TEKNILLINEN KORKEAKOULU
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan tutkinto-ohjelma

TDD:n edut - tarua vai totta?

Kandidaatintyö

Mikko Vestola

Tietotekniikan laitos
Espoo 2008

TEKNILLINEN KORKEAKOULU
Informaatio- ja luonnontieteiden tiedekunta
Tietotekniikan tutkinto-ohjelma

KANDIDAATINTYÖN
TIIVISTELMÄ

Tekijä: Mikko Vestola

Työn nimi:
TDD:n edut - tarua vai totta?

Päiväys: 1. joulukuuta 2008 **Sivumäärä:** 38 + 16

Pääaine: Ohjelmistotuotanto- ja liiketoiminta

Vastuuopettaja: prof. Lauri Savioja
Työn ohjaaja: TkL Jari Vanhanen

Testivetoinen ohjelmointi (Test-driven development, TDD) on suosittu suunnittelukäytäntö, jossa yksikkötestit kirjoitetaan pienissä osissa ennen varsinaista toiminnallisuuden toteutusta. TDD:n on väitetty tuovan monia etuja verrattuna perinteisiin menetelmiin, joissa yksikkötestit kirjoitetaan vasta varsinaisen toteutuksen jälkeen. Tässä työssä selvitettiinkin, mitä etuja TDD:n käytöstä on kirjallisuudessa esitetty. TDD:n on väitetty esimerkiksi parantavan koodin sisäistä ja ulkoista laatua pienentämällä koheesiota ja kompleksisuutta sekä tuottamalla vähemmän virheitä. Väitettyjä etuja ovat myös suurempi testikattavuus, parempi tuottavuus ja helppo opittavuus. Näiden esitettyjen etujen todenperäisyyttä analysoitiin alan tietokannoista löydettyjen tutkimusten pohjalta.

Tutkimukset TDD:stä olivat paikoin hyvin ristiriitaisia. Tutkimustuloksiin vaikutti paljon se, miten ja missä määrin yksikkötestausta käytettiin TDD:n vertailukohteissa? Monet TDD:n eduista voidaankin saavuttaa pelkästään yksikkötestausta lisäämällä. Tutkimusten perusteella osalle väitteistä ei löydy faktatietoon pohjautuvia perusteita ja osa väitteistä on suorastaan valheellisia. Esimerkiksi TDD:n vaikutukset tuottavuuteen ja koodin sisäiseen laatuun eivät ole niin selkeitä kuin TDD:n puolestapuhujat antavat ymmärtää. Sen sijaan tutkimukset tukevat esimerkiksi väitteitä virheiden vähenemisestä sekä testikattavuuden parantumisesta.

Työn tuloksena syntyi luettelo TDD:n esitetyistä eduista sekä yhteenveto TDD:n empiirisistä tutkimuksista, jotka löytyvät työn liitteinä. Työn tulokset pyrkivät antamaan tutkimuksiin pohjautuvan objektiivisen kuvan TDD:n eduista ja toivottavasti rohkaisevat tutkimaan niitä alueita TDD:stä, joiden esitetyille hyödyille ei löytynyt empiirisiä todisteita.

Avainsanat: TDD, testivetoinen ohjelmointi, edut, hyödyt, empiiriset tutkimukset
Kieli: Suomi

HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Degree Program of Computer Science and Engineering

ABSTRACT OF BACHELOR'S THESIS

Author: Mikko Vestola

Title of thesis:
Benefits of Test-Driven Development - Fact or Fiction?

Date: December 1 2008 **Pages:** 38 + 16

Professorship: Software Engineering and Business

Supervisor: Professor Lauri Savioja
Instructor: Jari Vanhanen, Lic. Sc. (Tech)

Test-driven development (TDD) is a popular software development technique where unit tests are written in small parts before actual coding. It is said that TDD yields many benefits compared to traditional development techniques where unit testing occurs after actual coding. In this study, the suggested benefits of TDD were collected from the literature. For example, it was claimed that TDD improves internal and external quality of software by lowering cohesion and complexity, in addition to lowering defect rate. The claimed benefits are also: higher test coverage, better productivity and easy to learn. The reality of the suggested benefits was analyzed based on empirical studies collected from the research databases.

The empirical studies about TDD were partly conflicting. The results were much affected by how and in what extent did unit testing occur in the projects of comparison. Many of the suggested benefits can be achieved by simply improving unit testing. Part of the claims are not really supported by existing empirical studies and some claims are simply false. For example, the impact of TDD to productivity and internal quality of software is not so clear than the advocates of TDD suggest. However, the claims that TDD improves test coverage and reduces defect rate appear to be true on the strength of the empirical studies.

The outcome of this study is a list containing the suggested benefits of TDD and a summary of the most important empirical studies of TDD. This study tries to give an objective view of TDD and hopefully the results will encourage to further study those areas of TDD where the impact of TDD is not so clear.

Keywords: TDD, Test-Driven Development, Test-Driven Design, benefits, empirical studies
Language: Finnish

Käytetyt lyhenteet

ACM	Association for Computing Machinery	Kansainvälinen tekniikan alan järjestö.
CBO	Coupling between Object Classes	Koodin kytkennän mitta.
CC	Cyclomatic Complexity	Koodin kompleksisuuden mitta.
IBM	International Business Machines	Yksi suurimmista teknologia-alan yrityksistä.
IEEE	Institute of Electrical and Electronics Engineers	Kansainvälinen tekniikan alan järjestö.
IFC	Information Flow Complexity	Koodin kytkennän mitta.
ITL	Iterative Test Last	Iteratiivinen ohjelmistokehitysmenetelmä, jossa testit kirjoitetaan välittömästä toteutuksen jälkeen.
LCOM	Lack of Cohesion of Methods	Koodin koheesion mitta.
MSN	Microsoft Network	Kokoelma Microsoftin Internet-palveluja.
NASA	National Aeronautics and Space Administration	Yhdysvaltain ilmailu- ja avaruushallinto.
NBD	Nested Block Depth	Koodin kompleksisuuden mitta.
NCLOC	Non Comment Lines of Code	Koodin määrän mitta.
TAC	Testing after coding	Sama kuin TLD.
TDD	Test-driven development, Test-driven design	Testivetoinen ohjelmointi. Ohjelmointikäytäntö, jossa testikoodi kirjoitetaan ennen varsinaista toteutusta.
TFD	Test-first design	Käytännössä sama kuin TDD.
TFC	Test-first coding	Käytännössä sama kuin TDD.
TFP	Test-first programming	Käytännössä sama kuin TDD.
TLD	Test-last development	Perinteinen ohjelmointikäytäntö, jossa testit kirjoitetaan vasta varsinaisen toteutuksen jälkeen.
TLP	Test-last programming	Sama kuin TLD.
WMC	Weighted Methods per Class	Koodin kompleksisuuden mitta.
XP	Extreme Programming	Ketterä ohjelmistokehitysmenetelmä, jossa TDD on yksi tärkeimpiä käytäntöjä.
YAGNI	You Ain't Gonna Need It	Käytäntö, joka muistuttaa ohjelmoijaa siitä, että pitää toteuttaa vain toiminnallisuuksia, jotka ovat todella tarpeen.

Sisältö

Luku 1

Johdanto

Testivetoinen ohjelmointi (engl. Test-driven development, TDD) on suosittu ohjelmoijan laatukäytäntö, jossa ohjelmoija kirjoittaa yksikkötestit pienissä osissa ennen varsinaista toiminnallisuuden toteutusta. TDD on hyvin olennainen osa suosittua ketterää ohjelmistokehitysmenetelmää nimeltä Extreme Programming (XP). TDD ei ole varsinaisesti testausmenetelmä, vaan enemmänkin suunnittelukäytäntö, joka saa ohjelmoijan ajattelemaan hieman pidemmälle ennen varsinaisen toiminnallisuuden toteutusta. Ajattelumalli on siis hieman päinvastainen kuin perinteisessä ohjelmoinnissa, jossa testit kirjoitetaan vasta jälkeenpäin. Esimerkiksi TDD:n luojana tunnettu Beck (2001) kiteyttää, miten hän koki eron TDD:n ja perinteisen ohjelmoinnin välillä:

> ”Until I started coding testing first, I had no idea how often I was coding without knowing what the right answer should be.”

Vaikka TDD:tä käytettiinkin jo 1960-luvulla NASA:n projektissa ”Project Mercury” (Larman ja Basili, 2003), ovat sen käytännön hyödyt yhä hieman hämärän peitossa. TDD:n on väitetty tuovan monia etuja verrattuna siihen, että testit kirjoitettaisiin perinteisesti vasta varsinaisen ohjelmoinnin jälkeen. Testivetoisen ohjelmoinnin on mm. väitetty tuottavan helpommin ylläpidettäviä ja virheettömämpiä ohjelmia (Astels, 2003). Osittain näiden seurauksena TDD:llä on väitetty olevan myös psykologisia ja sosiaalisia vaikutuksia mm. stressin vähentymisen ja tiimityöskentelyn paranemisen muodossa (Beck, 2002). Kaikki TDD:n puolesta puhuvat väitteet eivät kuitenkaan pohjaudu tutkittuun tietoon, vaan ne perustuvat enemmänkin näppituntumaan. Tämän työn tarkoituksena onkin etsiä vastauksia seuraaviin tutkimuskysymyksiin:

1. Mitä etuja TDD:n käytöstä on esitetty?
2. Mitä näistä väitetyistä eduista on tutkittu empiirisesti ja mitä ei?
3. Mitkä väitetyistä eduista ovat tutkimusten perusteella tarua ja mitkä totta?

Monet yritykset ovat jo ottaneet TDD:n osaksi ohjelmistokehitysprosessiaan, vaikka TDD:n todellisista eduista tiedetäänkin vielä melko vähän. Testivetoisen ohjelmoinnin omaksumista yrityksissä on tutkittu pienissä määrin. Microsoftilla lokakuussa 2006 tehdyssä kyselyssä (Begel ja Nagappan, 2007) noin 500 ohjelmistokehityksen ammattilaisesta 15 % kertoi käyttävänsä TDD:tä. Jos mukaan lasketaan myös vastanneet, jotka kertoivat käyttävänsä TDD:tä *joskus*, on luku hieman yli 50 %. Lisäksi 10 % vastanneista kertoi suunnittelevansa TDD:n käyttöä. Kuitenkin TDD oli selvästi heikointen käytetyimpiä ketterien menetelmien käytäntöjä. Hyvin samankaltaiseen lopputulokseen päätyivät myös Salo ja Abrahamsson (2008) Eurooppalaisissa yrityksissä vuosina 2004–2006 tehdyssä tutkimuksessa. Vastanneista 18 % kertoi hyödyntäneensä TDD:tä systemaattisesti tai hyvin usein projekteissaan. Mikäli mukaan lasketaan myös TDD:tä *jossain määrin* käyttäneet, on TDD:n käyttäjien osuus 52 %. Testivetoinen ohjelmointi oli kuitenkin myös tässä tutkimuksessa pariohjelmoinnin kanssa vähiten hyödynnetty ketterä käytäntö. Vuonna 2006 tehdyssä kyselyssä (Ambler, 2006), jossa haastateltiin hieman yli 4 000 IT-ammattilaista, TDD:n käyttäjien osuudeksi taas saatiin 23 %. Tutkimusten tuloksissa on siis luonnollisesti hieman vaihtelua, mutta selvästi TDD:tä käytetään teollisuudessa.

TDD:n käyttö yrityksissä on kuitenkin vielä melko vähäistä verrattuna muihin ketteriin käytäntöihin, joten sen etujen tutkiminen on mielekästä esimerkiksi tuottavuuden ja ohjelmiston laadun kannalta. Tämän työn tuloksena syntyy luettelo TDD:n hyödyistä sekä yhteenveto tähänastisista testivetoisen ohjelmoinnin empiirisistä tutkimuksista. Työn tuloksista on hyötyä esimerkiksi heille, jotka pohtivat TDD:n hyödyntämistä omassa ohjelmistokehityksessään, mutta eivät ole vakuuttuneita TDD:n esitetyistä hyödyistä. Tulokset myös toivottavasti rohkaisevat tutkimaan niitä alueita TDD:stä, joiden esitetyille hyödyille ei ole empiirisiä todisteita. Tässä työssä ei paneuduta TDD:n mahdollisiin haittoihin, vaikka niitäkin on esitetty. Esim. Beck (2001) mainitsee, että testivetoinen ohjelmointi soveltuu heikosti käyttöliittymien toteutukseen.

Aluksi tässä työssä käsitellään TDD:tä käsitteenä yleisesti. On tärkeää ymmärtää, mitä TDD on ja mitä se ei ole, jotta esitetyt edut eivät mene sekaisin toisten ohjelmointikäytäntöjen kanssa – TDD ei esimerkiksi ole pelkkää automaattista testausta, jolla on omat etunsa. TDD on ennen kaikkea suunnittelukäytäntö, johon liittyy olennaisesti testien kirjoittaminen etukäteen ja koodin refaktorointi.

Luvussa kolme esitellään kirjallisuudesta poimittuja väitteitä testivetoisen ohjelmoinnin hyödyistä. Väitteitä on kerätty pääosin alan lehtijulkaisuista ja oppikirjoista sekä ohjelmistoyritysten verkkosivuilta. Väitteiden todenperäisyyttä analysoidaan alan julkaisuista kerättyjen tutkimusten pohjalta. Empiiristä todistusaineistoa on etsitty seuraavista alan tietokannoista: ACM Digital Library, Google Scholar, IEEE Xplore, Inspec, ScienceDirect, Scopus ja SpringerLink. Hakusanoina käytettiin seuraavia sanojen ja lyhenteiden yhdistelmiä: test-driven development (TDD), test-driven design (TDD), test-first programming (TFP), test-first coding (TFC) ja test-first design (TFD). Lisäksi tutkimuksia etsittiin edellä mainitulla tavalla löydettyjen artikkelien viitteistä.

Neljännessä luvussa tehdään yhteenveto siitä, mitkä väitetyistä eduista olivat tutkitun tiedon perusteella tarua ja mitkä totta. Luvussa pohditaan myös sitä, mihin suuntaan tulevien TDD:tä käsittelevien tutkimusten tulisi edetä: mitä alueita TDD:stä olisi tärkeää tutkia enemmän ja miten tutkimukset pitäisi järjestää. Tutkimuksen lopussa olevista liitteistä löytyvät taulukot TDD:n väitetyistä eduista sekä yhteenveto oleellisimmista TDD:n empiirisistä tutkimuksista.

Luku 2

Mitä TDD on ja mitä se ei ole?

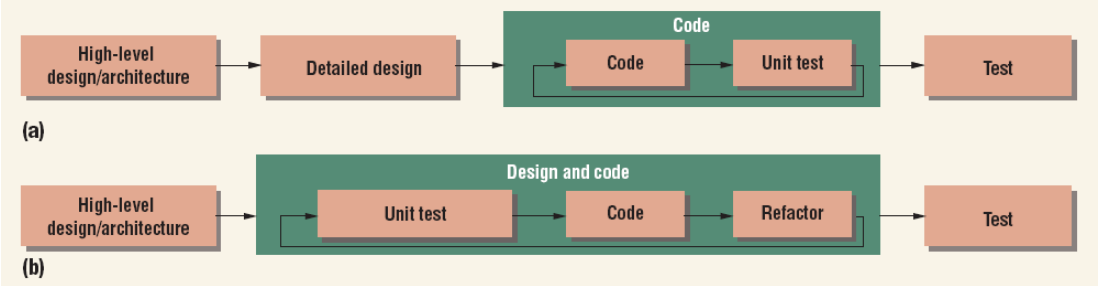
Perinteisesti ohjelmoinnissa yksikkötestit on kirjoitettu vasta varsinaisen toiminnallisuuden toteuttavan koodin jälkeen. Yksikkötestaus on siis ollut enemmänkin tapa varmistua jälkeenpäin siitä, että toiminnallisuuden toteuttava koodi toimii halutulla tavalla. Testien kirjoittamiseen ja mahdollisten virheiden korjaamiseen jälkeenpäin voi mennä muutamista minuuteista muutamiin kuukausiin.

TDD taasen kääntää yksikkötestien hyödyntämisen päälaelleen. Sen sijaan, että kaikki yksikkötestit kirjoitettaisiin vasta jälkeenpäin, testivetoisessa ohjelmistokehityksessä suunnitellaan aluksi yksikkötestit pienelle osalle ohjelman toiminnallisuutta. Vasta tämän jälkeen ohjelmoija kirjoittaa varsinaisen toiminnallisuuden toteuttavan koodin, joka läpäisee testin. Näin ohjelmoija voi ajaa yksikkötestejä jatkuvasti muokatessaan koodiaan ja saada näin välitöntä palautetta siitä, toimiiko toteutus testikoodia vastaavalla tavalla. Läpi menevät testit varmistavat, että toteutus täyttää testien kuvaamat vaatimukset.

Nimestään huolimatta TDD ei ole testaustekniikka vaan ennen kaikkea iteratiivinen suunnittelukäytäntö, joka panee ohjelmoijan suunnittelemaan toteutusta ennen yhdenkään varsinaisen koodirivin kirjoittamista. Yleinen harhaluulo on, että TDD:ssä kaikki testaaminen tapahtuu ennen kuin riviäkään koodia on kirjoitettu (Desai et al., 2008; Janzen ja Saiedian, 2008a). Tämä on luonnollisesti väärin: testivetoisessa ohjelmoinnissa testaus ja koodaus vuorottelevat nopeasti lyhyissä iteraatioissa.

Testivetoista ohjelmistokehitystä on käytetty jo 1960-luvulta lähtien, mutta nykyisen nimensä ja suosionsa se sai vasta, kun Kent Beck ja Ward Cunningham esittelivät ketterän ohjelmistokehitysmenetelmänsä nimeltä eXtreme Programming (XP) (Desai et al., 2008). TDD on olennainen osa XP:ssä hyödynnettäviä käytäntöjä. TDD:n hyödyntäminen ei kuitenkaan tarkoita, että samalla pitäisi siirtyä käyttämään XP:tä – testivetoista ohjelmointia voi soveltaa mihin tahansa

ohjelmistokehitysprosessiin.



