Test Driven Developement-menetelmän tehokkuus ja laatu

Petri Pihlajaniemi

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 29. maaliskuuta 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 Developement-menetelmän tehokkuus ja laatu

Oppiaine — Läroämne — Subject
Tietojenkäsittelytiede

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Kandidaatintutkielma	29. maaliskuuta 2015	12

Tiivistelmä — Referat — Abstract

Tiivistelmä.

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. Testauksen avulla ei voida todistaa ohjelman olevan virheetön, mutta sillä voidaan todistaa virheiden olemassaolo. 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. Vuonna 1970 Winston W. Roycen määrittelemä *vesiputousmalli* (Waterfall Model) kuvaa ohjelmistokehityksen viisivaiheisena prosessessina: 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 mukaan testaus on helppoa inkrementaalisissa menetelmissäi, kun taas vesiputousmallissa testauksessa havaittujen virheiden korjaaminen voi olla hankaa. Ketterät (Agile Model) ohjelmistokehityksen menetelmät perustuvat inkrementaaliseen malliin. [SMGM13]. Yksi erityisesti testaamiseen keskittyvät toimintapa on ketteriin malleihin perustuva *Test Driven Development*. [Cri06]

Aion tutkielmassani tarkastella TDD:n toimivuutta kahdella mittarilla: laadulla ja tehokkuudella. Nämä mittari ovat erittäin epämääräisiä, joten ensin on tutkittava mitä ne oikeastaan tarkoittavat.

2 Laadun käsitteestä

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

Transendentti näkökulma (Transcendental View) pitää laatua mahdotto-

mana määrittää tarkasti, mutta kuitenkin havaittavana ominasuutena [KP96]. Tämän näkökulman perusteella laatua on vaikea mitata, vaan se on lähinnä filosofinen käsite jota voidaan arvioda 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 noudattemista käytetään usein myyntivalttina, joten teollisesta näkökulmasta on tullut merkittävin perpektiivi 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]. Näkökulma on mielestäni järkevin valinta TDD:n tutkimiseen.

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 ISO/IEC9126 ohjelmistokehitystä käsittelevän laatumallin (quality model). [ISO01] ISO/IEC9126 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]

ISO9216 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. Standarin 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 koodi viittaa ohjelman toimintaan sitä ajettaessa, esimerkiksi virheiden määrään. ISO9216 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. Testivetoinen kehitys vaikuttaa myös ohjelman rakenteeseen, joten uskoisin menetelmän vaikuttavan positiivisesti tähän tekijään.

Luotettavuus (Reliability): kuinka ohjelmisto säilyttää suoritustason erilaisissa tilanteissa. Kypsyys, eli virheistä johtuvien häiriöiden välttäminen, virheiden sietäminen, virheistä palautuminen sekä luotettavuuteen liittyvien standardien noudattaminen. Yksi TDD:n tärkeimmistä esitetyistä ominaisuuksista on testien laajuus ja kattavuus, joten uskon tekniikan tuomien hyötyjen vaikuttavan erityisesti tähän kategoriaan.

Käytettävyys (Usability): ohjelman helppokäyttöisyys ja miellyttävyys. 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 luultavasti 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. Mielestäni testivetoisen kehi-

tyksen menetelmä ei vaikuta tähän ominaisuuteen käytännössä mitenkään.

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 esittelivät vuonna 1994 olio-ohjelmoinnin arviointiin kehittelemänsä kuusi mittaria.[CK94]. 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 on Method). 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

Aion tarkastella tehokkuutta 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. Aionkin erotella nämä kaksi merkitystä tehokkuudeksi ja taloudellisuudeksi. Tehokkuudella tarkoitan esimerkiksi testien kattavuutta ja taloudellisuudella testien kattavuutta verrattuna aikaan ja vaivaan. Suurin osa testauksesta tehdyistä tutkimuksista määrittelee termit samalla tavalla kuin edellä. [JMVS06]

Rothermel ja Harrold määrittelevät taloudellisuuden regressiotestaustekniikoita käsittelevässä artikkelissaan [RH96] laskentatehon kulutukseksi. Tälläinen määritelmä on testivetoista kehitystä käsittelevässä tutkielmassa mielestäni melko merkityksetön: jos testien ajaminen ylipäätään onnistuu on sen nopeudella vähäinen merkitys. 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 ohjelmointestaustekniikka 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 määrittelevät tehokkuuden käyttäjän ohjelmalla saavuttamien tavoitteiden tarkkuudeksi ja täydellisyydeksi [FHH00], 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 välillä tarkoittavan eria asioita eri artikkeleissa. Lukemissani 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]. 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:

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].

4 Tutkimuksia

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].

Testivetoinen kehitys on siis tutkimusten määrästä päätellen kiinnostanut myös akateemisia tahoja.

4.1 Laatu

Oman päättelyni mukaan testivetoinen kehitys vaikuttaa positiivisesti ohjelmiston laatuun. Testien kirjoittaminen ennen ohjelmakoodia vaikuttaa varmistavan testikattavuuden lisäksi myös tarkemman rakenteen suunnittelun. Aiemmin artikkelissa esitellyistä laatutekijöistä testikattavuus ja parempi rakenne vaikuttavat mielestäni erityisesti ylläpidettävyyteen ja luotettavuuteen. Aion tarkastella TDD:stä kirjoitettujen artikkelien tuloksia ISO 9126:n määrittelemien sisäisen ja ulkoisen laadun kategorioiden perusteella, vaikka erilaisia tuloksia onkin vaikea lokeroida luokkien sisälle.

