Tuire Lappalainen

TESTAUS OSANA OHJELMISTOKEHITYSTÄ

JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA
2019

TIIVISTELMÄ

Lappalainen, Tuire
Testaus osana ohjelmistokehitystä
Jyväskylä: Jyväskylän yliopisto, 2019, 26 s.
Tietojärjestelmätiede, kandidaatintutkielma
Ohjaaja: Seppänen, Ville

Ohjelmistokehitys koskettaa meistä jokaista tavalla tai toisella lähes päivittäin. Hyvin suunniteltuja ohjelmistoja ja järjestelmiä ei edes huomaa, kun taas huonosti toimivat järjestelmät jäävät mahdollisuuksien mukaan seuraavalla kerralla käyttämättä. Tässä kandidaatintutkielmassa pyritään kirjallisuuskatsauksen avulla kartoittamaan testaamisen roolia ohjelmistokehityksessä sekä ohjelmistotestaamisen tulevaisuuden kehityssuuntia. Tutkielmassa esitellään yleisimmin kirjallisuudessa esiintyvät mallit sekä ohjelmistokehityksestä että ohjelmistotestaamisesta. Lisäksi esitetään kirjallisuudesta löytyviä ajatuksia testaamisen roolista ja kehityssuunnista. Tutkielman löydöksenä on, että testaamisen merkitys ohjelmistokehityksessä on olennainen sekä onnistuneen ohjelmiston syntymisen että kustannusten kannalta. Tutkielmassa tulevaisuuden kehityssuuntina esitetään testiautomaation kehittämisen merkittävyys sekä neljä tulevaisuudennäkymää, jotka ovat universaalitestausteoria, automatisoitu testaaminen, kehitysympäristöt sekä testipohjainen ohjelmistorakenne.

Asiasanat: testaaminen, ohjelmistotestaus, testiautomaatio, ohjelmistokehitys,

ABSTRACT

Lappalainen, Tuire
Testing as a part of software development
Jyväskylä: University of Jyväskylä, 2019, 26 pp.
Information Systems Science, Bachelor´s Thesis
Supervisor: Seppänen, Ville

Software testing touches us all in a way or another on daily basis. Well designed programs and systems one does not even notice, but systems not working properly one will avoid using for the next time. With a literature review, this thesis aims to survey the role of testing in software development and future trends for software testing. Most common models for both software development and software testing are presented in this thesis. Also ideas about the role of testing and development trends of testing found from the literature are presented. The findings of the thesis are that the importance of testing is essential in software development both the costs and the properly working software point of view. As a future development trends presented in this thesis are the significance of developing test automation as well as four future trends which are universal test theory, automatic testing, development environments and test-based modelling.

Keywords: testing, software testing, test automation, software development

SISÄLLYS

1 JOHDANTO

Ohjelmistosuunnittelu ja ohjelmistotuotanto ovat nykypäivänä kaikkialla ja koskettavat meitä jokaista tavalla tai toisella. Hyvin suunniteltuja ohjelmistoja ei edes huomaa, käyttö sujuu niin jouhevasti, kun taas huonosti suunnitellut tai huonosti testatut ohjelmistot suututtavat ja hermostuttavat. Vaikka ohjelmistokehitystä on tehty kauan, syntyy edelleen ohjelmistoja, jotka ovat julkaisuvaiheessa virheellisiä tai toimimattomia. Sommervillen (2016) mukaan suurin osa näistä epäonnistumisista johtuu kahdesta syystä; 1. järjestelmät ovat kasvaneet niin monimutkaisiksi ja kehitysvauhti on ollut niin nopeaa, että entiset suunnittelutekniikat ja -metodit eivät enää vastaa vaatimuksiin. 2. ohjelmistoja kehitettäessä ei käytetä tutkittuja metodeja, mikä johtaa kalliimpiin ja epävarmempiin ohjelmistoihin.

Testaaminen on osa ohjelmistojen verifikaatio- ja validointiprosessia (Sommerville, 2016). Ohjelmistojen testaaminen ennen niiden liikkeellelaskua sekä päivitysten yhteydessä on olennaisen tärkeää ohjelmistojen toimintavarmuuden ja virheettömyyden kannalta. Testauksessa pyritään löytämään mahdollisia ohjelmistovirheitä ja tutkitaan, vastaako ohjelmisto käyttäjän syötteisiin halutulla ja odotetulla tavalla. Lisäksi tutkitaan mm. ohjelmiston toiminta-aikaa, käytettävyyttä ja toimintaa eri ympäristöissä. Sommervillen (2016) mukaan testaus on kuitenkin aina rajallista ja testaus ei koskaan voi todistaa, että ohjelmisto on virheetön. Kuten Edsger Dijkstra (1972) on todennut, testaus voi ainoastaan näyttää virheiden olemassaolon, ei niiden puuttumista.

Tämä tutkielma on toteutettu systemaattisena kirjallisuuskatsauksena. Kirjallisuutta on haettu Google Scholar, IEEE Xplore ja JYKDOK palveluista pääasiallisina hakusanoina testing, software development, testautomation, manual testing ja development process. Hakutuloksia valittaessa pyrittiin kiinnittämään huomiota julkaisuvuoteen ja tehtyjen viittausten määrään, lisäksi kirjallisuuden relevanttius tutkittavan aiheen kannalta rajasi kirjallisuutta. Kirjallisuuskatsaus pyrkii vastaamaan seuraaviin kysymyksiin:

1) *Millaisessa roolissa testaaminen on ohjelmistokehityksessä?*
2) *Mikä on testaamisen tulevaisuus ja kehityssuunnat?*

Kirjallisuuskatsauksessa käydään aluksi läpi ohjelmistokehitys ja ohjelmistosuunnittelu pääpiirteittäin; mitä vaiheita suunnittelussa ja kehityksessä on tai olisi oltava onnistuneen järjestelmän luomiseksi. Erilaiset ohjelmistokehityksen menetelmät käydään katsauksessa läpi samoin pääpiirteittäin, noudatellen Sommervillen (2016) kirjassaan esittämää kolmijakoa.

Seuraavassa luvussa, Ohjelmistotestaus, pyritään käymään läpi eri ohjelmiston kehitysvaiheiden testausta ja eri tyyppisiä testimetodeja. Näistä molemmista on myös otettu mukaan yleisimmin kirjallisuudessa esiintyvät menetelmät. Neljännessä luvussa Testaamisen rooli ja tulevaisuuden suuntaviivoja pyritään paitsi taustoittamaan tutkimuskysymyksiä, myös vastaamaan niihin tuomalla esille testaamisen roolia ja merkitystä sekä ajatuksia siitä, mihin suuntaan testaaminen on menossa tulevaisuudessa. Lopuksi yhteenvedossa tuodaan katsauksen ajatukset yhteen.

2 OHJELMISTOKEHITYS

Testaamista tehdään osana ohjelmistokehitystä, minkä vuoksi testaamisen merkityksen ymmärtämiseksi on olennaista hahmottaa myös ohjelmistojen kehitysprosessi. Tässä luvussa esitellään pääpiirteittäin ohjelmistokehityksen eri vaiheita.

