Samir Aissa Baccouche

Testivetoisen ohjelmistokehityksen hyödyntäminen oliopohjaisessa paradigmassa

Tietotekniikan kandidaatintutkielma

28. huhtikuuta 2015

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Samir Aissa Baccouche

Yhteystiedot: sakamiai@student.jyu.fi

Ohjaaja: Anneli Heimbürger

Työn nimi: Testivetoisen ohjelmistokehityksen hyödyntäminen oliopohjaisessa paradigmassa

Title in English: The invocation of test-driven development within the object-oriented paradigm

Työ: Kandidaatintutkielma

Sivumäärä: 30+2

Tiivistelmä: Testivetoisen kehityksen (lyhenne TDD) suosio on kasvanut ohjelmistotuotantomaailmassa, ja sen käyttöä suositellaan entistä enemmän. Tutkielmassa on tarkoitus selvittää todelliset vaikutukset testivetoisen kehityksen käytöstä kirjallisuuskatsauksen kautta. Tutkielma keskittyy TDD:n yleisiin hyötyihin ja haittoihin sekä sen vaikutukseen ohjelman sisäiseen ja ulkoiseen laatuun. Tulokset viittaavat siihen, että testivetoisella kehityksellä on sekä hyötyjä että haittoja, mutta lopullinen vaikutelma pysyy positiivisena ristiriitaisista tutkimustuloksista huolimatta. Testivetoinen kehitys paransi selkeästi ohjelmien ulkoista laatua, kun taas vaikutuksia sisäiseen laatuun ei voitu varmuudella todeta.

Avainsanat: testivetoinen kehitys, ohjelmistotuotanto, ohjelmistosuunnittelu, ohjelman laatu, testausmenetelmät

Abstract: Test-driven development (abbreviation TDD) has been gaining more and more popularity amongst agile software developers because of its many assumed benefits. In response to those assumptions, this bachelor's thesis aims at uncovering the real consequences of the use of TDD as a software development tool. For that purpose a literature review has been performed while focusing on the benefits and drawbacks of test-driven development. Its impacts on internal and external software quality have also been studied during the review. The results point towards both positive and negative outcomes, with an overall favourable impression. Furthermore, the positive effects on internal software quality were mitigated by

the multitude of metrics and conflicting results found in the studies. However, the most promising results were achieved in external software quality where many studies showed clear improvement when using test-driven development.

Keywords: test-driven development, software production, software design, software quality, testing techniques

Kuviot

Taulukot

Sisältö

1 Johdanto

Yleinen käsitys on, että ohjelmiston laatu paranee mitä enemmän testausta tehdään. Kuitenkaan testausta ei voi tehdä loputtomasti, sillä tuotannon ajan ja kustannusten on oltava mahdollisimman alhaiset (Schuh 2004; Baresi ja Pezze 2006). Näin ollen, miten voimme varmistaa sopivan tasapainon testauksen ja liiketoimintalogiikan kehityksen välillä? Yksi lupaava ratkaisu tähän ongelmaan on hyvin suosittu ja laajasti käytetty testaus- ja suunnittelumenetelmä ketterissä ohjelmistotuotantoprosesseissa: testivetoinen kehitys. Sen tunnettu lyhenne, TDD, on peräisin englanninkielisistä sanoista *Test Driven Development*, jotka kuvaavat ytimekkäästi sen kolmea pääkohtaa ohjelmistotuotannossa: testaus, suunnittelu, kehitys.

Tutkielmassa on tarkoitus selvittää todelliset vaikutukset testivetoisen kehityksen käytöstä ohjelmistotuotannossa kirjallisuuskautsauksen kautta. Tätä on syytä tutkia, sillä testivetoisen kehityksen oletetut monet hyödyt, muun muassa tuotettavuuden paraneminen, testien laajeneminen, ohjelmiston sisäisen ja ulkoisen laadun paraneminen, voivat parantaa ohjelmiston kehitysprosessia ja vähentää yleisiä kehityskustannuksia. Jos odotetut hyödyt konkretisoituvat tässä tutkielmassa (jos tutkimusten tulokset ovat myönteisiä TDD:lle), TDD:n periaatteita on syytä soveltaa muihin oliopohjaisiin tuotantoprosesseihin. Voimme myös nähdä tarpeen sen painottamiseen entistä voimakkaammin ohjelmoinnin opetuksessa, jotta opiskelijat omaksuisivat hyväksi todettuja ohjelmointitaitoja. On oletettava, että TDD:n käyttö voi mahdollisesti vakiintua ohjelmointiyrityksissä lähitulevaisuudessa, ja näin ollen opiskelijalla olisi valmiudet astua nopeammin työelämään.

Yleinen väärinkäsitys on, että testivetoinen kehitys on vain testausmenetelmä. Todellisuudessa TDD on pääasiassa suunnittelutyökalu, jonka yhteydessä käytetään testejä luomaan laadukkaampaa ja “puhtaampaa” lähdekoodia (eng. *clean code*) (Janzen ja Saiedian 2008). Näistä syistä tutkielma on jaettu kahteen pääosaan: Ensin keskitytään suunnitteluun eli ohjelmoijan ja ohjelmistotuotantoprosessin näkökulmiin. Toiseksi perehdytään TDD:n vaikutuksiin ohjelmiston laatuun, joka voi olla sisäinen tai ulkoinen. Sisäinen laatu koskee ohjelmiston koodia ja arkkitehtuuria. Ulkoinen laatu, joka välittyy loppukäyttäjälle, on suoraan verrattavissa ohjelman toiminnallisuuteen ja virheiden määrään. Näitä virheitä kehittäjät tekevät tuotantoprosessin aikana, ja näin ollen ohjelmiston sisäinen laatu on suorassa yhtey-

dessä ulkoiseen laatuun (McConnell 2004).

Tässä tutkielmassa aihe on rajattu TDD:n käyttöön oliopohjaisessa paradigmassa, jossa lähtökohtainen kieli on Java. Tämä valinta on tehty sillä perusteella, että Java:n kehitysympäristö sisältää helposti lähestyttävän JUnit -ohjelmistokehyksen testausta varten (Massol ja Husted 2004). Muita oliopohjaisia kieliä ei suljeta pois, jos niitä on käytetty olennaisissa tutkimuksissa.

Tutkielman rakenne on seuraavanlainen: Toisessa luvussa perehdytään keskeisiin käsitteisiin esittelemällä ketterä ohjelmistotuotanto, testaustyypit, testivetoinen kehitys ja ohjelman laatu. Kolmannessa luvussa tarkastellaan TDD:n tutkittuja hyötyjä ja haittoja ja pohditaan tuloksissa ilmenneitä ristiriitoja. Neljännessä luvussa käsitellään testivetoisen kehityksen vaikutuksia ohjelmiston sisäiseen ja ulkoiseen laatuun. Viimeisessä luvussa on yhteenveto.

2 Ohjelmistotestaus ketterässä maailmassa

Ohjelmistokehityksen kasvun aikana on erottunut kaksi eri mallia tuottaa ohjelmistoja: perinteinen vesiputousmalli ja mullistava ketterä malli. Uuden mallin myötä on kyseenalaistettu vanhoja tuotantotapoja, joissa on uskottu että hyvä suunnittelu ja vahva luottamus kehitysprosessiin takaa onnistuneen projektin (Williams 2012). Ketterä malli puolestaan pohjautuu nopeisiin muutoksiin ja vahvaan luottamukseen kehittäjien ammattitaitoon ja yhteistyöhön (Beck ym. 2001; Williams 2012). Vaikka molemmissa prosesseissa onkin eriäväisyyksiä, pyrkimys onnistuneeseen ja laadukkaaseen lopputulokseen on sama. Näin ollen identtisiä testaustyyppejä ja menetelmiä voisi hyödyntää molemmissa prosesseissa, sillä täsmällinen testaus pysyy tärkeänä vakiona ohjelmiston laadun varmistamiseen, kuten Chemuturi 2011 sen hahmottaa kirjassaan.

