien minimoimiseksi. Tällainen CI/CD-putki voidaan luoda esimerkiksi hyödyntäen Jenkinsiä.

3 Ohjelmistotestauksen menetelmät

Testausmenetelmien soveltuvuutta automaattiseen jatkuvaan testaukseen voidaan mitata eri tavoin. Yksittäiselle organisaatiolle testien automatisointipäätös voi riippua kymmenistä eri asioista [18]. Tässä tutkielmassa käytettäviä mittareita ovat automaattisten testien kattavuus (engl. test/code coverage), automaattisten testien kyky löytää virheitä (engl. mutation testing), testien automatisoinnin helppous eli työkalujen ja testikirjastojen olemassaolo, sekä testien automatisoinnin kustannustehokkuus (engl. cost-efficiency). Mittareita tarkastellaan kolmen pääkategorian kautta: *testaustasot*, kuten yksikkö-, integraatio- ja järjestelmätestaus. Testaustasot määrittävät mitä ohjelmiston osia testit kattavat ohjelmistosta. *Testaustyypit* jotka jaetaan toiminnallisiin ja ei-toiminnallisiin testeihin. Testaustyypit varmistavat ohjelmiston toiminnot ja suorituskyvyn. Testausmenetelmän alle tässä tutkielmassa luetaan regressiotestaus. Regressiotestauksella varmistetaan, että aiemmat toiminnallisuudet toimivat oletetulla tavalla. Nämä elementit yhdessä tarjoavat kattavan kuvan testausmenetelmien soveltuvuudesta automaattiseen jatkuvaan testaukseen.

3.1 Testaustasot

3.1.1 Yksikkötestaus

Yksikkötestaus (engl. unit testing) on testausmenetelmä, jossa ohjelmisto jaetaan joukkoon yksiköitä. Yksikkö muodostuu tyypillisesti joukosta funktioita. Jokaista yksikköä varten luodaan omat yksikkötestit ja testit ajetaan eristetysti muusta ohjelmistosta [19]. Yksikkötestauksen kuuluu kaksi merkittävää koulukuntaa: Lontoon koulukunta ja klassinen koulukunta [7]. Koulukunnat eroavat toisistaan merkittävästi ja tästä syystä on mielekästä tehdä valinta koulukuntien välillä.

Yksikkötestauksen Lontoon koulukunta korostaa yksiköiden testauksen eristämistä muusta lähdekoodista käyttämällä niin sanottuja valefunktioita (engl. mocks) ja tynkiä (engl. stubs). Valefunktiot ja tyngät ovat ohjelmistokomponentteja, jotka simuloivat toisten komponenttien toimintaa testin aikana, integraatioita ei siis testata suoraan. Näin saavutetaan eristetty testausympäristö, jossa voidaan varmistaa, että testitulokset heijastavat tarkasti kyseisen yksikön toimintaa ilman ulkoisten riippuvuuksien suoraa vaikutusta. Tämä lähestymistapa mahdollistaa virheiden paikantamisen tarkemmin, koska testit eivät epäonnistu ulkoisten riippuvuuksien vuoksi, vaan heijastavat yksinomaan testattavan yksikön toimintaa. Lisäksi valefunktioiden ja tynkien käyttö voi nopeuttaa testien suorittamista; ulkoisia resursseja, kuten tietokantayhteyksiä ei tarvitse hyödyntää. [7]

Lontoon koulukunnan etuna on tiukka yksikön eristys, joka tekee testituloksista luotettavia kyseisen yksikön osalta. Tämä helpottaa virheiden paikantamista. Tyypillisesti testi siis epäonnistuu paikassa, jossa sen virhe tapahtuu eikä sivuvaikutuksena virheestä muualla. Varjopuolena on, että valefunktiot ja tyngät johtavat monimutkaisiin testitietokantoihin, joiden ylläpito voi vaatia merkittäviä resursseja. Lisäksi on mahdollista, että simuloidut riippuvuudet poikkeavat todellisista riippuvuuksista, mikä voi johtaa virheellisiin johtopäätöksiin testauksen tuloksista.

Yksikkötestauksen klassinen koulukunta puolestaan painottaa aitojen riippuvuuksien käyttöä yksikkötestauksessa, ja pyrkii minimoimaan valefunktioiden ja tynkien käytön. Tämä lähestymistapa korostaa todellisen ympäristön käyttöä testauksessa. Klassinen koulukunta näkee yksikköjen riippuvuudet olennaisena osana niiden toimintaa, ja näin ollen testaa yksiköitä yhdessä niiden todellisten riippuvuuksien kanssa. Tämä lähestymistapa mahdollistaa myös yksiköiden välisen integraation ja yhteistoiminnan testaamisen, mikä voi auttaa löytämään virheitä, jotka johtuvat yksiköiden välisestä vuorovaikutuksesta. [7]

Klassisen koulukunnan etuna on se, että yksikkötestaus on lähempänä todellista käyttötilannetta; myös yksiköiden välinen integraatio testataan. Tällöin myös testit mahdollisesti löytävät yksiköiden väliset sivuvaikutuksia ja riippuvuuksien aiheuttamat virheet. Selkeänä heikkoutena on kuitenkin virheiden paikantamisen haastavuus; testi voi epäonnistua väärässä yksikössä ja vian paikantaminen voi viedä aikaa. Optimaalisesti yksikkötestit ajetaan siis automaattisesti esimerkiksi aina kokoamisen yhteydessä.

Klassisen koulukunnan yksikkötestaus on käytännön toteutuksen kannalta usein haasteellinen. Korkeampi testien määrä ei välttämättä tuo toivottua hyötyä eli kustannustehokkuus on heikkoa. Yksikkötestauksen skaalaaminen suurissa ja monimutkaisissa järjestelmissä on merkittävä haaste. Testien määrän kasvaessa niiden hallinta ja ajaminen voi muuttua hankalaksi ja epätehokkaaksi. Testausstrategian tulee siis olla onnistunut ja noudattaa moderneja yksikkötestauksen strategioita. Näiden haasteiden ratkaiseminen on keskeistä yksikkötestauksen onnistumiselle ja sen hyötyjen maksimoinnille ohjelmistokehitysprosessissa. Lisäksi integraatiotestauksessa testataan erikseen integraatioiden toimivuutta; tällöin integraatioiden hyödyntäminen klassisen koulukunnan yksikkötestauksessa voi olla turhaa.

Yksikkötestauksessa pyritään eristämään funktiojoukko muusta logiikasta. Tästä syystä on hyvin vaikeaa manuaalisesti suorittaa yksikkötestejä; yksikkötestit aje-

taan käytännössä aina automaattisesti. Yksikkötestauksen tutkimus keskittyy lähinnä testien luomisen automaation, jonka ympärillä on melko laajasti tutkimusta. Esimerkiksi Microsoftilla testien luomisen automatisoinnin jälkeen ohjelmiston loppukäyttäjät ovat raportoineet vähemmän virheitä [20].

Yksikkötestauksen testien luomisen automaatiosta löytyy laaja kirjallisuus. Yksikkötestauksen automatisaatiolla voidaan yleisesti saavuttaa laajempi testikattavuus kuin manuaalisesti luoduilla testeillä [21], [22]. Tuloksia löytyy myös manuaalisesti ja automaattisesti luotujen yksikkötestien komplementaarisesta luonteesta [23]. Manuaalisesti ja automaattisesti luodut testit saattavat siis löytää erilaisia virheitä, eivätkä siis täysin substituutteja toisilleen. Samassa tutkimuksessa tosin todettiin, että automaattisesti luodut testit tuovat paremman testikattavuuden kuin manuaalisesti luotujen testien. Automaattisesti luodut testit myös löytävät yleisesti paremmin mutaatiot kuin manuaaliset testit [21], [22]. Kracht, Petrovic ja Walcott-Justice [24] tekemässä kokeessa ohjelmoijan kirjoittamat testit kattoivat 31,5 % ja automaattisten työkalujen 31,8 % koodista. Automaattiset testit siis marginaalisesti paransivat testikattavuutta.

Myös mutaatioiden osalta manuaalinen ja automaattinen testien luonti saattavat olla komplementteja, mutta silti automaattisesti luodut testit löytävät todennäköisemmin mutaatiot [23]. Kracht, Petrovic ja Walcott-Justice [24] tekemässä kokeessa taas huomattiin, että ohjelmoijien kirjoittamat testit tosin olivat parempia mutaatioiden löytämisessä; ohjelmoijien kirjoittamat testit löysivät 42,1 % mutaatioista kuin automaattisesti kirjoitetut testit 39,8 %.

Yksikkötestauksen automatisaatio on käytännössä aina järkevämpää kuin niiden manuaalinen kirjoittaminen [24]. Yksikkötestauksen kohdalla automatisaatio on siis lähes itsestäänselvää; olisi hankalaa jo yksistään testata kaikki ohjelman funktiot. Yksikkötestien luomisen automatisaatioon löytyy useita työkaluja, esimerkiksi HyperTest. Yleinen tunnettu testauskehys automaattiselle yksikkötestaukselle

on JUnit. Yksikkötestit ovat tyypillisesti luonteeltaan sellaisia, että niitä luodaan jatkuvasti ja ne ajetaan usein. Tästä syystä sopivimman alustan valitseminen on tärkeää ja siihen kannattaa investoida resursseja.

