Test Driven Development- menetelmän tehokkuus ja laatu

Petri Pihlajaniemi

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 17. toukokuuta 2015

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Matemaattis-luonnontieteellinen	Tietojenkäsittelytieteen laitos

Tekijä — Författare — Author
Petri Pihlajaniemi

Työn nimi — Arbetets titel — Title

Test Driven Development- menetelmän tehokkuus ja laatu

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	17. toukokuuta 2015	19

Tiivistelmä — Referat — Abstract

Test Driven Development (TDD, testivetoinen kehitys) on ohjelmistokehitysmenetelmä, jossa testit kirjoitetaan ennen ohjelmakoodia. TDD on suosittu ketterä menetelmä, ja sen toimivuudesta löytyy runsaasti tutkimustietoa.

Test Driven Developmentissa kehittäjä noudattaa prosessia, jossa ensin kirjoitetaan vaatimusmäärittelyyn liittyvän tehtävän testi, testi ajetaan, kirjoitetaan testi ajetaan ja koodi siistitään. Kannattajien mukaan menetelmä takaa korkean testikattavuuden ja parantaa ohjelman rakennetta.

Tässä tutkielmassa TDD:n laatua ja tehokkuutta on mitattu arvioimalla ja vertaamalla 11 testivetoisesta kehityksestä tehtyä empiiristä tutkimusta. Tutkimuksia ei ole valikoitu minkään tutkimusaluetta tarkemman kriteerin perusteella, vaan otoksesta on pyritty tekemään mahdollisimman yleinen.

Laatu on määritelty ISO9126-standardin avulla, ja eri tutkimusten tuloksia on pyritty asettelemaan kategorioihin soveltaen standardin määrittelemiä laadun osa-alueita. TDD:n tutkimuksen kannalta tärkeimmiksi laatukategorioiksi nousivat luotettavuus ja ylläpidettävyys.

Tutkimusten yhteenvedon mukaan TDD parantaa luotettavuutta, mutta ylläpidettävyydessä tulos on neutraali. Taloudellisessa tehokkuudessa testivetoisen kehityksen havaittiin parantavan tehokkuutta.

ACM Computing Classification System (CCS):

Software and its engineering~Software development techniques
Software and its engineering~Agile software development

Avainsanat — Nyckelord — Keywords

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Sisältö

1 Johdanto

Ohjelmistotestauksen tarkoituksena on parantaa ohjelman laatua havaitsemalla ja poistamalla virheitä ohjelmakoodissa. Edsger Dijkstran sanontaa mukaillen: testauksen avulla ei voida todistaa ohjelman olevan virheetön, mutta sillä voidaan todistaa virheiden olemassaolo [RC+69]. Ohjelmaa testataan erilaisilla syötteillä, jonka jälkeen ohjelman toimintaa verrataan odotettuun oikeaan lopputulokseen. Jos lopputuloksissa on eroa, on testi löytänyt virheen [Muc08].

Testien kirjoittajan on tunnettava ohjelman rakenne ja toiminta pystyäkseen suunnittelemaan ja kirjoittamaan testejä. Testin kirjoittamiseni on vaikea ja aikaa vievä prosessi, ja se vaatii kehittäjältä hyviä taitoja [Whi00].

Ohjelmiston testaus suoritetaan eri vaiheissa riippuen valitusta ohjelmistokehitysmenetelmästä. *Vesiputousmalli* (waterfall model) kuvaa ohjelmistokehityksen vaiheisiin eroteltuna prosessina: ensin analysoidaan vaatimukset ja suunnitellaan koko ohjelma tarkasti, sitten kirjoitetaan varsinainen ohjelmakoodi ja lopuksi tuotettu ohjelmakoodi testataan. Vaiheittaisissa menetelmissä (Incremental Model) edellinen vesiputousmalli on jaettu useisiin pienempiin palasiin; koko ohjelmaa ei siis tarvitse suunnitella ja toteuttaa kerralla noudattaen vesiputousmallin järjestystä, vaan ohjelman rakentuu pienemmistä osista jotka on suunniteltu ja toteutettu erikseen. Stoican ja kumppanien [SMGM13] mukaan testaus on helppoa inkrementaalisissa menetelmissä, kun taas vesiputousmallissa testauksessa havaittujen virheiden korjaaminen voi olla hankaa. Ketterät ohjelmistokehityksen menetelmät (agile models) perustuvat inkrementaaliseen malliin. Yksi erityisesti testaamiseen keskittyvät toimintapa on ketteriin malleihin perustuva *Test Driven Development* [Cri06].

Tässä tutkimuksessa tarkastellaan TDD:n toimivuutta kahdella mittarilla: laadulla ja tehokkuudella. Mittareita on arvioitu 11 empiirisen tutkimuksen tuloksia tarkastelemalla.

Luvussa 2 käsitellään miten termiä laatu ja tehokkuus on käytetty tutkimuksessa ja mitä ne oikeastaan tarkoittavat, sekä esitellään ISO9126-laatustandardi. Luvussa 3 Test Driven Development- menetelmää ja sen käyttöä on kuvattu tarkemmin. Luvussa 4 on tarkasteltu empiirisiä tutkimuksia menetelmän vaikutuksista.

2 Laadun ja tehokkuuden käsitteistä

Näkemys ohjelmiston laadusta riippuu siitä, keneltä sitä kysytään. Ohjelman loppukäyttäjä saattaa olla aivan eri mieltä tietyn ohjelman laadusta verrattuna ohjelmojaan. Tavallista kuluttajaa tuskin kiinnostaa esimerkiksi ohjelmakoodin luettavuus, mutta toisaalta se on muille ohjelmoijille tärkeä laatuominaisuus. Ohjelmistokehityksen laadun käsitettä määrittäessä tärkeää

on myös tiedostaa erilaiset näkökulmat.

Barbara Kitchenham ja Shari Lawrence Pfleeger [KP96] esittelivät vuonna 1996 viisi laatunäkökulmaa: transendentin, käyttäjän, teollisuuden, tuotteen ja arvon näkökulman. Näkökulmat perustuvat David Garvinin näkemyksiin laadun käsitteestä muilla aloilla, esimerkiksi filosofiassa.

Transendentti näkökulma (transcendental view) pitää laatua mahdottomana määrittää tarkasti, mutta kuitenkin havaittavana ominasuutena [KP96]. Tämän näkökulman perusteella laatua on vaikea mitata, vaan se on lähinnä filosofinen käsite, jota voidaan arvioida vain subjektiivisten kokemusten perusteella

Käyttäjän näkökulma (user view) näkee tuotteen sen käyttäjän perspektiivistä. Laadukas tuote toteuttaa käyttäjän sille asettamat vaatimukset ja on vaivattomasti käytettävä [KP96]. Näkökulma vastaa parhaiten tavallista loppukäyttäjää, ohjelman toimivuus ja soveltuvuus tilanteeseen on tärkeintä. TDD:n käsittely tästä näkökulmasta on hankalaa: jos ohjelma on toimiva, ei loppukäyttäjälle ole merkitystä millaista toimintatapaa ohjelmoija on käyttänyt.

Teollinen näkökulma (manufacturing view) käsittelee tuotteen kehitysprosessia. Standardeja käyttävä ja vaatimukset huomioiva ohjelmistotuotantoprosessi johtaa laadukkaaseen tuotteeseen [KP96]. Perspektiivi nostaa ohjelman kehittäjien noudattamat työtavat tärkeään asemaan, mutta Kitchenhamin ja Pfleegerin mukaan näkökulma on ongelmallinen: liiallinen standardien noudattaminen voi johtaa huonoihin tuloksiin. Standardien noudattamista käytetään usein myyntivalttina, joten teollisesta näkökulmasta on tullut merkittävin perspektiivi ohjelmistokehityksessä [CSG07]. TDD:n käyttö edellyttää menetelmän määrittämiä toimintatapoja, joten tämän näkökulman huomiointi on perusteltua.