Tässä luvussa kuvataan yleisellä tasolla testausmenetelmiä ohjelmistotuotannossa. Luvun alussa esitellään lyhyesti ketterä ohjelmistotuotanto ja ohjelmistotestaus, jonka jälkeen luetellaan eri testaustyyppejä, joita käytetään ohjelmistokehityksessä. Luvun lopussa keskitytään ohjelmiston laatuun.

2.1 Ketterä ohjelmistotuotanto

Ketterä (eng. *agile*) ohjelmistotuotanto on ohjelmistokehityksen menetelmä, jossa painotetaan nopeita pieniä julkaisuja/iteraatioita, joiden ansiosta voidaan reagoida tehokkaammin muutoksiin, parantaa tiimin kommunikointia, nopeuttaa ohjelmistotuotannon prosessia sekä vähentää kustannuksia (Schuh 2004).

Historiallisesti ketterä ohjelmistotuotanto on saanut alkunsa monen kokeneen ohjelmistokehittäjän huomiosta, että ohjelmistokehitys on sitä hitaampaa mitä enemmän käytetään aikaa raskaisiin prosesseihin, ohjelman tarkkaan suunnitteluun ja dokumentaatioon, ennen varsinaista ohjelmiston kehitystä. Seurauksena on organisaation hitaus vastata muutoksiin sekä projektien viivästyminen useampaan vuoteen. Projektit eivät ole myös vastanneet asiakkaan toiveita ja laatukriteereitä, sillä asiakas ei ole pystynyt aktiivisesti osallistumaan lopulliseen tuotteeseen, mikä on johtanut asiakkaan tyytymättömyyteen ja projektin epäonnistu-

miseen. (Schuh 2004).

Ketterä ohjelmistotuotanto on ohjelmistokehityksen menetelmä, sillä se koostuu neljästä arvosta, jotka on koottu “agile manifesto” -julistukseen sekä kahdestatoista periaatteesta, joihin julistus pohjautuu (Beck ym. 2001). Moni ketterä prosessimalli on kehitetty seuraamaan ketteriä periaatteita, joista tunnetuimmat ovat Scrum, Extreme Programming (lyhenne XP), ominaisuusvetoinen kehitys (eng. *Feature-Driven Development*, lyhenne FDD) ja mukautuva ohjelmistokehitys (eng. *Adaptive Software Development*, lyhenne ASD) (Schuh 2004). Agile -menetelmät ovat hyvin keskeisiä tässä tutkielmassa, sillä niiden johdosta on syntynyt Extreme Programming prosessimalli, jossa on painotettu testivetoisen kehityksen käyttöä.

Edellä mainitut kaksitoista periaateetta ovat vielä ajankohtaisia, kuten Williams 2012 on tutkimuksessaan todennut. Niinpä hän on kysynyt 326 ammattitaitoiselta agile-kehittäjältä kuinka paljon he arvostavat julistuksen takana olevia kahtatoista periaatetta. Tutkimuksen tuloksena oli, että ketterät periaatteet vastaanotettiin hyvin ja että ne ovat olleet laajassa käytössä kehitysyhteisössä. Lisäksi tutkimuksessa ilmeni, että seitsemäs periaate, jonka mukaan “Toimiva ohjelmisto on edistymisen ensisijainen mittari” (Beck ym. 2001), on aiheuttanut eniten kommentteja kyselyyn vastanneilta. Kehittäjät olivat huolissaan siitä, että seitsemäs periaate ei huomioi ohjelman laadun tärkeyttä tuotantoprosessissa. Tämä voi johtaa kehityksen hidastumiseen sitä mukaa kun ohjelma kasvaa. Näin voi käydä, jos ei varmisteta kaikkien lisättyjen ominaisuuksien toimivuutta, mikä puolestaan voi aiheuttaa aikaa vieviä virheitä ja korjauskustannuksien lisääntymisen (Saff ja Ernst 2004). Nämä virheet voivat johtaa jopa henkilövahinkoihin, kuten traaginen Therac-25 -tapaus osoitti. Onnettomuudessa kolme ihmistä menehtyi säteily-yliannostukseen, jonka aiheutti virheellinen kontrolliohjelma, jossa ohjelmistonkehittäjä hyödynsi puutteellisesti testattuja ja virheellisiä koodikokonaisuuksia (Leveson ja Turner 1993).

2.2 Ohjelmistotestaus

Chemuturi 2010 kuvaa ohjelmistotestausta omassa kirjassaan yhtenä pääkeinona validoida (eng. *validate*) lopullisen ohjelmiston toimivuutta. Sen päätarkoitus on paljastaa kaikki virheet ja ongelmat ohjelmassa ennen lopullisen ohjelman saattamista asiakkaalle. Näin ollen

ohjelmistotestauksen avulla voidaan vahvistaa sen laatu, sen sijaan että varmistetaan (eng. *verify*) laadun saavuttaminen, kuten tehtäisiin kehityksen aikana (Chemuturi 2010). On muistettava erottaa varmistaminen ja validointi toisistaan. Varmistaminen (eng. *verification*) suoritetaan ennen sovelluksen suunnittelua ja kehitystä tai sen aikana. Sen sijaan sovelluksen vahvistaminen tai validointi (eng. *validation*) suoritetaan ohjelman tai ohjelmiston valmistumisen jälkeen (Chemuturi 2010). Nämä kaksi eri käsitettä voidaan erottaa kahdella eri kysymyksellä, kuten Boehm 1984 ne asettaa:

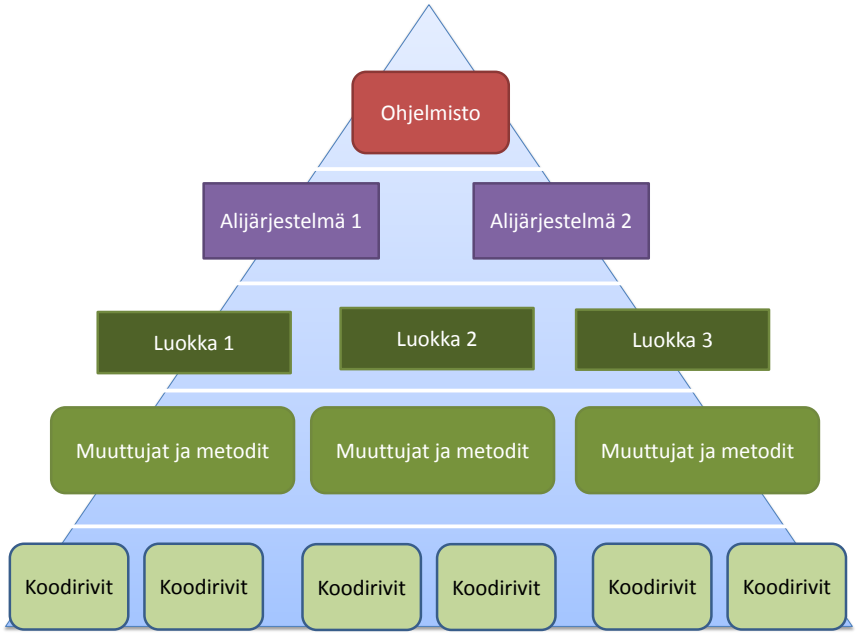
- Varmistaminen: "Olenko rakentamassa tuotetta oikein?" (eng. "Am I building the product right?")
- Validointi: "Olenko rakentamassa oikeaa tuotetta?" (eng. "Am I building the right product?")

Kuten näistä eri testaustavoitteista voi arvata, niin testaustyyppejä on hyvin monenlaisia, koska ne kohdistuvat eri ohjelmiston tuotantokehityksen vaiheisiin. Seuraavassa alaluvussa käsitellään lyhyesti näitä eri testaustyyppejä ja niiden käyttötarkoituksia.

2.3 Testaustyypit

Ohjelmiston eri testaustyypit voidaan luokitella sen mukaan minkälaisia ohjelman osia ne testaavat. Niinpä oliopohjainen ohjelmisto voidaan jakaa viiteen hierarkiseen tasoon, jossa pienempi taso sisältyy isompaan tasoon. Näin ollen jokainen taso koostuu eri osista, joita käyttävät ylemmän tason osat. Seuraavanlainen järjestys, pienemmästä osasta isompaan, on hyvin yleinen: koodirivit -> luokan sisäiset muuttujat ja metodit -> luokat -> alijärjestelmä-> järjestelmä (McConnell 2004). Nämä neljä tasoa on havainnollistettu kuviossa 1.