Ohjelmisto (software) on Sommervillen (2016, s 19-21) määritelmän mukaan tietokoneohjelma sekä siihen kuuluvat dokumentaatiot, kirjastot, tukisivustot ja konfiguraatiotiedot. Yksittäinen, omaan käyttöön tehty koodi ei siis tämän määritelmän mukaan ole ohjelmisto, samoin kuin ohjelmisto ei ole sama kuin tietokoneohjelma.

Ohjelmistokehityksen elinkaarimalli sisältää Sheten ja Jadhavin (2014) mukaan seuraavat vaiheet; vaatimusmäärittely, suunnittelu ja analyysi, kehitys, testaus ja ylläpito. Testaus siis sisältyy olennaisena osana ohjelmistokehitykseen.

Sommerville (2016, s. 45-53) jakaa kirjassaan ohjelmistokehityksen prosessit kolmeen eri malliin: vesiputousmalliin, inkrementaaliseen malliin sekä integraatio- ja konfiguraatiomalliin. Tätä jakoa noudatetaan myös tässä katsauksessa. Näiden lisäksi esitellään alla myös testivetoinen kehittämismalli, joka on tavallaan käänteinen tapa kehittää ohjelmistoja.

2.1 Vesiputousmalli

Vesiputousmallissa kehitysvaiheet seuraavat toinen toistaan järjestyksessä. Nämä vaiheet suunnitellaan ja aikataulutetaan etukäteen ja ne ovat

1. Vaatimusanalyysi ja -määrittely. Kehitettävälle järjestelmälle asetut odotukset ja vaatimukset listataan ja dokumentoidaan yksityiskohtaisesti.
2. Järjestelmän ja ohjelmiston suunnittelu. Tässä vaiheessa vaatimusmäärittelyn perusteella suunnitellaan kehitettävä järjestelmä. Järjestelmän arkkitehtuurista esitetään yleiskuva.

3. Toteutus ja yksikkötestaus. Seuraavassa vaiheessa suunnitelma toteutetaan koodaamalla ja testataan jokainen järjestelmän yksikkö erikseen toiminnan varmistamiseksi. Samalla varmistetaan vaatimusmäärittelyssä asetettujen ehtojen toteutuminen yksikkötasolla.
4. Integraatio ja systeemitestaus. Tässä vaiheessa edellä mainitut yksittäiset ohjelmat integroidaan yhteen. Testaus tehdään koko järjestelmälle sekä toiminnallisuuden että vaatimusmäärittelyn vaatimusten kannalta. Vaiheen lopuksi uusi järjestelmä luovutetaan asiakkaalle.
5. Käyttöönotto- ja ylläpitovaihe. Tämä on pitkäkestoisin vaihe järjestelmän elinkaarella. Tässä vaiheessa kehitetty järjestelmä otetaan käyttöön. Ajan myötä esiin tulevia mahdollisia virheitä korjataan ja järjestelmää kehitetään syntyvien uusien vaatimusten ja tarpeiden myötä koko järjestelmän elinkaaren ajan. (Sommerville, 2016, s. 47-49)

Myös Jainin (2018) mukaan perinteisessä ohjelmistokehityksessä kehitysvaiheet seuraavat toisiaan tarkassa järjestyksessä ja toinen vaihe alkaa siitä mihin edellinen päättyy. Kehittäminen noudattaa alussa tehtyjä määrittelyjä ja suunnitelmia. Tämä Jainin määritelmä vastaa Sommervillenkin esittämää vesiputousmallia.

Vesiputousmallissa jokainen yllä esitetyistä vaiheista dokumentoidaan ja hyväksytetään. Dokumentoinnin vaatimus tekee kehitysprosessista raskaan, kalliin ja hitaan – mikäli asiakas haluaa muuttaa kesken kehityksen jotain ohjelmiston toiminnallisuuksista, vaatii se uudelleen dokumentoinnin ja hyväksymisprosessin läpikäynnin. Nopeat muutokset eivät vesiputousmallin mukaisessa kehityksessä siis onnistu, mikä tekee siitä soveltuvan etenkin laajojen, kompleksisten ja pitkäikäisten kehityshankkeiden kehitysmalliksi. Erityisesti vesiputousmalli soveltuu prosessimalliksi, mikäli kehitettävän ohjelmiston täytyy kommunikoida ja muodostaa rajapinta aikaisemman kovakoodatun systeemin kanssa tai mikäli kehitettävä ohjelmisto on kriittinen erityisesti turvallisuuden vuoksi tai ohjelmisto on osa laajempaa kokonaisuutta, jossa on mukana useita yhteistyökumppaneita. (Sommerville, 2016, s. 47-49)

2.2 Inkrementaalinen kehitysmalli

Inkrementaalisessa ohjelmistokehitysprosessissa spesifikaatio, kehitys ja validaatio kulkevat lomittain. Heti ensimmäisestä kehitetystä sovelluksesta, prototyypistä, haetaan palaute käyttäjiltä ja saadun palautteen perusteella sovellusta kehitetään edelleen. Versioita ohjelmistosta voi siis olla useita, kunnes lopulta päädytään viimeiseen, lopulliseen versioon. Testausta tässä prosessimallissa tehdään jatkuvasti, koko ohjelmiston kehityksen ajan, sekä kehittäjien toimesta että asiakkaan toimesta. (Sommerville, 2016, s. 49-51)

Inkrementaalinen lähestymistapa prosesseihin ja suunnitteluun kuvastaa normaalia ongelmanratkaisua, jossa edetään vaiheittain, askel askeleelta kohti ratkaisua. Sen vahvuudet verrattuna vesiputousmalliin ovat 1. kustannustehokkuus; vaiheittaisessa etenemisessä ei esim. tehdä ylimääräisiä ja jatkossa

turhiksi osoittautuvia dokumentteja, 2. asiakaspalautteen saaminen alusta saakka sekä 3. nopea käyttöönotto – asiakas saa jo varhaisessa vaiheessa ohjelmiston ensimmäisen version käyttöönsä. (Sommerville, 2016, s. 49-51)

Jainin, Ahujan ja Sharman (2016) mukaan inkrementaalisiin ohjelmistokehitysmalleihin kuuluu useita hieman toisistaan poikkeavia malleja, kuten Agile, Scrum ja XP (Extreme Programming). Näiden kaikkien kehitysmallien olennaisena osana ja yhteisenä tekijänä on kommunikaatio ja yhteistyö kehitystiimien kesken. Inkrementaalisissa kehitysmalleissa ohjelmistokehitys on nopeaa ja tapahtuu sykleissä, testaamisen kulkiessa jatkuvasti kehityksen rinnalla.