Tuotteen näkökulma (product view) tutkii ohjelmaa itseään. Näkökulman mukaan ohjelmakoodin sisäiset laatuominaisuudet vaikuttavat suoraan ohjelman ulkoiseen laatuun. Näkökulman puolestapuhujien mukaan näkökulmaa on helpoin mitata, koska koodin ominaisuuksia voidaan mitata objektiivisesti [KP96]. Näkökulmaa voi pitää TDD:n arviointiin parhaimpana perspektiivinä, koska menetelmä vaikuttaa sen kannattajien mukaan eniten juuri koodin rakenteeseen.

Arvon näkökulma (value-based view) huomio tuotteen arvon olevan riippuvainen sitä käyttävän ryhmään näkökulmasta. Ryhmillä on erilaiset käsitykset laadusta, joten voi olla vaikeaa tasapainottaa esimerkiksi käyttäjän ja teollisuuden näkökulmaa. Eri näkökulmia tasapainottamalla pyritään tuotteeseen, josta asiakas suostuu maksamaan mahdollisimman paljon valmiista ohjelmasta [KP96].

2.1 ISO 9126

International Organization for Standardization eli ISO julkaisi vuonna 1991 ohjelmistokehitystä käsittelevän ISO/IEC9126 [ISO01] laatumallin. Malli toteuttaa Kitchenhamin ja Pfleegerin perspektiivit [CSG07] ja se on yksi suosituimmista laatustandardeista [BBC+04]. ISO9126:n perustuu myös uudempi ISO25010-standardi, joka tunnetaan myös nimellä SQuaRE [ISO11].

ISO9126 jakaa ohjelman laatuominaisuudet luokkiin joilla on alaluokkia. Alaluokkiin sisältyy erilaisia ominaisuuksia, joita voidaan mitata metriikoilla. Luokkien, alaluokkien ja ominaisuuksien kokonaisuus muodostaa monikerroksisen hierarkian [MMR14]. Nämä ominaisuudet on jaettu kolmeen osaan, sisäiseen laatuun, ulkoiseen laatuun ja käyttölaatuun. Standardin mukaan osat muodostavat kokonaisuuden, jossa eri osissa tehdyt ratkaisut vaikuttavat ja riippuvat muiden osien laatuun, esimerkiksi ulkoinen laatu on riippuvainen sisäisestä laadusta.

Sisäisellä laadulla tarkoitetaan koodin laatuominaisuuksia, esimerkiksi rakennetta ja luettavuutta. Ulkoinen laatu viittaa ohjelman toimintaan sitä ajettaessa, esimerkiksi virheiden määrään. ISO9126 jakaa sisäisen ja ulkoisen laadun osien luokiksi toimivuuden, luotettavuuden, käytettävyyden, tehokkuuden, ylläpidettävyyden ja siirrettävyyden. Kaikkiin luokkiin sisältyy useita alaluokkia.

Toimivuus (functionality): ohjelman kyky vastata sille asetettuihin vaatimuksiin. Luokan alaluokkia ovat soveltuvuus tehtävään, ohjelman antamien tuloksien oikeellisuus, toiminta muiden ohjelmien kanssa, turvallisuus sekä standardien, kuten lakien, noudattaminen. TDD:n tapa koodata testit ensin vaikuttaa ohjelman rakenteeseen, joten se saattaa vaikuttaa myös sen toimivuuteen.

Luotettavuus (reliability): kuinka ohjelmisto säilyttää suoritustason erilaisissa tilanteissa. Alaluokkina kypsyys, eli virheistä johtuvien häiriöiden välttäminen, virheiden sietäminen, virheistä palautuminen sekä luotettavuuteen liittyvien standardien noudattaminen. Yksi TDD:n tärkeimmistä ominaisuuksista on ennen koodia kirjoitetuista testeistä johtuva testikattavuus, joten menetelmä vaikuttanee luotettavuuteen.

Käytettävyys (usability): kuvaa ohjelman helppokäyttöisyyttä ja miellyttävyyttä. Käytettävyyden alaluokat ovat ymmärrettävyys, ohjelman opittavuus, käytettävyys, viehättävyys ja käytettävyysstandardien noudattaminen. Testivetoinen kehitys ei ota kantaa käyttöliittymän käytettävyyskysymyksiin, joten epäilen TDD:n vaikuttavan mitenkään käytettävyyteen.

Tehokkuus (efficiency): kuinka hyvin ohjelma toimii erilaisilla resursseilla. Luokan alaluokkia ovat ajankäyttö, eli kuinka ohjelman vasteaika muuttuu erilaisissa ympäristöissä, resurssien hyödyntämisen tehokkuus sekä tehokkuusstandardien noudattaminen. TDD:n käyttö parantaa todennäköisesti ohjelman rakennetta, joka saattaa vaikuttaa hieman myös tehokkuuteen.

Ylläpidettävyys (maintainability): mahdollisuudet muuttaa ohjelmaa.

Alaluokkina analysoitavuus, muutettavuus, stabiilisuus, testattavuus ja ylläpidettävyysstandardien noudatus. Testivetoinen kehitys pakottaa kehittäjän suunnittelemaan metodeita pieninä osina välttäen liiallista toiminnallisuutta ja refaktoroimaan koodia, tämä saattaa parantaa myös ylläpidettävyyttä.

Siirrettävyys (portability): ohjelman siirrettävyys alustalta toiselle, esimerkiksi eri käyttöjärjestelmälle. Sopeutuvuus, eli kuinka helposti ohjelmaa voidaan muuttaa toiselle alustalle sopivaksi, asennettavuus, rinnakkaiselo muiden ohjelmien kanssa, korvattavuus, eli toiminta jonkun toisen ohjelman tilalla, sekä siirrettävyysstandardien noudatus. Testivetoisen kehityksen menetelmä ei todennäköisesti vaikuta tähän ominaisuuteen käytännössä mitenkään, koska sen prosessin määrittelemät toimintavat eivät käsittele siirrettävyyteen liittyviä ohjelmakoodin ominaisuuksia.

Käyttölaatu (quality in use) viittaa ISO:n standardissa valmiin tuotteen käyttöön. Tuotteen loppukäyttäjän kokemusta mitataan tehokkuudella, tuottavuudella, turvallisuudella ja tyytyväisyydellä.

2.2 Laadun mittareita

Erilaisia tapoja mitata laatua esiintyy kymmeniä erilaisia tieteellisissä julkaisuissa. Shyam R. Chidamber ja Chris F. Kemerer [CK94] esittelivät vuonna 1994 olio-ohjelmoinnin arviointiin kehittelemänsä kuusi mittaria.. Tibor Gyimóthy, Rudolf Ferenc, ja István Siket ovat puolestaan arvioineet näiden mittareiden toimivuutta avoimen lähdekoodin Mozilla-ohjelmistopaketin virheiden ennustamiseen [GFS05]. Michaelin ja kumppanien mukaan Chidamberin ja Kemenerin mittarit ovat yleisimmin viitatut olio-ohjelmoinnin mittarit [MBS02].

- **Metodeita per luokka** (Weighted Methods Per Class). Mittaa metodien määrää luokassa. Metodien määrä ja monimutkaisuus vaikuttaa luokan ylläpidon hankaluuteen ja suuren määrän metodeita sisältävä luokka on hankalampi käyttää uudelleen [CK94]. Gyimóthy ja kumppanig havaitsivat luokan jolla määrä on korkea olevan virhealttiimpi kuin luokan jolla määrä on matala [GFS05].

- **Perintäpuun syvyys** (Depth of Inheritance Tree). Kuinka monta ylempää luokkaa luokalla on. Mitä syvemmällä perintäpuussa luokka on, sen enemmän se perii metodeita. Syvät puut ovat monimutkaisempia ja alttiimpia virheille [CK94]. Perintäpuussa syvällä oleva luokka on virhealttiimpi, mutta se ei ole yhtä hyvä mittari kuin jotkut muista tutkituista [GFS05].

