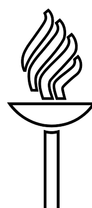
Juuso Perälä

TESTIVETOISEN OHJELMISTOKEHITYKSEN EDUT JA HAITAT OHJELMISTOPROJEKTEISSA



JYVÄSKYLÄN YLIOPISTO
INFORMAATIOTEKNOLOGIAN TIEDEKUNTA
2018

TIIVISTELMÄ

Perälä, Juuso
Testivetoisen ohjelmistokehityksen edut ja haitat ohjelmistoprojekteissa
Jyväskylä: Jyväskylän yliopisto, 2018, 25 s.
Tietojärjestelmätiede, kandidaatintutkielma
Ohjaaja: Pirhonen, Maritta

Tässä tutkielmassa on tarkoitus selvittää testivetoisella ohjelmistokehityksellä mahdollisesti saavutettavia etuja ohjelmistoprojekteissa. Tutkimusmenetelmänä on kirjallisuuskatsaus. Aluksi tutkielmassa käsitellään vesiputousmallia, ketteriä menetelmiä, ohjelmistotestausta ja ohjelmiston laadun käsitettä, jotta sen jälkeinen testivetoisen ohjelmistokehityksen tutkimukseen tutustuminen olisi selkeämpää. Testivetoinen ohjelmistokehitys, tai TDD, on suosittu ohjelmiston suunnittelumenetelmä, joka perustuu lyhyisiin iteraatioihin, joissa testit kirjoitetaan aina ennen varsinaista ohjelmakoodia, tai testattavaa toiminnallisuutta. Tutkielmassa saavutetut tulokset viittaavat siihen, että testivetoisen ohjelmistokehityksen implementoinnilla saavutettavia etuja voivat olla mm.: ohjelmiston ulkoisen laadun paraneminen, virheiden väheneminen, koodikattavuuden paraneminen, koodin ja vaatimusten parempi ymmärtäminen, kompleksisuuden paraneminen. Joitain haittoja, jotka saattavat myös hankaloittaa TDD:n implementointia, olivat: testivetoisen ohjelmistokehityksen implementoimiseen vaadittava aika ja vaiva, testivetoisen ohjelmistokehityksen kehittäjiltä vaatima korkeampi taitokynnys, mahdollisesta TDD:n prosessin väärinymmärtämisestä aiheutuvat ongelmat ja mahdollinen joidenkin koodin osien heikkeneminen. Tuloksissa havaittiin myös epäselvyyttä ja ristiriitaisuutta, joka viittaa lisätutkimuksen tarpeeseen.

Asiasanat: testivetoinen ohjelmistokehitys, test-first, TDD, test-driven development, tietojärjestelmätiede, ohjelmistosuunnittelu

ABSTRACT

Perälä, Juuso
Benefits and drawbacks of test-driven development in software projects
Jyväskylä: University of Jyväskylä, 2018, 25 pp.
Information Systems Science, bachelors thesis
Supervisor: Pirhonen, Maritta

The purpose of this thesis is to find out the benefits that may be gained in software projects by implementing test-driven development. The research method used in this thesis is literature review. Initially, the paper deals with the waterfall model, agile methods, software testing, and the concept of software quality to make it easier to understand when it later starts to review the research on test-driven development. Test-driven development, or TDD, is a popular software design method that is based on short iterations where writing test cases comes always before writing any functional code that is to be tested. The results obtained in this thesis suggest that the benefits of implementing test-driven development include: improvement of the software's external quality, reduction of errors, better code coverage, better understanding of the code and requirements, and better code complexity. Some disadvantages, which may also complicate the implementation of test-driven development were: time and effort required to implement test-driven development, higher skill threshold required by developers, possible problems caused by the misunderstanding of the TDD process and possible deterioration of some parts of the code. The results also revealed ambiguity and contradiction, which suggests to the need for further research.

Keywords: test-driven software development, test-first, TDD, test-driven development, information systems science, software design

KUVIOT

KUVIO 1 Testin suorittaminen. 9
KUVIO 2 Sisäinen ja ulkoinen laatu ISO 9126 12
KUVIO 3 Virheiden tiheys. 13
KUVIO 4 Testivetoinen ohjelmistokehitys.. 14

TAULUKOT

TAULUKKO 1 Testauksen päätasot kehitysprosessin aikana. 11
TAULUKKO 2 Koonti käsitellyistä tutkimuksista. 15

SISÄLLYS

TIIVISTELMÄ
ABSTRACT
KUVIOT
TAULUKOT
SISÄLLYS

1 JOHDANTO 6

2 OHJELMISTONKEHITYSPROSESSI 8
2.1 Vesiputousmalli ja ketterät menetelmät 8
2.2 Ohjelmistotestaus 9
2.3 Testausmenetelmät 10
2.4 Ohjelmiston laatu 12
2.5 Testivetoinen ohjelmistokehitys 13

3 TUTKIMUKSEN TULOKSET 15
3.1 Tutkimuksessa havaitut hyödyt 16
3.2 Tutkimuksessa havaitut haitat 17
3.3 Testivetoisen ohjelmistokehityksen vaikutus sisäiseen ja ulkoiseen laatuun 18
3.4 Tuloksien ristiriitaisuus 18

4 YHTEENVETO 20

LÄHTEET 22

1 JOHDANTO

Testivetoinen ohjelmistokehitys (eng. *test-driven development*), lyhennettynä TDD, on ohjelmiston suunnittelumenetelmä. Se tarkoittaa ohjelmiston iteratiivista ja inkrementaalista kehittämistä, jossa kehitettävänä olevaan toiminnallisuuteen kohdistuvat vaatimukset muutetaan erittäin spesifeiksi testitapauksiksi. Testitapaukset ovat, usein lyhyitä, yksittäisiin lähdekoodin osiin tai ohjelman moduuleihin keskittyviä testejä, joita tehdään samaan aikaan ohjelman kehitysprosessin kanssa. Aluksi kehitettävänä oleva toiminnallisuus ei pysty läpäisemään näitä testejä, koska sitä vastaavaa koodia ei ole vielä kirjoitettu. Vasta testien kirjoittamisen jälkeen kehitettävää toiminnallisuutta muokataan niin, että se läpäisee kyseiset testit. Prosessi on iteratiivinen ja sitä toistetaan läpi koko ohjelmiston kehitysprosessin. (Janzen & Saiedian, 2005; Beck, 2003, s. 9-10.) Testivetoinen ohjelmistokehitys eroaa siis perinteisestä vesiputousmallista, jossa ohjelmiston testaus tapahtuu vasta, kun varsinainen ohjelmakoodi on jo kirjoitettu.

Testivetoisella ohjelmistokehityksellä pyritään vastaamaan perinteisen vesiputousmallin mukaisen, test last development (TLD), ohjelmistokehityksen ongelmaan, jossa suunnitteluprosessissa tehtäviä virheitä ei havaita ajoissa jolloin tuotteen laatu, kustannusarvio sekä aikataulu voivat kärsiä. Testivetoisessa ohjelmistokehityksessä tähdätään yksinkertaisiin toteutuksiin, jossa kaikkien ohjelmiston osien pitää toiminnallisuudeltaan täyttää niille suunnitellut vaatimukset, mutta ei yhtään enempää. (Janzen & Saiedian, 2005.)