3.1.2 Integraatiotestaus

Integraatiotestaus (engl. integration testing) on vaihe, jossa testataan eri ohjelmistomoduulien välistä vuorovaikutusta sekä varmistetaan ohjelmiston toiminta yhtenä kokonaisuutena yksikköjen sijaan, integraatiot huomioiden. Integraatiotestauksessa siis tarkastellaan sitä, miten komponentit ja integraatiot toimivat yhdessä (vrt. yksikkötestauksen klassinen koulukunta) [25]. Integraatiotestaus on välttämätöntä nykyisten monimutkaisten ohjelmistojen kehityksessä, sillä integraatioiden toiminta tulee varmistaa ennen ohjelmiston käyttöönottoa. Integraatiotestauksen onnistuminen riippuu pitkälti siitä, kuinka hyvin testaus pystyy vastaamaan tuotantokäyttöä.

Nykyiset laajat ohjelmistot ovat usein riippuvaisia monista eri integraatiosta. Ohjelmistolla on tyypillisesti yhteyksiä yhteen tai useampaan tietokantaan ja lisäksi esimerkiksi asiakkaan ERP-järjestelmään. Vaikka yksikkötestauksen Lontoon koulukunnassa voidaan varmistaa yksittäisen komponentin toiminta esimerkiksi valefunktioiden avulla, tämä ei takaa sitä, että komponentti toimii oikein todellisessa ympäristössä, jossa se on yhteydessä muihin järjestelmiin ja tietokantoihin. Optimaalisesti integraatiotestaus tehdään komponenteilla, jotka on laadukkaasti ja hyväksytysti yksikkötestattu. Usein integraatiota tehdään kuitenkin ei-inkrementaalisesti, eli kaikki moduulit yhdistetään ja testataan kerralla. Tämä voi johtaa merkittäviin ongelmiin, sillä virheiden paikantaminen voi olla vaikeaa, kun useita eri komponentteja testataan samanaikaisesti. Tämä voi johtaa loputtomiin ongelmien korjauskierroksiin, joissa yksi korjattu virhe saattaa paljastaa uusia ongelmia. Tämän vuoksi integraatiotestausta suositellaan tehtäväksi inkrementaalisesti; yksi integraatio kerrallaan. [25]

Merkittävimmät haasteet integraatiotestauksessa liittyvät testauksen laajuuteen ja monimutkaisuuteen. Inkrementaalisen testauksen toteuttaminen saattaa vaatia paljon resursseja, mutta kääntöpuolena voidaan välttää merkittäviä virheitä tuotannossa. Integraatioiden testaus voi myös olla haastavaa ennen ohjelmiston käyttöä tuotannossa; tämä vaatii testiympäristön kehittyneisyyttä, mikä saattaa olla haastavaa erityisesti monimutkaisissa järjestelmissä, joissa on paljon integraatioita.

Integraatiotestauksessa on useita eri metodologioita, joista yleisimpiä ovat ylhäältä alas (engl. top-down) ja alhaalta ylös (engl. bottom-up) integraatiot. Ylhäältä alas integraatio on inkrementaalinen menetelmä, jossa moduulit integroidaan ylhäältä alaspäin kontrollihierarkiassa. Tämä tarkoittaa, että ensin integroidaan kontrollimoduuli, jonka alle tulevat muut moduulit integroidaan joko syvyys- (engl. depth-first) tai leveyssuuntaisesti (engl. breadth-first). Syvyyssuuntaisessa metodissa integroidaan ensin kontrollimoduulin alla oleva moduuli, jonka jälkeen siirrytään tämän alaisiin moduuleihin. Leveyssuuntaisessa metodissa taas integroidaan ensin kaikki saman tason moduulit ennen siirtymistä alemmille tasoille. [25]

Alhaalta ylös integraatiossa taas aloitetaan hierarkian alapäästä ja integroidaan ensin niin sanotut atomistiset moduulit. Atomistiset moduulit ovat pieniä ja yksinkertaisia toiminnallisia yksiköitä. Tämän etuna verrattuna ylhäältä alas integraatioon on, että alemmat moduulit ovat jo täysin toimivia; tällöin testaus on helpompaa. Alhaisen tason komponentit yhdistetään ryhmiksi, jotka suorittavat tietyn ohjelmiston alatoiminnon. Näiden ryhmien testaamiseksi luodaan ajuri, joka hallitsee testitapausten syötteitä ja tulosteita. Testauksen jälkeen ajurit poistetaan ja ryhmät yhdistetään ohjelman rakenteessa ylöspäin edeten inkrementaalisesti. [26]

Automaattinen integraatiotestaus on mahdollista tehdä kattavasti monissa eri käyttökohteissa [27], [28]. Testien kattavuus on yleisesti parempi automaattisessa integraatiotestauksessa, sillä testaaminen on mahdollista tehdä laajemmin ja ajaa enemmän testitapauksia. Automaattinen testaus myös löytää enemmän virheitä

ohjelmistosta kuin manuaalinen testaus. Automaattinen integraatiotestaus saattaa vaatia melko paljon resursseja ja erilaisten testistrategioiden ja -mallien ymmärtämistä. Tästä syystä laadukas automaattinen testaus vaatii osaamista ja resursseja. Hyötynä on laajempi kattavuus ja korkeampi todennäköisyys löytää virheet verrattuna manuaaliseen testaukseen [27], [28].

Integraatiotestauksen automaattisten testien luomiseen voidaan käyttää esimerkiksi JUnitia tai Hypertestiä. Automaattinen integraatiotestaus on todennäköisesti kustannustehokasta. Xu, Xu, Tu et al. [29] osoitti kokeessa, että hyödyntäen loogisia sopimuksia (engl. logical contract) automaattisessa integraatiotestauksessa voidaan mutaatiot löytää kustannustehokkaasti ohjelmistosta.

3.1.3 Hyväksyntätestaus

Hyväksyntätestaus (engl. user acceptance testing) on ohjelmistokehityksen viimeinen testausvaihe ennen tuotantoa, jossa varmistetaan, että ohjelmiston testitulokset täyttävät hyväksyntäkriteerit operatiiviseen käyttöön. Hyväksyntätestaus eroaa monista muista testausmenetelmistä keskittymällä ohjelmiston kokonaisvaltaisten toiminnallisuuksien testaamiseen loppukäyttäjän näkökulmasta [30]. Testit keskittyvät suurimpiin toiminnallisuuksiin, eli tyypillisesti käyttöliittymään ja virheiden käsittelykykyyn. Ohjelmiston ja sen dokumentaation tulee täyttää käyttäjän ja kehittäjän yhdessä sopimat hyväksymiskriteerit [31]. Lisäksi hyväksyntätestauksen tarkoitus on toimia viimeisenä luottamusta herättävänä tarkastuksena ohjelmiston laadusta ja käyttökelpoisuudesta.

Loppukäyttäjän sitoutuminen ja osallistuminen hyväksyntätestaukseen on kriittistä onnistumisen kannalta. Tämä voi kuitenkin olla haasteellista esimerkiksi fyysisen etäisyyden, aikaeron ja -pulan vuoksi. Loppukäyttäjä voi myös vastustaa kyseistä ohjelmistoa, jos hän kokee sen uhkaavaksi omalle työtehtävälleen[1]. Lisäksi hyväk-

[1]Ohjelmisto voi esimerkiksi automatisoida tuotantosuunnittelijan työtehtäviä. Tästä syystä tuotantosuunnittelija ei todennäköisesti ole oikea henkilö suorittamaan hyväksyntätestauksia.

symiskriteerien määrittely voi olla monimutkaista, mikä voi johtaa epäselvyyksiin ja ristiriitoihin käyttäjien ja kehittäjien välillä. Näiden haasteiden hallinta vaatii huolellista suunnittelua ja yhteistyötä kaikkien osapuolten välillä. Eri strategioiden vertailu auttaa valitsemaan sopivimman lähestymistavan projektin ja käyttäjien tarpeiden perusteella. Testausympäristö ja työkalut testauksessa suoritetaan usein ohjelmiston operatiivisella alustalla tai sitä hyvin simuloivassa ympäristössä. Tämä mahdollistaa realistisen käyttöympäristön jäljittelyn. Testaaja on ohjelmiston käyttäjä, joka suorittaa testit usein yhdessä kehittäjän kanssa, joka tarjoaa teknistä tukea ja neuvoja testitapausten valinnassa.