2.3 Integraatio ja konfiguraatio -kehitysmalli

Ohjelmistojen uudelleenkäyttö on monissa ohjelmistoprojekteissa yleistä. Uudelleenkäyttöä voi tapahtua muun kehityksen ohessa, informaalisti ja pieniltä osin, tai uudelleen käyttöön nojautuen voidaan rakentaa kokonaisia ohjelmistoja. Tämä integraatio ja konfiguraatio lähestymistapa ohjelmistokehitysprosessiin perustuu kokonaan uudelleenkäytettäviin ohjelmistokomponentteihin ja niiden integraatioon. (Sommerville, 2016, s. 52-54)

Integraatio ja konfiguraatio -mallin vaiheet ovat

1. Vaatimusmäärittely. Vaatimusmäärittelyvaiheessa alustavat toiveet ja vaatimukset ohjelmistolle esitetään. Tässä vaiheessa vaatimusten ei tarvitse olla vielä yksityiskohtaisia.
2. Ohjelmistojen löytäminen ja arviointi. Tässä vaiheessa vaatimusmäärittelyn perusteella etsitään ja arvioidaan komponentit ja ohjelmistot, jotka vastaavat vaatimuksiin ja tarjoavat halutun toiminnallisuuden.
3. Vaatimusten uudelleenmäärittely. Uudelleenmäärittelyvaiheessa alkuperäisiä ohjelmistolle asetettuja vaatimuksia tarkastellaan uudelleen löydettyjen komponenttien ja ohjelmistojen valossa. Tässä vaiheessa vaatimusmäärittelyn vaatimuksia muokataan niin, että vaatimukset kohtaavat olemassa olevien komponenttien tarjoamien mahdollisuuksien kanssa.
4. Mikäli sopiva applikaatio löytyy valmiina, se konfiguroidaan sopivaksi.
5. Mikäli valmista applikaatiota ei löydy, löydetyt komponentit adaptoidaan, kehitetään mahdollisesti uusia komponentteja ja lopuksi integroidaan kaikki komponentit.

Tämän prosessimallin etuna on kehitettävän ohjelmiston vähentäminen ja sen myötä myös kustannusten ja riskien vähentäminen. Lisäksi prosessi säästää aikaa ja valmis tuote on asiakkaalla nopeasti. Huonona puolena tässä kehitysprosessimallissa on vaatimuksista tinkiminen; valmiiden komponenttien käytössä on väistämätöntä, että joistakin järjestelmälle asetetuista vaatimuksista joudutaan tinkimään. (Sommerville, 2016, s. 52-54)

2.4 Testivetoinen kehitys

Crispinin (2006) mukaan inkrementaalisiin Agile-metodeihin lukeutuva test-driven development (TDD), testivetoinen kehitys ei nimestään huolimatta ole testausmenetelmä vaan ohjelmistojen kehitysmenetelmä. Testivetoisessa kehityksessä ohjelmoija kirjoittaa jokaista toiminnallisuutta varten ensin yksikkötestin, jonka jälkeen sama ohjelmoija kirjoittaa koodin, joka läpäisee testin. Sommerville (2016, s. 242-244) puolestaan kuvaa testivetoista menetelmää ohjelmistokehityksen lähestymistapana, jossa kirjoitetaan rinnakkain sekä testiä että koodia. Kun uusi toiminnallisuus halutaan lisätä koodiin, kirjoitetaan ensin testi, joka tietysti ajettaessa antaa virheen. Tämän jälkeen kirjoitetaan itse koodi toiminnallisuudelle ja testataan uudestaan.

Tämän tavallaan käänteisen kehitysmenetelmän etuna on mm., että se pakottaa ohjelmoijan miettimään ohjelmoitavaa asiaa monelta eri kannalta. Käytännössä on todettu, että testivetoisella menetelmällä koodatussa ohjelmassa on vähemmän virheitä. Vaikkakin saattaa näyttää siltä, että koodaaminen vie enemmän aikaa, on valmis koodi niin paljon luotettavampaa, että aika säästyy siinä. (Crispin, 2006)

Sommervillen (2016, s. 242-244) mukaan testivetoisen kehityksen edut ovat koodin luotettavuus, joka tulee testauksen kautta, sekä dokumentaation muodostuminen samanaikaisesti testitapausten kanssa. Lisäksi hän mainitsee, että testivetoisen kehityksen menetelmä sopii parhaiten uusien toiminnallisuuksien kehittämiseen uusissa ohjelmistoissa. Suurten koodikomponenttien uudelleenkäyttöön testivetoinen kehitys ei Sommervillen mielestä taivu.

Karac ja Turhan (2018) puolestaan toteavat artikkelissaan, että testivetoisen menetelmän edut eivät ole aivan kiistattomat. Testivetoinen kehitys menetelmänä on vaativa; menetelmän oppiminen vie aikaa, ohjelmoijilla ei yleensä ole riittävästi testaajan taitoja menetelmään ja tuottavuus kärsii, koska testivetoinen kehitys on hitaampi tapa koodata. Kirjoittajien mielestä iso osa testivetoisen kehityksen oletettua paremmuutta on työskentely nopeissa, lyhyissä sykleissä, jolloin virheet saadaan heti kiinni ja työskentely on sujuvampaa.

3 OHJELMISTOTESTAUS

Ohjelmistotestaus on olennainen osa ohjelmistojen ja järjestelmien laadun varmistusta ja mittaamista. Kaikkia virheitä ei valitettavasti ole hyvälläkään testaamisella mahdollista löytää ja peruskysymykseksi testatessa nouseekin, mitä strategiaa noudatetaan. (Khan, 2010)

Jovanovicin (2006) mukaan testaus on prosessi, jossa ohjelmisto suoritetaan tavoitteena löytää virheitä. Samoin Sneha ja Malle (2017) määrittelevät ohjelmistotestauksen prosessiksi, jossa suoritetaan testattava ohjelma ja pyritään löytämään ohjelman virheet, niin että lopputuloksena olisi mahdollisimman virheetön ohjelmisto.

Testausta tehdään ohjelmiston laadun arvioimiseksi ja parantamiseksi ja testauksen tavoitteena on löytää virheitä mahdollisimman pienellä vaivalla ja mahdollisimman lyhyessä ajassa (Jovanovic, 2006). Myös Khanin ja Khanin (2012) mukaan ohjelmistotestauksen päätarkoitus on ohjelmiston laadun varmistus, luotettavuuden arviointi, validaatio ja verifikaatio – joko joku tai osa näistä tai kaikki nämä yhtä aikaa. Samoin Jainin ym. (2018) mukaan testaamisella varmistetaan ohjelmiston toiminnallisuus, luotettavuus, ylläpidettävyys, performanssi ja käytettävyys.

Ohjelmistokehityksessä kehitettävän ohjelmiston sekä validointi että verifikaatio ovat merkittävässä osassa. Validoinnilla mitataan, onko ohjelmisto oikea esitettyyn tarpeeseen nähden, täyttääkö se asiakkaan tarpeet ja odotukset, tekeekö ohjelmisto sen, mitä asiakas odottaa sen tekevän. Verifikaatio puolestaan mittaa, onko ohjelmisto rakennettu oikein eli vastaako se vaatimusmäärittelyssä asetettuihin sekä toiminnallisiin että ei-toiminnallisiin vaatimuksiin. Verifikaatiossa tutkitaan enemmän ohjelmiston teknisiä ominaisuuksia, kun taas validoinnissa tutkitaan, vastaako ohjelmisto asiakkaan odotuksiin. (Ammann ja Offutt, 2016, s. 8-9)

Testatessa siis testataan, että järjestelmä vastaa vaatimusmäärittelyn mukaisiin vaatimuksiin. Testauksen tarkoitus on paitsi näyttää, että ohjelmisto toimii niin kuin on tarkoitettu ja toisaalta tavoitteena on myös löytää virheitä

ennen ohjelmiston käyttöön ottoa. Testatessa yritetään toisin sanoen siis löytää syötteitä, jolla saatu vaste olisi virheellinen. (Sommerville 2016, s. 231-249)

Sommerville (2016, s. 231-249) jakaa tyypillisimmät testauksen vaiheet kehitystestaukseen, julkaisutestaukseen ja käyttäjätestaukseen. Kehitystestauksessa ohjelmistoa testataan pienemissä osissa jo kehitysvaiheessa ja testaajina ovat sekä suunnittelijat, testaajat että ohjelmoijat. Julkaisutestauksessa puolestaan erillinen testaustiimi testaa valmiin ohjelmiston ennen sen liikkeellelaskua, kun taas käyttäjätestauksessa käyttäjät tai potentiaaliset käyttäjät testaavat ohjelmistoa omissa ympäristöissään.

