# Test Driven Developement-menetelmän tehokkuus ja laatu

Petri Pihlajaniemi

Kandidaatintutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Helsinki, 15. 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 | 15. maaliskuuta 2015 | 7 |

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 laadullisia ominaisuuksi, 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.

Luotettavuus (Reliability): kuinka ohjelmisto säilyttää suoritustason erilaisissa tilanteissa. Kypsyys, eli virheistä johtuvien häiriöiden välttäminen, virheiden sietäminen, virheistä palautuminen sekä luotettavuuteen liittyvien standardien noudattaminen.

Käytettävyys (Usability): ohjelman helppokäyttöisyys ja miellyttävyys. Ymmärrettävyys, ohjelman opittavuus, käytettävyys, viehättävyys ja käytettävyysstandardien noudattaminen.

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.

Ylläpidettävyys (Maintainability): mahdollisuudet muuttaa ohjelmaa. Alaluokkina analysoitavuus, muutettavuus, stabiilisuus, testattavuus ja ylläpidettävyysstandardien noudatus.

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.

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]

- **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]

**Lisää muiden näkemyksiä, guimothyssa listastattu paljon erimielisyyksiä! [GFS05]**

### 2.3 Tehokkuuden mittareita

Tarkastelen tehokkuutta ohjelmistotestauksen näkökulmasta. Valtaosa tarkastelemistani tutkimuksista on käsittänyt tehokkuuden käytetyn vaivan ja ajan suhteena testauksen kattavuuteen ja tarkkuuteen.

## 3 Test Driven Development

Test Driven Development on Kent Beckin "löytämä"ohjelmistokehitystekniikka [Bec03]. 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 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.

### 3.1 Tutkimuksia

babab

### 3.2 Tuloksia

babab

#### 3.2.1 Laatu

babab

#### 3.2.2 Tehokkuus

babab

### 3.3 Meta-analyysit

babab

## 4 Käyttö yritysmaailmassa

babab

## 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`.

[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.

[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.

[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.

[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.

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

[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.