Käyttäytymispohjaisessa hyväksyntätestauksessa, joka tunnetaan myös nimellä skenaariopohjainen testaus, keskitytään ohjelmistojärjestelmän ulkoisen käyttäytymisen tarkasteluun käyttäjän näkökulmasta. Tämä lähestymistapa koostuu useista osa-alueista. Ensimmäisenä on skenaarioiden määrittely ja dokumentointi, jossa käyttäjäskenaariot tunnistetaan ja kirjataan vaatimusten perusteella. Lisäksi käyttäjien ja ulkoisten järjestelmien määrittelyn tulee olla tarkkaa, sillä on olennaista tunnistaa järjestelmän loppukäyttäjät sekä integraatiot muihin järjestelmiin. Testimalli formalisoidaan skenaarioiden kuvaamiseksi, ja skenaarioihin perustuen rakennetaan äärelliset automaatit. Skenaariot tulee myös vahvistaa, mikä käytännössä tarkoittaa, että testien on peilattava tarkasti käyttäjän vuorovaikutusta järjestelmän kanssa. Määrittelypohjainen testaus on yleisimmin käytetty strategia hyväksyntätestauksessa ohjelmistokehitysprojekteissa. [31]

Toiminnallinen testimatriisi voi auttaa valitsemaan vähimmäisjoukon testitapauksia, jotka kattavat järjestelmän toiminnot. Tämä matriisi mahdollistaa järjestelmällisen suunnittelun ja normaalitapausten kattavuuden, jotta saadaan kattava toiminnallinen testijoukko. Testaus alkaa normaalien testitapausten määrittämisestä määrityksen ja järjestelmätoimintojen tunnistamisen perusteella. Sitten luodaan vähimmäisjoukko testitapauksia, jotka kattavat kaikki syötteet ja tulosteet sekä jär-

jestelmätoiminnot. Monimutkaiset tapaukset pyritään purkamaan yksinkertaisemmiksi, ja yksittäisten ehtotapausten yhdistelmiä tarkastellaan mahdollisten toiminnallisten yhdistelmien löytämiseksi. Lopuksi poistetaan tarpeettomat testitapaukset, jotka katetaan muiden testien toimesta. [31]

Testaus operatiivisen profiilin perusteella on tärkeä suurten teollisten ohjelmistojärjestelmien esikäyttöönoton testauksessa. Operatiivinen profiili määritellään joukoksi operaatioita ja niiden esiintymistaajuuksia odotetussa käyttöympäristössä. Tämä profiili kehitetään analysoimalla järjestelmän käyttöä. Ensiksi määritetään asiakasprofiili, jonka jälkeen laaditaan käyttäjäprofiili. Seuraavaksi määritetään järjestelmän toimintatilaprofiili ja toiminnallinen profiili, jonka jälkeen voidaan lopulta määrittää operatiivinen profiili. Operatiivinen profiili riippuu sovelluksen käyttäjistä, ja useimmilla sovelluksilla on useita kohdistettuja käyttäjäluokkia, joilla kaikilla on oma operatiivinen profiilinsa. Hyväksyntätestauksen aikana sovelluksen luotettavuutta on tarkistettava kaikkia näitä operatiivisia profiileja vasten, jotta varmistetaan sovelluksen hyväksyttävyys kaikille käyttäjäluokille. Operatiivisen profiilin käyttö testauksen ohjaamiseen tarkoittaa, että testitapaukset valitaan operaatioiden esiintymistodennäköisyyksien mukaan: useimmin käytetyt operaatiot saavat eniten testausaikaa, kun taas harvemmin käytetyt operaatiot saavat vähemmän. Lisäksi otetaan huomioon operaation kriittisyys, joka mittaa järjestelmän epäonnistumisen vaikutuksen vakavuutta, ja mitä kriittisempi operaatio on, sitä enemmän testejä se vaatii. [31]

Hyväksyntätestauksen automatisaatio perustuu hyvään asiakkaan kuulemiseen. Tätä varten on olemassa metodeja, jotta hyväksyntävaatimukset saadaan asiakkaalta laadukkaasti. Tosin keskiverto asiakas ilman erityistä IT-osaamista voi kokea tämän haasteelliseksi ja tulos olla huono [32]. Kattavan ja toimivan automaattisen hyväksyntätestauksen luominen vaatii todennäköisesti osaavan ja motivoituneen asiakkaan, jonka työntekijöillä on IT-alan osaamista. Asiakkaan on lisäksi oltava

sitoutunut kirjoittamaan testitapauksia, mikä ei välttämättä toteudu [33]. Mikäli asiakaan ja ohjelmistokehittäjien kommunikointi on motivoitunutta ja onnistunutta, voi automatisoitu funktionaalinen testaus olla järkevää [34].

Kommunikoinnin onnistuttua itse testien tekeminen on melko helppoa käyttäen modernia testauskehystä, esimerkiksi Seleniumia tai JAutomatea. Hyväksyntätestauksen automatisaatioon testien luomisesta tulee myös olla riittävä hyöty ottaen huomioon vaihtoehtoiskustannuksen. Voi hyvinkin olla, että kommunikointiin ja testien kirjoittamiseen menee enemmän aikaa kuin testien manuaaliseen ajamiseen. Kirjallisuudesta löytyy kuitenkin osoituksia automatisoidun hyväksyntätestauksen kustannustehokkuudesta, esimerkiksi edellä mainittu JAutomate on todettu kustannustehokkaaksi [35].

3.2 Testaustyypit

3.2.1 Toiminnallinen testaus

Toiminnallisessa eli funktionaalisessa testauksessa (engl. functional testing) testataan, mitä ohjelmiston tulisi toiminnallisesti tehdä, eikä miten se kyseisen toiminnallisuuden toteuttaa. Funktionaalinen testaus yleisesti identifioidaan syöte-tulosteanalyysiksi funktionaalisessa muodossa, ja toiminnot testataan niiden toimintamääritysten mukaisesti. Funktionaalinen testaus vaatii kaikkien funktioiden identifiointia ja niiden toiminnan määrittelyä. Tähän kuuluvat siis kaikki funktiot, mukaan lukien yksinkertaisemmat funktiot. Funktionaalisen testauksen elinkaarilähestymistavassa käytetään ohjelmiston vaatimuksia ja suunnitteludokumentaatioita funktioiden identifioimiseksi. Tällöin myös tiedetään missä vaiheessa ohjelmistoa tiettyjä funktioita käytetään. Vaatimusten ja suunnitteludokumenttien laatu on tärkeää, sillä niiden pohjalta luodaan testit. On myös hyödyllistä ottaa huomioon funktionaalisen testauksen dokumentaatiovaatimukset luotaessa toimintatavat dokumentaatiota

varten. Systemaattisessa lähestymistavassa käytetään arvojen tallentamisen sijaan paljastavia testejä, joissa käytetään niin sanottuja synteettisiä funktioita. Funktionaalinen muoto sisältää erilaisia ohjelman lausekkeita ja vaatimuksia, joita testataan testidatalla. Data-analyysiä voidaan hyödyntää funktioiden tunnistamisessa ja niiden testaamisen varmistamisessa. Tämä edellyttää kuitenkin edelleen väliarvojen tarkastelua, mikä on vaikeasti automatisoitavissa. Jos keskitytään vain ohjelman kokonaisulostulon analysointiin, testausprosessi voidaan automatisoida lähes kokonaan, mutta tämä voi estää funktionaalisten virheiden löytämisen. Funktionaaliseen testaukseen tarvitaan testausdataa. Testausdataa tulee pystyä hyödyntämään siinä tarkoituksessa, että mahdolliset virheet tulevat esiin. Esimerkkinä on aritmeettisten yhtälöiden oikea evaluoiminen ja aritmeettiset relaatiot. On myös toivottavaa, että datassa on normaalien skenaarioiden lisäksi ääriskenaarioita. [36]

Automatisoituun funktionaaliseen testaukseen löytyy menetelmiä kirjallisuudesta [37], [38], mutta automatisoidun funktionaalisen testauksen tehokkuudesta on rajallisemmin kirjallisuutta. Awédikian ja Yannou [39] mukaan automatisoitu funktionaalinen testaus löytää virheitä enemmän ja aiemmin kuin manuaalinen testaus. Funktionaalisessa testauksessa automaattisen testauksen testikattavuus ja kyky löytää virheitä saattaa kuitenkin olla parempi kuin manuaalisessa testauksessa. Funktionaalinen testaus koostuu useista testimetodologioiden osista, joten eri testauskehyksiä voidaan käyttää [39].

Funktionaalisen testauksen automatisaatiolla on useita eri työkaluja. Esimerkiksi Seleniumia ja Cypressiä voidaan käyttää testiautomaatioon. Automatisoitujen funktionaalisten testien toteuttaminen saattaa kuitenkin olla hieman monimutkaista ja aikaavievää, joten vaihtoehtoiskustannuksen laskeminen on olennaista. Tällöin voidaan paremmin arvioida kuinka kannattavaa funktionaalisen testauksen automatisointi on. Tosin manuaalinen funktionaalinen testaus voi olla verrattain kallista ja automatisaatiolla voidaan saada kustannuksia alas [40].