Khan (2010) puolestaan jakaa testaamisen neljään eri tyyppiseen testaukseen; oikeellisuustestaukseen, suoritustestaukseen, luotettavuustestaukseen ja turvallisuustestaukseen.

Käytännössä testausta tehdään usein sekä manuaalisesti että automaattisesti. Manuaalisessa testauksessa testaaja testaa ohjelmistoa tietyillä reunaehdoilla ja vertaa saatua tulosta odotettuun, kun taas testiautomaatiossa koodatulla testillä testataan ohjelmistoa, tarvittaessa useita kertoja ja aina uuden muutoksen jälkeen. (Sommerville 2006, s. 231-249)

Whittaker (2009) jakaa näistä kahdesta päätestaustavasta manuaalisen testauksen vielä kahteen kategoriaan; tutkivaan testaukseen ja koodilla testaamiseen, joista etenkään tutkivaa testausta ei hänen mukaansa voi korvata automaatiolla.

Seuraavaksi alla käydään läpi testausstrategioita sekä testausmetodeja kirjallisuuteen pohjautuen. Sekä strategioita että metodeja on alla esitettyjen lisäksi useita, mutta tässä katsauksessa keskitytään yleisimpiin kirjallisuudessa esitettyihin strategioihin ja metodeihin.

3.1 Testistrategiat

Alla olevassa otsikoinnissa on jaettu testauksen strategiat yksikkötestaukseen, integrointitestaukseen, järjestelmätestaukseen sekä hyväksymistestaukseen.

Jako noudattelee Snehan ja Mallen (2017) artikkelissaan esittämää mallia ja vastaavaa tasojakoa noudattaa kirjassaan myös Kasurinen (2013, s. 50-59). Sommervillen (2016, s. 231-249) esittämä kehitys-, julkaisu- ja käyttäjätestauksen vaihejako on sama asia laajemmin ilmaistuna; kehitystestauksen alle menevät sekä yksikkö-, integrointi- että testaus ja julkaisu- ja käyttäjätestauksen alle hyväksymistestaus.

3.1.1 Yksikkötestaus

Yksikkötestaus tarkoittaa nimensä mukaisesti testausta alimmalla tasolla, yksiköissä. Yksiköllä tarkoitetaan perusmoduulia eli pienintä mahdollista määrää koodia, joka on testattavissa. (Sneha ym., 2017)

Yksikkötestausta tehdään yleisesti kaikissa ohjelmistoprojekteissa ja se on yleisin testitapa. Yksikkötestauksessa yksittäisen osan, esim. moduulin tai funktion, toimintaa testataan toteutuksen yhteydessä. Testaajana toimii yleensä testattavan osan ohjelmoija. Yksikkötestauksen etuna on, että testaus tehdään heti, joten myös mahdolliset virheet saadaan selville ennen kuin ne menevät osaksi isompaa kokonaisuutta. Ongelmana yksikkötestauksessa puolestaan on saman asian kääntöpuoli; ohjelmisto toimii yleensä kokonaisuutena ja mahdollisesti osana vielä isompaa kokonaisuutta, jolloin yksittäinen moduuli ei välttämättä tee yksin mitään itsenäistä. Näihin tilanteisiin on testaamista helpottamaan mahdollista luoda järjestelmän osien liikennettä simuloivia komponentteja. (Kasurinen, 2013. s. 50-59)

3.1.2 Integrointitestaus

Integrointitestaus seuraa aina yksikkötestausta. Integrointitestauksessa testataan moduuleiden yhteenliittymää, moduuliryhmiä, sitä, kuinka moduulit toimivat, kun ne on integroitu keskenään. Tämä testaus voidaan tehdä joko ylhäältä alaspäin eli isoista kokonaisuuksista pienempiin tai alhaalta ylöspäin eli pienemmistä moduuleista isompiin kokonaisuuksiin. (Sneha ym., 2017)

Myös Kasurinen (2013) esittää nämä kaksi tapaa integrointitestaukselle. Alhaalta ylöspäin tapahtuvan testauksen etuna on se, että sillä löydetään alemman tason virheitä, jotka saattaisivat helposti jäädä huomaamatta. Ylhäältä alaspäin tapahtuva testaus puolestaan antaa yleensä testattavasta järjestelmästä hyvän yleiskuvan, mikä edesauttaa puuttuvien toimintojen havaitsemista. Kolmantena tapana integrointitestaukselle Kasurinen esittää voileipätestauksen, jossa tehtyä integraatiota testataan molemmista suunnista, ylhäältä ja alhaalta päin, samanaikaisesti. Voileipätestauksen lisäetu on lähinnä siinä, että vain testausta varten tehtyjen sijaismoduuleiden tarve on vähäisempi testauksen alussa. Toisaalta osien yhteensovittaminen voi tässä testimuodossa on haasteellista testaamisen loppuvaiheessa. (Kasurinen, 2013. s. 50-59)

3.1.3 Järjestelmätestaus

Järjestelmätestaus testaa koko ohjelmistoa tai järjestelmää kokonaisuutena. Testauksessa tarkastetaan vaatimusmäärittelyssä asetetut vaatimukset ja järjestelmän vastaavuus niihin sekä tutkitaan järjestelmän luotettavuutta, ylläpidettävyyttä ja turvallisuutta. (Sneha ym., 2017)

Järjestelmätestaus seuraa yksikkö- ja integrointitestausta ja siinä testataan toimivaa kokonaisjärjestelmää, ilman esim. integrointitestauksen sijaiskomponentteja. Järjestelmätestaus voi käsittää testitapausten suorittamista sekä black box, että white box -menetelmällä, tai se voi olla esim. kuormitustestausta tai käyttäjätestausta. Järjestelmätestausta tehdään aina testiympäristössä ja siinä etsitään edelleen virheitä myös yksittäisitä komponenteista. (Kasurinen, 2013. s. 50-59)

3.1.4 Hyväksymistestaus

Hyväksymistestaus suoritetaan siinä vaiheessa, kun järjestelmä luovutetaan asiakkaalle. Hyväksymistestauksen päätavoite on todentaa, että järjestelmä toimii ja vastaa asiakkaan tarpeisiin. Hyväksymistestauksessa ei niinkään enää etsitä ohjelmiston virheitä. (Sneha ym., 2017)

Hyväksymistestauksen päätavoitteena on todentaa, että järjestelmä on riittävän korkealaatuinen täyttämään vaatimusmäärittelyssä asetetut vaatimukset. Testauksessa painotetaan erityisesti järjestelmän toiminnallisuutta, yksittäisten komponenttien testaamisen sijaan ja testaaminen suoritetaan tavallisesti kohdeympäristössä. (Kasurinen, 2013, s. 50-59)

3.2 Testimenetelmät

Yllä esitettyjen testistrategioiden sisällä testausta tehdään monella tavalla. Jovanovicin (2006) mukaan testausmenetelmät voidaan jakaa kahteen kategoriaan; white box ja black box -testaukseen. Lisäksi viime vuosina on näiden tapojen rinnalle noussut myös grey box -testaus, joka on yhdistelmä kahdesta edellä mainitusta.