4.1.1 Toimivuus

4.1.2 Luotettavuus

Luotettavuus vaikuttaa olevan suosituin laadun osa testivetoista kehitystä käsittelevissä artikkeleissa. Useat testit ovat esimerkiksi mitanneet TDD:llä tuotetun ohjelmiston laatua testien läpäisyn perusteella, jota pidän lähinnä luotettavuuteen liittyvänä mittarina.

Georgen ja Williamsin [GW04] tutkimuksessa havaittiin TDD:tä käyttäneiden parien tuottamien ohjelmien läpäisevän 18% enemmän tutkijoiden

rakentamia testitapauksia kuin kontrolliryhmien. Tutkimuksessa oli kuitenkin ongelmia joiden takia pidän tulosta 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 Code Inspection-menetelmään, jossa erillinen ryhmä katselmoi koodia. Tutkijoiden mukaan menetelmillä on selvä ero, katselmointi poistaa ohjelmasta löytyviä virheitä, mutta TDD käsittelee ne joi 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. Tulos on mielestäni yllättävä, Wilkersonin ja kumppanien mukaan se saattaa selittyä TDD-menetelmän liian epämääräisellä määrittelyllä.

4.1.3 Käytettävyys

Testivetoinen kehitys tuskin vaikuttaa loppukäyttäjän kokemukseen ohjelman laadusta. TDD ei varsinaisesti ota kantaa käytettävyyteen, eikä menetelmä sovellu graafisten käyttöliittymien suunnitteluun. Hellman ja kumppanit [HHKM10] pitävät GUI:n kehitystä TDD:llä niin vaikeana, että ovat artikkelissaan pyrkineet kehittämään TDD:stä paremmin tehtävään sopivaa muunnosta.

En ole löytänyt tutkimuksia testivetoisen kehityksen vaikutuksesta käytettävyyteen, joten vertailenkin tässä osiossa itse menetelmän käytettävyyttä. TDD:n prosessi on yksinkertainen, joten uskon aloittelevankin ohjelmoijan ymmärtävän sen toimintatavat nopeasti.

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

4.1.4 Tehokkuus

4.1.5 Ylläpidettävyys

Testivetoisen kehityksen vaikutus ohjelman rakenteeseen vaikuttaa ehkä myös sen ylläpidettävyyteen. Ohjelman rakenteen mittaaminen tehokkasti vaikuttaa tutkimusten perusteella vaikeammalta kuin esimerkiksi luotettavuuden mittaaminen; virheitä on helpompi laskea kuin arvioida rakenteeseen liittyviä ominaisuuksia.

Siniaalto ja Abrahamsson [SA07] tutkivat TDD:n vaikutusta ohjelmiston laatuun. Tutkijat käyttivät mittareina tässäkin artikkelissa esiteltyjä 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. Tulos poikkeaa suuresti muista näkemistäni testivetoista kehitystä käsittelevistä artikkelista, joissa TDD:n yhdeksi tärkeimmäksi vahvuudeksi on nostettu juuri parempi rakenne. Epäilen Siniaallon ja Abrahamssonin tuloksen olevan poikkeustapaus.

4.1.6 Siirrettävyys

En ole löytänyt artikkeleja joissa oltaisiin tutkittu TDD:n vaikutusta ohjelmiston siirrettävyyteen. Testivetoisen kehityksen prosessi ei mielestäni millään tavalla sivua siirettävyyteen 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 melestäni käyttää lähes missä tahansa ohjelmontitilanteessa. Graafisten käyttöliittymien suunnitteluun TDD:tä pidetään huonosti sopivana [HHKM10].

4.2 Taloudellisuus

Vaikka koodin laatu olisi parempaa, omien kokemuksieni perusteella varsinainen ohjelman koodaus TDD:tä käyttäessä on kuitenkin erittäin hidasta. Uskonkin testivetoisen kehityksen hyötyjen olevan pienempiä kuin sen käyttämisessä kuluva aika, ja epäilen menetelmän taloudellista tehokkuutta.

George ja Williams [GW04] havaitsivat TDD:tä käyttäneiden pariohjelmoijien käyttäneen 16% enemmän aikaa tehtävän suorittamiseen kuin tavanomaista vesiputousmalli noudattanut kontrolliryhmä. Toisaalta suurin osa kontrolliryhmistä oli jättänyt testitapaukset tekemättä, joten mielestäni tulosta ei voi pitää vertailukelpoisena.

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

4.3 Käyttäjien mielipiteitä menetelmästä

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

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

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

4.4 Meta-analyysit

5 Käyttö yritysmaailmassa

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.

[CK94] Chidamber, S.R. ja Kemerer, C.F.: *A metrics suite for object oriented design.* Software Engineering, IEEE Transactions on, 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`.

[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.

[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.

[GFS05] Gyimothy, T., Ferenc, R. ja Siket, I.: *Empirical validation of object-oriented metrics on open source software for fault prediction.* Software Engineering, IEEE Transactions on, 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.

[JMVS06] Juristo, N., Moreno, A.M., Vegas, S. ja Solari, M.: *In Search of What We Experimentally Know about Unit Testing.* Software, IEEE, 23(6):72–80, Nov 2006, ISSN 0740-7459.

[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].* Software, IEEE, 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 *Software Reliability Engineering, 2002. ISSRE 2003. Proceedings. 13th International Symposium on*, sivut 117–128. IEEE, 2002.

[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.

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

[RH96] Rothermel, Gregg ja Harrold, Mary Jean: *Analyzing regression test selection techniques.* Software Engineering, IEEE Transactions on, 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.* Software Engineering, IEEE Transactions on, 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 *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, 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?* Software, IEEE, 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.* Software Engineering, IEEE Transactions on, 38(3):547–560, May 2012, ISSN 0098-5589.