Kuva 2.1: Perinteinen ohjelmistokehitys (a) verrattuna testivetoiseen ohjelmointiin (b). (Janzen ja Saiedian, 2008a)

TDD:n idea on esitetty kuvassa 2.1, jossa verrataan perinteisen ja testivetoisen ohjelmistokehityksen prosesseja. Jokainen ohjelmoija käyttää TDD:tä hieman omalla tavallaan, mutta hyvin vakiintuneena määritelmänä TDD:lle voidaan pitää Kent Beckin kirjassaan *Test-Driven Development By Example* esittämiä kuvauksia TDD:stä. Seuraavassa neljässä kappaleessa esitetyt kuvaukset testivetoisen ohjelmoinnin eri vaiheista pohjautuvat edellä mainittuun kirjaan (Beck, 2002). Beck kuvaa TDD:n koostuvan alla esitetyistä, jatkuvasti toistuvista vaiheista:

1. Lisää uusi yksikkötesti toteuttamattomalle toiminnallisuudelle.
2. Aja kaikki aikaisemmin lisätyt testit ja varmista että uusi testi ei mene läpi.
3. Kirjoita toiminnallisuuden toteuttava koodi.
4. Aja kaikki testit ja varmista, että ne menevät läpi.
5. Refaktoroi koodia.

Jokaisen uuden ominaisuuden lisääminen alkaa TDD:ssä uuden testin lisäämisellä. Kirjoittaakseen testikoodin ohjelmoijan täytyy ensin ymmärtää uuden ominaisuuden vaatimukset. Testikoodia kirjoittaessa ohjelmoija tulee ajatelleeksi toteutuksen toimintaa koodin käyttäjän kannalta (testikoodi) ja näin ollen keskittyy suunnittelussa ulkoisiin rajapintoihin eikä koodin sisäiseen toteutukseen. Uuden testin ei koskaan pitäisi mennä heti läpi, koska sitä vastaavaa toiminnallisuuden toteuttavaa koodia ei ole vielä kirjoitettu. Tämä testaa itse testikoodia ja varmistaa, että testi ei mene aina vahingossa läpi riippumatta siitä, miten toiminnallisuus on toteutettu.

TDD:n seuraava askel on kirjoittaa koodia siten, että uusi testi menee läpi. Koodin ei tarvitse olla täydellistä vaan riittää, että se läpäisee testin. Kirjoitetun

koodin pitää olla suunniteltu vain läpäisemään uusi testi, mitään muita tulevia toteutuksia ei pidä ennakoida. Näin ohjelmoija keskittyy vain kyseisen toiminnallisuuden toteutukseen eikä joudu miettimään kaikkea kerralla. TDD:n myöhemmät vaiheet parantavat koodin laatua. Neljäs vaihe on ajaa kaikki yksikkötestit onnistuneesti läpi. Mikäli jokin testi ei mene läpi, muokataan koodia niin, että testi menee läpi. Kun kaikki testit menevät läpi, ohjelmoija voi olla varma, että koodi toteuttaa kaikki testatut vaatimukset.

Viimeisin ja yksi tärkeimmistä vaiheista TDD:ssä on koodin refaktorointi. Tässä vaiheessa koodia muokataan laadukkaammaksi. Tarkoitus on mm. poistaa koodista turhat silmukat, yhdistellä päällekkäisyyksiä, jakaa pitkiä metodeita pienempiin osiin ja yleensäkin tuottaa helpommin ymmärrettävämpää sekä ylläpidettävämpää koodia. Aiemmin luotujen testien ansiosta ohjelmoija huomaa heti, jos koodin refaktorointi rikkoo jotain. Jatkuvalla yksikkötestien ajamisella tiedetään, missä virhe on ja tämä rohkaisee ohjelmoijaa suorittamaan isojakin muutoksia, koska testit kertovat toimiiko muutos vai ei. Tätä testaa–koodaa–refaktoroi-sykliä toistetaan TDD:ssä jokaisen uuden ominaisuuden kohdalla, kunnes halutut ominaisuudet ovat toteutettu. (Beck, 2002)

TDD:tä ei ole kuitenkaan pakko käyttää orjallisesti juuri Beckin esittämän prosessin mukaisesti. Jokainen ohjelmoija voi soveltaa sitä omien mieltymystensä mukaisesti. Jotkut lisäävät koodia hyvin pienin askelein ja refaktoroivat paljon koodia. Toiset taas saattavat kirjoittaa aluksi jo hieman täydellisempää koodia, jolloin refaktorointi jää vähemmälle. Myös testeihin voi joutua palaamaan myöhemmin, esimerkiksi lisäämään puuttuneita testejä tai refaktoroimaan vanhoja. TDD ei siis aina ole aivan niin suoraviivainen prosessi kuin edellisissä kappaleissa esitettiin.

Kuten Beck (2001) itsekin toteaa, TDD ei kuitenkaan sovellu kaikkeen eikä sitä ole aina pakko käyttää kaikessa ohjelmoinnissa – tähän se ei ole tarkoitettukaan. Joissain tapauksissa siitä voi olla enemmän haittaa kuin hyötyä, mutta joissain tapauksissa hyödyt voivat olla merkittäviä, joten se on ehdottomasti kokeilemisen arvoinen. Seuraavassa luvussa tarkastellaankin, mitä käytännön hyötyjä TDD:stä on kirjallisuudessa esitetty ja mihin väitetyt hyödyt perustuvat.

Luku 3

TDD:n hyödyt

3.1 Tutkimusten taustaa

Vaikka TDD:tä on käytetty jo vuosikymmeniä, empiirisiä tutkimuksia TDD:stä on tehty suhteellisen vähän. Tutkimukset testivetoisesta ohjelmoinnista voi jakaa pääosin kahteen kategoriaan: akateemiset tutkimukset ja teolliset tutkimukset. Akateemiset tutkimukset ovat toteutettu yleensä kontrolloidusti jonkin opiskelukurssin aikana ja tutkittavat ovat olleet opiskelijoita. Tutkimuksissa on usein mahdollista käyttää kahta tai useampaa vertailuryhmää, joista toinen toteuttaa saman järjestelmän TDD:llä ja toinen perinteisesti TLD:llä (Test-last development), jossa siis testit kirjoitetaan vasta toteutuksen jälkeen. Toteutettava ohjelma on tavallisesti melko pieni ja se toteutetaan vain opiskelukurssia varten. Akateemisiin tutkimuksiin luetaan myös tapaukset, joissa opiskelijat tekevät oikeaa projektia oikealle asiakkaalle, mutta jonkin opiskelukurssin aikana.

Sen sijaan teolliset tutkimukset ovat pääosin case-tapauksia, joissa tutkittavat ovat ammattilaisia työelämästä ja toteutettava ohjelma on konkreettinen tuote oikealle asiakkaalle. Teollisissa tutkimuksissa kontrollointi ei ole tavallisesti mahdollista niin tarkasti kuin akateemisissa tutkimuksissa. Teollisissa tutkimuksissa ei myöskään ole taloudellisista syistä yleensä mahdollista tehdä rinnan kahta täsmälleen samanlaista järjestelmää: toinen TDD:llä ja toinen perinteisesti. Vertailuryhminä on käytettävä tällöin mahdollisimman samankaltaisia toisia projekteja.

Rajanveto tutkimuksissa ei ole kuitenkaan aina näin yksinkertaista vaan osa tutkimuksista sijoittuu tähän välimaastoon. Tässä työssä näitä välimaastoon sijoittuvia tutkimuksia nimitetään puoli-teollisiksi. Puoli-teollisissa tutkimuksissa tutkittavat ovat pääosin ohjelmoijia teollisuudesta tai viimeisten vuosien opiskelijoita, jotka toteuttavat konkreettista järjestelmää oikealle asiakkaalle teollisessa

ympäristössä. Kontrollointi on kuitenkin hieman tiukempaa kuin teollisissa casetapauksissa. Tässä työssä käsitellään siis niin akateemisia, teollisia kuin puoliteollisia tutkimuksia.

Vaikka TDD onkin hyvin oleellinen osa Extreme Programming -menetelmää, tutkimuksista rajattiin pois kaikki XP:tä tutkineet julkaisut, jos näissä ei oltu keskitytty pelkästään TDD:n vaikutusten tutkimiseen. XP:ssä on myös monia muita käytäntöjä, joten on mahdotonta sanoa, johtuivatko mahdolliset positiiviset tai negatiiviset vaikutukset TDD:stä vai XP:n muista käytännöistä.

Niin test-driven development (TDD), test-driven design (TDD), test-first design (TFD), test-first programming (TFP) kuin test-first coding (TFC) tulkittiin kaikki samoiksi menetelmiksi, vaikka esimerkiksi Huang ja Holcombe (2008) väittävätkin TDD:n olevan yhtä kuin TFP + refaktorointi. Kuitenkin esimerkiksi Janzen ja Saiedian (2005), Jones (2004) sekä Madeyski ja Szala (2007) käyttävät termiä *test-first programming* synonyyminä TDD:lle. Tutkimuksista ei siis rajattu pois niitä tutkimuksia, joissa TDD:tä ei käytetty tarkasti luvussa 2 esitetyn viisivaiheisen prosessin mukaisesti. Oleellista empiiristen tutkimusten valinnassa oli se, että tavoitteena oli käyttää TDD:tä ja testaus oli tapahtunut pääosin ennen toteutusta – esimerkiksi refaktoroinnin käyttöä ei vaadittu. Tarkasti ottaen TDD ja TFP ovat eri asia, mutta koska mitään vakiintunutta termistöä ei oikeastaan ole olemassa ja käytännössä kaikista tutkimuksista on lähes mahdotonta tietää, onko ohjelmoija käyttänyt tiivistä koodaa–refaktoroi-sykliä vai onko hän kirjoittanut toteutuksen kerrasta, on rajausta lähes mahdotonta tehdä. Jokainen ohjelmoija käyttää TDD:tä hieman omalla tavallaan.

Seuraavassa luvussa esitellään kirjallisuudesta poimittuja väitteitä testivetoisen ohjelmoinnin hyödyistä. Väitteitä on kerätty pääosin alan lehtijulkaisuista ja oppikirjoista sekä ohjelmistoyritysten verkkosivuilta. Väitteiden todenperäisyyttä analysoidaan edellä mainittujen kriteerien perusteella kerättyjen alan tutkimusten pohjalta.

3.2 TDD:stä esitettyjä etuja

Kirjallisuudesta poimittuja väitteitä testivetoisen ohjelmoinnin eduista on esitetty liitteenä olevassa taulukossa A.1. Esitetyt edut on jaoteltu niiden vaikutusten mukaan, esimerkiksi liittyykö vaikutus koodin laadun, projektinhallinnan vai testauksen parantumiseen. Taulukossa esitetty viite siihen, missä väite on esitetty, ei välttämättä tarkoita sitä, että artikkelin kirjoittaja puoltaisi kyseistä väitettä vaan väite saattaa olla täysin hypoteettinen. Selkeyden vuoksi väitteen esittäjäksi on valittu vain se, jossa väite on mainittu selkeimmin, vaikka väite on saatettu

esittää useammassakin yhteydessä. Esitetyissä eduissa ei ole listattuna itsestäänselvyyksiä, kuten ”TDD luo automaattisia yksikkötestejä”. Sen sijaan, jos joku on esimerkiksi väittänyt TDD:n parantavan testien laatua, on tämä lisätty listaan. Hyvin laajat väitteet on pilkottu pienempiin osiin. Esimerkiksi Järvenpää (2006) esittää, että TDD tuottaa laadukkaampaa ohjelmakoodia. Koodin laatu on kuitenkin hyvin laaja käsite ja muualla on esitetty tarkempia väitteitä siitä, miten TDD parantaa koodin laatua. Näitä ovat mm. parempi koheesio ja pienempi kompleksisuus (Janzen ja Saiedian, 2008a), joten liian yleisen tason väitteen sijasta listaan on otettu tarvittaessa tarkempia väitteitä.

Seuraavissa aliluvuissa analysoidaan näiden kerättyjen väitteiden todenperäisyyttä alan tutkimusten pohjalta. Yhteenveto testivetoisen ohjelmoinnin tähänastisista empiirisistä tutkimuksista ja niiden päähavainnot on kerätty liitteenä olevaan taulukkoon B.1. Yhteenvedossa on esitetty tärkeimmät TDD:tä käsittelevät empiiriset tutkimukset vuosilta 2001–2008. Yhteenvetoja TDD:n empiirisistä tutkimuksista ovat koonneet myös Siniaalto ja Abrahamsson (2007), Desai et al. (2008), Jeffries ja Melnik (2007) sekä Gupta ja Jalote (2007), joita tässä tutkimuksessa tehty yhteenveto pyrkii täydentämään.

3.2.1 Vaikutus koodin sisäiseen laatuun

Vaikka TDD:n ensisijainen fokus on nimenomaan koodin sisäisen laadun parantaminen (TDD = Test-Driven *Design*), on TDD:n todellista vaikutusta koodin sisäiseen laatuun ryhdytty tutkimaan enemmän vasta viime vuosina. TDD:n puolestapuhujat väittävät testivetoisen ohjelmoinnin parantavan koodin sisäistä laatua monella eri tavalla:

1. TDD kasvattaa koodin koheesiota (engl. cohesion) (Beck, 2001).
2. TDD pienentää koodin kytkentää (engl. coupling) (Beck, 2001).
3. TDD pienentää koodin kompleksisuutta (engl. complexity) (Crispin, 2006).
4. TDD:llä tehty koodi on kompaktimpaa (engl. clean code) (Martin, 2007).

Koodin sisäistä laatua voidaan mitata monella tavalla, mutta hyvin yleisesti käytetyt mitat ovat koodin koheesio, kytkentä ja kompleksisuus. Suuri koheesio, pieni kytkentä ja pieni kompleksisuus mielletään tyypillisesti hyvän suunnittelun merkeiksi. Tällaisissa järjestelmissä luokilla on selkeä työnjako (suuri koheesio) ja niiden riippuvuus toisista luokista on pieni (pieni kytkentä), joten järjestelmät ovat helposti laajennettavissa ja paremmin testattavissa. TDD:n puolestapuhujat

väittävät testivetoisen ohjelmoinnin tuottavan myös kompaktimpaa koodia. Tällä tarkoitetaan sitä, että koodi on modulaarisempaa (pienempiä moduuleja) ja luokat sekä metodit ovat lyhyempiä: siis yksinkertaisempaa ja siistimpää koodia, jota on helpompaa muokata jälkeenpäin. Kompaktimpi koodi saattaa myös viitata pienempään kompleksisuuteen, koska mitä vähemmän koodirivejä on, sitä yksinkertaisempi järjestelmä yleensä on. Kompleksisuudella tässä työssä tarkoitetaan siis koodin sisäisen rakenteen monimutkaisuutta yleisesti – eli kompleksisuus on tässä muutakin kuin pelkästään McCaben syklomaattinen kompleksisuus (McCabe, 1976), johon kompleksisuudella usein viitataan. Esimerkiksi koodin kompaktius on tässä työssä yksi kompleksisuuden mitta.

Tutkimuksista yksikään ei viitannut selvästi siihen, että TDD:n käyttäminen olisi kasvattanut koodin koheesiota. Esimerkiksi Janzen ja Saiedian (2008a) tutkivat opiskelijoilla ja ammattilaisilla työelämästä TDD:n vaikutusta koodin sisäiseen laatuun. He eivät havainneet juuri mitään eroa koheesiossa käyttäessään mittana Henderson–Sellersin LCOM5:ttä (Lack of Cohesion of Methods). Koheesio oli TDD:llä parempi puolessa tutkituista projekteista, mutta sitä vastoin huonompi lopuissa projekteissa. Hyvin samaan tulokseen päätyivät myös Siniaalto ja Abrahamsson (2007), jotka tutkivat TDD:n vaikutuksia opiskelijoilla teollisissa olosuhteissa. Heidän tutkimuksessaan koheesio oli jopa selvästi huonompi TDD:llä verrattuna perinteiseen iteratiiviseen ohjelmistokehitykseen. Pieni koheesio antoi jopa aihetta huoleen siitä, että TDD ei välttämättä vaikuta positiivisesti ohjelmiston laatuun kokemattomien ohjelmoijien käsissä. Myös Siniaalto ja Abrahamsson (2008) sekä Müller (2006) saivat hyvin samanlaisia tuloksia: koheesio oli molemmissa tutkimuksissa pienempi TDD:tä käyttäneillä.

Jos tutkimukset TDD:n vaikutuksesta koheesioon ovat hyvin yksimielisiä, tutkimukset testivetoisen ohjelmoinnin vaikutuksista koodin kytkentään taas ovat hyvin ristiriitaisia. Siniaalto ja Abrahamsson (2007) sekä Müller (2006) havaitsivat TDD:n vaikuttavan positiivisesti kytkentään. Siniaalto ja Abrahamsson (2007) mainitsevat kuitenkin, että hajonta kytkennälle mitatuissa arvoissa oli suurinta juuri TDD:n kohdalla ja kytkentä oli melko pieni myös perinteisesti toteutetuissa projekteissa, joten on vaikea sanoa, johtuiko ero nimenomaan TDD:stä.

Janzen ja Saiedian (2006) sekä Siniaalto ja Abrahamsson (2008) taas päätyivät kytkennän suhteen hieman toisenlaisiin tuloksiin. Janzen ja Saiedian (2006) tutkivat opiskelijoilla TDD:n vaikutusta ohjelmiston sisäiseen laatuun. Kytkennän mittana käytettiin CBO:ta (Coupling between Object Classes) ja se paljasti, että kytkentä oli itse asiassa korkeampi TDD:llä kuin perinteisellä TLD-menetelmällä toteutetussa projektissa, jossa testit siis kirjoitettiin vasta varsinaisen toteutuksen jälkeen. Tulokseen saattoi tosin vaikuttaa se, että TDD:tä käyttäneet toteuttivat myös graafisen käyttöliittymän, johon TDD:n uskotaan sopivan heikosti

(Beck, 2001). Tosin kytkentä on silti korkeampi TDD:llä, vaikka käyttöliittymän osuus suljettaisiin kokonaan pois tarkastelusta. Siniaalto ja Abrahamsson (2008) taas eivät havainneet suurta eroa kytkennässä käyttäessään mittana niin ikään CBO:ta. Sitä vastoin Janzen ja Saiedian (2008a) havaitsivat TDD:n kasvattavan hieman koodin kytkentää käytettyään myös muita kytkentää mittaavia metriikoita kuten IFC:tä (Information Flow Complexity). Kyseenalaiseksi jäi kuitenkin oliko kasvanut kytkentä hyvänlaatuista vai ei. Korkeampi kytkentä kun ei kaikilla metriikoilla automaattisesti aina tarkoita, että koodin laatu olisi huonompaa.