Eri testaustyypit ja niiden käyttötarkoitus on lueteltu taulukossa 1. Näitä testaustyyppejä voidaan implementoida eri tavoin, esimerkiksi manuaalinen testaus on hyvin helppo ja intuitiivinen testaustapa. Siinä ajetaan kokonainen ohjelma tai ohjelman osa ja tarkistetaan manuaalisesti että se toimii odotetusti. Ongelmana on sen tehottomuus, sillä ohjelman osat voivat olla hyvinkin laajoja ja niiden jatkuva testaus käsin on aikaavievä. Lisäksi testit on toistettava jokaisen muutoksen jälkeen. Nopeampi testaustapa on kehittää automaattiset testit, eli ohjelma, joka käy läpi varsinaista sovellusta ja tarkistaa sen toimivuuden. Tätä varten on olemas-



Kuvio 1. Ohjelmiston eri tasot ja osat

sa valmiita ohjelmistokehyksiä (eng. *framework*) jotka helpottavat tätä työtä. Näistä voidaan mainita esimerkiksi JUnit, joka on Java -ohjelmointikieleen valmistettu automaattinen testauskehys (Massol ja Husted 2004). Vielä toinen tapa testata ohjelmia on koodikatselmointi. Tämä on staattinen tapa varmistaa, että sovellus toimii sisäisesti oikein (Vu ym. 2009).

Samoin testauksen ajankohta vaihtelee kehityksen suhteen. Perinteinen tapa on luoda testejä koodiyksikön tai ohjelman jälkeen, tätä tapaa kutsutaan esimerkiksi termeillä testaa-lopussa (eng. *test-last*) tai testaa koodauksen jälkeen. Toinen menetelmä kirjoittaa testit on ennen tuotantokoodin kirjoittamista, eli testaa-ensin (eng. *test-first*), ja tämä on juuri TDD:n ydinajatus.

2.4 Testivetoinen kehitys

Testivetoinen kehitys on tapa yhdistää ohjelmiston suunnittelu ja testaus. Sen avulla voidaan varmistaa automaattisesti ohjelman toimivuus, siksi että jokaiselle koodiyksikölle on suunniteltu ja toteutettu testit ennen varsinaista koodin toteutusta. Se on iteratiivinen kehitystapa, jossa jokaisen kehitysiteraation alussa rakennetaan testit. Yksi iteraatio tai sykli on kuvattu

kuviossa 2, ja sen eri osat ovat seuraavanlaiset:

1. Testaa: aloitetaan ongelman ratkaisulla ja ohjelman alustavalla suunnittelulla eli testien kirjoittamisella.
2. Aja testi ja epäonnistu: tässä tarkistetaan, että testi ei mene virheellisenä läpi.
3. Ohjelmoi: kirjoita tarpeeksi tuotantokoodia, jotta testi menee läpi.
4. Aja testi ja onnistu: tarkistetaan, että ohjelma läpäisee testin.
5. Paranna: varmista, että koodi on järkevä ja refaktoroi tarvittaessa. Tarkista kaikkien testien toimivuus.

Kuvio 2. Testivetoisen kehityksen sykli

Historiallisesti TDD:n käyttö on yleistynyt ohjelmistotuotantomaailmassa XP:n ansiosta. Kent Beck, XP:n kehittäjä (Beck ja Andres 2004), on yksi ketterän ohjelmistotuotannon aallon perustajista (Beck ym. 2001). Kuitenkin TDD:n alkuperäiset juuret yltävät NASA:n vuonna 1959 aloitettuun “Project Mercury”:n, jonka tarkoitus oli lähettää ensimmäinen amerikkalainen miehitetty alus avaruuteen. Projektissa käytettiin iteraatioita ja TDD:n perusmuotuoa, jossa suunniteltiin ja rakennettiin testit ennen jokaisen iteraation alkua (Larman ja Basili 2003).

Testauksen suhteen testivetoinen kehitys sisältää melkein jokaisen testaustyypin. Yksittäinen kehittäjä keskittyy etukäteen yksikkötestaukseen, jossa oma tuotantokoodi käydään läpi. Kehittäjätiimi käyttää sitä integraatiotestauksessa, jossa ennen jokaisen alijärjestelmän kehityksen alkua suunnitellaan ja toteutetaan testit, jotka sen varmentavat. Kun testit on jo toteutettu niinden avulla voidaan rakentaa integraatiotestaussarja (eng. *integration test suite*), jolloin havaitaan kaikki muutoksista aiheutuneet virheet. Lisäksi järjestelmätestejä voidaan suunnitella ennen ohjelman kehityksen alkua. Jo olemassa olevia järjestelmätestejä voidaan hyödyntää hyväksymistestauksessa.

Vaikka TDD:n yhteydet ohjelmiston testaukseen ovat selkeät, niin Jeffries ja Melnik 2007 muistuttavat, että se on pääsääntöisesti ohjelmointi- ja suunnittelutyökalu, jonka avulla voi muun muassa vähentää virheiden määrää ja kehittäjien stressiä, sekä parantaa heidän luottamustaan omaan ohjelmointiin. He osoittavat myös, että ainoa tapa ymmärtää TDD:tä on käyttää sitä. Niinpä vaikka testivetoisen kehityksen peruskäsitteet on helppo sisäistää, niiden oikeaoppinen implementointi vaatii kehittäjältä joustavuutta ja vahvoja ohjelmointitaitojen omaksumista.

2.5 Ohjelmiston laatu

"Laatu ei ole ikinä sattuma; se on aina kovan työn seuraus" (eng. *"Quality is never an accident; it is always the result of intelligent effort."*)

Näillä John Ruskinin sanoilla Chemuturi 2011 on johdattanut "Ohjelmiston laatu" (eng. Software Quality) -lukua omassa kirjassaan. Siinä hän esittää viisi eri näkökulmaa ohjelman laadulle:

- Transsendenttinen näkökulma: perustuu filosofiseen näkemykseen, jonka mukaan ohjelman laatu määritellään kokemuksen perusteella, mutta on liian monimutkainen pystyäkseen laskemaan sitä.
- Käyttäjän näkökulma: Ohjelma on laadukas, jos se täyttää kaikki käyttäjän odotukset ja on monipuolisesti soveltuva monelle käyttäjälle.
- Tuotannon näkökulma: Laatu on verrattavissa siihen miten ohjelmisto täyttää asetetut valmistusvaatimukset ja kriteerit.
- Tuotteen näkökulma: Tässä näkökulmassa tuotteen, eli ohjelmiston, ulkoinen laatu on verrattavissa ohjelman sisäiseen laatuun. Näin ollen jos ohjelma on laadullinen sisäisesti, niin sen ulkoinen laatu hyötyy myös siitä. Tämä näkökulma sopii hyvin testivetoisen kehityksen laadun paranemisen väitteisiin, koska sen käyttö tähtää ohjelman sisäisen laadun paranemiseen.
- Arvoon perustuva näkökulma: Kuinka paljon ohjelmalla on arvoa tiettyyn laatutasoon nähden? Arvo määräytyy sen mukaan kuinka paljon asiakas on valmis maksamaan tietyn laatuisesta tuotteesta. Näin ollen on tehtävä kompromissi laadun ja arvon välillä.

Ohjelmiston laatu voidaan myös jakaa sisäiseen ja ulkoiseen laattun, kuten McConnell 2004 on ehdottanut “Code Complete” -kirjassaan. Ohjelman sisäinen laatu on ohjelmiston rakenteiden laatu, eli lähdekoodin laatu. Se perustuu ylläpidettävyyteen, joustavuuteen, siirrettävyyteen, uudelleenkäytettävyyteen, luettavuuteen, testattavuuteen ja ymmärrettävyyteen. Ohjelman ulkoinen laatu taas kuvaa sen miltä ohjelma vaikuttaa käyttäjälle. Näin ollen sen tärkeät kohdat ovat oikeellisuus, käytettävyys, tehokkuus, luotettavuus, eheys, mukautuvuus, täsmällisyys ja sitkeys (eng. *robustness*). Monet näistä ulkoisista ja sisäisistä kriteereistä ovat hyvin läheisiä tai samanlaisia, koska molemmat laadulliset tekijät ovat tiheässä yhteydessä toisiinsa. Kehittäjä on kiinnostunut sekä ohjelman ulkoisesta että sisäisestä laadusta, kun taas käyttäjä havaitsee vain ohjelman ulkoisen laadun (McConnell 2004).