3.2.2 Ei-toiminnallinen testaus

Tutkielmassa käsitellään ei-toiminnallista testausta, keskittyen suorituskyky- ja tietoturvatestaukseen. **Suorituskykytestaus** (engl. performance testing) on keskeinen osa ohjelmiston laadunvarmistusta, jossa arvioidaan ohjelmiston suorituskykyä eri olosuhteissa. Suorituskyvyn arvioimiseksi tulee ensin määrittää ohjelmiston keskeiset suorituskykyindikaattorit (engl. Key Performance Index, KPI). KPI:t voidaan jakaa kahteen päätyyppiin: palveluorientoituneisiin ja tehokkuusorientoituneisiin indikaattoreihin. Palveluorientoituneet KPI:t mittaavat ohjelmiston toimintaa loppukäyttäjän näkökulmasta, keskittyen esimerkiksi saatavuuteen ja vasteaikaan. Tehokkuusorientoituneet KPI puolestaan arvioivat ohjelmiston kykyä käyttää isännöintialustaa (engl. hosting platform) tehokkaasti, keskittyen kapasiteettiin ja suoritustehoon. Mittausvaiheessa kerätään tietoa ohjelmiston toiminnasta eri testitilanteissa, ja tuloksia verrataan etukäteen asetettuihin vaatimuksiin. Mittauksissa voidaan käyttää useita eri tekniikoita, kuten aikaleimojen ja lokitietojen analysointia, jotka antavat tietoa esimerkiksi ohjelmiston vastausajoista ja suoritustehosta. Tärkeä osa mittausprosessia on myös jatkuva monitorointi (engl. continuous monitoring), joka mahdollistaa suorituskyvyn seurantaa tuotantoympäristössä. Tämä auttaa varmistamaan, että ohjelmisto säilyttää suorituskykynsä myös todellisessa käytössä, kun käyttäjämäärät ja kuormitustilanteet vaihtelevat. Yleisesti suorituskykytestaus jaetaan perustaso- (engl. baseline), kuormitus- (engl. load), kesto- (engl. soak) ja stressitestityyppeihin (engl. stress). [26]

Perustasotestityypissä pyritään testaamaan järjestelmää suorituskyvyn lähtötason normaalilla tai vähäisellä kuormituksella. Käytännössä tavoitteena on luoda vertailuarvot myöhempiä kuormittavampia testejä varten. Kuormitustestauksessa tyypillisesti tehdään usean tunnin tai jopa päivien kestävää testausta, jossa pyritään simuloimaan asiakkaan (engl. client) lähettämiä tuhansia tai miljoonia rinnakkaisia pyyntöjä (engl. request) sovellukselle [41]. Kestotestauksessa arvioidaan

järjestelmän suorituskykyä pidemmällä aikavälillä jatkuvassa kuormituksessa yksittäisen kuormituspiikin sijaan. Stressitestauksessa taas testataan sovelluksen kykyä kestää normaalia merkittävästi korkeampi kuormitus eli korkeampaa kuormitusta kuin mihin sovellus on suunniteltu.

Suorituskykytestauksen menetelmä sisältää useita vaiheita, joilla varmistetaan ohjelmiston suorituskyky eri olosuhteissa. KPT:t toimivat suorituskyvyn arviointiperusteina. Tämän jälkeen suunnitellaan ja valmistellaan testit, jotka kattavat mahdollisimman laajan valikoiman käyttötapauksia ja kuormitustilanteita. Testit suoritetaan yleensä erilaisten simulointityökalujen avulla, jotka jäljittelevät todellisia käyttötilanteita, kuten samanaikaisten käyttäjien määrän kasvua. Tulokset analysoidaan perusteellisesti, jotta voidaan tunnistaa mahdolliset suorituskykyyn liittyvät pullonkaulat tai ongelmakohdat. Lopuksi testitulosten perusteella tehdään tarvittavat optimointitoimenpiteet, ja testit toistetaan kunnes ohjelmisto täyttää asetetut vaatimukset. [26]

Suorituskykytestauksen automaatiosta ei löydy erityisen laajaa kirjallisuutta. Manuaalisen testauksen hyötynä on sen mittaaminen suoremmin käyttäjän näkökulmasta. Esimerkiksi käyttöliittymässä käyttäjän kannalta merkittävät ongelmat on helppo löytää käyttämällä ohjelmistoa. Tällöin KPI:nä voi toimia esimerkiksi käyttäjän kokema suorituskyky (engl. performance as perceived by the user) [42]. Suorituskykytestaus on myös riippuvainen käyttökohteesta ja alustasta (esimerkiksi mobiilisovellus, web-sovellus tai tietokanta). Automaattisten testien kattavuus suorituskykytestauksessa on yleisesti manuaalisia testejä heikompaa [42]. Eri tilanteiden testaaminen ja identifioiminen on huomattavasti vaikeampaa automaattisessa testauksessa, erityisesti kun kyseessä on käyttöliittymän kautta tapahtuva testaus. Tosin automatisoinnissa voidaan testata ja simuloida tehokkaasti järjestelmän kuormittamista ja näille olennaisten, esimerkiksi suorittimen ja muistin, kuormitusta ja toimintaa. Tämä ei välttämättä anna suoraan tietoa käyttäjälle välittyvästä koke-

muksesta järjestelmän suorituskyvystä. Kuitenkin testien täysi automaatio aiheuttaa pienemmän kattavuuden ja näin kriittisiä käyttäjälle näkyviä virheitä voi jäädä huomaamatta.

Suorituskykytestaus pystytään automatisoimaan hyödyntäen simuloituja käyttäjiä. Tämä mahdollistaa suuren käyttäjämäärän testauksen samanaikaisesti, jonka testaaminen manuaalisesti voi olla haastavaa. Ongelmana automaattisissa simuloiduissa suorituskykytestauksissa voi olla juurisyyn löytäminen, joka saattaa yhä vaatia manuaalista työtä. Itse simulaatioprosessin luominen ja eri testien tekeminen on melko yksinkertaista ja virheet pystytään tehokkaasti löytämään [43]. Työkaluja automaattiseen suorituskykytestaukseen löytyy useita, esimerkiksi JMeter apache. Testien automatisointi ei siis tyypillisesti ole erityisen haastavaa, ongelmat ovat lähinnä testauksen kattavuudessa.

Esimerkiksi muistin, suorittimen käyttöä tai vastausaikaa on melko helppoa mitata automaattisesti [44]. Testauksen automatisointi joko täysin tai osittain tulee käsitellä tapauskohtaisesti suorituskykytestauksen tavoitteiden ja vaatimusten pohjalta. Ainakin kuormitusosan testaus on mahdollista toteuttaa korkealla tarkkuudella, johon manuaalinen testaus ei välttämättä yhtä hyvin sovi [41]. Automatisointi ei kuitenkaan ole välttämättä järkevää esimerkiksi käyttöliittymän testauksessa, jossa käyttäjäkokemus ei suoraan näy eri mittareista. Näin voi esimerkiksi olla, jos käyttäjäkokemuksessa vastausaika tai palvelun toimintavarmuus eivät ole jostain syystä merkittäviä. Toisaalta tyypillisesti nämä molemmat ovat käyttäjälle tärkeitä ja näiden testauksen automatisointi ja jatkuva seuraaminen on tärkeää. Kokonaisuutena voidaan sanoa, että automaattisella suorituskykytestauksella kyetään usein esimerkiksi käyttöliittymätestauksessa säästämään kustannuksissa automatisoimalla kokonaan tai osittain suorituskykytestaus [42].

Turvallisuustestaus (engl. security testing) on prosessi, jonka avulla pyritään varmistamaan ohjelmiston ja järjestelmän kyky suojautua erilaisilta turvallisuusu-

hilta. Toimivan ohjelmiston tulee pystyä toimimaan myös poikkeustilanteissa, ylläpitäen sen toiminnallisuudet ja turvaten kriittinen data. Tämä on erityisen tärkeää SaaS-pohjaisissa ohjelmistoissa, joissa tietoturvan merkitys korostuu entisestään. Onnistunut turvallisuustestaus perustuu laadukkaaseen riskianalyysiin, jossa jo suunnitteluvaiheessa pyritään tunnistamaan riskiskenaariot. Turvallisuusriskejä voi esiintyä tietoverkko-, käyttöjärjestelmä ja applikaatiotasolla [45]. Esimerkiksi applikaation huono muistinhallinta voi johtaa vakaviin haavoittuvuuksiin, kuten puskurin ylivuotoon (engl. buffer-overflow). Näiden riskien minimoimiseksi kehittäjät voivat käyttää erilaisia työkaluja, jotka auttavat havaitsemaan tyypilliset ohjelmointivirheet. Nämä työkalut voivat olla osa automaattista testausta, mikä lisää ohjelmiston turvallisuutta.