TDD:n vaikutus koodin kompleksisuuteen on niin ikään hieman kyseenalainen. Koodin kompaktius on yksi kompleksisuuden mitta. Mitä pienempiä luokkia ja mitä lyhempiä metodeita koodissa on, sitä yksinkertaisempaa ja helpommin ymmärrettävää se yleensä on. Janzen ja Saiedian (2008a) tutkivat, miten TDD vaikuttaa koodirivien määrään. TDD:tä käyttäneet kirjoittivat selvästi pienempiä moduuleita, vähemmän metodeita luokkaa kohden ja metodit olivat keskimäärin lyhempiä kuin TLD:llä toteutetuissa projekteissa. TDD siis vaikuttaisi tuottavan luokkia ja metodeita, jotka ovat keskimäärin pienempiä sekä yksinkertaisempia. Myös Madeyskin ja Szalan (2007) tutkimus vihjaa samansuuntaisia tuloksia. Tosin aikaisemmassa tutkimuksessa (Janzen ja Saiedian, 2007) ei havaittu, että TDD:tä käyttäneet olisivat kirjoittaneet sen lyhempiä metodeita kuin TLD:tä käyttäneet.

Toinen kompleksisuuden mitta on tutkia, kuinka paljon eri polkuja koodissa on (engl. cyclomatic complexity). Mitä enemmän koodi haarautuu, sitä vaikeampaa sen ymmärtäminen ja ylläpitäminen on. Janzen ja Saiedian (2008a) havaitsivat TDD:n pienentävän koodin kompleksisuutta, kun mittana käytettiin seuraavia kompleksisuuden metriikoita: WMC (Weighted Methods Per Class), CC (Cyclomatic Complexity per method) ja NBD (Nested Block Depth). Kaikilla mitatuilla metriikoilla TDD näytti pienentävän kompleksisuutta, tosin paikoin ero oli pieni. Hyvin samankaltaisiin päätelmiin päätyivät myös Siniaalto ja Abrahamsson (2008). Tutkimuksessaan he päätyivät lopputulokseen, että TDD:llä näyttäisi olevan positiivinen vaikutus kompleksisuuteen, tosin verrokkiryhmien tulokset eivät myöskään olleet huonoja. Myös Müller (2006) havaitsi TDD:n hiukan pienentäneen kompleksisuutta.

Sen sijaan neljässä tutkimuksessa TDD:llä ei ollut positiivista vaikutusta koodin kompleksisuuteen. TDD:n vaikutuksia opiskelijoilla tutkineet Kaufmann ja Janzen (2003) sekä Siniaalto ja Abrahamsson (2007) eivät huomanneet mitään eroa koodin kompleksisuudessa TDD:n, TLD:n saati ITL:n välillä. Myöskään Janzen ja Saiedian (2006) eivät havainneet TDD:n pienentävän kompleksisuutta, päinvastoin, TDD:tä käyttäneet kirjoittivat koodia, joka oli kompleksisuudeltaan selvästi korkeampi.

Yhteenvetona TDD:n vaikutuksista koodin sisäiseen laatuun voi oikeastaan sanoa vain sen, että ei ole ollenkaan itsestään selvää, että testivetoinen ohjelmointi automaattisesti parantaisi ohjelmiston sisäistä laatua. Erot eri tutkimusten välillä saattavat hyvinkin johtua ohjelmoijien tasoeroista. Koodin sisäisen laadun mittaamisessa käytetyt mitat ovat myös hieman kiistanalaisia. Antaako pelkkä yksi luku todellisen kuvan ohjelmiston laadusta? Tosin parempiakaan objektiivisesti mitattavissa olevia menetelmiä laadun arviointiin ei oikeastaan ole, joten näihin on tyytyminen. Tutkimusten perusteella TDD ei selvästikään paranna koheesiota, mutta esimerkiksi testivetoisen ohjelmoinnin vaikutus kytkentään ja kompleksisuuteen vaativat vielä lisää tutkimuksia, jotta väitteiden todenperäisyydestä voisi tehdä johtopäätöksiä. Sen sijaan TDD:n vaikutusta koodin ulkoiseen laatuun on tutkittu enemmän, josta lisää seuraavassa luvussa.

3.2.2 Vaikutus koodin ulkoiseen laatuun ja virheiden paikallistamiseen

Testivetoisen ohjelmoinnin vaikutuksia koodin ulkoiseen laatuun ja virheiden määrään on tutkittu selvästi eniten. Koodin ulkoista laatua arvioitaessa ohjelmistoa tarkastellaan mustana laatikkona sen sijaan, että tutkittaisiin ohjelmiston sisäistä rakennetta. Ulkoista laatua mitataan yleensä löydettyjen virheiden perusteella esimerkiksi ajamalla testitapauksia tai tutkimalla käyttäjiltä saatuja virheraportteja. TDD:n on väitetty parantavan seuraavia ulkoiseen laatuun ja virheisiin liittyviä asioita:

5. TDD parantaa koodin ulkoista laatua (vähentää virheitä koodissa) (Beck, 2002).
6. TDD nopeuttaa virheiden paikallistamista ja korjausta (Martin, 2007).
7. TDD vähentää debuggausta (Martin, 2007).

TDD:ssä yksikkötestaus näyttelee hyvin keskeistä roolia. Jokaista toteutettua ominaisuutta kohti pitäisi olla yksikkötesti, koska uuden ominaisuuden lisääminen lähtee aina uuden testin lisäämisestä. TDD:tä käytettäessä projektiin syntyy laaja yksikkötestien joukko, joita ajetaan jatkuvasti. Näin suurin osa virheistä pitäisi löytyä jo hyvin aikaisessa vaiheessa projektia, jolloin niiden korjaaminen on halvempaa. Virheiden paikallistaminen ja korjaaminen pitäisi olla helppoa, koska TDD:ssä uusia ominaisuuksia lisätään pienissä osissa, jolloin ongelma voidaan rajata pieneen osaan koodia. TDD:n pitäisi näin ollen vähentää kallista ja aikaa vievää debuggausta.

Tukevatko sitten tutkimukset tätä ruusuista kuvaa TDD:stä vai ovatko väitteet pelkkää markkinointipuhetta? Maximilien ja Williams (2003) tutkivat, miten TDD:n käyttöönotto vaikutti ohjelmiston ulkoiseen laatuun IBM:llä. Tutkimuksessa raportoitiin case-tapauksesta, jossa aiemmin perinteisellä menetelmällä toteutettua kaupallista ohjelmistoa ryhdyttiin kehittämään edelleen TDD:llä. Maximilien ja Williams raportoivat virheiden määrän vähentyneen ohjelmistossa noin 40–50 % TDD:n käyttöönoton myötä. Saman ohjelmiston kehitys TDD:llä jatkui IBM:llä vielä noin neljä vuotta. Sanchez et al. (2007) IBM:ltä raportoivat, että neljän vuoden aikana virheiden määrä ohjelmistossa oli laskenut entisestään. Mitään tarkkoja lukuja ei annettu, mutta virheiden määrän kerrottiin olleen kuitenkin selvästi alle standardien.

Tässä IBM:n case-tapauksessa pitää kuitenkin ottaa huomioon se, miten ohjelmistoa kehitettiin ennen TDD:n käyttöönottoa. Maximilien ja Williams (2003) mainitsevat tutkimuksessaan, että yksikkötestaus ei ollut IBM:llä tässä kyseisessä projektissa aikaisemmin mitenkään systemaattista: liian usein yksikkötestit jäivät kokonaan kirjoittamatta ajan puutteen vuoksi. Huomattaviin parannuksiin olisi siis luultavasti päästy jo ottamalla käyttöön kurinalaisempi yksikkötestien kirjoittaminen. TDD:n käyttö oli IBM:llä myös hieman ”lepsua”. Osa ohjelmoijista kirjoitti testit ennen riviäkään varsinaista koodausta, osa kirjoittaessaan varsinaista koodia ja osa kirjoitettuaan noin puolet koodista (Sanchez et al., 2007). Näistä voi vetää johtopäätöksen, että testivetoisen ohjelmoinnin vaikutus virheiden vähentymiseen ei välttämättä ollut niin suuri, vaan suurempi vaikutus saattoi olla TDD:n sivutuotteena tulleilla yksikkötesteillä.

Bhat ja Nagappan (2006) tutkivat niin ikään TDD:n vaikutuksia koodin ulkoiseen laatuun Microsoftilla ottamalla TDD:n käyttöön kahdessa projektissa. Toinen liittyi Windowsin verkkokirjastoihin ja toinen MSN-palvelun kehittämiseen. Tutkimuksessa molempia projekteja verrattiin vastaavanlaisiin projekteihin, jotka oli tehty perinteisellä menetelmällä. Verrattaessa projekteja TDD vähensi tutkimuksen mukaan virheiden määrää jopa 62–76 %. Tosin tästä Microsoftin case-tapauksesta herää aivan sama kysymys kuin IBM:n tapauksessa: olisiko sama parannus saavutettu pelkästään yksikkötestausta parantamalla? Bhat ja Nagappan (2006) eivät mainitse, millä tavalla ja missä määrin yksikkötestausta tehtiin verrokkiprojekteissa. Tutkimusta tarkasteltaessa herääkin kysymys, käytettiinkö vertailuprojekteissa laisinkaan yksikkötestausta? Tällöin virheiden määrän vähenemistä ei voisi pitää yksistään TDD:n ansiona.

Akateemisissa tutkimuksissa Edwards (2004) päätyi hyvin samansuuruisiin lukemiin tutkittuaan 118 opiskelijalla TDD:n ja perinteisen ohjelmistokehityksen eroja. TDD:tä käyttäneet opiskelijat tuottivat koodia, jossa virheiden määrä oli keskimäärin 45 % pienempi. Tästäkään tutkimuksesta ei tosin selviä, missä mää-

rin yksikkötestausta tehtiin perinteisellä menetelmällä. Myös Yenduri ja Perkins (2006) tutkivat TDD:n vaikutuksia opiskelijoilla. Heidän tutkimuksessaan virheiden määrä väheni TDD:tä käyttäneillä keskimäärin 35 %. TDD:tä verrattiin perinteiseen vesiputousmalliseen ohjelmistokehitysmenetelmään, jossa yksikkötestit tehtiin vasta lopuksi. Tässäkin tutkimuksessa olisi voitu päästä erilaisiin lopputuloksiin, jos vesiputousmallin sijasta vertailukohteena olisi käytetty jotain iteratiivisempaa menetelmää. Niin ikään George ja Williams (2004) sekä Lui (2004) havaitsivat, että TDD:tä käyttäneet ohjelmoijat tuottivat keskimäärin vähemmän virheitä sisältävää koodia.

Pancur et al. (2003) taasen vertasivat testivetoista ohjelmointia hyvin iteratiiviseen ohjelmistokehitysmenetelmään, jossa uusia ominaisuuksia toteutettiin pienissä osissa ja yksikkötestit kirjoitettiin välittömästi toteutuksen jälkeen. Menetelmä on siis TLD:tä huomattavasti iteratiivisempi ja testien kirjoittamisen osalta kurinalaisempi. Tästä menetelmästä käytettiin nimitystä ITL (Iterative Test Last). Tutkimuksessa TDD:llä kehitetyt ohjelmistot eivät enää olleetkaan ulkoiselta laadultaan ylivoimaisia. Itse asiassa laatu oli TDD:llä jopa heikompaa – tosin ero oli hyvin pieni ja saattoi johtua silkasta sattumasta. Huimia yli 50 %:n parannuksia ei siis nähty. Tätä samaa linjaa noudattivat myös Erdogmus et al. (2005), jotka opiskelijoilla suoritetussa tutkimuksessa eivät myöskään havainneet suuria eroja virheiden määrässä TDD:n ja vertailukohteena toimineen ITL:n välillä. Samoihin lopputuloksiin päätyivät myös Huang ja Holcombe (2008) sekä Müller ja Hagner (2002). Edellä mainituissa tutkimuksissa pitää kuitenkin ottaa huomioon se, että tutkittavat olivat opiskelijoita. Esimerkiksi Erdogmus et al. (2005) huomasivat, että kokeneemmat opiskelijat olivat paljon tuottavaisempia TDD:n kanssa kuin kokemattomammat opiskelijat. Myös Pancur et al. (2003) havaitsivat, että TDD:n sisäistäminen oli opiskelijoille vaikeata. Ohjelmoijan kokemuksella saattaa siis ollut hyvin suuri merkitys tuloksissa.

Testivetoisen ohjelmoinnin vaikutusta virheiden paikallistamiseen ja korjaamiseen ei ole juuri tutkittu. Lui (2004) tutki TDD:n vaikutusta ohjelmistokehitykseen Kiinassa. Tässä case-tapauksessa havaittiin, että TDD:tä käyttäneet pystyivät korjaamaan virheet nopeammin. Testivetoista ohjelmointia käyttäneet pystyivät korjaamaan keskimäärin 97 % virheistään yhden päivän aikana, kun perinteisiä menetelmiä käyttäneet pystyivät korjaamaan vain 73 % virheistä samassa ajassa. Niin ikään TDD:n vaikutusta debuggauksen ei ole juurikaan tutkittu. Yhdessäkään tutkimuksessa ei ollut varsinaisesti mitattu, kuinka paljon aikaa ohjelmoijat käyttävät debuggaukseen TDD:llä ja ilman TDD:tä. George ja Williams (2004) toteuttivat kontrolloidun kokeen päätteeksi kyselyn, jossa 96 % ohjelmoijista kertoi TDD:n vähentäneen debuggausta. Hyvin samoihin tuloksiin päätyivät myös Kaufmann ja Janzen (2003), joiden kyselyssä opiskelijat kertoivat TDD:n selvästi auttaneen debuggauksessa.

Maalaisjärjellä ajateltuna TDD:n pitäisi vähentää debuggausta ja nopeuttaa virheiden paikallistamista ja näin ollen myös nopeuttaa virheiden korjausta. Jos ohjelmoija noudattaa testivetoisen ohjelmoinnin prosessia, ajaa hän testejä hyvin tiheässä syklissä, jolloin virhe pitäisi löytyä testien avulla muutamassa minuutissa. Virheen paikallistamiseen pitäisi harvoin tarvita debuggeria, koska ohjelmoijan pitäisi muistaa, mitä koodia hän juuri muutti ennen kun testi ei enää mennytkään läpi. TDD:tä käyttäessään ohjelmoija voi tavallisesti peruuttaa muutoksensa ja palata tilaan, jossa testi vielä meni läpi ja lähteä tätä kautta selvittämään, mikä muutos virheen aiheutti. Vaikka tämä kuulostaa hyvin loogiselta, ei tästä kuitenkaan ole niin paljoa tutkimuksia, että väitteen todenperäisyydestä voisi vetää pitkälle meneviä johtopäätöksiä. Vähentääkö TDD debuggausta esimerkiksi verrattuna ITL:ään, jossa testausta tapahtuu niin ikään hyvin paljon? Entä nopeuttaako TDD virheiden paikallistamista ylläpitovaiheessa?

Yhteenvetona TDD vaikuttaisi hieman parantavan koodin ulkoista laatua vähentämällä virheiden määrää. Tämä on hyvin luonnollinen seuraus siitä, että TDD:ssä yksikkötestaus on hyvin tiivistä, jolloin virheet pitäisi havaita aikaisessa vaiheessa. Kuinka paljon TDD sitten pienentää virheiden määrää, riippuu täysin siitä, mihin TDD:tä vertaa. Jos yksikkötestausta ei ole liiemmälti harrastettu, voi TDD vähentää virheitä jopa yli 50 %. Jos taas kurinalainen ja iteratiivinen yksikkötestaus on jo oleellinen osa ohjelmistokehitysprosessia, ei TDD välttämättä tuo kovin merkittäviä parannuksia ulkoiseen laatuun.

3.2.3 Vaikutus testeihin ja testaukseen

Vaikka TDD ei olekaan varsinaisesti testaustekniikka vaan enemmänkin suunnittelukäytäntö, on sillä väitetty olevan positiivisia vaikutuksia myös testaukseen. Kirjallisuudessa on esitetty seuraavia väitteitä:

8. Ohjelmoija kirjoittaa enemmän testejä TDD:llä (Canfora et al., 2006).
9. TDD parantaa järjestelmän testikattavuutta (Astels, 2003).
10. TDD parantaa testaustaitoja ja motivoi testaamiseen (Desai et al., 2008).

On luonnollista väittää, että testivetoinen ohjelmointi parantaisi myös testaukseen liittyviä osa-alueita, koska testien kirjoittaminen on hyvin oleellinen osa TDD:tä. Testivetoista ohjelmointia käytettäessä jokaista toteutettua ominaisuutta kohti pitäisi olla vähintään yksi testitapaus. Vaikka TDD lisäisikin testien määrää, ei se automaattisesti tarkoita sitä, että ohjelmisto olisi kattavammin testattu. Enemmän testejä tarkoittaa yleensä paremmin testattua ohjelmaa, mutta

suurempi määrä testejä ei välttämättä takaa sitä, että testit kattaisivat yhtään sen enempää koodirivejä. Tavoiteltava testikattavuus olisi 100 %, mutta tähän harvoin päästään. Teollisuudessa pyritäänkin yleensä saavuttamaan noin 80–90 %:n testikattavuus (George ja Williams, 2004).