- **Lasten määrä** (Number of Children). Kuinka monta alaluokkaa luokalla on. Luokka jolla on paljon lapsia, vaatii paljon testausta. Luokka jolla on paljon aliluokkia on koodin uudelleenkäyttöä, toisaalta jos aliluokkia on paljon voi kyseessä olla aliluokkien väärinkäyttö [CK94].

Lasten suuren määrän ei havaittu olevan merkittävä mittari virheiden ennustamiseen [GFS05].

- **Luokkien väliset kytkennät** (Coupling Between Object Classes). Luokat ovat kytketty, jos ne käyttävät toistensa metodeja tai muuttujia. Liiallinen kytkentä heikentää modulaarista suunnittelua ja vaikeuttaa uudelleenkäyttöä. Moneen luokkaan kytkettyä luokkaa pitää testata enemmän [CK94]. Kytkentöjen vähäinen määrä oli merkittävä, Gyimothyn ja kumppanien tutkimuksessa merkittävin ja paras, mittari [GFS05].

- **Vastausten määrä** (Response For a Class). Kuinka monta metodia ajetaan vastauksena luokalle tulleeseen viestiin. Virheiden korjaus on hankalaa, jos ajettavia metodeita on paljon ja luokka on monimutkainen [CK94]. Suuren määrän vastauksia sisältävä luokka oli merkittävästi virhealttiimpi [GFS05].

- **Metodien välisen koheesion puute** (Lack of Cohesion in Methods). Mittaa luokasta löytyviä metodeita joilla ei ole yhteisiä muuttujia. Tavoite on siis se, että saman luokan metodit käsittelevät samoja muuttujia. Jos mittari havaitsee heikon koheesion, olisi kannattavaa pilkkoa luokka pienemmiksi osiksi [CK94]. Matalan koheesion luokat havaittiin virhealttiimmiksi [GFS05].

- **Rivien määrä** (Lines of Code). Luokan koodirivien määrä. Luokka jossa on suuri määrä koodirivejä on virhealttiimpi. Mittari oli toisiksi paras Gyimothyn ja kumppanien tutkimista mittareista, paljon koodirivejä sisältävissä luokissa oli enemmän virheitä [GFS05].

2.3 Tehokkuuden arviointi

Tehokkuutta voidaan tarkastella testien perusteella; tehokas testimenetelmä löytää mahdollisimman suuren osan ohjelman virheistä pienimmällä mahdollisella vaivalla. Useissa ohjelmistokehitystä ja testausta käsittelevissä artikkeleissa on kuitenkin käytetty kahta eri termiä, effectiveness ja efficiency. Effectiveness viittaa kykyyn saavuttaa lopputulos, efficiency taasen lopputuloksen saavuttamiseen vaadittuun aikaan ja vaivaan. Tässä tutkimusessa onkin eroteltu nämä kaksi merkitystä tehokkuudeksi ja taloudellisuudeksi. Tehokkuudella tarkoitan esimerkiksi testien kattavuutta ja taloudellisuudella testien kattavuutta verrattuna aikaan ja vaivaan.

Rothermel ja Harrold määrittelevät taloudellisuuden regressiotestaustekniikoita käsittelevässä artikkelissaan [RH96] laskentatehon kulutukseksi. Tällaisen määritelmän tarkoituksenmukaisuus riippuu ohjelman koosta: pienessä ohjelmassa testien ajamisen nopeus tuskin on kovin merkittävä tekijä, mutta toisaalta jos ohjelma on suuri voi hitaiden testien ajaminen kestää

kohtuuttoman kauan. Eldh ja kumppanit laajentavat Rothermelin ja Harroldin termin käsittämään myös testitapauksen ymmärtämisen ja luomiseen kuluvan ajan [EHP+06].

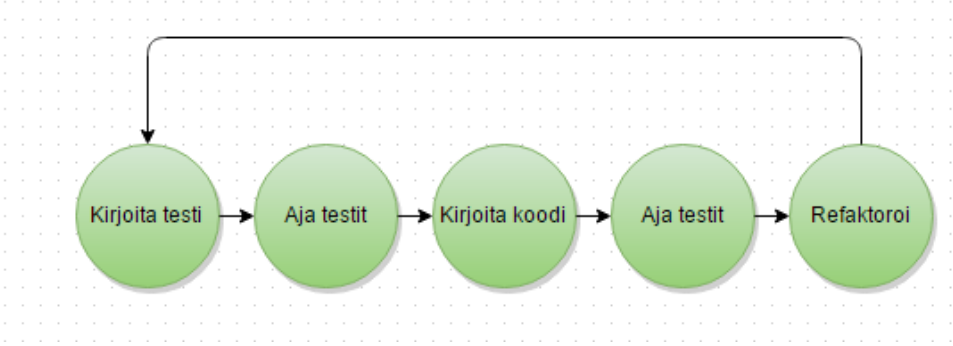
Rothermel ja Harrold käyttävät myös termejä sisältyvyys (inclusiveness), tarkkuus ja yleisyys. Sisältyvyydellä tarkoitetaan kuinka hyvin käytetty ohjelmistotestaustekniikka valitsee virheiden löytämiseen sopivat testi. Tarkkuus mittaa kuinka paljon tekniikka pystyy välttämään turhien, virheitä löytämättömien, testien kirjoittamista. Yleisyys viittaa tekniikan kykyyn toimia erilaisissa tilanteissa, esimerkiksi käyttäessä monimutkaisia rakenteita.[RH96]

Frøkjær ja kumppanit [FHH00] määrittelevät tehokkuuden käyttäjän ohjelmalla saavuttamien tavoitteiden tarkkuudeksi ja täydellisyydeksi , heidän artikkelinsa tosin käsittelee termejä käytettävyyden perspektiivistä. Eldhin ja kumppanien näkemys tehokkuudesta testauksen näkökulmasta on yksinkertaisesti tekniikan löytämien virheiden määrä [EHP+06].

Yleisesti ottaen termit tehokkuus ja taloudellisuus tuntuvat tarkoittavan eri asioita eri artikkeleissa. Tätä tutkimusta varten kerätyissä TDD:tä käsittelevissä tutkimuksissa kuitenkin tehokkuudella on useimmiten tarkoitettu vertailua kulutetun ajan ja testien kattavuuden suhteesta.

3 Test Driven Development

Test Driven Development on Kent Beckin "löytämä" ohjelmistokehitystekniikka [Bec03]. Beck itse kertoo Quora-sivustolla [Quo] käyttävänsä sanaa "löytämä", koska hän löysi kuvauksen hyvin samankaltaisesta menetelmästä 60-luvun ohjelmointikirjasta. Menetelmästä käytetään myös termiä Test First Development [Cri06]. Suomenkieliseksi vastineeksi TDD:lle on muodostunut testivetoinen kehitys, mutta myös termiä testilähtöinen kehitys käytetään. Menetelmän perusideana on luoda ensin ohjelmalle testitapaus, ja sen jälkeen luoda testitapauksen läpäisevä osa ohjelmaa. Menetelmä on osa ketteriä menetelmiä, ryhmää ohjelmointitekniikoita joissa kehitys tehdään lyhyissä osissa, Tekniikan kulku Beckin menetelmässä on seuraava:



Kuva 1: TDD:n toiminta yksinkertaistettuna.

1. Kehittäjä tutkii esimerkiksi vaatimusmäärittelyn käyttötapauksia, ja kirjoittaa niiden perusteella testin.

2. Kehittäjä ajaa testit ja varmistaakseen antaako ohjelma testin vaatiman lopputulokset ilman muutoksia. Testi saattaa myös mennä läpi vaikkei kehittäjä ole mielestään kirjoittanut koodia kyseiselle tapaukselle, tällöin laadittu testi saattaa olla virheellinen antaen aina hyväksytyn tuloksen.