Monessa tämän työn tutkimuksessa (Siniaalto ja Abrahamsson 2007; Sanchez, Williams ja Maximilien 2007; Vu ym. 2009; Shelton ym. 2012; Rafique ja Misic 2013) on käytetty tätä kaksijakoisuutta laadun arviointiin. Vaikka tamä jako on suhteellisen selkeä, niin yksiselitteinen arviointiperustelu laadullisiin kriteereihin on hyvin hankalaa. Siksi eri tutkimuksissa on käytetty eri kriteerejä laadun arvointiin. Tästä syystä tulokset eivät ole täysin vertailukelpoisia.

Oliopohjaiset mittarit ohjelman sisäisen laadun arviointiin ovat: painotetut metodit luokissa (eng. *Weighted Methods per Class*, lyhenne WMC), periytyvyyspuun syvyys (eng. *depth of inheritance tree*, lyhenne DIT), jälkeläisten määrä (eng. *number of children*. lyhenne NOC) ja kytkökset olioiden välissä (eng. *coupling between objects*, lyhenne CBO) (Siniaalto ja Abrahamsson 2007).

Yleisin tapa arvioida ohjelman ulkoinen laatu on laskea monta ohjelmavirhettä on löydetty järjestelmätestauksessa ja hyväksymistestauksessa. Tätä arvoa voi sitten verrata esimerkiksi koodirivien määrään (Rafique ja Misic 2013), jolloin tuloksena on virheiden tiheyden määrä ohjelmassa. Edellä mainittua tulosta voidaan verrata sen vaihteleviin arvoihin eri ohjelmajulkaisuissa (Slyngstad ym. 2008). Ohjelman absoluuttista ulkoista laatua on tässä kuitenkin mahdoton arvioida, koska emme voi olla ikinä varmoja ohjelman kaikkien virheiden löytämisestä.

Ohjelman laatuun on syytä panostaa, koska se voi johtaa selkeisiin säästöihin ohjelmistoyri-

tyksessä. Ohjelman laadun kustannukset voidaan jakaa kahteen eri tyyppiin: mukautuminen ja ei mukautuminen (Slaughter, Harter ja Krishnan 1998; Schulmeyer 2008). Jos ohjelmistoyritys ei panosta ohjelman laatuun hyvissä ajoin (mukautumisen kustannukset) se voi johtaa kalliimpiin korjauksiin ohjelman julkaisun jälkeen (ei mukautumisen kustannukset). Näin Slaughter, Harter ja Krishnan 1998 väittävät tutkimuksessaan. He osoittavat, että investointi laadukkaampaan sovellukseen on perusteltava, ja kustannustehokkain hetki panostaa ohjelman laadun paranemiseen on kehityksen alkuvaiheessa, jonka jälkeen laadun paraneminen heikkenee.

Taulukko 1. Kootut testaustyypit Chemuturin mukaan (Chemuturi 2011).

Nimi	Käyttö kehitysprosessissa	Tavoitteet
Yksikkötestaus	Ohjelmistokehityksen aikana eli koodin kirjoituksen aikana.	Havaita kaikki ongelmat, jotka ovat voineet syntyä kehityksen aikana.
Integraatiotestaus	Kun jokin yksikkö on valmis integroida alijärjestelmään tai moduuliin.	Tarkistaa, että eri yksiköt ja alijärjestelmät pystyvät toimimaan yhdessä.
Järjestelmätestaus	Osa sovelluksesta on valmis tai koko sovellus on valmis.	Varmistaa, että sovellus vastaa kehityksen alussa asetettuja vaatimuksia. Siihen kuuluvat laajat testausmenetelmät.
Hyväksymistestaus	Sovellus on valmis ja toimitettu asiakkaalle.	Asiakas validoi ohjelmiston eli tarkistaa sen laadun. Ei ole siis tarkoitus etsiä virheitä ohjelmasta, vaan katsoa vastaako ohjelma odotettuja tarpeita.
Regressiotestaus	Kun on tehty muutoksia sovellukseen.	Varmistaa, että sovellus toimii odotetusti muutosten jälkeen. Regressiotestauksessa ei tehdä uusia testejä, vaan käytetään jo aiemmissa vaiheissa luotuja testejä.

3 Testivetoinen kehitys suurennuslasin alla

Tässä luvussa on tarkoitus esitellä eri tuloksia, joita on saavutettu testivetoista kehitystä käsittelevissä tutkimuksissa. Luku on jaettu kolmeen osaan: ensin esitetään tutkitut hyödyt ja haitat ensimmäisessä ja toisessa osassa, jonka jälkeen kolmannessa osassa käsitellään ristiriitaiset tutkimustulokset. Yhteenveto tutkimuksista löytyy liitteestä 1. Ohjelman laadullisia käsitteitä saattaa tulla esiin, mutta niitä käsitellään tarkemmin luvussa 4.

3.1 Tutkitut hyödyt

Yksi mahdollinen hyöty TDD:n käytöstä koodikatselmointiin verrattuna, kuten Wilkerson, Jr. ja Mercer 2012 sen esittävät, on kustannustehokkuus. Heidän tutkimuksessaan haluttiin osoittaa, onko TDD yhtä tehokas kuin koodikatselmointi vähentämään ohjelman virheitä. Tulokset viittaavat siihen, että TDD:n käyttö kehitysprosessissa on edullisempi kuin koodikatselmointi. On myös käynyt ilmi, että kaikista tehokkain tapa vähentää virheitä, on käyttää molempia menetelmiä yhdessä (Wilkerson, Jr. ja Mercer 2012).

Toinen löydetty hyöty liittyy parempiin testeihin. Esimerkiksi on huomattu, että TDD:n testejä ajetaan useammin kuin perinteisen ohjelmistokehityksen testejä. Tästä on päätelty, että TDD parantaa ohjelman laatua, sillä testejä tehdään ja ajetaan useammin. Tutkimuksessa on verrattu ohjelman laatua ja testien käyttöä TDD:ssä ja perinteisessä testaa-lopussa menetelmässä (Geras, Smith ja Miller 2004). Toisessa tutkimuksessa Siniaalto ja Abrahamsson 2007 osoittavat, että testien kattavuus paranee huomattavasti TDD:n käytön ansiosta säilyttämällä kehityksen tuotettavuuden, kun on verrattu 16 eri TDD:tä käsittelevää tutkimusta. Parempi testien kattavuus tarkoittaa helpompaa ylläpitoa regressiotestien ansiosta ja laadukkaampia sovelluksia, jotka vastaavat paremmin vaatimuksia.

Kehitysprosessi hyötyy myös testivetoisesta kehityksestä luomalla järjestetyn ja sulavamman kehitysympäristön, joka vähentää kehityspaineita (Gupta ja Jalote 2007). On edelleen huomattu, että se parantaa refaktorointia kannustamalla muutosten tekemiseen. Esimerkiksi Slyngstad ym. 2008 hahmottavat tutkimuksessaan miten testaa-lopussa menetelmä heikkensi muutosten määrää projektissa, mutta kun käytettiin TDD:tä niin muutoksia ohjelmaan tuli

tehtyä enemmän. Edellä mainitussa tutkimuksessa on myös huomattu sovelluksen ylläpidettävyyden paraneminen. Näistä tuloksista on konkreettista hyötyä yrityksille, jotka haluavat siirtyä käyttämään TDD:tä parantaakseen ohjelmien kehitysprosessia ja ylläpidettävyyttä. Näin Canfora ym. 2006 arvelee tutkimuksessaan analysoimalla tuloksia, joiden mukaan TDD:n käyttö mahdollistaa tasaisemman kehitystahdin verrattuna testaa-lopussa menetelmään.