Tutkimukset TDD:n vaikutuksesta testien määrään ovat hyvin yksimielisiä. George ja Williams (2004) sekä Geras et al. (2004) tutkivat TDD:n vaikutusta testien määrään ammattilaisilla teollisuudesta. Molemmat havaitsivat, että ohjelmoijat kirjoittivat selvästi enemmän testejä TDD:llä. Myös akateemisissa piireissä opiskelijoilla suoritetut tutkimukset tukevat näitä havaintoja. Janzen ja Saiedian (2006) havaitsivat TDD:tä käyttäneiden kirjoittavan lähes kaksi kertaa enemmän testejä koodiriviä kohden. Niin ikään Yenduri ja Perkins (2006) panivat merkille, että TDD:tä käyttäneet kirjoittivat lähes kolme kertaa enemmän testitapauksia. Myös Erdogmus et al. (2005) sekä Janzen ja Saiedian (2008a,b) päätyivät hyvin samanlaisiin lopputuloksiin. Niin ikään Canfora et al. (2006) havaitsivat TDD vaikuttaneen positiivisesti testien määrään. Ero ei tosin ollut tilastollisesti merkittävä, joten he päätyivät tulokseen, että TDD:tä käyttäneet eivät kirjoittaneet sen enempää testejä kuin perinteisiä menetelmiä käyttäneet.

Jos TDD näyttää kasvattavan testien määrää, kasvattaako se samalla järjestelmän testikattavuutta? Tämän suhteen tutkimusten tulokset eivät ole niin yksimielisiä kuin edellä. Janzen ja Saiedian (2008a) tutkivat TDD:n vaikutuksia kuudessa eri projektissa niin opiskelijoilla kuin ammattilaisilla työelämästä. Tutkituista projekteista viidessä testikattavuus oli TDD:tä käyttäneillä selvästi parempi kuin vertailuryhmillä, jotka käyttivät ITL:ää. Samanlaisiin päätelmiin päätyivät myös Siniaalto ja Abrahamsson (2007). Heidän tutkimuksessaan TDD:tä hyödyntäneillä testikattavuus oli huomattavasti parempi kaikilla kolmella kattavuusmitalla: metodikattavuus (engl. method coverage), koodikattavuus (engl. statement coverage) ja haarakattavuus (engl. branch coverage). Ero testikattavuuksissa oli huomattava, vaikka myös vertailuryhmässä testien kirjoittamista todella tapahtui. Teollisissa tutkimuksissa testikattavuus on TDD:n osalta ollut myös korkea. Esimerkiksi Microsoftilla (Bhat ja Nagappan, 2006) saavutettiin TDD:llä 79–88 %:n lohkokattavuus (engl. block coverage). Perinteisiä menetelmiä käyttäneiden vertailuryhmien kattavuuslukuja tutkimuksessa tosin ei ilmoitettu, joten vertailua ei voinut tehdä, mutta testikattavuus on silti hyvä. George ja Williams (2004) taas laskivat tutkimuksessaan testikattavuudeksi TDD:llä seuraavat luvut: 98 %:n metodikattavuus, 92 %:n koodikattavuus ja 97 %:n haarakattavuus.

Kaikissa tutkimuksissa ei kuitenkaan havaittu TDD:n parantaneen testikattavuutta. Esimerkiksi Janzen ja Saiedian (2006) havaitsivat testivetoista ohjelmointia käyttäneiden kirjoittaneen jopa testikattavuudeltaan heikompaa koodia. Tosin TDD:tä käyttäneet toteuttivat myös graafisen käyttöliittymän, johon testi-

vetoisen ohjelmoinnin uskotaan sopivan heikosti. Kun graafinen käyttöliittymä jätettiin pois vertailusta, ei TDD:n ja perinteisten menetelmien välillä ollut tilastollisesti merkittävää eroa koodikattavuudessa (engl. line coverage). Sen sijaan haarakattavuus oli TDD:llä 86 % korkeampi. Myöskään Geras et al. (2004) eivät huomanneet eroa testikattavuudessa TDD:n ja perinteisten menetelmien välillä. Testikattavuus oli tosin korkea niin TDD:tä käyttäneillä kuin ilman TDD:tä ohjelmoineilla. Hyvin samoihin lukemiin päätyivät myös Pancur et al. (2003) vertailtuaan TDD:tä iteratiiviseen ohjelmistokehitykseen. Heidän tutkimuksessaan TDD ei vaikuttanut parantaneen testikattavuutta, mutta molemmissa ryhmissä keskimääräinen testikattavuus oli hyvin korkea: 92,6 % TDD:tä käyttäneillä ja 95,1 % ITL:ää käyttäneillä.

Testien suurempi määrä ja parempi testikattavuus viittaisivat siihen, että TDD todella motivoisi ohjelmoijia paremmin testaukseen. Opiskelijoita tutkineet Huang ja Holcombe (2008) havaitsivat tutkimuksessaan, että TDD:tä käyttäneet ohjelmoijat käyttivät paljon enemmän aikaa testaukseen ja vähemmän aikaa varsinaiseen koodaukseen kuin perinteistä TLD-menetelmää käyttäneet. Osasyynä tähän tulokseen saattoi tosin olla se, että tukittavat olivat opiskelijoita, joilla ei ollut aikaisempaa kokemusta TDD:stä, joten aika on saattanut mennä yksinkertaisesti TDD:n opiskeluun.

Entä sitten TDD:n vaikutus testaustaitoihin? Voisiko TDD todella parantaa ohjelmoijien testaustaitoja? Janzen ja Saiedian (2008a) tekivät mielenkiintoisen havainnon tutkimuksessaan, jossa he vertasivat TDD:tä iteratiiviseen ohjelmistokehitysmenetelmään (ITL) siten, että ohjelmoijat toteuttivat ensin osan ohjelmaa ITL:llä ja sitten loput TDD:llä. Sama toistettiin toisilla ohjelmoijilla toisinpäin, eli ensin puolet ohjelmasta tehtiin TDD:llä ja sitten loput puolet ITL:llä. Tutkittaessa teollisuudesta tulleiden ohjelmoijien testikattavuutta viimeisessä ITL-vaiheessa paljastui, että heidän testikattavuutensa oli jopa parempi kuin sitä edeltäneessä TDD-vaiheessa. Tämä ilmiö esiintyi siis vain tapauksessa, jossa TDD:tä seurasi ITL-vaihe – testikattavuus ei ollut parempi ITL-vaiheessa silloin, kun ITL edelsi TDD:tä. Tästä herääkin kysymys, voiko TDD:n käytöllä olla jokin testaustaitoja parantava vaikutus? Janzen ja Saiedian havaitsivat täsmälleen saman ilmiön myös opiskelijoilla toisessa tutkimuksessaan (Janzen ja Saiedian, 2008b). Mikäli TDD todella parantaisi testaustaitoja, olisi mielenkiintoista tietää, kuinka kauan sen vaikutus kestää.

Selvästikään väitteet testien määrän ja testikattavuuden paranemisesta TDD:n avulla eivät ole aivan tuulesta temmattuja. Yllä esitettyjen tutkimusten perusteella vaikuttaisi siltä, että testivetoinen ohjelmointi saattaa tosiaankin kasvattaa testien määrää ja samalla parantaa ohjelmiston testikattavuutta. Lähes kaikissa testikattavuutta mitanneissa tutkimuksissa testikattavuus oli TDD:llä melko

korkea myös niissä tutkimuksissa, joissa TDD:llä ei havaittu olevan vaikutusta testikattavuuteen. Sen sijaan TDD:n vaikutus testaustaitojen parantamiseen ja testauksen motivointiin kaipaisi lisää tutkimuksia. Suurempi testikattavuus kyllä viittaisi siihen, että ohjelmoijilla on myös parempi motivaatio testaukseen ja osa tutkimuksista viittaisi myös siihen suuntaan, että TDD:llä voisi olla testaustaitoja parantava vaikutus.

3.2.4 Vaikutus tuottavuuteen ja projektin hallintaan

Tuottavuus on laadun ohella yksi tärkeimmistä tekijöistä, kun halutaan parantaa ohjelmistokehitysprosessia. Parempilaatuinen koodi on tietysti aina hyvästä, mutta mikäli laadun parantaminen vaatii liian suuria menetyksiä tuottavuudessa, eivät projektit pysy aikataulussa ja rahaa palaa. Suurempi tuottavuus merkitsee yrityksille aina rahan säästöä ja sitä kautta suurempia liikevoittoja. Mikäli TDD huomattavasti heikentäisi ohjelmoijan tuottavuutta, se tuskin koskaan tulisi saamaan kovin suurta jalansijaa yritysmaailmassa.

TDD:ssä on kuitenkin monia sellaisia vaiheita, joilla on potentiaalia parantaa tuottavuutta. Esimerkiksi testin kirjoittaminen etukäteen vaatii, että ohjelmoija todella ymmärtää ongelman. Tämän pitäisi vähentää väärinymmärryksiä ja sitä myötä vähentää uudelleentyötä. Testien ja toteutusten kirjoittaminen pienissä osissa taas pitäisi auttaa keskittymään ja rajaamaan ongelmaa paremmin: kaikkea ei tarvitse miettiä kerralla. Tälläkin saattaa olla positiivinen vaikutus tuottavuuteen. Tiivis testien ajaminen taas johtaa aikaa vievän debuggauksen vähentymiseen, joka niin ikään parantaa tuottavuutta. Mutta tapahtuuko näin todellisuudessa? Kirjallisuudessa on esitetty seuraavia väitteitä testivetoisen ohjelmoinnin hyödyistä liittyen tuottavuuteen ja projektin hallintaan, joiden todenperäisyyttä analysoidaan seuraavaksi:

11. TDD parantaa ohjelmoijan tuottavuutta (Jeffries ja Melnik, 2007).
12. TDD parantaa tehtävien työmääräarvioita (Canfora et al., 2006).
13. TDD parantaa projektin seurantaa (Lui, 2004).
14. TDD vähentää ”turhien” ominaisuuksien toteuttamista (Rendell, 2008).
15. TDD vaatii vähemmän uudelleen työtä (Jeffries ja Melnik, 2007).

Tuottavuutta voidaan mitata usealla eri tavalla. Tuottavuuden indikaattorina voidaan käyttää esimerkiksi projektiin käytettyä kokonaisaikaa. Jos kokonaisaika

kasvaa, tuottavuus heikkenee. Hyvin usein mittana käytetään kuitenkin tuotettujen koodirivien määrää aikayksikköä kohden, esimerkiksi NCLOC/h (Non Comment Lines of Code / h). Mikäli ohjelmoija tuottaa enemmän koodirivejä tunnissa, voidaan häntä pitää tuottavampana kuin henkilöä, joka kirjoittaa vähemmän koodirivejä tunnissa. Tuottavuuden pieneneminen ei kuitenkaan välttämättä tarkoita sitä, että käytetty menetelmä olisi taloudellisesti huonompi vaihtoehto. Jos pienellä tuottavuuden heikentymisellä saadaan aikaiseksi parempilaatuista koodia, maksaa heikentynyt tuottavuus itsensä takaisin esimerkiksi helpompana ylläpitona. Laatu pitää siis ottaa huomioon tuottavuutta tarkasteltaessa.

Kaufmann ja Janzen (2003) vertailivat TDD:n ja TLD:n eroja opiskelijoilla. Tutkimuksessa kävi ilmi, että TDD:tä käyttäneet opiskelijat kirjoittivat 50 % enemmän koodia, joka siis voisi viitata siihen, että TDD:tä hyödyntäneet olivat paljon tuottavaisempia. Tutkimuksessa ei tosin mainittu, paljonko aikaa tähän 50 %:n parannukseen saavuttamiseen tarvittiin. Erdogmus et al. (2005) havaitsivat myös, että TDD:tä käyttäneet opiskelijat olivat hieman tuottavaisempia kuin vertailuryhmänä toimineet ITL:ää käyttäneet, kun mittana käytettiin toteutettujen ominaisuuksien määrää suhteutettuna projektiin käytettyyn kokonaisaikaan. Kasvaneen tuottavuuden arveltiin johtuneen siitä, että testien kirjoittaminen etukäteen parantaa tehtävän ymmärtämistä ja vaatii näin myöhemmin vähemmän uudelleentyötä.

Samankaltaisiin tuloksiin päätyivät myös Janzen ja Saiedian (2006). Verrattuna TLD-menetelmään TDD:llä ohjelmoineet opiskelijat käyttivät keskimäärin 57 % vähemmän aikaa yhden ominaisuuden toteuttamista kohden. Tutkimuksessa pohdittiin, josko ero olisi voinut ymmärrettävästi johtua ohjelmoijien tasoeroista, mutta ennen tutkimusta teetetyssä kyselyssä ei havaittu tilastollisesti merkittäviä eroja eri vertailuryhmien välillä. Niin ikään Gupta ja Jalote (2007) tutkivat TDD:n vaikutusta tuottavuuteen opiskelijoilla. Tutkimuksessa TDD:tä hyödyntäneet käyttivät selvästi vähemmän aikaa projekteissa kokonaisuudessaan. Kun taas tuottavuuden mittana käytettiin koodirivien määrää henkilötyötuntia kohden, ei ero vertailuryhmään ollutkaan enää niin suuri, mutta oli silti hieman suurempi TDD:llä. Tutkimusten tulokset siis vihjaavat siihen suuntaan, että TDD:llä olisi positiivinen vaikutus tuottavuuteen. Myös muut opiskelijoilla tehdyt tutkimukset tukevat tätä havaintoa (Huang ja Holcombe, 2008; Yenduri ja Perkins, 2006).

Kaikissa edellä mainituissa tutkimuksissa, joissa TDD:n havaittiin vaikuttaneen positiivisesti ohjelmoijan tuottavuuteen, tutkittavat olivat opiskelijoita. Löydetyistä tutkimuksista vain yksi oli sellainen, jossa TDD:n havaittiin kasvattaneen tuottavuutta, kun tutkimuskohteena oli ammattilainen teollisuudesta. Madeyski ja Szala (2007) keskittyivät tutkimuksessaan nimenomaan testivetoisen ohjelmoinnin vaikutuksiin tuottavuuteen. Tutkimuksessa vertailtiin yhden yritys-

maailmasta tulleen ohjelmoijan tuottavuutta TDD:llä ja TLD:llä. Tutkimus oli jaettu kolmeen vaiheeseen: ensin ohjelmoija toteutti kolmanneksen ohjelmistosta TLD:llä, sitten toisen kolmanneksen TDD:llä ja viimeiseksi loput taas TLD:llä. Verrattuna ensimmäiseen TLD-vaiheeseen tuottavuus oli usealla eri mittarilla selkeästi parempi TDD-vaiheessa. Kun taas verrattiin TDD-vaihetta ja viimeistä TLD-vaihetta, ei tuottavuus ollutkaan enää parempi TDD-vaiheessa. Madeyski ja Szala arvelivat, että projektin edetessä ohjelmoija alkoi ymmärtää ongelmaa paremmin ja sen vuoksi tuottavuus kasvoi viimeisessä TLD-vaiheessa. Ero tuottavuudessa ei kuitenkaan ollut niin suuri kuin ensimmäisen TLD-vaiheen ja TDD-vaiheen välillä. On syytä ottaa huomioon myös se, että tutkimuksessa tutkittiin vain yhtä henkilöä, tosin hänen koettiin edustavan tyypillistä teollisuuden ohjelmoijaa.

Jos useassa tutkimuksessa oltiin sitä mieltä, että TDD parantaa ohjelmoijan tuottavuutta, vähintään yhtä monta tutkimusta päätyi siihen, että TDD ei välttämättä parannakaan ohjelmoijan tuottavuutta. Maximilien ja Williams (2003) raportoivat testivetoisen ohjelmoinnin käyttöönotosta IBM:llä. Useat kehittäjät ja johtohenkilöt olivat aluksi huolissaan siitä, että TDD:n tiukka prosessi heikentäisi tuottavuutta niin paljon, että projektin aikataulut venyisivät. Lopulta projekti kuitenkin pysyi aikataulussaan ja tutkimuksessa arvioitiin, että TDD:n käyttöönotto pienensi tuottavuutta vain hyvin vähän. Microsoftilla tehdyssä tutkimuksessa (Bhat ja Nagappan, 2006; Nagappan et al., 2008) havaittiin myös tuottavuuden hieman laskeneen. Projektin johto arvioi, että projektiin käytetty kokonaisaika kasvoi noin 15–35 %. Pitää kuitenkin ottaa huomioon, että molemmissa tutkimuksissa ohjelmiston laatu parani selvästi. Pienentynyt tuottavuus ei näissä tapauksissa siis välttämättä ole ollenkaan huono asia, jos samalla saavutetaan huomattavia parannuksia laatuun, jotka maksavat itsensä takaisin myöhemmin esimerkiksi helpompana ylläpitona.

George ja Williams (2004) havaitsivat, että TDD:tä hyödyntäneet teollisuuden ammattilaiset käyttivät tutkittuun projektiin keskimäärin 16 % enemmän aikaa kuin vertailukohteena toimineet TLD:tä käyttäneet ohjelmoijat. Ohjelmiston laatu oli kuitenkin tässäkin tutkimuksessa keskimäärin parempaa TDD:llä mm. testikattavuuksien osalta. Tutkimuksessa havaittiin hienoinen korrelaatio käytetyn ajan ja ohjelmiston laadun välillä. Lisääntynyt ajankäyttö johtui siis todennäköisesti siitä, että ohjelmoijat tekivät laadukkaampaa koodia. Asian voi myös ajatella toisinpäin: parantunut laatu ei välttämättä johtunut TDD:stä itsestään vaan yksinkertaisesti enemmän käytetystä ajasta. Myös Canfora et al. (2006) päätyivät siihen tulokseen, että TDD vaatii selvästi enemmän aikaa kuin TLD tutkittuaan TDD:tä yksikkötestien kirjoittamisen näkökulmasta.

Sen sijaan Siniaalto ja Abrahamsson (2007) havaitsivat TDD:n parantaneen testi-

kattavuutta, mutta eivät havainneet TDD:llä olleen vaikutusta ohjelmoijan tuottavuuteen suuntaan tai toiseen. Myöskään Geras et al. (2004), Janzen ja Saiedian (2008b) sekä Müller ja Hagner (2002) eivät havainneet merkittäviä eroja tuottavuudessa TDD:n ja perinteisten menetelmien välillä.