Turvallisuustestauksen mittaaminen perustuu testitulosten tarkkaan analysointiin, jonka avulla arvioidaan ohjelmiston kykyä vastata erilaisiin uhkakuviin. Mittausvaiheessa kerätään tietoa esimerkiksi siitä, kuinka monta haavoittuvuutta löydettiin ja niiden vakavuusasteet. Lisäksi voidaan mitata kuinka tehokkaasti testit kattavat kaikki olennaiset hyökkäysvektorit ja miten ohjelmisto suoriutuu niistä. Erilaisia pisteytyksiä voidaan hyödyntää ohjelmiston turvallisuuden arvioinnissa. Lisäksi voidaan mitata ohjelmiston aikaa palautua vakavasta hyökkäyksestä normaalitilaan tai kuinka laajaa hyökkäystä ohjelmisto kestää menettämättä kykyään toimia. [45]

Vaativassa turvallisuustestauksessa testaajien osaamistason on ratkaisevaa. Hyvät testaajat käyttävät usein niin sanottua riskipohjaista (engl. risk-based) lähestymistapaa, jossa asetutaan hyökkääjän asemaan. Tämä lähestymistapa mahdollistaa aidon hyökkäyksen simuloinnin. Tämän onnistumiseksi hyvän testaajan tulee olla luova. Seuraavaksi esitellään riskipohjaisen lähestymistavan menetelmät. [45], [46]

Staattinen analyysi (engl. static analysis) on yksi turvallisuustestauksen perustekniikoista. Staattisessa testauksessa ohjelmisto analysoidaan ilman, että sitä suo-

ritetaan. Tällöin voidaan automaattisesti löytää esimerkiksi muistin hallintaan liittyviä ongelmia, kuten puskurin ylivuotoja, tunnistaa kuollutta koodia (engl. dead code) ja erilaisia turvallisuusriskejä aiheuttavia suorituspolkuja. Tämä menetelmä tarjoaa kehittäjille mahdollisuuden löytää ja korjata virheet varhaisessa kehitysvaiheessa. Staattinen testaus saattaa myös sisältää eri riippuvuuksien aiheuttamia tietoturvariskejä. [46]

Riskianalyysi on turvallisuustestauksen lähestymistapa, jossa ohjelmistoarkkitehtuuria arvioidaan tietoturvariskien näkökulmasta. Riskianalyysin tavoitteena on tunnistaa ohjelmiston haavoittuvuudet, ymmärtää niiden vaikutukset järjestelmään sekä arvioida ja hallita näistä aiheutuvia riskejä. Tämä analyysi auttaa priorisoimaan turvallisuustestauksen kohteita ja kohdentamaan resurssit tehokkaasti suurimpiin uhkiin. [46]

Penetraatiotestaus on toinen keskeinen metodologinen lähestymistapa. Penetraatiotestauksessa yritetään tarkoituksellisesti murtautua ohjelmistoon ja arvioida sen kykyä kestää hyökkäyksiä sekä suojata arkaluonteista dataa. Onkin tärkeää suunnitella nämä hyökkäykset mahdollisimman realistisesti. Penetraation aikana tulee järjestelmästä kerätä systemaattisesti tietoa, jotta haavoittuvuudet voidaan analysoida ja raportoida tehokkaasti. [46]

Turvallisuustestauksessa kohdataan monia haasteita, jotka voivat vaikuttaa testauksen tehokkuuteen ja tarkkuuteen. Testaukselle tyypillinen haaste, virheiden havaitseminen tarpeeksi aikaisessa vaiheessa on erityisen kriittistä ohjelmiston turvallisuustestauksessa. Ohjelmiston turvallisuudessa löytyvien kriittisten haavoittuvuuksien löytämättä jääminen voi aiheuttaa merkittäviä seurauksia ohjelmiston kehittäjälle ja loppukäyttäjälle. [46]

Tyypillisesti ohjelmiston turvallisuuteen liittyvät virheet tapahtuvat, kun virheen tai vikatilan hallintaan ei kiinnitetä riittävästi huomiota, tiedon suojaaminen on puutteellista tai pääsynhallintaa ei luoda laadukkaasti. Tietoturvahyökkäysten

jatkuva kehitys ja taloudelliset motiivit tekevät vastapuolesta usein erittäin innovatiivisen ja taitavan, mikä lisää painetta tehokkaan turvallisuustestauksen suorittamiseen. Lisäksi hyviä testaajia voi olla vaikeaa löytää ja tiettyjen testien automatisointi voi olla haastavaa.

Yleisin turvallisuushaavoittuvuus ohjelmistossa on suunnittelutason haavoittuvuus. Tällaisia virheitä on hyvin vaikeaa automatisoida, joten manuaalista testausta tyypillisesti tarvitaan ainakin jonkin verran [46]. Automaattisten testien kattavuus ei siis tyypillisesti ole yhtä hyvä kuin manuaalisilla testauksilla, jolloin näitä virheitä voidaan löytää. Manuaaliset testit ovat siis todennäköisesti myös tehokkaampia. Empiiristä näyttöä koodimuutosten aiheuttamien turvallisuusriskien löytämisestä automaattisesti kehitettyjen ja ajettujen testien myötä löytyy [27]. Kuitenkin yleisesti laajempi empiirinen näyttö kokonaisvaltaisesta automaattisesta turvallisuustestauksesta puuttuu. Onkin todennäköistä, että turvallisuustestausta ei voida ainakaan vielä täysin automatisoida, vaan jonkinlainen kombinaatio automaattisia ja manuaalisia testauksia on optimaalinen. Tällöin voidaan saavuttaa automaation tuoma tehokkuus ja manuaalisten testien tuoma kattavuus suunnittelutason turvallisuusriskien löytämiseksi.

Turvallisuustestauksen automatisointi on yleisesti haastavaa, mutta siihen löytyy erilaisia lähestymistapoja. Turvallisuustestien automaatiossa kyetään löytämään hyökkäysvektoreita ja testaamaan onnistuneesti esimerkiksi olemassa olevien tietovarastojen (engl. repositories) turvallisuusriskejä [47]. Ongelmana tietysti on osoittaa kuinka suuri osa turvallisuusriskeistä onnistuttiin löytämään, tyypillisesti niin sanottuja nollapäivähaavoittuvuuksia (engl. zero-day vulnerability) jää huomaamatta.

Xu, Tu, Sanford et al. [27] pystyivät osoittamaan, että kokeessa käytetty automaattinen turvallisuustestaus kykeni löytämään suurimman osan luoduista mutaatioista. Turvallisuustestauksella voidaan parantaa ohjelmiston laatua ja löytää turvallisuusriskejä staattisia testejä paremmin esimerkiksi hyödyntäen niin sanot-

tua Fuzzing-testausmenetelmää, jossa annetaan ohjelmistolle vääriä syötteitä haavoittuvuuksien löytämiseksi [48]. Automaattista turvallisuustestausta siis kannattaa hyödyntää merkittävän osana turvallisuustestausta monien haavoittuvuuksien identifiointiin, yksinkertaisimmillaan tarkistamaan onko jokin portti jäänyt auki tai palomuuri aktivoitu. Tosin osa laadukkaasta turvallisuustestauksesta ainakin vielä nojaa taitavan testaajan oman päättelykyvyn ja luovuuteen, jota on vaikeaa automatisoida.

Automaattisen turvallisuustestien luomiseen on eri työkaluja. Esimerkiksi Android-sovelluksille on kehitetty metodeja automaattiselle testaukselle käyttämällä evolutionaarista algoritmia [49]. Myös web-sovelluksille on kehitetty malleja [50]. Testit voidaan toteuttaa esimerkiksi käyttäen Jitiä. Turvallisuustestauksen kustannustehokkuudesta ei löydy laajasti tutkimuksia. Kuitenkin kirjallisuudessa on mainittu, että olisi hyödyllistä tehdä kustannustehokkuusarvio sekä staattisesta että automaattisesta turvallisuustestauksesta. Samalla tulee tehdä arvioi, onko manuaalinen vai automaattinen lähestymistapa tavoitteiltaan ja kustannuksiltaan optimaalinen [27].

3.3 Regressiotestaus

Regressiotestaus (engl. regression testing) on prosessi, jossa testataan olemassa olevaa ohjelmistoa muutosten jälkeen. Muutos voi olla esimerkiksi ohjelmistossa olevan virheen korjaus tai uuden logiikan lisääminen. Monimutkaisissa ohjelmistoissa muutoksilla voi olla odottamattomia sivuvaikutuksia, jotka vaikuttavat ohjelmiston muihin toimintoihin. Regressiotestauksella voidaan varmistua siitä, että ohjelmisto toimii edelleen suunnitellusti ja uudet ominaisuudet toimivat odotetulla tavalla. Regressiotestaus on myös merkittävä osa ohjelmiston ylläpitoa, sillä sen avulla varmistetaan, että ohjelmiston aiemmat toiminnot eivät lakkaa toimimasta tai muutu ja suorituskyky pysyy hyvällä tasolla [51], [52]. Regressiotestaus voidaan kohdistaa

vain muokattuihin osiin tai koko ohjelmistoon. Automaattisen testauksen hyötynä regressiotestauksessa on, että testit voidaan helposti ajaa säännöllisesti pienellä marginaalikustannuksella koko ohjelmistolle. Uuden toiminnallisuuden regressiotestaus aloitetaan tyypillisesti välittömästi, kun uusi toiminnallisuus on implementoitu.