Empiirisisessä tutkimuksissa on huomattu eniten hyötyä testivetoisen kehityksen käytöstä. Sen käytöstä on seurannut runsaasti positiivisia havaintoja monessa eri tutkimuksessa. Tämän Buchan, Li ja MacDonell 2011 osoittavat, kun on haastateltu kehittäjiä pitkäaikaisen TDD:n käyttöönoton jälkeen yritysmaailmassa. Siellä TDD ei ole ollut kovin käytetty ennen sen painottamista kehitysprosessissa. Alussa se on otettu vastaan epäilevästi mutta skeptisyys muuttui nopeasti myönteisyydeksi. Raportoidut hyödyt ovat avoimempi suhtautuminen kehitykseen, laadukkaampi lähdekoodi joka vastaa odotuksia, laajempi testien kattavuus, luottamus koodin laatuun ja sekä kehittäjän että asiakkaan luottamus siihen, että ohjelmat toimivat niin kuin pitää (Buchan, Li ja MacDonell 2011). Crispin 2006 antaa myös samanlaisen näkökulman artikkelissaan TDD:n hyödyistä yritysmaailmassa.

3.2 Tutkitut haitat

On huomattu, että testivetoisessa kehityksessä ominaisuuksien kehittäminen ja testien kirjoittaminen vie paljon aikaa (Vu ym. 2009). Canfora ym. 2006 ovat myös päätyneet samaan tulokseen. Siksi he harkitsevat, että tutkimuksen aikaa pitäisi lisätä, sillä kehittäjät eivät pysty keskittymään testien kirjoittamiseen, jos projektille asetetaan liian tiukka aikataulu.

Yksi suuri haitta, joka on mainittu monessa tutkimuksessa, on testivetoisen kehityksen hankala omaksuminen. Monessa tutkimuksessa mainitaan että henkilöillä on vaikeuksia oppia TDD:n perusperiaatteet, varsinkin opiskelijoilla (Geras, Smith ja Miller 2004; Vu ym. 2009; Rafique ja Misic 2013).

Yksi muista vaikutuksista on virheiden lisääntyminen. Kun TDD on otettu käyttöön niin on tullut enemmän virheitä sovellukseen, mikä voi johtua siitä että TDD:n oppimiseen vaaditaan enemmän aikaa ja kokemusta (Slyngstad ym. 2008). Buchan, Li ja MacDonell 2011 ovat

samaa mieltä huomauttaessaan, että kului noin vuosi ennen kuin kaikki ymmärsivät täysin TDD:n hyödyt ja käyttivät sitä tehokkaasti. Joissakin tapauksissa motivaatio ei aina riitänyt TDD:n implementointiin (kehittäjän näkökulmasta) vanhaan tuttuun ja toimivaan lähdekoodiin (Buchan, Li ja MacDonell 2011).

TDD:n käyttö on myös aiheittanut lähdekoodin laadun heikkenemistä. Testivetoisen kehityksen testaustyyli, jossa testit kirjoitetaan ennen koodia, hankaloittaa ohjelmoijan työtä ja johtaa poikkeaviin ratkaisuihun, jotka heikentävät yleistä projektin laatua. (Gupta ja Jalote 2007). Muussa tutkimuksessa on käynyt ilmi, että TDD:n käyttö ei tuota kovin yhtenäistä koodia jos kehittäjät ovat kokemattomia (Siniaalto ja Abrahamsson 2007).

3.3 Ristiriidat tuloksissa

Monet ristiriidat tutkimustuloksissa asettavat haasteita TDD:n lopulliseen arviointiin. Rafique ja Misic 2013 viittaavat siihen, että TDD:stä on vain pientä hyötyä ohjelmistotuotannossa verrattuna perinteisiin testaustapoihin. Heidän meta-analyysissään otetaan kantaa ristiriitaisiin tutkimustuloksiin, joita saavutettiin akateemisissa ja yritysten ohjelmistokehitysympäristöissä. Usein akateemisissa tutkimuksissa testiaineistoa kerätään kokemattomilta ohjelmistokehittäjiltä, jotka eivät osaa täysin ottaa käyttöön TDD:n perusteita esimerkiksi testitapausten tekemiseen. Parempia tuloksia TDD:n hyödyistä on saatu tutkimusaineistosta, jota on kerätty kokeneilta ohjelmistoyritysten työntekijöiltä. Tulokset viittaavat siihen että TDD:n käyttö parantaa ohjelman laatua akateemisessa ympäristössä 10%, kun yritystutkimuksissa sen osuus on jopa 52% (Rafique ja Misic 2013).

Sekä Crispin 2006 että Rafique ja Misic 2013 ovat samaa mieltä siitä, että TDD on hankala implementoida projektiryhmässä, sillä se on vaikea oppia ja käyttää. Crispin 2006 kertoo tekstissään, että monien harjoitusten ja keskustelun kautta on saavutettu laadukasta ja puhdasta koodia. Rafique ja Misic 2013 osoittavat, että vain kokeneemmat yritysmaailman työntekijät pystyvät tehokkaimmin omaksumaan ja hyödyntämään TDD:n peruskäsitteitä. Näin ollen voimme olettaa tutkimustulosten erojen johtuvan eri osaamistasoista koehenkilöillä ja liian lyhyestä tutkimusajasta.

Muita ristiriitaisia tuloksia ovat muun muassa lähdekoodin laadun heikkeneminen (Gupta

ja Jalote 2007), joka on selvästi ristiriidassa empiirisen tutkimuksen kanssa (Buchan, Li ja MacDonell 2011), jonka mukaan lähdenkoodin laatu on parantunut. Edelleen ristiriidassa ovat tutkimukset, joissa ilmenee että TDD on hidas käyttää (Vu ym. 2009; Canfora ym. 2006) ja ne, joissa tuotettavuus säilyy ennallaan (Siniaalto ja Abrahamsson 2007).

Kaikesta huolimatta, vaikka kyseessä onkin haasteellinen testausmenetelmä, tulokset ovat olleet myönteisiä suurimmassa osassa tutkimuksista (Geras, Smith ja Miller 2004; Slyngstad ym. 2008; Janzen ja Saiedian 2008; Vu ym. 2009; Causevic, Sundmark ja Punnekkat 2011; Buchan, Li ja MacDonell 2011; Rafique ja Misic 2013; Munir ym. 2014).

4 Testivetoinen kehitys ohjelmiston laadun parantajana

Tässä luvussa on koottu tutkielmassa tutkitut vaikutukset testivetoisen kehityksen käytöstä. Tutkimukset on koottu kolmeen taulukkoon niiden laadun muutoksen mukaan. Ensimmäisessä taulukossa 2 on koottu tutkimukset, joissa on havaittu positiivista parannusta TDD:n käytöstä. Toisessa taulukossa 3 on koottu tutkimukset, joissa on havaittu negatiivinen vaikutus. Kolmas taulukko 4 kokoaa tutkimukset, joissa ei ole havaittu muutosta TDD:n käytön jälkeen. Taulukon tulosten etuliitteenä "+" -merkki kuvaa positiivista tulosta TDD:n käytöstä, "-" -merkki negatiivisen ja "=" -merkki kuvaa neutraaleja tuloksia. Yhteenveto kaikista tutkimuksista on koottu liitteessä 1.

Seuraavassa kahdessa alaluvussa käydään läpi tutkielman tuloksia ohjelman sisäisen ja ulkoisen laadun näkökulman mukaan. Aloitetaan sisäisellä laadulla ja siirrytään sitten ulkoiseen laatuun.