Kasurinen (2013, s. 64-70.) esittää kirjassaan testimenetelmien kahtia jaon staattiseen ja dynaamiseen testaamiseen. Staattisessa testaamisessa järjestelmää tutkitaan esim. koodiarvioinnin tai arkkitehtuurin näkökulmasta, sitä ei siis testata käyttämällä. Dynaamisessa testaamisessa puolestaan testattavaa järjestelmää testataan ja käytetään. Myös Kasurinen jakaa edellä mainitut dynaamiset menetelmät black box, white box ja grey box -testaamiseen.

3.2.1 Black box -testaus

Black box -testauksessa testataan ohjelmistoa vaatimusmäärittelyn pohjalta ilman tietoa ohjelmiston sisällöstä tai koodista. Black box eli musta laatikko onkin kuvaava nimitys tälle testimetodille. Black box –testaamisessa tutkitaan ohjelmiston toiminnallisuutta; antaako ohjelmisto odotetun vasteen tiettyyn toimintoon. Testauksen tuloksena voi löytyä esimerkiksi puuttuvia tai virheellisiä toimintoja tai toiminnallisuuksia, virheitä käyttöliittymässä tai virheellisiä vasteita tietyille syötteille. (Jovanovic, 2006)

Myös Khanin ym. (2012) mukaan black box -testauksessa testataan ainoastaan järjestelmän / ohjelmiston ulkoista toimintaa tietämättä mitään ohjelmiston sisällä tapahtuvasta toiminnasta. Testaajalla tulee olla tieto testattavan ohjelmiston tai järjestelmän arkkitehtuurista mutta ei varsinaisesta lähdekoodista.

Black box -testausta voidaan tehdä kaikissa järjestelmän testauksen vaiheissa, joissa on olemassa käynnistyvä versio järjestelmästä, eli käytännössä yksikkö-, integrointi-, järjestelmä- ja hyväksymistestitasoilla. (Kasurinen, 2013, s. 64-70)

Black box -testauksen etuina Khan ym. (2012) mainitsevat nopean testitapausten kehitysprosessin, testaajan havaintojen yksinkertaisuuden ja sen, että käyttäjänäkökulma on selkeästi erotettu kehittäjän näkökulmasta. Miinuspuolina puolestaan mainitaan rajoitettu testiskenaarioiden määrä. Rajoitetulla määrällä testiskenaarioita on luonnollisesti vain rajoitettu kattavuus. Lisäksi testaaminen voi olla helposti tehotonta. Black box -testausta kutsutaan myös toiminnalliseksi testaamiseksi, suljetun laatikon testaamiseksi tai käyttäytymistestaamiseksi. (Khan ym., 2012)

3.2.2 White box -testaus

White box -testauksessa testaaja tietää, millainen on testattavan ohjelmiston sisäinen rakenne. Testaajalla on siis tieto siitä, mitä esim. koodiin on kirjoitettu ja kuinka ohjelmiston eri komponentit toimivat keskenään. Testauksessa selvitetään, kuinka ohjelmisto suorittaa esimerkiksi erilaiset silmukat, toistettavat tapahtumat ja kuinka sen sisäinen rakenne vastaa validiteetti vaatimuksiin. (Jovanovic, 2006)

Khan ym. (2012) mukaan white box -testaus on ohjelmiston koodin sisäisen logiikan ja rakenteen yksityiskohtaista tutkimista – koko koodin ja ohjelmiston rakenteen tunteminen on siis välttämätöntä testauksen onnistumiseksi.

Kasurisen (2013, s. 64-70.) mukaan white box -testaus on black box -testauksen täydellisempi versio, siinä järjestelmälle annetaan syöte, seurataan syötteeseen reagointia ja samalla huomioidaan, mitä järjestelmän sisällä tapahtuu.

White box -testauksen etuina ovat erityisesti koodissa olevien virheiden havaitseminen ja testiskenaarioilla saavutettava laaja kattavuus. Miinuspuolena puolestaan Khan ym. (2012) mainitsevat white box -testaamisen korkean hinnan. White box -testausta kutsutaan myös rakennetestaamiseksi, lasilaatikko- tai avoin laatikko -testaamiseksi. (Khan ym., 2012)

3.2.3 Grey box -testaus

Grey box -testaus on kahden edellisen testitavan yhdistelmä. Grey box -testauksessa testaajalla on jonkin verran tietoa ohjelmiston rakenteesta tai koodista ja hän suunnittelee testitapaukset tämän tiedon pohjalta. Tätä metodia voidaan erityisesti hyödyntää silloin, kun testataan esim. kahden eri ohjelmoijan kirjoittamien ohjelmistojen integrointia. (Jovanovic, 2006)

Grey box -menetelmän lähtöajatuksena on yhdistää black box ja white box -testauksen parhaat puolet. Black box -menetelmästä tulevat vaatimusmäärittelyn perusteella tehdyt testitapaukset ja white box -menetelmästä puolestaan järjestelmän sisäpuolen tarkastelu. (Kasurinen, 2013, s. 64-70)

Myös Khanin ym. mukaan parhaimmillaan grey box -testaaminen yhdistää black box ja white box -testaamisen parhaat puolet. Grey box -testaamisessa testaaja suunnittelee testit käyttäjän näkökulmasta, kuitenkin niin, että tieto myös ohjelmiston sisällä olevasta rakenteesta on selvillä. Tämän vuoksi testiskenaariot ovat hyvin paikkansa pitäviä. Grey box -testaamisen heikkona puolena voidaan pitää testikattavuuden vähyyttä, mikä johtuu siitä, että koko lähdekoodi ei ole tiedossa. Lisäksi päällekkäisten testien mahdollisuus on suuri. (Khan ym., 2012)

4 TESTAAMISEN ROOLI JA TULEVAISUUDEN SUUNTAVIIVOJA

Tässä luvussa pyritään taustoittamaan ja vastaamaan kahteen alussa esitettyyn tutkimuskysymykseen, jotka ovat *Millaisessa roolissa testaaminen on ohjelmistokehityksessä?* ja *Mikä on testaamisen tulevaisuus ja kehityssuunnat?*

4.1 Testaamisen rooli ohjelmistokehityksessä

Testaamisen rooli ohjelmistokehityksessä on muuttunut ajan myötä. Gillensonin, Zhangin, Staffordin ja Shinin (2018) mukaan testaamisessa voidaan löytää 5 kehitysvaihetta alkaen debuggaukseen keskittyvästä testaamisesta aina nykypäivän testivetoiseen ohjelmistokehitys orientoituneeseen aikaan. 1990-luvun lopulla alkaneen testivetoisen ohjelmistokehityksen aikana testaaminen on kehittynyt ohjelmiston virheiden arvioinnista koko ohjelmistokehitykseen vaikuttavaksi prosessiksi. (Gillenson ym., 2018)