Testivetoisen ohjelmistokehityksen paremmuudesta perinteisiin malleihin verrattuina on tehty monia väitteitä. Näihin väitteisiin lukeutuvat mm.: testivetoisesti kehitettyjen ohjelmien kattavammat testit, ohjelmien sisäisen ja ulkoisen laadun paraneminen, ja kehityskustannuksien laskeminen vähäisemmästä virhemäärästä johtuen.

Tämän tutkielman tavoite on kirjallisuuskatsauksen menetelmää apuna käyttäen tutustua aiempaan testivetoisesta ohjelmistokehityksestä tehtyyn tutkimukseen, ja pyrkiä tunnistamaan mahdollisia todellisia etuja perinteisiin malleihin verrattuina. Tutkimuksessa käytetyt artikkelit haettiin JYDOKin ja Google Scholarin avulla. Tutkimusaineiston etsimiseen käytettyjä hakusanoja

olivat mm.: test-driven development, test-driven development benefits, test-driven development limitations and restrictions, test-driven development adoption, test-driven development adaptation, test-driven development industrial setting, test-driven development results, test-driven development empirical. Tutkimuskysymykset, joihin tutkielmassa pyritään vastaamaan ovat:

- Mitä etuja testivetoisella ohjelmistokehityksellä voidaan saavuttaa ohjelmistonkehitysprojekteissa perinteisiin, testaus kehityksen jälkeen, malleihin verrattuna?
- Mitä haittoja testivetoisen ohjelmistokehityksen käyttöönotolla voi olla ohjelmistonkehitysprojekteissa?

Toisessa luvussa esittelen vesiputousmallin ja ketterän ohjelmistokehityksen menetelmät, ohjelmistotestauksen konseptia, ohjelmiston laadun käsitettä, sekä testivetoista ohjelmistokehitystä. Kolmannessa luvussa perehdytään testivetoisen ohjelmistokehityksen piirissä tehtyyn tutkimukseen.

2 OHJELMISTONKEHITYSPROSESSI

Tässä luvussa perehdytään ohjelmistonkehitysprosessiin, sen erilaisiin menetelmiin ja erityisesti ohjelmiston testauksen konseptiin. Luvussa käsitellyillä asioilla pohjustetaan seuraavaa lukua, jossa käsitellään varsinaisen tutkimuksen tuloksia.

2.1 Vesiputousmalli ja ketterät menetelmät

Ohjelmistokehityksessä on ajan kuluessa erottunut kaksi toisistaan eroavaa menetelmää tuottaa ohjelmistoja: perinteinen vesiputousmalli ja uudempi ketterän ohjelmistokehityksen malli. Kyseiset mallit ovat pohjimmiltaan tapa organisoida ohjelmistokehitykseen liittyvää työtä, eivätkä ne ota kantaa ohjelmistokehitysprojektin tekniseen lähestymistapaan (Agile Alliance, 2018; Centers for Medicare & Medicaid Services, 2008).

Perinteinen, vesiputousmallin mukainen, ohjelmistonkehitysprosessi kulkee useiden peräkkäisten vaiheiden läpi. Näihin vaiheisiin kuuluvat yleensä mm.: vaatimusmäärittely, järjestelmän suunnittelu, ohjelmiston kehitystyö, testaus, implementaatio, käyttöönotto ja ylläpito. Perinteinen vesiputousmalli ei suosittele projektin aiempiin vaiheisiin palaamista tai tarkastelua enää uudestaan niiden jo valmistuttua. (Centers for Medicare & Medicaid Services, 2008; Balaji & Murugaiyan, 2012.) Tämä joustamattomuus puhtaassa vesiputousmallissa oli osaltaan mukana halussa uusien ketterimpien menetelmien kehittämiseen. Yksi ketterän ohjelmistokehityksen perusperiaatteita on iteratiivinen kehitystyö, jossa ohjelmistoon kohdistuvat muuttuvat vaatimukset hyväksytään ja tunnustetaan osaksi ohjelmiston kehitysprosessia (Beck ym., 2001; Balaji & Murugaiyan, 2012).

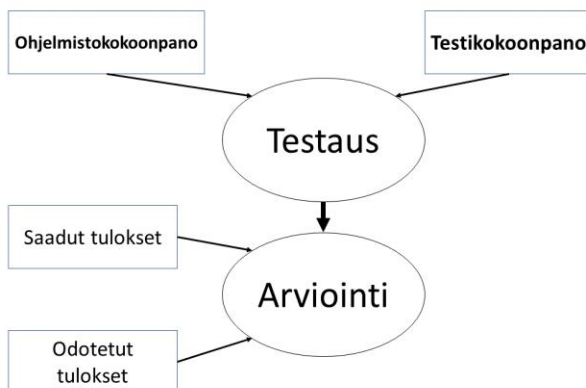
Ohjelmistotestaus on tärkeä ja olennainen osa sekä perinteistä vesiputousmallia että ketterän ohjelmistokehityksen mallia, mutta niiden lähestymistapa testausprosessiin on hyvin erilainen. Siinä missä vesiputousmallissa on aina erillinen testausvaihe vasta varsinaisen ohjelman rakennus- tai koodaus-

vaiheen jälkeen, ketterässä ohjelmistokehityksessä kehitystyö on iteratiivista ja testausta suoritetaan moneen kertaan samoissa iteraatioissa koodauksen kanssa. Toisin sanoen, ketterässä ohjelmistokehityksessä ei tehdä kerralla valmiiksi koko ohjelmaa, vaan ohjelmistokehitys koostuu useista iteraatioista, joissa tehdään valmiiksi ohjelmasta pieni osa, jota voidaan testata ja arvioida. (Balaji & Murugaiyan, 2012.)

Tässä luvussa esitellään seuraavaksi ohjelmistotestauksen konseptia ja tapoja, ohjelmiston laadun käsitettä, ja testivetoista ohjelmistokehitystä.

2.2 Ohjelmistotestaus

Ohjelmistotestaus on tapa tutkia empiirisesti testattavan ohjelman laatua ja ominaisuuksia. Tarkasteltavia kohteita testauksessa ovat esimerkiksi ohjelman luotettavuus, ohjelman suorituskyky, ylläpidettävyys, tietoturva ja käytettävyys (Ammann & Offutt, 2016, s. 48). Ohjelmistotestaus on tärkeä osa ohjelmistojen laadunvarmistusta, ja monet ohjelmistokehitysorganisaatiot käyttävät jopa 26 prosenttia IT-budjetistaan testauksessa (World Quality Report 2017-18, 2018, s. 41). Kuviossa 1 on havainnollistettu ohjelmistotestauksen prosessia.



KUVIO 1 Testin suorittaminen (Jovanović, 2006).