4.1 Ohjelmiston sisäinen laatu

Ohjelmiston sisäinen laatu on tärkeämpi kuin ulkoinen laatu, sillä sisäisen laadun heikkeneminen voi johtaa lopullisessa tuotteessa vakavampiin ongelmiin, joita on hankalampi korjata. Tutkimuksissa on huomattu parannusta muun muassa seuraavissa alueissa: testien ajon määrä (Geras, Smith ja Miller 2004), parempi testien kattavuus (Siniaalto ja Abrahamsson 2007), vastaavat virheenpoistotulokset Cleanroom menetelmän kanssa (Percival ja Harrison 2013), koodiyksiköiden pieneneminen (Janzen ja Saiedian 2008) ja koodivirheiden määrän väheneminen (Munir ym. 2014). Negatiiviset tulokset ovat olleet muun muassa seuraavat: koodikatselmointi on TDD:tä tehokkaampaa vähentämään koodivirheiden määrää (Wilkerson, Jr. ja Mercer 2012), TDD heikentää ohjelman sisäistä laatua yleisesti (Gupta ja Jalote 2007) ja ei tuota parempaa sisäistä koodin yhtenäisyyttä (Siniaalto ja Abrahamsson 2007).

Tuloksista voidaan päätellä, että TDD voi sekä parantaa että heikentää koodin sisäistä laatua. Tulokset ovat hyvin poikkeavia toisistaan ja voivat selittyä sillä, että TDD on hankala implementoida ja ohjelman sisäistä laatua on hankala arvioida objektiivisesti. Myös tutkimusmenetelmät poikkeavat toisistaan ja näin ollen emme voi vetää yhtenäistä johtopäätöstä

Taulukko 2. Tutkimukset, joissa havaittu laadun paraneminen.

	Tutkimus	Ohjelman laadun näkökulmasta
1	Munir ym. 2014	= Ei merkittävää lopputulosta ohjelman laadun paranemiseen. + Viitteitä siihen, että TDD vähentää koodivirheiden määrää.
2	Rafique ja Misic 2013	+ TDD:llä on pieni positiivinen vaikutus ohjelmiston ulkoiseen laatuun verrattun iteratiiviseen testaa-lopussa ja vesiputous -kehitykseen.
3	Buchan, Li ja MacDonell 2011	+ Parannusta vaatimusten ymmärtämiseen ja niiden toteuttamiseen joka johti parempaan ulkoiseen ohjelman laatuun ja asiakkaan tyytyväisyyteen.
4	Causevic, Sundmark ja Punnekkat 2011	48:stä kootusta tutkimuksesta: + 13 parannusta / - 1 huononemista /= 2 pysyi ennallaan /= 2 on tärkeitä havaintoja tutkimuksen kulusta./ = 20 tutkimuksista ei ole käsitellyt koodin laatua tai niissä ei ole ollut mainintaa siitä.
5	Vu ym. 2009	+ testaa-ensin kehityksessä 39% vähemmän virheitä kuin testaa-lopussa (ei yleistettävä havainto pienen koeryhmän takia).
6	Janzen ja Saiedian 2008	+ TDD:n käyttäjät kirjoittavat enemmän pienempiä yksiköitä, jota on helpompi testata: sisäisen laadun paraneminen. = Ei voitu varmistaa yhtenäisyyden paranemista ja kytkennän vähenemistä.
7	Slyngstad ym. 2008	+ On huomattu 35.86% vähemmän virheitä sovelluksessa.
8	Geras, Smith ja Miller 2004	= Ei suuria eroja testaa-ensin ja testaa-lopussa menetelmien välillä. + TDD testejä ajetaan useammin kuin perinteisiä testejä.
9	Williams, Maximilien ja Vouk 2003	+ 40% vähemmän virheitä ohjelmassa, kun on käytetty TDD:tä ilman mitään merkittäviä muutoksia tuotettavuuteen.

Taulukko 3. Tutkimukset, joissa havaittu laadun huonominen.

	Tutkimus	Ohjelman laadun näkökulmasta
1	Wilkerson, Jr. ja Mercer 2012	- koodikatselmointi on tehokkaampi kuin TDD vähentämään koodivirheitä.
2	Gupta ja Jalote 2007	- TDD heikentää ohjelman sisäistä laatua verrattuna perinteiseen kehitykseen.

Taulukko 4. Tutkimukset, joissa laatu pysyi ennallaan.

	Tutkimus	Ohjelman laadun näkökulmasta
1	Yahya ja Bakar 2014	= Akateemisissa tutkimuksissa on löydetty ristiriitaisia tuloksia.
2	Percival ja Harrison 2013	= Suhteellisen samat tulokset kyselytutkimuksessa Cleanroom ja TDD:n käytössä tehokkuuden, nopeuden, helppokäyttöisyyden, tyytyväisyyden näkökulmasta.
3	Siniaalto ja Abrahamsson 2007	+ Parempi testien kattavuus tarkoittaa laadukkaampia sovelluksia, jotka vastaavat paremmin vaatimuksia, ja helpompaa ylläpitoa. - Ei tuota parempaa sisäistä koodin yhtenäisyyttä.
4	Canfora ym. 2006	= Tulokset eivät olleet merkittäviä, sillä pidettiin liian lyhyitä koeotoksia; 5 tuntia koodausta.

näistä tuloksista. Nämä tulokset ovat kuitenkin arvokkaita, sillä ne havainnollistavat sen, että TDD:hen on suhtauduttava kriittisesti, koska sen tulokset voivat riippua monesta tekijästä.

4.2 Ohjelmiston ulkoinen laatu

Ohjelmiston ulkoisen laadun suhteen tutkimusten tulokset ovat paljon selkeämpiä ja yksimielisempiä, sillä tulokset ovat lähes täysin positiivisia. Ohjelmistossa on huomattu 39% (Vu ym. 2009) ja 35,86% (Slyngstad ym. 2008) vähemmän virheitä. Edelleen Buchan, Li ja MacDonell 2011 osoittavat parannusta vaatimusten ymmärtämisessä ja niiden toteuttamisessa. Toisessa tutkimuksessa, joka kokoaa monta eri TDD:n ulkoisen laadun vaikutuksia käsittelevää tutkimusta, on huomattu pientä positiivista vaikutusta ohjelmiston laatuun. Näissä tutkimuksissa on kuitenkin huomattu myös suurta hajautumista tuloksissa (Rafique ja Misic 2013).

Tilastot eivät kerro kaikkea, ja siksi Crispin 2006 korostaa kokemustaan ohjelmistotuotantoalalla esittelemällä kolme eri tapausta kolmesta eri firmasta, joissa hän on huomannut selkeitä parannuksia ohjelmien laadussa ja työn selkeydessä. Hän osoittaa myös, että sidosryhmien tyytyväisyys on parantunut selkeästi testivetoisen kehityksen ansiosta (Crispin 2006).

Huomataan liitteestä 1, että kaikissa tutkimuksissa, joissa käsiteltiin ohjelman ulkoista laatua (Slyngstad ym. 2008; Vu ym. 2009; Buchan, Li ja MacDonell 2011) tai yleisesti ohjelman laatua (Yahya ja Bakar 2014; Canfora ym. 2006; Williams, Maximilien ja Vouk 2003), on ollut positiivisia tuloksia TDD:n käytöstä. Saman havainnon ovat tehneet Causevic, Sundmark ja Punnekkat 2011 omassa tutkimuksessaan.

5 Yhteenveto

Tässä tutkielmassa käsiteltiin testivetoisen kehityksen käytön vaikutuksia oliopohjaisissa ohjelmistoissa. Tavoitteena on ollut ensiksi hahmottaa paremmin TDD:n hyödyt ja haitat ja toiseksi kartoittaa sen vaikutuksia ohjelman laatuun. Kirjallisuuskatsauksen kautta on analysoitu 15 tutkimusta, joiden perusteella on havaittu sekä positiivisia että negatiivisia vaikutuksia TDD:n käytöstä ohjelmistotuotannossa. Olennaisimmat positiiviset vaikutukset ovat: kustannustehokkuus, kattavammat ja käytetymmät testit, regressiotestien nopeutuminen, kehityspaineiden väheneminen, muutosten ja refaktoroinnin lisääntyminen, sekä luottamus siihen, että ohjelma toimii. Vastakohtana näihin tuloksiin on löydetty seuraavat negatiiviset vaikutukset: aikaa vievä prosessi, hankala omaksua ja implementoida, josta seuraa virheiden lisääntyminen ja lähdekoodin laadun heikkeneminen. Kuitenkin nämä kaikki vaikutukset ovat tapauskohtaisia, sillä tutkimustulosten suuren hajonnan ja ristiriitojen takia ei voida vetää lopullista johtopäätöstä sen vaikutuksista. Selvää myönteisyyttä on havaittu empiirisissä tutkimuksissa ja kyselytutkimuksissa, joissa vaikean käyttöönoton jälkeen on havaittu selkeää hyötyä testivetoisen kehityksen käytöstä.