Normaalisti ohjelmistoprojektissa noin puolet kaikesta projektiin käytetystä ajasta ja puolet käytetyistä kustannuksista menee testaamiseen tai testaamisen valmisteluun liittyviin toimenpiteisiin. Ohjelmiston testaaminen ja sen varmistaminen, että ohjelmisto tekee sen, mitä on suunniteltu, on siis välttämätön osa kehitysprosessia. Ohjelmistoprojektin käytettävissä olevat resurssit, kuten aika, työ ja raha, ovat rajallisia ja nämä rajalliset resurssit on osattava kohdistaa oikein ja testauksen ollessa kyseessä, kriittisimpiin testitapauksiin. (Gillenson ym. 2018)

Myös Kasurinen (2013. s. 11-12) vahvistaa, että toimialasta ja teollisuudenalasta riippuen testaamisen osuus projektin kokonaiskustannuksista on 25-65% välillä, ollen yleisimmin noin puolet. Testaaminen on siis alasta riippumatta ohjelmistokehityksen rahallisesti arvokkain työvaihe.

Ohjelmistotuotanto ja ohjelmistokehitys ovat myös Milin ja Tchierin (2015) mukaan ainoat tuotannon alat, jossa tuotteen testaaminen on paitsi suuri tekninen ja organisatorinen haaste, myös merkittävä kustannustekijä. Ohjelmistotuotteet ovat yleensä niin monimutkaisia ja kooltaan suuria, että se tekee suunnittelusta haastavaa ja virheille altista. Toisena näkökulmana voidaan pitää sitä, että ohjelmistotuotannossa ei ole standardoituja prosesseja, joten sen sijaan että kontrolloitaisiin prosesseja, täytyy kontrolloida itse tuotetta. Lisäksi ohjelmistoalalta puuttuvat artikkelin kirjoittajien mukaan skaalattavat, käytännönläheiset metodit, jotka takaavat tuotteen laadun läpi kehitysprosessin. Näiden seikkojen vuoksi testaaminen on todella merkittävässä roolissa ohjelmistokehityksessä. (Mili ja Tchier, 2015)

Jain, Sharma ja Ahuja (2018) toteavat artikkelissaan, että laadukkaiden ohjelmistotuotteiden kehitys on välttämättömyys ohjelmistoalalle. Tämän vuoksi on tärkeää määritellä laadukkaan lopputuloksen takaava ohjelmistonkehitysprosessi. Sekä inkrementaalisessa eli Agile-tyyppisessä ohjelmistokehityksessä että perinteisessä vesiputousmallisessa ohjelmistokehityksessä on havaittavissa 5 vaihetta; vaatimusmäärittely, suunnittelu, toteutus, testaaminen ja käyttöönotto. Vesiputousmallissa nämä seuraavat toinen toistaan, kun taas inkrementaalisessa kehitysmallissa edellä mainitut kehitysvaiheet limittyvät ja toistuvat jokaisen inkrementin kohdalla. Onnistunut ja laadukas ohjelmisto edellyttää kaikkien näiden vaiheiden laadukasta, testattua läpivientiä. (Jain ym., 2018)

Sheten ja Jadhavin (2014) mukaan hyvin suunnitellulla ja toteutetulla testaamisella ohjelmiston virheiden määrä pienenee ja sitä kautta käyttäjätyytyväisyys kasvaa. Jainin ym. (2018) mukaan puolestaan ohjelmistotuote on laadukas, mikäli se on asiakkaan hyväksymä ja täyttää kaikki vaatimusmäärittelyssä asetetut vaatimukset. Mitä suurempi on asiakkaan tyytyväisyys tuotteeseen, sitä laadukkaampi tuote on, ja juuri testaamisella varmistetut toiminnallisuus, luotettavuus, ylläpidettävyys, performanssi ja käytettävyys ovat tässä avainasemassa. (Jain ym., 2018)

Laajoissa ohjelmistoprojekteissa käy usein niin, että aikataulut venyvät tai budjetti osoittautuu riittämättömäksi. Molemmat näistä haasteista saattavat vaikuttaa testaamisen määrän karsimiseen, mikä puolestaan saattaa näkyä lopputuotteen epävarmana toimintana. Mikäli jo projektin alussa kiinnitetään huomiota systemaattiseen testaamiseen, maksaa virheiden korjaaminen yleensä murto-osan siitä, että virhe löydetään vasta valmiista tuotteesta. (Kasurinen, 2013, s. 16-17)

Kuten edellä esitetyistä useista tutkimuksista voi päätellä, testaamisen rooli ohjelmistokehityksessä näyttää siis olevan varsin merkittävä. Testaaminen ja sen onnistuminen on olennaista ja jopa kriittistä kehitettävän ohjelmiston laadun ja toiminnan kannalta. Lisäksi testaaminen sitoo sekä projektin aikaa että pääomaa, minkä vuoksi se kannattaa suunnitella ja toteuttaa huolellisesti ja laadukkaasti.

4.2 Testaamisen tulevaisuus

Kuten edellä on useaan kertaan jo todettu, testausta tehdään ohjelmistojen laadun ja toiminnallisuuden varmistamiseksi. Wangin (2018a) mukaan ohjelmistot ja järjestelmät tulevat jatkossa entisestään kehittymään ja monimutkaistumaan ja ketterät inkrementaaliset kehitystavat, kuten Agile, tulevat lisääntymään. Myös Gillensonin ym. (2018) mukaan ohjelmistokehityksessä on siirrytty perinteisestä vesiputouskehitysmallista, jossa testaamista tehdään pääasiassa ohjelmistokehityksen loppuvaiheessa, yhä enemmän kohti inkrementaalisia kehitysmalleja, kuten Scrum ja Agile, joissa testaamista tehdään jokaisessa kehitysvaiheessa, läpi ohjelmiston kehityksen. Testaamisen merkitys ohjelmistokehityksessä on siis vain kasvanut vuosien kuluessa.

Inkrementaalisessa, ketterässä kehityksessä etenkin testiautomaatio on noussut tärkeään rooliin, koska automaatiolla voidaan ajaa testejä kustannustehokkaasti ja toistuvasti. Testaamisen tehokkuuden ja kattavuuden lisäämiseksi testaamista on automatisoitava yhä enenevässä määrin. (Wang, 2018a)

Myös Akin, Sentürk ja Garousi (2018) totesivat tutkimuksessaan finanssialalta, että testaamisen automatisointi aikaisemman manuaalisen testauksen sijaan mahdollisti virheiden aikaisemman havaitsemisen sekä yksityiskohtaisemmat ja nopeammat testitulokset, jonka ansiosta myös asiakastyytyväisyys kasvoi. Lisäksi testaamisen automatisoinnin ansiosta tutkimuksessa olevalta yritykseltä säästyi jopa 48 työtuntia kuukaudessa, mikä on merkittävä resurssisäästö.