ISO/IEC/IEEE 29119 Software Testing-standardit ovat kokoelma kansainvälisiä, ohjelmistojen testausta varten kehitettyjä, standardeja, joita voidaan käyttää minkä tahansa ohjelmistokehitysmallin tai organisaation sisällä. Ohjelmiston testaamisen tavoitteet on näiden standardien sisällä jaettu kahteen eri ryhmään: 1. varmistus (eng. *verification*) ja 2. validointi. Varmistukseen pyrkivä ohjelmistotestaus arvioi toteuttaako jokin ohjelmistokehityksen tietyn vaiheen tulos aiemmassa vaiheessa asetetut vaatimukset. Validointiin pyrkivä ohjelmistotestaus tarkoittaa ohjelmistokehityksen lopussa tapahtuvaa arviointiprosessia, jonka tarkoituksena on varmistaa, että ohjelmisto noudattaa sille haluttua käyttötarkoitusta. (ISO/IEC/IEEE 29119 Software Testing, 2018.)

Ohjelmiston testauksella voidaan mm. tarjota sidosryhmille tärkeää tietoa testattavana olevan ohjelmistotuotteen tai -palvelun laadusta. Se voi myös tarjo-

ta yrityksille, objektiivisen ja erillisen, näkökulman testattavana olevaan ohjelmistoon, jota yritykset voivat hyödyntää esimerkiksi ohjelmiston implementointiin liittyvien riskien arvioinnissa ja tunnistamisessa. Ohjelmistotestaus on kaikista tehokkain tapa arvioida ja parantaa ohjelmiston laatua (Orso & Rothermel, 2014).

Ohjelmiston testaamisella voidaan vähentää myös sen käyttöönottoon liittyvää taloudellista riskiä, sillä testikäytössä esiin tulevat virheet on usein helpompi korjata, ja ne aiheuttavat vähemmän kustannuksia kuin tuotantokäytössä esiin tulevat virheet (Ammann & Offutt, 2016, s. 38-39).

Analysoidessaan aineistoa ohjelmistovioista, jotka havaittiin ja korjattiin useiden suurten valtiollisten urakoiden aikana, Ammann ja Offutt huomasivat, että mitä myöhäisemmässä vaiheessa ohjelmistovirheet havaittiin, sitä enemmän niiden korjaaminen maksoi. Käyttöönoton jälkeen havaittujen ohjelmistovirheiden korjaaminen saattoi maksaa jopa 50 kertaa enemmän kuin aikaisessa vaiheessa havaittujen virheiden korjaaminen (Ammann & Offutt, 2016, s. 38).

National Institute of Standards and Technology (NIST) viraston vuonna 2002 tekemä tutkimus kertoo, että ohjelmistovirheiden arvioidaan maksavan Yhdysvaltain taloudelle 59,5 miljardia dollaria vuosittain (Tassey, 2002). Tutkimuksen tarjoama tieto on nykypäivänä katsottuna erittäin vanhentunutta, mutta sitä voidaan peilata esimerkiksi Dale Jorgensonin ja Khuong Vun (2016) artikkelissaan esittelemään maailman kauppajärjestön (WTO) dataan, jonka mukaan Yhdysvaltojen tieto- ja viestintäteknistenpalveluiden vienti kasvoi vuosien 2000-2013 välillä 241 miljardilla dollarilla (WTO, 2015, Jorgenson & Vu, 2016, 395 mukaan).

2.3 Testausmenetelmät

Kansainvälinen Software Engineering Body of Knowledge (SWEBOK) -standardi tunnistaa kolme päätasoa, joilla testaamista voidaan suorittaa kehitysprosessin aikana:

1. **Yksikkötestaus** (eng. *unit testing*), joka tarkoittaa ohjelman yksittäisten, erikseen testattavissa olevien, osien toiminnallisuuteen keskittyvää testausta. Usein yksikkötestauksen suorittaa koodaaja itse, mutta joissain tapauksissa sen voi suorittaa myös joku muu, jolla on pääsy testattavaan koodiin ja vianetsintätyökaluihin (eng. *debugging tools*).
2. **Integraatiotestaus** (eng. *integration testing*), joka tarkoittaa ohjelmistokomponenttien välisten vuorovaikutusten testausta. Se pyrkii paljastamaan virheet rajapinnoissa ja integroitujen komponenttien (moduulien) välisessä vuorovaikutuksessa.
3. **Järjestelmätestaus** (eng. *system testing*), jonka tarkoituksena on testata kokonaan integroidun järjestelmän käyttäytymistä. (Bourque & Fairley, 2014, s. 86; Nidhra & Dondeti, 2012.)

Ohjelmistotestaustekniikat voidaan myös yleisesti luokitella kahteen eri kategoriaan: 1. black box -testaus ja 2. white box -testaus. Black box -testauksessa ohjelmiston testaaja ei pääse käsiksi sisäiseen lähdekoodiin, vaan tarkoitus on arvioida ohjelman käyttäytymistä niin kuin se olisi "iso musta laatikko", jonka sisään testaaja ei voi nähdä. Testaajalla on odotus siitä, mitä mustan laatikon pitäisi tulostaa tietyllä syötteellä, ja hän testaa, että varsinainen tuloste vastaa tätä odotettua tulostetta. White box -testaus tarkoittaa lähdekoodista saataviin tietoihin perustuvien testitapausten suunnittelua. Testaajalla (useimmiten koodin kehittäjä) on pääsy koodiin ja hän kirjoittaa testitapauksia suorittamalla ohjelman metodeja tietyillä parametreillä. (Nidhra & Dondeti, 2012; Khan & Khan, 2012.) Taulukossa 1 on esitelty edellä mainitut päätasot.

TAULUKKO 1 Testauksen päätasot kehitysprosessin aikana (Nidhra & Dondeti, 2012).

Testaustyyppi	**Näkyvyys**	**Kuka suorittaa testauksen?**	**Laajuus**	**Tavoite**
Yksikkö	White box.	Usein ohjelmoijat, jotka testaavat kirjoittamaansa koodia.	Pienet osat koodia, yleisesti ei yhtä luokkaa isompia.	Havaita kehitysvaiheessa syntyneet virheet koodissa.
Integraatio	White & Black box.	Usein ohjelmoijat, jotka testaavat kirjoittamaansa koodia.	Useille luokille.	Varmistaa ohjelmiston eri osien välinen toimivuus.
Järjestelmä	Black box.	Riippumattomat testaajat.	Koko tuotteille niille tarkoitetuissa ympäristöissä.	Varmistaa, että järjestelmä vastaa sille määriteltyihin vaatimuksiin.

Testivetoista ohjelmistokehitystä voidaan käyttää kaikilla näillä tasoilla, mutta se on yleisintä yksikkötestitasolla.

Koodikatselmointi on myös yksi tapa arvioida ja testata ohjelmakoodia. IEEE Standard Glossary of Software Engineering Terminology -standardi määrittelee koodikatselmoinnin seuraavasti: kokous, jossa ohjelmistokoodi esitetään projektihenkilöstölle, johtajille, käyttäjille, asiakkaille tai muille asianomaisille osapuolille kommentoitavaksi tai hyväksyttäväksi (ISO/IEC/IEEE International Standard - Systems and software engineering, 2010, s. 63).