3. Jos testejä ei läpäisty, siirrytään kirjoittamaan koodia. Kehittäjä täydentää ohjelmaa antamaan hyväksyttävän tuloksen hylättyyn testiin. Lisätyn koodin pitäisi keskittyä ainoastaan testin määrittelemiin toimintoihin, tarkoitus ei ole kirjoittaa muuta toiminnallisuutta.

4. Testit ajetaan uudestaan. Jos testi yhä epäonnistuu, palataan kohtaan kolme ja koodi parannellaan. Jos testi läpäistään, siirrytään seuraavaan kohtaan.

5. Kirjoitettu koodi refaktoroidaan. Koodista esimerkiksi poistetaan turha toisto ja mahdollisesti ylipitkiksi kasvaneet metodit pilkotaan. Tämän jälkeen palataan ensimmäiseen kohtaan ja prosessi aloitetaan alusta uudella testillä.

Testivetoisen kehityksen testit kirjoitetaan tutkimaan ohjelman pienimpiä mahdollisia yksikköjä, eli ohjelman osia joita voidaan testata [Cri06]. Crispinin mukaan pienimmän mahdollisen yksikön määritelmä on kiistanalainen, mutta olio-ohjelmoinnissa sitä on yleisesti pidetty metodina. Testien kirjoittaminen vasta varsinaisen koodin jälkeen voi kestää minuuteista kuukausiin, kun taas test driven developmentia käyttäessä testit ovat käytettävissä välittömästi. [JS05].

Testivetoisen kehitys siis perustuu testaukseen, mutta se viekkaasti myös vaikuttaa suunnitteluun. Koska testit kirjoitetaan ensin, ohjelmoija joutuu miettimään miten toteuttaa koodi siten, että se on myös testattava. Menetelmä myös kehottaa refaktoroimaan koodia säännöllisesti jokaisen kierroksen lopussa.

TDD:llä on myös muutamia samankaltaisia sukulaisia, esimerkiksi Acceptance Test Driven Development, Behavior-Driven Development ja Story-Test-Driven Development [TLD+10], mutta niitä ei käsitellä tässä tutkimuksessa tarkemmin.

4 TDD:n vaikutus laatuun

Test Driven Development-menetelmästä on tehty useita tutkimuksia. Esimerkiksi Rafique ja kumppanit ovat tutkineet 27 empiiristä ennen helmikuuta 2011 julkaistua tutkimusta aiheesta [RM13]. Turhan ja kumppanit löysivät

peräti 325 artikkelia, joista he siivilöivät 22 tarkoituksenmukaista tutkimusta [TLD+10].

Tässä tutkielmassa käsitellään 11 empiiristä tutkimusta TDD:n vaikutuksesta testiryhmän tekemän ohjelman laatuun ja taloudellisuuteen, ja niiden tuloksia arvioidaan seuraavissa kappaleissa. Tutkimuksia ei ole valikoitu esimerkiksi testiryhmän taustan tai kirjoitetun ohjelman ominaisuuksien perusteella, vaan tarkoituksena on ollut valita mahdollisimman yleinen otos. TDD:stä kirjoitettujen artikkelien tuloksia luokitellaan ISO 9126:n määrittelemien sisäisen ja ulkoisen laadun kategorioita soveltamalla, vaikka erilaisia tuloksia onkin vaikea lokeroida luokkien sisälle: onko esimerkiksi virheiden määrä ohjelmassa luotettavuuteen vai toimivuuteen liittyvä termi?

4.1 Toimivuus

Yksikään yhdestätoista tutkimuksesta ei arvioinut tuotetun ohjelman toimivuutta. Yksi ongelma on englanninkielinen termi functionality, jolla usein TDD:tä käsittelevissä artikkeleissa viitataan ohjelman osien, kuten metodien, toimintaan. ISO:n laatustandardi kuitenkin määrittelee toimivuuden ohjelman toimivuudeksi, joten tutkimusten tapa käyttää termiä ei sovellu selkeästi tähän luokkaan. TDD:n väitetyt parannukset ohjelman rakenteen laatuun saattavat vaikuttaa toimivuuteen, mutta mielestäni tutkimukset aiheesta asettuvat loogisemmin muihin laatukategorioihin.

4.2 Luotettavuus

Luotettavuus vaikuttaa olevan suosituin laadun osa testivetoista kehitystä käsittelevissä artikkeleissa, ja sitä on tutkittu kymmenessä yhdestätoista tutkielmassa käsitellystä tutkimuksesta. Useat tutkimukset ovat mitanneet TDD:llä tuotetun ohjelmiston laatua testien läpäisyn perusteella, jota voidaan pitää luotettavuuteen liittyvänä tekijänä. Toisaalta ISO:n laatuluokitusten mukaan luotettavuus voidaan ymmärtää myös tarkoittavan virheistä palautumista, eikä niinkään virheiden vähyyttä. ISO 9126-3 kuitenkin esittää koodikattavuuden ja katselmoidusta koodista havaittujen virheiden määrän luotettavuuden mittareina, joten virheiden havaitsemiseen perustuvat tutkimukset ovat tässä osiossa perusteltuja.

Georgen ja Williamsin [GW04] tutkimuksessa havaittiin TDD:tä käyttäneiden parien tuottamien ohjelmien läpäisevän 18 prosenttia enemmän tutkijoiden rakentamia testitapauksia kuin kontrolliryhmien. Tutkimuksessa oli kuitenkin ongelmia joiden takia tulosta voidaan pitää kyseenalaisena: Georgen ja Williamsin tutkimuksessa ohjelmat olivat erittäin lyhyitä, pituudeltaan n. 200 riviä, ja suurin osa kontrolliryhmistä ei ollut tehnyt minkäänlaisia testejä ennen ohjelman palautusta tutkijoille.

Wilkersonin ja kumppanit [WNM12] vertailevat TDD:tä koodikatselmointimenetelmään, jossa erillinen ryhmä arvioi koodia. Tutkijoiden mukaan mene-

telmillä on selvä ero, katselmointi poistaa ohjelmasta löytyviä virheitä, mutta TDD käsittelee ne jo kehitysvaiheessa. Tutkimuksen tulosten mukaan katselmointia ja TDD:tä yhdistävän ryhmän ohjelmassa oli vähiten virheitä kun taas pelkkää TDD:tä käyttäneellä ryhmällä tulos ei ollut perinteisiä menetelmiä parempi. Wilkersonin ja kumppanien mukaan tutkimuksen neutraali tulos saattaa selittyä TDD-menetelmän liian epämääräisellä määrittelyllä.

Erdogmus, Morisio ja Torchiano [EMT05] tutkivat testivetoista kehitystä verrattuna testien kirjoittamiseen jälkeenpäin (test-last). Tutkimuksen osaanottaneista opiskelijoista TDD:tä käyttäneet olivat kirjoittaneet keskimäärin 28 prosenttia enemmän testejä. Ohjelman laadussa testivetoinen kehitys olikin yllättäen jälkeenpäin testausta huonompi: test-last ryhmän mediaani virheiden määrässä käytettäessä tutkijoiden laatimaa testipakettia oli kymmenen prosenttia pienempi. Tutkijoiden ehdottama selitys TDD:n hieman heikomman laadun johtumisesta ohjelman pienuudesta ja koehenkilöiden erilaisista taitotasoista on perusteltu. Testihenkilöiden tuottama ohjelma oli keilauksen pistelaskujärjestelmä, joka on sen verran yksinkertainen tehtävä, että virheet pystyy helposti löytämään myös ilman yhtäkään automaattista testiä. Causevicin ja kumppanien [CSP12] tutkimus oli menetelmiltään hyvin samankaltainen kuin Erdogmusin ja kumppanien. Tässäkin tutkimuksessa vertailtiin keilailun pistelaskua suorittavan ohjelman testauksen laatua Test-First ja Test-Last menetelmillä käyttäen testiryhmänä yliopisto-opiskelijoita. Causevicin ja kumppanien tutkimuksessa molempien ryhmien testien laatu oli hyvin samankaltainen, eikä merkittäviä eroja havaittu. Itse ohjelman koodin laadussa oli kuitenkin ero TDD:n hyväksi. Vaikka tutkimukset olivat lähes identtiset, oli tuloksissa pieni ero. Todennäköisesti ohjelman ja koeryhmien pienuus vaikuttaa Causevicin ja Erdogmusin tutkimuksissa tulokseen enemmän kuin ohjelmointimenetelmä.