Testiautomaatiotakin on jatkuvasti kehitettävä, jotta automaation hyödyt, mm. tehokkuus säilytetään. Organisaatioiden on tunnistettava, missä kypsyys- ja kehitysvaiheessa yrityksen testiautomaatio on milläkin hetkellä ja miten automaatiota voisi viedä eteenpäin. Tämän arvioimiseen on kehitetty erilaisia arviointimalleja, joista tunnetuimmat ovat TMMI eli Test Maturity Model integration ja TMM eli Test Maturity Model. (Wang 2018a)

Garousi, Felderer ja Hacaloğlu (2018) tunnistavat omassa tutkimuksessaan saman yllämainitun haasteen; liian usein testaamista ja testiautomaatiota tehdään ja kehitetään ad hoc -periaatteella, mikä johtaa monesti aikatauluhaasteisiin ja siihen, ettei testaus toimi halutulla tavalla. Tutkijat löysivät omassa tutkimuksessaan peräti 58 erilaista testiautomaation kypsyyden arviointimenetelmää, joiden avulla yrityksissä voitaisiin arvioida ja kehittää testiautomaation prosesseja parempaan ja luotettavampaan suuntaan.

Orso ja Rothermel (2014) listaavat artikkelissaan tulevaisuuden ja varmasti tämän hetkenkin testaamisen haasteita. Näistä esimerkkeinä ei-toiminnallisten ominaisuuksien testaaminen – miten ohjelmiston performanssia voidaan testata. Tämä koskee etenkin web applikaatioita, joissa esim. vasteaika voi olla avainasemassa käyttäjätyytyväisyydessä. Toisena testausta tarvitsevana ei-toiminnallisena ominaisuutena Orso ym. mainitsee ohjelmiston energiatehokkuuden.

Testaamisen tulevaisuutta on tarkasteltu ja kartoitettu useasta eri näkökulmasta. On tutkittu mm. sitä, mitä teknologiaa testaaminen tulee jatkossa hyödyntämään ja vaihtoehtoina on esitetty tekoälyä, kasvavaa laskentatehoa ja erilaisia ohjelmistokehitystyökaluja. Bertolino (2007) on artikkelissaan esittänyt neljä innovaatiota testausalan tulevaisuudentavoitteiksi. Nämä testaamisen tulevaisuudentavoitteet, unelmat Bertolinon mukaan, ovat:

1. Universaali testausteoria. Yleinen testausteoria olisi testauksen arvioinnin ja kehittämisen kehikko, jonka avulla testausta voitaisiin arvioida ja kehittää eteenpäin testaushankkeesta riippumatta.
2. Täysin automatisoitu testaaminen. Tällä tarkoitetaan kehittynyttä tekoälyä hyödyntävää testauslaitetta, joka testaisi täysin ilman opastusta.
3. Tehokkuuden maksimoivat kehitysympäristöt. Esimerkkeinä kehitysympäristöistä voisivat olla esim. erilaiset itsenäisesti korjausehdotuksia antavat työkalut.
4. Testipohjainen ohjelmistorakenne. Tässä tavoitteessa testattavuus toimisi lähtökohtana ohjelmistojen kehitystyölle ja ohjelmistot rakennettaisiin testivarmoista komponenteista.

Bertolino on esittänyt tulevaisuudentavoitteensa jo vuonna 2007 ja vielä edelleen nämä tavoitteet ovat lyhyellä aikavälillä epärealistisia toteuttaa etenkin laajassa mittakaavassa. Tulevaisuudentavoitteista tehokkuuden maksimoiva kehitysympäristö on jo kuitenkin lupaavassa vauhdissa kohti ratkaisuaan tekoälytutkimuksen sovellusten ja virheiden yleisimmän sijainnin tutkimustiedon ansiosta. Samoin testipohjainen ohjelmistorakenne on mahdollista saada toimimaan oliosuunnittelun malleja, kuvauskieliä ja simulaattoreita hyödyntämällä. Testivetoinen ohjelmistokehitys hyödyntää jo osaa testipohjaisen ohjelmistorakenteen perusperiaatteista. (Kasurinen, 2013, s. 186-187)

Sen sijaan universaalin testausteorian, samoin kuin täysin automatisoidun testaamisen toteutuminen on vielä kauempana tulevaisuudessa. Testaaminen on ihmisten tekemää toimintaa, minkä vuoksi kaikkiin olosuhteisiin sovellettava malli tulisi olemaan liian yleisluontoinen, että siitä olisi mitään käytännön hyötyä. Sen sijaan useimpiin projekteihin ja organisaatioihin sopiva yleishyödyllinen testaamisen toimintamalli on mahdollisesti saatavilla jo lyhyemmän ajan sisällä. Täysin automatisoitu testaaminen on varmasti haasteellisin tulevaisuudentavoitteista. Itsenäinen testiautomaatio vaatisi todella kehittyneen tekoälyn, joka olisi lähellä ihmisen simulaatiota ja kysyä voikin, onko sellaisen tekoälyn kehittäminen testaamista varten järkevää. (Kasurinen, 2013, s. 186-187)

Kasurinen (2013, s. 188) pitää realistisena tulevaisuuden näkymänä, että testaamisen arvostus tulee kasvamaan entisestään. Laadukkaan ohjelmiston tärkein perusta on huolellinen suunnittelu ja kehitys, ja testaus on olennainen osa lopullisen ohjelmiston laatua. Vaikka testaamalla ei voida korjata huonon ja väärin suunnitellun ohjelman ongelmia, menee kaikki ohjelmiston suunnittelutyö hukkaan, jos sitä ei testata kunnolla.

Testaamisen ja testiautomaation jatkuva kehittäminen sekä Bertolinon (2007) esittämien tulevaisuuden unelmien määrittämien suuntaviivojen viemi-

nen eteenpäin määrittävät varmasti tulevaisuudessa testaamisen kehittymistä ja sen roolin vahvistumista.

5 YHTEENVETO

Tässä tutkielmassa tavoitteena oli kartoittaa testaamisen merkitystä ohjelmistokehityksessä sekä samalla löytää mahdollisia tulevaisuuden suuntaviivoja siitä, mihin testaaminen on kehittymässä. Tutkielma toteutettiin kirjallisuuskatsauksena.

Ohjelmistokehityksessä on löydettävissä useita eri lähestymistapoja ja menetelmiä. Tässä tutkielmassa on esitelty kehitysmenetelmistä yleisimmät kirjallisuudessa esiintyvät menetelmät. Perinteisimmässä vesiputousmenetelmässä kehitysvaihteet seuraavat toisiaan ennalta määrätyssä järjestyksessä, edeten vaihteesta toiseen. Testaaminen tapahtuu tässä kehitysmallissa jokaisessa vaiheessa määrätyn mallin mukaan. Erityisesti dokumentointivaatimus tekee vesiputousmallista suhteellisen raskaan kehitysmallin, joka soveltuu pääasiassa isojen kokonaisuuksien suunnitteluun ja prosessimalliksi. Vesiputousmallin rinnalle ovat nousseet inkrementaaliset kehitysmallit, joista mainitaan Scrum ja Agile esimerkkeinä. Inkrementaalinen ohjelmistokehitys perustuu spesifikaation, kehityksen ja validaation lomittaiseen ja rinnakkaiseen tekemiseen. Kehitettävästä ohjelmistosta voi olla matkan varrella useita eri versioita, joista lopuksi valitaan paras. Testaamista inkrementaalisessa kehitysprosessissa tehdään jatkuvasti sekä asiakkaan että kehittäjien toimesta.