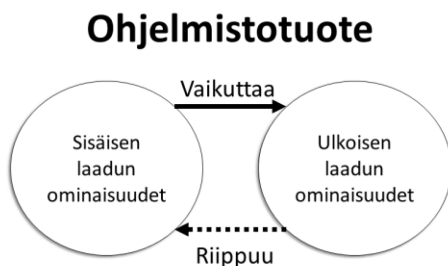
2.4 Ohjelmiston laatu

IEEE Standard Glossary of Software Engineering Terminology -standardin mukaan ohjelmatuotteiden laatu (eng. *software quality*) määritellään seuraavanlaisesti:

> ”Ohjelmistotuotteiden kyky täyttää ilmoitetut ja implisiittiset tarpeet, kun niitä käytetään spesifeissä olosuhteissa” (ISO/IEC/IEEE International Standard - Systems and software engineering, 2010, s. 334).

Ohjelmiston laatu voidaan myös jakaa ISO 9126 standardin mukaisesti sisäiseen laatuun, eli sellaisiin ominaisuuksiin, joita voidaan arvioida suorittamatta ohjelmaa, ja ulkoiseen laatuun, eli sellaisiin ominaisuuksiin, joita voidaan arvioida tarkastelemalla ohjelmaa sen suorittamisen aikana (ISO/IEC IS 9126-1, 2001, Miguel, Mauricio & Rodríguez, 2014, 39 mukaan).

Sisäisiä laatuominaisuuksia ovat mm. ylläpidettävyys, siirrettävyys, uudelleenkäytettävyys sekä testattavuus, ja ulkoisia laatuominaisuuksia ovat mm. toiminnallisuus, luotettavuus, käytettävyys, tehokkuus, joustavuus, luotettavuus ja yksinkertaisuus. Ulkoiset laatuominaisuudet ovat käyttäjille tärkeämpiä ja kehittäjät ovat kiinnostuneita sekä sisäisistä että ulkoisista laatuominaisuuksista. (Stavrinoudis & Xenos, 2008.) Sisäinen ja ulkoinen laatu ovat toisistaan riippuvaisia kuviossa 1 osoitetulla tavalla (ISO/IEC IS 9126-1, 2001, Miguel ym., 2014, 40 mukaan).



KUVIO 2 Sisäinen ja ulkoinen laatu ISO 9126 (Miguel ym., 2014, s. 40)

Sisäisen ja ulkoisen laadun käsitteet ovat jatkossa oleellisia tässä tutkielmassa referoitujen tutkimusten suhteen, koska monet niistä käyttävät kyseisiä käsitteitä arvioidessaan laatua.

Ulkoisen laadun mittaamista pidetään usein erittäin tärkeänä. Onkin yleistä yrittää ennustaa ulkoista laatua mittaamalla sisäisen laadun attribuutteja, koska sisäisen laadun attribuutit ovat mittaukseen käytettävissä jo ohjelmiston elinkaaren aikaisessa vaiheessa, kun taas ulkoisen laadun attribuutit ovat mitattavissa vasta silloin, kun tuote on jo valmis. Toiseksi, sisäisen laadun attribuut-

teja on helpompi, ja usein myös halvempi, mitata kuin ulkoisen laadun attribuutteja. (Fenton & Bieman, 2014, s. 464; Stavrinoudis & Xenos, 2008.)

Yksi yleinen tapa arvioida ohjelman ulkoista laatua on virheiden tiheyden mittaaminen. Mikäli hyväksymme, että virheitä on kahdentyyppisiä: tunnetut virheet, jotka on havaittu testauksen, tarkastelun sekä muiden keinojen avulla, ja piilevät virheet, jotka voivat olla läsnä järjestelmässä, mutta joista emme ole vielä tietoisia, voimme määrittää virheiden tiheyden kuviossa 3 esitellyllä tavalla. Tuotteen koko mitataan yleensä koodin rivimäärällä, tai joskus jollain muulla organisaation määrittämällä mittayksiköllä. (Fenton & Bieman, 2014, s. 472.)

$$\text{Virheiden tiheys} = \frac{\text{Tunnettujen virheiden määrä}}{\text{Tuotteen koko}}$$

KUVIO 3 Virheiden tiheys (Fenton & Bieman, 2014, s. 472).

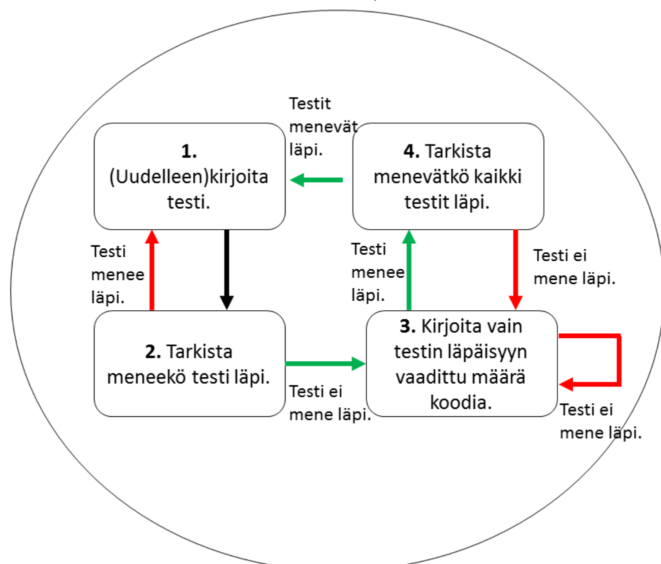
Ohjelmistokehityksen aikana on hyödyllistä hankkia aikaisia arvioita ohjelmistokomponenttien virheiden tiheydestä, koska kyseisillä arvioilla voidaan tunnistaa koodista sellaisia alueita, jotka vaativat lisätestausta (Nagappan & Ball, 2005).

2.5 Testivetoinen ohjelmistokehitys

Konsepti testien kirjoittamisesta ennen varsinaisen ohjelmakoodin kirjoittamista tuli tunnetuksi alun perin osana ketterää ohjelmistokehitys metodologiaa Extreme Programming. Se on sittemmin erkaantunut myös omaksi erilliseksi ohjelmistokehitysprosessikseen, joka tunnetaan tänä päivänä nimellä testivetoinen ohjelmistokehitys (TDD). (Bourque & Fairley, 2014, s. 94.)

Testivetoisessa ohjelmistokehityksessä testitapaukset kirjoitetaan aina ensimmäisenä ennen niiden testaamaa varsinaista toiminnallisuutta ohjelmakoodissa. Testin kirjoittamisen jälkeen ohjelmakoodia muokataan vain sen verran, että se läpäisee kirjoitetun testin. Ohjelmakoodia kirjoitetaan vain silloin, jos jokin kirjoitetuista testeistä ei mene läpi, koska kirjoitetut testit konkretisoivat ne vaatimukset, jotka ohjelmakoodin täytyy tyydyttää. Testien olisi siis tärkeä perustua esimerkiksi käyttötapauksista ja käyttäjätarinoista johdettuihin vaatimuksiin. Mikäli kaikki kirjoitetut testit menevät läpi tarkoittaa se, että ohjelmakoodilla ei, sillä hetkellä, ole vaatimuksia, joita pitäisi implementoida. TDD estää implementoimasta sellaista toiminnallisuutta, jota ei ole testattu, tai jolle ei ole tarvetta ohjelmassa. (Vorontsov & Newkirk, 2004, s. 24-25; Janzen & Saiedian, 2005.) George ja Williams (2004) huomasivat tutkimuksensa yhteydessä teetetyssä kyselyssä, että 80 % vastanneista ammattilaisohjelmoijista piti TDD:tä tehokkaana käytäntönä ja 78 % uskoi sen parantavan ohjelmoijien tuottavuutta.