Regressiotestausta on kaksi päätyyppiä: progressiivinen (engl. progressive) ja korjaava (engl. corrective) regressiotestaus. Päätyyppi riippuu siitä, muuttuuko ohjelmiston määrittely vai ei. Progressiivinen regressiotestaus sisältää muutetun määritelmän. Korjaavassa regressiotestauksessa taas määrittely ei muutu. Ohjelman ohjeita ja suunnitteluratkaisuja voidaan muuttaa, mutta suurin osa aiemmista testitapauksista on edelleen käyttökelpoisia, koska ne määrittävät oikein syöte-tulos -suhteen. Muutokset ohjelman ohjauksessa ja tietovirroissa voivat kuitenkin vaikuttaa siihen, että jotkin testitapaukset eivät enää testaa aiemmin kohdennettuja ohjelmarakenteita. Progressiivinen regressiotestaus tehdään yleensä sovittavan tai täydentävän ylläpidon jälkeen, kun taas korjaava regressiotestaus tehdään ohjelmiston korjausten jälkeen. Sovittava tai täydentävä ylläpito tapahtuu tyypillisesti säännöllisin väliajoin, jolloin myös progressiivinen regressiotestaus tulee tehdä säännöllisesti. Joidenkin ohjelmistojen osalta ajetaan öisin kaikki automaattiset regressiotestit. [53]

Testitapausten luomiseen regressiotestauksessa käytetään kahta päämenetelmää. Ensimmäinen on määrittelypohjainen testaus, eli niin sanottu mustan laatikon testaus (black-box testing). Nämä testit perustuvat ohjelmiston spesifikaatioon ilman suunnittelu- ja toteutusdokumentaation huomioimista. Funktionaalinen testaus on esimerkki tämän tyyppisestä testauksesta. Toinen on rakenteellinen testaus eli niin sanottu valkoisen laatikon testaus (white-box testing), jossa testit perustuvat ohjelman ohjaus- ja tietovirtoihin. Tämä menetelmä edellyttää testejä ohjelman eri osien suorittamiseksi tai niiden yhdistelmien suorittamiseksi. [51], [54]

Automaattinen regressiotestaus on varmempaa ja löytää paremmin virheet kuin manuaalinen testaus [55]. Automaattiset testit löysivät enemmän virheitä ja toden-

näköisemmin vakavia virheitä. Automaattisessa testauksessa ei kuitenkaan ole ongelmana manuaaliselle testaukselle tyypillisiä testaajasta riippuvia inhimillisiä virheitä. Tutkimuksessa lisäksi todettiin, että automaattinen testaus vaati isommat kustannukset alussa johtuen esimerkiksi automaatioskriptien kirjoittamisesta, mutta kustannukset kääntyvät melko nopeasti automaattisen testauksen eduksi. Laadun kannalta on siis selkeästi suositeltavaa regressiotestauksen automatisaatio testitapausten pohjalta. Lisäksi manuaalisten testien suorittamisessa on harvoin tarpeeksi aikaa suorittaa suurta määrää testejä [52]. Automaattisilla testeillä saadaan siis tyypillisesti suoritettua merkittävästi enemmän testejä ja ilman riskiä inhimillisille virheille.

Regressiotestauksen automatisointiin löytyy paljon eri työkaluja ja testauskehyksiä. Esimerkiksi Robot framework on yleisesti käytetty testien automatisaatioon. Testien luomiseen kuitenkin liittyy merkittävä kiinteä kustannus, eikä testejä voi yleisesti luoda automaattisesti.

Regressiotestauksen automatisaatiossa ongelmana on merkittävät automatisaatiokustannukset, jotka testien tekemisestä syntyy. Testien automatisaatio ei ole järkevää kaikissa tilanteissa, vaan riippuu testien luomisen kiinteistä kustannuksista, niiden ylläpitokustannuksista ja kuinka usein testejä ajetaan. Tapauksissa joissa kaikki regressiotestejä ajetaan säännöllisesti esimerkiksi säännöllisten ohjelmistomuutoksien vuoksi, on automaattinen testaus todennäköisesti järkevämpää. Esimerkiksi N Bhatt, Rajasekhara Babu ja J Bhatt [56] ovat osoittaneet automaattisen regressiotestauksen olevan kustannustehokas tapa verrattuna manuaaliseen testaukseen huolimatta ajasta, joka menee testien kirjoittamiseen.

3.4 Analyysi

Taulukossa 3.1 esitellään löydetyt testausmenetelmät, jotka soveltuvat automaattiseen jatkuvaan testaukseen. Menetelmät on kategorisoitu kyllä- ja osittain-katego-

rioihin.

Taulukko 3.1: Jatkuvan testauksen metodologioiden soveltuvuus automaattiseen testaukseen.

Menetelmä	**Automatisoitava**
Yksikkötestaus	Kyllä
Integraatiotestaus	Kyllä
Suorituskykytestaus	Kyllä
Turvallisuustestaus	Osittain
Regressiotestaus	Kyllä
Hyväksyntätestaus	Kyllä
Funktionaalinen testaus	Kyllä

Luvussa 2 esittelin ohjelmistotestauksen neljä eri tasoa: testausmetodologia, testaustasot, testaustyyppi ja testausmenetelmä. Testausmetodologia on tässä tapauksessa koko jatkuva testaus. Tutkielmassa jätettiin läpikäymättä menetelmät, jotka kirjallisuushaussa todettiin epäsopiviksi automaattisen jatkuvan testauksen alle ja rajattiin pois, esimerkiksi tutkiva testaus (engl. exploratory testing). Kokonaisuudessaan voidaan todeta, että jatkuva testaus, jo määritelmänsäkin mukaan, sopii melko hyvin automatisoitavaksi. Ihmistyötä tarvitaan vielä tyypillisesti testien kirjoittamiseen ja testausmenetelmiin, joissa ihmisen luovuus on tärkeää, kuten vaativassa turvallisuustestauksessa.

Testaustasoja määriteltiin neljä: yksikkötestaus, integraatiotestaus, järjestelmätestaus ja hyväksyntätestaus. Näistä yksikkötestaus on erityisen hyvin automaattiseen testaukseen sopiva menetelmä, testien luominen ja testien ajaminen kannattaa automatisoida käytännössä aina. Hyväksyntätestauksen suoritus on myös automatisoitavissa, mikäli ohjelmiston loppukäyttäjän kanssa saadaan luotua hyvä yhteys ja loppukäyttäjän tapa käyttää järjestelmää saadaan tarkasti määriteltyä ja muutet-

tua automaattisten testien muotoon. Integraatiotestaus on myös automatisoitavissa; automaattisia testejä tulee kirjoittaa sitä mukaa kun uusia integraatiota tehdään eri rajapintoihin. Järjestelmätestauksen automatisointia ei käsitelty kirjallisuuskatsauksessa ulosrajauksen vuoksi. Ongelmana on, että koko järjestelmä tulee testata ja tällöin automatisointi on todennäköisesti hankalaa. Tämä vaatisi ison määrän automatisoituja testejä eri osa-alueilla ja erikoisosaamista. Manuaalinenkin järjestelmätestaus on hankalaa sen laajuuden vuoksi.

Testaustyypeistä käsiteltiin toiminnallinen tyyppi, funktionaalinen testaus, ja eitoiminnallisista tyypeistä suorituskykytestaus ja turvallisuustestaus. Funktionaalinen testaus on mahdollista automatisoida kokonaan. Automatisoinnin haittapuolena voi olla on heikompi testikattavuus kuin manuaalisessa testauksessa. Suorituskykytestaus on mahdollista automatisoida ja erityisesti muistin ja suorittimen käyttöä ja vastausaikaa koskevat testit ovat mahdollista automatisoida. Automaatiossa käytetään simulointia, jolla voidaan kuormittaa järjestelmää ja tähän automatisointi sopii hyvin. Ongelmana saattaa tosin olla automaattisissa testeissä juurisyyn löytäminen. Testin epäonnistumisen syy ei välttämättä ole yhtä selvää kuin tilanteessa, jossa testaajalla on korkea osaaminen kyseisistä testeistä. Turvallisuustestaus voidaan automatisoida osittain. Osa-alue, jota on vaikeaa automatisoida, on luova turvallisuustestaus, jossa yritetään eri tavoin löytää haavoittuvuuksia järjestelmästä samalla kun sen toimintaa opetellaan. Tämä vaatii korkeaa osaamista ja ihmiselle tyypillistä luovuutta.