Tutkielman viimeisessä osassa on koottu eri laadulliset vaikutukset TDD:n käytöstä. Sisäinen laatu on hankala määritellä, mikä näkyy siinä miten eri menetelmiä on käytetty mittaamaan sitä. Tämän vuoksi tutkimustuloksista ei voida vetää johtopäätöksiä siitä minkälaisia vaikutuksia TDD:llä on ohjelmiston sisäiseen laatuun. Tätä olisi hyvä tutkia tarkemmin tulevaisuudessa. Ennen sitä olisi hyvä tutkia parhaita ohjelman sisäisen laadun mittareita, sillä siitä olisi runsaasti hyötyä kehittäjille ja yrityksille.

Ohjelman ulkoinen laatu on tarjonnut paljon selkeämmän kuvan, sillä suurimmassa osassa tutkimuksista on raportoitu kohtalaista parannusta TDD:n ansiosta (Slyngstad ym. 2008; Vu ym. 2009; Buchan, Li ja MacDonell 2011; Causevic, Sundmark ja Punnekkat 2011; Rafique ja Misic 2013). Parempi ulkoinen laatu tarkoittaa sidosryhmien tyytyväisyyttä ja suurempaa luottamusta sovelluksen ja yrityksen kuvaan. Tämä onkin tärkeä havainto yrityksille, jotka haluavat parantaa ohjelman laatua lisätäkseen asiakkaiden tyytyväisyyttä.

Tutkielman oletetut tulokset poikkeavat osittain lopullisista tuloksista. Näin tapahtui lyhyis-

sä ohjatuissa tutkimuksissa, joissa on tarkkaan määritellyt koeympäristöt. Odotetut tulokset vahvistuivat empiirisissä tutkimuksissa, joissa kehittäjien kokemuksen perusteella TDD:n käytöstä on tullut paljon positiivista palautetta. Nämä tulokset on saavutettu yrityksissä, joissa TDD:n käyttöönotto on ollut systemaattista ja joilla on ollut aikaa oppia ja implementoida sitä omissa projekteissaan. Olisi tärkeää tarkistaa nämä empiiriset väitteet uusissa tutkimuksissa, joissa koeympäristö vastaisi enemmän yritysmaailmaa.

Päällimmäisenä pysyy mielessä testivetoisen kehityksen innovatiivinen tapa yhdistää suunnittelu ja testaus ohjelmistotuotannossa. Tämä mahdollistaa luotettavamman tavan kehittää laadukkaampia sovelluksia, jotka vastaavat enemmän asiakkaan odotuksia ja laatukriteerejä ja jotka ovat luotettavampia kehittää. On kuitenkin muistettava, että TDD ei ole hopealuoti ja sen käyttöön on suhtauduttava kriittisesti.

Lähteet

Baresi, Luciano, ja Mauro Pezze. 2006. “An Introduction to Software Testing”. *Electronic Notes in Theoretical Computer Science* 148, numero 1 (helmikuu): 89–111.

Beck, K., M. Beedle, A. V. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning ym. 2001. *Ketterän ohjelmistokehityksen julistus.* `http://agilemanifesto.org/iso/fi/`.

Beck, Kent, ja Cynthia Andres. 2004. *Extreme programming explained : embrace change* [kielellä eng]. 2. ed. xxii, 189 s. Lisäpainokset: Repr. 2005. - 10th pr. 2012. Boston MA: Addison-Wesley. ISBN: 0-321-27865-8 (nid.) :

Boehm, Barry W. 1984. “Verifying and validating software requirements and design specifications”. *IEEE Software:* 75–88.

Buchan, J., Ling Li ja S. G. MacDonell. 2011. “Causal Factors, Benefits and Challenges of Test-Driven Development: Practitioner Perceptions”. Teoksessa *Software Engineering Conference (APSEC), 2011 18th Asia Pacific,* 405–413. ID: 11. ISBN: 1530-1362.

Canfora, Gerardo, Aniello Cimitile, Felix Garcia, Mario Piattini ja Corrado Aaron Visaggio. 2006. “Evaluating Advantages of Test Driven Development: A Controlled Experiment with Professionals”. Teoksessa *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering,* 364–371. ISESE ́06. Rio de Janeiro, Brazil: ACM. ISBN: 1-59593-218-6. `http://doi.acm.org/10.1145/1159733.1159788`.

Causevic, A., Daniel Sundmark ja S. Punnekkat. 2011. “Factors Limiting Industrial Adoption of Test Driven Development: A Systematic Review”. Teoksessa *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on,* 337–346. ID: 4.

Chemuturi, Murali. 2010. *Mastering Software Quality Assurance : Best Practices, Tools and Technique for Software Developers* [kielellä English]. ID: 10520105. Ft. Lauderdale, FL, USA: J. Ross Publishing Inc., 2010. ISBN: 9781604276954.

Chemuturi, Murali. 2011. *Mastering software quality assurance : best practices, tools and techniques for software developers* [kielellä eng]. xvii, 358 p. Includes index. Fort Lauderdale Fla.: J. Ross Pub., ISBN: 9781604276954.

Crispin, L. 2006. "Driving Software Quality: How Test-Driven Development Impacts Software Quality". *Software, IEEE* 23, numero 6 (marraskuu): 70–71. ISSN: 0740-7459. doi:10.1109/MS.2006.157.

Geras, A., M. Smith ja J. Miller. 2004. "A prototype empirical evaluation of test driven development". Teoksessa *Software Metrics, 2004. Proceedings. 10th International Symposium on,* 405–416. ID: 1. ISBN: 1530-1435.

Gupta, A., ja P. Jalote. 2007. "An Experimental Evaluation of the Effectiveness and Efficiency of the Test Driven Development". Teoksessa *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on,* 285–294. ID: 4. ISBN: 1938-6451.

Janzen, D. S., ja H. Saiedian. 2008. "Does Test-Driven Development Really Improve Software Design Quality?" [Kielellä English]. Source type: scholarlyjournals; Object type: Article; Object type: Feature; CSAUnique: 5e197333ebd191e66d860a4aa5310c0aef8d64dc; AccNum: 20091252659 (AN); AccNum: 200909-70-0069742 (CI); AccNum: 200909-90-0079460 (EA); DOI: 10.1109/MS.2008.34; ISSN: 0740-7459; Peer Reviewed: true, *IEEE Software* 25 (2): 77–84. http://search.proquest.com/docview/34573097?accountid=11774.

Jeffries, Ron, ja G. Melnik. 2007. "Guest EditorsÍntroduction: TDD–The Art of Fearless Programming". ID: 1, *Software, IEEE* 24 (3): 24–30.

Larman, C., ja V. R. Basili. 2003. "Iterative and incremental developments. a brief history". ID: 1, *Computer* 36 (6): 47–56.

Leveson, N. G., ja C. S. Turner. 1993. "An investigation of the Therac-25 accidents". ID: 1, *Computer* 26 (7): 18–41.

Massol, V., ja T. Husted. 2004. *JUnit in Action.* USA: Manning Publications Co. ISBN: 1930110995.

McConnell, Steve. 2004. *Code complete* [kielellä eng]. 2nd ed. 914 s. Alanimeke kannessa: A practical handbook of software construction. Redmond (WA): Microsoft Press. ISBN: 0-7356-1967-0.

Munir, Hussan, Krzysztof Wnuk, Kai Petersen ja Misagh Moayyed. 2014. "An Experimental Evaluation of Test Driven Development vs. Test-last Development with Industry Professionals". Teoksessa *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering,* 50:1–50:10. EASE '14. London, England, United Kingdom: ACM. ISBN: 978-1-4503-2476-2. http://doi.acm.org/10.1145/2601248.2601267.