Canfora ja kumppanit [CCG+06] ovat testanneet menetelmää Espanjalaisen IT-yrityksen ammattilaisilla. Tutkijoiden mukaan TDD:n testit eivät ole tilastollisesti merkittävästi tarkempia vesiputousmallilla tuotettuihin verrattuna. Canfora ja kumppanit kuitenkin toteavat olevansa "vakuuttuneita"TDD:n positiivisesta vaikutuksesta laatuun, ja uskovat tulokset olevan merkittävä pidemmässä tutkimuksessa.

Pancur ja Ciglaric [PC11] havaitsivat TDD:n tuottavan paremman päätöskattavuuden kuin iteratiivinen Test-Last menetelmä. Mutaatiotestauksessa ja hyväksyttyjen testien määrässä ei havaittu eroa. Myös Munirin ja kumppanien [MWPM14] tutkimuksessa päätöskattavuus oli parantanut, mutta tulos ei ollut tilastollisesti merkittävä johtuen liian pienestä koeryhmästä.

Nagappan ja kumppanit [NMBW08] tutkivat TDD:n vaikutusta laatuun ja tehokkuuteen IBM:n ja Microsoftin ohjelmistotuotannossa. Tutkijat määrittelivät laadun koodin virhetiheydeksi, ja virheiden määrä verrattuna samankaltaisiin projekteihin joissa TDD:tä ei oltu käytetty oli selvästi matalampi. IBM:n projekteissa vähennys oli 40% ja Microsoftin projekteissa 60-90%. Tulos on erittäin positiivinen, ja vaikutus selvästi suurempi kuin

muissa tässä artikkelissa käsitellyissä tutkimuksissa. Tutkijoiden mukaan tulos saattaa selittyä esimerkiksi projektien vaikeustasossa: TDD ryhmille on saattanut osua helpompia projekteja.

4.3 Käytettävyys

Testivetoinen kehitys tuskin vaikuttaa loppukäyttäjän kokemukseen ohjelman laadusta, koska TDD:n prosessi ei varsinaisesti ota kantaa käytettävyyteen. Hellman ja kumppanit [HHKM10] pitävät graafisen käyttöliittymän kehitystä TDD:llä niin vaikeana, että ovat artikkelissaan pyrkineet kehittämään TDD:stä paremmin tehtävään sopivaa muunnosta.

Yksikään tutkielmaan valituista artikkeleista ei ottanut kantaa käytettävyyteen, joten vertailenkin tässä osiossa itse menetelmän käytettävyyttä. TDD:n prosessi on yksinkertainen, joten todennäköisesti aloittelevakin ohjelmoija ymmärtää sen toimintatavat nopeasti. Useissa tutkimuksissa käyttäjien omia näkemyksiä menetelmästä on selvitetty, mutta tähän osioon on valikoitu vain tutkimuksia joissa TDD:n käytön oikeellisuutta on mitattu muutenkin kuin käyttäjien mielipiteiden perusteella.

Latorren [Lat14] tutkimuksessa mittauksien mukaan menetelmä oli helppo oppia, varsinkin kehittyneet ja keskitason ohjelmoijat oppivat sen nopeasti.

Oulun yliopiston Fucci, Turhan ja Oivo [FTO14] tutkivat TDD:n prosessin noudattamisen tarkkuuden vaikutusta laatuun. Fuccin ja kumppaneiden mukaan TDD:tä koskevissa tutkimuksissa menetelmän seuraamisen arviointi ei ole tehty tarpeeksi, vaikka se saattaa oleellisesti vaikuttaa tuloksiin. Tutkimuksen TDD:tä käyttäneiden opiskelijoiden tuottama laatu ei ollut ollut sidoksissa siihen, kuinka hyvin he noudattivat menetelmää. Tutkijat havaitsivat opiskelijoiden usein keskittyvän vain testien kirjoittamiseen ja jättävän refaktorointivaiheen kokonaan pois. Fucci ja kumppanit kyseenalaistavat TDD:n opettelun ja noudattamisen hyödyn, mutta pitävät mahdollisena hyötyjen tulevan esiin pidemmissä tutkimuksissa.

4.4 Tehokkuus

ISO 9126 mittaa tehokkuutta mittareilla kuten vasteaika ja muistinkäyttö. Tutkimus testivetoisen kehityksen vaikutuksesta ohjelmistojen tehokkuuten ei selvästi ole ollut kovin suosittu tutkimuskohde, koska yksikään yhdestätoista tutkimuksesta ei sitä noteerannut. Tutkimusta itse menetelmän tehokkuudesta löytyy runsaasti, mutta se asettuu paremmin omaan osioonsa TDD:n taloudellisuudesta. TDD:n toimintavat eivät suoranaisesti vaikuta ohjelman sellaiseen suunnitteluun, joka parantaisi ohjelman resurssienkäyttöä, vaan siihen tarvittaisiin tarkempia rakenteen suunnittelun ohjeita. TDD:n etuna esitetty selkeämpi ja modulaarinen rakenne tuskin suoraan vaikuta ISO-standardissa esiteltyjen tehokkuuden mittareiden tuloksiin.

4.5 Ylläpidettävyys

Testivetoisen kehityksen vaikutus ohjelman rakenteeseen vaikuttanee myös sen ylläpidettävyyteen, koska modulaarinen ja selkeä ohjelmakoodi on helpommin laajennettavaa ja korjattavaa. Ohjelman rakenteen mittaaminen tehokkaasti vaikuttaa tutkimusten perusteella vaikeammalta kuin esimerkiksi luotettavuuden mittaaminen; virheitä on helpompi laskea kuin arvioida rakenteeseen liittyviä ominaisuuksia. Koska mahdolliset uudet virheet löydetään helpommini kattavan testipaketin avulla kuin ilman, testien laatu ja luotettavuus ovat myös selkeästi ylläpidettävyyteen ja laajennettavuuteen liittyviä tekijöitä, joten näitä laadun osa-alueita koskevilla tutkimuksilla on yhteys.

Siniaalto ja Abrahamsson [SA07] tutkivat TDD:n vaikutusta ohjelmiston laatuun. Tutkijat käyttivät mittareina tässäkin tutkielmassa esiteltyjä Chidamberin ja Kemenerin laadun mittareita, poislukien rivien määrä. Siniaalto ja Abrahamsson vertasi TDD:llä tehtyä ohjelmaa kahteen ohjelmaan, joissa testaus oli tehty vasta koodin jälkeen. Ohjelmien tulokset olivat samankaltaisia, mutta TDD:n oli luokkien metodien välisen koheesion metriikassa selvästi muita huonompi.

Pancurin ja Ciglaricin tutkimuksessa [PC11] TDD tuotti paremman syklomaattisen kompleksisuuden verrattuna testaamiseen jälkeenpäin, mutta vaikutus oli liian pieni ollakseen tilastollisesti merkittävä. Syklomaattinen kompleksisuus on Thomas J. McCaben kehittämä mittari [McC76], joka arvioi ohjelman lähdekoodissa olevien erillisten polkun määrää. Korkea määrä polkuja on merkki rakenteesta, jota on McCaben mukaan vaikea testata ja ylläpitää. Munirin ja kumppanit [MWPM14] tarkastelivat myös McCaben mittaria, eikä ero verrattuna Test-Lastiin ollut tilastollisesti merkittävä.