Testausmenetelmä regressiotestaus on melko yksinkertaisesti automatisoitava menetelmä. Regressiotestaus on luonteeltaan sellaista, että se halutaan suorittaa säännöllisesti, kun ohjelmistoa kehitetään. Regressiotestit voidaan ajaa päivittäin jo ohjelmistokehityksen aikana ja automatisoinnin hyödyt ovat tällöin ilmeisiä. Tällaista testausta ei ihmistyövoimalla realistisesti voida suorittaa suurien taloudellisten kulujen vuoksi.

4 Yhteenveto

Ohjelmistotestaus on merkittävä kustannus ohjelmistokehityksessä, vieden joissain tapauksissa jopa 50 % ohjelmistokehityksen resursseista [13]. Tutkielmassa selvitettiin kirjallisuudesta löytyvät automatisoitavat jatkuvan testauksen menetelmät jakaen ne testauksen osa-alueisiin: testaustasot, testaustyypit ja testausmenetelmät.

Tutkimuskysymyksenä TK1 oli: ”Mitkä ohjelmistotestauksen osa-alueet soveltuvat automaattiseen jatkuvaan testaukseen?”. Tutkielmassa havaittiin, että kaikki testauksen osa-alueet voivat hyödyntää automaatiota ainakin osittain. Esimerkiksi yksikkötestaus ja regressiotestaus soveltuvat hyvin automaatiolle, kun taas järjestelmätestauksen ja turvallisuustestauksen kohdalla ihmisen rooli on edelleen tärkeä, sillä luovuutta vaativia testejä on vaikeaa automatisoida.

Tutkimuskysymyksenä TK2 oli: ”Mitkä ovat automaattisen jatkuvan testauksen merkittävimmät edut ja haasteet?”. Keskeisiksi eduiksi tunnistettiin kustannustehokkuuden paraneminen usein toistetuissa testeissä, virheiden nopeampi ja todennäköisempi löytäminen sekä ohjelmistokehityksen pysähdysten välttäminen. Suurimpana haasteena on alkuvaiheen investointikustannukset, sillä automaattisten testien kehittäminen ja ylläpito vaatii huomattavia resursseja. Lisäksi erityisesti hyväksyntätestit edellyttävät hyvää yhteistyötä loppukäyttäjän kanssa.

Tutkimuskysymyksenä TK3 oli: ”Miten automaattinen jatkuva testaus voidaan toteuttaa tehokkaasti ohjelmistokehitysprosessissa?”. Testien luominen tulee aloittaa ohjelmistokehitysprosessin alussa hyödyntäen optimaalisia testausmenetelmiä

ja -työkaluja. Testien onnistuminen ja niiden suorittamisen kustannustehokkuus on kriittistä ohjelmistoprojektin onnistumisen kannalta. Testien tulee näin ollen kattaa tarpeeksi suuri osa ohjelmistoa ja löytää tehokkaasti virheitä. Muuten riskinä on ohjelmiston laadun yliarviointi. Testien tulee myös pohjautua ohjelmistoprojektin tavoitteisiin; joissain ohjelmistoissa virheet ovat sallittuja ja kevyempi ohjelmistotestaus on riittävä, kun taas kriittisissä ohjelmistoissa virheitä ei sallita lainkaan[1]

Hyödyllinen jatkotutkimusaihe on automaattisen testauksen yhdistäminen keinoälyyn, sillä kirjallisuudesta löytyy vain rajatusti artikkeleita huolimatta ohjelmistotestauksen merkityksestä. Keinoäly esimerkiksi avaa mahdollisuuden luoda esimerkiksi erilaisia ennakoivia testejä ja löytää erilaisia yhteyksiä esimerkiksi hyödyntäen valvomatonta koneoppimista (engl. unsupervised machine learning). Lisäksi olisi mielenkiintoista selvittää, kuinka automaattinen testaus voidaan integroida vielä entistä tiiviimmin muihin jatkuviin menetelmiin.

Kokonaisuutena ohjelmistotestauksen automatisointi hyödyntäen jatkuvaa testausta on usein optimaalinen valinta ja jopa ainut mahdollisuus erityisesti laajoissa ohjelmistoprojekteissa. Automaattisten jatkuvien testien käyttö säästää työvoimakustannuksia, pysähdykset ohjelmistokehityksessä saadaan minimoitua, virheitä voidaan löytää enemmän ja inhimillisten virheiden riski poistetaan. Automaattisten testien luominen manuaalisesti tosin vaatii merkittäviä alkuinvestointeja, mikäli testit ohjelmoidaan manuaalisesti. Automaattisen jatkuvan testauksen hyödyntäminen voi näin ollen merkittävästi parantaa ohjelmistokehitysprosessin kustannustehokkuutta, nopeutta ja ohjelmiston laatua.

[1]Esimerkiksi ydinvoimalassa tai auton hallintajärjestelmässä käytetty ohjelmisto.

Lähdeluettelo

[1] B. Oliinyk ja V. Oleksiuk, "Automation in software testing, can we automate anything we want?", teoksessa *CS&SE@ SW*, 2019, s. 224–234.

[2] L. Baresi ja M. Pezzè, "An Introduction to Software Testing", *Electronic Notes in Theoretical Computer Science*, vol. 148, nro 1, s. 89–111, 2006, Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004), ISSN: 1571-0661. DOI: 10.1016/j.entcs.2005.12.014.

[3] D. Saff ja M. Ernst, "Reducing wasted development time via continuous testing", s. 281–292, 2003. DOI: 10.1109/ISSRE.2003.1251050.

[4] L. Lazic ja N. Mastorakis, "Cost effective software test metrics", vol. 7, kesäkuu 2008.

[5] A. Memon, Z. Gao, B. Nguyen et al., "Taming Google-scale continuous testing", teoksessa *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, s. 233–242. DOI: 10.1109/ICSE-SEIP.2017.16.

[6] M. Shahin, M. Ali Babar ja L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices", *IEEE Access*, vol. 5, s. 3909–3943, 2017, ISSN: 2169-3536. DOI: 10.1109/access.2017.2685629.

[7] V. Khorikov, *Unit Testing:Principles, Practices and Patterns*. Manning Publications, 2020.

[8] IBM. "What is continuous testing?" (Ei julkaisupäivää), url: `https://www.ibm.com/topics/continuous-testing` (viitattu 13. 07. 2024).

[9] Capgemini, "World Quality Report", Capgemini, 2023.

[10] A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams", teoksessa *Future of Software Engineering (FOSE '07)*, 2007, s. 85–103. DOI: `10.1109/FOSE.2007.25`.

[11] M. A. Umar, "A Study of Software Testing: Categories, Levels, Techniques, and Types", kesäkuu 2020. DOI: `10.36227/techrxiv.12578714.v1`.

[12] I. Hooda ja R. S. Chhillar, "Software test process, testing types and techniques", *International Journal of Computer Applications*, vol. 111, nro 13, 2015.

[13] F. Elberzhager, A. Rosbach, J. Münch ja R. Eschbach, "Reducing test effort: A systematic mapping study on existing approaches", *Information and Software Technology*, vol. 54, nro 10, s. 1092–1106, 2012, ISSN: 0950-5849. DOI: `10.1016/j.infsof.2012.04.007`.

[14] L. Li-hong, "Research and implementation of software automatic test", teoksessa *IOP Conference Series: Earth and Environmental Science*, IOP Publishing, vol. 69, 2017, s. 012 159.

[15] B. Fitzgerald ja K.-J. Stol, "Continuous software engineering and beyond: trends and challenges", teoksessa *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, sarja RCoSE 2014, Association for Computing Machinery, 2014, s. 1–9, ISBN: 9781450328562. DOI: `10.1145/2593812.2593813`.

[16] A. van Hoorn, W. Hasselbring, J. Waller, J. Ehlers, S. Frey ja D. Kieselhorst, "Continuous monitoring of software services: Design and application of the Kieker framework", 2009.

[17] A. Birk ja D. Rombach, "A Practical Approach to Continuous Improvement in Software Engineering", *Software Quality: State of the Art in Management, Testing, and Tools*, s. 34–45, 2001.

[18] V. Garousi ja M. V. Mäntylä, "When and what to automate in software testing? A multi-vocal literature review", *Information and Software Technology*, vol. 76, s. 92–117, 2016, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2016.04.015.

[19] K. Sen, D. Marinov ja G. Agha, "CUTE: a concolic unit testing engine for C", ESEC/FSE-13, s. 263–272, 2005. DOI: 10.1145/1081706.1081750.

[20] L. Williams, G. Kudrjavets ja N. Nagappan, "On the Effectiveness of Unit Test Automation at Microsoft", teoksessa *2009 20th International Symposium on Software Reliability Engineering*, 2009, s. 81–89. DOI: 10.1109/ISSRE.2009.32.

[21] D. Serra, G. Grano, F. Palomba, F. Ferrucci, H. C. Gall ja A. Bacchelli, "On the Effectiveness of Manual and Automatic Unit Test Generation: Ten Years Later", teoksessa *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, 2019, s. 121–125. DOI: 10.1109/MSR.2019.00028.