Vaikuttaisikin siis siltä, että TDD tuskin saa aikaan mitään radikaaleja muutoksia tuottavuudessa suuntaan tai toiseen. Tuottavuus ei liioin heikkene eikä myöskään parane. Jos vertailuun ottaa kuitenkin mukaan myös ohjelmiston laadun, joka TDD:llä vaikuttaisi hieman parantuvan, voidaan sanoa, että testivetoisen ohjelmoinnin kokonaisvaikutus tuottavuuteen suhteessa laatuun on todennäköisesti positiivinen. Jos ohjelmiston laatua saadaan parannettu minimaalisin vaikutuksin tuottavuuteen, on TDD jo silloin kokeilemisen arvoinen. Tutkimukset näyttivät kallistuvan lisäksi siihen suuntaan, että tuottavuus saattaa parantua enemmän opiskelijoilla kuin kokeneimmilla ohjelmoijilla. Johtuuko tämä sitten pelkästä sattumasta, vai pystyvätkö opiskelijat sisäistämään TDD:n paremmin? Tähän on vaikea antaa mitään tyhjentävää vastausta.

Muutamassa tutkimuksessa (Canfora et al., 2006; Geras et al., 2004; Lui, 2004) tehtiin mielenkiintoinen havainto liittyen tehtävien työmääräarvioihin. Geras et al. (2004) huomasivat, että tehtävään todellisuudessa käytetty aika suhteessa arvioituun aikaan vaihteli vähemmän TDD:tä käyttäneillä. Tämä voisi merkitä sitä, että työmääräarviot TDD:llä pitävät paremmin paikkansa. Projektin aikataulun kannalta tämä olisi erittäin positiivinen asia, koska tällöin aikataulu olisi paremmin ennustettavissa. Myös Canfora et al. (2006) päätyivät tutkimuksessaan samoihin johtopäätöksiin. He arvioivat, että tarkemmat työmääräarviot voivat johtua siitä, että TDD:ssä jokaista toteutettua toiminnallisuutta kohden pitäisi olla yksikkötesti. Testejä pitäisi siis muodostua suunnilleen sama määrä riippumatta ohjelmoijasta. Sen sijaan käytettäessä perinteistä TLD-menetelmää, jossa testit kirjoitetaan vasta toteutuksen jälkeen, testauksen määrä voi vaihdella huomattavasti. Toiset saattavat käyttää testaukseen huomattavasti enemmän aikaa kuin toiset. Tällöin myös tehtäviin käytetty aika vaihtelee enemmän.

Tätä samaa havaintoa tukee myös Kiinassa tehty tutkimus TDD:n vaikutuksista (Lui, 2004). Lui raportoi, että testivetoisen ohjelmoinnin käyttöönoton jälkeen tehtävien työmääräarviot paranivat selvästi. Hän arveli tämän johtuvan siitä, että kokemattomat ohjelmoijat tyypillisesti aliarvioivat tehtävien työmääriä ja sen seurauksena vähentävät testausta lopussa tai jopa jättävät testauksen kokonaan tekemättä. Tämä luonnollisesti johtaa siihen, että tehtäviin joudutaan todennäköisesti palaamaan piakkoin uudestaan virheiden korjausten merkeissä. Lui arvioi TDD:n parantaneen asiaa siten, että kehittäjän on helpompaa arvioida, kuinka kauan aikaa kuluu yhden yksikkötestin kirjoittamiseen ja tämän testin läpäisevään toteutukseen kuin lähteä liikkeelle toteutukseen kuluvan ajan arvioinnista.

Lui myös havaitsi, että TDD auttoi arvioimaan sitä, kuinka valmiita tehtävät ovat. Perinteisiä menetelmiä käytettäessä Lui (2004) kertoo seuraavista ongelmista:

> ”There is little value in the report of an inexperienced programmer that his or her team has completed 40% of its coding. There are a number of reasons for this: (i) the code cannot be executed since it is incomplete; (ii) even if the 40% of code were completed in four days, that isn't to say that the remaining 60% can be completed in the next six days; (iii) the inexperienced programmers are not sure of how many lines the program will ultimately require, so the 40% is just a guess; (iv) the report does not include the progress of testing, so even a report of 100% done is not useful as the code still has to be tested.”

Lui (2004) arvioi, että TDD tuo tähän huomattavan parannuksen, koska testit käytännössä kertovat, missä vaiheessa tehtävä on. Jos ominaisuuksista puolelle on olemassa yksikkötestit ja niitä vastaavat toteutukset menevät läpi testeistä, on tehtävä arviolta puoliksi valmis. Ohjelmoijat pystyvät näin antamaan paljon objektiivisempia arvioita siitä, missä vaiheessa tehtävät ovat. Vaikka tämä ei välttämättä tarkasti kerrokaan, kauanko tehtävän valmistumiseen menee aikaa, antaa se silti projektin johdolle paremman kuvan siitä, missä mennään.

Tuottavuuden, projektin seurannan ja työmäärien arvioinnin parantumisten lisäksi TDD:n on myös väitetty vähentävän ”turhien” ominaisuuksien toteuttamista (Rendell, 2008). YAGNI (You Ain't Gonna Need It) on käytäntö, joka jokaisen ohjelmoijan tulisi pitää aina mielessä. Vaikka jokin ominaisuus tuntuisi kuinka hienolta ja trendikkäältä, ei sitä ole syytä toteuttaa, jos sille ei ole todellista tarvetta. Loppukäyttäjälle näistä ominaisuuksista ei välttämättä ole mitään hyötyä, jolloin niiden toteuttamiseen kulutetaan vain turhaa aikaa. Miten TDD sitten auttaisi näiden ”turhien” ominaisuuksien vähentämisessä? Yksikään tutkimus ei vastannut tähän kysymykseen. Väite ei kuitenkaan ole täysin hatusta vetäisty. TDD:n ylhäältä alas -periaate voi hyvinkin auttaa tähän ongelmaan. TDD:ssä uuden ominaisuuden lisääminen lähtee aina siitä, että jollekin korkeamman tason vaatimukselle lisätään testitapaus. Koska mitään muuta koodia kuin kyseisen testitapauksen läpäisevää koodia ei pitäisi kirjoittaa, olisi hyvin loogista ajatella, että tämä estää turhien ominaisuuksien hiipimisen ohjelmistoon.

TDD:n on myös väitetty vähentävän uudelleentyötä (Jeffries ja Melnik, 2007). Vaikka yksikään tutkimus ei tätä varsinaisesti tutkinut esimerkiksi mittaamalla uudelleentyöhön käytettyä aikaa, on tässäkin väitteessä selvästi jotain todellisuusperää. On hyvin loogista päätellä, että koska TDD vaikuttaisi vähentävän

virheiden määrää, vähentäisi se myös sitä uudelleentyön määrää, joka syntyy, kun joudutaan palaamaan takaisin koodin ääreen korjaamaan virheitä. Toisaalta taas refaktorointi on hyvin oleellinen osa TDD:tä ja se saattaa jopa lisätä uudelleentyötä. Kuten väite turhien ominaisuuksien vähenemisestä myös tämä väite tarvitsisi taakseen tutkimuksia, jotta voisi sanoa pitääkö väite paikkaansa vai ei.

3.2.5 Vaikutus ylläpidettävyyteen ja laajennettavuuteen

Testivetoisen ohjelmoinnin on väitetty tuovan seuraavia ohjelmiston ylläpidettävyyteen ja laajennettavuuteen liittyviä etuja:

16. TDD tuottaa helpommin ylläpidettävää koodia (Astels, 2003).
17. TDD vähentää ylläpitokuluja (Beck, 2002).
18. TDD parantaa koodin uudelleenkäytettävyyttä (Janzen, 2005).
19. TDD parantaa koodin laajennettavuutta (Martin, 2007).
20. TDD parantaa koodin integroitavuutta (Järvenpää, 2006).
21. TDD tuottaa luotettavampia ja vakaampia järjestelmiä (Astels, 2003).

TDD:n vaikutukset ylläpidettävyyteen ja laajennettavuuteen riippuvat paljon siitä, miten TDD vaikuttaa koodin sisäiseen ja ulkoiseen laatuun. Ulkoiselta laadultaan paremmat järjestelmät ovat luonnollisesti helpommin ylläpidettävissä. Mitä vähemmän virheitä koodissa on ja mitä suurempi testikattavuus testeillä on, sitä helpompaa järjestelmään on tehdä muutoksia. Koska virheiden korjaaminen ylläpitovaiheessa on paljon kalliimpaa kuin ohjelmiston kehityksen alkuvaiheessa, on luonnollista ajatella, että TDD myös vähentää ylläpitokuluja. Näin ollen virheiden määrän vähentyminen ja testikattavuuden parantuminen johtavat helpompaan ylläpitoon ja lopulta ylläpitokulujen vähentymiseen.

Ulkoisen laadun ohella myös ohjelmiston sisäinen laatu vaikuttaa ylläpidon helppouteen. Koodia on helpompi ymmärtää ja muutosten tekeminen on helpompaa, jos luokkajako on selkeä. Sanchez et al. (2007) raportoivat TDD:n käytöstä IBM:llä neljän vuoden aikana. Heidän havaintojensa mukaan TDD näyttäisi hillitsevän koodin kompleksisuuden kasvua ohjelmiston kehittyessä eteenpäin. Tämä siis vahvistaisi käsitystä, että TDD helpottaa ylläpitoa. Toisaalta taas Siniaalto ja Abrahamsson (2008) havaitsivat, että TDD saattaa kasvattaa pakettien välisiä riippuvuuksia, jonka johdosta niiden muuttaminen ja ylläpito voi olla hankalampaa. Koska kerättyjen tutkimusten pohjalta ei voinut vetää johtopäätöksiä siitä,

parantaako TDD koodin sisäistä laatua, ei TDD:n vaikutuksista ylläpidettävyyteen voi sanoa juuri mitään koodin sisäisen laadun osalta. Väitteet koodin paremmasta uudelleenkäytettävyydestä, laajennettavuudesta ja integroitavuudesta riippuvat niin ikään paljon siitä, onko TDD:llä kirjoitettu koodi sisäiseltä laadultaan parempaa.

Yhdessäkään tutkimuksessa ei varsinaisesti keskitytty tutkimaan TDD:n pitkäaikaisia vaikutuksia, mutta paremman testikattavuuden ja virheiden vähentymisen valossa voidaan väittää TDD:n tuottavan luotettavampia ja vakaampia järjestelmiä, joten väite, jonka Astels (2003) esitti, voidaan sanoa pitävän paikkansa.

3.2.6 Vaikutus dokumentointiin ja ymmärrettävyyteen

Testivetoisen ohjelmoinnin on esitetty tuovan seuraavia etuja:

22. TDD parantaa koodin toiminnan ymmärtämistä (Martin, 2007).
23. TDD auttaa ymmärtämään järjestelmän vaatimuksia paremmin (Jeffries ja Melnik, 2007).
24. TDD pystyy vastaamaan paremmin vaatimusten muutoksiin (Wasmus ja Gross, 2007).

Noin puolet ohjelmoijien ajasta järjestelmän ylläpitovaiheessa on väitetty kuluvan ohjelman koodin ymmärtämiseen (Corbi, 1989). Mikäli TDD parantaisi ohjelmiston ymmärtämistä kooditasolla, voisi tästä siis olla paljon taloudellista etua ohjelmiston ylläpidossa. Testivetoisen ohjelmoinnin vaikutuksia ohjelmiston ymmärtämiseen ei kuitenkaan juuri tutkimuksissa käsitelty. Vain Müller ja Hagner (2002) raportoivat, että TDD voisi parantaa järjestelmän ymmärtämistä metodien paremman uudelleenkäytön muodossa.

Loogisesti ajateltuna TDD:n sivutuotteena syntyvät yksikkötestit saattavat parantaa ohjelmiston ymmärtämistä. Yksikkötestithän toimivat ikään kuin ohjelmiston ohjekirjana: ne kertovat miten järjestelmän pitää käyttäytyä. Jos ohjelmoija haluaa esimerkiksi tietää, miten jokin tietty olio (engl. object) luodaan, pitäisi tämä löytyä yksikkötesteistä. Jos taas tarvitsee tietää, miten jotain metodia pitää kutsua, selviää tämäkin tarkastelemalla kyseisen metodin yksikkötestiä. Testivetoinen ohjelmointi siis rohkaisee ohjelmoijia dokumentoimaan yksikkötestien avulla, miten järjestelmä toimii. Tämän pitäisi auttaa toisia ohjelmoijia ymmärtämään koodia paremmin. Toisaalta tämä on enemmänkin yksikkötestauksen tuoma etu kuin TDD:n, joten kurinalaista yksikkötestausta käyttäville TDD ei

välttämättä tässä tuo mitään parannuksia ohjelman ymmärtämiseen. Jotta voisi sanoa TDD:n todella parantavan järjestelmän ymmärrettävyyttä, pitäisi tätä tutkia enemmän empiirisesti.

Yksikään tutkimus ei myöskään tuonut vastausta siihen, auttaako TDD ymmärtämään järjestelmän vaatimuksia paremmin saati pystyykö TDD vastaamaan paremmin vaatimusten muutoksiin. Ketterät menetelmät, kuten Scrum, todennäköisesti pystyvät parantamaan näitä näkökohtia paremmin.

3.2.7 Psykologiset vaikutukset

Testivetoisen ohjelmoinnin on väitetty vaikuttavan psykologisesti seuraavilla tavoilla:

25. TDD parantaa ohjelmoijan luottavaisuutta koodinsa laatuun (Beck, 2002).
26. TDD antaa enemmän luottamusta muokata koodia jälkeenpäin (Beck, 2002).
27. TDD parantaa ohjelmoijan kuria (Lui, 2004).
28. TDD kasvattaa ohjelmoijan työstä saamaansa tyydytystä (Beck, 2002).
29. TDD vähentää ohjelmoijan stressiä (Jeffries ja Melnik, 2007).
30. Asiakaan odotukset täyttyvät todennäköisemmin TDD:llä (Wasmus ja Gross, 2007).
31. TDD on nautinnollisempaa kaikille sidosryhmille (Wasmus ja Gross, 2007).

Alla oleva lainaus kuvastaa hyvin tyypillistä asennetta ohjelmoijien piirissä:

> ”Why don’t developers clean up code? They’re afraid that they’ll break it. The old maxim, ’If it ain’t broke, don’t fix it!’ is a common attitude among software developers.” (Martin, 2007)

Muutosten tekemistä siis vältetään, koska ohjelmoijilla ei ole riittävästi luottamusta muokata koodia – oli se sitten ohjelmoijan itsensä kirjoittamaa tai jonkun muun. Pienelläkin muutoksella saattaa olla suuret vaikutukset. Testivetoinen ohjelmointi vaikuttaisi kuitenkin tuovan hieman parannusta tähän luottamuspulaan. TDD:n vaikutusta ohjelmoijan luottavaisuuteen ei voi oikeastaan muutoin mitata kuin kyselyjen avulla. Edwards (2004) suorittikin kyselyn TDD:tä kokeilleilla opiskelijoilla. Vastanneista 65,3 % kertoi TDD:n parantaneen luottamusta

oman koodin oikeellisuuteen ja 67,3 % kertoi TDD:n parantaneen luottamusta muokata koodia jälkeenpäin. Myös muutamassa muussa kyselyssä havaittiin TDD:llä olleen hienoinen positiivinen vaikutus ohjelmoijien oman koodin luottavuuteen (Janzen ja Saiedian, 2008b; Kaufmann ja Janzen, 2003; Kollanus ja Isomöttönen, 2008; Müller ja Tichy, 2001). Sen sijaan Gupta ja Jalote (2007) päätyivät tutkimuksessaan siihen tulokseen, että TDD:tä käyttäneet olivat vähemmän luottavaisempia koodin designin laatuun kuin perinteisiä menetelmiä käyttäneet. Tämän arveltiin johtuvan siitä, että perinteisiä menetelmiä käytettäessä panostetaan ehkä TDD:tä enemmän koodin designin suunnitteluun ennen varsinaista ohjelmointia. Sen sijaan TDD:ssä ohjelmointi aloitetaan nopeammin lisäämällä testejä ja samalla design muodostuu ajan myötä selkeämmäksi.

Luottamuksen kasvu oman koodin laatuun ja kasvanut luottamus muokata koodia myöhemmin johtuvat luultavasti siitä, että TDD:n parantaessa testikattavuutta, ohjelmoijat luottavat siihen, että mahdolliset virheet löytyvät testejä ajamalla. Mitä enemmän testejä, sitä luottavaisempia ohjelmoijat ovat koodin laatuun ja sitä paremmin he uskaltavat tehdä siihen muutoksia.

Entä parantaako TDD ohjelmoijan kuria? Pystyvätkö ohjelmoijat noudattamaan TDD:llä paremmin sovittuja menetelmiä? Lui (2004) raportoi TDD:n käyttöönotosta Kiinassa. TDD:n koettiin parantaneen ohjelmoijan kuria siinä muodossa, että prosessia noudatettiin paremmin. Tutkimuksen mukaan kokemattomat ohjelmoijat eivät välttämättä seuraa sovittuja käytäntöjä vaan työskentelevät omalla tavallaan. Esimerkiksi yksikkötestejä ei välttämättä kirjoiteta, vaikka pitäisi. Tutkimuksen mukaan TDD:n käyttöönoton jälkeen ohjelmoijat kirjoittivat testejä paljon kurinalaisemmin. Toisaalta taas Abrahamsson et al. (2005) havaitsivat, että ohjelmoijat olivat hyvin haluttomia käyttämään TDD:tä vaikka heitä oli tähän neuvottu. Lopulta TDD:tä ei juuri käytetty ja testikattavuus jäi niinkin pieneksi kuin 7,8 %. Näin pieni testikattavuus on kaukana kurinalaisesta yksikkötestauksesta.

TDD saattaa siis tuoda parannusta kurinalaisempaan yksikkötestaukseen, mutta tätä näkökulmaa on tutkittu niin vähän, että johtopäätöksiä on vaikea vetää. Niin ikään TDD:n muiden psykologisten vaikutusten todenperäisyydestä on vaikea sanoa mitään, koska näitä alueita ei ole empiirisesti tutkittu. Monet väitteistä, kuten asiakaan odotusten täyttyminen todennäköisemmin TDD:llä, vaikuttaisivat olevan enemmänkin ketterien menetelmien tuomia etuja (esim. XP ja Scrum) kuin pelkästään TDD:n.