4.6 Siirrettävyys

Valittuihin artikkeleihin ei osunut tutkimuksia joissa oltaisiin tutkittu TDD:n vaikutusta ohjelmiston siirrettävyyteen. Testivetoisen kehityksen prosessi ei sivua siirrettävyyteen vaikuttavia tekijöitä, joten tutkimuksen vähäisyys oli odotettavissa. Samaan tapaan kuin käytettävyyttä tutkiessa, onkin ehkä järkevämpää tutkia miten itse TDD:n tekniikka soveltuu eri alustoilla. Tästäkään aiheesta en ole löytänyt tutkimuksia, mutta luultavasti TDD toimii kaikilla alustoilla ja ohjelmointikielillä joissa voi kirjoittaa testejä. Tämä kriteeri on niin löysä, että TDD:tä voi todennäköisesti käyttää lähes missä tahansa ohjelmointitilanteessa.

4.7 Vaikutus taloudellisuuteen

George ja Williams [GW04] havaitsivat TDD:tä käyttäneiden pariohjelmoijien käyttäneen 16 prosenttia enemmän aikaa tehtävän suorittamiseen kuin tavanomaista vesiputousmalli noudattanut kontrolliryhmä. Toisaalta suurin

osa kontrolliryhmistä oli jättänyt testitapaukset tekemättä, joten tuloksesta ei voi tehdä suuria johtopäätöksiä.

Ohjelman rakenteeseen keskittyneessä tutkimuksessaan Siniaalto ja Abrahamsson [SA07] toteavat TDD:n tuottavan korkean testikattavuuden ilman tuottavuuden kärsimistä.

Erdogmus ja kumppanit [EMT05] havaitsivat testivetoista kehitystä käyttäneiden oppilaiden olleen tehokkaampia kuin jälkeenpäin testit rakentaneet oppilaat. Tehokkuuden kasvussa oli vaihtelua riippuen ohjelmoijan taidoista: positiivinen vaikutus tehokkuuteen oli suurin taitavilla ohjelmoijilla. Erdogmus ja kumppanit määrittelivät tehokkuuden hyväksyttävästi läpäistyjen käyttötapausten määränä verrattuna aikaan. Tutkimuksessa kuitenkin havaittiin myös pieni negatiivinen vaikutus ohjelman laatuun TDD:tä käyttäessä, joten käytetty laadun määritelmä ei ehkä ollut täysin tutkimukseen sopiva, vaan myös laatu olisi pitänyt laskea mukaan taloudellisuuteen.

Canforan ja kumppanit [CCG+06] tilastollisen analyysin mukaan testivetoinen kehitys vaatii enemmän aikaa kuin jälkeenpäin testaaminen, mutta tutkijat uskovat TDD:n paremman laadun parantavan taloudellisuutta.

Pancur ja Ciglaric [PC11] olivat tutkimuksessaan verranneet TDD:tä iteratiiviseen Test-Last menetelmään. Test-Last kontrolliryhmälle määrätty toimintapa muistutti hyvin paljon TDD:tä, ainoastaan testien ja koodin järjestystä prosessilla oli muutettu. Tutkijat päättelivät pieniin iteraatiohin perustuvan suunnittelun vaikuttavan merkittävästi tuloksiin, johon heidän kontrolliryhmälleen laatima prosessi kannustaa. Pancurin ja Ciglaricin tutkimusten mukaan vaikutus tehokkuuteen, eli tässä tapauksessa tuotettuihin testeihin per tunti, oli positiivinen, mutta liian pieni ollakseen tilastollisesti merkittävä.

Madeyski ja Szala [MS07] havaitsivat TTD:tä käyttävän ohjelmoijan olevan tehokkaampi kuin käyttäessä Test-Last menetelmää. Kiinnostavaa tässä tutkimuksessa on se, että koehenkilönä oli vain yksi kokenut ohjelmoija, joka toteutti ohjelmaa vaihdellen menetelmiä. Tutkimuksessa oli useita mielenkiintoisia piirteitä: yhden ohjelmoijan tehokkuuden mittaus pitkällä aikavälillä, tässä ohjelman toteutus kesti 122 tuntia, on kiinnostava näkökulma. Tutkijat myös mittasivat kuinka pitkän ajan käyttötapauksen kehityksestä ohjelmoija oli passiivinen eikä kirjoittanut koodia, eli kuinka pitkään hän luki koodia tai etsi bugeja. Yhden hengen koeryhmää ei mielestäni voi pitää mitenkään tieteellisesti uskottavana, joten haluaisin nähdä saman tutkimuksen isommalla koeryhmällä.

Nagappan ja kumppanit [NMBW08] tutkivat TDD:tä yrityskäytössä. TDD:tä käyttäneet ryhmät käyttivät 15 %-35 % enemmän aikaa projektin tuottamiseen, mutta laatu oli selkeästi parempaa. Tutkijoiden mukaan suuri parannus laadussa oli merkittävämpi kuin lasku tuotannon nopeudessa, joten menetelmä on taloudellisempi. Myös yrityksen ohjelmointiryhmät olivat samaa mieltä TDD:n paremmasta taloudellisuudesta.

4.8 Kehittäjien mielipiteitä menetelmästä

Eräissä artikkeleissa on tehty myös kvalitatiivista tutkimusta, esimerkiksi selvittämällä testihenkilöiden henkilökohtaisia näkemyksiä testivetoisen kehityksen toimivuudesta.

Georgen ja Williamsin [GW04] tutkimuksen osaanottajat pitivät menetelmää erityisen hyvänä koodin laadun ja ohjelmoijan tuottavuuden kannalta, mutta oikeaan "mielenlaatuun"pääsemistä oli pidetty vaikeana. Tutkimusryhmänä tutkimuksessa oli 24 ammattilaisohjelmoijaa.

Latorren [Lat14] tutkimuksessa testivetoisen kehityksen oppimisen vaativuudesta 24 ammattilaista 25:stä piti menetelmää helppona oppia, ja tutkijan mittausten perusteella he myös käyttivät sitä oikein.

Ljubljanan yliopistossa tehdyssä Pancurin ja Ciglaricin[PC11] tutkimuksessa menetelmää pidettiin vaikeasti opittavana verrattuna testien kirjoittamiseen jälkeenpäin. Koeryhmänä toimineet neljännen vuoden tietojenkäsittelytieteen opiskelijat eivät tutkijoiden mukaan mahdollisesti hallinneet menetelmää tarpeeksi hyvin kymmenen viikon harjoittelun jälkeen.

Munirin ja kumppanien [MWPM14] tutkimuksessa ammattilaiset eivät pitäneet TDD:stä. Kyselytutkimuksessa menetelmää pidettiin vaikeana oppia sekä vaativan enemmän aikaa ja itsekuria. Suurin osa 15:sta TDD:tä tutkimuksessa käyttäneestä koehenkilöistä ei ollut ennen käyttänyt menetelmää. Ammattilaiset pitivät testien kirjoittamista ensin ongelmallisena, koska kehittäjän ja asiakkaan näkökulmasta varsinainen ohjelman toiminnallisuus nähdään arvokkaampana kuin testit. Munirin ja kumppanien mukaan TDD:n tehokas käyttö vaatii harjoittelua ja hyvää testienkirjoitustaitoa.

4.9 Kirjallisuuskatsaukset

Lyhyessä kandidaatintutkielmassa ei tietenkään voi käsitellä kaikkia maailman tutkimuksia TDD:stä. Onneksi TDD:stä on tehty myös meta-analyysia, jota tutkimalla saamme paremman kuvan menetelmän vaikutuksista.