Koodin refaktorointi, eli parantaminen, on myös iso osa testivetoista ohjelmistokehitystä. Vaikka aluksi onkin oleellista kirjoittaa vain testin läpäisyyn vaadittu määrä koodia, on jälkeenpäin suositeltavaa palata siistimään ja parantamaan, eli refaktoroimaan, jo kirjoitettua koodia (Beck, 2003, s. 18). Kuviossa 4 havainnollistetaan testivetoisen ohjelmistokehityksen elinkaarta. Refaktorointi tapahtuu neljännen askeleen jälkeen ja siitä palataan aina takaisin neljänteen askeleeseen varmistamaan, että kaikki testit menevät edelleen läpi.



KUVIO 4 Testivetoinen ohjelmistokehitys

Testivetoinen ohjelmistokehitys voi olla testauksessa kokemattomien ohjelmoijien kannalta vaativaa, mikä voi johtaa virheisiin sen prosessin toteuttamisessa. Testivetoista ohjelmistokehitystä harjoittavilla ohjelmoijilla teetetyssä kyselytutkimuksessa havaittiin, että 25 % ohjelmoijista tekee testivetoisen kehityksen prosessissa tiettyjä virheitä usein, tai jatkuvasti (Aniche & Gerosa, 2010).

3 TUTKIMUKSEN TULOKSET

Testivetoinen ohjelmistokehitys on herättänyt kiinnostusta myös akateemisessa maailmassa ja sen piirissä tehdyissä tutkimuksissa on havaittu sekä positiivisia vaikutuksia että ristiriitaisia tuloksia. Alla olevassa taulukossa 2 on koottu yhteen joitain tutkielmassa käsiteltyjä tutkimuksia, joissa oli havaittu testivetoisen ohjelmistokehityksen käyttöönottoon liittyviä etuja ja haittoja. + Tarkoittaa positiivista vaikutusta, - tarkoittaa negatiivista vaikutusta ja = tarkoittaa muuttumatonta.

TAULUKKO 2 Koonti käsitellyistä tutkimuksista.

Nro.	Tutkimus	Menetelmä	Tulokset
1	Pančur & Ciglarič, 2011.	Kontrolloidut kokeet.	+Koodikattavuus. +Koodin kompleksisuus
2	Bhat & Nagappan, 2006.	Tapaustutkimus TDD vs ei-TDD.	+Koodikattavuus. +Laatu
3	Siniaalto & Abrahamsson, 2007.	Tapaustutkimus TDD vs ei-TDD.	+Koodikattavuus. -Koodin yhtenäisyys.
4	Siniaalto & Abrahamsson, 2008.	Tapaustutkimus TDD vs ei-TDD.	-Joidenkin koodin osien heikkeneminen.
5	Madeyski, 2009	Empiirinen tutkimus.	+Koodikattavuus.
6	Janzen & Saiedian, 2008.	Empiirinen tutkimus.	+Koodin kompleksisuus
7	George & Williams, 2004.	Empiirinen tutkimus.	+Koodikattavuus. +Vähemmän ohjelmistovirheitä +Ulkoinen laatu. -Kehitystyö vei enemmän aikaa -TDD:n ajattelutapaan sopeutuminen vaikeaa.
8	Maximilien & Williams, 2003.	Empiirinen tutkimus.	+Vähemmän ohjelmistovirheitä +Ulkoinen laatu.
9	Nagappan ym., 2008.	Tapaustutkimukset.	+Vähemmän ohjelmistovirheitä +Ulkoinen laatu. -Kehitystyö vei enemmän aikaa

10	Gupta & Jalote, 2007.	Empiirinen tutkimus TDD vs ei-TDD.	+Tuottavuus =Laatu
11	Erdogmus ym., 2005.	Kontrolloitu koe TDD vs ei-TDD.	+Tuottavuus =Laatu
12	Flohr & Schneider, 2006.	Kontrolloitu koe TDD vs ei-TDD.	+Tuottavuus
13	Romano ym., 2017.	Monitapaustutkimus.	-Refraktoroinnin laiminlyömisestä johtunut koodin laadun heikkeneminen.
14	Aniche & Gerosa, 2010.	Kyselytutkimus.	-Refraktoroinnin laiminlyömisestä johtunut koodin laadun heikkeneminen.
15	Rafique & Mišić, 2013.	Meta-analyysi.	+Ulkoinen laatu. =Tuottavuus

Tässä luvussa perehdytään testivetoisesta ohjelmistokehityksestä tehtyyn tutkimukseen ja pyritään kokoamaan yhteen havaittuja positiivisia ja negatiivisia vaikutuksia ohjelmistoprojekteissa. Lopuksi tutkitaan testivetoisen ohjelmistokehityksen vaikutusta ohjelmiston sisäiseen ja ulkoiseen laatuun ja käsitellään tutkimustuloksissa esiintynyttä ristiriitaisuutta.

3.1 Tutkimuksessa havaitut hyödyt

Koodikattavuudella, tai testien kattavuudella, mitataan sitä kuinka kattavasti kirjoitetut testit suorittavat ohjelmakoodia. Tämä tieto auttaa päättelemään sitä kuinka tehokkaita testejä on kirjoitettu. (Microsoft Developer Network, 2018.) Useat tutkimukset ovat havainneet testivetoisen ohjelmistokehityksen positiivisen vaikutuksen koodikattavuuteen perinteisiin menetelmiin verrattuna (Pančur & Ciglarič, 2011; Bhat & Nagappan, 2006; Siniaalto & Abrahamsson, 2007; Madeyski, 2009). Tutkimuksessaan Pančur ja Ciglarič (2011) argumentoivat, että koska testivetoinen ohjelmistokehitys velvoittaa kehittäjän kirjoittamaan vain testien läpäisyyn vaaditun määrän koodia, pitäisi sen perusteellisesti seurattuna taata 100 % koodikattavuus. Muussa tutkimuksessa on käynyt ilmi, että siinä missä alan standardi koodikattavuus on 80-90 % luokkaa, oli se TDD:tä käyttävillä ohjelmoijilla 98 % (George & Williams, 2004).

Janzen ja Saiedian (2008) huomasivat tutkimuksessaan, että ”testaus jälkeenpäin” -ryhmän koodin kompleksisuus oli merkittävästi korkeampi kahdessa kuudesta analysoidusta tapauksesta TDD ryhmään verrattuna. Myös Pančur ja Ciglarič (2011) huomasivat tutkimuksessaan testivetoisella ohjelmistokehityksellä olevan positiivinen, vaikkakin pieni, vaikutus koodin kompleksisuuteen. Schroederin (1999) mukaan koodin korkeampi kompleksisuus korreloi korkeamman virhemäärän kanssa. Erittäin kompleksinen koodi on myös vaikeammin ymmärrettävää.