3.2.8 Sosiaaliset vaikutukset

Psykologisten vaikutusten ohella testivetoisen ohjelmoinnin on väitetty vaikuttavan positiivisesti ohjelmoijien sosiaaliseen kanssakäymiseen:

32. TDD parantaa suhteita työtovereihin (Beck, 2002).
33. TDD parantaa ohjelmoijien välistä kommunikointia (Crispin, 2006).
34. TDD parantaa ohjelmoijien ja bisnesihmisten välistä kommunikointia (Crispin, 2006).

Yksikään TDD:tä käsittelevä tutkimus ei tutkinut TDD:n sosiaalisia vaikutuksia. Kuitenkin esimerkiksi Beck (2002) antavaa hyvin ruusuisen kuvan testivetoisen ohjelmoinnin psykologisista ja sosiaalisista vaikutuksista väitteen TDD:n mm. vähentävän stressiä ja parantavan suhteita työtovereihin. Mikäli TDD todella parantaa ohjelmiston sisäistä laatua ja vähentää virheiden määrää, voisi sillä kuvitella olevan myös tiimihenkeä nostattava vaikutus. Kun ohjelmoijat eivät enää TDD:n tuottamien yksikkötestien ansiosta riko ohjelmistoa niin usein, saattaa suhtautuminen aikaisemmin paljon virheitä tuottaneisiin työtovereihin parantua. Lisäksi jokainen ohjelmoija haluaa varmasti muokata siistimpää koodia, joka on helpommin ymmärrettävissä – siis sellaista koodia, jota TDD:n on väitetty edesauttavan. TDD:n avulla tuotetut testit toimivat järjestelmän dokumentointina, joten ohjelmoijien on helpompaa muokata toistensa tuottamaa koodia, kun he voivat testejä tarkastelemalla paremmin ymmärtää järjestelmän toimintaa. Ilman laajaa yksikkötestien joukkoa kehittäjät joutuvat pahimmillaan arvailemaan, mitä koodin kirjoittaja on mahtanut ajatella. Väitteet suhteiden parantumisesta työtovereihin ja ohjelmoijien välisen kommunikoinnin parantumisesta eivät siis vaikuttaisi olevan aivan tuulesta temmattuja, tosin niiden tueksi ei ole yhtään empiiristä todistusaineistoa. Mitään pidemmälle meneviä johtopäätöksiä ei siis voi näillä tiedoilla vetää.

Sen sijaan väite, että TDD parantaisi ohjelmoijien ja bisnesihmisten välistä kommunikointia näyttäisi olevan enemmänkin puhdasta markkinointipuhetta. Projektin johtoa TDD kyllä auttaa esimerkiksi paremman seurannan myötä, mutta en näe, miten TDD parantaisi esimerkiksi kehittäjien ja markkinoinnin välistä kommunikaatiota. Väite vaikuttaisikin liittyvän enemmän Extreme Programming -menetelmään, jossa käyttäjäkertomukset (engl. user stories) auttavat ohjelmoijia ja bisnesihmisiä puhumaan samaa kieltä. TDD:llä ei ole tähän osaa eikä arpaa.

3.2.9 Opittavuus ja käytettävyys

Mikäli TDD todella tuo edellisissä luvuissa esitettyjä parannuksia ohjelmistokehitysprosessiin, miten helppoa sen sisäistäminen on? Missä kaikkialla testivetoista ohjelmointia on järkevää käyttää? Kirjallisuudessa on esitetty seuraavia väitteitä liittyen TDD:n opittavuuteen ja käytettävyyteen:

35. TDD on helppo oppia (Pollice, 2004).
36. TDD:tä voidaan käyttää missä tahansa kohtaa projektia (Wasmus ja Gross, 2007).
37. TDD on hauskempaa (Astels, 2003).

TDD:tä ei todellakaan ole helppo oppia, päinvastoin. Vaikka luvussa 2 esitetty testivetoisen ohjelmoinnin prosessi vaikuttaakin melko yksinkertaiselta, vaatii sen kunnollinen sisäistäminen aikaa. Useissa tutkimuksissa havaittiin, että TDD:n oppiminen on hankalaa (Abrahamsson et al., 2005; Janzen ja Saiedian, 2007, 2008b; Keefe et al., 2006; Kollanus ja Isomöttönen, 2008; Müller ja Tichy, 2001). TDD on aivan uusi tapa ajatella ja monen on hankala ymmärtää, miten toteutukselle voi kirjoittaa testin, jos toteutusta ei ole edes olemassa. Tätä kuvastaa hyvin opiskelijoiden kommentit TDD:stä, joita Müller ja Tichy (2001) olivat koonneet tutkimukseensa:

> ”Why should we implement test cases if we don’t know exactly what we have todo? We are still figuring out the desired functionality.”

> ”The whole TDD practice, where you write tests before the program code, is stupid... The amount of tests will grow so large there is no sense in that.”

TDD on siis selvästi aloitteleville opiskelijoille hyvin vaikeasti sisäistettävissä. Keefe et al. (2006) toteuttivat kyselyn XP:tä kokeilleilla opiskelijoilla ja selvästi vaikeimmaksi XP:n käytännöksi koettiin juuri TDD. Vanhemmat opiskelijat sen sijaan vaikuttaisivat suhtautuvan keskimäärin positiivisemmin testivetoiseen ohjelmointiin (Janzen ja Saiedian, 2007, 2008b; Melnik ja Maurer, 2005).

Niin ikään Abrahamsson et al. (2005) saivat huomata tutkimuksessaan, että TDD:n sisäistäminen voi olla hyvin hankalaa, myös kokeneille ohjelmoijille. Tutkimuksen mukaan TDD ei myöskään välttämättä sovellu kaikkialle ohjelmistokehitykseen. Tutkimuksessa kehittäjät olivat hyvin haluttomia käyttämään TDD:tä.

He eivät nähneet testien kirjoittamista hyödyllisenä ja loppujen lopuksi vain 7,8 % koodista oli katettu yksikkötesteillä. Kehittäjät sanoivat, että he olisivat tarvinneet lisää koulutusta TDD:n käyttöön. He myös mainitsivat jälkeenpäin, että paremmista testeistä olisi ollut ehdottomasti apua esimerkiksi tapauksessa, jossa kehittäjät käyttivät kaksi tuntia yhden virheen debuggaukseen. Tutkimuksessa pitää kuitenkin ottaa huomioon, että sen aikana kehitetty järjestelmä koostui suurelta osin käyttöliittymästä, johon TDD:n uskotaan soveltuvan melko huonosti (Beck, 2001). Tutkimuksen tulokset tukevat tätä uskomusta. Kehittäjillä ei myöskään ollut kovin suurta motivaatiota käyttää TDD:tä sen vuoksi, että se parantaisi ohjelmiston laatua ja näin helpottaisi ylläpitoa, koska he eivät kuitenkaan olleet järjestelmän kanssa tekemisissä ylläpitovaiheessa. Abrahamsson et al. (2005) uskovat, että TDD:tä ei opita yhdessä päivässä vaan se vaatii useiden kuukausien käyttämistä. Kokemattomien ohjelmoijien käsissä ja vähäisellä koulutuksella TDD ei välttämättä ole parhaimmillaan.

Astels (2003) väittää, että TDD olisi kehittäjille hauskempaa kuin perinteisten menetelmien käyttö. Hauskuutta on hyvin vaikea mitata ja sitä ei olekaan oikeastaan tutkittu yhdessäkään tutkimuksessa. Monissa tutkimuksissa kyllä kysyttiin TDD:tä kokeilleiden ohjelmoijien mielipiteitä (Janzen ja Saiedian, 2006, 2007, 2008b). Esimerkiksi Janzen ja Saiedian (2007) kysyivät opiskelijoilta, minkä menetelmän he valitsisivat jatkossa: TDD:n vai TLD:n. Testivetoista ohjelmointia kokeilleista vanhemmista opiskelijoista noin 85 % valitsi TDD:n. Sen sijaan vain 40 % vanhemmista opiskelijoista valitsi TDD:n, kun he eivät olleet kokeilleet sitä. Nuoremmat opiskelijat taas päätyivät valinnassaan paljon useammin TLD:n käyttöön, vaikka he olisivatkin kokeilleet TDD:tä. Tästä voisi päätellä, että TDD voisi olla vanhempien opiskelijoiden mielestä hauskempaa. Nuoremmilla opiskelijoilla TDD:n hieman vaikea sisäistäminen voi syödä hauskuutta. TDD:n voisi ajatella myös olevan ohjelmoijalle palkitsevampaa, koska testien ajaminen antaa koko ajan konkreettista palautetta siitä, toimiiko koodi vai ei. Läpi menevät testit antavat positiivista palautetta ja ohjelmoijan luottavuus koodiinsa kasvaa. Tällaiset pienet parannukset saattavat tehdä ohjelmoinnista hauskempaa.

Testivetoista ohjelmointia ei siis ole erityisen helppoa oppia. Tämä korostuu etenkin nuoremmilla ohjelmoijilla, joiden on erityisen vaikea sisäistää TDD:tä. Opettelu vie aikaa ja siihen on syytä panostaa, jos haluaa kaikki irti TDD:n eduista. Testivetoinen ohjelmointi ei myöskään sovellu kaikkialle – tutkimusten perusteella ongelmia on ainakin käyttöliittymien toteuttamisessa. Onko TDD sitten sen hauskempaa kuin perinteiset menetelmät, siihen eivät tutkimukset selkeästi vastanneet.

Luku 4

Johtopäätökset

Kirjallisuudesta löytyi lukuisia väitteitä TDD:n eduista, joita käsiteltiin tarkemmin luvussa 3.2. TDD:n on väitetty esimerkiksi parantavan koodin sisäistä laatua mm. kasvattamalla koodin koheesiota ja pienentämällä koodin kytkentää. Myös tuottavuuden, testikattavuuden, ylläpidettävyyden ja koodin ymmärrettävyyden on väitetty parantuvan TDD:llä. Kaiken näiden parannusten lisäksi TDD:n on myös väitetty olevan helppo oppia. Empiiristen tutkimusten perusteella testivetoisen ohjelmoinnin vaikutukset eivät ole kuitenkaan aivan näin ruusuisia kuin TDD:n puolestapuhujat antavat ymmärtää. Osa väitteistä on selvästi valheellisia ja osalle ei löydy todistusaineistoa kerättyjen tutkimusten joukosta. Nämä väitteet on koottu liitteeksi taulukkoon A.1. Taulukosta selviää, mitä esitetyistä väitteistä oli tutkittu empiirisesti sekä mitkä väitteet olivat tutkimusten perusteella tarua ja mitkä totta.

Tutkimustuloksiin vaikutti paljon se, mihin TDD:tä verrattiin. On aivan eri asia verrata TDD:tä menetelmään, jossa yksikkötestausta hädin tuskin tapahtuu kuin esimerkiksi ITL-menetelmään (Iterative test last), jossa yksikkötestit kirjoitetaan välittömästi toteutuksen jälkeen. Osassa tutkimuksissa ei mainittu selvästi, mikä menetelmä toimi TDD:n vertailukohteena. Luonnollisesti TDD tuo selvästi enemmän etuja, jos yksikkötestausta ei aikaisemmin ole tehty lainkaan. Monet TDD:n väitetyistä eduista voidaankin todennäköisesti saavuttaa pelkästään yksikkötestausta lisäämällä.

Selvästi kiistanalaisin väite on TDD:n vaikutus koodin sisäiseen laatuun, kuten kytkennän ja kompleksisuuden pienentymiseen. Tutkimusten perusteella ei ole ollenkaan itsestään selvää, että testivetoinen ohjelmointi automaattisesti parantaisi ohjelmiston sisäistä laatua. Sen sijaan TDD todella vaikuttaisi hieman parantavan koodin ulkoista laatua vähentämällä virheiden määrää. Kuinka paljon TDD sitten pienentää virheiden määrää, riippuu täysin siitä, mihin TDD:tä vertaa. Jos

yksikkötestausta ei ole aikaisemmin juuri tehty, voi TDD vähentää virheitä jopa yli 50 %, muutoin lukema on huomattavasti pienempi. Tutkimukset tukevat myös väitteitä testauksen parantumisesta TDD:llä. Esimerkiksi testikattavuus ja testien määrä oli TDD:llä keskimäärin selvästi korkeampi.

Testivetoisen ohjelmoinnin vaikutus ohjelmoijan tuottavuuteen on sitä vastoin toinen hyvin kiistanalainen väite. Tutkimusten perusteella vaikuttaisikin siltä, että TDD tuskin saa aikaan mitään järisyttäviä muutoksia tuottavuudessa suuntaan tai toiseen. Tuottavuus ei liioin heikkene eikä myöskään parane. Tosin verrattaessa tuottavuutta suhteessa laatuun, voi TDD:llä olla positiivinen kokonaisvaikutus ohjelmoijan tuottavuuteen.

Väitteet ylläpidettävyyden ja laajennettavuuden parantumisesta riippuvat paljon siitä, miten TDD vaikuttaa koodin sisäiseen ja ulkoiseen laatuun. Dokumentointiin ja ymmärrettävyyteen liittyviä näkökulmia taas ei oltu kerätyissä tutkimuksissa tutkittu. Tutkimuksissa tehtyjen kyselyjen perusteella TDD saattaa parantaa ohjelmoijan luottavaisuutta koodinsa laatuun ja antaa enemmän luottamusta muokata koodia jälkeenpäin. Sen sijaan väitteet muista TDD:n psykologisista ja sosiaalisista vaikutuksista jäivät epäselviksi tutkimusten vähyyden vuoksi.

TDD:n käytöstä voi varmasti sanoa sen, että sen opettelu ei ole helppoa eikä se ole mikään hopealuoti, jota voisi käyttää menestyksekkäästi kaikkien ohjelmistojen kehityksessä. Erityisen ongelmallista testivetoisen ohjelmoinnin käyttö vaikuttaisi olevan graafisten käyttöliittymien kehittämisessä. TDD:n käyttöönotossa tärkeää on ohjelmoijien perusteellinen opettaminen ja motivointi TDD:n käyttöön. Testivetoinen ohjelmointi on vaikea oppia ja hyvin pinnallinen opettelu ei välttämättä tuo mitään parannuksia – päinvastoin tuottavuus ja koodin laatu saattavat jopa heiketä. TDD vaikuttaisi olevan tehokkain kokeneiden ohjelmoijien käsissä, sen sijaan aloittelevat opiskelijat olivat tutkimusten mukaan haluttomimpia käyttämään TDD:tä. Testivetoinen ohjelmointi ei siis tuo mukanaan vain pelkkiä etuja vaan myös haasteita ja ongelmia.

Tulevien TDD:tä käsittelevien tutkimusten olisi syytä keskittyä tutkimaan, miten TDD vaikuttaa koodin sisäiseen laatuun. TDD:hän on nimenomaan suunnittelukäytäntö, joten tämän alueen tarkempi tutkimus olisi hyvin luonnollista. Tämä toisi vastauksia myös liittyen parempaan ylläpidettävyyteen ja laajennettavuuteen. Tutkimuksissa pitäisi myös keskittyä vertaamaan TDD:tä sellaisiin iteratiivisiin menetelmiin, jossa yksikkötestausta todella tapahtuu – eli keskittyä todella siihen, mikä TDD:ssä on erilaista: testien kirjoittaminen *etukäteen*. Tämä suunta oli jo havaittavissa tuoreimmissa tutkimuksissa, joissa vertailukohteena oli monessa ITL. Kuitenkin myös moni vertasi TDD:tä menetelmiin, joissa yksikkötestaus oli lepsumpaa tai sitä ei tehty ehkä laisinkaan. Yksikkötestaus ei kuitenkaan ole enää mikään uusi juttu, joten sen voinee jo mieltää kuuluvan perinteiseen

ohjelmistokehitykseen. Ja kuinka moni yritys yhä toteuttaa ohjelmistoja tiukasti vesiputousmallisesti tehden testejä vasta, kun kaikki toteutus on tehty? Onko siis järkevää verrata TDD:tä kankeisiin vesiputousmalleihin? Mielestäni ei.

Pitää kuitenkin muistaa, että tässä työssä käsitellyt tutkimukset TDD:stä ovat monet melko pieniä ja tulokset eivät ole aina olleet tilastollisesti merkittäviä, joten tulosten yleistäminen on hankalaa. TDD:n tuomia etuja on vaikea eristää ja eri tulokset saattavat monessa tutkimuksessa johtua puhtaasti ohjelmoijien tasoeroista. Siksi testivetoista ohjelmointia olisikin syytä tutkia enemmän, jotta väitettyjen etujen voidaan todella sanoa johtuvan nimenomaan TDD:stä. Vaikka TDD:n vaikutukset jäivät tässäkin työssä osin yhä hieman hämärän peittoon, vaikuttaisi TDD:llä ainakin olevan enemmän positiivisia kuin negatiivisia vaikutuksia, joten sen käyttäminen on ehdottomasti kokeilemisen arvoista.

Kirjallisuutta

P. Abrahamsson, A. Hanhineva ja J. Jäälinoja. Improving business agility through technical solutions: A case study on test-driven development in mobile software development. *IFIP International Federation for Information Processing*, 180: 227–243, 2005. doi: 10.1007/0-387-25590-7_14.

S. W. Ambler. Survey says: Agile works in practice. *Dr. Dobb's Journal*, 31(9):62–64, 2006. URL `http://www.ddj.com/architect/191800169?cid=Ambysoft`. [Viitattu 18.11.2008].

D. Astels. *Test Driven development: A Practical Guide*. Prentice Hall Professional Technical Reference, 2003. ISBN 0-13-101649-0.

K. Beck. Aim, fire. *IEEE Software*, 18(5):87–89, 2001. ISSN 0740-7459. doi: 10.1109/52.951502.

K. Beck. *Test Driven Development: By Example*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 0-321-14653-0.

A. Begel ja N. Nagappan. Usage and perceptions of agile software development in an industrial context: An exploratory study. *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, sivut 255–264, 2007. ISSN 1938-6451. doi: 10.1109/ESEM.2007.12.