Rafiquen ja Misic [RM13] ovat analysoineet testivetoisen kehityksen vaikutusta laatuun ja tehokkuuteen. Tutkimuksessa laatu on määritelty virheiden määränä verrattuna koodin määrään ja tehokkuus tuotettuna koodina verrattuna aikaan, tutkimukset joissa on tutkittu TDD:tä muilla mittareilla on jätetty pois analyysista. Ohjelman rakenteen laatu on jätetty pois tutkitumuksesta, koska tutkijoiden mukaan suuunnittelun laatua on erittäin vaikea arvioida. Rafique ja Misic ovat tarkoituksella valikoineet ainoastaan tutkimuksia joissa TDD:tä on verrattu tavanomaisiin kehitysmenetelmiin, kuten vesiputousmalliin ja iteratiiviseen Test-Last menetelmään. Myös tutkimukset joissa TDD:n prosessia ei ole noudatettu tarkasti on jätetty pois analyysista. Tutkimukset oli tehty opiskelija- ja ammattilaisryhmillä.

Rafique ja Misic toteavat analyysin tuloksien osoittavan TDD:n tuottavan pienen parannuksen laatuun, mutta tehokkuudessa tulos ei ole ratkaiseva.

Tutkimus	Luotettavuus	Ylläpidettävyys	Taloudellisuus	Mielipide
[GW04]	+		=	+
[WNM12]	=			
[EMT05]	-		+	
[CSP12]	=	+		
[CCG+06]	=		=	
[PC11]	+	=	=	-
[MWPM14]	=	=		-
[NMBW08]	+		+	
[SA07]	=	-	+	
[MS07]			+	
[Lat14]			+	+

Taulukko 1: Tulosten yhteenveto. Mielipide-sarakkeella tarkoitetaan kehittäjien näkemystä TDD:stä

Teollisessa kontekstissa positiivinen vaikutus laatuun oli selvästi suurempi kuin akatemisessa, mutta toisaalta ammattilaisilla myös tehokkuus oli matalampi.

4.10 Johtopäätökset

Tutkimuksia-osiossa käsitellyt tutkimukset eivät anna testivetoisesta kehityksen hyödyistä selkeää kuvaa. Tulokset ovat useimmiten neutraaleja tai lievästi positiivisia. Taulukkoon 1 on koottu tässä tutkielmassa käsiteltyjä tutkimuksia. Positiiviset tulokset on merkitty plussalla, negatiiviset miinuksella, neutraalit yhtäsuuruusmerkillä ja tyhjällä ne tutkimukset joissa aiheeseen ei otettu kantaa. Luokittelut perustuvat tutkimusten tekijöiden näkemykseen, ja niissä on hieman tulkinnan varaa. Taulukkoon on valikoitu ne laadun osa-alueet joihin tutkimukset ottivat kantaa, tälläisiksi luokitteltiin luotettavuus, ylläpidettävyys ja taloudellisuus. Lisäksi käyttäjien mielipide menetelmästä on kiinnostava "laatunäkökulma".

Parempi luotettavuus on esiintynyt TDD:tä käsittelevissä artikkeleissa sen tärkeimpänä ehdotettuna hyötynä. Luotettavuus ei kuitenkaan ole erityisen voimakkaasti tullut esille näissä tutkimuksissa, vaan neutraaleja tuloksia on enemmän kuin positiivisia. Käsittellyt tutkimukset olivat kestoltaan ja laajuudeltaan melko lyhyitä, joten menetelmä ei ehkä ollut niissä vahvimmillaan. Useissa tutkimuksissa onkin mainittu syynä heikoille tuloksille tutkimusten pieni koko.

Ylläpidettävyydessä useat tutkimukset kärsivät liian pienestä tutkimuksen koosta, molemmissa neutraaliksi merkityssä tutkimuksessa paremmasta laadusta oli merkkejä, mutta tulos ei tutkijoiden mukaan ollut tilastollisesti merkittävä johtuen pienestä otoksesta. Siniahon ja Abrahamssonin

tutkimuksessa ohjelman rakenteessa oli havaittu negatiivinen vaikutus.

Taloudellisuudessa tulokset ovat yllättävän positiivisia. On perusteltua päätellä paremman laadun ja pidemmän prosessin johtavan neutraaliin vaikutukseen, mutta käsitellyissä tutkimuksissa vaikutus onkin selkeästi positiivinen. Osassa tutkimuksista kirjoittajat havaitsivat ohjelman valmistuvan hitaammin, mutta arvioivat laatuvaikutuksen olevan suurempi. Rafiquen ja Misicin meta-analyysissä vaikutuksen taloudellisuuteen on kuitenkin havaittu olevan neutraali ja ammattilaisilla jopa negatiivinen. Tutkielmaan valitut tutkimukset olivat kuitenkin melko lyhyitä, joten ehkä TDD tuottaa paremman taloudellisuuden varsinkin pienissä ohjelmissa.

Mielipide-sarake on lisätty taulukkoon, koska kehittäjien näkemys menetelmästä saattaa vaikuttaa tuloksiin ja ohjelman laatuun. Vanhat toimintatavat saattavat tuntua paremmilta, koska ohjelmointi "takaperin"pakottaa ohjelmoijan toimimaan uudella tavalla. Jos menetelmä vaikuttaa vastenmieliseltä, vaikuttanee se myös vaikuttaa työnjälkeen negatiivisesti. Otos on kuitenkin sen verran pieni, ettei näistä tuloksista voi vetää johtopäätöksiä.

5 Yhteenveto

Artikkelissa tutkittiin TDD:n vaikutusta laatuun ja tehokkuuteen. Laatu määriteltiin käyttämällä apuna ISO 9126 standardin laadun osa-alueita. Tehokkuuden arviointiin käytettiin tarkasteltujen tutkimusten käsitystä tehokkuudesta, eli useimmiten tuotetun koodin määrää verrattuna aikaan. Tutkimusten tulokset aseteltiin parhaiten soveltuviin kategorioihin ja niitä verrattiin meta-analyysien tuloksiin. Useimmin käsitellyiksi ISO-standardin kategorioiksi nousivat luotettavuus ja ylläpidettävyys.

Luotettavuuden havaittiin paranevan hieman TDD:tä käytettäessä. Tehokkuudessa käsitellyt tutkimukset osoittivat parannusta, mutta tulos eroaa meta-analyyseistä. Ylläpidettävyydessä tulos oli neutraali.

Lähteet

[BBC+04] Botella, P, Burgués, X, Carvallo, JP, Franch, X, Grau, G, Marco, J ja Quer, C: *ISO/IEC 9126 in practice: what do we need to know?* Teoksessa *Proceedings of the 1st Software Measurement European Forum*, 2004.

[Bec03] Beck, Kent: *Test-driven development: by example.* Addison-Wesley Professional, 2003.

[CCG+06] Canfora, Gerardo, Cimitile, Aniello, Garcia, Felix, Piattini, Mario ja Visaggio, Corrado Aaron: *Evaluating Advantages of Test Driven Development: A Controlled Experiment with Professionals.* Teoksessa *Proceedings of the 2006 ACM/IEEE*

International Symposium on Empirical Software Engineering, ISESE '06, sivut 364–371, New York, NY, USA, 2006. ACM, ISBN 1-59593-218-6. http://doi.acm.org/10.1145/1159733.1159788.

[CK94] Chidamber, S.R. ja Kemerer, C.F.: *A metrics suite for object oriented design.* IEEE Transactions on Software Engineering, 20(6):476–493, Jun 1994, ISSN 0098-5589.

[Cri06] Crispin, L.: *Driving Software Quality: How Test-Driven Development Impacts Software Quality.* Software, IEEE, 23(6):70–71, Nov 2006, ISSN 0740-7459.

[CSG07] Côté, Marc Alexis, Suryn, Witold ja Georgiadou, Elli: *In search for a widely applicable and accepted software quality model for software quality engineering.* Software Quality Journal, 15(4):401–416, 2007, ISSN 0963-9314. http://dx.doi.org/10.1007/s11219-007-9029-0.