Joissain tutkimuksissa on havaittu, että testivetoista ohjelmistokehitystä käyttämällä voidaan vähentää ohjelmistovirheitä (George & Williams, 2004;

Maximilien & Williams, 2003; Nagappan, Maximilien, Bhat & Williams, 2008). Uudemmassa TDD:tä ja koodikatselmointia vertailleessa tutkimuksessaan Wilkerson, Nunamaker ja Mercer (2012) huomasivat, että vaikka koodikatselmointi oli TDD:tä tehokkaampi ohjelmistovirheiden vähentämisessä, oli sen implementointi myös huomattavasti kalliimpaa. TDD:tä voitaisiin siis ajatella kustannustehokkaana vaihtoehtona virheiden vähentämisessä.

Testivetoisella ohjelmistokehityksellä on joissain tutkimuksissa huomattu olevan myös positiivinen vaikutus tuottavuuteen perinteisiin menetelmiin verrattuna, vaikkakin tämä ero on usein pieni (Gupta & Jalote, 2007; Erdogmus, Morisio & Torchiano, 2005; Flohr & Schneider, 2006). Erdogmus ym. (2005) argumentoivat tutkimuksessaan, että positiivinen vaikutus tuottavuuteen selittyy tietyillä TDD:n synergistisillä eduilla, joita ovat: 1. Parempi tehtävän ymmärrys, 2. Parempi tehtävään keskittyminen, 3. Nopeampi oppiminen, 4. Pienempi kynnys koodin uudelleentyöstämiseen. TDD näytti myös tutkimuksessa parantavan vaatimusten ymmärtämistä kehittäjillä (Erdogmus ym., 2005).

3.2 Tutkimuksessa havaitut haitat

Joissain tutkimuksissa on havaittu testivetoisen ohjelmistokehityksen vaativan enemmän aikaa ja vaivaa sitä käyttäviltä kehittäjiltä. Esimerkiksi George ja Williams (2004) huomasivat 24 ammattilaista pariohjelmoijaa sisältäneessä tutkimuksessaan, että vaikka TDD:tä käyttäneet ohjelmoijat tutkimuksen perusteella tuottivat laadukkaampaa koodia, käyttivät he myös kehitystyössä 16 % enemmän aikaa. Myös Nagappan ym. (2008) huomasivat Microsoftin ja IBM:n sisäisillä kehitystiimeillä teetetyssä tapaustutkimuksessaan, että vaikka koodin virheiden tiheys laski 40-90 %, niin kehitystiimit myös kokivat 15-35 % kasvun alkuvaiheen kehitysajassa otettuaan TDD:n käyttöön. Näiden tulosten voidaan ajatella olevan ristiriitaisia tutkimusten kanssa, joiden perusteella TDD:llä on positiivinen vaikutus tuottavuuteen.

Romano, Fucci, Scanniello, Turhan ja Juristo (2017) huomasivat kaksikymmentä ohjelmistokehittäjää sisältäneessä, testivetoista ohjelmistokehitystä käsitelleessä, monimenetelmätutkimuksessaan, että kehittäjät keskittyivät kirjoittamaan nopeaa ja huonolaatuista koodia läpäistäkseen testit, mutta he eivät enää myöhemmin palanneet refaktoroimaan koodiaan kuten TDD vaatii. Kehittäjät eivät pitäneet refaktorointivaihetta tärkeänä (Romano ym., 2017). Kyseistä tendenssiä on havaittu myös muussa tutkimuksessa (Aniche & Gerosa, 2010). Tämä voi johtaa koodin laadun heikkenemiseen.

Siniaalto ja Abrahamsson (2008) huomasivat viisi pienen skaalan ohjelmistokehitysprojektia sisältäneessä vertailevassa tapaustutkimuksessaan, että TDD:n käyttö voi johtaa joidenkin koodin osien heikkenemiseen. Toisessa tutkimuksessaan Siniaalto ja Abrahamsson (2007) huomasivat, että kokemattomien kehittäjien käsissä TDD ei tuotakaan niin yhtenäistä koodia kuin aiempien tutkimustuloksien perusteella olisi syytä odottaa.

Eräiden kyselytulosten mukaan myös sopeutuminen TDD:n vaatimaan ajattelutapaan voi olla joillekin kehittäjille vaikeaa (George & Williams, 2004).

3.3 Testivetoisen ohjelmistokehityksen vaikutus sisäiseen ja ulkoiseen laatuun

TDD:llä on useissa tutkimuksissa havaittu olevan positiivinen vaikutus ohjelmiston ulkoiseen laatuun. Esimerkiksi George ja Williams (2004) huomasivat ammattilaisilla pariohjelmoijilla teetetyssä tutkimuksessaan, että TDD:tä käyttäneiden parien koodi läpäisi 18 % enemmän tutkimusta varten kehitetyistä black-box testitapauksista. Nagappan ym. (2008) huomasivat Microsoftin ja IBM:n sisäisillä kehitystiimeillä teetetyssä tapaustutkimuksessaan, että koodin virheiden tiheys laski 40-90 % TDD:n käyttöönoton seurauksena. Merkittävää koodin virhetiheyden laskemista on raportoitu myös muussa TDD:n käyttöä yritysmaailmassa käsittelevässä tapaustutkimuksessa (Maximilien & Williams, 2003). Myös Rafique ja Mišić (2013) huomasivat 27 tutkimusta käsitelleessä systemaattisessa meta-analyysissään, että TDD:n vaikutus ulkoiseen laatuun on positiivinen.

Ohjelmiston sisäisen laadun suhteen TDD:n vaikutus on epäselvä. Esimerkiksi aikaisempaan TDD:tä käsittelevään empiiriseen tutkimukseen keskittyvässä systemaattisessa kirjallisuuskatsauksessaan Kollanus (2010) huomasi, että TDD:n vaikutuksesta ohjelmiston parempaan sisäiseen laatuun on hyvin vähän todisteita. Tutkimuksessa analysoitiin 40 tarkkaan valittua tutkimusta, jotka raportoivat empiiristä todistusaineistoa testivetoisesta ohjelmistokehityksestä. Myös muissa tutkimuksissa on saavuttu tuloksiin, joissa ei havaita todisteita TDD:n positiivisesta vaikutuksesta sisäiseen laatuun (Shull ym., 2010).

3.4 Tuloksien ristiriitaisuus

Testivetoisen ohjelmistokehityksen tutkimuksessa on saavutettu paljon ristiriitaisia tuloksia. Esimerkiksi useista tutkimustuloksista poiketen Erdogmus ym. (2005) eivät omassa kontrolloidussa test-first ja test-last ohjelmistokehitystä vertailleessa tutkimuksessaan löytäneet todisteita siitä, että TDD auttaisi kehittäjiä saavuttamaan keskimäärin korkeampilaatuisen lopputuotteen. Sen sijaan he huomasivat, että koodin vähimmäislaatu kasvoi lineaarisesti ohjelmoijien kirjoittamien testien määrän perusteella riippumatta käytetystä kehitystyylistä (Erdogmus ym., 2005).