Kahden yllämainitun kehitysmenetelmän lisäksi tutkielmassa esitellään testivetoinen ohjelmistokehitys, joka perustuu käänteiseen tapaan kirjoittaa ohjelmistoja. Testivetoisessa menetelmässä kirjoitetaan ensin testi, jonka jälkeen vasta koodi tai ohjelma, joka läpäisee kirjoitetun testin.

Testaamisen menetelmiä ja strategioita käydään tutkielmassa läpi kirjallisuudessa yleisimmin esiintyvien mallien mukaisesti. Testistrategioista esitellään yksikkötestaus, joka on pienimmän mahdollisen yksikön, komponentin testaamista, integrointitestaus, jossa testataan yksittäisten komponenttien integroinnin onnistumista, järjestelmätestaus, jossa testataan kokonaisen järjestelmän toimivuutta sekä hyväksymistestaus, joka suoritetaan yleensä jo asiakkaan loppukäyttöympäristössä ja asiakkaan toimesta.

Testimenetelmistä puolestaan tutkielmassa käydään läpi black box, white box ja grey box testaaminen. Black box -testaamisessa testaajalla ei ole tietoa ohjelmiston sisällöstä tai koodista, vaan testaaminen tehdään ainoastaan vaatimusmäärittelyssä asetettujen vaatimusten pohjalta, tutkitaan, tekeekö ohjelmisto sen, mitä sen on ajateltu tekevän. White box -testaamisessa puolestaan testaaja tuntee myös ohjelmiston sisällön ja tutkii myös koodin toimintaa, kuinka esim. silmukat suoritetaan. Grey box -testaaminen on nimensä mukaisesti edellä mainittujen yhdistelmä, eli testaajalla on tietoa sekä ohjelmiston sisällöstä että vaatimusmäärittelyssä asetetuista vaatimuksista.

Tutkielmaa varten läpikäytyjen artikkeleiden perusteella testaamisen roolia ja merkitystä ohjelmistokehityksessä ei voi korostaa liikaa. Onnistunut ja toimiva ohjelmisto vaatii aina lukuisia testauskertoja ennen julkaisua ja testaamisen voidaan katsoa onnistuneen silloin, kun virheitä löytyy. Mikäli kehitettävässä ohjelmistossa ei havaita virheitä, voi pohtia, ovatko testitapaukset olleet riittävän monipuolisia ja kattavia ja pitäisikö testaamista jatkaa. Ohjelmistokehitysprojektin kustannuksista noin puolet aiheutuu testaamisesta, joten testaamisen huolellinen suunnittelu sekä testitapausten analysointi ja kattavuus ovat merkittävässä osassa myös kokonaisprojektin onnistumista.

Ohjelmistotestaamisen tulevaisuuden haasteita ja näkymiä nousee läpikäytyjen artikkeleiden perusteella selkeästi esiin. Testiautomaatiolla saadaan katettua kustannustehokkaasti ja nopeasti laajojakin testattavia osa-alueita, minkä vuoksi testiautomaation jatkuva kehittäminen ja olemassa olevan automaation kriittinen arviointi ovat olennaisessa roolissa siinä, että testaamisesta on tulevaisuudessa entistä enemmän hyötyä. Ohjelmistoyritysten tulisikin arvioida kriittisesti oman testiautomaationsa kypsyyttä ja siitä, miten sitä olisi vietävissä eteenpäin.

Tulevaisuuden testaamisen kehityssuuntia tässä tutkielmassa esitetään neljä. Kaksi näistä, universaali testausteoria ja täysin automatisoitu testaaminen, ovat vielä joidenkin vuosien päässä tulevaisuudessa, mutta sekä erilaiset kehitysympäristöt, jotka maksimoivat tehokkuutta tekoälyn avulla, että testipohjainen ohjelmistokehitys ovat lähiaikoina mahdollisesti täysin saavutettavissa ja osin jo käytössäkin. Testaamisen arvostus tullee kasvamaan myös tulevaisuudessa. Testaamalla ei voida korjata huonoa ohjelmistoa, mutta hyvästäkin ohjelmistosta voi tulla epävarma, mikäli sitä ei testata kunnolla.

Kandidaatintutkielman laajuuden vuoksi tässä tutkimuksessa päädyttiin käsittelemään testaamista yleisimmin kirjallisuudessa esiintyvien menetelmien ja strategioiden valossa. Jatkotutkimusta olisi mielenkiintoista tehdä lisää erityisesti testaamisen tulevaisuuden haasteista ja näkymistä.

LÄHTEET

Ammann, P., & Offutt, J. (2016). Introduction to software testing. Cambridge University Press.

Akin, A., Sentürk, S. & Garousi, V. (2018). Transitioning from manual to automated software regression testing: Experience from the banking domain doi:10.1109/APSEC.2018.00074

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams doi:10.1109/FOSE.2007.25

Crispin, L. (2006). Driving software quality: How test-driven development impacts software quality doi:10.1109/MS.2006.157

Dijkstra, E. W. (1972). The humble programmer.

Garousi, V., Felderer, M., & Hacaloğlu, T. (2018). What we know about software test maturity and test process improvement doi:10.1109/MS.2017.4541043

Gillenson, M. L., Zhang, X., Stafford, T. F. & Shi, Y. (2018). A literature review of software test cases and future research doi:10.1109/ISSREW.2018.00015

Jain, P., Ahuja, L. & Sharma, A. (2016). Current state of the research in agile quality development

Jain, P., Sharma, A. & Ahuja, L. (2018). The impact of agile software development process on the quality of software product doi:10.1109/ICRITO.2018.8748529

Jovanović, I. (2006). Software testing methods and techniques. The IPSI BgD Transactions on Internet Research, 30.

Karac, I. & Turhan, B. (2018). What do we (really) know about test-driven development? doi:10.1109/MS.2018.2801554

Kasurinen, J. (2013). Ohjelmistotestauksen käsikirja. Docendo, Jyväskylä: Saarijärven Offset Oy.

Khan, M. E. (2010). Different forms of software testing techniques for finding errors. International Journal of Computer Science Issues (IJCSI), 7(3), 24.

Khan, M. E., & Khan, F. (2012). A comparative study of white box, black box and grey box testing techniques. Int. J. Adv. Comput. Sci. Appl, 3(6).

Mili, A., & Tchier, F. (2015) Software testing : Concepts and operations, John Wiley & Sons.

Orso, A., & Rothermel, G. (2014, Toukokuu). Software testing: a research travelogue (2000–2014). In Proceedings of the on Future of Software Engineering (s. 117-132). ACM.

Shete, N. & Jadhav, A. (2014). An empirical study of test cases in software testing doi:10.1109/ICICES.2014.7033883

Sneha, K. & Malle, G. M. (2017). Research on software testing techniques and software automation testing tools doi:10.1109/ICECDS.2017.8389562

Sommerville, I. (2016). Software engineering (Tenth edition, global edition.). Boston: Pearson.

Wang, Y. (2018a). Test automation maturity assessment doi:10.1109/ICST.2018.00052

Whittaker, J.A. (2009) Exploratory Software Testing : Tips, Tricks, Tours, and Techniques to Guide Test Design, Pearson Eduation