[CSP12] Causevic, A., Sundmark, D. ja Punnekkat, S.: *Test case quality in test driven development: A study design and a pilot experiment.* Teoksessa *16th International Conference on Evaluation Assessment in Software Engineering (EASE 2012)*, sivut 223–227, May 2012.

[EHP+06] Eldh, Sigrid, Hansson, Hans, Punnekkat, Sasikumar, Pettersson, Anders ja Sundmark, Daniel: *A framework for comparing efficiency, effectiveness and applicability of software testing techniques.* Teoksessa *Testing: Academic and Industrial Conference-Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, sivut 159–170. IEEE, 2006.

[EMT05] Erdogmus, H., Morisio, Maurizio ja Torchiano, Marco: *On the effectiveness of the test-first approach to programming.* IEEE Transactions on Software Engineering, 31(3):226–237, March 2005, ISSN 0098-5589.

[FHH00] Frøkjær, Erik, Hertzum, Morten ja Hornbæk, Kasper: *Measuring usability: are effectiveness, efficiency, and satisfaction really correlated?* Teoksessa *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, sivut 345–352. ACM, 2000.

[FTO14] Fucci, Davide, Turhan, Burak ja Oivo, Markku: *Impact of Process Conformance on the Effects of Test-driven Development.* Teoksessa *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Mea-*

surement, ESEM '14, sivut 10:1–10:10, New York, NY, USA, 2014. ACM, ISBN 978-1-4503-2774-9. http://doi.acm.org/10.1145/2652524.2652526.

[GFS05] Gyimothy, T., Ferenc, R. ja Siket, I.: *Empirical validation of object-oriented metrics on open source software for fault prediction.* IEEE Transactions on Software Engineering, 31(10):897–910, Oct 2005, ISSN 0098-5589.

[GW04] George, Boby ja Williams, Laurie: *A structured experiment of test-driven development.* Information and Software Technology, 46(5):337 – 342, 2004, ISSN 0950-5849. http://www.sciencedirect.com/science/article/pii/S0950584903002040, Special Issue on Software Engineering, Applications, Practices and Tools from the {ACM} Symposium on Applied Computing 2003.

[HHKM10] Hellmann, T.D., Hosseini-Khayat, A. ja Maurer, F.: *Supporting Test-Driven Development of Graphical User Interfaces Using Agile Interaction Design.* Teoksessa *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, sivut 444–447, April 2010.

[ISO01] ISO/IEC: *ISO/IEC 9126. Software engineering – Product quality.* ISO/IEC, 2001.

[ISO11] ISO, ISO: *IEC 25010: 2011: Systems and software engineering–Systems and software Quality Requirements and Evaluation (SQuaRE)–System and software quality models.* International Organization for Standardization, 2011.

[JS05] Janzen, David ja Saiedian, Hossein: *Test-driven development: Concepts, taxonomy, and future direction.* Computer, (9):43–50, 2005.

[KP96] Kitchenham, B. ja Pfleeger, S.L.: *Software quality: the elusive target [special issues section].* IEEE Software, 13(1):12–21, Jan 1996, ISSN 0740-7459.

[Lat14] Latorre, R.: *Effects of Developer Experience on Learning and Applying Unit Test-Driven Development.* Software Engineering, IEEE Transactions on, 40(4):381–395, April 2014, ISSN 0098-5589.

[MBS02] Michael, James Bret, Bossuyt, Bernard J ja Snyder, Byron B: *Metrics for measuring the effectiveness of software-testing tools.* Teoksessa *13th International Symposium on Software Reliability*

Engineering, 2002. ISSRE 2003. Proceedings., sivut 117–128. IEEE, 2002.

[McC76] McCabe, T.J.: *A Complexity Measure.* IEEE Transactions on Software Engineering, SE-2(4):308–320, Dec 1976, ISSN 0098-5589.

[MMR14] Miguel, Jose P, Mauricio, David ja Rodríguez, Glen: *A Review of Software Quality Models for the Evaluation of Software Products.* arXiv preprint arXiv:1412.2977, 2014.

[MS07] Madeyski, Lech ja Szała, Łukasz: *The Impact of Test-Driven Development on Software Development Productivity — An Empirical Study.* Teoksessa Abrahamsson, Pekka, Baddoo, Nathan, Margaria, Tiziana ja Messnarz, Richard (toimittajat): *Software Process Improvement*, nide 4764 sarjassa *Lecture Notes in Computer Science*, sivut 200–211. Springer Berlin Heidelberg, 2007, ISBN 978-3-540-74765-9. http://dx.doi.org/10.1007/978-3-540-75381-0_18.

[Muc08] Muccini, Henry: *Software testing: Testing new software paradigms and new artifacts.* Wiley Encyclopedia of Computer Science and Engineering, 2008.

[MWPM14] Munir, Hussan, Wnuk, Krzysztof, Petersen, Kai ja Moayyed, Misagh: *An Experimental Evaluation of Test Driven Development vs. Test-last Development with Industry Professionals.* Teoksessa *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, sivut 50:1–50:10, New York, NY, USA, 2014. ACM, ISBN 978-1-4503-2476-2. http://doi.acm.org/10.1145/2601248.2601267.

[NMBW08] Nagappan, Nachiappan, Maximilien, E.Michael, Bhat, Thirumalesh ja Williams, Laurie: *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. http://dx.doi.org/10.1007/s10664-008-9062-z.

[PC11] Pančur, Matjaž ja Ciglarič, Mojca: *Impact of test-driven development on productivity, code and tests: A controlled experiment.* Information and Software Technology, 53(6):557 – 573, 2011, ISSN 0950-5849. http://www.sciencedirect.com/science/article/pii/S0950584911000346, Special Section: Best papers from the {APSEC} Best papers from the {APSEC}.

[Quo] *Why does Kent Beck refer to the rediscovery of test driven development*. `http://goo.gl/ZYnMMD`. Accessed: 2015-05-15.

[RC+69] Randell, Brian, Committee, NATO Science *et al.*: *Software Engineering Techniques: Report on a Conference Sponsored*. Scientific Affairs Division, NATO, 1969.

[RH96] Rothermel, Gregg ja Harrold, Mary Jean: *Analyzing regression test selection techniques*. IEEE Transactions on Software Engineering, 22(8):529–551, 1996.

[RM13] Rafique, Y. ja Misic, V.B.: *The Effects of Test-Driven Development on External Quality and Productivity: A Meta-Analysis*. IEEE Transactions on Software Engineering, 39(6):835–856, June 2013, ISSN 0098-5589.

[SA07] Siniaalto, M. ja Abrahamsson, P.: *A Comparative Case Study on the Impact of Test-Driven Development on Program Design and Test Coverage*. Teoksessa *First International Symposium on Empirical Software Engineering and Measurement, 2007. ESEM 2007.*, sivut 275–284, Sept 2007.

[SMGM13] Stoica, Marian, Mircea, Marinela ja Ghilic-Micu, Bogdan: *Software Development: Agile vs. Traditional*. Informatica Economica, 17(4):64–76, 2013.

[TLD+10] Turhan, Burak, Layman, Lucas, Diep, Madeline, Erdogmus, Hakan ja Shull, Forrest: *How effective is test-driven development*. Making Software: What Really Works, and Why We Believe It, sivut 207–217, 2010.

[Whi00] Whittaker, J.A.: *What is software testing? And why is it so hard?* IEEE Software, 17(1):70–79, Jan 2000, ISSN 0740-7459.

[WNM12] Wilkerson, J.W., Nunamaker, J.F., Jr. ja Mercer, R.: *Comparing the Defect Reduction Benefits of Code Inspection and Test-Driven Development*. IEEE Transactions on Software Engineering, 38(3):547–560, May 2012, ISSN 0098-5589.