Myös Gupta ja Jalote (2007) huomasivat TDD:tä ja perinteisiä kehitystyylejä vertailleessa empiirisessä tutkimuksessaan, että ryhmien testaukseen käyttämä vaivannäkö kehitysvaiheessa vaikutti lopulliseen koodin laatuun huomattavasti riippumatta käytetystä kehitystyylistä. Guptan ja Jaloten mukaan tämä voi viitata siihen, että koodin laadussa havaitut eroavaisuudet TDD:tä ja muita

kehitystyylejä vertailevissa tutkimuksissa eivät välttämättä aiheudu vain mistään tietyn kehitystyylin käytöstä, vaan kehitysvaiheessa testaukseen käytetyn vaivannäön vaihtelevuudesta. Tutkimuksessa koodin laatua mitattiin sen kyvyllä suoriutua hyväksymistestitapauksista. (Gupta & Jalote, 2007.) Myös muussa tutkimuksessa on havaittu, että yksikkötestaukseen käytetty aika selittää, käytetyn kehitystyypin sijaan, ulkoisen laadun vaihtelevuuden TDD:n tutkimustuloksissa (Huang & Holcombe, 2009).

Pančur ja Ciglarič (2011) toteavat tutkimuksessaan, että vaikka TDD:n ja testaus jälkeenpäin menetelmien välillä oli eroja tutkimuksessa käsiteltyjen viiden muuttujan näkökulmasta, eivät mitkään niistä olleet tilastollisesti merkittäviä. Pančurin ja Ciglaričin mukaan TDD:tä verrataan usein sitä suosivissa tutkimuksissa puhtaaseen vesiputousmalliin, jossa kaikki testaus tapahtuu erillisenä vaiheena vasta kehitystyön jälkeen. Pančur ja Ciglarič erottivat TDD:n sisältämän mikroiteratiivisuuden yhtälöstä vertaamalla sitä samanlaisia mikroiteraatioita sisältävään vesiputousmalliin, iterative test-last development (ITL), jossa testaus tapahtui koodin kirjoittamisen jälkeen pienissä TDD:tä vastaavissa iteraatioissa. (Pančur & Ciglarič, 2011.) Saavutettujen tuloksien perusteella on mahdollista pohtia, onko osa TDD:llä saavutetuista eduista, varsinaisen testivetoisuuden sijaan, sen sisältämän mikroiteratiivisuuden ansiota. Myös muussa uudemmassa tutkimuksessa on tuettu TDD:n pieniä mikroiteraatioita sisältävän kehitysrytmin roolia, tutkimuksissa havaituissa positiivisissa tuloksissa, testivetoisuuden sijasta (Fucci ym., 2017).

TDD:n vaikutuksesta tuottavuuteen on myös epäselvyyksiä tutkimuksessa. Mm. Rafique ja Mišić (2013) huomasivat 27 tutkimusta sisältäneessä meta-analyysissään, että TDD:llä ei ollut huomattavaa vaikutusta tuottavuuteen. Muissakin tutkimuksissa on saavutettu ristiriitaisia tuloksia TDD:n raportoidusta positiivisesta vaikutuksesta tuottavuuteen (Nagappan ym., 2008; George & Williams, 2004).

4 YHTEENVETO

Tässä tutkielmassa pyrittiin selvittämään testivetoisen ohjelmistokehityksen etuja ja haittoja ohjelmistoprojekteissa käymällä läpi kirjallisuuskatsauksen keinoin 24 testivetoiseen ohjelmistokehitykseen liittynyttä tutkimusta. Tutkimusten perusteella on havaittu etuja testivetoisen ohjelmistokehityksen käyttöönotossa mm. ohjelmiston ulkoisen laadun, virheiden vähenemisen, koodikattavuuden, koodin ja vaatimusten ymmärtämisen, keskittymisen ja koodin kompleksisuuden suhteen. Löydettyjä haittoja, jotka voivat vaikeuttaa sen käyttöönottoa, olivat mm. testivetoisen ohjelmistokehityksen implementoimiseen vaadittava aika ja vaiva, testivetoisen ohjelmistokehityksen kehittäjiltä vaatima korkeampi taitokynnys, mahdollisesta TDD:n prosessin väärinymmärtämisestä aiheutuvat ongelmat ja mahdollinen joidenkin koodin osien heikkeneminen.

Yllättävää oli testivetoisen ohjelmistokehityksen tutkimustulosten ristiriitaisuus. Ristiriitaisia tuloksia havaittiin mm. TDD:n vaikutuksesta ohjelmiston laatuun, sen vaikutuksesta tuottavuuteen, ja joidenkin koodinosien heikkenemiseen. Jotkut tutkimustulokset kyseenalaistivat jopa varsinaisen testivetoisuuden roolin TDD:n käytöllä saavutettuihin etuihin, ja herättivät kysymyksen siitä, onko TDD-prosessin sisältämä kehitysvaiheen pienten askelten asettama rytmi ja iteratiivisuus olennaisempaa kuin se, tulevatko testit ennen koodia, vai vasta sen jälkeen (Pančur & Ciglarič, 2011; Fucci ym., 2017).

Yllättäviä olivat myös tutkimustulokset, joiden mukaan ristiriitaiset tulokset laadun suhteen testivetoisen ohjelmistokehityksen tutkimuksessa voisivat osaltaan johtua, vertailtavien kehitystyylien sijaan, testien kirjoittamiseen uhratun vaivannäön vaihtelevuudesta (Erdogmus ym., 2005; Gupta & Jalote, 2007; Huang & Holcombe, 2009). Toisin sanoen, voi olla mahdollista, että testien laatuun ja huolelliseen kirjoittamiseen panostaminen olisi jopa TDD:n implementoimista tärkeämpää mahdollisten etujen saavuttamiseksi.

Hyvin suunniteltu, toteutettu ja kontrolloitu lisätutkimus testivetoiseen ohjelmistokehitykseen on tarpeen, jotta voimme paremmin ymmärtää sen prosessin tarjoamia etuja. Etenkin tarkasti kontrolloiduilla empiirisillä kokeilla voitaisiin saavuttaa tärkeää lisätietoa siitä, mitkä muuttujat todella vaikuttavat tutkimuksissa havaittuihin etuihin ja haittoihin. Nykyiseltään useissa tutkimuksis-

sa on monesti hyvin vaikea sanoa johtuvatko havaitut tulokset esimerkiksi testihenkilöiden taitojen vaihtelevuudesta, varsinaisesta testivetoisesta prosessista, vai jostain vain siihen liittyvästä asiasta. Hyvin suunniteltu, toteutettu ja kontrolloitu lisätutkimus voisi auttaa meitä vastaamaan siihen, mitkä osat testivetoisesta ohjelmistokehityksestä ovat todella seuraamisen arvoisia, ja mitkä osat kenties kaipaisivat uudenlaista näkemystä. Tämä voisi johtaa uusien, vieläkin tehokkaampiin, kehitysmenetelmien ja suunnittelumenetelmien syntyyn.