T. Bhat ja N. Nagappan. Evaluating the efficacy of test-driven development: Industrial case studies. *ISCE'06 - 5th ACM-IEEE International Symposium on Empirical Software Engineering*, osa 2006, sivut 356–363, 2006. ISBN 1-59593-218-6.

G. Canfora, A. Cimitile, F. Garcia, M. Piattini ja C. A. Visaggio. Evaluating advantages of test driven development: A controlled experiment with professionals. *ISCE'06 - 5th ACM-IEEE International Symposium on Empirical Software Engineering*, osa 2006, sivut 364–371, 2006. ISBN 1-59593-218-6.

T. A. Corbi. Program understanding: challenge for the 1990's. *IBM Syst. J.*, 28 (2):294–306, 1989. ISSN 0018-8670.

L. Crispin. Driving software quality: How test-driven development impacts software quality. *IEEE Software*, 23(6):70–71, 2006. ISSN 0740-7459.

C. Desai, D. S. Janzen ja K. Savage. A survey of evidence for test-driven development in academia. *SIGCSE Bull.*, 40(2):97–101, 2008. ISSN 0097-8418. doi: 10.1145/1383602.1383644.

S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, sivut 26–30, New York, NY, USA, 2004. ACM. ISBN 1-58113-798-2. doi: 10.1145/971300.971312.

H. Erdogmus, M. Morisio ja M. Torchiano. On the effectiveness of the test-first approach to programming. *Software Engineering, IEEE Transactions on*, 31 (3):226–237, 2005. ISSN 0098-5589. doi: 10.1109/TSE.2005.37.

B. George ja L. Williams. A structured experiment of test-driven development. *Information and Software Technology*, 46(5 SPEC. ISS.):337–342, 2004. ISSN 0950-5849. doi: 10.1016/j.infsof.2003.09.011.

A. Geras, M. Smith ja J. Miller. A prototype empirical evaluation of test driven development. *Software Metrics, 2004. Proceedings. 10th International Symposium on*, sivut 405–416, 2004. ISSN 1530-1435. doi: 10.1109/METRIC.2004.1357925.

A. Gupta ja P. Jalote. An experimental evaluation of the effectiveness and efficiency of the test driven development. *1st International Symposium on Empirical Software Engineering and Measurement, ESEM 2007*, sivut 285–294, 2007. ISBN 0-7695-2886-4. doi: 10.1109/ESEM.2007.41.

L. Huang ja M. Holcombe. Empirical investigation towards the effectiveness of test first programming. *Information and Software Technology,*, In Press, Corrected Proof, 2008. ISSN 0950-5849. doi: 10.1016/j.infsof.2008.03.007.

D. S. Janzen. Software architecture improvement through test-driven development. *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, sivut 240–241, New York, NY, USA, 2005. ACM. ISBN 1-59593-193-7. doi: 10.1145/1094855.1094954.

D. S. Janzen ja H. Saiedian. On the influence of test-driven development on software design. *CSEET '06: Proceedings of the 19th Conference on Software Engineering Education & Training*, sivut 141–148, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2557-1. doi: 10.1109/CSEET.2006.25.

D. S. Janzen ja H. Saiedian. A leveled examination of test-driven development acceptance. *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, sivut 719–722, 2007. ISSN 0270-5257. doi: 10.1109/ICSE.2007.8.

D. S. Janzen ja H. Saiedian. Does test-driven development really improve software design quality? *IEEE Software*, 25(2):77–84, 2008a. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2008.34.

D. S. Janzen ja H. Saiedian. Test-driven development concepts, taxonomy, and future direction. *IEEE Computer*, 38(9):43–50, 2005. ISSN 0018-9162. doi: 10.1109/MC.2005.314.

D. S. Janzen ja H. Saiedian. Test-driven learning in early programming courses. *SIGCSE Bull.*, 40(1):532–536, 2008b. ISSN 0097-8418. doi: 10.1145/1352322.1352315.

R. Jeffries ja G. Melnik. Guest editors' introduction: TDD–The art of fearless programming. *IEEE Software*, 24(3):24–30, 2007. ISSN 0740-7459. doi: 10.1109/MS.2007.75.

C. G. Jones. Test-driven development goes to school. *J. Comput. Small Coll.*, 20 (1):220–231, 2004. ISSN 1937-4771.

J. Järvenpää. Reaktor Innovations - Test-Driven Development, 2006. URL `http://www.ri.fi/web/fi/teknologia-ja-tutkimus/tdd`. [Viitattu 18.11.2008].

R. Kaufmann ja D. S. Janzen. Implications of test-driven development: a pilot study. *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, sivut 298–299, New York, NY, USA, 2003. ACM. ISBN 1-58113-751-6. doi: 10.1145/949344.949421.

K. Keefe, J. Sheard ja M. Dick. Adopting XP practices for teaching object oriented programming. *ACE '06: Proceedings of the 8th Austalian conference on Computing education*, sivut 91–100, Darlinghurst, Australia, 2006. Australian Computer Society, Inc. ISBN 1-920682-34-1.

S. Kollanus ja V. Isomöttönen. Test-driven development in education: Experiences with critical viewpoints. *ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education*, sivut 124–127, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-078-4. doi: 10.1145/1384271.1384306.

C. Larman ja V. R. Basili. Iterative and incremental developments - A brief history. *IEEE Computer*, 36(6):47–56, 2003. ISSN 0018-9162. doi: 10.1109/MC.2003.1204375.

K. M. Lui. Test driven development and software process improvement in China. *Springer-Verlag, 2004*, sivut 219–222, 2004. ISSN 0302-9743.

L. Madeyski ja L. Szala. The impact of test-driven development on software development productivity - An empirical study. osa 4764 LNCS sarjasta *14th European Software Process Improvement Conference, EuroSPI 2007*, sivut 200–211, 2007.

R. C. Martin. Professionalism and test-driven development. *IEEE Software*, 24 (3):32–36, 2007. ISSN 0740-7459. doi: 10.1109/MS.2007.85.

E. M. Maximilien ja L. Williams. Assessing test-driven development at IBM. *25th International Conference on Software Engineering*, sivut 564–569, 2003. ISBN 0-7695-1877-X.

T. J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, 1976. ISSN 0098-5589.

G. Melnik ja F. Maurer. A cross-program investigation of students' perceptions of agile methods. *ICSE '05: Proceedings of the 27th international conference on Software engineering*, sivut 481–488, New York, NY, USA, 2005. ACM. ISBN 1-59593-963-2. doi: 10.1145/1062455.1062543.

M. M. Müller. The effect of test-driven development on program code. *Lecture Notes in Computer Science*, 4044, 2006. ISSN 0302-9743.

M. M. Müller ja O. Hagner. Experiment about test-first programming. *Software, IEE Proceedings* -, 149(5):131–136, 2002. ISSN 1462-5970. doi: 10.1049/ip-sen:20020540.

M. M. Müller ja W. F. Tichy. Case study: Extreme programming in a university environment. *Software Engineering, 2001. ICSE 2001. Proceedings of the 23rd International Conference on*, sivut 537–544, 2001. ISSN 0270-5257. doi: 10.1109/ICSE.2001.919128.

N. Nagappan, E. M. Maximilien, T. Bhat ja L. Williams. Realizing quality improvement through test driven development: Results and experiences of four industrial teams. *Empirical Software Engineering*, 13(3):289–302, 2008. ISSN 1382-3256. doi: 10.1007/s10664-008-9062-z.

M. Pancur, M. Ciglaric, M. Trampus ja T. Vidmar. Towards empirical evaluation of test-driven development in a university environment. *EUROCON 2003. Computer as a Tool. The IEEE Region 8*, 2:83–86 vol.2, 2003. doi: 10.1109/EURCON.2003.1248153.

G. Pollice. Test before you code, 2004. URL `http://www.ibm.com/developerworks/rational/library/4929.html`. [Viitattu 18.11.2008].

A. Rendell. Effective and pragmatic test driven development. *Agile, 2008. AGILE '08. Conference*, sivut 298–303, 2008. doi: 10.1109/Agile.2008.45.

O. Salo ja P. Abrahamsson. Agile methods in European embedded software development organisations: A survey on the actual use and usefulness of Extreme Programming and Scrum. *Software, IET*, 2(1):58–64, 2008. ISSN 1751-8806. doi: 10.1049/iet-sen:20070038.

J. C. Sanchez, L. Williams ja E. M. Maximilien. On the sustained use of a test-driven development practice at IBM. *AGILE 2007*, sivut 5–14, 2007. doi: 10.1109/AGILE.2007.43.

M. Siniaalto ja P. Abrahamsson. A comparative case study on the impact of test-driven development on program design and test coverage. *1st International Symposium on Empirical Software Engineering and Measurement, ESEM 2007*, sivut 275–284, 2007. ISBN 0-7695-2886-4. doi: 10.1109/ESEM.2007.35.

M. Siniaalto ja P. Abrahamsson. *Does test-driven development improve the program code? Alarming results from a comparative case study*, osa 5082 LNCS sarjasta *2nd IFIP TC 2 Central and East European Conference on Software Engineering Techniques, CEE-SET 2007*. Poznan painos, 2008.

H. Wasmus ja H.-G. Gross. Evaluation of test-driven development. *2nd Working Conference on Evaluation of Novel Approaches to Software Engineering*, L. A. Maciaszek C. Gonzales-Perez, toimittaja, sivut 103–110. Insticc Press, 2007. ISBN 978-989-8111-10-4.

L. Williams, E. M. Maximilien ja M. Vouk. Test-driven development as a defect-reduction practice. *Software Reliability Engineering, 2003. ISSRE 2003. 14th International Symposium on*, sivut 34–45, 2003. ISSN 1071-9458. doi: 10.1109/ISSRE.2003.1251029.

S. Yenduri ja L. Perkins. Impact of using test-driven development: A case study. *Software Engineering Research and Practice*, H. R. Arabnia ja H. Reza, toimittajat, sivut 126–129. CSREA Press, 2006. ISBN 1-932415-90-4.

Liite A

Taulukko TDD:n eduista

Taulukossa A.1 on koottuna kirjallisuudessa esitettyjä väitteitä TDD:n eduista. Taulukon viimeisessä sarakkeessa on tehty empiiristen tutkimusten pohjalta päätelmä siitä, pitääkö väite paikkansa vai ei. Jos sarakkeessa on kysymysmerkki (?), tarkoittaa se sitä, että väitettä ei ole empiirisesti tutkittu tai tutkimusten tuloksista ei ole voinut vetää lopullisia johtopäätöksiä väitteen pätevyydestä suuntaan tai toiseen.

Väite	Missä esitetty?	Onko tutkittu?	Näyttöä puolesta	Ei näyttöä puolesta	Tarua vai totta?
Vaikutus koodin sisäiseen laatuun					
1. TDD kasvattaa koodin koheesiota (engl. cohesion)	Beck (2001)	x		Janzen ja Saiedian (2008a); Müller (2006); Siniaalto ja Abrahamsson (2007, 2008)	Tarua
2. TDD pienentää koodin kytkentää (engl. coupling)	Beck (2001)	x	Müller (2006); Siniaalto ja Abrahamsson (2007)	Janzen ja Saiedian (2006, 2008a); Siniaalto ja Abrahamsson (2008)	?
3. TDD pienentää koodin kompleksisuutta (engl. complexity)	Crispin (2006)	x	Janzen ja Saiedian (2008a); Müller (2006); Siniaalto ja Abrahamsson (2008)	Janzen ja Saiedian (2006); Kaufmann ja Janzen (2003); Siniaalto ja Abrahamsson (2007)	?
4. TDD:llä tehty koodi on kompaktimpaa (engl. clean code)	Martin (2007)	x	Janzen ja Saiedian (2008a); Madeyski ja Szala (2007)	Janzen ja Saiedian (2006)	?
Vaikutus koodin ulkoiseen laatuun ja virheiden paikallistamiseen					
5. TDD parantaa koodin ulkoista laatua (vähentää virheitä koodissa)	Beck (2002)	x	Bhat ja Nagappan (2006); Edwards (2004); George ja Williams (2004); Lui (2004); Maximilien ja Williams (2003); Sanchez et al. (2007); Yenduri ja Perkins (2006)	Erdogmus et al. (2005); Huang ja Holcombe (2008); Müller ja Hagner (2002); Pancur et al. (2003)	Totta

jatkuu seuraavalla sivulla...

Väite	Missä esitetty?	Onko tutkittu?	Näyttöä puolesta	Ei näyttöä puolesta	Tarua vai totta?
6. TDD nopeuttaa virheiden paikallistamista ja korjausta	Martin (2007)	x	Lui (2004)		?
7. TDD vähentää debuggausta	Martin (2007)				?
Vaikutus testeihin ja testaukseen					
8. Ohjelmoija kirjoittaa enemmän testejä TDD:llä	Canfora et al. (2006)	x	Erdogmus2005, George2004, Geras2004, Janzen2006, Janzen2008, JanzenCourses2008, Yenduri2006	Canfora et al. (2006)	Totta
9. TDD parantaa järjestelmän testikattavuutta	Astels (2003)	x	Janzen ja Saiedian (2006, 2008a); Siniaalto ja Abrahamsson (2007)	Geras et al. (2004); Pancur et al. (2003)	Totta
10. TDD parantaa testaustaitoja ja motivoi testaamiseen	Desai et al. (2008)	x	Huang ja Holcombe (2008); Janzen ja Saiedian (2008a,b)		Totta
Vaikutus tuottavuuteen ja projektin hallintaan					
11. TDD parantaa ohjelmoijan tuottavuutta	Jeffries ja Melnik (2007)	x	Erdogmus2005, Gupta2007, Huang2008, Janzen2006, Kaufmann2003, Madeyski2007, Yenduri2006	Bhat ja Nagappan (2006); Canfora et al. (2006); George ja Williams (2004); Geras et al. (2004); Janzen ja Saiedian (2008b); Maximilien ja Williams (2003); Müller ja Hagner (2002); Siniaalto ja Abrahamsson (2007)	?

jatkuu seuraavalla sivulla...

Väite	Missä esitetty?	Onko tutkittu?	Näyttöä puolesta	Ei näyttöä puolesta	Tarua vai totta?
12. TDD parantaa tehtävien työmääräarvioita	Canfora et al. (2006)	x	Canfora et al. (2006); Geras et al. (2004); Lui (2004)		Totta
13. TDD parantaa projektin seurantaa	Lui (2004)	x	Lui (2004)		Totta
14. TDD vähentää ”turhien” ominaisuuksien toteuttamista	Rendell (2008)				?
15. TDD vaatii vähemmän uudelleentyötä	Jeffries ja Melnik (2007)				?
Vaikutus ylläpidettävyyteen ja laajennettavuuteen					
16. TDD tuottaa helpommin ylläpidettävää koodia	Astels (2003)	x	Sanchez et al. (2007)	Siniaalto ja Abrahamsson (2008)	Totta
17. TDD vähentää ylläpitokuluja	Beck (2002)	x	Ks. väitteet 1–9		Totta
18. TDD parantaa koodin uudelleenkäytettävyyttä	Janzen (2005)	x	Ks. väitteet 1–4		?
19. TDD parantaa koodin laajennettavuutta	Martin (2007)	x	Ks. väitteet 1–4		?
20. TDD parantaa koodin integroitavuutta	Järvenpää (2006)	x	Ks. väitteet 1–4		?
21. TDD tuottaa luotettavampia ja vakaampia järjestelmiä	Astels (2003)	x	Ks. väitteet 5–9		Totta

jatkuu seuraavalla sivulla...

Väite	Missä esitetty?	Onko tutkittu?	Näyttöä puolesta	Ei näyttöä puolesta	Tarua vai totta?
Vaikutus dokumentointiin ja ymmärrettävyyteen					
22. TDD parantaa koodin toiminnan ymmärtämistä	Martin (2007)	x	Müller ja Hagner (2002)		?
23. TDD auttaa ymmärtämään järjestelmän vaatimuksia paremmin	Jeffries ja Melnik (2007)				?
24. TDD pystyy vastaamaan paremmin vaatimusten muutoksiin	Wasmus ja Gross (2007)				?
Psykologiset vaikutukset					
25. TDD parantaa ohjelmoijan luottavaisuutta koodinsa laatuun	Beck (2002)	x	Edwards (2004); Janzen ja Saiedian (2008b); Kaufmann ja Janzen (2003); Müller ja Tichy (2001)	Gupta ja Jalote (2007)	Totta
26. TDD antaa enemmän luottamusta muokata koodia jälkeenpäin	Beck (2002)	x	Edwards (2004); Janzen ja Saiedian (2006)		Totta
27. TDD parantaa ohjelmoijan kuria	Lui (2004)	x	Lui (2004)	Abrahamsson et al. (2005)	?
28. TDD kasvattaa ohjelmoijan työstä saamaansa tyydytystä	Beck (2002)				?
29. TDD vähentää ohjelmoijan stressiä	Jeffries ja Melnik (2007)				?

jatkuu seuraavalla sivulla...

Väite	Missä esitetty?	Onko tutkittu?	Näyttöä puolesta	Ei näyttöä puolesta	Tarua vai totta?
30. Asiakaan odotukset täyttyvät todennäköisemmin TDD:llä	Wasmus ja Gross (2007)				?
31. TDD on nautinnollisempaa kaikille sidosryhmille	Wasmus ja Gross (2007)				?
Sosiaaliset vaikutukset					
32. TDD parantaa suhteita työtovereihin	Beck (2002)				?
33. TDD parantaa ohjelmoijien välistä kommunikointia	Crispin (2006)				?
34. TDD parantaa ohjelmoijien ja bisnesihmisten välistä kommunikointia	Crispin (2006)				Tarua
Opittavuus ja käytettävyys					
35. TDD on helppo oppia	Pollice (2004)	x		Abrahamsson et al. (2005); Janzen ja Saiedian (2007, 2008b); Keefe et al. (2006); Kollanus ja Isomöttönen (2008); Müller ja Tichy (2001)	Tarua

jatkuu seuraavalla sivulla...