[22] A. Bacchelli ja P. Ciancarini, "On the Effectiveness of Manual and Automatic Unit Test Generation", lokakuu 2008, s. 252–257. DOI: 10.1109/ICSEA.2008.66.

[23] B. Souza ja P. Machado, "A Large Scale Study On the Effectiveness of Manual and Automatic Unit Test Generation", teoksessa *Proceedings of the XXXIV*

Brazilian Symposium on Software Engineering, sarja SBES '20, Natal, Brazil: Association for Computing Machinery, 2020, s. 253–262, ISBN: 9781450387538. DOI: 10.1145/3422392.3422407.

[24] J. S. Kracht, J. Z. Petrovic ja K. R. Walcott-Justice, "Empirically Evaluating the Quality of Automatically Generated and Manually Written Test Suites", teoksessa *2014 14th International Conference on Quality Software*, 2014, s. 256–265. DOI: 10.1109/QSIC.2014.33.

[25] H. Leung ja L. White, "A study of integration testing and software regression at the integration level", teoksessa *Proceedings. Conference on Software Maintenance 1990*, 1990, s. 290–301. DOI: 10.1109/ICSM.1990.131377.

[26] I. Molyneaux, *The Art of Application Performance Testing: From Strategy to Tools*. O'Reilly Media, 2015.

[27] D. Xu, M. Tu, M. Sanford, L. Thomas, D. Woodraska ja W. Xu, "Automated Security Test Generation with Formal Threat Models", *IEEE Transactions on Dependable and Secure Computing*, vol. 9, nro 4, s. 526–540, 2012. DOI: 10.1109/TDSC.2012.24.

[28] M. Bures, B. S. Ahmed, V. Rechtberger et al., "PatrIoT: IoT Automated Interoperability and Integration Testing Framework", teoksessa *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2021, s. 454–459. DOI: 10.1109/ICST49551.2021.00059.

[29] D. Xu, W. Xu, M. Tu, N. Shen, W. Chu ja C.-H. Chang, "Automated integration testing using logical contracts", *IEEE Transactions on Reliability*, vol. 65, nro 3, s. 1205–1222, 2015.

[30] R. Miller ja C. T. Collins, "Acceptance testing", *Proc. XPUniverse*, vol. 238, 2001.

[31] H. K. N. Leung ja P. W. L. Wong, "A study of user acceptance tests", *Software Quality Journal*, vol. 6, nro 2, s. 137–149, lokakuu 1997, ISSN: 0963-9314. DOI: 10.1023/A:1018503800709.

[32] G. Melnik, F. Maurer ja M. Chiasson, "Executable acceptance tests for communicating business requirements: customer perspective", teoksessa *AGILE 2006 (AGILE'06)*, 2006, 12 pp.–46. DOI: 10.1109/AGILE.2006.26.

[33] G. Melnik ja F. Maurer, "Multiple Perspectives on Executable Acceptance Test-Driven Development", teoksessa *Agile Processes in Software Engineering and Extreme Programming*, G. Concas, E. Damiani, M. Scotto ja G. Succi, toim., Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, s. 245–249, ISBN: 978-3-540-73101-6.

[34] B. Haugset ja G. K. Hanssen, "The Home Ground of Automated Acceptance Testing: Mature Use of FitNesse", teoksessa *2011 Agile Conference*, 2011, s. 97–106. DOI: 10.1109/AGILE.2011.37.

[35] E. Alégroth, M. Nass ja H. H. Olsson, "JAutomate: A Tool for System- and Acceptance-test Automation", teoksessa *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, s. 439–446. DOI: 10.1109/ICST.2013.61.

[36] W. E. Howden, "A functional approach to program testing and analysis", *IEEE Transactions on Software Engineering*, vol. SE-12, nro 10, s. 997–1005, 1986. DOI: 10.1109/TSE.1986.6313016.

[37] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. H. Tse, F.-C. Kuo ja T. Y. Chen, "Automated functional testing of online search services", *Software Testing, Verification and Reliability*, vol. 22, nro 4, s. 221–243, 2012. DOI: 10.1002/stvr.437.

[38] L. Hillah, A.-P. Maesano, L. Maesano, F. De Rosa, F. Kordon ja P.-H. Wuillemin, ”Service functional testing automation with intelligent scheduling and planning”, huhtikuu 2016, s. 1605–1610. DOI: 10.1145/2851613.2851807.

[39] R. Awédikian ja B. Yannou, ”A practical model-based statistical approach for generating functional test cases: application in the automotive industry”, *Journal of Software Testing, Verification and Reliability*, vol. 24, nro 2, s. 85–123, elokuu 2012. DOI: 10.1002/stvr.1479.

[40] P. Tramontana, D. Amalfitano, N. Amatucci ja A. R. Fasolino, ”Automated functional testing of mobile applications: a systematic mapping study”, *Software Quality Journal*, vol. 27, s. 149–201, 2019.

[41] Z. M. Jiang, A. E. Hassan, G. Hamann ja P. Flora, ”Automated performance analysis of load tests”, teoksessa *2009 IEEE International Conference on Software Maintenance*, 2009, s. 125–134. DOI: 10.1109/ICSM.2009.5306331.

[42] A. Adamoli, D. Zaparanuks, M. Jovic ja M. Hauswirth, ”Automated GUI performance testing”, *Software Quality Journal*, vol. 19, nro 4, s. 801–839, joulukuu 2011. DOI: 10.1007/s11219-011-9135-x.

[43] J. Wang ja J. Wu, ”Research on performance automation testing technology based on JMeter”, teoksessa *2019 International Conference on Robots & Intelligent System (ICRIS)*, IEEE, 2019, s. 55–58.

[44] G.-H. Kim, Y. G. Kim ja S.-K. Shin, ”Software Performance Test Automation by Using the Virtualization”, vol. 215, s. 1191–1199, joulukuu 2013. DOI: 10.1007/978-94-007-5860-5_143.

[45] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu ja A. Pretschner, ”Chapter One - Security Testing: A Survey”, teoksessa sarja Advances in Computers, A. Memon, toim., vol. 101, Elsevier, 2016, s. 1–51. DOI: 10.1016/bs.adcom.2015.11.003.

[46] B. Potter ja G. McGraw, ”Software security testing”, *IEEE Security & Privacy*, vol. 2, nro 5, s. 81–85, 2004. DOI: 10.1109/MSP.2004.84.

[47] Á. J. Varela-Vaca, R. M. Gasca, J. A. Carmona-Fombella ja M. T. Gómez-López, ”AMADEUS: towards the AutoMAteD secUrity teSting”, teoksessa *Proceedings of the 24th ACM Conference on Systems and Software Product Line: Volume A - Volume A*, sarja SPLC ’20, Montreal, Quebec, Canada: Association for Computing Machinery, 2020, ISBN: 9781450375696. DOI: 10.1145/3382025.3414952.

[48] G. Tóth, G. Kőszegi ja Z. Hornák, ”Case study: automated security testing on the trusted computing platform”, teoksessa *Proceedings of the 1st European workshop on system security*, 2008, s. 35–39.

[49] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek ja A. Stavrou, ”A whitebox approach for automated security testing of Android applications on the cloud”, teoksessa *2012 7th International Workshop on Automation of Software Test (AST)*, 2012, s. 22–28. DOI: 10.1109/IWAST.2012.6228986.

[50] F. Wotawa ja J. Bozic, ”Plan It Automated Security Testing Based on Planning”, teoksessa *Testing Software and Systems*, M. G. Merayo ja E. M. de Oca, toim., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, s. 48–62, ISBN: 978-3-662-44857-1.

[51] T. Xie ja D. Notkin, ”Checking inside the black box: regression testing by comparing value spectra”, *IEEE Transactions on Software Engineering*, vol. 31, nro 10, s. 869–883, 2005. DOI: 10.1109/TSE.2005.107.

[52] F. A. K. P. G. Sutapa, S. S. Kusumawardani ja A. E. Permanasari, ”A Review of Automated Testing Approach for Software Regression Testing”, *IOP Conference Series: Materials Science and Engineering*, vol. 846, nro 1, s. 012 042, toukokuu 2020. DOI: 10.1088/1757-899X/846/1/012042.

[53] H. Leung ja L. White, "Insights into regression testing (software testing)", teoksessa *Proceedings. Conference on Software Maintenance - 1989*, 1989, s. 60–69. DOI: 10.1109/ICSM.1989.65194.

[54] S. Nidhra ja J. Dondeti, "Black box and white box testing techniques-a literature review", *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, nro 2, s. 29–50, 2012.

[55] I. Dobles, A. Martínez ja C. Quesada-López, "Comparing the effort and effectiveness of automated and manual tests", s. 1–6, 2019. DOI: 10.23919/CISTI.2019.8760848.

[56] A. N Bhatt, M. Rajasekhara Babu ja A. J Bhatt, "Automation testing software that aid in efficiency increase of regression process", *Recent Patents on Computer Science*, vol. 6, nro 2, s. 107–114, 2013.