Percival, Jonathan, ja Neil Harrison. 2013. "Developer Perceptions of Process Desirability: Test Driven Development and Cleanroom Compared". Teoksessa *System Sciences (HICSS), 2013 46th Hawaii International Conference on,* 4800–4809. ID: 1. ISBN: 1530-1605.

Rafique, Y., ja V. B. Misic. 2013. "The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis". ID: 1, *Software Engineering, IEEE Transactions on* 39 (6): 835–856.

Saff, David, ja Michael D. Ernst. 2004. "An Experimental Evaluation of Continuous Testing During Development". *SIGSOFT Softw.Eng.Notes* 29, numero 4 (heinäkuu): 76–85. http://doi.acm.org/10.1145/1013886.1007523.

Sanchez, J. C., L. Williams ja E. M. Maximilien. 2007. "On the Sustained Use of a Test-Driven Development Practice at IBM". Teoksessa *Agile Conference (AGILE), 2007,* 5–14. ID: 1.

Schuh. 2004. *Integrating Agile Development in the Real World* [kielellä English]. ID: 10078511. Hingham, MA, USA: Charles River Media, 200411.

Schulmeyer, G. Gordon. 2008. *Handbook of software quality assurance* [kielellä eng]. 4th ed. 1 online resource (xx, 464 p.) Boston: Artech House, ISBN: 9781596931879.

Shelton, W., Nan Li, P. Ammann ja J. Offutt. 2012. "Adding Criteria-Based Tests to Test Driven Development". Teoksessa *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on,* 878–886. ID: 3.

Siniaalto, M., ja P. Abrahamsson. 2007. “A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage”. Teoksessa *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on,* 275–284. ID: 5. ISBN: 1938-6451.

Slaughter, Sandra A., Donald E. Harter ja Mayuram S. Krishnan. 1998. “Evaluating the Cost of Software Quality”. *Commun.ACM* 41, numero 8 (elokuu): 67–73. `http://doi.acm.org/10.1145/280324.280335`.

Slyngstad, O. P. N., Jingyue Li, R. Conradi, H. Ronneberg, E. Landre ja H. Wesenberg. 2008. “The Impact of Test Driven Development on the Evolution of a Reusable Framework of Components Â– An Industrial Case Study”. Teoksessa *Software Engineering Advances, 2008. ICSEA ́08. The Third International Conference on,* 214–223. ID: 7.

Wilkerson, J. W., J. F. Nunamaker Jr. ja R. Mercer. 2012. “Comparing the Defect Reduction Benefits of Code Inspection and Test-Driven Development”. ID: 1, *Software Engineering, IEEE Transactions on* 38 (3): 547–560.

Williams, L., E. M. Maximilien ja M. Vouk. 2003. “Test-driven development as a defect-reduction practice”. Teoksessa *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on,* 34–45. ID: 1. ISBN: 1071-9458.

Williams, Laurie. 2012. “What Agile Teams Think of Agile Principles”. *Commun.ACM* 55, numero 4 (huhtikuu): 71–76. `http://doi.acm.org/10.1145/2133806.2133823`.

Vu, J. H., N. Frojd, C. Shenkel-Therolf ja D. S. Janzen. 2009. “Evaluating Test-Driven Development in an Industry-Sponsored Capstone Project”. Teoksessa *Information Technology: New Generations, 2009. ITNG ́09. Sixth International Conference on,* 229–234. ID: 2.

Yahya, N., ja N. S. Awang Abu Bakar. 2014. “Test driven development contribution in universities in producing quality software: A systematic review”. Teoksessa *Information and Communication Technology for The Muslim World (ICT4M), 2014 The 5th International Conference on,* 1–6. ID: 10.

Liitteet

1 Tutkimusten yhteenveto

Seuraavalla sivulla.

Tutkimus	Ohjelman laadun näkökulmasta	laadun vaikutuksen kohde	tutkitut testausmenetelmät
Wilkerson, Jr. ja Mercer (2012)	- koodikatselmointi on tehokkaampi kuin TDD vähentämään koodivirheitä.	sisäinen	TDD ja koodikatselmointi
Geras, Smith ja Miller (2004)	=Ei suuria eroja testaa-ensin ja testaa-lopussa menetelmien välillä. + TDD testejä ajetaan useammin kuin perinteisiä testejä.	sisäinen	testaa-lopussa ja TDD:n testaa-ensin
Gupta ja Jalote (2007)	- TDD heikentää ohjelman sisäistä laatua verrattuna perinteiseen kehitykseen.	sisäinen	TDD -koodaus ja CCD - koodaus (Conventional Code Development)
Siniaalto ja Abrahamsson (2007)	+ Parempi testien kattavuus tarkoittaa laadukkaampia sovelluksia, jotka vastaavat paremmin vaatimuksia, ja helpompaa ylläpitoa. - Ei tuota parempaa sisäistä koodin yhtenäisyyttä.	sisäinen	iteratiivinen testaa lopussa ja TDD
Percival ja Harrison (2013)	+ Suhteellisen samat tulokset kyselytutkimuksessa Cleanroom ja TDD:n käytössä tehokkuuden, nopeuden, helppokäyttöisyyden, tyytyväisyyden näkökulmasta.	sisäinen	cleanroom ja TDD
Janzen ja Saiedian (2008)	+ TDD:n käyttäjät kirjoittavat enemmän pienempiä yksiköitä, jota on helpompi testata: sisäisen laadun paraneminen. = Ei voitu varmistaa yhtenäisyyden paranemista ja kytkennän vähenemistä.	sisäinen	testaa-lopussa ja TDD
Munir ym. (2014)	= Ei merkittävää lopputulosta ohjelman laadun paranemiseen. + Viitteitä siihen, että TDD vähentää koodivirheiden määrää.	sisäinen	testaa-lopussa ja TDD
Vu ym. (2009)	+ testaa-ensin kehityksessä 39% vähemmän virheitä kuin testaa-lopussa (ei yleistettävä havainto pienen koeryhmän takia).	ulkoinen	testaa-lopussa ja TDD:n testaa-ensin
Slyngstad ym. (2008)	+ On huomattu 35.86% vähemmän virheitä sovelluksessa.	ulkoinen	testaa-lopussa ja TDD
Buchan, Li ja MacDonell (2011)	+ Parannusta vaatimusten ymmärtämiseen ja niiden toteuttamiseen joka johti parempaan ulkoiseen ohjelman laatuun ja asiakkaan tyytyväisyyteen.	ulkoinen	TDD ja vesiputous testaa-lopussa
Rafique ja Misic (2013)	+ TDD:llä on pieni positiivinen vaikutus ohjelmiston ulkoiseen laatuun verrattun iteratiiviseen testaa-lopussa ja vesiputous -kehitykseen.	ulkoinen	iteratiivinen testaa-lopussa, vesiputousmallai ja TDD
Yahya ja Bakar (2014)	= Akateemisissa tutkimuksissa on löydetty ristiriitaisia tuloksia.	yleinen	TDD:n tutkitut hyödyt akateemisessa ympäristössä
Causevic, Sundmark ja Punnekkat (2011)	48:stä kootusta tutkimuksesta: + 13 tutkimuksessa on huomattu parannusta ohjelman sisäisessä laadussa - 1 tutkimuksessa on havaittu negatiivisen tuloksen TDD:n käytön seurauksesta. = 2 ei ole havaittu koodin paranemista. = 2 on tärkeitä havaintoja tutkimuksen kulusta. = 20 tutkimuksista ei ole käsitellyt koodin laatua tai niissä ei ole ollut mainintaa siitä	yleinen	TDD ja mikä hidastaa sen käyttöä yrityksissä
Canfora ym. (2006)	= ulokset eivät olleet merkittäviä, sillä pidettiin liian lyhyitä koeotoksia; 5 tuntia koodausta.	yleinen	testaa-lopussa ja TDD
Williams, Maximilien ja Vouk 2003	+ 40% vähemmän virheitä ohjelmassa, kun on käytetty TDD:tä ilman mitään merkittäviä muutoksia tuotettavuuteen.	yleinen	TDD ja perinteinen koodaustapa

vihreä = positiivinen vaikutus | punainen = negatiivinen vaikutus | valkoinen = neutraali vaikutus