LÄHTEET

Agile Alliance. (6.3.2018). What is agile software development. Haettu osoitteesta https://www.agilealliance.org/agile101/.

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

Aniche, M. F., & Gerosa, M. A. (2010, Huhtikuu). Most common mistakes in test-driven development practice: Results from an online survey with developers. In *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on* (s. 469-478). IEEE.

Balaji, S., & Murugaiyan, M. S. (2012). Waterfall vs. V-Model vs. Agile: A comparative study on SDLC. *International Journal of Information Technology and Business Management*, *2*(1), 26-30.

Beck, K., Beedle, M., Van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., ... & Kern, J. (2001). Manifesto for agile software development.

Beck, K. (2003). *Test-driven development: by example*. Addison-Wesley Professional.

Bhat, T., & Nagappan, N. (2006, Syyskuu). Evaluating the efficacy of test-driven development: industrial case studies. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering* (s. 356-363). ACM.

Bourque, P., & Fairley, R. E. Guide to the Software Engineering Body of Knowledge, Version 3.0. 2014. *IEEE Computer Society*.

Centers for Medicare & Medicaid Services. (2008). Selecting a development approach. *Centers for Medicare & Medicaid Services*, 1-10.

Erdogmus, H., Morisio, M., & Torchiano, M. (2005). On the effectiveness of the test-first approach to programming. *IEEE Transactions on software Engineering*, *31*(3), 226-237.

Fenton, N., & Bieman, J. (2014). *Software metrics: a rigorous and practical approach*. CRC Press.

Flohr, T., & Schneider, T. (2006, Kesäkuu). Lessons learned from an xp experiment with students: Test-first needs more teachings. In *International Conference on Product Focused Software Process Improvement* (s. 305-318). Springer, Berlin, Heidelberg.

Fucci, D., Erdogmus, H., Turhan, B., Oivo, M., & Juristo, N. (2017). A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?. *IEEE Transactions on Software Engineering*, *43*(7), 597-614.

George, B., & Williams, L. (2004). A structured experiment of test-driven development. *Information and software Technology*, *46*(5), 337-342.

Gupta, A., & Jalote, P. (2007, Syyskuu). An experimental evaluation of the effectiveness and efficiency of the test driven development. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on* (s. 285-294). IEEE.

Huang, L., & Holcombe, M. (2009). Empirical investigation towards the effectiveness of Test First programming. *Information and Software Technology*, *51*(1), 182-194.

ISO/IEC/IEEE 29119 Software Testing. (6.3.2018). The international standard for software testing. Haettu osoitteesta http://www.softwaretestingstandard.org/index.php

ISO/IEC/IEEE International Standard - Systems and software engineering. (15.12.2010). Vocabulary. Haettu osoitteesta http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5733835&isnumber=5733834

Janzen, D., & Saiedian, H. (2008). Does test-driven development really improve software design quality?. *Ieee Software*, *25*(2).

Janzen, D., & Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *Computer*, *38*(9), 43-50.

Janzen, D. S. (2005, Lokakuu). Software architecture improvement through test-driven development. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (s. 222-223). ACM.

Jorgenson, D. W., & Vu, K. M. (2016). The ICT revolution, world economic growth, and policy issues. *Telecommunications Policy*, *40*(5), 383-397.

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

Kaufmann, R., & Janzen, D. (2003, Lokakuu). Implications of test-driven development: a pilot study. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (s. 298-299). ACM.

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).

Kollanus, S. (2010, Syyskuu). Test-driven development-still a promising approach?. In *Quality of Information and Communications Technology (QUATIC), 2010 Seventh International Conference on the* (s. 403-408). IEEE.

Madeyski, L. (2009). *Test-driven development: An empirical evaluation of agile practice*. Springer Science & Business Media.

Maximilien, E. M., & Williams, L. (2003, Toukokuu). Assessing test-driven development at IBM. In *Software Engineering, 2003. Proceedings. 25th International Conference on* (s. 564-569). IEEE.

Miguel, J. P., Mauricio, D., & Rodríguez, G. (2014). A review of software quality models for the evaluation of software products. *arXiv preprint arXiv:1412.2977*.

Microsoft Developer Network. (30.3.2018). Using Code Coverage to Determine How Much Code is being Tested. Haettu osoitteesta https://msdn.microsoft.com/en-us/library/dd537628.aspx

Nagappan, N., & Ball, T. (2005, Toukokuu). Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering* (s. 580-586). ACM.

Nagappan, N., Maximilien, E. M., Bhat, T., & Williams, L. (2008). Realizing quality improvement through test driven development: results and experiences of four industrial teams. *Empirical Software Engineering*, *13*(3), 289-302.

Nidhra, S., & Dondeti, J. (2012). Black box and white box testing techniques-a literature review. *International Journal of Embedded Systems and Applications (IJESA)*, *2*(2), 29-50.

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.

Pančur, M., & Ciglarič, M. (2011). Impact of test-driven development on productivity, code and tests: A controlled experiment. *Information and Software Technology*, *53*(6), 557-573.

Rafique, Y., & Mišić, V. B. (2013). The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering*, *39*(6), 835-856.

Romano, S., Fucci, D., Scanniello, G., Turhan, B., & Juristo, N. (2017). Findings from a multi-method study on test-driven development. *Information and Software Technology*, *89*, 64-77.

Schroeder, M. (1999). A practical guide to object-oriented metrics. *IT professional*, *1*(6), 30-36.

Shull, F., Melnik, G., Turhan, B., Layman, L., Diep, M., & Erdogmus, H. (2010). What do we know about test-driven development?. *IEEE software*, *27*(6), 16-19.

Siniaalto, M., & Abrahamsson, P. (2007, Syyskuu). A comparative case study on the impact of test-driven development on program design and test coverage. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on* (s. 275-284). IEEE.

Siniaalto, M., & Abrahamsson, P. (2008). Does test-driven development improve the program code? Alarming results from a comparative case study. In *Balancing Agility and Formalism in Software Engineering* (s. 143-156). Springer, Berlin, Heidelberg.

Stavrinoudis, D., & Xenos, M. N. (2008, Kesäkuu). Comparing internal and external software quality measurements. In *JCKBSE*(s. 115-124).

Tassey, G. (2002). The economic impacts of inadequate infrastructure for software testing. *National Institute of Standards and Technology, RTI Project*, *7007*(011).

Vorontsov, A., & Newkirk, J. W. (2004). *Test-driven development in Microsoft. Net*. Microsoft Press.

Wilkerson, J. W., Nunamaker, J. F., & Mercer, R. (2012). Comparing the defect reduction benefits of code inspection and test-driven development. *IEEE Transactions on Software Engineering*, *38*(3), 547-560.

World Quality Report 2017-18. (11.4.2018). World Quality Report 2017–18 Ninth Edition. Haettu osoitteesta https://www.sogeti.com/globalassets/global/downloads/testing/wqr-2017-2018/wqr_2017_v9_secure.pdf