Väite	Missä esitetty?	Onko tutkittu?	Näyttöä puolesta	Ei näyttöä puolesta	Tarua vai totta?
36. TDD:tä voidaan käyttää missä tahansa kohtaa projektia	Wasmus ja Gross (2007)	x		Abrahamsson et al. (2005)	Tarua
37. TDD on hauskempaa	Astels (2003)				?

Taulukko A.1: Kirjallisuudessa esitettyjä väitteitä TDD:n eduista

Liite B

Yhteenveto TDD:n tutkimuksista

Taulukossa B.1 on esitetty yhteenveto tärkeimmistä TDD:n empiirisistä tutkimuksista. Tutkimukset on jaoteltu akateemisiin, teollisiin tai puoli-teollisiin tutkimuksiin riippuen siitä, käytettiinkö tutkittavina opiskelijoita, missä ympäristössä tutkimus suoritettiin ja miten tarkasti tutkimuksia kontrolloitiin (katso tarkemmin kappale 3.1). Tutkimukset jaoteltiin myös sen mukaan, mihin havainnot perustuvat. Suurin osa tutkimuksista oli kontrolloituja kokeita tai case-tapauksia, joissa havainnot perustuvat tutkimusten aikana kerättyihin metriikoihin, kuten koodirivien määrä, käytetty aika tai testikattavuus. Toiset tutkimukset taas olivat puhtaita kyselyitä, joissa tutkittavat laitettiin yleensä toteuttamaan ohjelma TDD:llä ja lopuksi heiltä kysyttiin mielipiteitä TDD:n käytöstä. Kyselyissä havainnot eivät siis pohjaudu varsinaisesti mittaustuloksiin, mutta ovat oleellinen osa esimerkiksi oppimisen ja psykologisten vaikutusten mittaamisessa.

Tutkimuksista on lisäksi mainittu tutkimuksiin osallistuneiden määrä sekä tutkimuksen vertailukohde. Kaikissa tutkimuksissa ei kerrottu tarkasti tutkittavien määrää tai niiden jakautumista eri vertailuryhmiin, joten osassa tämä tieto on puutteellinen. Vertailukohteella tarkoitetaan sitä menetelmää, johon TDD:tä verrattiin. TLD (Test-last development) tarkoittaa tässä perinteistä vesiputousmaista ohjelmistokehitysmenetelmää, jossa testit kirjoitetaan vasta sen jälkeen, kun toteutus on valmis. Kaikista tutkimuksista ei selvinnyt, missä vaiheessa testit kirjoitettiin, kun vertailukohteena käytettiin perinteistä ohjelmistokehitysmenetelmää, joten TLD on hyvin laaja käsite. TLD:ssä testit saatettiin kirjoittaa tunteja, päiviä, viikkoja tai vasta kuukausia toteutuskoodin jälkeen. Osassa tutkimuksissa vertailukohteena käytettiin TLD:tä paljon iteratiivisempaa menetelmää: ITL:ää (Iterative test last). ITL on siis käytännössä sama kuin TDD, mutta sen sijaan, että testit kirjoitettaisiin *välittömästi ennen toteutusta*, ne kirjoitetaankin pienissä osissa *välittömästi toteutuksen jälkeen* (ei vasta päiviä tai kuukausia

myöhemmin). Joissain tutkimuksissa vertailukohdetta ei ollut kerrottu selkeästi tai se oli mainittu vain muodossa ”ei-TDD”. Vertailukohde on aina syytä ottaa huomioon verrattaessa eri tutkimusten tuloksia, koska TDD:n tuomat edut ovat aivan eri luokkaa esimerkiksi ITL:n ja pelkän ad-hoc-yksikkötestauksen välillä.

Havainnoissa plus (+) tarkoittaa, että havainto on positiivinen TDD:stä esitettyjen etujen kannalta. Miinusmerkki (–) taas tarkoittaa, että havainto on TDD:n kannalta negatiivinen (esimerkiksi virheiden määrä ei olekaan vähäisempi). Tähti (*) tarkoittaa neutraalia havaintoa, jolla ei oikeastaan ole positiivista eikä negatiivista vaikutusta.

Tutkimus	Tutkimuksen tyyppi	Havainnot pohjautuvat	Tutkittavat	Vertailu-kohde	Tärkeimmät havainnot
Edwards (2004)	Akateeminen	Kontrolloitu koe	118 opiskelijaa: 59 TDD:llä + 59 ei-TDD:llä	Ei-TDD	+ Virheiden määrä väheni 45 % TDD:llä + Opiskelijat kertoivat TDD:n parantaneen luottamusta oman koodin oikeellisuuteen ja lisänneen luottamusta muokata koodia jälkeenpäin
Erdogmus et al. (2005)	Akateeminen	Kontrolloitu koe	24 opiskelijaa: 11 TDD:llä + 13 ITL:llä	ITL	+ TDD:tä käyttäneet opiskelija kirjoittivat selvästi enemmän testejä + TDD vaikuttaa parantavan tuottavuutta – TDD ei parantanut koodin laatua
Gupta ja Jalote (2007)	Akateeminen	Kontrolloitu koe	22 opiskelijaa: 11 TDD:llä ja 11 TLD:llä	TLD	+ TDD vaikuttaisi parantavan hiukan ohjelmoijan tuottavuutta + TDD:tä käyttäneillä kului kokonaisuudessaan vähemmän aikaa toteutukseen + TDD paransi osittain koodin laatua, mutta tulos ei ollut vertailukelpoinen – TDD:tä käyttäneet olivat vähemmän luottavaisia designin laatuun
Huang ja Holcombe (2008)	Akateeminen	Kontrolloitu koe	39 opiskelijaa	TLD	+ TDD:tä käyttäneet ohjelmoijat käyttivät enemmän aikaa testaukseen ja vähemmän varsinaiseen koodaukseen + TDD:llä näytti olevan positiivinen vaikutus tuottavuuteen (tosin ei tilastollisesti merkittävästi) – Ei tilastollista eroa koodin laadussa TDD:n ja TLD:n välillä

jatkuu seuraavalla sivulla...

Tutkimus	Tutkimuksen tyyppi	Havainnot pohjautuvat	Tutkittavat	Vertailu-kohde	Tärkeimmät havainnot
Janzen ja Saiedian (2006)	Akateeminen	Kontrolloitu koe	10 opiskelijaa 3 ryhmässä: 1 TDD:llä + 1 TLD:llä + 1 ilman testejä	TLD & ei testejä lainkaan	+ TDD:llä oli positiivinen vaikutus opiskelijoiden tuottavuuteen + Opiskelijoilla oli paljon positiivisempi asenne TDD:tä kohtaan sen kokeilun jälkeen + TDD:tä käyttäneet kirjoittivat lähes kaksi kertaa enemmän testejä koodiriviä kohden. * TDD:n testikattavuus (line coverage) ei kuitenkaan ollut yhtään suurempi kuin TLD:n paitsi haarakattavuuden (branch coverage) osalta, joka oli 86 % parempi TDD:llä – TDD ei pienentänyt koodin kompleksisuutta tai kytkentää, päinvastoin se kasvatti niitä
Janzen ja Saiedian (2007)	Akateeminen	Kysely	184 opiskelijaa	TLD	+ TDD:tä kokeilleet opiskelijat kertoivat käyttävänsä TDD:tä todennäköisemmin myös jatkossa * Vanhemmat opiskelijat ovat halukkaampia käyttämään TDD:tä sen kokeilun jälkeen kuin aloittelevat opiskelijat
Janzen ja Saiedian (2008b)	Akateeminen	Kontrolloitu koe	140 opiskelijaa	ITL	+ TDD:tä käyttäneet opiskelijat olivat luottavaisempia koodiinsa + TDD:tä käyttäneet opiskelijat kirjoittivat enemmän testejä * TDD:tä käyttäneet kirjoittivat enemmän testejä myös ITL:llä käytettyään ensin TDD:tä – TDD:llä ei tilastollisesti merkittävää vaikutusta tuottavuuteen – Aloittelevilla opiskelijoilla haluttomuutta käyttää TDD:tä

jatkuu seuraavalla sivulla...

Tutkimus	Tutkimuksen tyyppi	Havainnot pohjautuvat	Tutkittavat	Vertailu-kohde	Tärkeimmät havainnot
Kaufmann ja Janzen (2003)	Akateeminen	Kontrolloitu koe	8 opiskelijaa: 4 TDD:llä + 4 TLD:llä	TLD	+ TDD:tä käyttäneet kirjoittivat 50 % enemmän koodia + TDD:tä käyttäneet olivat paljon luottavaisempia koodiinsa – TDD:llä ei vaikutusta koodin kompleksisuuteen
Keefe et al. (2006)	Akateeminen	Kysely	12 opiskelijaa	-	– TDD oli opiskelijoiden mielestä vaikein XP:n käytäntö
Kollanus ja Isomöttö-nen (2008)	Akateeminen	Kysely	52 opiskelijaa	-	+ Vaikeuksista huolimatta opiskelijat uskoivat TDD:n parantavan koodinsa laatua ja luottamusta omaan koodiin * TDD:tä ei koettu keskimäärin niin vaikeaksi, mutta hajonta vastauksissa oli suuri
Melnik ja Maurer (2005)	Akateeminen	Kysely	240 opiskelijaa	-	+ 73 % vastanneista kertoi TDD:n parantavan ohjelmiston laatua * Kokeneemmilla opiskelijoilla oli hieman positiivisempi asenne TDD:tä kohtaan
Müller ja Tichy (2001)	Akateeminen	Kysely	11 opiskelijaa	-	+ Opiskelijoista 87 % kertoi TDD:n tuottamien regressiotestien ajamisen parantaneen heidän luottamusta koodiinsa – TDD:n sisäistäminen ei ole helppoa
Müller ja Hagner (2002)	Akateeminen	Kontrolloitu koe	19 opiskelijaa: 10 TDD:llä + 9 TLD:llä	TLD	+ TDD vaikuttaisi parantavan ohjelman ymmärtämistä – TDD ei nopeuta ohjelman valmistumista – TDD ei paranna ohjelman ulkoista laatua
Pancur et al. (2003)	Akateeminen	Kontrolloitu koe	34 opiskelijaa: 19 TDD:llä + 15 ITL:llä	ITL	– Testikattavuudessa ei ollut eroa TDD:n ja ITL:n välillä – TDD ei parantanut koodin ulkoista laatua – Opiskelijoiden mielestä TDD oli vaikea sisäistää ja heidän mielipiteensä menetelmän tehokkuudesta eivät olleet niin positiivisia kuin muissa vastaavissa kyselyissä.

jatkuu seuraavalla sivulla...

Tutkimus	Tutkimuksen tyyppi	Havainnot pohjautuvat	Tutkittavat	Vertailu-kohde	Tärkeimmät havainnot
Yenduri ja Perkins (2006)	Akateeminen	Kontrolloitu koe	18 opiskelijaa: 9 TDD:llä + 9 TLD:llä	TLD	+ Virheiden määrä väheni 35 % TDD:llä + Tuottavuus kasvoi 25 % TDD:llä + TDD:tä käyttäneet kirjoittivat lähes 3 kertaa enemmän testitapauksia
Abrahamsson et al. (2005)	Puoli-teollinen	Kontrolloitu koe	4 ohjelmoijaa: 3 opiskelijaa + 1 teollisuudesta	-	– Kehittäjät olivat haluttomia käyttämään TDD:tä: testikattavuus oli vain 7,8 % – Kehittäjät olisivat tarvinneet enemmän koulutusta TDD:lle – TDD ei välttämättä sovellu kaikkiin ohjelmistokehityksen osa-alueisiin (mm. käyttöliittymien kehittäminen)
Canfora et al. (2006)	Puoli-teollinen	Kontrolloitu koe	28 ohjelmoijaa	TLD	+ TDD oli paremmin ennustettavissa (pienempi hajonta mitatuissa arvoissa) – TDD:tä käyttäneet eivät kirjoittaneet tilastollisesti merkittävästi enempää testitapauksia – TDD:n käyttö vei selvästi enemmän aikaa kuin TLD
George ja Williams (2004)	Puoli-teollinen	Kontrolloitu koe	24 pariohjel-moijaa: 12 TDD:llä + 12 TLD:llä	TLD	+ Virheiden määrä väheni. TDD:llä kirjoitettu koodi läpäisi 18 % enemmän testitapauksia + Testikattavuus oli korkea TDD:llä: 98% (method coverage), 92% (statement coverage) ja 97% (branch coverage) + TDD vaikuttaa rohkaisevan kirjoittamaan enemmän testejä + Ohjelmoijista 80 % mielsi TDD:n tehokkaaksi menetelmäksi ja 78 % uskoi sen parantavan tuottavuutta – TDD vaati keskimäärin 16 % enemmän aikaa

jatkuu seuraavalla sivulla...

Tutkimus	Tutkimuksen tyyppi	Havainnot pohjautuvat	Tutkittavat	Vertailu-kohde	Tärkeimmät havainnot
Geras et al. (2004)	Puoli-teollinen	Kontrolloitu koe	14 ohjelmoijaa teollisuudes-ta: 7 TDD:llä + 7 TLD:llä	TLD	+ TDD on paremmin ennustettavissa (pienempi hajonta käytetyssä ajassa) + TDD:tä käyttäneet kirjoittivat enemmän testejä + Ohjelmoijat eivät käyttäneet TDD:llä sen enempää aikaa kuin TLD:llä – TDD ei parantanut testikattavuutta, tosin testikattavuus oli molemmissa ryhmissä korkea
Janzen ja Saiedian (2008a)	Puoli-teollinen	Kontrolloitu koe	6 tutkimusta, yhteensä 27 ohjelmoijaa	ITL	+ TDD paransi selvästi testikattavuutta + TDD:tä käyttäneet kirjoittivat pienempiä moduuleita + TDD:tä käyttäneet kirjoittivat keskimäärin lyhempiä metodeita ja pienempiä luokkia + TDD näytti pienentävän koodin kompleksisuutta * TDD:tä käyttäneet tuottivat koodia paremmalla testikattavuudella myös ITL:llä käytettyään ensin TDD:tä – TDD vaikutti hieman kasvattaneen koodin kytkentää, mutta kyseenalaiseksi jäi, oliko kasvanut kytkentä ”hyvälaatuista” kytkentää vai ei – TDD:llä ei ollut tilastollisesti merkittävää vaikutusta koheesioon

jatkuu seuraavalla sivulla...

Tutkimus	Tutkimuksen tyyppi	Havainnot pohjautuvat	Tutkittavat	Vertailukohde	Tärkeimmät havainnot
Madeyski ja Szala (2007)	Puoli-teollinen	Kontrolloitu koe	1 kokenut ohjelmoija teollisuudesta	ITL	+ TDD näyttäisi parantavan tuottavuutta + TDD saattaa tuottaa kompaktimpaa koodia
Müller (2006)	Puoli-teollinen	Case-tapauksia	8 projektia : 5 TDD:llä & 3 ei-TDD:llä	Ei-TDD	+ TDD näyttäisi parantavan testattavuutta + TDD vaikuttaisi pienentävän kytkentää ja kompleksisuutta – TDD ei parantanut koheesiota
Siniaalto ja Abrahamsson (2007)	Puoli-teollinen	Kontrolloitu koe	13 opiskelijaa: 4 TDD:llä + 9 ITL:llä	ITL	+ Testikattavuus TDD:llä oli huomattavasti parempi + TDD:n käyttö pienensi hieman koodin kytkentää, mutta ei tilastollisesti merkittävästi * TDD:n käyttö ei vaikuttanut millään lailla tuottavuuteen (ohjelma toimitettiin ajallaan) – TDD ei kasvattanut koodin koheesiota. Koheesio oli itse asiassa huonompi TDD:llä – Ei eroa koodin kompleksisuudessa
Siniaalto ja Abrahamsson (2008)	Puoli-teollinen	Kontrolloitu koe	17 ohjelmoijaa, joista 15 opiskelijaa: 8 TDD:llä + 9 ITL:llä	ITL	+ TDD pienensi koodin kompleksisuutta – Ei tilastollisesti merkittävää eroa kytkennässä tai koheesiossa – TDD näyttäisi kasvattavan pakettien välisiä riippuvuuksia, jonka johdosta niiden muuttaminen ja ylläpito voi olla hankalampaa
Bhat ja Nagappan (2006)	Teollinen	Kontrolloitu koe	4 projektia: 25–28 kehittäjää	Ei-TDD	+ Virheiden määrä väheni TDD:llä 62–76 % – Projektin johto arvioi TDD:n vaatineen 15–35 % enemmän aikaa
Lui (2004)	Teollinen	Case-tapaus	2 tiimiä TDD:llä ja 3 tiimiä ei-TDD:llä	Ei-TDD	+ TDD paransi tehtävien työmääräarvioita + TDD paransi projektien seurantaa + TDD paransi ohjelmoijan kuria: he noudattivat paremmin sovittua prosessia ja tätä pystyttiin paremmin tarkkailemaan + TDD paransi ohjelmien laatua: virheiden määrä väheni ja virheet pystyttiin korjaamaan paljon nopeammin

jatkuu seuraavalla sivulla...

Tutkimus	Tutkimuksen tyyppi	Havainnot pohjautuvat	Tutkittavat	Vertailukohde	Tärkeimmät havainnot
Maximilien ja Williams (2003); Williams et al. (2003)	Teollinen	Case-tapaus	9 ohjelmoijaa	Ad-hoc-yksikkötestaus	+ TDD vähensi virheiden määrää ohjelmistossa 40–50 % + TDD:n avulla koodin integrointi parani: päivittäinen integrointi esti viime hetken integrointiongelmat * Edellä mainitut tulokset saavutettiin ilman suurempia vaikutuksia tuottavuuteen
Sanchez et al. (2007)	Teollinen	Case-tapaus	9–17 ohjelmoijaa	Aiemmat versiot TDD:llä	+ Virheiden määrä ohjelmistossa pieneni entisestään ja oli selvästi alle standardien + TDD näyttäisi hillitsevän koodin kompleksisuuden kasvua ohjelmiston kehittyessä eteenpäin

Taulukko B.1: Yhteenveto TDD:n empiirisistä tutkimuksista